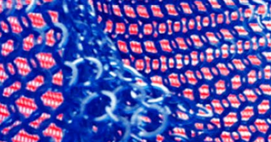


Research Article

Distributed SCFMBF Based Protocol for Integrity In Cloud Storage System (DPICS)

Hananeh Sasaniyan Asl[1] · Behzad Mozaffari Tazehkand[1] · Mir Javad Museviniya[1]

Received: 11 September 2019 / Accepted: 3 January 2020 / Published online: 14 January 2020
© Springer Nature Switzerland AG 2020

Abstract
Cloud storage allowed the data owners to store their data without the burden of local hardware. There were some security concerns because of the cloud being untrusted. To make sure the stored data was intact since it was impractical to download the full data, a security mechanism named integrity check was used. Integrity check was achieved by probabilistic structures such as bloom filter and cuckoo filter. Uploading data file on to one cloud server could be a point of failure, instead the file could be distributed over l servers. Distributed SCFMBF Based Protocol for Integrity in Cloud Storage System (DPICS) was proposed to remotely check data integrity in a more effective way. Smart Cuckoo Filter Modified by Bloom Filter (SCFMBF) was a protocol that improved some of the performance properties of cuckoo filter such as false positive rate and insertion performance and solved some drawbacks of the cuckoo algorithm such as endless loop. Main characteristic of the bloom filter was used to improve cuckoo filter as the protocol name describes. First bloom filter was used as the integrity check method in cloud but bloom filter was not suitable for dynamic environment, SCFMBF was substituted for bloom filter because it had low false positive probability in integrity check and could cope well with dynamic data. How DPICS was more efficient than other previous works, was shown in terms of false positive probability, integrity check and retrievability.

Keywords Bloom filter · Cloud · Integrity · Hash Function · Distribution · Channel Coding · Cuckoo filter

1 Introduction

Cloud Storage is one of the first and the most demanding field in current digital area. Cloud computing allows data storage at a remote place. Because of service unreliability such as service outage, service corruption by Byzantine failures and/or malicious attacks, it is necessary to ensure of data outsourcing security. There are several security components in cloud computing such as physical, infrastructure, and software security. Security requirements for cloud storage vary with applications. Since storage is an important core in cloud, data security such as confidentiality, availability, and integrity are key concerns of any cloud computing systems. To achieve these components, there are some tools such as access control, authentication, hash functions, digital signature, encryption, auditing.

There is a strong motivation by the cloud to hide data loss in order to keep its reputation, or remove the stored data file for economy reasons, and change or replace the stored data. To prevent that from happening, it is necessary to have data integration verification process. User cannot access the entire data for data integrity verification since it is typical to have a huge amount of archived data, so whole data file checking is impractical, therefore, traditional cryptographic primitives for data integrity in cloud cannot be adopted.

The data integrity verification mechanism in cloud proposed by researchers is based on different techniques

✉ Hananeh Sasaniyan Asl, sasaniyan@tabrizu.ac.ir; Behzad Mozaffari Tazehkand, mozaffary@tabrizu.ac.ir; Mir Javad Museviniya, niya@tabrizu.ac.ir | [1]Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran.

SN Applied Sciences (2020) 2:210 | https://doi.org/10.1007/s42452-020-1951-9

named remote data checking. Data integrity is a property that is used in both data communication and data storage systems. It is about checking data's correctness, its being unchangeable and being unmodified. In data communication and storage there are many techniques for integrity checking such as parity checking, CRC (Cyclic Redundancy Code), or cryptographic integrity methods such as MAC (Message Authentication Code). Furthermore, there are some probabilistic data structures used for testing if an item belongs to a set or not (set membership verification) such as bloom filter [1] (known as Standard bloom filter (STD) and cuckoo filter [2]. Nowadays advanced processors enable us to take out complicated calculations. With parallel implementation, bloom filter and cuckoo filter are more applicable. We decided to design a probabilistic filter based on Bloom filter.

In cloud storage service, data owner initially stores data and other information based on data (metadata such as digital signature) on cloud storage server, later, at any time data owner can challenge server to integrity check or retrievability, the server then generates a response based on data or information (metadata such as digital signature) that is passed to the data owner for integrity verification.

Since outsourced data integrity verification in cloud system without keeping the local copy of data files is a big challenge, the spot-checking based technique is proposed. Verifying the integrity of data file by spot checking uses only a fraction of data file via various cryptographic primitives. User randomly samples small portions of data file and checks the integrity. By spot checking, data owner can detect if a fraction of the stored data file on the cloud server has been corrupted.

Bloom filter is a space efficient probabilistic data structure which is used to represent a set and perform membership queries [3] to query whether an element is a member of a set or not. Bloom filter was first proposed by Bloom [1] 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 Sect. 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 [4].
- Routing: Locating the demanded content is one of the major challenges in information-centric networking. Servers advertise their content objects using bloom filters [5]. 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. False positives in this cause 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: Reducing the download latency of web documents is crucial for web users and web content providers. Caching is one of the primary mechanisms for lessening the latency as well as bandwidth requirements for delivering web content. 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 [6].

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, it will not store that key in that bucket, it will simply mark it 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. According 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 provide member deletion for bloom filter, member insertion for dynamic sets, find the similarity of two sets, reduce the size of bloom filter, make bloom filter scalable, reduce false positive probability, and update bloom filter. 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 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.

In this article, a smart cuckoo filter is proposed 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 inserting more items becomes impossible, we solve this with a change in cuckoo's structure by using bloom filter as the fingerprint of data and adding a new phase 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 when it cannot find an empty bucket for the new insertion and 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) has many extensions for different applications, for our protocol we found standard bloom filter to be adequate. Comparison results demonstrates our protocol's being effective in lowering the key design factor of these filters, FPP, to such a low negligible amount and even with changing the filter elements, it can fall to even lower amounts.

With the changes made to cuckoo filter, a new filter is made that is named Smart Cuckoo filter Modified by Bloom Filter (SCFMBF). Standard bloom filter and cuckoo filter characteristics are explained then they are related and compared to SCFMBF protocol.

In this paper bloom filter and SCFMBF are explored that act in a spot-checking way for integrity verification in cloud. Use of a single server in cloud is a point of failure. To overcome this matter, data is distributed among multiple servers. By data distribution on different servers, redundant bloom filters are made in a way that achieves less false positive rate. Using channel coding on multiple servers allows retrievability in general but integrity is focused in this paper. DPICS method uses a distributed probabilistic data structure on multiple servers to check data integrity.

In the following, the article outline is summarized. Bloom filter and cuckoo filter concepts are explored further in the Sect. 3, because they are the base of the proposed method. In Sect. 4, SCFMBF protocol is introduced and the designed algorithms are explained. Section 5 is dedicated to cloud storage systems and Sect. 6 shows how data is distributed among multiple servers and integrity check is done by bloom filter and SCFMBF. In Sect. 7 the calculation results and the figures of comparison are considered, the proposed protocol s compared with cuckoo filter to check its efficiency and SCFMBF is compared to bloom filter in cloud storage systems. Section 8 is a summarized conclusion about this work.

2 Related work

Rae et al. [7] introduces neural bloom filter, that train neural networks to replace data structures that have been crafted by hand for faster execution. In this setting, a neural data structure is instantiated by training a network over many epochs of its inputs until convergence. Neural bloom filter is able to achieve significant compression gains over classical bloom filter. Fan et al. [8] 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. Jiang et al. [9] employs an algorithm based on bloom filter in barcode recognition

and processing. The traditional way is to scan the barcodes and read the database to identify the barcodes, but in giant automatic sorting systems using databases is too slow for barcode recognition, so rapid identification of barcodes using bloom filters is a good option. Guo et al. [10] 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. [11] 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 1 s in the resultant bit array. Compressed bloom filter by Mitzenmacher [12] 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. [13] 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. [14] 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 and Matias [15] suggested spectral bloom filter to support frequency queries by counters to store the frequency values. Matsumoto et al. [16] proposed Adaptive bloom filter, a modification of counting bloom filters for the cases that large counters are needed. Almeida proposed scalable bloom filter [17] that adds slices of bloom filter as the set grows. Weighted bloom filter is designed by Bruck et al. [18]. 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. [19] 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. [2], 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. [2] 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 Sect. 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. Jeong et al. [27] used bloom filter for the internet of things providing secure cloud storage service. Since the data collected from the IoT are generated in large quantities, cloud computing is used to store and analyze the data. Their research shows using bloom filter saves time and has no significant difference with existing methods although it causes false positives. Najafimehr et al. [28] proposed Single-hash Lookup Cuckoo Filter (SLCF) that uses one hash function in the query phase to decrease the lookup time. In the programming phase of SLCF, it utilizes a sequence of hash functions without any relocation to find an empty bucket and storing the overflowed items. It finds an empty bucket and sets an index in the original bucket to the address of found bucket.

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 a greater number of counters instead to reduce false positive probability. TBF provides much lower false 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. [34] 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. [35]. 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. [36] 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. [37] 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 and Lemir [38] 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. [39] to solve these requirements. Recently, an overview of bloom filter and its variants with optimization methods are deliberated by Grandi [40] with performance and generalization review in more details.

Yan et al. [41] suggested a dynamic integrity verification scheme of cloud storage based on lattice and bloom filter. With development of quantum computers, it's necessary to design a signature scheme that can resist quantum attacks. In the cloud storage data calculation, using of lattice and bloom filter can not only resist the quantum attacks, but also improve the utilization of cloud storage space. Zhang et al. [42] employs cross encoding for recovery from errors and counting bloom filter for integrity check and update. This method uses a TPA for dynamic operation which leads to point of failure and can be also out of reach.

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.

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 $(0\le\alpha\le1)$
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 1 s in the finger print
r	Maximum number of OR operation
SBF	Server Bloom filter
RBF	Redundant Bloom filter
CS	Cloud server
S_i	i_{th} Server
(N,l)	Systematic channel coding scheme
$bn^{(i)}$	Block number
N	Total number of servers
L	Number of redundant servers
t	Number of data servers

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 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.

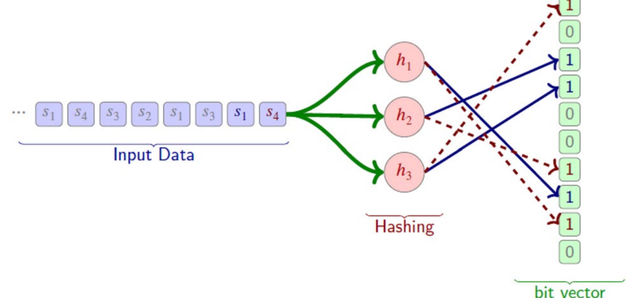


Fig. 1 Bloom filter example

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. 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

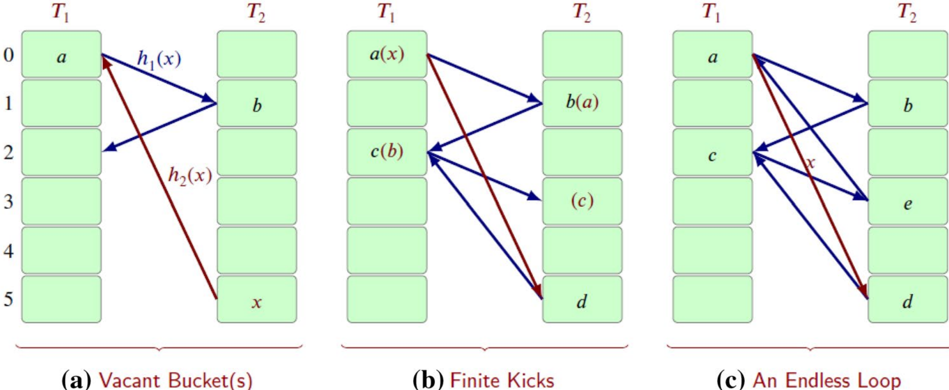


Fig. 2 The conventional cuckoo hashing data structure [22]

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. 2a. 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. 2b. Two candidate positions of Item x are 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. 2c. 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.

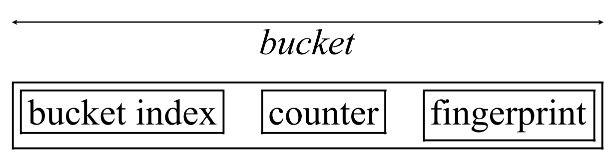


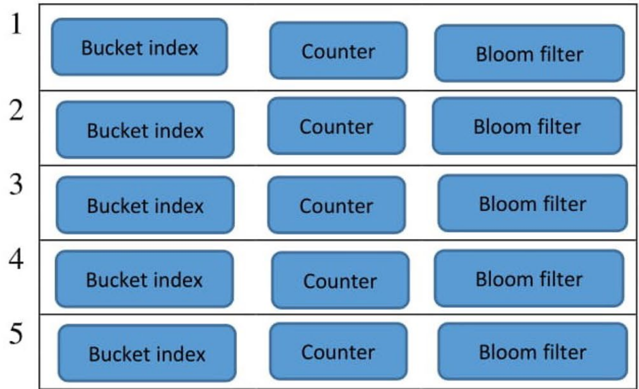
Fig. 3 bucket structure

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. Figure 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



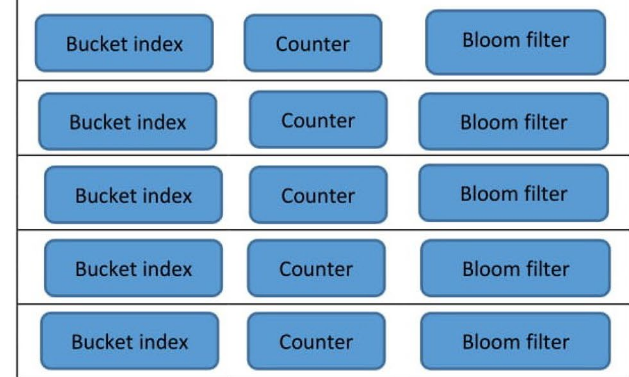


Fig. 4 structure of modified cuckoo filter

stored. In cuckoo filter if bucket i (called the primary) is full then the cuckoo filter attempts to store f in bucket $i \oplus h(f)$, ($\oplus$ 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 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: 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

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)

```
Want to insert fingerprint(x) to the table;
Goto i1's bucket and take the saved fingerprint;
Sum up fingerprint(x) and the already saved fingerprint by OR operation;
Put the OR result in the i1's bucket;
Increment the cuckoo counter by 1;
```

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.

Algorithm 3: Insertion algorithm

```
Insert data block xi;
Hash the block and calculate its fingerprint fi;
Derive two indices from the hash and fingerprint: i1 and i2;
if the table is not full then
  Check the derived indices;
  *: if empty then
      Insert the block's fingerprint into the corresponding bucket
     Reset j1;
  else
      set j1=1;
     Check current item's alternative indices;
     if j1 = 0 in the alternative buckets then
       Goto *
      else
          a bucket with j1=1 is reached, endless loop detected;
      delete fi;
      end
    end
else
      CSA: substitute B(i1) + f(xi) in the Bi Bucket;
      Increment j2;
End
```

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.

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 1 s 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 1 s 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.

Algorithm 4: Look up algorithm

```
f = fingerprint(x);
 i1 = hash(x);
 i2 = i1 ⊕hash(f);
check i1's cuckoo counter;
if j2 =0 then
        Goto cuckoo algorithm;
else
        Goto CSA algorithm;
end
Cuckoo algorithm: if bucket(i1) or bucket(i2) 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
```

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 1 s 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.

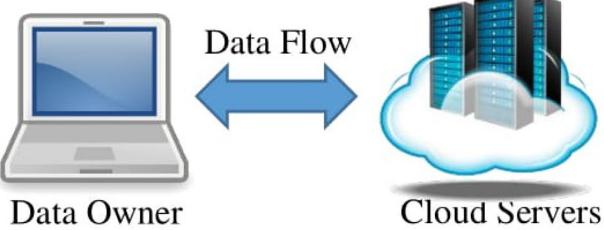
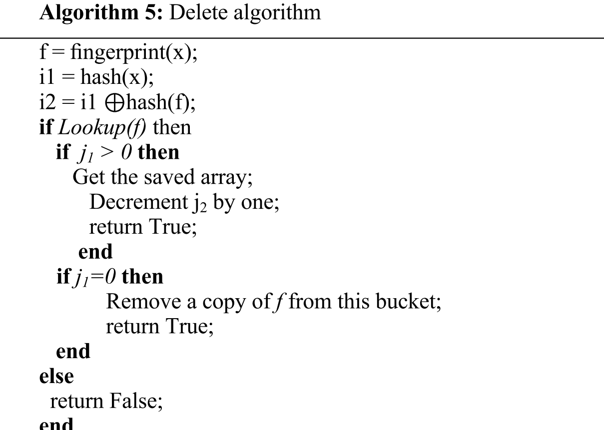


Fig. 5 Network architecture



Algorithm 5: Delete algorithm

```
f = fingerprint(x);
i1 = hash(x);
i2 = i1 ⊕hash(f);
if Lookup(f) then
   if j1 > 0 then
      Get the saved array;
         Decrement j2 by one;
         return True;
      end
   if j1=0 then
         Remove a copy of f from this bucket;
         return True;
   end
else
  return False;
end
```

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 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 Cloud storage systems

This process can be examined in cloud storage systems. First, we studied bloom filter's use in multi-server mode then we replaced bloom filter by SCFMBF protocol.

We define different network entities as:

- Data owner who has data to be stored in cloud and needs the cloud for data storage and computation.
- Cloud Server (CS): provides data storage service and has significant storage space and computation resources.

The network architecture for cloud storage service is represented in Fig. 5.

In [41], bloom filter is employed to enforce data integrity for outsourced data in cloud environments. Their method is block-oriented. They have studied different cases about encryption and the place of storage for data blocks and bloom filters. They only upload their file blocks to one server and makes a bloom filter for every file block, but we are going to upload files over *l* servers to prevent point of failure and provide simple retrievability. Since bloom filter doesn't have removal ability, we will replace it with SCFMBF that is going to be described later.

In these storage resources, the owner's data is stored on a group of servers. Data owner needs to check data integrity after it's saved on the cloud server without needing to access the full data or downloading it completely. The main goal of data storage on cloud is finally the retrievability of it. Since data is not only exposed to adversary attacks but also it can be under Byzantine attacks like the usual channel errors, we use channel coding techniques to survive such problems.

In this section we present the key fractions of our protocol. We will explain each fraction later in details. In our method the data owner must first organize the file that needs to be stored. This organization means making integrity check structures for data verification goal and retrievability, and changing the file in a way that it fulfills the goal and is suitable for the distribution in cloud. According to what we said, file distribution preparation, data integrity checking and data retrievability is the three main subjects of data storage in cloud. We focus on block-oriented methods in every section since in cloud's data no rapid change in a relative short period is expected. In our model we have a point to point communication channel between each cloud server and the user.

5.1 File distribution preparation

Channel coding is used to tolerate multiple failures in distributed storage systems. For distributing data file F on N servers, we use a systematic channel coding (N, l) (N = l + t) to disperse the data blocks on N servers. An (N, l) systematic channel coding, makes t parity vectors for l data vectors in a way that l data vectors can be retrieved from l + t vectors. The unmodified l data file vectors together with t parity vectors are distributed across l + t different servers. A simple representation of file distribution on a group of servers is displayed in Fig. 6.

We divide the file F into l blocks. $F = \left(F_1, F_2, \dots, F_l\right)$. Each F_i is divided into l sub blocks $F_i = \left(f_{1i}, f_{2i}, \dots, f_{il}\right)$. Block numbers are defined as $bn^{(1)}, \dots, bn^{(l)}$. f_i s can be distributed cross servers or in a server way. Cross server distribution means to put each block in a sequence on different servers. Server distribution means putting sequential blocks in a single server. By channel coding, parity blocks are constructed that will be stored on t servers. $S_{l+1}, S_{l+2}, \dots, S_{l+t}$. The systematic layout with parity vectors is achieved with the information dispersal matrix A. After a sequence of elementary row transformations, the desired matrix A can be written as A = (I|P) where I is a l × l identity matrix and P is the secret parity generation matrix with size l × (l + t). By multiplying F by A, the user obtains the encoded file G as G = F.A.

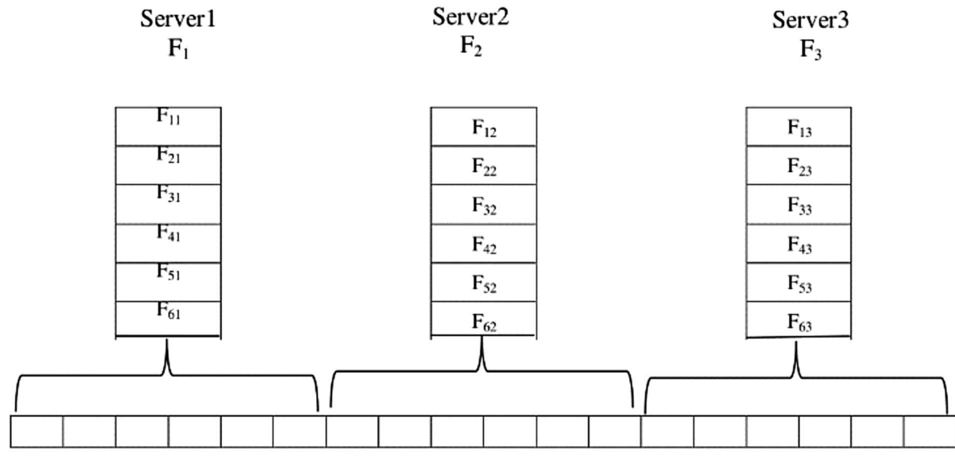


Fig. 6 File distribution

5.2 Data integration preparation

At first, we suggest our designed method Distributed Bloom Filter (DBF) for cloud storage integrity. According to 3.1 and 3.2, we choose our desirable bloom filter properties. Since our data is distributed on l servers, we calculate BF for each server naming it Server Bloom filter (SBF), $\{SBF_1, SBF_2, \ldots, SBF_l\}$ and we calculate bloom filter in a cross server way naming it Redundant Bloom filter (RBF), $\{RBF_1, RBF_2, \ldots, RBF_l\}$.

5.3 Data retrievability preparation

Suppose data integrity is checked for a sub block of our data, if the answer from the integrity check algorithm is TRUE, we find out the data is intact with a false positive probability, but if the answer is False, part of our data is damaged that is when data retrievability methods come in handy. The sub block checked for data integrity shows the location of error, it means we find out the malicious or under attack server then the corresponding blocks of the channel coding is requested to correct the error by the parity blocks. According to the bloom filter's parameters, a suitable channel coding is chosen and used.

6 Distributed Bloom filter (DBF) schema

We consider the simplest approach by using bloom filter for enforcing integrity of outsourced data in cloud [17]. That approach distributes the data file on a single server. They compared their result with the traditional security hash functions such as SHA-1 and MD5, which showed that bloom filter implementation is highly space efficient at the expense of additional computational overhead. This result was the inspiration for our approach but unlike them, we distributed the file across N servers. Our integrity check method first is the standard bloom filter and then improved to SCFMBF. bloom filter is a probabilistic data structure that tests whether an element is a member of a set with a false positive rate by the way.

6.1 System entities

We take benefit from standard bloom filter that was described previously. There are some entities can be identified as follows:

Data File (F): the owner's data

Server Bloom Filter (SBF): probabilistic data structure for integrity check that is applied on the data blocks over each server.

Redundant Bloom Filter (RBF): standard bloom filter that is applied intersectionally on the data over different servers.

Index Table (IT): stores the indexes of data blocks and bloom filters to show where to look for the required block.

6.2 Setup phase

The following steps describe setup phase.

- We divide file F into l blocks $F_1 \ldots F_l$ that they will be stored across l cloud servers. Each block consists of sub blocks: $F_i = (f_{1i}, f_{2i}, \ldots, f_{il})$ as shown in Fig. 7.
- We calculate SBF for each data server $SBF_1, SBF_2, \ldots, SBF_l$, shown in Fig. 8.
- We calculate RBF in a cross-server order for data servers: $RBF_1, RBF_2, \ldots, RBF_l$,shown in Fig. 8.
- We make IT according to the blocks and bloom filter indexes
- We save IT
- We apply channel coding to the data blocks.
- We upload the blocks on the corresponding servers according to the channel coding scheme (we distribute

Server 1	Server 2			Server l	Server (l+1)	Server (l+2)			Server (l+t)
F(1)	F(2)			F(l)	F(l+1)	F(l+2)			F(t)
f_{11}	f_{12}	....	....	f_{1l}	$f_{1(l+1)}$	$f_{1(l+2)}$	....	....	f_{1t}
f_{21}	f_{22}	....	....	f_{2l}	$f_{2(l+1)}$	$f_{2(l+2)}$	....	....	F_{2t}
⋮	⋮			⋮	⋮	⋮			⋮
f_{l1}	f_{l2}	....	....	f_{1l}	$f_{l(l+1)}$	$f_{l(l+2)}$	....	....	f_{lt}

Fig. 7 Setup phase 1

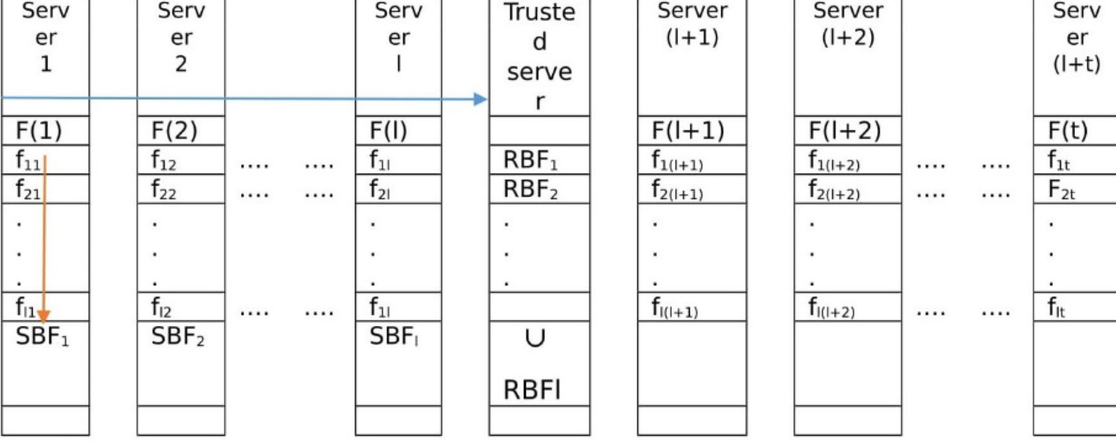


Fig. 8 Setup phase 2

data files on l servers and the parity blocks on t servers) and remove from the owner's device.

- We encrypt SBFs and RBFs. SBFs can be uploaded to their corresponding data server and RBFs can be put on whether a trusted server or parity servers or the main data servers.

This procedure is expressed in Algorithm 6.

Algorithm 6: DBF Setup Algorithm

- Divide file F into n blocks $F = (F_1, F_2, ..., F_l)$
- Divide each block into sub blocks $F_i = (f_{1i}, f_{2i}, ..., f_{il})$
- Calculate SBF for $f_{ij}s$ with the same j index
- Calculate RBF for $f_{ij}s$ with the same i index
- Make the index table
- Apply an (N,l) channel coding to $f_{ij}s$ with the same i index
- Put $f_{ij}s$ and SBF$_j$ with the same j index in the $j'th$ server
- Put the parity blocks and RBFs in $S_{l+1}...S_{l+t}$

6.3 Challenge-response phase

Due to bloom filter's set membership property we can test whether a block belongs to the data set or not that it means integrity check. In the following the procedure of challenge response is given, you also can check it in Algorithm 7:

Algorithm 7: DBF Challenge-response Algorithm

- Owner selects a block by its index from the index table F_{ij}.
- Owner sends the bloom filter parameters and the block index to the cloud
- Owner asks the cloud to find the selected block and calculate its bloom filter BF_{ij}
- Owner downloads the encrypted bloom filters and decrypts them.
- Owner compares BF_{ij} to corresponding SBF and RBF
- If BF_{ij} belongs to both of the bloom filters
    Return True
  Else
    Return False
  Goto DBF retrievability algorithm
End

- Owner selects a block by its index from the index table F_{ij} .
- Owner asks the cloud to find the selected block and calculate its bloom filter BF_{ij}.
- Owner downloads the encrypted bloom filters and decrypts them.
- Owner compares BF_{ij} to the corresponding SBF and RBF
- If BF_{ij} belongs to both of the bloom filters (SBF and RBF), integrity of data is proved with a reasonable false positive rate. (FPP of DBF).

6.4 Retrievability phase

If the challenge-response algorithm returns "False", Retrievability algorithm is performed as followed in Algorithm 8.

Algorithm 8: DBF Retrievability Algorithm

- Data owner asks the l cloud servers for $f_{ij}s$ with the same i index as the corrupted block and the corresponding parity blocks
- Data owner performs error correction
- Data owner sends the corrected block to the cloud and asks it to replace the corrupted block with the corrected block

7 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.

7.1 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. 9, 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.

7.2 False error probability of SCFMBF

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 $\frac{2}{m}$. Therefore, the probability of such q items sharing the same two buckets is [16]:

$$\left(\frac{2}{m} \times \frac{1}{f}\right)^{q-1} \tag{5}$$

The upper bound of the total probability of a false fingerprint hit is [17]:

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

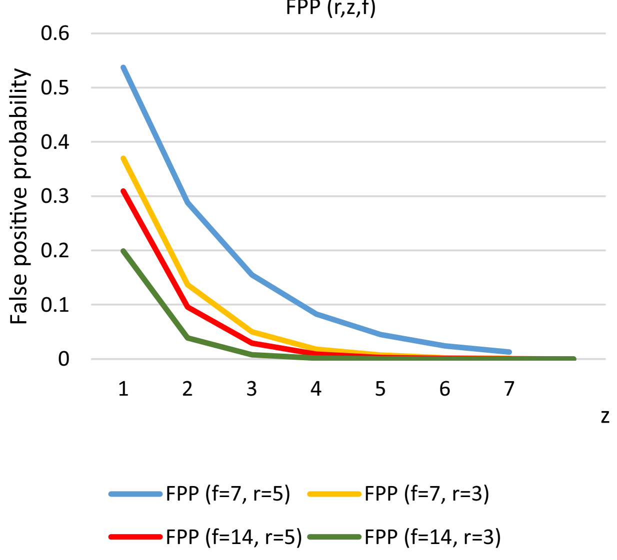


Fig. 9 CSA FPP

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 [17]:

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

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 Eq. 8 in Fig. 10.

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.

$$FPP_{SCFMBF} = FPP_{cuckoo} \times FPP_{CSA} \quad (8)$$

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

We see in Fig. 11, 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. 12. Figure 12 shows that our method surprisingly outdistances the performance of cuckoo filter and maintains a lower false positive probability in every case of comparison.

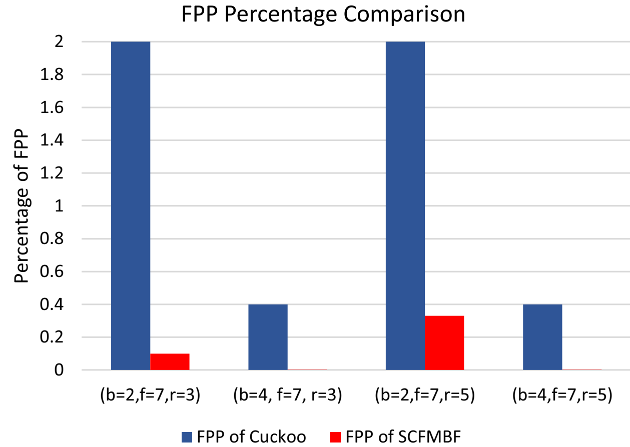


Fig. 11 FPP percentage comparison

7.3 False positive probability of distributed bloom filter

We have compared false positive probability (FPP) results of standard bloom filter and DBF schema. For optimal value of k, false positive probability of the bloom filter as we said before is:

$$FPP = (1 - e^{(-kn/mb)})^k \approx 1/2^k \quad (10)$$

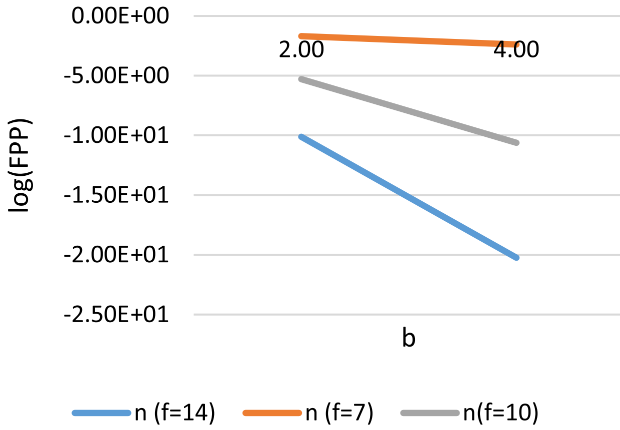


Fig. 10 Cuckoo FPP

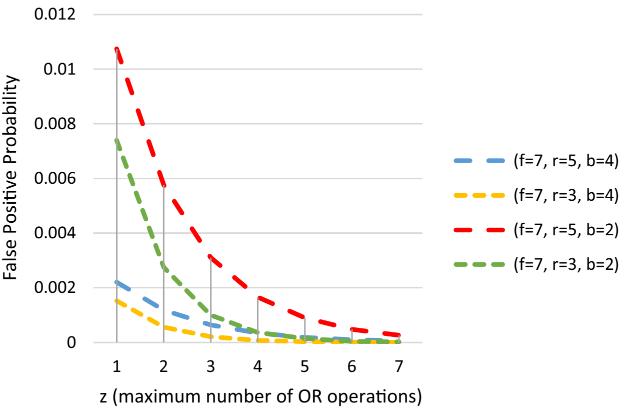


Fig. 12 SCFMBF FPP

For DBF schema seems false positive probability is dependent to SBF and RBF. FPP is random function with Bernoulli distribution that can be closely approximated by a Poisson distribution. An advantage of this approximation is that it is independent of the specific value of n. so we have two independent Poisson distribution that their intersection is calculated by multiplying the false positive rates:

$$FPP\left(RBF_j\right) \times FPP\left(SBF_j\right) = \left(\left(1 - e^{-\frac{kn}{mb}}\right)^k\right)^2 \approx \frac{1}{2^{2k}} \tag{11}$$

For a comparison between FPP of DBF method and FPP of BF, Eqs. 10 and 11 are plotted in Fig. 13 for different values of k. we see FPP of DBF is always smaller than DBF of simple bloom filter. Distributing data blocks on different servers causes a great improvement in false positive probability compared to single server mode, as it is demonstrated.

We improved the common use of bloom filter by distributing data and bloom filters among different servers and making redundant BFs.

7.4 DBF by SCFMBF

If the bloom filter structure of distributed bloom filter protocol is replaced by another set membership query structure, we can further improve DBF's efficiency. Candidate method is using SCFMBF in DBF protocol ndamed as DPICS and we will show how our protocol would be improved. SCFMBF solves the shortcomings of the cuckoo filter for making it usable for data storage integrity in cloud. It has got the notion from the standard bloom filter to make cuckoo filter better in insertion performance. Cuckoo filter has the problem of endless loops, SCFMBF is designed to make cuckoo filter smart to detect endless loops and get out of them, which leads to a higher cuckoo table capacity.

Instead of server-bloom filter and redundant-bloom filter we have server-SCFMBF and redundant-SCFMBF. We suppose that S-SCFMBF and R-SCFMBF are independent so the false positive probability of the DPICS algorithm is the multiplication of them.

$$FPP_{SCFMBF} = \binom{n}{2b+1}\left(\frac{2}{2^f.m}\right)^{2b} \times \left(1 - \left(1 - \frac{1}{f}\right)^r\right)^z \tag{12}$$

$$FPP_{DPICS-SCFMBF} = \left(\binom{n}{2b+1}\left(\frac{2}{2^f.m}\right)^{2b} \times \left(1 - \left(1 - \frac{1}{f}\right)^r\right)^z\right)^2 \tag{13}$$

FPP_{SCFMBF} is low enough in most of the cases but when we use the DPICS distribution for the SCFMBF algorithm, we will have far more improvement.

False positive probability of DPICS is calculated for different values of *r* and *b*, (*r* is the maximum number of OR operation and *b* is the bucket size) while changing *z* (*z* is the number of 1 s in the fingerprint) from 1 to 7, because a fingerprint probably has 1 to 7 number of 1 s in it. Then we got an average from the results. The calculations are shown in Figs. 14 and 15. Apparently as r increases, more bloom filters are piled up and FPP rises, but still it is in an acceptable range. Figure 16 shows the FPP results of DPICS and cuckoo filter in comparison. FPP of DPICS in all cases is much lower than cuckoo filter, therefore Our protocol has been successful.

8 Conclusion

In this paper it was tried to eliminate cuckoo filter's limitations. Cuckoo filter's endless loop problem was solved by the endless loop algorithm, there was no need for extra capacity because the protocol benefited from using small counters and a single bit for every bucket and by CSA algorithm, cuckoo filter was able to handle more insertions, the

Fig. 13 False positive probability

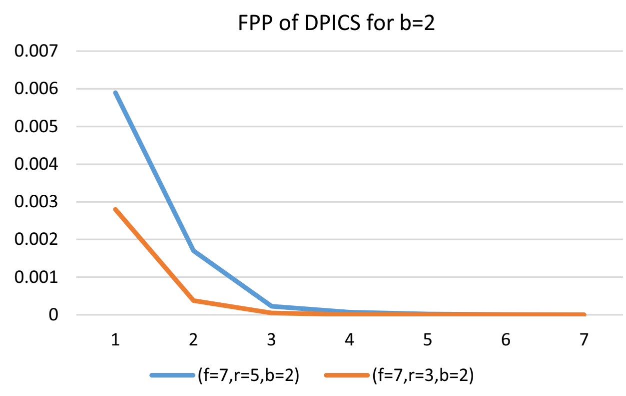


Fig. 14 FPP of DPICS for $b = 2$

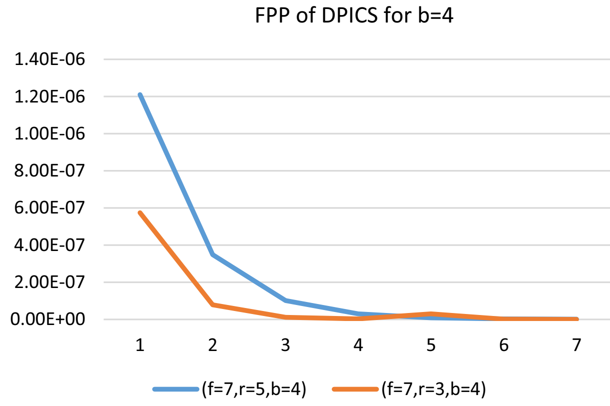


Fig. 15 FPP of DPICS for $b=4$

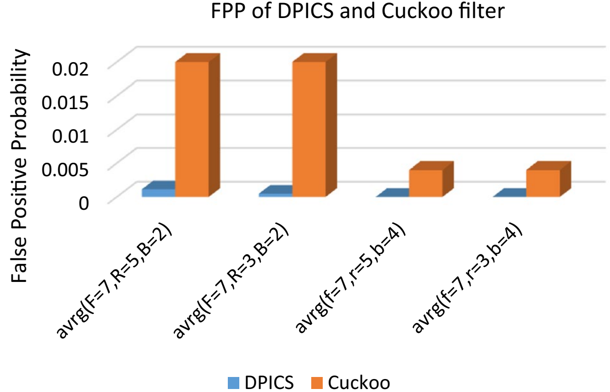


Fig. 16 FPP of DPICS versus cuckoo filter

idea of CSA was inspired from the bloom filter's basic logical operation (OR). The false positive probability according to the derived mathematical equation, was improved a lot. The relation between SCFMBF's parameters (same as cuckoo filter) and false positive probability was studied. CSA algorithm's false positive probability was plotted for different values of f (fingerprint length) and r (number of 1's in the fingerprint). As r decreased, FPP plot fell. As the fingerprint length increased, FPP decreased. 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 was plotted. For longer fingerprints, FPP had lower values, but long fingerprint length needed more storage space. Larger bucket size resulted in less FPP. Finally, SCFMBF was compared with cuckoo filter for f = 7, b = 2,4 and r = 3,5. In every case of study, there was a lot of improvement. FPP of SCFMBF in the worst case was four times less than the FPP of cuckoo filter so the capacity of cuckoo filter was improved to a great extent. The aim of the protocol was fulfilled.

For cloud storage systems, the common use of bloom filter was improved by distributing data and bloom filters among different servers and making redundant BFs. DBF protocol was an integrity check algorithm furthermore it could perform error correction, in other words, the single server mode of standard bloom filter was changed to a more realistic multi-server mode. In multi-server mode data was more immune to mobile adversaries comparing to the single server mode because just a portion of data was vulnerable to attacks at a time step. By Integrity check and the successive channel coding, small errors were detected and corrected regularly. DBF's false positive probability was improved a lot comparing to the standard bloom filter and calculation complexities were at a reasonable level just close to the standard bloom filter. It was also suggested to substitute SCFMBF protocol for bloom filter in DBF method and it was named DPICS. DPICS had all the advantages of DBF and even more. It was suitable for dynamic data and was more successful in lowering FPP which leaded to better integrity checking than other methods that used bloom filter.

Compliance with ethical standard

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426
2. Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD (2014) Cuckoo filter: practically better than bloom. In: Proceedings of the 10th ACM international on conference on emerging networking experiments and technologies, pp 75–88. ACM
3. Patgiri R, Nayak S, Borgohain SK (2019) Shed more light on bloom filter's variants. In: International conference on information and knowledge engineering IKE 18, pp 14–21
4. Mullin JK, Margoliash DJ (1990) A tale of three spelling checkers. Softw Pract Exp 20(6):625–630
5. Marandi A, Braun T, Salamatian K, Thomos N (2019) Pull-based bloom filter-based routing for information-centric networks. In: Consumer communications & networking conference (CCNC), pp 1–6
6. Bhomini PA, Jayasudha JS (2018) Enhanced bloom filter technique for latency reduction in mobile environment. Int J Appl Eng Res 13(17):13386–13391
7. Rae JW, Bartunov S, Lillicrap TP (2019) Meta-learning neural bloom filters, ICML 19
8. Fan L, Cao P, Almeida J, Broder AZ (1998) Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans Netw 28(4):254–265
9. Jiang M, Zhao C, Mo Z, Wen J (2018) "An improved algorithm based on Bloom Filter and its application in barcode recognition and processing. EURASIP J Image Video Process 2018(1):139

10. Guo D, Wu J, Chen H, Yuan Y, Luo X (2010) The dynamic Bloom filters. IEEE Trans Knowl Data Eng 22(1):120–133
11. Geravand S, Ahmadi M (2011) A novel adjustable matrix Bloom filter-based copy detection system for digital libraries. In: Proceedings of IEEE international conference on computer and information technology
12. Mitzenmacher M (2001) Compressed Bloom filters. In: Proceedings of the 20th annual ACM symposium on Principles of distributed computing, pp 144–150
13. Bonomi F, Mitzenmacher M, Panigrah R, Singh S, Varghese G (2006) Bloom filters via d-left hashing and dynamic bit reassignment. In: 44th Allerton conference
14. Shanmugasundaram K, Bronnimann H, Memon N (2004) Payload attribution via hierarchical Bloom filter. In: Proceedings of the 11th ACM conference on Computer and communicationsecurity (CCS 04), pp 31–41
15. Cohen S, Matias Y (2003) Spectral Bloom filters. In: SIGMOD'03: proceedings of the 2003 ACM SIGMOD international conference on management of data. New York, NY, USA: ACM, pp 241–252
16. Matsumoto Y, Hazeyama H, Kadobayashi Y (2008) Adaptive Bloom filter: a space-efficient counting algorithm for unpredictable network traffic. IEICE Trans Inf Syst E91-D(5):1292–1299
17. Almeida PS, Baquero C, Preguiça N, Hutchison D (2007) Scalable Bloom filters. Inf Process Lett 101(6):255–261
18. Bruck J, Gao J, Jiang A (2006) Weighted Bloom filter. In: 2006 IEEE international symposium on information theory (ISIT'06)
19. Bender MA, Colton MF, Johnson R, Kuszmaul BC, Medjedovic D, Montes P, Shetty P, Spillanne RP, Zadoc E (2012) Don't Thrash: How to Chache you Hash on Flash. In: Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems (HOTStorage11)
20. Grashofer J, Jacob F, Hartenstein H (2018) Towards application of Cuckoo filters in network security monitoring. In: 14th international conference on network and service management (CNSM)
21. Mitzenmacher M, Pontarelli S, Reviriego P (2017) Adaptive Cuckoo filters, arXiv preprint
22. Sun Y, Hua Y, Jiang S, Li Q, Cao S and Zue P (2017) SmartCuckoo: a fast and cost-efficient hashing index scheme for cloud storage systems. In: USENIX annual technical conference, pp 553–565
23. Kirsch A, Mitzenmacher A, Wieder U (2009) More robust hashing: Cuckoo hashing with a stash. SIAM J Comput 39(4):1543–1561
24. Erlingsson U, Mcsherry F, Manasse M (2006) A cool and practical alternative to traditional hash tables. In: Proceedings of the 7th workshop on distributed data and structures (WDAS)
25. Fan B, Andersen DG, Kaminsky M (2013) MemC3: compact and concurrent memcache with dumber caching and smarter hashing. In: Proceedings of the 10th USENIX conference on networked systems design and implementation
26. Singh A, Garg S, Kaur K, Choo KKR (2018) Fuzzy-folded Bloom filter-as-a-Service for Big Data Storage in the Cloud". IEEE Trans Ind Inf 15:1–10
27. Jeong J, Joo JWJ, Lee Y, Son Y (2019) Secure cloud storage service using Bloom filters for the internet of things. IEEE Access 7:60897–60907
28. Najafimehr M, Ahmadi M (2019) SLCF: single-hash lookup Cuckoo filter. J High Speed Netw Pre-press(Pre-press):1–12
29. Reviriego P, Pontarelli S, Maestro JA, Ottavi M (2015) A synergetic use of Bloom filters for error detection and correction. IEEE Trans Very Large Scale Integr (VLSI) Syst 23(3):584–587
30. Lim H, Lee J, Byun H, Yim C (2016) Ternary Bloom filter replacing counting Bloom filter. Commun Lett 21(2):278–281
31. Ficara D, Pietro AD, Giordano S, Procissi G, Vitucci F (2010) Enhancing counting Bloom filters through huffman-coded multilayer. IEEE/ACM Trans Netw 18(6):1977–1987
32. Kumar A, Xu J, Wang J (2006) Space-code Bloom filter for efficient per-flow traffic measurement. IEEE J Sel Areas Commun 24(12):2327–2339
33. Hao F, Kodialam M, Lakshman TV, Song H (2012) Fast dynamic multiple-set membership testing using combinatorial Bloom filters. IEEE/ACM Trans Netw 20(1):295–304
34. Geravand S, Ahmadi M (2011) A novel adjustable matrix bloom filter-based copy detection system for digital libraries. In: 2011 IEEE 11th international conference on computer and information technology (CIT), pp 518–525. IEEE
35. Xiong S, Wang F, Cao Q (2016) A bloom filter based scalable data integrity check tool for large-scale dataset. In: First joint international workshop on parallel data storage and data intensive scalable computing systems (PDSW-DISCS), pp 55–60
36. Qiao Y, Li T, Chen S (2013) Fast bloom filters and their generalization. IEEE Trans Parallel Distrib Syst 25:93–103
37. Reviriego P, Christensen K, Maestro JA (2015) A comment on "fast Bloom filters and their generalization". IEEE Trans Parallel Distrib Syst 27:303–304
38. Crainiceanu A, Lemire D (2015) Bloofi: multidimensional Bloom filters. Inf Syst 54:311–324
39. Singh A, Garg S, Batra S, Kuma N, Rodrigues JJ (2018) Bloom filter based optimization scheme for massive data handling in IoT environment. Future Gener Comput Syst 82:440–449
40. Grandi F (2018) On the analysis of Bloom filters. Inf Process Lett 129:35–39
41. Yan Y, Wu L, Gao G, Wang H, Xu W (2018) A dunamic integrity verification scheme of cloud storage data based on lattice and bloom filter. J Inf Secur Appl 39:10–18
42. Zhang S, Zhou H, Yang Y, Wu Z (2017) A joint Bloom filter and cross-encoding for data verification and recovery in cloud. In 2017 IEEE symposium on computers and communications (ISCC), pp 614–619. IEEE

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.