Perfect Cuckoo Filters

Pedro Reviriego
Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid
Leganés, Madrid, Spain
revirieg@it.uc3m.es

Salvatore Pontarelli
Department of Computer Science, Sapienza University of Rome
Rome, Italy
salvatore.pontarelli@uniroma1.it

ABSTRACT

Bloom filters and cuckoo filters are used in many applications to reduce the amount of memory needed to check if an element belongs to a set. The main drawback of these filters is that with low probability, a positive is returned for an element that is not in the set. Recently, the concept of Bloom filters with a false positive free zone has been introduced showing that false positives can be avoided when the universe from which elements are taken and the number of elements inserted in the filter are both small. Unfortunately, this limits the use of such false positive free Bloom filters in many practical applications. In this paper, a false positive free, i.e. perfect, cuckoo filter is presented and evaluated. The proposed design supports universe sizes of billions of elements and stores millions of elements, making it practical for a wide range of applications. The perfect cuckoo filter can be also used to perform <key,value> mapping, further extending the range of scenarios in which can be used. The benefits of the proposed perfect cuckoo filter are illustrated with two case studies: IP address blacklisting and longest prefix match for IP forwarding.

CCS CONCEPTS

• **Theory of computation → Sorting and searching**; • **Networks → Network algorithms**.

KEYWORDS

Approximate membership checking, false positives, cuckoo filters, packet processing.

ACM Reference Format:
Pedro Reviriego and Salvatore Pontarelli. 2021. Perfect Cuckoo Filters. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December, 2021.*. ACM, New York, NY, USA, 7 pages. https://doi.org/TBA

1 INTRODUCTION

Testing if an element belongs to a set is a common operation in many computing and networking applications such as checking if a given source IP address is in a blacklist [4] or if a DNA sequence contains a given pattern [12]. This checking can be implemented by storing the full set in the memory and searching if the requested element is among those stored. Unfortunately, when the set is large, the amount of memory needed can be excessive such that for example it does not fit into the cache and operations are slower. When that is the case, an approximate rather than exact membership check can be used to reduce the memory footprint and speed up the checking. Bloom filters [11] and cuckoo filters [6] are widely used to implement approximate membership checking and reduce the amount of memory needed at the cost of introducing false positives with low probability. The impact of these false positives depends largely on the application. When the filter is used to detect if an element is stored in an external table and only then access the table [3, 11, 16], a false positive only incurs in an unneeded access to the external table. Instead, if the filter is used to check if the source IP address of a packet is in a blacklist to then block the packet, a false positive for a given IP address would disable communication from that IP address. This is an example in which Bloom filters cannot be applied, since even a very small false positive probability is not acceptable. However, in all cases it is beneficial to reduce the false positives and if possible, eliminate them. Indeed, many efforts have been made to reduce false positives by proposing new filter structures such as the cuckoo filter [6] or by modifying the Bloom filter [5, 10, 11, 15, 18, 21].

More recently, the concept of a Bloom filter with a False Positive Free Zone (FPFZ) has been introduced [8]. These filters completely eliminate false positives for a given universe size when the number of elements inserted in the filter is below a given threshold, i. e. the filter is inside its FPFZ. This is achieved by using mappings of elements to bits in the filter that have special properties. For example, mappings based on polynomials or error correction codes have been proposed to provide a FPFZ [19]. Unfortunately, in all cases, the FPFZ is small supporting only a few elements and the universe is also small. This significantly limits the range of applications in which they can be used.

Efforts have also been made to reduce the false positive rate of cuckoo filters, for example by using adaptation to remove false positives once they are detected [13]. However, to the best of our knowledge, all existing cuckoo filter variants have a false positive probability that is larger than zero. In this paper, we explore the design of false positive free, i.e. perfect cuckoo filters and present a scheme to implement them. The proposed scheme completely eliminates false positives for a given universe size and, in some configurations, requires a memory footprint that is similar to that of traditional cuckoo filters. This makes the perfect cuckoo filters of interest since they can be used in applications that cannot tolerate false positives and also in all the applications in which minimizing false positives is important. The main contributions of this paper can be summarized as:

CoNEXT '21, December, 2021,
© 2021 Association for Computing Machinery.
ACM ISBN TBA…$TBA
https://doi.org/TBA

- A novel data structure derived from the cuckoo filter, with no false positives is proposed.
- The proposed structure, differently from existing Bloom filters with a false positive free zone, can support both large universes and sets.
- We show that additionally, the new data structure provides better results in terms of load factor with respect to standard cuckoo filters when a small number of bits is used for the fingerprint (see §3.3).
- We show how to extend the perfect cuckoo filter to provide a compressed {key,value} mapping (§3.2).
- The benefits of the proposed data structure are shown in two practical use cases: IP blacklisting and Longest Prefix Match.

The rest of the paper is organized as follows. Section 2 covers the preliminaries on Bloom filters with a FPFZ and cuckoo filters. The proposed false positive free cuckoo filters are presented in Section 3. Section 4 presents two case studies used to evaluate the proposed scheme, validating that no false positives occur over the entire universe, and comparing the memory footprint to that of traditional solutions. Section 5 draws the conclusion and some ideas for future work.

2 PRELIMINARIES

2.1 Bloom filters

A Bloom filter [2] maps elements to an array of m bits using k hash functions $h_1(x), h_2(x), .., h_k(x)$. To insert an element x in the filter, the k bits on the positions given by those hash functions are set to one. To check if an element has been inserted in the filter, those same positions are checked and when all of them are set to one, a positive is returned. Otherwise the result is a negative. Bloom filters, by construction cannot have false negatives as if element x has been inserted on the filter, then bits on positions $h_1(x), h_2(x), .., h_k(x)$ are one. Instead, false positives can occur as those positions may be one when x has not been inserted in the filter if other elements inserted in the filter also map to them.

The false positive probability of a traditional Bloom filter depends on the array size m, the number of elements inserted in the filter n, and the number of hash functions used k. False positives can be avoided by using mapping functions with special features that ensure that a set of elements of size d or less cannot create false positives. These filters are denoted as Bloom filters with a False Positive Free Zone (FPFZ) and several constructions have been proposed to build such filters [8],[19]. In more detail, three constructions based on prime numbers (EGH), polynomials (POL) and orthogonal latin squares error correction codes (OLS) have been presented.

The memory complexity and number of probes for lookups of existing constructions to implement a FPFZ of size d over a universe of size U are summarized in Table 1 that also includes that of the proposed perfect cuckoo filter (PCF). From the table, it becomes apparent that existing filters are not practical when d and U are large. As an example, let us consider a false positive free zone of size $d = 2^{10}$ over a universe of size $U = 2^{24}$. The required memory for such setting for each of the constructions in table 1 would be 170Mb for EGH, 4Mb for OLS and 500K bits for POL with $t = 3$ respectively. This corresponds to approximately 500 bits per element in the filter

Table 1: Memory and number of probes required by different Bloom filters with a FPFZ of size d over a universe of size U. For the POL filter, t is a controllable parameter that has integer values larger than one. The proposed perfect cuckoo filters (PCF) are included for completeness.

filter	memory complexity	# probes
EGH	$O(d^2 \cdot \log U)$	$O(d \cdot \log U)$
OLS	$O(d \cdot \sqrt{U})$	$O(d)$
POL	$O(t \cdot d \cdot \sqrt[t]{U})$	$O(t \cdot d)$
PCF	$O(d \cdot \log U)$	2

in the best case. An exact membership check implementation using for example cuckoo hash [14], would require approximately 24 bits per element so the FPFZ filter requires more memory than an exact representation. Therefore, it is of interest to find constructions of data structures that can implement a FPFZ for large values of d and U using fewer bits per element than an exact representation. This is indeed achieved by the proposed PCF that has a memory cost that is linear on the set size and logarithmic on the universe size as seen in table 1. The PCF needs also the lowest number of probes for lookups making it faster than existing FPFZ Bloom filters.

2.2 Cuckoo filters

A cuckoo filter is an approximate membership check data structure derived from cuckoo hashing [14]. In a cuckoo filter, a fingerprint of the element $fp(x)$ is computed using a hash function and stored on a table. The fingerprint can be stored in two positions given by addresses $a_1 = h_1(x)$ and $a_2 = h_1(x)\ xor\ h_2(fp(x))$ where $h_1(x), h_2(x)$ are also hash functions. A position on the table can store c fingerprints and typically $c = 4$ as shown in Figure 1. To check if an element x is stored in the filter, the two addresses are computed and the fingerprints stored on them are compared to $fp(x)$. A positive is returned if there is a match and a negative otherwise. If the fingerprints have f bits and the filter occupancy is o, false positive rate of the cuckoo filter can be approximated by $\frac{2 \cdot c \cdot o}{2^f}$.

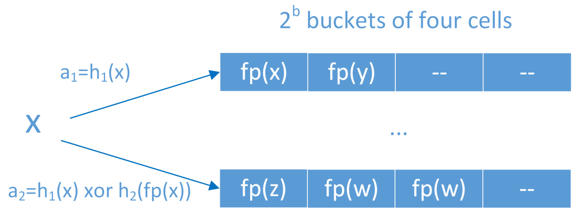

Figure 1: Illustration of a cuckoo filter with four cells per bucket

To insert an element x in the filter, its two addresses are computed and if there is an empty cell on any of them, $fp(x)$ is stored in one of the empty cells. Instead, when all cells in both addresses are occupied, one of the stored fingerprints $fp(y)$ is selected randomly and moved to its alternate bucket and $fp(x)$ is inserted on its place. If when moving $fp(y)$ its alternate address is also fully occupied, one of the fingerprints stored there is selected and moved and the process continues until an address with an empty cell is found.

To the best of our knowledge, no construction of a cuckoo filter that avoids false positives has been presented so far. In the following section, a scheme to implement false positive free cuckoo filters is presented.

3 PERFECT CUCKOO FILTERS

In this section, we first present the construction of the perfect cuckoo filters, their application for <key,value> mapping and then discuss the minimum fingerprint size that can be used in the proposed filters and their use when some false positives are allowed.

3.1 Construction

Let us consider a universe of elements U with size 2^u so that each element in U can be represented with u bits and a cuckoo filter of 2^b buckets. For example, a cuckoo filter with $b = 16$ and 64K buckets each with 4 cells that can store up to 256K IPv4 addresses of 32 bits so that $u = 32$. Our goal is to design a cuckoo filter like structure that has no false positives over the universe U.

A first observation is that there are some hash functions that map U to U with no collisions. That is, $h()$ is bijective, so for any $x \neq y \in U$ we will have $h(x) \neq h(y)$. In particular, CRCs of u bits based on primitive polynomials are bijective when the input has u bits [20]. For example, a CRC32 maps the universe of 32-bit values with no collisions. The same seems to apply to other hash functions like murmurhash[1] [1]. Let us denote this function as $m(x)$. Then a data structure that stores $m(x)$, will not have collisions on U.

A second observation is that since $m(x)$ is a hash function, its bits can be used to form smaller hash functions needed in a cuckoo filter. In particular, on a cuckoo filter, a fingerprint $fp(x)$ is stored in one of two buckets with addresses $a_1 = h_1(x)$ and $a_2 = h_1(x)\ xor\ h_2(fp(x))$. Therefore, hash functions $h_1(x)$ and $fp(x)$ are needed.

To build a false positive free cuckoo filter we can proceed as follows:

(1) Assign $h_1(x)$ to be the lower b bits of $m(x)$.
(2) Assign $fp(x)$ to be the remaining $u - b$ bits of $m(x)$.

This generation of $h_1(x)$, $fp(x)$ from x using $m(x)$ is illustrated in Figure 2.

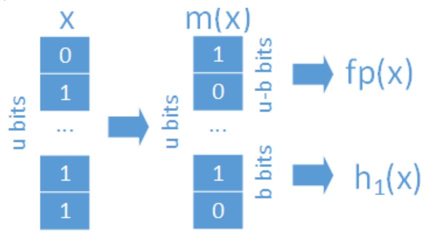

Figure 2: Generation of the fingerprint $fp(x)$ and the first bucket address $h_1(x)$ for element x using a collision free hash function $m(x)$

Then, an element inserted in its first bucket is given by $\{fp(x), a_1\} = \{fp(x), h_1(x)\} = m(x)$. Let us consider the cases that can lead to a false positive:

(1) Since $m(x) \neq m(y)$ for all x, y in U, no false positives can be created between elements inserted on their first bucket.

(2) Similarly, for the second bucket we have $\{fp(x),a_2\} = \{fp(x), h_1(x)\ xor\ h_2(fp(x))\} = m(x)$ xor $h_2(fp(x))$. As $m(x) \neq m(y)$ either $fp(x) \neq fp(y)$ or $h_1(x) \neq h_1(y)$ or both. A false positive can only be created if $fp(x) = fp(y)$. But if that is the case, $m(x)$ xor $h_2(fp(x))$ and $m(y)$ xor $h_2(fp(y))$ will be different as the second term is the same and the first one is different. So again, no false positives can be created between elements inserted on their second bucket.

(3) The last possibility for a false positive would be that $m(x) = m(y)\ xor\ h_2(fp(y))$, so that when looking for the first bucket we get a false positive due to the insertion of an element on its second bucket or the other way around. Again, a false positive can only be created if $fp(x) = fp(y)$ so that the lower bits of $m(x), m(y)$ are different. However, in this case if the difference is exactly $h_2(fp(y))$, a false positive can occur.

Therefore, there is a case where FPs are still possible. To avoid this, an additional bit s is stored in the filter. That is, each cell stores $\{s, fp(x)\}$ and s is set to zero if the element is stored in its a_1 bucket and to one if it is stored on its a_2 bucket. This means that false positives are not possible as now in case 3) we would have $s_x \neq s_y$. The proposed filter is illustrated in Figure 3.

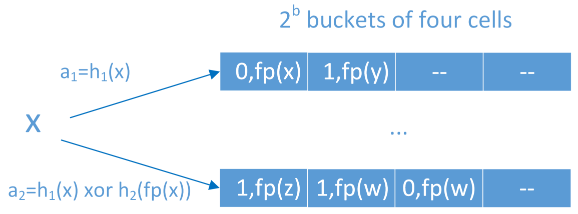

Figure 3: Proposed Perfect Cuckoo Filter: a selector bit is added to the entries to mark if the element is on its first (second) bucket with 0 (1). In the figure, a lookup for x is shown.

From the previous analysis, it seems we can build false positive free cuckoo filters when the universe U is not very large. For the IPv4 example discussed, with a 128K buckets filter we would need 15+1 =16 bits for the cells to achieve a false positive free filter. This seems attractive and more powerful than FPFZ-BFs. In particular it enables us to filter IPv4 addresses with no false positives using only 16 bits. The speed of the proposed PCFs would be implementation dependent but they require the same number of memory accesses as a plain cuckoo filter.

Finally, it is interesting to note that since there can be at most two elements that map to the same position with the same fingerprint, the PCF does not seem to be vulnerable to adversarial inputs that force an insertion failure by inserting more than eight elements with the same fingerprint mapping to the same positions [17].

3.2 <key,value> mapping with PCFs

In standard cuckoo filters, each cell can correspond to multiple keys sharing the same hash value and fingerprint. Thus it is not possible to associate a value to the key, since several keys can collide in the same position. Instead, in the perfect cuckoo filter, there is a one-to-one mapping between the cell location and the key inserted in the filter. Therefore, the cell location can be used as a pointer to associate a value to each key stored in the filter. Given a set of pairs $\{x, v_x\}$, it is sufficient to store in each cell the triple $\{s, fp(x), v_x\}$ instead of $\{s, fp(x)\}$. The query operation provides as answer a *not found* value if the key x is not in the filter, or the value v_x if the key is present. We exploit this additional feature of the perfect cuckoo filter in the use case of longest prefix match presented in Section 4.2.2.

[1]We have tested this by running murmurhash3 on all possible 32-bit values with different seeds.

3.3 Minimum fingerprint size

A limitation of cuckoo filters is that the occupancy that can be achieved degrades when the fingerprint size is small and tables are large [6]. This is in part caused by having too many elements with the same fingerprint mapping to a given bucket. Since the second bucket is determined by the fingerprint, elements that map to a bucket with the same fingerprint would also map to the same alternative bucket and if there are more than eight[2] such elements, an insertion failure would occur [6].

Let us consider our perfect cuckoo filter when using only three bits per fingerprint, a value for which the traditional cuckoo filter suffers a large degradation in its maximum occupancy. Then, the filter would have 2^{u-3} buckets and exactly eight elements would map to each bucket with $h_1(x)$ with fingerprints 000,001,010,011,100,101, 110,111. This means that there would be no conflict between the fingerprints as all would have different values. Let us now consider another element y that maps to another bucket $h_1(y)$ such that $h_1(x) = h_1(y)\ xor\ h_2(fp(y))$ and $fp(x) = fp(y)$. Then, that element y would be unique as it has to map to bucket $h_1(x)\ xor\ h_2(fp(y))$ and only one element in that bucket will have $fp(y) = fp(x)$. Therefore, the proposed perfect cuckoo filters should not suffer any degradation in the achievable occupancy when using small fingerprints due to having the same fingerprint more than eight times on a bucket. This is an interesting result that we have checked by simulation (see Section 4.1) and that has practical implications for one of the case studies presented in the evaluation section (Section 4.2 with $f = 5$).

3.4 Allowing false positives in the PCF

The proposed PCF can be used with fewer fingerprint bits than needed to avoid false positives. Let us consider a PCF that uses $f - r$ fingerprint bits instead of f. Then for each element there are $2^r - 1$ elements that can cause a false positive and thus the false positive probability would be approximately $\frac{4 \cdot o \cdot 2^b \cdot (2^r - 1)}{2^u} = \frac{4 \cdot o \cdot (2^r - 1)}{2^f}$ where o is the filter occupancy. This compares with the false positive probability when using non bijective hash functions that would be approximately $\frac{4 \cdot o}{2^{f-r}}$. When r is large, both probabilities are the almost the same but when r is small, the use of bijective hash functions reduces the false positive probability significantly. For example, when $r = 1$, the probability is only half that of the traditional filter. Therefore, the proposed PCF can also reduce the false positive probability when fewer than f bits are used for the fingerptints.

This means that bijective hash functions can also be used to implement filters with few fingerprint bits when the universe is not large but we do not need to avoid false positives. However, this is not explored further in this paper as our main objective is indeed to completely avoid false positives.

4 EVALUATION

To evaluate the proposed perfect cuckoo filter a C++ simulator has been developed.[3]. The hash function used is configurable and can be selected to be CRC24, CRC32, MurMurHash3, CityHash or XXHash[4]. When not differently stated, the experiments are performed targeting the universe of 32-bit elements that corresponds to the IPv4 address space. The implementation is first used to validate that the proposed perfect cuckoo filter does not have false positives. After, two case studies are used to illustrate the benefits of the proposed filter: (i) the implementation of a blacklist of IPv4 addresses and (ii) a Longest Prefix Match algorithm. These case studies illustrate the applicability and benefits of the proposed filter in a real scenario.

4.1 Validation

The first part of the evaluation focuses on checking that our perfect cuckoo filter has no false positives when the hash function is bijective. To that end, two bijective functions on the 32-bit universe (CRC32 and MurMurHash3) were tested. Additionally, two hash functions that are not bijective, CityHash and XXHash were also tested. The proposed filter was constructed for different sizes and filled to a 95% occupancy. Then the number of false positives over the entire universe was measured. The experiment was run multiple times and in all cases the proposed perfect cuckoo filter did not have any false positive when using a bijective hash function. The number of false positives when using functions that are not bijective were similar in all runs. The results are summarized in Table 2 for one of the runs. The table also shows the theoretical estimate for the number of false positives that is given by $\frac{4 \cdot o}{2^f}$ where o is the filter occupancy (in our experiments 0.95) and f the number of fingerprint bits. It can be seen that when a bijective hash function is used, our proposed scheme has no false positives. Instead, when a hash function that is not bijective is used, there is a number of false positives that matches well with the theoretical estimate. This first experiment confirms that the proposed approach does indeed avoid false positives. Furthermore, the experiment suggests that the use of bijective hash functions does not affect the maximum load factor achievable with cuckoo filters. The same experiment was repeated using $f - r$ fingerprint bits and the false positive probability matched that of the analysis in section 3.4. In a second experiment, we tested the occupancy that the proposed perfect cuckoo filter can achieve when using few fingerprint bits. In particular, a universe of $u = 24$ bits was considered with a filter with $b = 20$ and $f = 4$.

Table 2: Number of false positives over the 32-bit universe for a perfect cuckoo filter with 2^b buckets of four cells and fingerprints of f bits when using different hash functions

b	f	CRC32	MurMur Hash3	XXHash	CityHash	Theoretical
10	22	0	0	3936	3812	3891
12	20	0	0	15577	15543	15564
14	18	0	0	62436	62317	62259
16	16	0	0	248662	248852	249036
18	14	0	0	995500	996863	996147
20	12	0	0	3980877	3979894	3984588
22	10	0	0	15853006	15848709	15938355

The CRC24 hash function was used to build the filter and then other not bijective hash functions were also tested. The filter is

[2]As each bucket has four cells, at most eight elements can be placed on a pair of buckets.

[3]The source code is available in this link: https://github.com/pontarelli/perfectCF

[4]Further details on the hash functions are available at https://github.com/rurban/smhasher

constructed and elements are inserted until the first insertion fails. The occupancy at that point is logged and the minimum value across all the runs is reported. The results in terms of the worst case occupancy that is achieved over one thousand trials are summarized in Table 3. It can be seen that when the function is bijective (CRC24), occupancy is above 95%, similar to that achieved with larger fingerprints. Instead, for the functions that are not bijective on the 24-bit universe, occupancy is much lower as previously observed for cuckoo filters [6]. This confirms the analysis in section 3.3 and shows that perfect cuckoo filters can be constructed with few fingerprint bits.

Table 3: Worst case occupancy achieved over 1,000 runs with the 24-bit universe for cuckoo filters with 2^{20} buckets of 4 cells and fingerprints of 4 bits when using different hash functions

CRC24	MurMurHash3	XXHash	CityHash
96.71%	46.15%	60.64%	56.32%

4.2 Case studies

The benefits of the PCF can be significant when the number of bits needed to represent the universe is not much larger than the number of hash bits needed to represent the buckets on the filter. This is for example the case for most applications that use IPv4 so that the universe has 32 bits and the hash for the bucket may have 10-20 bits. Instead, when the universe is much larger, as for example in IPv6 with 128 bits, the benefits of the proposed PCFs are much smaller. To illustrate the potential benefits of the proposed scheme, its use in two applications is evaluated. The first one is the blacklisting of IP addresses. The second is the implementation of Longest Prefix Match for IPv4 for which cuckoo filters have been proposed in the past [9].

4.2.1 IP Blacklist. In networks, it is common practice to identify malicious host by their IP address and block any traffic coming from those IP addresses [4],[23]. This requires checking every incoming packet to see if its source IP address is in the blacklist. The number of IP addresses in the list is potentially very large and thus the blacklist may require a large amount of memory. Therefore, it would be beneficial to use a filter to check if an element is in the blacklist and only when the filter returns a positive check the full list. Note that in this case, the check of the full list is needed to avoid false positives from blocking legitimate IP addresses. Clearly, in this application a filter that is false positive free would be beneficial as it would remove the need to check the full list on a positive. In the case of IPv4, addresses have 32 bits and thus the universe has a size of 2^{32} while the number of elements stored can be in the order of hundreds of thousands.

To illustrate the benefits of using our perfect cuckoo filters, a public spam blacklist with close to 200,000 IP addresses was used[5]. The blacklist can be stored in a cuckoo hash table with 2^{16} buckets of four cells using the 32-bit IPv4 addresses as key. This would require $2^{16} \cdot 4 \cdot 32$, so 1MB of memory. Instead, the proposed filter requires only $2^{16} \cdot 4 \cdot 17$ that is 544K so approximately half the memory.

[5]http://iplists.firehol.org/?ipset=stopforumspam_90d

4.2.2 Longest Prefix Match in IPv4. The second case study considers the implementation of longest prefix match in IPv4 using a binary search tree on prefix lengths [22]. In this application, the proposed perfect cuckoo filters can significantly reduce the memory needed. As an example, let us consider a tree with only three nodes so that the number of prefix lengths checked per lookup is two in the worst case and thus comparable to the well-know DIR-24-8 algorithm [7], which instead has a higher memory footprint[6]. The root node is fixed at /24 and the right node at /32 while the left node was selected to minimize the memory footprint and set to /20. All the prefixes with other lengths are expanded to the closest larger value. Let us summarize here the method proposed in [22], applied to the specific example: The tree is populated as follows:

(1) all the prefixes less than /20 are expanded to /20 and inserted in the /20 node.
(2) all the prefixes between /21 to /23 are expanded to /24 and inserted in the /24 node.
(3) all the prefixes greater than /25 are expanded to /32. The prefix is inserted in the /32 node and an additional marker is inserted in the /24 node.

The tree lookup starts from the /24 root node. If the searched IP matches a prefix, the search ends. If the searched IP matches a marker, then it is possible that the IP matches also a prefix in the /32 node, so the algorithm does a second lookup in the right leaf of the tree. If there is no match in /24, a second lookup in the left leaf of the tree is done. The tree requires in the worst case two node accesses, thus providing a fast lookup. However, prefix expansion can lead to a large number of entries and thus to a large memory usage. In this design, using perfect cuckoo filters can significantly reduce the memory needed. To better illustrate the benefits, let us consider several Internet scale routing tables with close to one million prefixes.[7] Table 4 presents the number of prefixes on each routing table and also the number of /24 prefixes that account for the majority of the prefixes.

Table 4: Number of routing tables prefixes

Table	/24	Total
AS20	351455 (58%)	602680
AS6447	529421 (58%)	910419
RIPE RRC00	504440 (58%)	864260

The number of prefixes after expansion for routing table AS6447 are shown in Table 5 as an example. It can be seen that setting the length of the left node to /20 results in the lowest number of prefixes. Similar results were obtained for the other tables. Selecting /20, /24 and /32 as prefix lengths, for /20 we can use an array with size 2^{20}. Instead, for /24, such an array would require 16 million entries. Therefore, we can use a perfect cuckoo filter with $b = 19$ and $f = 5$. Similarly, for /32 we can use a PCF with $b = 17$ and $f = 15$.

[6]Note that the original algorithm in [22] uses much more nodes (one per prefix length) but requires up to five prefix length checks, in our evaluation we restricted the tree to three nodes to have a larger lookup throughput.

[7]The original FIB tables were taken from https://bgp.potaroo.net/and http://data.ris.ripe.net/rrc00/ at different dates. The actual tables used for the experiments are available at https://github.com/pontarelli/perfectCF.

Table 5: Number of prefixes after expansion for the tree nodes for the AS6447 table

Length Left	Left	Root (/24)	Right (/32)	Total
/16	34840	4184784	425079	4644703
/17	74122	3366133	425079	3865334
/18	154901	2660502	425079	3240482
/19	324772	2001507	425079	2751358
/20	668726	1414803	425079	2508608
/21	1359642	1053287	425079	2838008

Those filters would not have false positives and thus are functionally equivalent to an exact match that stores the full prefix. However, storing the full prefixes requires more memory. The detailed analysis of the memory requirements for each solution is summarized in Table 6. The actions for the prefixes are supposed to have eight bits which would be enough to code more than 200 outgoing next hops. It can be seen that the use of PCFs reduces the memory required almost by half. When storing the full prefixes, a cuckoo hash table with 512K buckets of four cells is used for /24 and with 128K buckets for /32. In the first case, each cell has 24+8 bits and in the second 32+8 bits. When using the proposed cuckoo filters, the buckets and cells are the same but have sizes of 6+8 and 16+8 respectively. The results shows a memory savings close to 50% for the three routing tables taken into account.

Table 6: Memory in MB required for the LPM implementation storing the full prefixes (FULL), using PCFs, and using the DIR-24-8 (DIR) scheme for different routing tables

	AS20			AS6447			RIPE RCC		
Node	FULL	PCFs	DIR	FULL	PCFs	DIR	FULL	PCFs	DIR
/20	1.0	1.0	N/A	1.0	1.0	N/A	1.0	1.0	N/A
/24	4.0	1.9	16.0	8.0	3.5	16.0	8.0	3.5	16.0
/32	0.0	0.0	0.0	2.5	1.5	1.3	2.5	1.5	0.1
total	5.0	2.9	16.0	11.5	6.0	17.3	11.5	6.0	16.1
[%]	31%	18%	100%	66%	35%	100%	71%	37%	100%

5 CONCLUSIONS

This paper has presented the Perfect Cuckoo Filter (PCF) a data structure derived from the cuckoo filter that completely eliminates false positives. The proposed PCFs extend the concept of false positive free Bloom filters supporting much larger universes and set sizes, making them applicable in many scenarios. Additionally, PCFs can also be used for <key,value> mapping. A side benefit of PCFs is that their occupancy does not degrade as much as that of traditional cuckoo filters when reducing the number of fingerprint bits. The PCF has been simulated to check that it does avoid false positives and two network functions have been implemented (a blacklist and longest prefix match for IPv4), showing that the use of PCFs can reduce the memory needed by approximately half in both cases.

6 ACKNOWLEDGEMENTS

This work was supported by the ACHILLES project PID2019-104207RB-I00 and the Go2Edge network RED2018-102585-T funded by the Spanish Agencia Estatal de Investigación (AEI) 10.13039/501100011033 and by the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496.

REFERENCES

[1] Austin Appleby. 2012. MurmurHash3, 2012. *URL: https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3. cpp* (2012).

[2] B. Bloom. 1970. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970).

[3] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[4] B. Coskun. 2017. (Un)wisdom of Crowds: Accurately Spotting Malicious IP Clusters Using Not-So-Accurate IP Blacklists. *IEEE Transactions on Information Forensics and Security* 12, 6 (2017), 1406–1417. https://doi.org/10.1109/TIFS.2017.2663333

[5] Benoit Donnet, Bruno Baynat, and Timur Friedman. 2006. Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Proceedings of the 2006 ACM CoNEXT conference.* 1–12.

[6] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo filter: Practically better than Bloom. In *ACM CoNEXT.*

[7] Pankaj Gupta, Steven Lin, and Nick McKeown. 1998. Routing lookups in hardware at memory access speeds. In *Proceedings. IEEE INFOCOM'98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98*, Vol. 3. IEEE, 1240–1247.

[8] S. Z. Kiss, É. Hosszu, J. Tapolcai, L. Rónyai, and O. Rottenstreich. 2018. Bloom Filter with a False Positive Free Zone. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications.* 1412–1420. https://doi.org/10.1109/INFOCOM.2018.8486415

[9] M. Kwon, P. Reviriego, and S. Pontarelli. 2016. A length-aware cuckoo filter for faster IP lookup. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS).* 1071–1072. https://doi.org/10.1109/INFCOMW.2016.7562258

[10] Hyesook Lim, Nara Lee, Jungwon Lee, and Changhoon Yim. 2014. Reducing false positives of a Bloom filter using cross-checking Bloom filters. *Applied Mathematics & Information Sciences* 8, 4 (2014), 1865.

[11] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. 2019. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *IEEE Communications Surveys and Tutorials* 21, 2 (2019), 1912–1949.

[12] Páll Melsted and Jonathan K Pritchard. 2011. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC bioinformatics* 12 (August 2011), 333. https://doi.org/10.1186/1471-2105-12-333

[13] Michael D. Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2018. Adaptive Cuckoo Filters. In *Workshop on Algorithm Engineering and Experiments (ALENEX).*

[14] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144.

[15] Salvatore Pontarelli, Pedro Reviriego, and Juan Antonio Maestro. 2016. Improving Counting Bloom Filter Performance with Fingerprints. *Inform. Process. Lett.* 116, 4 (2016), 304–309.

[16] Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. 2018. EMOMA: Exact Match in One Memory Access. *IEEE Transactions on Knowledge and Data Engineering* 30, 11 (Nov 2018), 2120–2133. https://doi.org/10.1109/TKDE.2018.2818716

[17] Pedro Reviriego and David Larrabeiti. 2020. Denial of Service Attack on Cuckoo Filter Based Networking Systems. *IEEE Communications Letters* 24, 7 (2020), 1428–1432. https://doi.org/10.1109/LCOMM.2020.2983405

[18] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. 2014. The Variable-Increment Counting Bloom Filter. *IEEE/ACM Trans. Netw.* 22, 4 (2014), 1092–1105.

[19] Ori Rottenstreich, Pedro Reviriego, Ely Porat, and S. Muthukrishnan. 2020. Constructions and Applications for Accurate Counting of the Bloom Filter False Positive Free Zone. In *Proceedings of the Symposium on SDN Research* (San Jose, CA, USA) *(SOSR '20).* Association for Computing Machinery, New York, NY, USA, 135–145. https://doi.org/10.1145/3373360.3380845

[20] Martin Stigge, Henryk Plötz, Wolf Müller, and Jens-Peter Redlich. 2006. Reversing CRC – Theory and Practice.

[21] János Tapolcai, András Gulyás, Zalán Heszbergery, József Biro, Péter Babarczi, and Dirk Trossen. 2012. Stateless multi-stage dissemination of information: Source routing revisited. In *2012 IEEE Global communications conference (GLOBECOM).* IEEE, 2797–2802.

[22] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. 2001. Scalable High-Speed Prefix Matching. *ACM Trans. Comput. Syst.* 19, 4 (Nov. 2001), 440–482. https://doi.org/10.1145/502912.502914

[23] C. Wilcox, C. Papadopoulos, and J. Heidemann. 2010. Correlating Spam Activity with IP Address Characteristics. In *2010 INFOCOM IEEE Conference on Computer Communications Workshops.* 1–6. https://doi.org/10.1109/INFCOMW.2010.5466660

APPENDIX: ARTIFACTS

The source code for the implementation of the Perfect Cuckoo Filter has been made available at https://github.com/pontarelli/perfectCF.

The artifact directory contains:

- the source code to build the simulator,
- README.md — a brief description of the repository.

The simulator was developed in C++ and can be compiled with a standard gnu compilation toolchain.