Reviriego, P., Martinez, J., Larrabeiti, D., Pontarelli, S. (2020). Cuckoo Filters and Bloom Filters: Comparison and Application to Packet Classification. *IEEE Transactions on Network and Service Management*, 17(4), 2690–2701.

DOI: https://doi.org/10.1109/TNSM.2020.3024680

Cuckoo Filters and Bloom Filters: Comparison and Application to Packet Classification

Pedro Reviriego, Jorge Martínez, David Larrabeiti, and Salvatore Pontarelli

Abstract—**Bloom filters are used to perform approximate membership checking in a wide range of applications in both computing and networking, but the recently introduced cuckoo filter is also gaining popularity. Therefore, it is of interest to compare both filters and provide insights into their features so that designers can make an informed decision when implementing approximate membership checking in a given application. This paper first compares Bloom and cuckoo filters focusing on a packet classification application. The analysis identifies a shortcoming of cuckoo filters in terms of false positive rate when they do not operate close to full occupancy. Based on that observation, the paper also proposes the use of a configurable bucket to improve the scaling of the false positive rate of the cuckoo filter with occupancy.**

Index Terms—**Cuckoo filters, Bloom filters, Packet classification, SDN.**

I. INTRODUCTION

Approximate membership checking is widely used to speed up many computing and networking applications like DNA sequencing [1], caching [2] or network security [3]. For example, instead of performing a costly access to an external memory to search for an element, a small filter can be used first to check if the element is stored in that memory. Then, only when that is the case, the external memory is accessed [4]. In this scenario, it is important that the approximate check does not have false negatives as those would mean that elements that are stored in the external memory will not be found. Instead false positives would only cause an unneeded access to the external memory. Therefore, the structures used to implement the checking, commonly referred to as filters, are designed to avoid false negatives and achieve a low false positive probability.

The Bloom filter has traditionally been used to implement this checking. However, the cuckoo filter has been recently proposed as an alternative to Bloom filters [5] given its advantages, such as supporting deletions or achieving a lower false positive rate in some settings. Cuckoo filters have since been considered for several networking applications like traffic monitoring [6], longest prefix matching for IP packet forwarding [7] or security [8]. However, cuckoo filters have also a few drawbacks. For example, in a cuckoo filter, the number of elements that can be inserted is limited and, when the occupation is very high, the insertion may fail, as opposed to a Bloom filter where insertions cannot fail, they just degrade the false positive rate.

In this paper, Bloom and cuckoo filters are compared discussing their advantages and drawbacks and providing some insights that can be useful to designers. In particular, we look into cuckoo filter performance when it operates below the maximum occupancy as would be the case in many packet classification applications. The results show that, in practical configurations, differently from what previous analysis suggest, Bloom filters may perform better than cuckoo filters in terms of false positive rate. Based on that observation, an optimization of the cuckoo filter to improve the reduction of the false positive rate as the occupancy gets lower is proposed and evaluated. The proposed scheme extends the occupancy range for which the cuckoo filter outperforms the Bloom filter making it more competitive for practical configurations.

The contributions of this paper are threefold. The first is to present a detailed comparison of Bloom and cuckoo filters. The second is to make designers aware that cuckoo filters would in many cases have worse false positive rates than Bloom filters when they operate below their maximum occupancy. This is of practical interest as in many applications the filters would not work at full occupancy. The third contribution is to present the Configurable Bucket Cuckoo Filter (CBCF), a scheme that enables cuckoo filters to better scale their false positive rates with occupancy thus extending the range of occupancy for which the cuckoo filter outperforms the Bloom filter.

The rest of the paper is organized as follows. In section II, Bloom and cuckoo filters are described and compared.

P. Reviriego and D. Larrabeiti are with Universidad Carlos III de Madrid, Avenida de la Universidad 30, 28911 Leganés, Madrid, Spain. email: (revirieg,dlarra)@it.uc3m.es

J. Martínez is with Universidad Antonio de Nebrija, Calle Pirineos 55, 28040 Madrid, Spain. email: jmartine@nebrija.es

S. Pontarelli is with the Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Via del Politecnico 1, 00133 Rome, Italy. email: salvatore.pontarelli@uniroma2.it

Then in section III the use of filters for packet classification is discussed as our target application. The proposed cuckoo filter optimization is presented and evaluated in section IV. The paper ends with the conclusions in section V.

II. BLOOM FILTERS AND CUCKOO FILTERS

Bloom filters and cuckoo filters have become popular data structures to perform approximate membership checking. In the following both Bloom and cuckoo filters are briefly described and compared with respect to different parameters. Then the scaling of the false positive rate versus occupancy for both Bloom and cuckoo filters is discussed to show that the Bloom filter can outperform the cuckoo filter for practical settings.

A. Bloom Filters

A Bloom filter uses an array of m bits, initially set to zero, on which elements are inserted or checked using a set of k hash functions $h_1(x), h_2(x), ..., h_k(x)$ [4]. To insert an element x, the bit positions given by $h_1(x), h_2(x), ..., h_k(x)$ are set to '1'. Conversely, to check if an element has been inserted in the filter, those positions are read and if and only if all of them are '1' a positive is returned. The Bloom filter will always return a positive for an element that has been inserted. Instead, when checking for an element that has not been inserted, the filter will in most cases return a negative, but with low probability, a positive can be obtained. This occurs when the k positions have been set to '1' due to the insertion of other elements. The probability or rate of a false positive on a check for an element not stored in the filter can be approximated by $(p_1)^k$, where p_1 is the probability that a bit is set to one which is a function of the number of elements inserted. This probability can be estimated when n elements have been inserted on the filter as:

$$p_1 \simeq 1 - (1 - \frac{1}{m})^{k \cdot n}. \tag{1}$$

and thus the false positive rate:

$$fpr_{BF} \simeq \; 1 - (1 - \frac{1}{m})^{k \cdot n})^k. \tag{2}$$

A Bloom filter is shown in Figure 1 with several elements inserted on the filter.

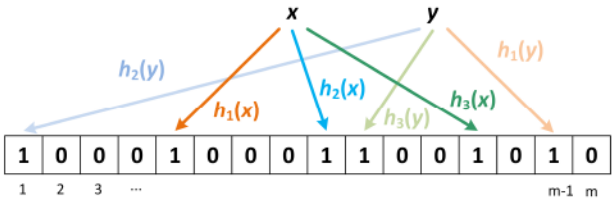

Fig. 1. Illustration of a Bloom filter checking membership of elements x and y

For a given filter size m and elements inserted n, the number of hash functions k that minimizes the false positive rate is given by $k_{opt} = \frac{m}{n} \cdot log(2)$ as stated in [9]. Therefore the optimal number of hash functions increases with the number of memory bits per element stored $\frac{m}{n}$. However, from a practical standpoint, using a large value of k has disadvantages as up to k memory accesses are needed to complete a lookup and the same applies to the number of hash functions. Therefore, increasing k increases the filter complexity making large values not practical in many settings. Additionally, the benefits in terms of false positive rate reduction are smaller as k approaches k_{opt} and thus lower values can be used with a small impact on the false positive rate.

A limitation of Bloom filters is that elements cannot be removed since a given position may have been set to one by more than one element. Therefore, setting it to zero when removing an element could lead to false negatives. The filter can be extended to be a Counting Bloom Filter (CBF) with a counter per position to support deletions [9] but that increases the memory requirements significantly. For example, four bit counters have been shown to achieve a low probability of counter saturation but using them implies a 4x increase on the memory used. This can be mitigated by storing the counters in the slower memory and keeping a single bit in the faster memory as counters are only needed for insertions and removals but not for lookups. On the other hand, there is no limitation on the number of elements that can be inserted on a Bloom filter, but as more elements are inserted, p_1 will increase leading to more false positives.

The work to improve Bloom filters has continued over the years and many variants have been proposed to optimize different aspects [4]. In particular, the reduction of the false positive rate combined with the support of deletions has been widely studied. For example, the Deletable Bloom filter (DlBF) [10] and the Ternary Bloom filter (TBF) [11] add some additional information to the filter without using a counter per position and support some deletions but not all. In more detail, the DlBF divides the filter in regions and keeps track of the regions on which there have been no collisions among the inserted elements. When removing an element, the bits that are in regions with no collisions can be safely cleared to zero. Instead bits that are in regions with collisions cannot be removed. The TBF uses three values for each position on the array 0,1 and where the indicates a collision. When removing an element, only the positions that store a one can be cleared to zero. From the discussion, it becomes apparent that both, the DlBF and the TBF only partially support deletion.

A different approach is taken in the Variable Increment Counting Bloom Filter (VI-CBF) [12] that uses a counter per position in the filter but with variable increments that are computed using a hash function on the inserted elements. By carefully selecting those increments, this enables a reduction in the false positives when only one element is inserted on a position if the increment of the searched element does not match the value stored in the filter. This scheme has been recently extended in the Tandem Counting Bloom filter that groups counters in pairs to further reduced false positives [13].

B. Cuckoo Filters

The cuckoo filter (CF) [5] uses partial key cuckoo hashing to implement approximate membership checking [14]. In more detail, for each element , a fingerprint is computed using a hash function and it is stored in the filter instead of . The filter is formed by an array of buckets formed by cells each of which can store a fingerprint. To achieve a good trade-off between false positive rate and the maximum achievable occupancy, is commonly used [5]. The fingerprint for can be stored in two buckets given respectively by and where again, and are hash functions. The cuckoo filter is illustrated in Figure 2.

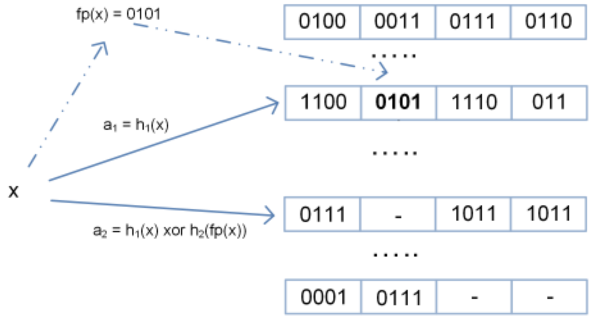

Fig. 2. Illustration of a cuckoo filter that is checking membership of element

To check if element is stored in the filter, we compute , of and check if is stored in any of those buckets. In that case a positive is returned, if not the result is negative. A lookup for an element not stored in the filter can return a positive if another element is stored in either of those buckets and . On the other hand, if an element has been inserted in the filter, a lookup will always return a positive. Therefore, both cuckoo filters and Bloom filters can have false positives but not false negatives. For a filter that uses fingerprints of bits, the false positive rate can be approximated by — where is the filter occupancy, defined as —, where is the number of elements stored in the filter. The false positive rate of Bloom and cuckoo filters has been compared when is close to one showing that the cuckoo filter provides a lower false positive rate when is larger than eight [5]. This is one of the key advantages of cuckoo filters.

To insert an element , buckets , are accessed and if there is an empty cell, the fingerprint is stored there. If there are no empty cells, one of the elements stored in those buckets is removed from the filter and the fingerprint of is stored in that cell. Then, is inserted on its alternate bucket (if it was stored in and the other way around). This alternate bucket can be computed by doing the of with the position on which the fingerprint of was stored. As in cuckoo hashing, insertion is the most complex procedure and may require several iterations until an empty cell is found.

Unlike Bloom filters, cuckoo filters support deletion. To remove an element, the corresponding buckets and are accessed and the matching fingerprint is removed. It should be noted that, unlike Bloom Filters, in this structure there is a one-to-one relationship between stored fingerprint entries and elements and the same fingerprint might appear several times in the same slot. On the other hand, since to insert an element in a cuckoo filter, an empty cell has to be found, the number of elements that can be inserted in a cuckoo filter is limited and can be in the best case the number of cells of the filter. In practice, insertion fails before reaching full occupancy but the filter can operate up to approximately 95% occupancy [5]. However, if the filter operates at close-to-full occupancy, there will be no room for further insertions during system operation. This is an important limitation as in many applications, the number of entries stored in the filter is dynamic and not known in advance. In other applications, even if the number of entries stored in the filter is static, their number may vary from one device to another and thus the filter has to be sized for the worst case. In both scenarios, the filter would operate in most cases below its maximum occupancy to support additional insertions or because the configuration of the device is not worst case. In both scenarios, the false positive rate of the cuckoo filter may be larger than that of a Bloom filter as it will be seen in the evaluation results presented in subsection II.D.

TABLE I
COMPARISON OF BLOOM, COUNTING BLOOM (CBF), TERNARY BLOOM (TBF), VARIABLE INCREMENT BLOOM (VI-CBF) AND CUCKOO FILTERS

Filter	Deletion support	Deletion #accesses	Positive lookup #accesses	Negative lookup #accesses	Insertion #accesses	FPR for optimal settings	FPR scaling with occupancy	External memory
Bloom	No	N.A.				Better for small BPE	Good	No
CBF	Yes					Same as Bloom	Good	Counters
TBF	Partial					Always worse	Good	No
VI-CBF	Yes					Always worse	Good	No
Cuckoo	Yes				Variable	Better for large BPE	Poor	No

C. Comparison

Table I compares Bloom, counting Bloom, ternary Bloom, variable increment counting Bloom and cuckoo filters for the more relevant parameters. The TBF has been included in the comparison as an example of filter that partially supports deletions and the VI-CBF as an example of filter that supports deletions using more sophisticated processing. The goal is to illustrate that those types of filters will not be competitive in the scenario considered. In more detail, the comparison is done considering that the filter is stored in a small on-chip memory and that the full set is stored in a larger off-chip memory. It is assumed that all the filters are given the same amount of on-chip memory and their performance is compared. Therefore minimizing the amount of information stored on-chip by the filter becomes critical.

The first parameter is the support of deletions which is one of the advantages of cuckoo filters. To support deletions a Bloom filter needs to have counters instead of bits and that leads to a large increase in the memory needed for a counting Bloom filter. However, the counters can be stored in the off-chip memory keeping only one bit per position of the array in on-chip memory. This means that the counting Bloom filter uses the same amount of on-chip memory and thus has the same performance as the Bloom filter except from the need to store the counters in external memory. The ternary Bloom filter (TBF) only supports deletions partially as if a given position stores an   value it can not be set to zero when removing an element. Instead the variable increment counting Bloom filter (VI-CBF) fully supports deletions. However differently from the counting Bloom filter, it needs to store the counter value in the on-chip memory. The same applies to the TBF. Therefore, both Bloom filter variants would have a smaller array when using the same amount of on-chip memory as a Bloom filter. This will degrade the false positive rate as shown in the simulation results presented in the following.

The next four columns provide an indication of the time that would be needed for each of the operations. This would be implementation dependent and for some platforms it may depend heavily on the optimizations made to implement the filters [15]. To give a general indication of performance, the metric used is the number of memory accesses that in many cases provides a reasonable indication of the time and cost of an operation. It can be seen that for the Bloom filter, all operations are bounded by , which as discussed before, normally takes a small value to minimize the impact on performance. For the cuckoo filter, lookups and deletions are always completed in at most two memory and three memory accesses respectively. Instead, for insertions the number of accesses is not bounded and can be very large when the filter operates at high occupancy which is one of the disadvantages of cuckoo filters.

In many applications, lookups are by far the most common operation. Looking closer into the time required for them, it can be seen that for positive lookups the cuckoo filter would outperform the Bloom filter as   is at least two. Instead for negatives, Bloom filters have an advantage as a lookup ends as soon as one position read has a value of zero. Therefore, the average number of accesses for a Bloom filter, would depend on its occupancy. Instead, for the cuckoo filter, two accesses are always needed for a negative lookup. Since the goal of the filter is to avoid checking the external memory on a negative, in most cases the filter is used when the fraction of negative lookups is dominant and thus the performance of negative lookups is the one that contributes most to the performance of the filter.

The following parameter included in the comparison is the false positive rate (FPR) that would determine the effectiveness of the filter in avoiding accessing the full table for elements that would not find a match. Two scenarios are considered for the false positive rate. The first one is when the number of elements and the amount of memory is known in advance and the filter parameters can be optimized (for the Bloom filter and the fingerprint size and number of buckets for the cuckoo filter). This ideal case is not realistic in many applications. To account for this, the scaling of the false positive rate with occupancy is also compared. The cuckoo filter outperforms the Bloom filter in an optimum configuration when the number of bits per element (BPE) is larger than approximately eight while the Bloom filter is better for fewer bits [5]. Therefore, the cuckoo filter provides better performance when the BPE is large as seen in Table I. The counting Bloom filter has the same FPR as the Bloom filter as it is assumed that counters are stored off-chip as discussed before. Instead, both the TBF and the CBF will have significantly worse FPR as they need more on-chip memory per filter position and thus when using the same amount of on-chip memory have a smaller array. The scaling of the false positive rate with occupancy is better for the Bloom filter in all its variants than for the cuckoo filter making it better in some practical configurations. This can be clearly seen by comparing the false positive rates of both filters and noting the dependency with the number of elements n is exponential for the Bloom filter and linear for the cuckoo filter:

$$\begin{aligned} fpr_{BF} &\simeq \ 1-(1-\tfrac{1}{m})^{k \cdot n})^{k} \\ fpr_{CF} &\simeq \tfrac{8 \cdot o}{2^{f}} = \tfrac{8 \cdot n}{c \cdot m \cdot 2^{f}} \end{aligned} \tag{3}$$

Finally, the last column shows the need to store part of the filter information on the external memory. This is only needed for the counting Bloom filter to store the counters. The rest of the filters considered have all their information in on-chip memory.

D. False positive rate versus occupancy

To illustrate the dependency of the false positive rate on occupancy of the cuckoo filter and the Bloom filters considered, they have been implemented and the false positive rate has been measured for several configurations. In more detail, a cuckoo filter with $m = 8192$ buckets of four cells was simulated with fingerprints of $f = 12, 15, 18$ bits. The reasoning behind this choice of parameters is as follows. The number of cells per bucket has an impact on both the false positive probability and on the maximum occupancy that can be achieved. In more detail, each element is compared to $2 \cdot c$ fingerprints and thus increasing c increases linearly the false positive probability making lower values more attractive. On the other hand, larger values of c result in larger occupancy before an insertion fails [16]. In most cuckoo filter implementations c is set to four to achieve a balance between the two conflicting requirements. As for the fingerprint bits f, the plain cuckoo filter starts to outperform the Bloom filter in terms of false positive probability when the number of bits per element is larger than approximately eight [5]. Therefore, the values selected are in that range (above eight) and cover different false positive probabilities from 0.2% (f = 12) to 0.003% (f = 18). Finally, the value of m should have no impact on the results as long as it is much larger than one. Then, a Bloom filter, a Ternary Bloom filter and a Variable Increment Counting Bloom filter of the same size ($8192 \cdot 4 \cdot f$ bits) were also simulated. For the Bloom filter the optimal value of k for an occupancy of 95% was used. In particular, the values of k used were $9, 11, 13$ for $f = 12, 15, 18$. For the VI-CBF, the value of $L = 4$ was used using counters of five bits and the number of hash functions was set to $k = 4$. The cuckoo filter was constructed and elements were inserted until the desired occupancy was reached. Elements are generated randomly but ensuring that the are no duplicates. Note that this should have no impact on the results when using well behaved hash functions. Then a number of element replacements (removal of an element from the filter and insertion of a new one) were done to simulate the steady state operation at that occupancy. Finally, one million lookups for elements not stored in the filter were done and the false positive rate was measured. For each configuration, the process was repeated one thousand times and the average false positive rate across all runs is reported. The same process was done for the Bloom filters considered except for the replacement operations.

The results are shown in Figure 3. The first observation that becomes apparent is that the TBF and VI-CBF have a much larger false positive rate than the Bloom filter. This is because both the TBF and the VI-CBF require to store additional information on-chip to support removals. This reduces the size of the filter array and increases the probability of false positives. Instead, in the case of the counting Bloom filter, since its counters are stored off-chip, it stores the same information as a Bloom filter on-chip and thus has the same false positive rate. This illustrates, how in our target applications, for a fixed on-chip memory size, the traditional Bloom filter (or the counting Bloom filter with the counters stored off-chip) achieves the lowest false positive rate. Therefore, in the rest of the paper, the comparison concentrates on the Bloom filter versus the cuckoo filter.

As expected, it can be seen that the false positive rate of the cuckoo filter reduces linearly as the occupancy decreases while for the Bloom filter, the reduction is steeper. This means that the Bloom filter outperforms the cuckoo filter in terms of false positive rate when the occupancy is approximately 90%,85%,80% or lower when $f = 12, 15, 18$ respectively. As discussed before, in many applications, the filter will not operate at maximum occupancy and for those, the Bloom filter may provide better performance.

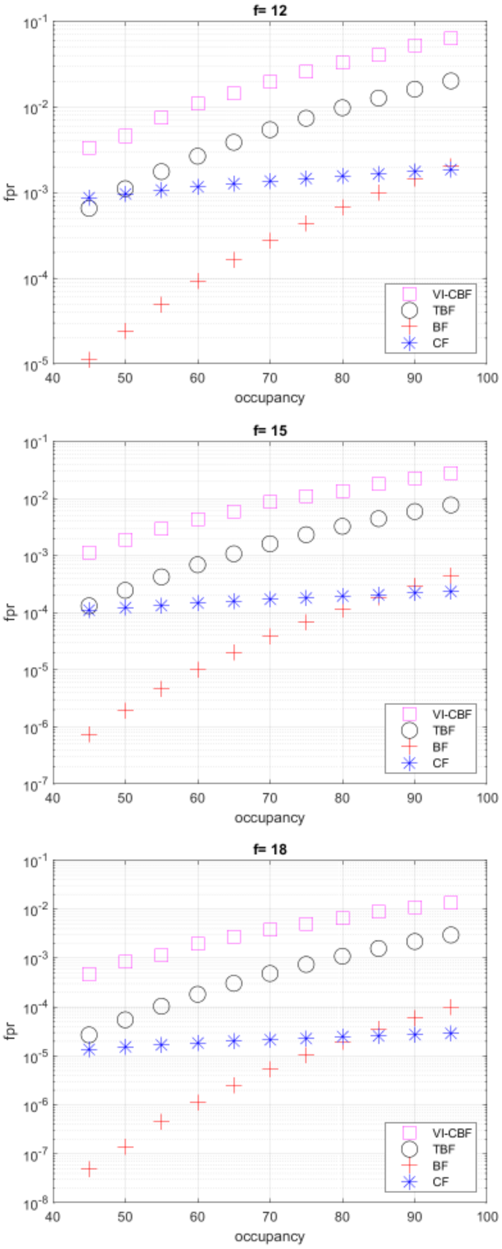

Fig. 3. False Positive Rate of the VI-CBF, TBF, Bloom filter and cuckoo filter for different fingerprint sizes and occupancy

III. FILTERING FOR PACKET CLASSIFICATION

Packet classification is a key functionality in modern networks and is needed for example to apply security policies, to implement quality of service or to process packets in switches, routers, firewalls, bandwidth managers, intrusion detection systems, etc [17]. With the adoption of Software Defined Networking (SDN), packet classification will become more generic and complex [18]. To classify packets, their headers are parsed to extract the relevant fields that are then compared with a set of rules to decide the actions to apply to the packet. Implementing this checking at current wire speeds is challenging as in some cases, for example, the set of rules is large and has to be stored in a external memory that provides limited bandwidth. To alleviate this issue, in many cases, smaller filters are stored in a faster memory to reduce the number of accesses to the full tables stored in a slower memory [19].

In most packet classification functions, a group of packet header fields are extracted from each incoming packet to build a key that is checked against a set of rules to determine the action that must be applied to the packet. The key can be as simple as the destination IP address for packet forwarding to 15-field rules considered in some software defined networking applications [18]. The set of rules to check against may require a significant amount of memory, for example when it is large as for example in Internet scale routing tables that have close to one million entries or because each rule requires hundreds of bits, as in many field rules. In many networking systems, it is common to have a small amount of on-chip SRAM and a large amount of off-chip DRAM [20]. For example, external DRAM can be used to store counters [21] or large lookup tables [22]. Switching ASICs that have internal custom logic and SRAM combined with external DRAM are commercially available and are widely used [23].

Therefore, when the rule set is large it does not fit into the on-chip memory and has to be placed on the external memory. The time needed to access the DRAM is orders of magnitude larger than that of accessing the internal SRAM. This means that accessing the full rules introduces a larger latency and consumes a significant amount of memory bandwidth leading to a performance bottleneck. On the other hand, the DRAM is very large so it can store very large tables while the SRAM is small. Therefore, the critical resource is the on-chip SRAM while the off-chip memory is abundant.

In many cases, all the effort to access the external memory to check the rule set is done to find that the search does not match any rule. This is the case for example in Longest Prefix Matching (LPM), where a search is done for several prefix lengths and most of them will not find a match [7], or in more general packet classification with Tuple Space Search [19]. In those cases, it is interesting to perform an initial filtering that gives us an indication of whether it makes sense to access the full table or not. To that end, traditionally Bloom filters have been used as they can have false positives but not false negatives such that if there is a matching rule we would always find it [9]. A Bloom filter can eliminate more than 95% of the accesses to external memory for lookups that will not match any rule using only eight bits per rule which in many cases makes it possible to store the filter in a faster on-chip memory.

Another example where filtering can be useful is the black listing of malicious IP addresses [24]. To that end, the source IP of each packet is checked against a list and on a match, the packet is discarded. For IPv6, the amount of memory needed to store the table is large and so is the bandwidth to check the full address. Therefore, as in most cases the packet will not come from a blacklisted address, it can be beneficial to first check a filter, so that on a negative, we can safely accept the packet and only on a positive we need to access the full table. This is illustrated in Figure 4.

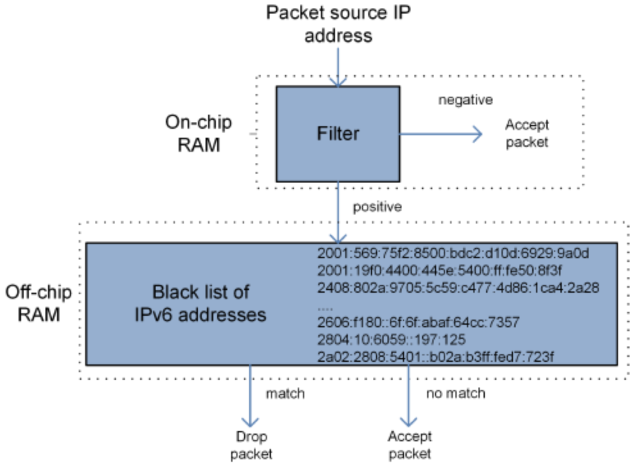
Packet source IP address
On-chip RAM
Filter
negative
Accept packet
positive
Off-chip RAM
Black list of IPv6 addresses
2001:569:75f2:8500:bdc2:d10d:6929:9a0d
2001:19f0:4400:445e:5400:ff:fe50:8f3f
2408:802a:9705:5c59:c477:4d86:1ca4:2a28
....
2606:f180::6f:6f:abaf:64cc:7357
2804:10:6059::197:125
2a02:2808:5401::b02a:b3ff:fed7:723f
match
no match
Drop packet
Accept packet

Fig. 4. Speeding up IP address blacklisting with filtering

An important feature of packet classification applications is that insertions or deletions in the rule sets are orders of magnitude less frequent than lookups. As an example, the peak number of BGP updates in a router may be at most in the order of thousands per second with much lower values on average [25], [26]. This compares with the hundreds of millions of packets that can be processed per second in a high speed router. The same reasoning applies to other network functions such as MAC learning or OpenFlow tables that have similar update rates. Therefore, for packet classification, the performance is dominated by the lookup speed, being the impact of insertions and deletions negligible in most cases.

Finally, it is important to note that in filtering for packet classification, since the goal is to accelerate the checking of the rule set, it can be assumed that the full rule set is stored in the external memory. Otherwise, on a positive on the filter, there would be no way to check the rule set to determine if there is some matching rule which is the goal of the entire classification process. Furthermore, the rule set can be used to optimize the filter. An example of this is the adaptive cuckoo filter recently proposed to remove false positives once they occur in order to prevent subsequent packets from causing further false positives [27].

IV. MAKING CUCKOO FILTERS BETTER THAN BLOOM FILTERS AT PRACTICAL OCCUPANCY

Given the advantages of CFs over BFs such as element deletion support and lower FPR at high occupancy, we try to address the aspects of CFs where BFs are better. In particular, in this section, a novel Configurable Bucket Cuckoo Filter (CBCF) is introduced that reduces the false positive rate of cuckoo filters when occupancy is below maximum. Then we evaluate this data structure to prove its effectiveness. The section ends with a brief discussion of the additional logic needed to implement the proposed scheme compared to that of the cuckoo filter in order to show that it would be acceptable in many implementations.

A. Description of the CBCF

To achieve a better behaviour of the false positive rate of cuckoo filters at moderate occupancy levels, we propose a configurable bucket that adapts the size of the fingerprints stored to reduce the false positive rate when the number of elements stored in a bucket is fewer than its capacity. A selection bit is used to configure the bucket either with four cells or with three. Then, when a given bucket stores fewer than four cells, we can use the second configuration with three larger cells so that fingerprints have – bits thus reducing the contribution of this bucket to the false positive rate by a factor of $\frac{}{3}$. This bucket is illustrated in Figure 5.

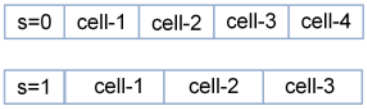

Fig. 5. Configurable bucket in the proposed CBCF

On a lookup, buckets , (note that is computed using always the smallest fingerprint) are accessed and for each of them, first the selection bit is read to determine the length of the fingerprints. Then the fingerprint of the searched element is computed to the required length and the comparisons are made. The complete procedure is shown in Algorithm 1,

A similar procedure can be used for removals but removing the fingerprint once it is found. However, if when removing an element the bucket had four fingerprints, we should reconfigure the bucket to store longer

Algorithm 1 Query for element in a CBCF

```
Compute for
Access buckets
for each cell in do
    Read bit
    if cell is used and then
        Compare with fingerprint in the cell
        If equal return positive
    end if
    if cell is used and then
        Compare with fingerprint in the cell
        If equal return positive
    end if
end for
return negative
```

Algorithm 2 Removal of element in a CBCF

```
Compute for
Access buckets
for each cell in do
    Read bit
    if cell is used and then
        Compare with fingerprint in the cell
        if equal then
            Access full table on that bucket and cell
            if element stored is then
                Remove element and fingerprint
                return success
            end if
        end if
    end if
    if cell is used and then
        Compare with fingerprint in the cell
        if equal then
            Access full table on that bucket and cell
            if element stored is then
                Remove element and fingerprint
                Update fingerprints to
                Set
                return success
            end if
        end if
    end if
end for
return failure
```

fingerprints for the remaining three elements. To be able to do so, we need the full elements so that we can compute the longer fingerprints. Therefore, the CBCF can only be used in applications where full elements are also stored, in a larger slower memory. This is the case when the filter is used to reduce the cost of accessing the full elements and is also needed for other cuckoo filter enhancements such as the adaptive cuckoo filter [27]. A practical configuration could have a replica of the filter in the larger memory that stores the full elements instead of the fingerprints so that there is a one-to-one correspondence between fingerprints and elements. This is illustrated in Figure 6 and assumed for the rest of the paper. In this configuration, when removing an element, if there are several matching fingerprints, the full-element table is accessed to locate the element that is to be removed and avoid removing another element that has the same fingerprint as that would lead to an inconsistent state. Once the element is located, both the element and its fingerprint are removed to preserve the one-to-one correspondence between fingerprints in the CBCF and elements in the main table. The procedure to remove an element is shown in Algorithm 2. It can be seen that when the bucket stored short fingerprints, we also update the fingerprints to long as now there is an empty cell in the bucket.

To insert an element, buckets , are accessed and priority is given to the bucket that stores fewer fingerprints. Then, the short or long fingerprint is stored in an empty cell depending on whether the bucket is full or not. If there are no empty cells, an element is moved as in the original cuckoo filter. The same operations should be done in the table that stores the full elements. The procedure to insert an element is shown in Algorithm 3.

The reduction that can be achieved on the false positive rate will depend on the fraction of buckets that is not full and thus can benefit from using larger fingerprints. This will obviously depend on the filter occupancy and will in the best case be $\frac{}{3}$. To maximize the reduction, a scrubbing procedure has been implemented. This procedure sequentially reads all the buckets in the filter and for those that are full, removes one of the elements randomly and tries to insert it on its other bucket moving elements until a bucket that has fewer than cells is found (so that after inserting the element is still not full) or twenty movements have been made. In the second case, after the twenty movements, the procedure inserts the element in any bucket that has empty cells and stops. This scrubbing reduces the number of buckets that are full and makes the CBCF more effective. The scrubbing procedure can be run after a given number of insertions or periodically to ensure that the filter has as few full buckets as possible.

The proposed scheme could be extended by using for example two selection bits so that different fingerprint lengths can be used when a bucket has four, three, two

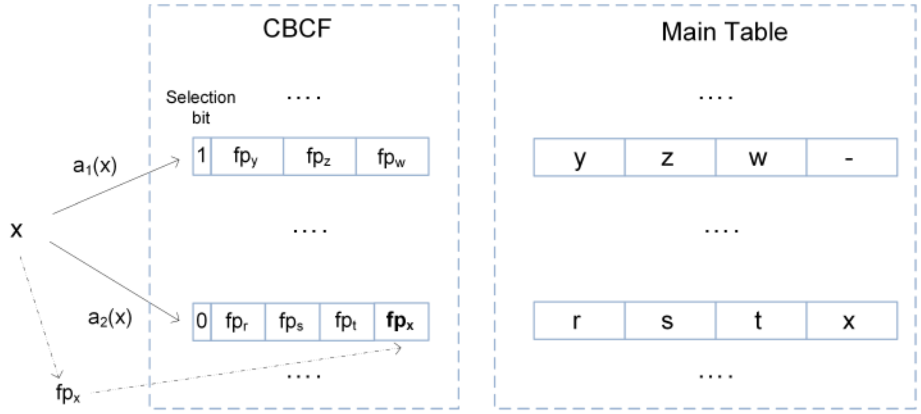

Fig. 6. Diagram of a CBCF on which the full elements are stored in a cuckoo table with one-to-one correspondence to the CBCF

```
Algorithm 3 Insertion of element   in a CBCF
Compute                    for
Access buckets
if there are empty cells on buckets        then
    Select bucket with more empty cells
    if one cell empty then
        Store
    else
        Store
    end if
    Store    on main table
    return success
end if

for          to          do
    Remove an element        from
    Store      in its place
    Compute                    for
    Access buckets
    if there are empty cells on buckets        then
        Select bucket with more empty cells
        if one cell empty then
            Store
        else
            Store
        end if
        Store        on main table
        return success
    end if

end for
return failure
```

or one elements. However, the largest benefit of this extension would occur when most of the buckets have fewer than three elements which means that occupancy would be quite low. This seems to have less interest as even if the filter has to have some margin to its maximum occupancy, it is not optimal to operate at low occupancy. In any case, the study of such an extension is left for future work.

B. *Evaluation*

As discussed in the comparison presented in section II.D, for our target applications on which the on-chip memory is the scarce resource, the TBF and VI-CBF have worse false positive rate than the traditional Bloom filter. Therefore, in the following, the proposed CBCF is compared only with the Bloom filter as it is the one that achieves the lowest false positive rate among the different Bloom filter variants.

To compare the cuckoo filter, the Bloom filter and the proposed CBCF, they have been implemented and the false positive rate has been measured for several configurations[1]. In more detail, a cuckoo filter and CBCF with buckets of four cells were simulated with fingerprints of bits for which the cuckoo filter is expected to outperform the Bloom filter in terms of false positive rate [5]. Then, a Bloom filter of the same size was also simulated using the optimal value of for an occupancy of 95%. In particular, the values of used were for . The cuckoo filter and CBCF were constructed and elements were inserted until the desired occupancy was reached. Then a number of element replacements (removal of an element from the filter and insertion of a new one) were done to simulate the steady state operation at that

---

[1] The source code used for the CBCF and CF is available in this link https://github.com/mladron/CBCF

occupancy. For the CBCF, the scrubbing operation is then executed to try to minimize the number of buckets that are full. Finally, one million lookups for elements not stored in the filter were done and the false positive rate was measured. For each configuration, the process was repeated one thousand times and the average false positive rate across all runs is reported. The same process was done for the Bloom filter except for the replacement operations.

The results are shown in Figure 7. Comparing the cuckoo filter and the Bloom filter, it can be seen that the false positive rate of the cuckoo filter reduces linearly as the occupancy decreases whereas for the Bloom filter, the reduction is steeper. This means that the Bloom filter outperforms the cuckoo filter in terms of false positive rate when the occupancy is approximately 90%,85%,80% or lower for the fingerprint sizes considered. As discussed in the introduction, in many applications, the filter will not operate at maximum occupancy and for those, the Bloom filter may provide better performance.

Looking at the CBCF, it can be seen that it is indeed able to provide a larger reduction of the false positive rate as occupancy reduces. In more detail, with the CBCF the Bloom filter outperforms the cuckoo filter in terms of false positive rate when the occupancy is below 60%,55%,55% compared to 90%,85%,80% for the original cuckoo filter. This clearly shows that the CBCF is able to extend the range of occupancy for which the cuckoo filter outperforms the Bloom filter in terms of false positive rate. This is achieved with no impact on the false positive rate at maximum occupancy, which in fact is slightly reduced as some buckets will not be full at 95% occupancy.

A closer look at the false positive rate of the CBCF reveals that the reduction compared to a CF ends approximately at 70% occupancy. This means that at this point there are almost no full buckets. Interestingly this corresponds to the occupancy of a cuckoo filter that has which is around 90% [16] which multiplied by gives approximately the 70% observed. This would be the maximum occupancy that could be achieved with no full buckets. Therefore, the simulation results seem to be consistent with the theoretical analysis.

To compare the performance of the proposed CBCF with the CF and BF, the average number memory accesses per negative lookup has also been logged during the simulations. As discussed before, in the scenario considered, most of the accesses would be negative lookups for which the filter avoids checking the external memory. Therefore, the speed of negative lookups will be the dominant factor for the speed of the filter. The results are shown in Figure 8. It can be seen that in the case of both the CF and the proposed CBCF, the number of memory accesses is constant and equal to two as expected from Table I. This is because to return a negative, the two buckets to which an element maps have to be checked. Instead, for the Bloom filter, the number of accesses depends on the occupancy. This is again expected as for lower occupancy, the probability of finding a zero increases and the Bloom filter returns a negative when the first zero is found. The average number of memory accesses for the Bloom filter is also two for the maximum occupancy. This is also expected as the optimal for that occupancy was used in the

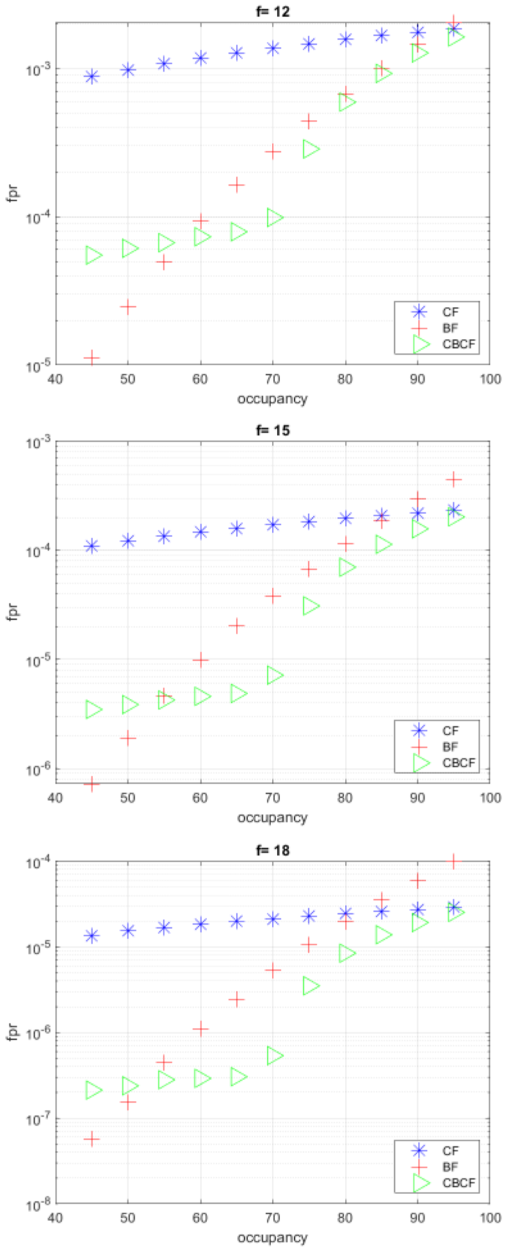

Fig. 7. False Positive Rate of the proposed CBCF, the original cuckoo filter and a Bloom filter for different fingerprint sizes and occupancy

simulations and this corresponds to a probability of a bit in the filter being zero of − so that the average number of accesses is − − − that tends to a value of two [9]. Finally, it can be seen that the reduction in the number of memory accesses with occupancy is similar for the different values of . These results show that the Bloom filter will outperform both the CF and CBCF in terms of memory accesses when occupancy is below the maximum value. This is in line with recent studies that show a speed advantage of Bloom filters over cuckoo filters for software implementation [15].

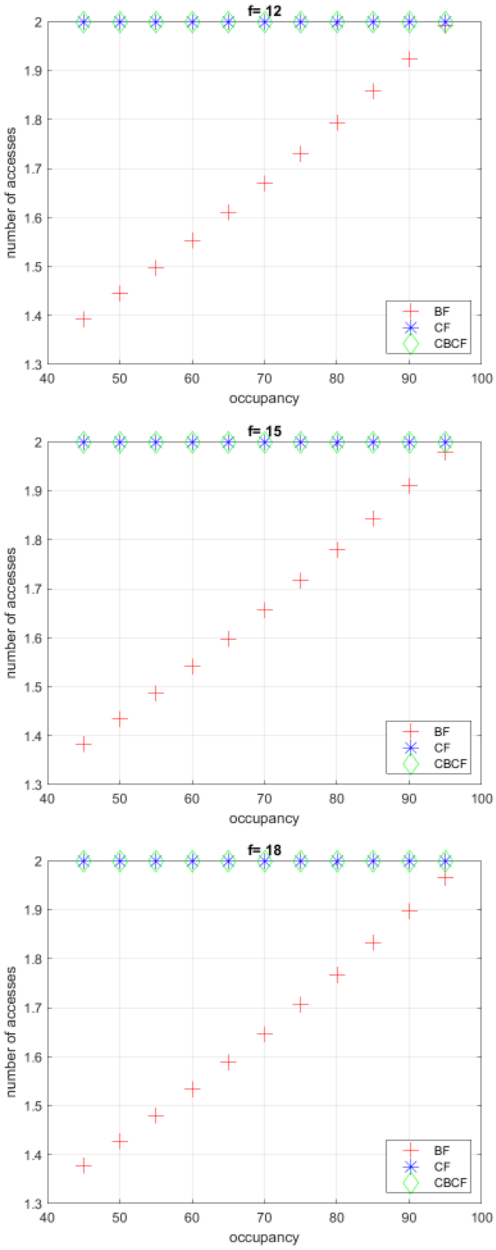

Fig. 8. Average number of memory accesses to complete a negative lookup

Finally, simulations have been run to evaluate the performance for insertions and removals. Again, the number of memory accesses is used as the metric for comparison. In the case of the Bloom filter, the number of memory accesses for insertion is constant and equal to (which as discussed before is different for each value of). The results for the CF and CBCF are similar as the insertion procedure is the same and are shown in Figure 9. It can be observed that the number of memory accesses increases very significantly as we approach the filter capacity. This is in line with the expected behaviour of cuckoo filters [5]. For insertions, the CF and CBCF outperform the Bloom filter except when their occupancy is above 90%. As for removals the results are shown in Figure 10. In this case, the Counting Bloom Filter (CBF) is used for comparison. It can be seen that again the CF and CBCF have similar results. In this case they outperform the Counting Bloom filter as they only need approximately 2.5 memory accesses; that is smaller than the optimum values of used in the Bloom filters. The results for the CF and CBCF do not depend on the number of fingerprint bits . Instead, for the Bloom filter as the optimum values of varies with , so does the number of memory accesses. Finally, it is worth mentioning that in a dynamic scenario, both insertions and removals will take place and thus the larger cost of insertions on the CF and CBCF when occupancy is high will be partially compensated with the lower cost of removals.

The evaluation results show that the CBCF can effectively extend the occupancy range for which the cuckoo filter outperforms the Bloom filter in terms of false positive rate. In terms of performance, the Bloom filter requires fewer memory accesses per negative lookup on average than both the CF and CBCF. This would lead to faster lookups in some platforms as recently shown in [15]. For insertions both the CF and CBCF require a large number of memory accesses when the filter operates close to full occupancy but outperform the Bloom filter at lower occupancy. Finally, removals require fewer memory accesses in the CF and CBCF than in the Bloom filter.

C. Implementation overheads

A potential drawback of the proposed CBCF could be the implementation overhead compared to a plain CF. In most networking applications, the most frequent operation is the lookup, therefore in the following, the discussion focuses on it. For a lookup, the number of memory accesses for elements not in the set (which is the worst case) is two both for the CBCF and the CF. For lookups of elements that are in the set, the

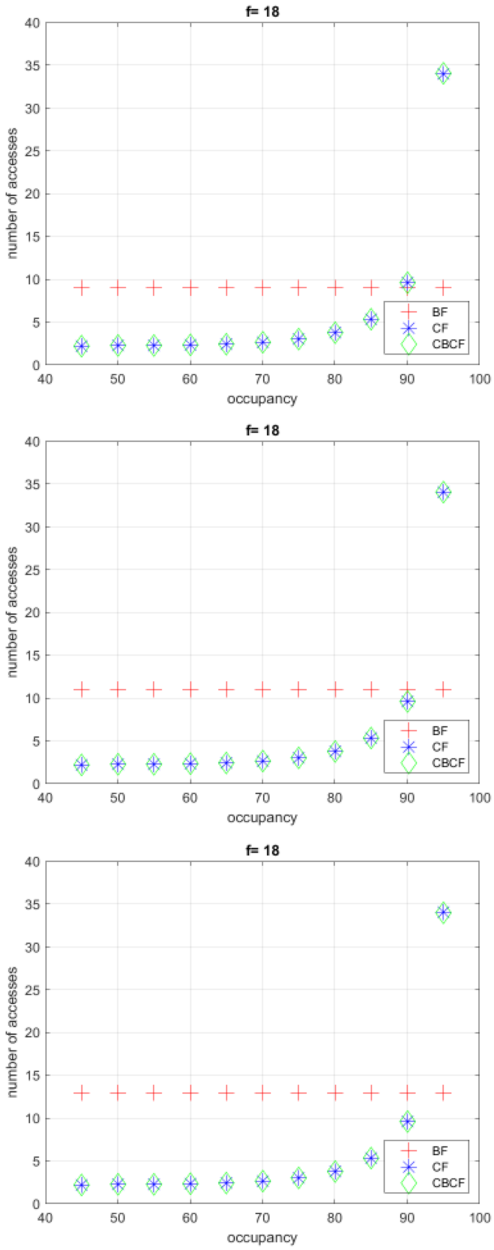

Fig. 9. Average number of memory accesses to complete an insertion

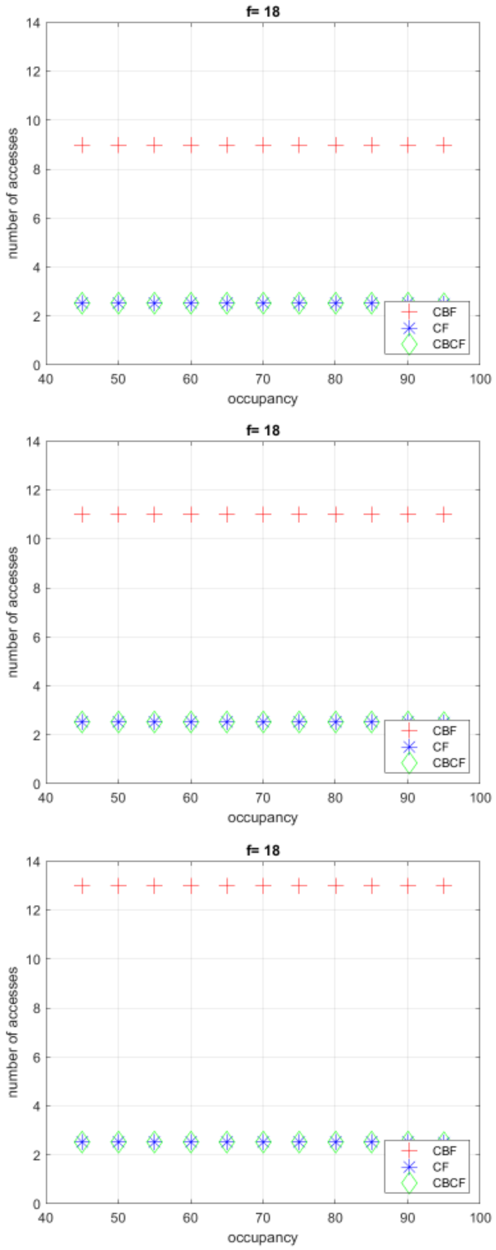

Fig. 10. Average number of memory accesses to complete a removal

number of memory accesses depends on whether the fingerprint is stored in its first or second bucket but again is the same for the CBCF an the CF. Therefore, the CBCF does not introduce any overhead in terms of memory accesses. However, a lookup in the CBCF may require computing larger fingerprints of the searched elements (when the selection bit). The overhead to do this should be negligible as modern processors have dedicated instructions that compute a hash of the word size [28] and thus provide a larger number of fingerprint bits than needed both for the CF and the CBCF. Therefore, the additional complexity comes from reading the selection bit and then based on it perform comparisons against three or four elements. The CBCF does require a small overhead in terms of memory as an additional bit is needed per bucket to store the selection bit. For example, when this overhead would be approximately 2%. This suggests that the overhead should be acceptable in many implementations.

Finally, the scrubbing procedure used to minimize the number of buckets that are full is also an overhead. However, as insertion and removals on most networking applications are not frequent and the procedure only needs to be run after a significant number of insertions, it will be run rarely thus having no relevant impact on the filter operation.

V. CONCLUSIONS

This paper has illustrated the importance of filters and compared Bloom filters and cuckoo filters. The analysis shows that one of the disadvantages of cuckoo filters is that their false positive rate reduces linearly as occupancy lowers whereas Bloom filters instead have a faster reduction. This means that when the filter operates below its maximum occupancy as it would be the case of many practical configurations, a Bloom filter could provide better performance.

Given the evident interest in cuckoo filters over Bloom filters for applications that require deletion support and low false positive rates at very high occupancy, we focus on extending the occupancy ranges for which the cuckoo filters are competitive. To this end, an optimized elastic cuckoo filter has been proposed in this article. The Configurable-Bucket Cuckoo Filter (CBCF) exploits unused cells in buckets to use longer fingerprints for the elements stored in the bucket thus reducing its contribution to the false positive rate. The simulation results show that indeed the CBCF can extend the occupancy values for which the cuckoo filter outperforms the Bloom filter. To support fingerprints of different sizes, the CBCF needs to store the full elements (or at least the longest fingerprints) in another table. Therefore, it is only appropriate for applications that by their nature store the full elements or that have a slower larger memory on which the full elements can be stored with no relevant impact on cost.

The proposed CBCF could be extended to support multiple configurations on a bucket, for example, by using two selection bits per bucket. Those configurations are left for future work as it seems that they would provide most of their benefit at lower occupancy where the filter should not operate in configurations with practical interest.

ACKNOWLEDGMENT

Pedro Reviriego and David Larrabeiti would like to acknowledge the support of the ACHILLES project PID2019-104207RB-I00 and the Go2Edge network RED2018-102585-T funded by the Spanish Ministry of Science and Innovation and of the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496. David Larrabeiti acknowledges the support of EU project PASSION, Grant Agreement 780326. Salvatore Pontarelli has been partly funded by the EU commission in the context of the 5G-PICTURE project, Grant Agreement 762057.

REFERENCES

[1] D. Pellow, D. Filippova, and C. Kingsford, "Improving Bloom filter performance on sequence data using k-mer Bloom filters," Journal of Computational Biology, vol. 24, no. 6, pp. 547–557, 2017.

[2] T. J. Ashby, P. Díaz and M. Cintra, "Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters," in IEEE Transactions on Computers, vol. 60, no. 4, pp. 472-483, April 2011.

[3] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," Computer Networks, vol. 57, no. 18, pp. 4047–4064, Dec. 2013.

[4] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich and X. Luo, "Optimizing Bloom Filter: Challenges, Solutions, and Comparisons," in IEEE Communications Surveys and Tutorials, vol. 21, no. 2, pp. 1912-1949, 2019.

[5] B. Fan, D. Andersen, M. Kaminsky and M. Mitzenmacher, "Cuckoo Filter: practically better Than Bloom," in Proceedings of CoNext 2014.

[6] J. Grashöfer, F. Jacob and H. Hartenstein, "Towards application of cuckoo filters in network security monitoring," 14th International Conference in Network and Service Management (CNSM), pp. 373-377, 2018.

[7] M. Kwon, P. Reviriego and S. Pontarelli, "A length-aware cuckoo filter for faster IP lookup," in Proceedings of the IEEE Infocom (WKSHPS), pp. 1071-1072, 2016.

[8] J. Cui, J. Zhang, H. Zhong and Y. Xu, "SPACF: A Secure Privacy-Preserving Authentication Scheme for VANET with Cuckoo Filter," in IEEE Transactions on Vehicular Technology, vol. 66, no. 11, pp. 10283-10295, Nov. 2017.

[9] A. Z. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," Internet Mathematics, 2005.

[10] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, and M. F. Magalhaes, "The deletable bloom filter: a new member of the Bloom family,“ IEEE Communications Letters, vol. 14, no. 6, pp. 557-559, June 2010.

[11] H. Lim, J. Lee, H. Byun, and C. Yim, "Ternary bloom filter replacing counting bloom filter“ IEEE Communications Letters, vol. 21, no. 2, pp. 278-281, 2017.

[12] O. Rottenstreich, Y. Kanizo and I. Keslassy, ”The Variable-Increment Counting Bloom Filter," in IEEE/ACM Transactions on Networking, vol. 22, no. 4, pp. 1092-1105, Aug. 2014.

[13] P. Reviriego and O. Rottenstreich, ”The Tandem Counting Bloom Filter - It Takes Two Counters to Tango," in IEEE/ACM Transactions on Networking, vol. 27, no. 6, pp. 2252-2265, Dec. 2019.

[14] S. Pontarelli and P. Reviriego, "Cuckoo Cache: A Technique to Improve Flow Monitoring Throughput," in IEEE Internet Computing, vol. 20, no. 4, pp. 46-53, July-Aug. 2016.

[15] H. Lang, T. Neumann, A. Kemper, and P. Boncz, "Performance-optimal Filtering:Bloom overtakes cuckoo at high throughput," PVLDB, 12(5):502–515, Jan. 2019.

[16] U. Erlingsson, M. Manasse and F. Mcsherry, "A cool and practical alternative to traditional hash tables," in Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS), 2006.

[17] P. Gupta and N. McKeown, "Algorithms for packet classification," IEEE Network, vol. 15 no. 2, pp. 24-32, 2001.

[18] C. Hsieh and N. Weng, ”Many-field packet classification for software-defined networking switches," in Proceedings of ACM/IEEE ANCS, Santa Clara, CA, 2016, pp. 13-24.

[19] H. Lim and S. Kim, "Tuple Pruning Using Bloom Filters for Packet Classification," IEEE Micro, vol. 30, no. 3, pp. 784-794, May/June 2010.

[20] Y. Kanizo, D. Hay, and I. Keslassy, “Maximizing the Throughput of Hash Tables in Network Devices with Combined SRAM/DRAM Memory,” IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 3, pp. 796-809, March 2015.

[21] J. Mogul, and P. Congdon. “Hey, you darned counters! get off my ASIC!,” in Proceedings of the first workshop on Hot topics in software defined networks. 2012.

[22] K. Daehyeok, et al. “Generic external memory for switch data planes,” in Proceedings of the 17th ACM Workshop on Hot Topics in Networks. 2018.

[23] BCM88690–10 Tb/s StrataDNX Jericho2 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690, 2018.

[24] F. Soldo, K. Argyraki and A. Markopoulou, “Optimal Source-Based Filtering of Malicious Traffic,” in IEEE/ACM Transactions on Networking, vol. 20, no. 2, pp. 381-395, April 2012.

[25] G. Huston and A. Grenville “Projecting future IPv4 router requirements from trends in dynamic BGP behaviour,”, in Proceedings of the Australian Telecommunication Networks and Applications Conference (ATNAC), 2006.

[26] A. Elmokashfi, A. Kvalbein and C.Dovrolis, “On the scalability of BGP: the roles of topology growth and update rate-limiting,” in Proceedings of the ACM CoNEXT Conference, 2008.

[27] M. Mitzenmacher, S. Pontarelli and P. Reviriego, “Adaptive Cuckoo Filters,” in Proceedings of ALENEX, 2018.

[28] “Intel-64 and IA-32 Architectures Software Developer’s Manual”, volume 2, Intel, 2001.

Pedro Reviriego received the M.Sc. and Ph.D. degrees in telecommunications engineering from the Technical University of Madrid, Madrid, Spain, in 1994 and 1997, respectively. From 1997 to 2000, he was an Engineer with Teldat, Madrid, working on router implementation. In 2000, he joined Massana to work on the development of 1000BASE-T transceivers. From 2004 to 2007, he was a Distinguished Member of Technical Staff with the LSI Corporation, working on the development of Ethernet transceivers. From 2007 to 2018 he was with Nebrija University. He is currently with Universidad Carlos III de Madrid working on high speed packet processing and fault tolerant electronics.

Jorge Martínez received the B.Sc. degree in computer science from Tecnológico de Monterrey, México in 1989, the M.Sc. degree from Universidad Complutense de Madrid, Madrid, Spain, in 2013 and the Ph.D. from Universidad Antonio de Nebrija, Madrid, Spain in 2017. He is currently with Universidad Antonio de Nebrija. His research interests include fault tolerance and reliability.

David Larrabeiti is professor in Switching and Network Architectures at the Telematics Department of UC3M since 1998. He has participated in a number of EU research projects on next generation networks, like INDECT, FED4FIRE and the Networks of Excellence e-Photon/One, e-Photon/One+, BONE and the H2020 5GPPP CrossHaul project. His research interests include fast switching technologies and cybersecurity in networking. He is currently participating in 5GPPP H2020 project BlueSPACE and KET PASSION H2020 project. His publications include papers at IEEE Communications magazine, IEEE Network, IEEE Multimedia, IEEE Communications letters, Journal of Lightwave Technology, Journal of Optical Communications and Networking, HPSR, ECOC, Optical Switching and Networking and others.

Salvatore Pontarelli received a master degree in electronic engineering at University of Bologna and the PhD degree in Microelectronics and Telecommunications from the University of Rome Tor Vergata. Currently, he works as Senior Researcher at CNIT (Italian National Inter-University Consortium for Telecommunications), in the research unit of University of Rome Tor Vergata. In the past he worked with the National Research Council (CNR), the Department of Electronic Engineering of University of Rome Tor Vergata, the Italian Space Agency (ASI), the University of Bristol. He participated in several national and EU funded research programs (ESA, FP7 and H2020). His research interests include hash based structures for networking applications, use of FPGA for high speed network monitoring and hardware design of software defined network devices.