Elastic Cuckoo Filter: Virtualizing, Shrinking, and Extending Cuckoo Filters

Jinyang Li
Peking University
lijinyang@pku.edu.cn

Jizhou Li
Peking University
ljzh2014@pku.edu.cn

Tong Yang
Peking University
yangtongemail@gmail.com

Zhaodong Kang
Peking University
kzd@pku.edu.cn

Aoran Li
Peking University
liaoran@pku.edu.cn

Dagang Li
Peking University
dgli@pkusz.edu.cn

Steve Uhlig
Queen Mary, University of London
steve@eecs.qmul.ac.uk

Ori Rottenstreich
Technion
or@cs.technion.ac.il

***Abstract*—Compared with Bloom filters, Cuckoo filters achieve a similar false positive rate for a given amount of memory. However, Cuckoo filters can support deletions while Bloom filters cannot, and thus Cuckoo filters have attracted great interests and attention in recent year. Unfortunately, the Cuckoo filter data structure is inflexible: 1) its size must be 2^n, where n is a positive integer; and 2) its size cannot be dynamically tuned. In this paper, we aim to make Cuckoo filters elastic. We propose the Elastic Cuckoo filter with three techniques: virtualization, shrinkage and extension. The size of the Elastic Cuckoo filter is flexible, and tunable, *i.e.*, it can be shrunk or extended flexibly. Theoretical and experimental results show that our Elastic Cuckoo filter is truely flexible and tunable, and inherits all the advantages of the standard Cuckoo filter: same false positive rate, same speed, and support of deletions. We provide the source code of our Elastic Cuckoo filter at Github.**

***Index Terms*—Cuckoo Filter, Elastic Cuckoo Filter, Virtualization, Shrinkage, Extension**

I. Introduction

A. Background and Motivation

Membership query is to tell whether a given item is a member of a given set or not. It is an important operation in many fields, such as databases [1], [2], network algorithms [3]–[5], distributed systems [6]–[8], caches [9]–[11], routers [12], [13], *etc.* The most classic solution for membership query is to rely on Bloom filters [14], thanks to their small overhead in time and space [14]–[19]. However, the standard Bloom filter cannot support deletions. Therefore, Cuckoo filters [20], which support deletions with limited additional overhead, have recently replaced Bloom filters in many applications [21]–[23].

Compared with the standard Bloom filter, 1) the Cuckoo filter achieves a comparable false positive rate; 2) it enjoys faster query speed, because for each query it only needs two hash probes while Bloom filters often need more than two hash probes; 3) it can support deletions because of its record of fingerprints[1], while the standard Bloom filter cannot; 4) its insertion speed is often faster than that of the Bloom filter when the Bloom filter needs to use many hash functions. Due to the importance of support for deletions, Cuckoo filters have more potential in practical applications than the Bloom filter [24], and have attracted wide attention in recent years.

However, the Cuckoo filter (CF) has two key shortcomings. First, its size is restricted: it must be 2^n long (the reason is detailed in Section II-A). Given the set size and the desired false positive rate, we can calculate the number of buckets (x) needed. For example, if $x = 1G + 1$, then CF needs 2G buckets, which is a waste of memory. Our goal is to use only 1G+1 buckets. Second, the size of a Cuckoo filter cannot be dynamically tuned (without reinserting all items), which is inconvenient especially when the set is dynamic or the set size is unknown in advance. The goal of this paper is to address the above two shortcomings to make the Cuckoo filter elastic.

B. Prior Art

The first shortcoming of Cuckoo filters was not pointed out in the original paper proposing the Cuckoo filter. In the open-source code they offered [25], the size of the Cuckoo filter is fixed as an integer power of 2. To the best of our knowledge, no prior work can overcome this limitations of size without additional memory overhead or loss of accuracy.

For the second shortcoming, the Dynamic Cuckoo filter (DCF) [22] proposes a solution. The idea of the DCF is straightforward: when the Cuckoo filter (CF) is nearly full, it just creates a new CF, and new incoming items will be inserted into the new CF. When the new CF is nearly full again, another new CF will again be created. To query an item, every CF must be queried in the worst case, which means that with more and more CFs created, this extension of the capacity linearly slows down the query speed, and linearly increases the false positive rate. Although the DCF claims to support shrinkage, its shrinkage is done by deleting CFs, not shrinking a standard CF. Performance analyses of the DCF are provided in Section II-B. Our goal is to shrink and extend a standard CF with no increase of the false positive rate.

[1]For example, we set the length of a fingerprint to be 16 bits, and for item e we use hash function $f(.)$ to compute $f(e) = 0x123456$. We use the lower 16 bits of this result as e's fingerprint ($\mathbb{F}_e = 0x3456$).

C. Our Solution

In this paper, we propose an enhanced Cuckoo filter, namely Elastic Cuckoo filter (ECF), which leverages *virtualization* to overcome the first shortcoming, and *shrinkage and extension* to overcome the second shortcoming.

(i) Virtualization. To enable Cuckoo filters to be of any size L, we use the technique of virtualization. For a given integer L, we can find the largest integer n such that $L = 2^n + w$, where $0 \leqslant w < 2^n$ ($n = \lfloor \log_2 L \rfloor$). For a given item e, we use a hash function to compute an offset (the offset value ranges from 0 to $L - 1$), and acquires 2^n buckets from this offset in a cyclic manner. These 2^n buckets are considered as a *virtual Cuckoo filter* (vCF) of item e, and all operations of Cuckoo filters can be done within this scope as usual. Namely, during any operation, item e is confined to these 2^n buckets, *i.e.*, this virtual Cuckoo filter (vCF). Since the starting position of a vCF is an offset, and an offset is computed by hashing an incoming item, then each item corresponds to a vCF. Different items often have different offsets, and thus are usually confined to different vCFs. As a result, n items corresponds to n different vCFs, regardless of hash collisions. All vCFs overlap, and form the physical CF. When the number of items is large, all vCFs are uniformly distributed across the physical Cuckoo filter of size L, so that each bucket has equal opportunity to hold items. In this way, the Cuckoo filter can be of any size with very small overhead: one additional hash function to compute the offset of the incoming item.

(ii) Shrinkage and Extension. To make the Cuckoo filter adapt to the size of sets, we propose to shrink and extend an ECF. Note that our shrinkage algorithm can also be applied to the standard CF and its variants, but extension algorithm can only be applied to ECF. When shrinking a CF, our key idea is called *divide-by-2*. Specifically, we traverse CF, and for each fingerprint in the i^{th} bucket, after shrinkage, its new bucket index is $\lfloor i/2 \rfloor$. Details are provided in Section IV. When extending a CF, our key idea is to copy the current CF and perform *lazy update*. Specifically, we first make a duplication of the current CF. As a result, each fingerprint has a redundant copy. A straightforward solution is to traverse the whole CF, and delete all redundant copies. We propose a new strategy called *lazy update*: instead of scanning the whole CF at once, we check all fingerprints and delete redundant copies in a bucket only when an item is tried to be inserted into this bucket. Details are provided in Section V.

D. Key Contributions

- We propose the Elastic Cuckoo filter, which uses virtualization, shrinkage and extension to make the Cuckoo filter elastic. We provide the source code of our Elastic Cuckoo filter at Github [26].
- We analyze the false positive rate and insertion failures of Elastic Cuckoo filter. Mathematical proofs show that our Elastic Cuckoo filter has the same false positive rate as Cuckoo filter.
- We conduct extensive experiments, and our experimental results show that the Elastic Cuckoo filter achieves almost the same performance (insertion speed, query speed and false positive rate) as the standard CF. And they also show that the ECF is truly elastic: (i) it can be of any size, and (ii) the shrinkage and extension processes are simple and fast.

II. Background and Related Work

In this section, we first detail the Cuckoo filter (CF) and dynamic Cuckoo filter (DCF), and then briefly survey the related work. For other related work, we refer interested readers to the literature [21], [27]–[32]. The notations and descriptions used in this paper are listed in Table I.

TABLE I
Notations and Descriptions

Notations	Descriptions
L	Number of buckets in the filter
n, w	$L = 2^n + w$, where $0 \leqslant w < 2^n$ and $n \geqslant 0$
M	Number of cells in each bucket
$\mathcal{T}$	The maximum number of relocations
$\Delta(.)$	a hash function to determine the offset of an item
$f(.), h(.), g(.)$	independent hash functions
$\mathbb{F}_e$	Fingerprint of item e
p	Length of a fingerprint in bits
$B[i]$	the i^{th} bucket
$B[i][j]$	the j^{th} cell in the i^{th} bucket
s_0	the offset (starting position) of a VCF
i_1, i_2	indices of two candidate buckets
d_1, d_2	distances between the two candidate buckets and the offset

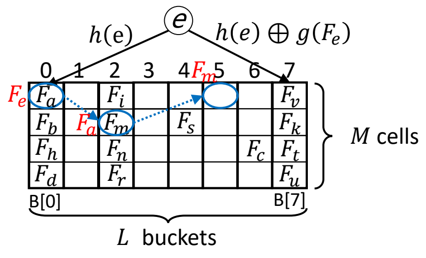


Fig. 1. An example of the Cuckoo filter.

A. The Cuckoo Filter

An example of CF: We use a specific example in Figure 1 to show the details of the Cuckoo filter. This figure shows a CF with 8 buckets, and each bucket has 4 cells. When inserting item e, CF computes $h(e)\%8 = 0$ and maps e to the first bucket $B[0]$. Then it computes $0 \oplus (g(\mathbb{F}_e)\%8) = 7$ and maps e to the last bucket $B[7]$, where $\mathbb{F}_e$ is the fingerprint of e. $B[0]$ and $B[7]$ are called the *candidate buckets* of e. We refer to $B[0]$ the as the *alternate bucket* of $B[7]$, and similarly, $B[7]$ the alternate bucket of $B[0]$. There is no empty cell in $B[0]$ and $B[7]$, so CF randomly kicks out a fingerprint ($\mathbb{F}_a$, the fingerprint of item a) in these two buckets to insert $\mathbb{F}_e$. Then it compute $\mathbb{F}_a$'s alternate bucket by $0 \oplus (g(\mathbb{F}_a)\%8) = 2$. Bucket $B[2]$ is also full, and CF kicks out $\mathbb{F}_m$, inserts $\mathbb{F}_a$ into $B[2]$, and computes $\mathbb{F}_m$'s alternate bucket by $2 \oplus (g(\mathbb{F}_m)\%8) = 5$. $B[5]$ has empty cells, so CF inserts $\mathbb{F}_m$ here and insertion succeeds.

Data Structure: A Cuckoo filter consists of L buckets. Each bucket consists of $M = 4$ cells. Each cell is used to store the fingerprint of an item.

Insertion: When inserting item e, CF first computes e's two candidate buckets $B[i_1]$ and $B[i_2]$ as follows:

$$\begin{aligned} i_1 &= h(e)\%L \\ i_2 &= i_1 \oplus (g(\mathbb{F}_e)\%L) \qquad (1) \\ meanwhile \quad i_1 &= i_2 \oplus (g(\mathbb{F}_e)\%L) \end{aligned}$$

If either of the two candidate buckets has an empty cell, $\mathbb{F}_e$ is inserted and the insertion succeeds immediately. If both candidate buckets are full, as the example in Figure 1, CF randomly *kicks out* a fingerprint to insert $\mathbb{F}_e$, and computes the *alternate bucket* of the evicted fingerprint by Equation (1). When the number of such kicks/relocations exceeds the predefined threshold, the insertion fails.

Query and Deletion: When querying or deleting an item e, CF computes its two candidate buckets, checks in these two buckets whether there is a fingerprint equal to $\mathbb{F}_e$ and reports the answer or deletes the fingerprint.

Advantages: First, when using the same size of memory, CF achieves similar false positive rate to that of Bloom filters [14]. Second, CF supports deletions while the Bloom filter cannot.

Shortcomings: CF is inflexible: (i) restricted size, *i.e.,* the size of a CF can only be an integer power of 2; (ii) the size of a CF cannot be dynamically tuned.

(i) Restricted Size: The reason is that the outcome of the XOR operation between two n-bit integers is also a n-bit integer, ranging from 0 to $2^n - 1$. For a CF whose size is not 2^n, the outcome of XOR could overflow its size, which will incur mistakes. For example, if the size of a CF is 10, then both i_1 and i_2 are 4-bit long. After the XOR operation, the result could range from 0 to 15, and errors will happen if the result falls into the range from 10 to 15.

(ii) No Support for Shrinkage or Extension: The second shortcoming making CF inflexible is that it does not support shrinkage or extension. When the number of items in a set is unknown in advance, if the size of CF is too large, it is a waste of memory; and if the size is too small, CF will soon become full. At present, only Dynamic Cuckoo filter (DCF) deals with the second shortcoming, and below we analyze how it works.

B. Dynamic Cuckoo Filter

The details of DCF [22] has been presented above in Section I-B. In this subsection, we mainly show the analysis of DCF.

Analysis of DCF: DCF inherits the main properties of CF and also offers to tune its size. However, the extension increases the false positive rate, and decreases the query and deletion speed linearly. Specifically, the false positive rate of DCF increases to $z \cdot f$, where z is the number of CFs and f is the false positive rate of one CF. In addition, a query or a deletion needs to probe all CFs in DCF in the worst case, with time complexity $O(z)$. Hence, with more extensions, the query and deletion speed slows down, and the false positive rate increases. The shrinkage of DCF is to reduce the number of CFs by re-inserting elements, which is time-consuming. This is actually not a shrinkage algorithm, because it cannot shrink one standard Cuckoo filter.

C. Other Related Work

Cuckoo filters can also be extended for other queries, such as range queries [33], [34], frequency queries [35]–[38], and more [39], [40]. For membership queries, other related work include Morton filter [41], Counting Quotient Filter [42], and Persistent Bloom filters [43]. Morton filter manages to improve the update speed of Cuckoo filter, but inherits the above mentioned two shortcomings of Cuckoo filter. Counting Quotient filter has more functions, supporting membership query, deletions, and counting, but is not elastic. Persistent Bloom filters [43] aims to find whether an item occurs during any time period in the history, and is thus quiet different from the design goal of this paper.

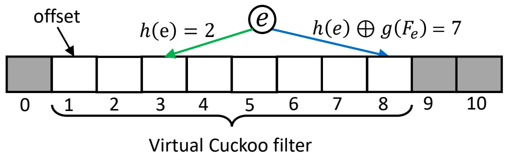


Fig. 2. An example of the virtualizing Cuckoo filters.

III. Virtualizing Cuckoo Filters

A. An Example of Virtualizing Cuckoo Filters

We use an example to show how we enable Cuckoo filters be any size. Figure 2 shows an Elastic Cuckoo filter (ECF) of size 11. For a given item e, we use hash function $\varDelta(.)$ to locate its *offset*. Suppose $\varDelta(\mathbb{F}_e)\%11 = 1$, then bucket $B[1]$ is e's offset. Since the number of buckets $L = 11 = 2^3 + 3$, the length of a *virtual Cuckoo filter (vCF)* is 8. We regard buckets $B[1 \sim 8]$ (the white buckets in the figure) as e's vCF. All operations of a standard CF can be done within this vCF as usual. The use of hash function $\varDelta(.)$ guarantees that vCFs of different items are distributed uniformly, so each bucket has the same probability to be mapped to. Different items have different offsets, and thus have different vCFs when regardless of hash collisions.

B. Virtualization of Cuckoo Filter

Data Structure: The Elastic Cuckoo filter consists of L buckets (L can be any integer), and each bucket consists of 4 cells. Each cell is used to store the fingerprint of an item. For a given item e, we use $\varDelta(\mathbb{F}_e)\%L$ to locate the start of its virtual Cuckoo filter (the offset). The 2^n buckets starting from the offset, form the vCF of e. e is confined to this range and all operations of the standard Cuckoo filter (including XOR and Kick) can be done within this scope as usual. We refer to the L buckets in a cyclic manner: the first bucket is the next one from the last one. Therefore, no matter where the offset is, we can find 2^n continuous buckets and get the vCF of an item.

Insertion: The pseudo-code of insertion is shown in Algorithm 1. Initially, each cell in every bucket of ECF is empty. For each incoming item e, we first compute its fingerprint $\mathbb{F}_e$

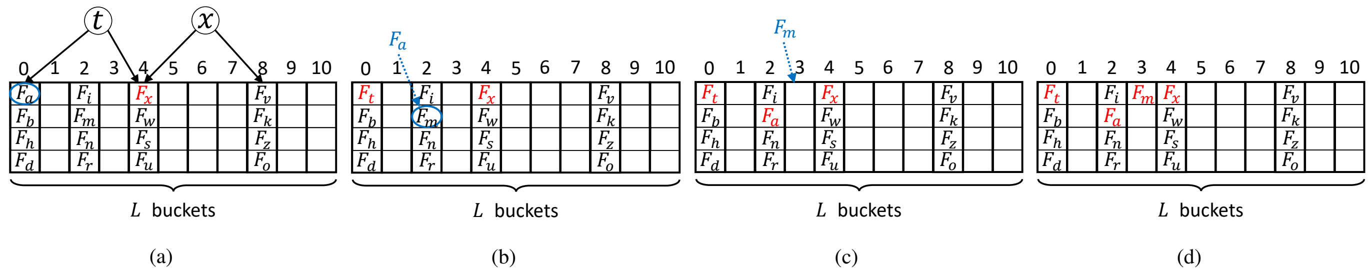


Fig. 3. Examples of Insertion in Elastic Cuckoo filter.

```
Algorithm 1: Insertion
  Input: A given item $e$
  Output: An indication whether $e$ is successfully inserted.
1 Compute $e$'s fingerprint $\mathbb{F}_e$;
2 Compute $e$'s offset $s_0$ by $s \leftarrow \Delta(\mathbb{F}_e)\%L$, $s_0 \leftarrow s - s\%2$;
3 Compute $e$'s two candidate buckets $B[i_1]$ and $B[i_2]$;
4 if $B[i_1]$ or $B[i_2]$ has empty cells then
5     Insert $e$ into one among the empty cells;
6     Return true;
7 else
8     $i \leftarrow$ randomly pick $i_1$ or $i_2$;
9     for $n = 0; n < \mathcal{T}; n++$ do
10        Randomly select a cell from $B[i]$;
11        Kick out the fingerprint $\mathbb{F}$ in that cell;
12        Store $\mathbb{F}_e$ in that cell;
13        Compute $\mathbb{F}$'s alternate bucket $B[i']$;
14        $i \leftarrow i'$;
15        $\mathbb{F}_e \leftarrow \mathbb{F}$;
16        if $B[i]$ has an empty cell then
17            Store $\mathbb{F}_e$ in that cell;
18            Return true;
19 Return false;
```

and offset s_0. Then we compute the two candidate buckets $B[i_1]$ and $B[i_2]$ using the following equations:

$$\begin{aligned} i_1 &= (s_0 + d_1)\%L \\ i_2 &= (s_0 + d_2)\%L \\ \textit{where} \quad \mathbb{F}_e &= f(e)\%(2^p) \\ s &= \Delta(\mathbb{F}_e)\%L \\ s_0 &= s - s\%2 \\ d_1 &= h(e)\%(2^n) \\ d_2 &= d_1 \oplus (g(\mathbb{F}_e)\%(2^n)) \\ &= ((i_1 - s_0 + 2^n)\%2^n) \oplus (g(\mathbb{F}_e)\%(2^n)) \end{aligned} \tag{2}$$

Note that we confine the offset s_0 to an even integer by $s_0 = s - s\%2$, due to the need of shrinkage, which will be detailed in Section IV. After getting the two candidate buckets, there are two cases:

Case 1: There is at least one empty cell in the two buckets. Then, we insert e's fingerprint into the first empty cell in that bucket. If both buckets have empty cells, we choose bucket $B[i_1]$.

Case 2: Neither of these two buckets has empty cells. In this case, we randomly select a bucket and randomly select a fingerprint from it, then *kick out* the fingerprint to insert $\mathbb{F}_e$ into that cell. Let $B[i]$ containing fingerprint $\mathbb{F}$ be the bucket we select. We replace $\mathbb{F}$ with $\mathbb{F}_e$ and relocate the *evicted fingerprint* $\mathbb{F}$ to its *alternate bucket* $B[alt]$.

We compute

$$\begin{aligned} alt &= (s'_0 + d')\%L \\ \textit{where} \quad s' &= \Delta(\mathbb{F})\%L \\ s'_0 &= s' - s'\%2 \\ d' &= ((i - s'_0 + 2^n)\%2^n) \oplus (g(\mathbb{F})\%(2^n)) \end{aligned} \tag{3}$$

and get $\mathbb{F}$'s alternate bucket $B[alt]$. If $B[alt]$ has an empty cell, we insert $\mathbb{F}$ into it, and the insertion succeeds. Otherwise, we continue to select a cell and kick out the existing fingerprint to insert $\mathbb{F}$. Then, we relocate the new evicted fingerprint using the same method. This process ends when an empty cell is found or when the number of relocations exceeds the predefined threshold $\mathcal{T}$. If an empty cell is found while the number of relocations is less than $\mathcal{T}$, the insertion succeeds. Otherwise, the Elastic Cuckoo filter is considered too full and our extension algorithm will be activated (see Section V.)

Examples of Insertion: Figure 2 shows how to compute two candidate buckets using virtualization, and figure 3 shows the process of kick mechanism during insertions. We set L to 11 and $\mathcal{T}$ (the maximum number of relocations) to 10. In Figure 2, when inserting item e, we first compute its offset and get its vCF. As mentioned above, e's vCF is buckets $B[1 \sim 8]$. Then we have $h(e)\%8 = 2, (1+2)\%11 = 3$, so e's first candidate bucket is $B[9]$. Suppose $2 \oplus (g(\mathbb{F}_e))\%8 = 7$, $(1+7)\%11 = 8$, and thus the second candidate bucket is $B[8]$. So item e is mapped to $B[3]$ and $B[8]$. Based on this method to compute an item's two candidate buckets, we have figure 3. As shown in Figure 3(a), when inserting item x, we compute its two candidate buckets: $B[4]$ and $B[8]$. There is an empty cell in $B[4]$, so this is **case 1**. We insert the fingerprint of x into this cell ($\mathbb{F}_x$ in red in the figure). When inserting item t, we compute its two candidate buckets: $B[0]$ and $B[4]$, both of which are full. So this is **case 2**. Then we randomly kick out a fingerprint in these two buckets to insert $\mathbb{F}_t$. We kick out and replace $\mathbb{F}_a$ with $\mathbb{F}_t$ in $B[0]$, and compute the alternate bucket of $\mathbb{F}_a$ by $0 \oplus (g(\mathbb{F}_a))\%8 = 2$ (Figure 3(b)). Bucket $B[2]$ is also full, and we kick out $\mathbb{F}_m$ to insert $\mathbb{F}_a$ in $B[2]$ (Figure

3(c)). Then we compute $\mathbb{F}_m$'s alternate bucket as $B[5]$, which has empty cells. Therefore, we insert $\mathbb{F}_m$ into an empty cell in $B[5]$ and the insertion succeeds (3(d)).

Query: To query whether an item e is in Elastic Cuckoo filter, we first compute $e's$ fingerprint $\mathbb{F}_e$, and get two candidate buckets according to Equation (2). If there is an existing fingerprint stored in either of the two buckets equal to $\mathbb{F}_e$, the answer is true. Otherwise, the answer is false. Due to space limitation, pseudo-codes of query and deletion are in Section D of our technical report [44].

Deletion: To delete an item e, we first query e. If the answer is true, we delete the corresponding fingerprint from the candidate bucket. Otherwise, we do nothing.

Time Complexity: The time complexities of ECF insertion, query and deletion are the same as those of CF. Specifically, the amortized time complexity for ECF (CF) insertion is $O(1)$, and time complexities for ECF (CF) query and deletion are both $O(1)$.

C. Proofs of Relocation

Now we show a theorem about the correctness of the relocation process. In other words, given a fingerprint in a bucket, the alternate bucket can be calculated using the same equation (Equation (3)). The XOR operation in Equation (2) guarantees this property.

Theorem III.1. *The size of Elastic Cuckoo filter is L, and $2^n \leqslant L < 2^{n+1}$, where n is an integer. For an item e, let B_1, B_2 be its two buckets as expressed in Equation (2), with indices i_1, i_2.*

Let B_j (for $j \in \{1,2\}$) be one such bucket with index i_j containing a finger F. The other bucket B_{3-j} with index i_{3-j} can be computed as:

$$\begin{aligned} i_{3-j} &= (s_0 + d_{3-j})\%L \\ where \quad s &= \Delta(\mathbb{F}_e)\%L \\ s_0 &= s - s\%2 \\ d_{3-j} &= ((i_j - s_0 + L)\%L) \oplus (g(\mathbb{F}_e)\%2^n) \end{aligned} \tag{4}$$

Proof. By Equation (2) the value d_j is given by $d_j = (i_j - s_0 + L)\%L$, and by the connection between d_1, d_2, necessarily,

$$\begin{aligned} d_{3-j} &= d_j \oplus (g(\mathbb{F}_e)\%(2^n)) \\ &= ((i_j - s_0 + L)\%L) \oplus (g(\mathbb{F}_e)\%2^n) \end{aligned}$$

Thus the other index can be computed as $i_{3-j} = (s_0 + d_{3-j})\%L$, which is the same as Equation (4). Theorem holds. □

IV. Shrinking Elastic Cuckoo Filters

In this section, we show how to shrink Elastic Cuckoo filters, and how to extend them in the next section. Note that our shrinkage method can also be applied to the standard Cuckoo filter and its variants. We offer to shrink an ECF from size L to $L/2$, and extend an ECF from size L to αL, where α is a positive integer. We call the ECF before shrinkage or extension the *old ECF*, and call the ECF after shrinkage or extension the *new ECF*.

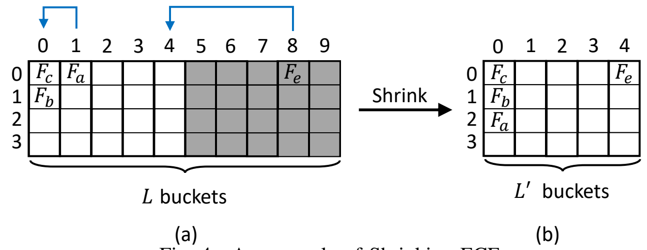


Fig. 4. An example of Shrinking ECF.

A. An Example of Shrinking ECF

Before detailing the shrinkage algorithm, we show an example of shrinking an ECF from size L to $L/2$. Figure 4 shows an example of shrinking an ECF from size 10 ($L = 10$) to size 5 ($L' = 5$). We first traverse the old ECF in Figure 4(a) from $B[0]$ to $B[9]$. For the two fingerprints in $B[0]$, since $\lfloor 0/2 \rfloor = 0$, they keep in the same positions. For fingerprint $\mathbb{F}_a$ in $B[1]$, since $\lfloor 1/2 \rfloor = 0$, we move $\mathbb{F}_a$ to $B[0]$. So in the new ECF in Figure 4(b), $\mathbb{F}_a$ is in cell $B[0][2]$. For fingerprint $\mathbb{F}_e$ in $B[8]$, since $\lfloor 8/2 \rfloor = 4$, we move $\mathbb{F}_e$ to $B[4]$. Now we have processed all fingerprints in the old ECF, so we delete the second half of the old ECF and get the new ECF, which is shown in Figure 4(b).

Algorithm 2: Shrinkage

```
   Input: An ECF of size $L$
   Output: An ECF of size $L/2$
1  if $L$ is an odd integer then
2      $L \leftarrow L + 1$;
3  for $i = 0; i < L/2; i++$ do
4      for $j = 0; j < 4; j++$ do
5          if $B[i][j]$ is empty then
6              Continue;
7          else
8              /*Assume $B[i][j]$ holds fingerprint $\mathbb{F}$*/
9              if $B[i/2]$ is full then
10                 Move $\mathbb{F}$ to the stash;
11             else
12                 Move $\mathbb{F}$ to the $B[i/2]$;
13 Delete buckets from $B[L/2]$ to $B[L-1]$, inclusively;
```

B. Basic Version of the Shrinkage Algorithm

We show the basic version of the shrinkage algorithm. The pseudo-code is provided in Algorithm 2. Our key idea is *divide-by-2*. We divide the offset of each item by 2 (divide s_0 by 2), and divide the distance between the candidate buckets and the offset by 2 (divide d_1, d_2 by 2). *Note that in this paper, if the result of a division is not an integer, we round it to the next integer, e.g.,*$3/2 = 1$. Recall that the indices of the two candidate buckets satisfy $i_1 = s_0 + d_1$ and $i_2 = s_0 + d_2$.

Since s_0 is confined to an even integer (see Equation (2) in Section III), with rounding, we have $i_1/2 = s_0/2 + d_1/2$ and $i_2/2 = s_0/2 + d_2/2$. Thus, when shrinking an ECF from size L to size $L' = L/2$, we just move all fingerprints forward, from bucket $B[i]$ to $B[i/2]$. If $B[i/2]$ is full, we move this fingerprint to a *stash*[2]. When querying or deleting an item, if it is not found in the ECF, we need to query it in the stash.

Note that when L is an odd integer, we simply add an empty bucket at the end of the filter, and shrink the ECF from size $L+1$ to $(L+1)/2$. After shrinkage, the new offset $s_0' = s_0/2$ may not be an even integer, so the equation $i_1'/2 = s_0'/2 + d_1'/2$ may not hold. This basic version can therefore only shrinks the ECF once. For repeated shrinkage, we have the *multiple version*. Due to space limitation, we present analysis of the basic version of shrinkage in technical report [44], Section B.

C. The Multiple Version of Shrinkage Algorithm

To shrink the ECF for one or more times, the offset is not constrained to be even. The main idea is to move each fingerprint forward, from $B[i]$ to $B[i/2]$ or $B[i/2-1]$. The choice depends on the value of $i\%2$ and $s_0\%2$.

For fingerprint $\mathbb{F}_e$ in $B[i]$, there are three cases.

Case 1: i is odd. In this case, we move $\mathbb{F}_e$ to $B[i/2]$.

Case 2: i is even, and we compute $\mathbb{F}_e$'s offset, which is also even. In this case, we move $\mathbb{F}_e$ to $B[i/2]$.

Case 3: i is even, and we compute $\mathbb{F}_e$'s offset, which is odd. In this case, we move $\mathbb{F}_e$ to $B[i/2-1]$.

Analysis: The computation of two candidate buckets and alternate bucket in the new ECF is similar to the basic version (Equation (15), (16)), and the only difference is that $s_0 = \varDelta(\mathbb{F}_e)\%L$. Thus, we omit them. We only prove that the shrinkage algorithm moves fingerprints to their candidate buckets in the new ECF. For fingerprint $\mathbb{F}_e$ in $B[i]$ (in the old ECF), we analyze the three cases above one by one. Without loss of generality, we assume $i = i_1$.

Case 1: i is odd. It means that one of s_0 and d_1 is odd, and the other is even. Then we have $i_1/2 = s_0/2 + d_1/2$. Thus, $i_1' = i_1/2$. Therefore, in this case we move $\mathbb{F}_e$ to $B[i/2]$.

Case 2: i is even, and $\mathbb{F}_e$'s offset is also even. It means that both s_0 and d_1 are even. Then we have $i_1/2 = s_0/2 + d_1/2$. Thus, $i_1' = i_1/2$. Therefore, in this case we move $\mathbb{F}_e$ to $B[i/2]$.

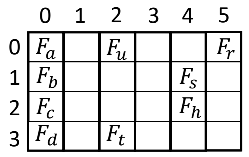
Case 3: i is even, and $\mathbb{F}_e$'s offset is odd. It means that both s_0 and d_1 are odd. Then we have $i_1/2 - 1 = s_0/2 + d_1/2$. Thus, $i_1' = i_1/2 - 1$. Therefore, in this case we move $\mathbb{F}_e$ to $B[i/2-1]$.

Thus, our shrinkage algorithm moves each fingerprint to one of its candidate buckets in the new ECF.

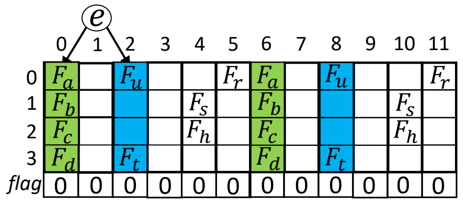
V. Extending Cuckoo Filters

In this section, we show how to extend an ECF *from size L to size $L' = \alpha L$, where α is a positive integer.* Similar to $L = 2^n + w$, we have $L' = 2^{n'} + w'$, where $0 \leqslant w < 2^{n'}$ and $n' \geqslant 0$. We detail it as follows. Note that our extension method *cannot* be applied to the standard Cuckoo filter.

[2]A stash [45] is a small list outside the table, which is used to store a constant number of items so as to reduce the times of rehashing.

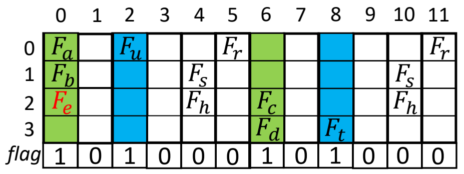


(a) Old ECF.



e
0 1 2 3 4 5 6 7 8 9 10 11
0 F_a F_u F_r F_a F_u F_r
1 F_b F_s F_b F_s
2 F_c F_h F_c F_h
3 F_d F_t F_d F_t
flag 0 0 0 0 0 0 0 0 0 0 0 0

(b) New ECF: After Copying.



0 1 2 3 4 5 6 7 8 9 10 11
0 F_a F_u F_r F_r
1 F_b F_s F_s
2 F_e F_h F_c F_h
3 F_d F_t
flag 1 0 1 0 0 0 1 0 1 0 0 0

(c) New ECF: After Inserting Item e.

Fig. 5. Extension of ECF.

A. An Example of Extension

Our idea of the extension algorithm is to copy the current ECF and perform *lazy updates*. Note that the size of the ECF increases, but the size of a Virtual Cuckoo filter maintains the same (2^n). Before detailing the extension algorithm, we provide an example.

As shown in Figure 5, we set L to be 6, L' to be 12, α to be 2. First, we copy and connect two ECFs in Figure 5(a), and get the new ECF of size 12. We add to each bucket a *flag bit*, initially set to 0. Then, we perform *lazy updates*: update a bucket only when it is accessed in an insertion. When inserting item e, we first compute its two candidate buckets $B[i_1']$ and $B[i_2']$ using the following equation.

$$\begin{aligned} i_1' &= (s_0' + d_1)\%L' \\ i_2' &= (s_0' + d_2)\%L' \\ where \quad \mathbb{F}_e &= f(e)\%(2^p) \\ s' &= \varDelta(\mathbb{F}_e)\%L' \\ s_0' &= s' - s'\%2 \\ d_1 &= h(e)\%(2^n) \\ d_2 &= d_1 \oplus (g(\mathbb{F}_e)\%(2^n)) \\ &= ((i_1 - s_0' + 2^n)\%2^n) \oplus (g(\mathbb{F}_e)\%(2^n)) \end{aligned} \tag{5}$$

As shown in Figure 5(b), assume that e's two candidate buckets are $B[0]$ and $B[2]$. The flag bits of these two buckets are both false, so before inserting e, we need to update them. We only show how to update $B[0]$ as an example. Since $(0+L)\%L' = (0+6)\%12 = 6$, $B[0]$ and $B[6]$ store the same fingerprints in the same cells, and one of the two same fingerprints is redundant, which we need to delete redundancy. For fingerprint $\mathbb{F}_a$ in $B[0][0]$ and $B[6][0]$, we compute its offset by $\varDelta(\mathbb{F}_a)\%L' = 10$. Since the length of a vCF keeps unchanged after extension (4 in this example), $\mathbb{F}_a$'s vCF is

$B[10 \sim 11]$ and $B[0 \sim 1]$. $B[0]$ is located within this scope while $B[6]$ is not, so $\mathbb{F}_a$ in $B[6][0]$ is redundant, and we delete it. Similarly, we delete $\mathbb{F}_b$ in $B[6][1]$, $\mathbb{F}_c$ in $B[0][2]$ and $\mathbb{F}_d$ in $B[0][3]$. Then, buckets $B[0]$ and $B[6]$ have been updated and we set their flag bits to true. After updating the $B[0]$ and $B[6]$, we insert item e's fingerprint to an empty cell ($B[0][2]$) in these two candidate buckets and the insertion succeeds. We get the new ECF in Figure 5(c).

B. The Extension Algorithm

When extending an ECF from size L to L', where $L' = \alpha L$, we first copy the old ECF to get α identical ECFs. Then, we merge them together and get a new ECF of size L'. Note that after extension, the size of a vCF keeps unchanged. Each fingerprint in the new ECF has $\alpha - 1$ redundant copies, and we delete them through *lazy updates*: update a bucket only when an item is tried to be inserted into this bucket. We add a *flag bit* to each of the L' buckets to record whether it has been *updated*. Initially, all flag bits are set to false (0). When inserting an item e, we calculate its two candidate buckets $B[i'_1]$ and $B[i'_2]$ using Equation (5). If both buckets' flag bits are true, which means that they have been updated, we perform the insertion directly. Otherwise, for each of the two candidate buckets, if its flag bit is false, we *update* it, and then continue the insertion. The process of updating a bucket will be detailed later. When querying or deleting an item, we just check fingerprints in the two candidate buckets using Equation (5) but do not update buckets. When we need to extend the ECF again, we first update all buckets that have not been updated, and then carry out the extension.

Update: When updating a bucket, we repeat the same operations for each fingerprint in it. We take fingerprint $\mathbb{F}_e$ in cell $B[t][m]$ for example. After the copy operation, there are α cells including $B[t][m]$ storing the same fingerprint $\mathbb{F}$. However, only one cell is the correct position for $\mathbb{F}_e$, so we need to delete the fingerprints in the other cells. We first calculate $\mathbb{F}_e$'s offset in the new ECF $B[s'_0]$ as follows:

$$\begin{aligned} s' &= \mathit{\Delta}(\mathbb{F})\%L' \\ s'_0 &= s' - s'\%2 \end{aligned} \tag{6}$$

From the offset, we can find $\mathbb{F}_e$'s vCF with size 2^n. Then among all the above α cells, only the cell located in the vCF is the correct position for $\mathbb{F}$, so we maintain this fingerprint and remove the fingerprints in the other $\alpha - 1$ cells. After repeating these operations for each of the fingerprints in the α buckets, we set the flag bits of all these α buckets to true, since when we update a bucket, actually we update all the α buckets at the same time. We represent the analysis of the extension algorithm in technical report [44] Section C.

Cost of Extension: The time complexities of ECF extension and DCF extension are both $O(1)$. DCF extension does not need additional memory, while ECF extension need L' additional bits (L' is the length of ECF after extension). After extension, for ECF, query and deletion both need $O(1)$ time, while for DCF, query and deletion both need $O(z)$ time, where z is the number of CFs in a DCF. When all the CFs in DCF have the same false positive f, DCF extension increases false positive rate from f to zf, while our ECF extension maintains the false positive rate f unchanged.

VI. Mathematical Analysis

In this section, we first derive the lower bound for the probability of an insertion failure of the Elastic Cuckoo filter, then we show the lower bound for probability of insertion failure of the Elastic Cuckoo filter.

A. Upper Bound for the False Positive Rate of the Elastic Cuckoo filter

Theorem VI.1. *Given an Elastic Cuckoo filter with L buckets, each with M cells. The probability of a false positive satisfies*

$$P \leqslant 1 - \left(1 - \frac{1}{2^p}\right)^{2M} \approx \frac{2M}{2^p} \tag{7}$$

where p is the length of each fingerprint in bits. Note that this false positive rate equation of the Elastic Cuckoo filter is the same as that of the Cuckoo filter in [20].

Proof. Recall that when querying an item e, the Elastic Cuckoo filter checks every cell in the two candidate buckets and reports true if e is matched against a fingerprint in any cell. In each cell, the probability that e is matched against the fingerprint stored there is $\frac{1}{2^p}$. So the probability that e is not matched is $1 - \frac{1}{2^p}$. The probability that each of the $2M$ cells is not matched is $(1 - \frac{1}{2^p})^{2M}$. Thus we have

$$P \leqslant 1 - \left(1 - \frac{1}{2^p}\right)^{2M} \approx \frac{2M}{2^p}$$

So the theorem holds. □

B. Lower Bound for Probability of Insertion Failure of the Elastic Cuckoo filter

Theorem VI.2. *Given a set S with N random items, and an Elastic Cuckoo filter with L buckets, each with M cells, the insertion fails with a probability P_{fail}. We do not take shrinkage into consideration so we do not need to confine the offset as even. Let p be the length of each fingerprint in bits. We have*

$$\begin{aligned} P_{fail} &\geqslant \binom{n}{2M+1}\left(\frac{2}{2^{n+p}} + \frac{\alpha}{2^{2n}}\right)^{2M} \\ where \;\; & 2^n \leqslant L < 2^{n+1} \\ & \alpha = -\frac{2}{3}\beta^2 + 4\beta - 8 + \frac{20}{3\beta} \in (2/3, 2] \\ and \;\; & \beta = \frac{L}{2^n} \in [1, 2) \end{aligned} \tag{8}$$

Proof. First, we derive the probability that a set with q items collide in the same two buckets. Recall that for item e and x, the calculation of their mapped buckets $B[i_{e1}], B[i_{e2}]$ for e and $B[i_{x1}], B[i_{x2}]$ for x is as follows:

$$
\begin{aligned}
&i_{e1} = (s_e + d_{e1})\%L \\
&i_{e2} = (s_e + d_{e2})\%L \\
where \quad &\mathbb{F}_e = f(e)\%(2^n) \\
&s_e = \Delta(\mathbb{F}_e)\%L \\
&d_{e1} = h(e)\%(2^n) \\
&d_{e2} = ((i_{e1} - s_e + L)\%L) \oplus (g(\mathbb{F}_e)\%(2^n))
\end{aligned} \tag{9}
$$

$$
\begin{aligned}
and \quad &i_{x1} = (s_x + d_{x1})\%L \\
&i_{x2} = (s_x + d_{x2})\%L \\
where \quad &\mathbb{F}_x = f(x)\%(2^n) \\
&s_x = \Delta(\mathbb{F}_x)\%L \\
&d_{x1} = h(x)\%(2^n) \\
&d_{x2} = ((i_{x1} - s_x + L)\%L) \oplus (g(\mathbb{F}_x)\%(2^n))
\end{aligned}
$$

Assume that item e with fingerprint $\mathbb{F}_e$ is mapped to buckets $B[i_{e1}]$ and $B[i_{e2}]$, and its offset is s_e. For another item x, there are three possible cases where it collides with e in the same two buckets.

Case 1: Item x satisfies: 1) It has the same fingerprint as $\mathbb{F}_e$ and 2) x's first mapped bucket is either $B[i_{e1}]$ or $B[i_{e2}]$. The probability of the fingerprint collision is $1/2^p$. For the second term, there are three situations:

Situation 1: $d_{e1} \neq d_{e2}$ and $d_{x1} = d_{e1}$.

Situation 2: $d_{e1} \neq d_{e2}$ and $d_{x1} = d_{e2}$.

Situation 3: $d_{e1} = d_{e2}$ and $d_{x1} = d_{e1}$.

The probability of the first two situations is:

$$\frac{2^n - 1}{2^n} \cdot \frac{2}{2^n}$$

The probability of the third situation is:

$$\frac{1}{2^n} \cdot \frac{1}{2^n}$$

Therefore, the probability of **Case 1** is:

$$
\begin{aligned}
P_1 &= \frac{1}{2^p} \cdot \left(\frac{2^n - 1}{2^n} \cdot \frac{2}{2^n} + \frac{1}{2^n} \cdot \frac{1}{2^n} \right) \\
&= \frac{2^{n+1} - 1}{2^{2n+p}}
\end{aligned} \tag{10}
$$

Case 2: Item x's fingerprint is different from e's, but x's two mapped buckets are the same as e's. The probability of no fingerprint collision is $(1 - 1/2^p)$. Assume that the length of the overlapping part of the two items' virtual Cuckoo filters is Y. We have $2^{n+1} - L \leqslant Y \leqslant 2^n$. There are two situations leading to bucket collisions.

Situation 1: $d_{e1} = d_{e2}$, $d_{x1} = d_{e1}$ and $d_{x2} = d_{e2}$.

Situation 2: $d_{e1} \neq d_{e2}$, $d_{x1} = d_{e1}$ and $d_{x2} = d_{e2}$.

Situation 3: $d_{e1} \neq d_{e2}$, $d_{x1} = d_{e2}$ and $d_{x2} = d_{e1}$.

The probability of the first situation is:

$$\frac{1}{2^n} \cdot \frac{Y}{2^n} \cdot \frac{1}{2^n} \cdot \frac{1}{2^n} = \frac{Y}{2^{4n}}$$

The probability of the other two situations is:

$$\left(1 - \frac{1}{2^n}\right)\left(\frac{Y}{2^n}\right)^2 \cdot \frac{2}{2^n 2^n} = \frac{2Y(Y-1)}{2^{4n}}$$

The probability of all the three situations is hence:

$$\frac{Y}{2^{4n}} + \frac{2Y(Y-1)}{2^{4n}} = \frac{2Y^2 - Y}{2^{4n}}$$

Recall that we have $2^{n+1} - L \leqslant Y \leqslant 2^n$. The probability that $Y = 2^n$ is $1/L$. The probability that $Y = 2^{n+1} - L$ is $(2^{n+1} - L + 1)/L$. The probability that Y is equal to any other integer is $2/L$. Thus, the probability of **Case 2** is as follows:

$$
\begin{aligned}
P_2 = &\left(1 - \frac{1}{2^p}\right) \cdot \frac{1}{L} \cdot \frac{2 \cdot 2^{2n} - 2^n}{2^{4n}} \\
&+ \left(1 - \frac{1}{2^p}\right) \cdot \frac{2^{n+1} - L + 1}{L} \cdot \frac{2(2^{n+1} - L)^2 - (2^{n+1} - L)}{2^{4n}} \\
&+ \left(1 - \frac{1}{2^p}\right) \cdot \frac{2}{L} \cdot \sum_{Y = 2^{n+1} - L + 1}^{2^n - 1} \frac{2Y^2 - Y}{2^{4n}} \\
&\approx \left(1 - \frac{1}{2^p}\right) \frac{1}{2^{2n}} \left(-\frac{2}{3}\beta^2 + 4\beta - 8 + \frac{20}{3\beta}\right) \\
&where \ \beta = \frac{L}{2^n} \in [1, 2)
\end{aligned} \tag{11}
$$

Therefore, the probability that two different items' fingerprints collide in the same two buckets is

$$
\begin{aligned}
P_{12} &= P_1 + P_2 \\
&\approx \frac{2^{n+1} - 1}{2^{2n+p}} + \left(1 - \frac{1}{2^p}\right) \frac{1}{2^{2n}} \left(-\frac{2}{3}\beta^2 + 4\beta - 8 + \frac{20}{3\beta}\right) \\
&\approx \frac{2}{2^{n+p}} + \frac{\alpha}{2^{2n}} \\
&where \ \alpha = -\frac{2}{3}\beta^2 + 4\beta - 8 + \frac{20}{3\beta} \in (2/3, 2]
\end{aligned} \tag{12}
$$

Then, the probability that a set with q items collide in the same two buckets is

$$P_c = P_{12}^{q-1} \tag{13}$$

During a construction process which inserts N items in a filter with L buckets, each bucket consisting of M cells, whenever $2M + 1$ items collide in the same two buckets, an insertion failure happens. The number of different possible sets of $2M + 1$ items out of N items is $\binom{n}{2M+1}$. Since such kind of collisions is a typical type among all collisions, the probability of such collisions is the lower bound of P_{fail}. Then we have

$$
\begin{aligned}
P_{fail} &\geqslant \binom{n}{2M+1} P_{12}^{q-1} \\
&= \binom{n}{2M+1} \left(\frac{2}{2^{n+p}} + \frac{\alpha}{2^{2n}}\right)^{2M}
\end{aligned} \tag{14}
$$

Therefore, the theorem holds. □

VII. Experimental Results

A. Experimental Setup

Platform: Our experiments are performed on a server with an 8-core CPU (Intel(R) Core(TM) i7-6700HQ CPU @2.60 GHz) and 16 GB total system memory. Each core has one

L1 cache with 256KB memory and one L2 cache with 1MB memory. All cores share one L3 cache with 6MB memory.

Datasets:

(i) CAIDA: This dataset is from the *CAIDA Anonymized Internet Trace 2016*, in which each data is 13 bytes long. [46], consisting of IP packets. Each item is identified by the source and the destination IP address. The dataset contains 10M items, with around 4.2M distinct items.

(ii) Synthetic dataset: We generate a synthetic dataset by randomly generating unique numbers. Each number in this dataset is 13 bytes long.

Implementation: The implementation of the Elastic Cuckoo filter (ECF) is done in C++. We also implemented the other related algorithms, including Cuckoo filter (CF) in C++.

B. Evaluation Metrics

Successfully inserted items: Given a certain false positive rate and let each bucket have four cells, we change the memory size and count the number of items successfully inserted in ECF and that of CF over two datasets.

Throughput: We perform insertions and queries for all items, record the total time taken, and calculate the throughput. The throughput is defined as $\frac{N}{T}$, where T is the total measured time and N is the total number of items. We use Thousand of operations per second (Kips) as unit to measure the throughput.

Kicks: We perform insertions of a number of items, record the total number of relocations, i.e., the total number of kick out operations, for different load ratios.

Reads: We count a read for each reading operation of an address in an array, *i.e.*, reading a cell increases the number of reads by 1. We perform insertions of all items, and record the total number of memory reads.

Writes: We count a write for each writing operation of an address in an array, *i.e.*, writing a cell increases the number of writes by 1. We perform insertions of all items, and record the total number of memory writes.

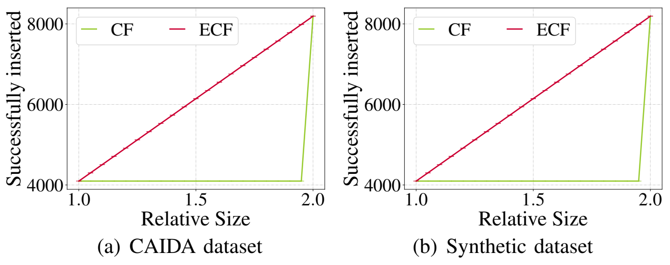


Fig. 6. The number of successfully inserted items.

C. Experimental Results

Number of Successfully Inserted Items (Figure 6): Our results show that for ECF, the number of successfully inserted items linearly increases with the increase of memory size, while for CF, the number of successfully inserted items keeps unchanged until the memory reaches an integer power of 2. The main reason is that the memory size of ECF can be any positive integer, while that of CF is limited as an integer power of 2.

ECF Insertion Throughput of Different Memory Sizes (Figure 7): Our results show that ECF's insertion throughput for two different memory sizes (24K buckets and 32K buckets) are roughly the same. And as the load ratio increases from 0.60 to 0.98, the insertion throughput mainly decreases from 4K Kips to 1.5K Kips. We perform experiments for ECF on the CAIDA and the synthetic data set for two different memory sizes (24K buckets and 32K buckets) and each bucket has 4 cells. Note that 24K is not the size of an standard CF. The major reason for the same performance is that our ECF works well for different sizes, including sizes that are not an integer power of 2. And the reason for the decrease of throughpot is that as more items are inserted into ECF, it is more different to find an empty cell, so more relocations are needed, and insertion speed decreases.

Insertion Throughput of ECF and CF (Figure 8): Our results show that ECF's insertion throughput for two different memory sizes (24K buckets and 32K buckets) are similar, and the throughput of CF is a little higher than that of ECF. As the load ratio increases from 0.60 to 0.98, the insertion throughput mainly decreases from 4K Kips to 1.5K Kips. We perform the experiments on CF and ECF with the same memory size (32K buckets, 4 cells for each bucket) on the CAIDA and synthetic datasets. We record the insertion throughput of the two data structures for load ratios ranging from 0.6 to 0.98. The main reason for the difference of ECF and CF is that ECF needs to compute one more hash function. The main reason for the decrease of insertion throughput is that when the CF or the ECF is nearly full, it is more difficult to find an empty cell to insert the fingerprint.

ECF Query Throughput (Figure 11, 12): Our results show that ECF's query throughput before and after shrinkage keeps roughly unchanged, and query throughput before and after extension also keeps roughly unchanged. We shrink and extend ECF using CAIDA and the synthetic data set. The load ratio in these figures is the load ratio of ECF before shrinkage or extension. For the experiment on shrinkage in Figure 11, we first insert a number of items into an ECF, until its load ratio reaches a certain value, ranging from 0.30 to 0.50. Then, we query a number of items and compute the query throughput. Then we shrink ECF to half its original size and query items again. For the experiment on extension in Figure 12, we do similarly, and the load ratio before extension ranges from 0.75 to 0.95. The main reason for these results is that one query always probes two buckets, regardless of the load ratio, under both shrinkage and extension.

Number of Kicks (Figure 9): Our results show that the number of kicks of ECF and CF are almost the same, and it increases from nearly 0K to 350K as the load ratio increases from 0.70 to 0.98. We perform the experiments on CF and ECF with the same memory size on both two datasets, and record the number of kicks of the two data structures when varying the load ratio from 0.70 to 0.98. The main reason for the same kick numbers of the two data structures is that the

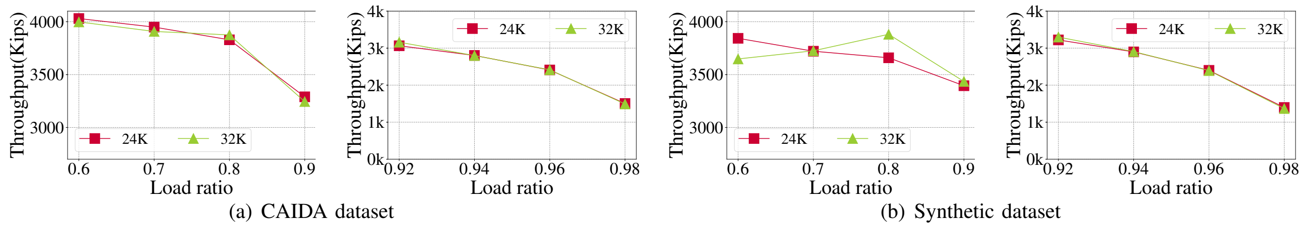


Fig. 7. Elastic Cuckoo filter's Insertion throughput of different load ratio on CAIDA and synthetic datasets.

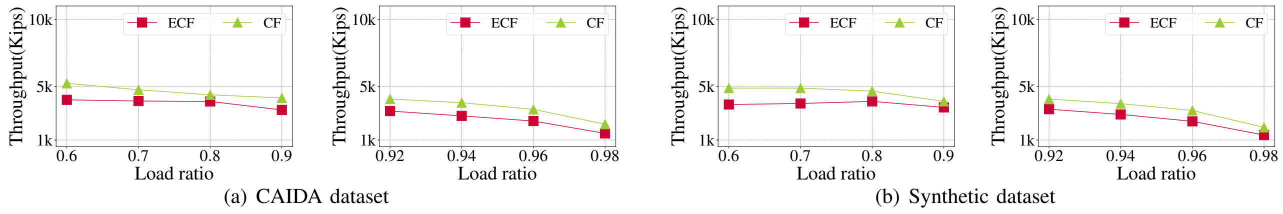


Fig. 8. Insertion throughput of different ratio of on CAIDA and synthetic datasets.

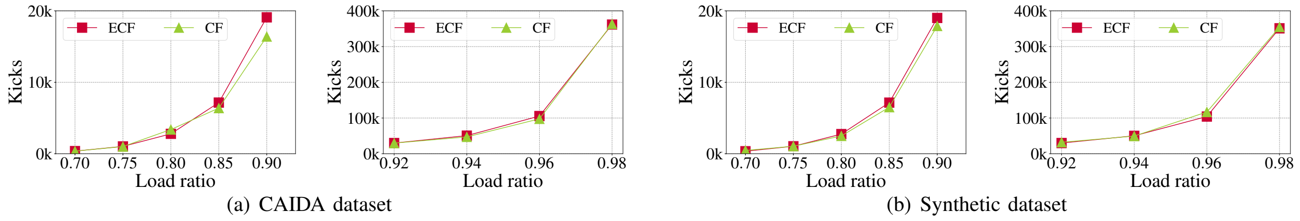


Fig. 9. Kicks of different load ratio on CAIDA and synthetic datasets.

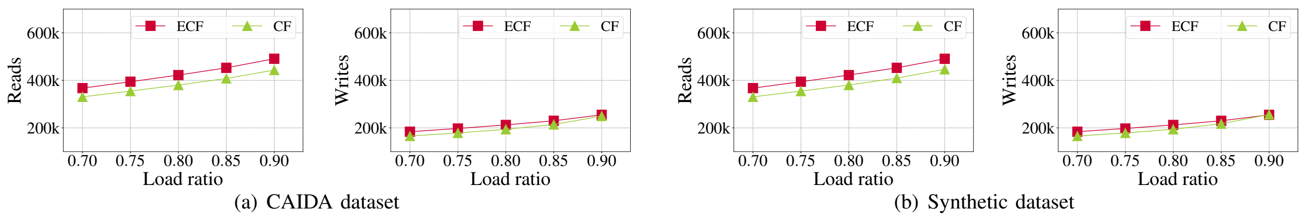


Fig. 10. Memory access number of different load ratio on CAIDA and synthetic datasets.

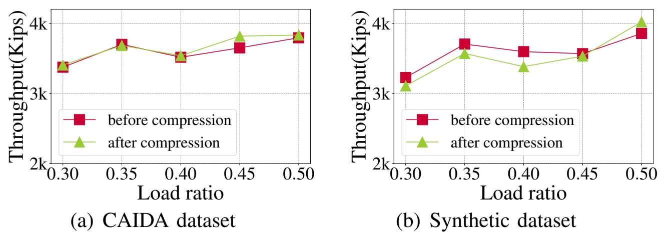


Fig. 11. Query throughput before and after shrinkage on CAIDA and synthetic datasets.

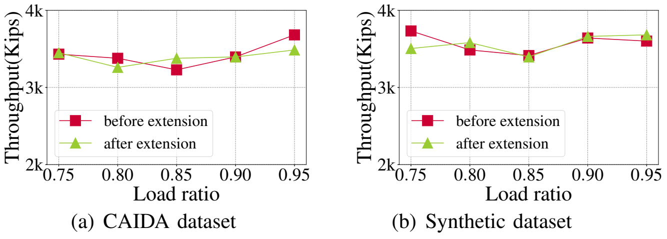
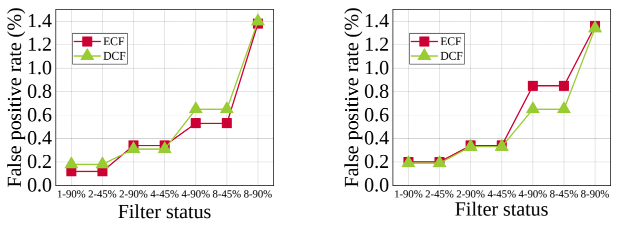


Fig. 12. Query throughput before and after extension on CAIDA and synthetic datasets.

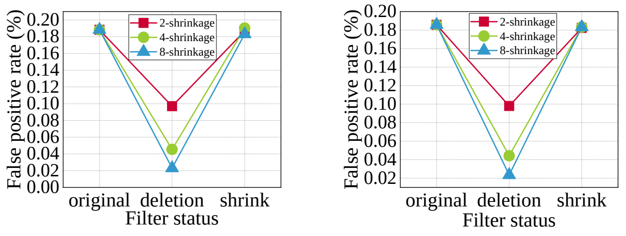
only difference between ECF and CF is the use of the offset. However, the use of the offset does not affect the kicks, so for a given load ratio, the kick number of CF and ECF are roughly the same. The main reason for the increase of kick numbers is that as the load ratio increases, there are fewer empty cells in CF or ECF, so on average more relocations are needed for each insertion.

Reads and Writes (Figure 10): Our results show that the number of reads of ECF is a little higher than for CF, while the number of writes is nearly the same. And the number of reads and writes for both ECF and CF increases as the load ratio increases from 0.70 to 0.90. We perform the experiments on CF and ECF for the same memory size on both datasets, and record the number of reads and writes of the two data



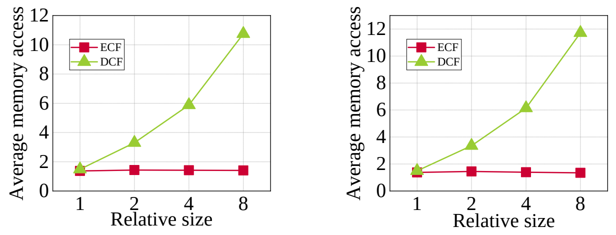
(a) CAIDA dataset (b) Synthetic dataset

Fig. 13. False positive rate of ECF and DCF.



(a) CAIDA dataset (b) Synthetic dataset

Fig. 15. The number of successfully inserted items.



(a) CAIDA dataset (b) Synthetic dataset

Fig. 14. Average memory access time for lookup.

structures for load ratios ranging from 0.70 to 0.90. The main reason for the small difference between ECF and CF is that the number of kicks of ECF can be a little higher than that of CF. And the reason for the increase of reads and writes is that as the load ratio increases, there are fewer empty cells, so more relocations are needed for each insertion, leading to more reads and writes.

False Positive Rate (Figure 13): Our experimental results show that our ECF achieves nearly the same false positive rate with DCF. In this experiment, we compare the false positive rate between ECF and DCF before and after extension. The horizontal axis indicates the relative size and load ratio of the filters. For example, 2-90% means the relative size is 2 and the load ratio reaches 90%, and 4-45% means that ECF doubles its size and two more CFs are created for DCF under the present circumstances. We first insert elements to both filters until the load ratio reaches 90%, then we evaluate the false positive rate of them, after that we extend them and evaluate again, finally we come back to insert new elements and perform these recursively. Though the real size of ECF doubled after extension, the virtual size of CF keeps in order to ensure correctness. Thus, the false positive rate of ECF will not decrease after the extension, and will increase with new elements inserting.

Query Performance (Figure 14): Our experimental results show that no matter how ECF extends, the average memory access time holds around 1.4. In this experiment, we compare the query performance between ECF and DCF before and after their extension. We regard visiting four cells of one single bucket as memory access once, and experiment on the average memory access time for querying elements. While for DCF, it grows above 10 when there are 8 CFs in a DCF. For each element, our ECF always finds two buckets, while DCF finds two buckets just for inserting but more buckets for looking up.

False Positive Rate after Shrinkage (Figure 15): We compare the false positive rate of ECF before and after shrinking. We first insert elements into the filter until the load ratio reaches 95% and evaluate the false positive rate. Then, we delete some elements from the filter until the filter can be shrunk successfully and the load ratio reaches 95% after shrinkage. In order to get a shrunk ECF whose size is half of the original size, we randomly delete half of the elements that are already inserted. Finally we evaluate the false positive rate before and after shrinkage. We study three different cases, and in each case the size of ECF after shrinkage is a half, one fourth or one eighth of the original size respectively. Figure 15 shows that when the load ratio is the same, the false positive rate of ECF before and after shrinkage remains the same.

VIII. Conclusion

The Cuckoo filter has created much interest, and has been applied to various fields. It supports deletions while achieving a similar false positive rate as the Bloom filter, for a given amount of memory. However, the Cuckoo filter is inflexible. Indeed, its size must be 2^n, where n is a positive integer, and its size cannot be dynamically tuned. In this paper, we propose a new variant of the Cuckoo filter called Elastic Cuckoo filter, which uses virtualization, shrinkage and extension, to overcome the shortcomings of the Cuckoo filter. We prove that our solution has a similar false positive rate and lower bound on insertion failures. In addition, experimental results show that our Elastic Cuckoo filter maintains nearly the same performance as the Cuckoo filter. At the same time, the Elastic Cuckoo filter is elastic: 1) it can be of any size; and 2) its size can be tuned through shrinkage and extension, without degrading its performance in basic operations.

References

[1] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proc. ACM SOSP*, pages 1–13, 2011.

[2] Amihai Motro. Accommodating imprecision in database systems: Issues and solutions. *ACM Sigmod Record*, 19(4):69–74, 1990.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

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

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

[6] Haoyu Song, Fang Hao, Murali Kodialam, and TV Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *IEEE INFOCOM 2009*, pages 2518–2526, 2009.

[7] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms for multicast communication groups. In *International Workshop on Distributed Algorithms*, pages 292–312, 1992.

[8] Ramin Modiri and Hossein Moiin. System and method for determining cluster membership in a heterogeneous distributed system, February 20 2001. US Patent 6,192,401.

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

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

[11] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121, 2007.

[12] Carlo Giacomo Prato. Route choice modeling: past, present and future research directions. *Journal of choice modelling*, 2(1):65–100, 2009.

[13] Shlomo Bekhor, Moshe E Ben-Akiva, and M Scott Ramming. Evaluation of choice set generation algorithms for route choice models. *Annals of Operations Research*, 144(1):235–247, 2006.

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

[15] Denis Charles and Kumar Chellapilla. Bloomier filters: A second look. In *Proc. European Symposium on Algorithms*, pages 259–270, 2008.

[16] Hanhua Chen, Hai Jin, Xucheng Luo, Yunhao Liu, Tao Gu, Kaiji Chen, and Lionel Ni. BloomCast: Efficient and effective full-text retrieval in unstructured P2P networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):232–241, 2011.

[17] Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[18] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended Bloom filter: An aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005.

[19] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E Taylor. Longest prefix matching using Bloom filters. In *Proc. ACM SIGCOMM*, pages 201–212, 2003.

[20] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proc. ACM CoNext*, pages 75–88, 2014.

[21] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. SmartCuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In *Proc. USENIX Annual Technical Conference*, pages 553–565, 2017.

[22] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. The dynamic Cuckoo filter. In *Proc. IEEE ICNP*, pages 1–10, 2017.

[23] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proc. ACM Conext 2013*.

[24] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with CuckooSwitch. In *Proc. ACM CoNext*, pages 97–108, 2013.

[25] https://github.com/efficient/cuckoofilter.

[26] Source codes of Elastic Cuckoo filter and related works. https://github.com/ElasticCuckooFilter/ElasticCuckooFilter.

[27] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. USENIX NSDI*, pages 371–384, 2013.

[28] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. In *Proc. European Symposium on Algorithms*, pages 1543–1561, 2008.

[29] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking SIMD vectorization for in-memory databases. In *Proc. ACM SIGMOD*, pages 1493–1508, 2015.

[30] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endowment*, 8(11):1226–1237, 2015.

[31] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent Cuckoo hashing. In *Proc. ACM EuroSys*, page 27, 2014.

[32] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. A cool and practical alternative to traditional hash tables. In *Proc. Workshop on Distributed Data and Structures*, 2006.

[33] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proc. ACM SIGMOD*, pages 323–336, 2018.

[34] Anshumali Shrivastava, Arnd Christian Konig, and Mikhail Bilenko. Time adaptive sketches (ada-sketches) for summarizing data streams. In *Proc. ACM SIGMOD*, pages 1417–1432, 2016.

[35] Jiecao Chen and Qin Zhang. Bias-aware sketches. *Proc. VLDB Endowment*, 10(9):961–972, 2017.

[36] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proc. ACM SIGMOD*, pages 1481–1496, 2016.

[37] Zengfeng Huang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. Efficient matrix sketching over distributed data. In *Proc. ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 347–359, 2017.

[38] Haida Zhang, Zengfeng Huang, Zhewei Wei, Wenjie Zhang, and Xuemin Lin. Tracking matrix approximation over distributed sliding windows. In *Proc. IEEE ICDE*, pages 833–844, 2017.

[39] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. SketchML: Accelerating distributed machine learning with data sketches. In *Proc. ACM SIGMOD*, pages 1269–1284, 2018.

[40] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. Sketching linear classifiers over data streams. In *Proc. ACM SIGMOD*, pages 757–772, 2018.

[41] Alex D Breslow and Nuwan S Jayasena. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.

[42] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proc. ACM SIGMOD*, pages 775–787, 2017.

[43] Yanqing Peng, Jinwei Guo, Feifei Li, Weining Qian, and Aoying Zhou. Persistent Bloom filter: Membership testing for the entire history. In *Proc. ACM SIGMOD*, pages 1037–1052, 2018.

[44] Technical report. https://github.com/ElasticCuckooFilter/ElasticCuckooFilter/blob/master/Technical%20Report.pdf.

[45] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

[46] The caida anonymized internet traces 2016. http://www.caida.org/data/overview/.

APPENDIX

A. Open Source Code Description

We provide the source codes of our Elastic Cuckoo filter at Github [26] without identity information, along with a detailed Readmd to enable others to reproduce our work. The open source codes include the code of implementation of our algorithms, along with the datasets and a technique report.

1) Datasets: Our CAIDA datasets is from the CAIDA Anonymized Internet Trace. We also provide the source codes for generating synthetic datasets, which is done by randomly generating unique numbers. All the data is 13 bytes long.

2) Implementation of Algorithms: We provide the source code of implementation of CF, ECF and DCF.

3) Experiments: We provide the source code of experiments for estimating the performance of our algorithm and comparing our algorithm with other algorithms. The evaluation metrics include: 1) the maximum number of items that can be inserted into filters; 2) insertion or query throughput; 3) query throughput after compression or extension; 4) load ratio; 5) kicks and memory access; and 6) false positive rate. Besides, users can modify the program to further evaluate the algorithm or write their own programs by including the header files.

B. Analysis of the Basic Version of the Shrinkage Algorithm

First, we show how to compute a fingerprint's two candidate buckets in the new ECF. Second, we prove that the shrinkage algorithm moves fingerprints to their candidate buckets in the new ECF. Third, we show how to compute a fingerprint's alternate bucket during the working of the KICK mechanism (in the new ECF).

First we explain how to compute a fingerprint's two candidate buckets in the new ECF. Based on the idea of *divide-by-2*, for item e, we divide s_0 by 2, divide d_1 by 2 and divide d_2 by 2, then we compute $i_1' = (s_0/2 + d_1/2)\%L'$ and $i_2' = (s_0/2 + d_2/2)\%L'$, which are item e's two candidate buckets in the new ECF. The detailed computation of the two candidate buckets of e in the new ECF is as follows:

$$\begin{aligned}
& i_1' = (s_0' + d_1')\%L' \\
& i_2' = (s_0' + d_2')\%L' \\
where\quad & \mathbb{F}_e = f(e)\%(2^p) \\
& s = [\varDelta(\mathbb{F}_e)\%L] \\
& s_0 = s - s\%2 \\
& s_0' = s_0/2 \\
& d_1' = d_1/2 = [h(e)\%(2^n)]/2 \\
& d_2' = d_2/2 = d_1' \oplus [(g(\mathbb{F}_e)\%(2^n))/2]
\end{aligned} \tag{15}$$

Second, we prove that the shrinkage algorithm moves fingerprints to their candidate buckets in the new ECF. s_0 is constrained to be even. If d_1 is even, then i_1 is even. From $i_1 = s_0 + d_1$, we have $i_1/2 = s_0/2 + d_1/2$, with no rounding down. If d_1 is odd, then i_1 is odd. From $i_1 = s_0 + d_1$, we have $i_1/2 = s_0/2 + d_1/2$, with rounding down in $i_1/2$ and $d_1/2$. Thus, $B[i_1/2]$ is a candidate bucket in the new ECF. So is $B[i_2/2]$.

Third, we show how to compute a fingerprint's alternate bucket during the working of the KICK mechanism (in the new ECF). For fingerprint $\mathbb{F}$ in $B[i]$, we compute its alternate bucket $B[alt]$ using the following equation.

$$\begin{aligned}
& alt = (s_0' + d')\%L' \\
where\quad & s = [\varDelta(\mathbb{F}_e)\%L] \\
& s_0 = s - s\%2 \\
& s_0' = s_0/2 \\
& d' = [(i - s_0' + L')\%L'] \oplus [(g(\mathbb{F})\%2^n)/2]
\end{aligned} \tag{16}$$

C. Analysis of the Extension Algorithm

We mainly show why our extension algorithm can guarantee fingerprints located in the correct positions. In our extension algorithm, the key technique is to copy the existing ECF. Before extension, an item e's two candidate buckets $B[i_1]$ and $B[i_2]$ are computed using Equation 2. Without loss of generality, we assume that e's fingerprint $\mathbb{F}_e$ is stored in $B[i_1]$. After copying, the α identical fingerprints of e are stored in buckets $B[(i + \beta L)\%L']$, where $0 \leqslant \beta < \alpha$. After extension, item e's two candidate buckets $B[i_1']$ and $B[i_2']$ are computed using Equation 5. From the computation of s_0 and s_0', we can find a positive integer γ such that $s_0' = (s_0 + \gamma L)\%L'$. Then we can find a positive integer ϵ such that

$$\begin{aligned}
i_1' &= (s_0' + d_1)\%L' \\
&= ((s_0 + \gamma L)\%L' + d_1)\%L' \\
&= ((s_0 + d_1)\%L' + \gamma L)\%L' \\
&= ((((s_0 + d_1)\%L) + \epsilon L)\%L' + \gamma L)\%L' \\
&= (i_1 + ((\epsilon + \gamma)\%\alpha)L)\%L'
\end{aligned} \tag{17}$$

We set β to be $(\epsilon+\gamma)\%\alpha$, then $B[(i+\beta L)\%L']$ is one of e's two candidate buckets in the new ECF, and the fingerprint $\mathbb{F}_e$ stored in this bucket is in its correct position. Thus, our extension algorithm maintains the fingerprint in the right position and deletes redundant fingerprints. After extension, when computing an item e's two candidate buckets, we use Equation 5. When computing the alternate bucket $B[alt]$ of a fingerprint $\mathbb{F}_e$ stored in $B[cur]$, we use the following equation.

$$\begin{aligned}
& alt = (s_0' + d)\%L' \\
where\quad & s' = offset(\mathbb{F}_e)\%L' \\
& s_0' = s' - s'\%2 \\
& d = ((cur - s_0' + 2^n)\%2^n) \oplus (g(\mathbb{F}_e)\%2^n)
\end{aligned} \tag{18}$$

Therefore, after extension, we can conduct insertions, queries and deletions as usual.

D. Pseudo-codes

In this section, we give the pseudo-codes of ECF's query and deletion algorithms. Recall that a query or deletion of ECF is to check an item's two candidate buckets without updating them.

Algorithm 3: Query

```
Input: A given item $e$
Output: Whether $e$ is in the Elastic Cuckoo filter
1 Compute $e$'s fingerprint $\mathbb{F}_e$;
2 Compute $e$'s offset $s_0$ by $s \leftarrow \mathit{\Delta}(\mathbb{F}_e)\%L$, $s_0 \leftarrow s - s\%2$;
3 Compute $e$'s two candidate buckets $B[i_1]$ and $B[i_2]$;
4 if $B[i_1]$ or $B[i_2]$ has a cell storing $\mathbb{F}_e$ then
5     Return true;
6 else
7     Return false;
```

Algorithm 4: Deletion

```
Input: A given item $e$
1 Compute $e$'s fingerprint $\mathbb{F}_e$;
2 Compute $e$'s offset $s_0$ by $s \leftarrow \mathit{\Delta}(\mathbb{F}_e)\%L$, $s_0 \leftarrow s - s\%2$;
3 Compute $e$'s two candidate buckets $B[i_1]$ and $B[i_2]$;
4 if $B[i_1]$ or $B[i_2]$ has a cell storing $\mathbb{F}_e$ then
5     Set one such cell to be empty;
6 Return;
```