Adaptive Cuckoo Filters

MICHAEL MITZENMACHER, Harvard University
SALVATORE PONTARELLI, Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT)
PEDRO REVIRIEGO, Universidad Carlos III de Madrid

We introduce the adaptive cuckoo filter (ACF), a data structure for approximate set membership that extends cuckoo filters by reacting to false positives, removing them for future queries. As an example application, in packet processing queries may correspond to flow identifiers, so a search for an element is likely to be followed by repeated searches for that element. Removing false positives can therefore significantly lower the false-positive rate. The ACF, like the cuckoo filter, uses a cuckoo hash table to store fingerprints. We allow fingerprint entries to be changed in response to a false positive in a manner designed to minimize the effect on the performance of the filter. We show that the ACF is able to significantly reduce the false-positive rate by presenting both a theoretical model for the false-positive rate and simulations using both synthetic data sets and real packet traces.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; **Data structures design and analysis**; **Sorting and searching**; **Routing and network design problems**; • **Networks → Network algorithms**;

Additional Key Words and Phrases: Cuckoo hashing, Bloom filters, adaptive, Markov chains, false positive rate

ACM Reference format:
Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2020. Adaptive Cuckoo Filters. *J. Exp. Algorithmics* 25, 1, Article 1.1 (March 2020), 20 pages.
https://doi.org/10.1145/3339504

1 INTRODUCTION

Modern networks need to classify packets at high speed. A common function in packet processing is to check if a packet belongs to a set S. As a simple example, if the source of the packet is on a restricted list, then the packet should be subject to further analysis. In some applications, the cost of

Michael Mitzenmacher is supported in part by NSF Grants No. CCF-1563710, No. CCF-1535795, No. CCF-1320231, and No. CNS-1228598. Salvatore Pontarelli is partially supported by the Horizon 2020 project 5G-PICTURE (Grant No. 762057). Pedro Reviriego was with Universidad Antonio de Nebrija at the time of writing the initial version of this article and would like to acknowledge the support of the TEXEO Project No. TEC2016-80339-R funded by the Spanish Ministry of Economy and Competitivity and of the Madrid Community research project TAPIR-CM Grant No. P2018/TCS-4496.

Authors' address: M. Mitzenmacher, Harvard University, School of Engineering and Applied Sciences, 33 Oxford Street, Cambridge MA 02138, USA; email: michaelm@eecs.harvard.edu; S. Pontarelli, Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Via del Politecnico, 1, 00133, Rome, Italy; email: salvatore.pontarelli@uniroma2.it; P. Reviriego, Universidad Carlos III de Madrid, Departamento de Ingenería Telemática, Avenida de la Universidad 30, 28911 Leganés, Madrid, Spain; email: revirieg@it.uc3m.es.

1084-6654/2020/03-ART1.1 $15.00
https://doi.org/10.1145/3339504

such a check can be beneficially reduced via an approximate membership check that may produce false positives but not false negatives; that is, it may with small probability say an element is in a set even when it is not. We first use the approximate membership check, and if we are told the element is not in the set, then we can safely continue regular processing. On a positive response, a more expensive full check may be done to find the false positives, removing them from further analysis. Packets with a source on the restricted list are always correctly identified. This approach speeds the common case by storing the approximate representation of the set S in a small fast memory, and the exact representation of S in a bigger but slower memory that is accessed infrequently.

The well-known example of a data structure for approximate set membership is the Bloom filter [Bloom 1970], which is widely used in networking applications. Many variants and enhancements for Bloom filters have been proposed [Broder and Mitzenmacher 2004]. There are other data structures that also provide approximate membership checks with improved performance, such as the recently proposed cuckoo filter [Fan et al. 2014] that stores fingerprints of the elements in a suitable cuckoo table [Pagh and Rodler 2004], as we describe in more detail in Section 2. A key parameter for such data structures is the false-positive probability, which gives the probability that an element not in the set yields a false positive. A slightly different quantity is the false-positive rate, which gives the fraction of false positives for a given collection of queries (or, in some settings, for a given distribution of the input queries). If an element that gives a false positive is queried many times, then the false-positive rate may be much higher than the false-positive probability, even though in expectation over suitably randomly chosen hash functions the false-positive rate should be equal to the false-positive probability.

In many network applications, the queried elements are repeated. For example, let us consider an application in which we track the packets of a subset of the 5-tuple[1] flows on a link. We may use an approximate membership check for each packet to tell us if the packet belongs to one of the flows that we are tracking, in which case it is subject to more expensive processing. In this application, receiving a packet with 5-tuple x makes it very likely that we will receive other packets with the same 5-tuple value in the near future. Now suppose that a packet with 5-tuple x returns a false positive on the approximate membership check. It is clear that we would like to adapt the approximate membership check structure so that x will not return a false positive for the following packets of the same flow. Theoretically, the false positive responses could be reduced to one per flow, lowering the false-positive rate.

Such adaptation cannot be easily done for the existing approximate membership check structures [Chazelle et al. 2004; Bender et al. 2018]. A variation of a Bloom filter, referred to as a retouched Bloom filter, removes false positives (by changing some bits in the filter to 0), but can create false negatives by doing so [Donnet et al. 2006]. Other options include changing the number of hash functions according to the likelihood of an element being in a set [Zhong et al. 2008; Bruck et al. 2006], but this approach requires additional offline information and computation that does not appear natural for many settings. In all of these cases, removing the false positives significantly changes the nature of the data structure.

We introduce the adaptive cuckoo filter (ACF), an approximate membership check structure that is able to selectively remove false positives *without* introducing false negatives. The scheme is based on cuckoo hashing [Pagh and Rodler 2004] and uses fingerprints, following standard cuckoo filters [Fan et al. 2014]. In contrast with the cuckoo filter, where movements in the cuckoo table can be based solely on the fingerprints, here we assume we have the original elements available, although they can be in a slower, larger memory. When inserting a new element, existing

[1] The 5-tuple is commonly used to refer to the source and destination IP addresses and ports and the protocol field in the packet header.

elements can be moved. Our enhancement is to allow different hash functions for the fingerprints, which allows us to remove and reinsert elements to remove false positives without affecting the functionality of the filter. The removal of false positives is almost as simple as a search operation, and does not substantially impact performance. Instead, the impact is felt by having a slightly more complex insertion procedure, and requiring slightly more space than a standard cuckoo filter to achieve the same false-positive probability. Of course, since the ACF is adaptive, it generally achieves a better false-positive rate than a standard cuckoo filter with the same space, and our goal here is to improve the false-positive rate (as opposed to the false positive probability).

We provide a theoretical model that provides accurate estimates for ACF performance. We validate ACF performance through simulations using synthetic data and real packet traces. The results confirm that the ACF is able to reduce false positives, making it of interest for many networking applications.

Before beginning, we emphasize that the ACF is not a general replacement for a Bloom filter or cuckoo filter, but an optimization that is particularly well-suited to the setting when a pre-filter is desired to avoid unnecessary accesses to memory for data that is not stored there. There are many applications where the original, complete data is required in any case, and the filter is used as a first stage screen to prevent the need to access slower memory to check the original data. For example, for both whitelists and blacklists, a pre-filter could make lookups much more efficient, but all proposed positives would be checked against the original data. Even if only an approximate set membership data structure without the original data is required, in some architectures the memory cost of a slower and larger memory to hold the original data may be relatively inexpensive. However, we note that our need for storing the original data might make the use of ACFs unsuitable for many applications.

The rest of the article is organized as follows. Section 2 briefly reviews the data structures on which the ACF is based, namely, cuckoo hash tables and cuckoo filters. Section 3 introduces the ACF, describing its implementation, and providing a model of its expected performance. The ACF is evaluated in Section 4, using both randomly generated packets and real packet traces to show its effectiveness. We conclude with a summary and some ideas for future work.

2 PRELIMINARIES

We provide a brief overview of cuckoo hashing [Dietzfelbinger and Weidling 2007; Fotakis et al. 2005; Pagh and Rodler 2004] and cuckoo filters [Fan et al. 2014].

2.1 Cuckoo Hashing

Cuckoo hash tables provide an efficient dictionary data structure [Dietzfelbinger and Weidling 2007; Fotakis et al. 2005; Pagh and Rodler 2004]. A cuckoo hash table can be implemented as a collection of d subtables composed of b buckets each, with each bucket having c cells. The hash table uses d hash functions $h_1, h_2, \ldots, h_d$, where the domain is the universe of possible input elements and the range is $[0, b)$. We assume that all functions are independent and uniform; such assumptions are often reasonable in practice [Mitzenmacher and Vadhan 2008]. A given element x can be placed in bucket $h_i(x)$ in the ith subtable; each element is placed in only one bucket. The placement is done so that there are at most c elements in any bucket. This limitation can require moving elements among their choices of buckets when a new element is inserted. Values associated with elements can be stored with them if that is required by the application; alternatively, pointers to associated values kept in an external memory can also be stored. The structure supports the following operations:

Lookup: Given an element x, the buckets given by the $h_i(x)$ are examined to see if x is in the table.

Insertion: Given an element x, we first check that x is not already in the table via a lookup. If not, then we sequentially examine the buckets $h_i(x)$. If there is empty space, then x is inserted into the first available space. If no space is found, then a value j is chosen independently and uniformly at random from $[1, d]$, and x is stored in the jth subtable at bucket $h_j(x)$. This displaces an element y from that bucket, and then the insertion process is recursively executed for y.

Deletion: Given an element x, it is searched for via a lookup; if it is found it is removed.

Cuckoo hash tables are governed by a threshold effect; asymptotically, if the load, which is the number of elements divided by the number of cells, is below a certain threshold (which depends on c and d), then with high probability all elements can be placed.

The above description of insertion is somewhat incomplete, as there is room for variation. For example, the displaced element y is often chosen uniformly at random from the bucket, and it is often not allowed to put y immediately back into the same bucket. The description also does not suggest what to do in case of a failure. Generally after some number of recursive placement attempts, a failure occurs, in which case elements can be re-hashed, or an additional structure referred to as a stash can be used to hold a small number of items that would not otherwise be placed [Kirsch et al. 2009].

An alternative implementation of cuckoo hashing uses a single table of buckets, so each hash function can return any of the buckets. The two alternatives provide the same asymptotic performance in terms of memory occupancy.

As previously stated, the asymptotic load threshold depends on the values of c and d, the number of cells per bucket and the number of hash functions. In practice, even for reasonably sized systems one obtains performance close to the asymptotic load threshold. Two natural configurations include $d = 2$ and $c = 4$, which gives a threshold over 98%, and $d = 4$ and $c = 1$, which gives a threshold of over 97%. These configurations give high memory utilization, while requiring a small memory bandwidth [Erlingsson et al. 2006]. The decision for what configuration to use may depend on the size of the item being stored and the natural cell size for memory.

Note that if subtables can be put on distinct memory devices, then lookup operations can possibly be completed in one memory access cycle by searching subtables in parallel [Pontarelli et al. 2016].

2.2 Cuckoo Filters

A cuckoo filter is an approximate membership check data structure based on cuckoo hash tables [Fan et al. 2014]. Instead of storing set elements, a cuckoo filter stores fingerprints of the elements, obtained using an additional hash function. In this way, it uses less space, while still being able to ensure a low false-positive probability. In the suggested implementation, each element is hashed to $d = 2$ possible buckets, with up to $c = 4$ elements per bucket.

To achieve small space, the cuckoo filter does not store the original elements. A difficulty arises in moving the fingerprints when buckets are full, as is required by the underlying cuckoo hash table. The cuckoo filter uses the following method, referred to as partial-key cuckoo hashing. If the fingerprint of an element x is $f(x)$, then its two bucket locations will be given by $h_1(x)$ and $h_1(x) \oplus h_2(f(x))$. Notice that, given a bucket location and a fingerprint, the other bucket location can be determined by computing $h_2(f(x))$ and xoring the result with the current bucket number.

The structure supports the following operations, where in what follows we assume an insertion is never performed for an element already in the structure:

Lookup: Given an element x, compute its fingerprint $f(x)$, and the bucket locations $h_1(x)$ and $h_1(x) \oplus h_2(f(x))$. The fingerprints stored in these buckets are compared with $f(x)$. If any fingerprints match $f(x)$, then a positive result is returned, otherwise a negative result is returned.

Insertion: Given an element x, compute the two bucket locations and the fingerprint $f(x)$. If a cell is open in one of these buckets, then place $f(x)$ in the first available open cell. Otherwise, a fingerprint $f(y)$ in one of the buckets is displaced, and then recursively placed, in the same manner as in a cuckoo hash table. Here, we use that both buckets for y can be determined from $f(y)$ and the bucket from which $f(y)$ was displaced.

Deletion: Given an element x, it is searched for via a lookup. If the fingerprint $f(x)$ is found in one of the buckets, then one copy of the fingerprint is removed.

The false-positive probability of a cuckoo filter can be upper bounded as follows. If a bits are used for the fingerprints and the load of the hash table is ℓ, then the false-positive probability will be approximately $8\ell/2^a$. This is because on average the search will compare against 8ℓ fingerprints (assuming 2 choices of buckets with 4 cells per bucket), with each having a probability 2^{-a} of yielding a false positive.

3 ADAPTIVE CUCKOO FILTER CONSTRUCTION

Before beginning, we note that pseudocode for our algorithms appear in the Appendix.

Our proposed ACF stores the elements of a set S in a cuckoo hash table. A replica of the cuckoo hash table that stores fingerprints instead of full elements is constructed, and acts as a cuckoo filter. The key difference from the cuckoo filter is that we do *not* use partial-key cuckoo hashing; the buckets an element can be placed in are determined by hash values of the element, and not solely on the fingerprint. This is because we want the ability to change the fingerprint to be adaptive; also, as we are aiming for adaptivity, we have the data elements available to check for and remove false positives. While this may mean some performance loss in lookups and insertions compared to a non-adaptive cuckoo filter (depending on the implementation), the goal here is to significantly reduce the false-positive rate. To be clear, the filter uses the same hash functions as the main cuckoo hash table; the element and the fingerprint are always placed in corresponding locations. Using the filter, false positives occur when an element not in the set has the same fingerprint as an element stored in one of the positions accessed during the search. A false positive is detected by examining the cuckoo hash table when a positive result is found; if the element is not found in the corresponding location—specifically, in the same bucket and same position within that bucket as the corresponding fingerprint—a false positive has occurred, and moreover we know what element has caused the false positive. As we explain in Sections 3.1 and 3.2, to remove the false positive, we need to change the fingerprint associated with that element, by using a different fingerprint function. We suggest two ways of accomplishing this easily below.

Before describing the structure in more detail, we define the parameters that will be used.

- The number of tables used in the cuckoo hash: d.
- The number of cells per bucket: c.
- The total number of cells over all tables: m.
- The number of buckets per table: $b = m/(d \cdot c)$.
- The occupancy or load of the ACF: ℓ.
- The number of bits used for the fingerprints: a.
- The number of bits used for the hash selector: s.

To see the potential benefits of the ACF, let us consider its behavior over a window of time. Let us suppose that a fraction p_1 of operations are lookups, p_2 are insertions, and p_3 are deletions. Let us further suppose that of the lookup operations, a fraction q_1 are true positives, and the remaining fraction $1 - q_1$ have a false-positive rate of q_2. If we assume costs c_I, c_D, c_P, and c_F for the inserts, deletes, positive lookups, and negative lookups, then we find the total cost is

$$p_1((q_1 + (1 - q_1)q_2)c_P + (1 - q_1)(1 - q_2)c_F) + p_2c_I + p_3c_D.$$

In the types of implementations we target, we expect p_1 to be large compared to p_2 and p_3, so that the cost of lookups dominate the costs of insertions and deletions. We also expect q_1 to be small, and c_F to be much less than c_P, so a filter can successfully and at low cost prevent lookups to slow memory. In this case the dominant cost corresponds to the term $p_1(q_1 + (1 - q_1)q_2)c_P$, and as q_1 is small, the importance of minimizing the false-positive rate q_2 is apparent.

We now present variants of the ACF and discuss their implementation.

3.1 ACF for Buckets with One Cell

We present the design of the ACF when the number of tables is $d = 4$ and the number of cells per bucket is $c = 1$. This configuration achieves similar utilization as $d = 2$ and $c = 4$ and requires less memory bandwidth [Erlingsson et al. 2006] at the cost of using more tables. The ACF consists of a filter and a cuckoo hash table, with both having the same number of tables and numbers of cell per bucket, so that there is a one-to-one correspondence of cells between the two structures. The ACF stores only a fingerprint of the element stored in the main table.

To implement the ACF, we use a small number of bits within each bucket in the ACF filter to represent a choice of hash function for the fingerprint, allowing multiple possible fingerprints for the same element. We use s hash selector bits, which we denote by α, to determine which hash function is used to compute the fingerprint stored in that bucket. We refer to the fingerprint hash functions as $f_0, f_1, \ldots, f_{2^s-1}$. For example, by default we can set $\alpha = 0$ and use $f_0(x)$. If we detect a false positive for an element in that bucket then we increment α to 1 and use $f_1(x)$ (for convenience, we think of the s bits as representing values 0 to $2^s - 1$). In most cases this will eliminate the false positive. On a search, we first read the hash selector bits, and then we compute the fingerprint using the corresponding fingerprint function before comparing it with the stored value. The downside of this approach is we are now using a small number of additional bits per cell to represent the choice of hash function, increasing the overall space required.

In more detail, to remove a false positive, we increment α (modulo 2^s) and update the fingerprint on the filter accordingly (sequential selection is slightly better than random selection as it avoids picking a value of α that recently produced another false positive). We verify that the adaptation removes the false positive by checking that the new fingerprint is different from the new fingerprint for the element that led to the false positive. For example, if x created a false positive with an element y stored on the ACF when $\alpha = 1$, then we can increment α and check that $f_2(x) \neq f_2(y)$; if the fingerprints match, then we can increment α again. This refinement is not considered in the rest of the article as the probability of removing the false positive is already close to one $(1 - 1/2^a)$, and we want to keep the adaptation procedure as simple as possible.

The insertion and deletion procedures use the standard insertion or deletion methods for the cuckoo hash table, with the addition that any insertions, deletions, or movement of elements are also performed in the filter to maintain the one-to-one relationship between elements and fingerprints.

To perform a search for an element x, the filter is accessed and buckets $h_1(x), \ldots, h_4(x)$ are read. From each bucket the corresponding fingerprint and α value is read and the fingerprint value is compared against $f_\alpha(x)$. If there is no match in any table, the search returns that the element is not found. If there is a match, the cuckoo hash table is accessed to see if x is indeed stored there. If there is a match but x is not found, then a false positive is detected. The search in the filter of an

ACF is similar to a search in a cuckoo filter and should have a similar false-positive probability; however, we can reduce the false-positive rate. We emphasize again that the false-positive probability corresponds to the probability that a "fresh," previously unseen element not in the set yields a false positive. Instead, the false-positive rate corresponds to the fraction of false positives over a given sequence of queries, which may repeatedly query the same element multiple times.

To model the behavior of the ACF, we will define a Markov chain that describes the evolution of α in a single bucket of the filter, with changes to α triggered by the occurrence of false positives. We assume that we have ζ elements that are not in the hash table that are to be checked against a given bucket; this is the set of potential false positives on that bucket. We assume that requests from the ζ elements are generated independently and uniformly at random; while arguably this assumption is not suitable in many practical situations, it simplifies the analysis, and is a natural test case to determine the potential gains of this approach. (More skewed request distributions will generally lead to even better performance for adaptive schemes.) We further assume that the hash table is stable, that is no elements are inserted and deleted, in the course of this analysis of the false-positive rate. Finally, we first perform the analysis from the point of view of a single bucket; we then use this to determine the overall expected false-positive rate over all buckets. We analyze the false-positive rate assuming that these are the only queries; true matches into the hash table are not counted in our analysis, as the frequency with which items in the table are queried as compared to items not in the table is application-dependent.

Without loss of generality, we may assume that an element x is in the hash table, and the fingerprint value $f_0(x)$ is located in the cell. (If there is no element in the cell, then there are no false positives on that cell.) The number of elements that provide a false positive on the first hash function is a binomial random variable $\mathrm{Bin}(\zeta, 2^{-a})$, which we approximate by a Poisson random variable with mean $\zeta 2^{-a}$ for convenience [Mitzenmacher and Upfal 2017, Chapter 5]. We refer to this value by the random variable Z_0, and similarly use Z_j to refer to the corresponding number of elements that provide a false positive for the $(j+1)$st hash function. Because we are treating the Z_j as Poisson random variables, through standard results on Poisson splitting it is a reasonable approximation to treat the Z_j as independent for practical values of ζ and a, and we do so henceforth [Mitzenmacher and Upfal 2017, Chapter 5].

We can now consider a Markov chain with states $0, 1, \ldots, 2^s - 1$; the state i corresponds to $\alpha = i$, so the cell's fingerprint is $f_i(x)$. The transitions of this Markov chain are from i to $i+1$ modulo 2^s with probability Z_i/ζ; all other transitions are self-loops. Note that if any of the Z_i are equal to zero, we obtain a finite number of false positives before finding a configuration with no additional false positives for this set of ζ elements; if all Z_i are greater than zero, then we obtain false positives that average to a constant rate over time. Using this chain, we can determine the false-positive rate.

Specifically, let us suppose that we run for n queries. Let $\mu = \zeta 2^{-a}$ be the mean of our Poisson random variables, so that $e^{-\mu}$ is our approximation for the probability that there are no false positives for a given hash function. We first note that for any value $k < 2^s$, we have a probability

$$P_{stop}(k) = (1 - e^{-\mu})^k e^{-\mu}$$

to finish in a configuration with no additional false positives after the bucket is triggered by k false positives. For this case the false-positive rate is simply k/n, if we suppose that n is sufficiently large to trigger the system up to the final state.

For the case where all hash functions have at least one element that yields a false positive, so that all the Z_i are greater than 0, the expected number of queries to complete a cycle from the first hash function and back again, given the Z_i values, is given by the following expression:

$$\sum_{j=0}^{2^s-1} \frac{\zeta}{Z_j}.$$

Over this many queries, we obtain 2^s false positives.

The overall asymptotic false-positive rate $F(\zeta, n)$ for a bucket that stores an element and to which $\zeta > 0$ elements not in the hash table map can therefore be expressed as

$$F(\zeta, n) = \sum_{i=1}^{2^s-1} (1 - e^{-\mu})^i e^{-\mu} \frac{i}{n} + \sum_{Z_0, Z_1, \ldots, Z_{2^s-1} > 0} \left(\prod_{i=0}^{2^s-1} \frac{e^{-\mu} \mu^{Z_i}}{Z_i!} \right) \frac{2^s}{\sum_{i=0}^{2^s-1} \frac{\zeta}{Z_i}}.$$

Notice that this expression is a good approximation even for reasonably small values of n. Further, when n is large, the first term will be relatively small compared to the second. With this expression, we can compute $F(\zeta, n)$ to suitable accuracy for small s by simply computing the terms for enough combinations of Z_i values to cover the space of possibilities so only a negligible probability of events is discarded; alternatively, the expression can be approximated via Monte Carlo simulation.

The previous analysis considered a bucket in isolation. We now move to consider the false-positive rate for an entire table that has bd buckets with an occupancy ℓ. The number of elements ζ that map to each bucket is a binomial random variable, asymptotically well-approximated by a Poisson random variable. Let us assume that there are A elements not stored that are searched for in the table; then ζ can be approximated by a Poisson random variable with mean $e_b = A/b$. Over a collection of N queries to the table, the total number of queries j that map to a bucket with ζ items is itself a binomial random variable, asymptotically well-approximated by a Poisson random variable with mean $N\zeta/A$. We conclude that the (asymptotic, approximate) average false-positive rate F over all buckets is

$$F = bd\ell \sum_{\zeta > 0} \frac{e^{-e_b} e_b{}^{\zeta}}{\zeta!} \sum_{j \geq 0} \frac{e^{-\zeta N/A} (\zeta N/A)^j F(\zeta, j)(j/N)}{j!}.$$

While we do not provide a full proof here, we note that standard concentration results can be used to show that in this model the false-positive rate over all the buckets is close to the expected false-positive rate calculated above. Such concentration means we should not see large variance in performance over instantiations of the ACF structure in this setting.

Although we have looked at a model where each query is equally likely to come from each of the ζ flows, so each has approximately the same size n_e, we could generalize this model to settings where the ζ flows correspond to a small number of types (or rates), and then consider similar calculations based on the number of false positives of each rate. Intuitively, the setting we have chosen where each flow is equally likely to be queried at each step is the worst case for us, as it leads to the fastest cycling through the 2^s fingerprint hash functions in the case that each fingerprint has at least one false positive. This will be confirmed by the results presented in the evaluation section.

3.2 ACFs with One Bucket and Varying Fingerprint Ranges

Thus far, we have described our one cell ACF as having all fingerprints of the same length; that is, the bucket stores $a + s$ bits, where s bits keep track of the state, and a bits keep track of the fingerprint. We could, alternatively, think of all the bits in the bucket as being the fingerprint, and divide the range of fingerprints among the hash functions unequally. We may naturally want to use unequal ranges of fingerprints, because we expect most cells may only have a small number of possible false positives. This suggests, for example, using most of the available fingerprint space on the first fingerprint hash function; if no first false positive occurs, then we need not worry about further false positives. If we are willing to keep a table of offsets, giving the first value associated with each hash function, then we can use any number of hash functions and perform any split we wish. We describe this setting in this general way, and then discuss specific natural implementations.

Our previous analysis framework translates to unequal fingerprint ranges readily, with small changes. Previously, we used Z_j to refer to the Poisson random variable corresponding to the number of elements that provide a false positive for the $(j+1)$st hash function, and the Z_j were identically distributed. Now the Z_j are not necessarily identically distributed; instead, Z_j will have mean $\mu_j = \zeta\kappa_j$, where κ_j is the fraction of fingerprints associated with the $(j+1)$st hash function. Let H be the number of hash functions. Then, following the same analysis as previously and generalizing the expression for $F(\zeta, n)$, we find the overall asymptotic false-positive rate $F'(\zeta, n)$ for a bucket is now expressed as

$$F'(\zeta, n) = \sum_{i=0}^{H-2} \prod_{j=0}^{i} (1 - e^{-\mu_j}) e^{-\mu_{j+1}} \frac{i+1}{n} + \sum_{Z_0, Z_1, \ldots, Z_{H-1} > 0} \left(\prod_{i=0}^{H-1} \frac{e^{-\mu_i} \mu_i^{Z_i}}{Z_i!} \right) \frac{H}{\sum_{i=0}^{H-1} \frac{\zeta}{Z_i}}.$$

The (asymptotic, approximate) average false-positive rate F' over all buckets has the same formula as for F previously, with $F'(\zeta, j)$ replacing $F(\zeta, j)$ in the formula.

A natural approach using this generalization that we suggest and consider in our experiments requires some small additional logic but is quite simple to implement. We use a parameter r to set the size of the fingerprint hash functions. Here, we use $2^s + 1$ hash functions, and the bucket stores $a + s$ bits. The first fingerprint function f_0 has range size $R_0 = 2^{(a+s)} \cdot \frac{(2^r-1)}{2^r}$. The remaining fingerprint functions $f_1, f_2, \ldots, f_{2^s}$ functions have range size $R_i = 2^{(a-r)}$.

To store an element, if f_0 is used, then a value between 0 and $2^{(a+s)} \cdot \frac{(2^r-1)}{2^r} - 1$ is generated and stored. Otherwise, the first r bits are set to one, the next s bits are used to identify the fingerprint hash function $f_1, f_2, \ldots, f_{2^s}$, and an $a - r$ bit fingerprint is stored on the lower bits. On a look up, the first r bits of the cell are checked. If they are not r ones, then the element has been stored with f_0. Otherwise, the element has been stored with one of the other hash functions, as given by the following s bits of the cell.

Our suggested approach only gives the first hash function a larger range than other hash functions, but our experiments show it can yield improvements over the approach that uses the same range size for all hash functions.

3.3 ACF for Buckets with Multiple Cells

We now examine the case where the number of tables is $d = 2$ and the number of cells per bucket is $c = 4$, the case that was studied originally in constructing cuckoo filters [Fan et al. 2014]. The ACF again consists of a filter and a cuckoo hash table with a one-to-one correspondence of cells between the two structures; the ACF holds the fingerprint of the element stored in the main table. To enable adaptation to reduce the false-positive rate as explained below, the hash functions used for the fingerprint are different for each of the cell locations, so that in our example there will be four possible fingerprints for an element, $f_1(x)$, $f_2(x)$, $f_3(x)$, and $f_4(x)$, each corresponding to a cell (1, 2, 3, and 4, respectively) in the bucket.

The insertion and deletion procedures use the standard insertion or deletion methods for the cuckoo hash table, with the addition that any changes are matched in the filter to maintain the one-to-one relationship between elements and fingerprints. Note that if an element is moved via insertion to a different cell location, its fingerprint may correspondingly change; if an element x

moves from cell 1 to cell 2 (either in the same bucket or a different bucket), its fingerprint changes from $f_1(x)$ to $f_2(x)$.

The search is similar to the case described in the previous subsection. When there is a match in the filter but not the main table, a false positive is detected.

We describe how this ACF responds when a false positive occurs. Suppose a search for an element x yields a false positive; the fingerprints match, but element y is stored at that location in the cuckoo hash table. For convenience let us assume that the false positive was caused by the fingerprint stored in the first cell of the first table, and that there is another element z in the second cell on the first table. Then, we can swap y and z in the ACF, keeping them in the same bucket, so that now instead of $f_1(y)$ and $f_2(z)$, the bucket will hold $f_1(z)$ and $f_2(y)$ in the first two cells. If subsequently we search again for x, then in most cases $f_1(z)$ and $f_2(y)$ will be different from $f_1(x)$ and $f_2(x)$, and therefore the false positive for x will be removed. If we search for y or z, then we still obtain a match. As a result of the change, however, we might create false positives for other elements. More precisely, when a lookup provides a positive response but the cuckoo hash table does not have the corresponding element, there was a false positive and the filter adapts to remove it by randomly selecting one of the elements in the other $c - 1$ cells on the bucket and performing a swap.[2]

To model the ACF for buckets with multiple cells, we define a Markov chain that takes into account the evolution of a bucket as cells are swapped. As before, we assume fingerprints of a bits, with ζ elements creating potential false positives, and each request being independently and uniformly chosen from this set of elements. Without loss of generality we may assume that elements w, x, y, and z are in the bucket being analyzed, initially in positions 1, 2, 3, and 4, respectively. Let $Z_{w,1}$ be the number of elements that yield a false positive with element w when it is in position 1, that is when the corresponding value is $f_1(w)$, and similarly for the other elements and positions. Here, since some buckets may not be full, we can think of some of these elements as possibly being "null" elements that do not match with any of the ζ potential false positives. For non-null elements, as before, $Z_{w,1}$ is distributed as a binomial random variable $\text{Bin}(\zeta, 2^{-a})$, approximated by Poisson random variable with mean $\mu = \zeta 2^{-a}$, and we treat the random variables as independent. We use $\hat{Z}$ to refer to a 16-dimensional vector of values for $Z_{w,1}$, $Z_{w,2}$, and so on.

In this setting, we consider a Markov chain with states being the 24 ordered tuples corresponding to all possible orderings of w, x, y, and z. The transition probability from for example the ordering (w, x, y, z) to (x, w, y, z) is given by

$$\frac{Z_{w,1}}{3\zeta} + \frac{Z_{x,2}}{3\zeta};$$

that is, with probability $Z_{w,1}/\zeta$, the query is for an element that gives a false positive against w, and then with probability $1/3$ the elements x and w are swapped, and similarly for the second term in the expression. Any transition corresponds to either a swap of two elements or a self-loop. It is possible an absorbing state can be reached where there is no transition that leaves that state—that is, there are no false positives in the given state—if there is some permutation where the corresponding Z variables for transitions out of that state are all 0. However, there may be no such absorbing state, in which case the false positives average to a constant rate over time. (This is similar to what we have previously seen in the ACF with one cell per bucket.) While there is no simple way to write an expression for the false-positive rate for a bucket based on $\hat{Z}$, as there was in the setting of one cell per bucket, the calculations are straightforward.

Given values for a vector of $\hat{Z}$, we can determine the false-positive rate by determining either the expected number of transitions to reach the absorbing state, or the expected overall rate of

[2] Actually, there is a small probability that a single item matches more than one fingerprint on a query. Also in this case, the ACF performs just one swap operation, thus correcting just one fingerprint collision. The next time that the element is queried another swap operation will be triggered to solve the second collision and so forth.

false positives, depending on the values. Thus, in theory, we can determine the overall expected false-positive rate, similarly to the case of buckets with one cell. That is, we can calculate the probability of each vector $\hat{Z}$, and determine either the probability distribution on the number of false positives before reaching an absorbing state (if one exists), or the expected time between false positives (which can be derived from the equilibrium distribution if there is no absorbing state), and take the corresponding weighted sum. In practice, we would sum over vectors $\hat{Z}$ of sufficient probability to obtain an approximation good to a suitable error tolerance; however, we find the number of relevant vectors $\hat{Z}$ is very large, making this computation impractical.

Instead, we suggest a sampling-based approach to determine the false-positive rate. Vectors $\hat{Z}$ are sampled a large number of times with the appropriate probability, with the corresponding false-positive rate for a bucket determined for each sampled $\hat{Z}$, to obtain a reasonable approximation. This sampling process must take into account that some cells may be left unoccupied. Based on this, we can calculate an estimate for the expected false-positive rate for a single bucket to which ζ elements map, and then find the overall false-positive rate by taking appropriate weighted averages, again similarly to the case of one cell per bucket. We note that this sampling process remains much more efficient in terms of space and memory than simulating the entire data structure, allowing for more accurate performance estimates with smaller resources.

3.4 Implementation Considerations

Both variants of the ACF we have proposed are simple, and straightforward to implement. The search operation is similar to that of a cuckoo hash table, with the cuckoo filter acting as a prefilter. Indeed, if there is no false positive, the cuckoo filter will determine what bucket the element is in, so that only one bucket in the cuckoo hash table will need to be checked for the element. On a false positive, after checking the table the additional work to modify the table to adapt requires only a few operations.

Both variants of the ACF require the computation of different hash functions to obtain the fingerprints. This adds some complexity to the implementation. However, the overhead is expected to be small as hash functions can be computed in very few clock cycles (e.g., CLHASH is able to process a string of 8 bytes in less than 10 clock cycles [Lemire and Kaser 2016]) in modern processors exploiting the last x86 ISA extensions [Intel 2011] or implemented in hardware with low cost. In both variants, it may be possible to compute multiple fingerprints of an element at once, since most hash functions return either 32 or 64 bits, and these bits can be split into multiple fingerprints. In a hardware or modern processor implementation, hash computations are commonly much faster than memory accesses. Therefore, the time required to perform a lookup in the filter is typically dominated by the memory accesses to the filter cells. The number of such memory accesses is the same for both the cuckoo filter and the ACF. Therefore, we can argue that they will achieve similar lookup speeds.

We have assumed thus far that adaptation, or the procedure to remove a false positive, is done every time a false positive is found. This is straightforward, but might not be optimal in some settings, depending on the costs of changing the ACF cuckoo hash table and filter. We leave consideration of alternative schemes that do not try to modify the ACF on each false positive for future work. For our algorithms, we have not considered the additional possibility of moving items to another bucket when false positives are detected, although this is possible with cuckoo hashing. Our motivation is in part theoretical and in part practical. On the practical side, such movements would potentially be expensive, similar to additional cuckoo hash table insertions. On the theoretical side, analyzing the false-positive rate when items are moving in the table seems significantly more difficult. Again, considerations of such schemes would be interesting for future work.

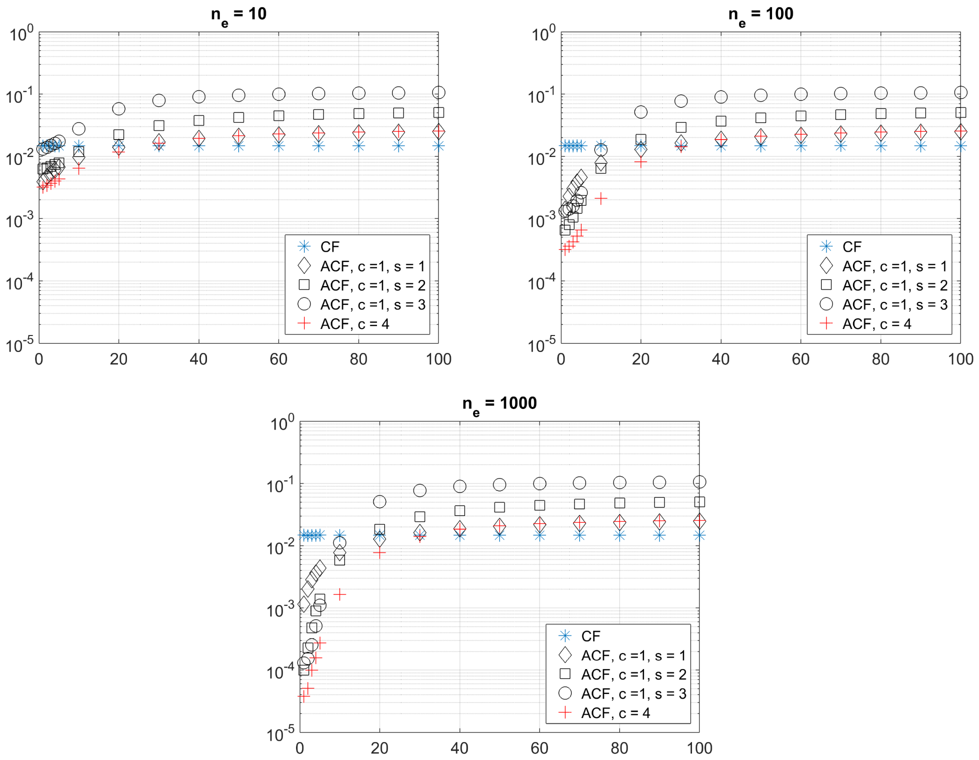

Fig. 1. False-positive rate versus A/S ratio for the different configurations when using 8 bits per cell.

4 EVALUATION

To evaluate the performance of our proposed ACF implementations, we have simulated them both with queries that are generated according to different parameters and also with real packet traces.[3] In the first case, the goal is to gain understanding about how the performance of the ACF depends on the different parameters while in the second, the goal is to show that the ACF can provide reductions on the false-positive rate for real applications. The results are also compared with those of the analytic estimates presented in Section 3.

We note that we do not compare with possible alternatives such as the retouched Bloom filter or varying the number of hash functions, because they are not directly comparable with the ACF; the retouched Bloom filter introduces false negatives, and varying the number of hash functions requires additional offline information. The original cuckoo filter is the best natural alternative with which to compare.

4.1 Simulations with Generated Queries

The first set of simulations aims to determine the effect of the different parameters on the performance of the ACF. The ACF is filled with randomly selected elements up to 95% occupancy; let S be the number of elements stored in the ACF. Then A elements that are not in the ACF are randomly generated. The queries are then generated by taking randomly elements from A with all elements having the same probability and the false-positive rate is measured. We consider the following configurations:

[3] The code used to simulate the ACF is available at https://github.com/pontarelli/ACF.

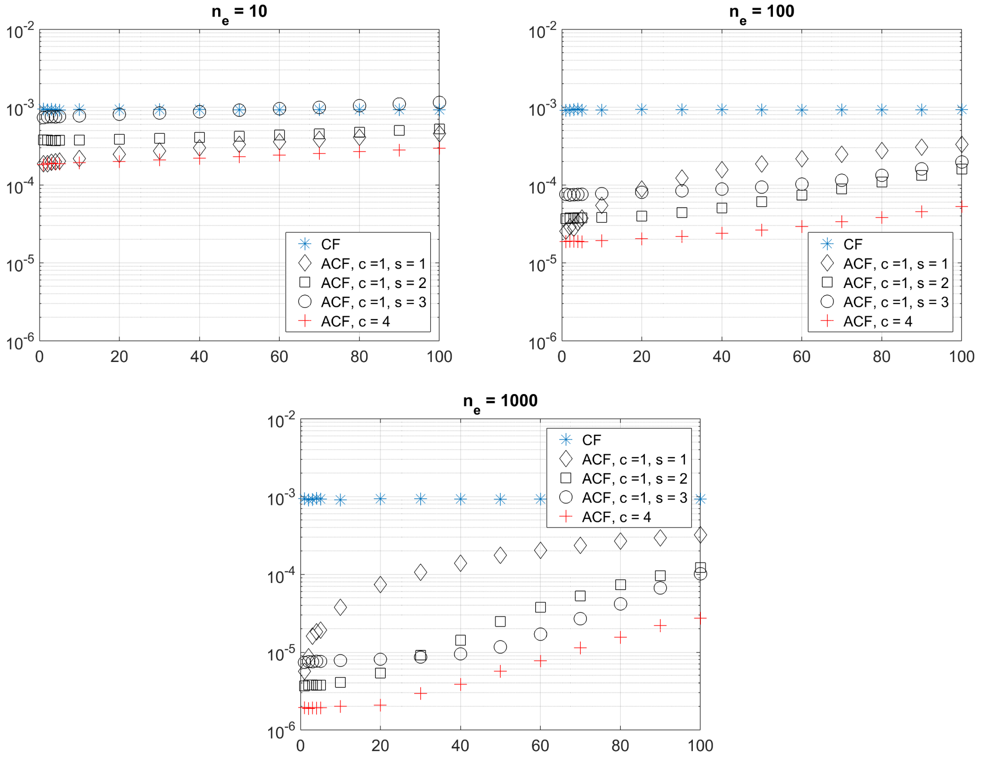

Fig. 2. False-positive rate versus A/S ratio for the different configurations when using 12 bits per cell.

- ACF with $d = 4, c = 1$ (first variant) and $s = 1, 2, 3$.
- ACF with $d = 2, c = 4$ (second variant).
- Cuckoo filter (CF) with $d = 4, c = 1$.

We note that we only provide results for the cuckoo filter with $d = 4,\ c = 1$ and not for the cuckoo filter with $d = 2,\ c = 4$ in our comparisons as the former achieves a significantly better false-positive rate than the latter. To provide an initial understanding of the ACF performance, we fix the same probability of being selected for a query for all elements, so that on average each has n_e queries. For the first variant, we set the number of buckets to 32,768 for each table, so the overall number of cells is 131,072. For the second variant there are two tables and each table has 16,384 buckets. The overall number of cells is 131,072 also in this case. The number of bits for the fingerprint a is set to 8, 12 and 16 so that the standard cuckoo filter with $d = 4, c = 1$ gets a false-positive rate of approximately 1.5%, 0.1%, and 0.006% at 95% occupancy. The same number of bits per cell is used for the different ACF configurations. This means that for the first variant the number of fingerprint bits are $8 - s, 12 - s$, and $16 - s$. The ACFs are adapted on every false positive. The value of n_e is set to 10, 100, and 1000 and the experiment is repeated for several ratios of elements not in the set (A) to elements in the set (S) that appear in the queries ranging from 1 to 100. The A/S ratios used in the experiment are $\{1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.

Figures 1, 2, and 3 present results for 8, 12, and 16 bits per cell, respectively. The average over 10 trials is reported. We observe, as expected, that the false-positive rates for the ACFs are better for lower A/S ratios, and that the performance of the ACF improves when the number of queries per element n_e increases. We also see that the second variant of the ACF ($d = 2, c = 4$) provides

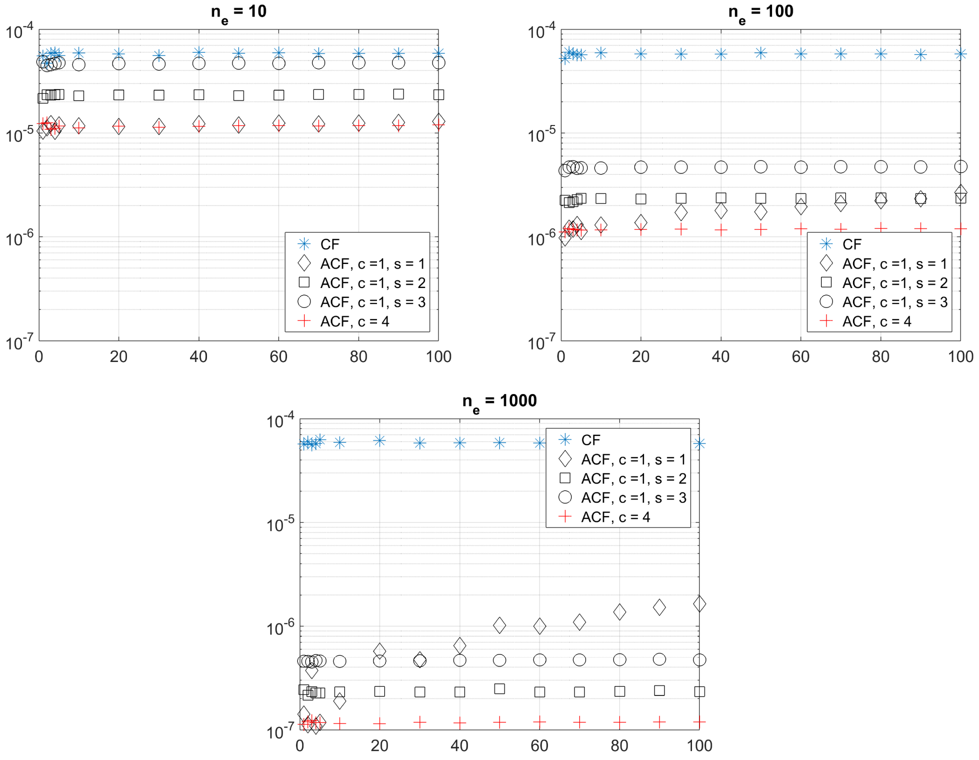

Fig. 3. False-positive rate versus A/S ratio for the different configurations when using 16 bits per cell.

better performance than the first variant ($d = 4, c = 1$) in almost all cases. This result backs the intuition that the second variant would perform better as it does not require any bits to track the hash function currently selected in the cell.

Focusing on a comparison with a standard cuckoo filter, the ACFs outperform the cuckoo filter for all A/S ratios considered when the number of bits per cell is 12 or 16. (Here, we count only the data in the filter part, and do not count the additional memory for the original data.) When the number of bits per cell is 8, the ACF has a lower false-positive rate only when the A/S ratio is small. In configurations where A/S is large and the number of bits per cell is small, there are a large number of false positives, causing many cells to fail to adapt sufficiently, as they simply rotate through multiple false positives. In contrast, the reductions in the false-positive rate for ACFs can be of several orders of magnitude for small values of the A/S ratio and large values of n_e. For example, when there is a single false positive in a bucket, the expected reduction of the FPR for the ACF with $c = 4$ would be by a factor of approximately $n_e/2$. This would occur when $\frac{A}{S} * FPR_{CF} \ll 1$.[4] Taking a closer look at the results, we can clearly see this effect when the number of fingerprint bits is 16 for all the values of $\frac{A}{S}$ considered. For 8 and 12 fingerprint bits, this amount of gain is only achieved for the lower values of $\frac{A}{S}$.

These results confirm the potential of the ACF to reduce the false-positive rate on networking applications. In fact, the configuration considered in which all elements are queried the same

[4] The factor of two comes from the difference in the probability a new item has a false positive in a cuckoo filter with $d = 4$, $c = 1$, which is approximately $4/2^a$, versus the similar probability for a cuckoo filter with $d = 2$, $c = 4$, which is approximately $8/2^a$.

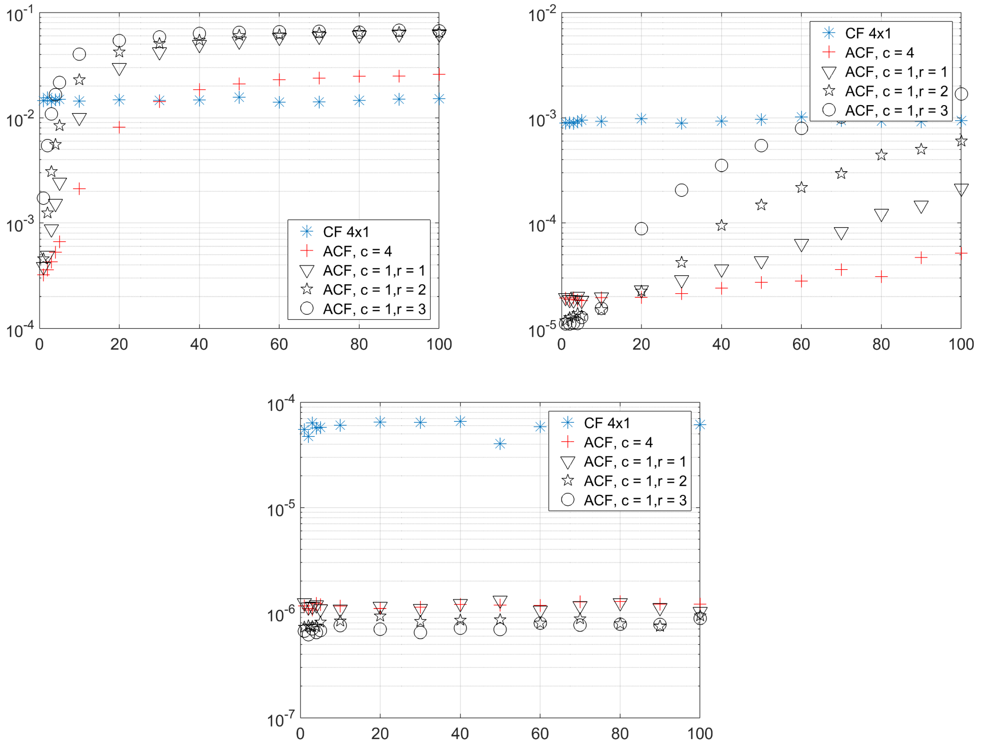

Fig. 4. False-positive rate versus A/S ratio for the ACFs with one bucket and varying fingerprint ranges when $n_e = 100$, $s = 2$, and the number of bits per cell is 8 (left), 12 (middle), and 16 (right).

number of times is a worst case for the ACF; if some elements have more queries than others, then the ACF is able to more effectively remove the false positives that occur on those elements. This holds in simulations and is seen in the next subsection with real data. (Generally network traffic is highly skewed [Sarrar et al. 2012].)

The theoretical estimates were computed using the Markov chains and equations discussed in the previous section for all the configurations. The estimates matched the simulation results and the error was in most cases well below 5%. We note that in cases with larger error, the deviations appear to be primarily due to the variance in the simulation results; low probability events can have a significant effect on the false-positive rate. For example, in the first ACF variant when $s = 1$ (so there are two fingerprints available for an element in the cell), the probability of having a false positive on the two fingerprints when the A/S ratio is 5 and the number of bits per cell is 16 is below 10^{-6}, but when that event occurs it can create up to n_e false positives and thereby significantly contribute to the false-positive rate, particularly when n_e is large. Our results demonstrate that the theoretical estimates can be used as an additional approach to guide choosing parameters for performance, along with simulations.

Additionally, we performed a set of simulations to evaluate the potential benefits of using ACFs with one bucket and varying fingerprint ranges, as described in Section 3.2. Recall, we use r as a parameter where the first fingerprint function obtains a fraction $1 - 1/2^r$ of the possible fingerprints, and the remaining fingerprints are equally distribution among 2^s additional fingerprint functions. We tested configurations with $r = 1, 2, 3$ and $s = 0, 1, 2, 3$, with the same A/S ratios, values of n_e, occupancy, and cell sizes as the previous simulations. The comparison was made against

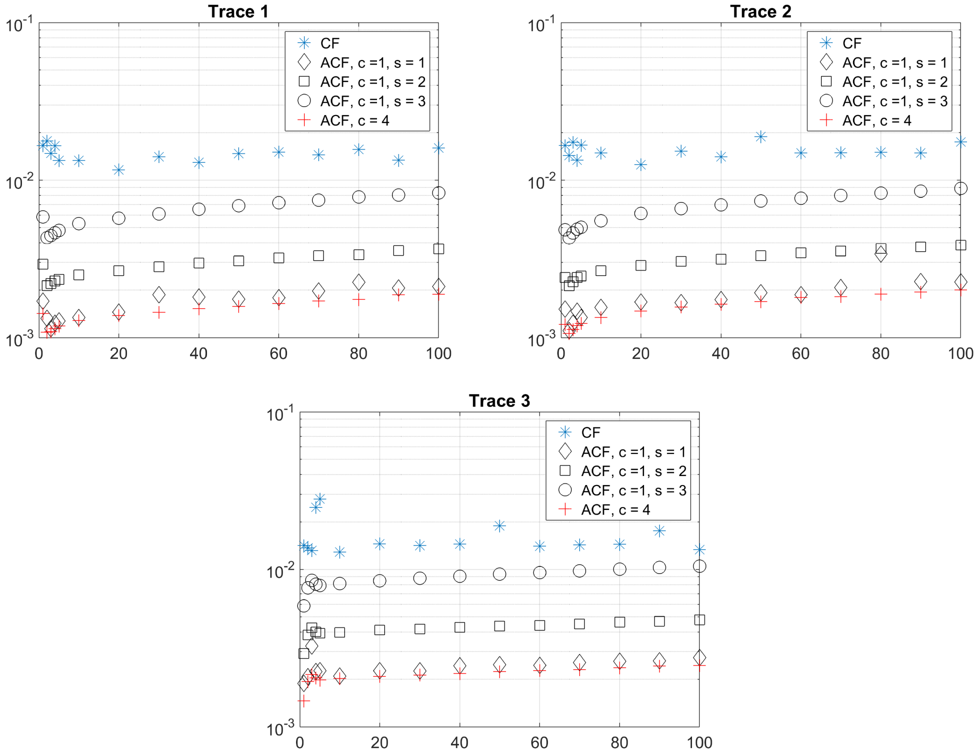

Fig. 5. False-positive rate versus A/S ratio for the three CAIDA traces when using 8 bits per cell.

the ACF for buckets with multiple cells, because, as discussed previously, it provides the lowest false-positive rate for most configurations.

Our experiments show that using varying fingerprint ranges can reduce the false-positive rate when the number of bits per cell is sufficiently large. In particular, for 16 bits per cell, we found improvement for $r = 2, 3$ and $s = 2, 3$. For 12 bits per cell, the use of varying fingerprint ranges only provided a gain for A/S ratios lower than 10, for $r = 2, 3$ and $s = 1, 2, 3$. For 8 bits per cell, there was no benefit, and using varying fingerprint ranges led to increased false-positive rate as the A/S ratio increased. The relative performance of the ACFs with one bucket and varying fingerprint ranges and the ACF for buckets with multiple cells showed little dependence on the value of n_e. The results when $n_e = 100$ and $s = 2$ are shown in Figure 4 for cell sizes of 8, 12, and 16 bits. The false-positive rate of the cuckoo filter with $d = 4, c = 1$ is also shown for reference.

These results confirm that the use of ACFs with one bucket and varying fingerprint ranges can further reduce the false-positive rate when the number of bits per cell is large enough so that there are a large enough number of fingerprints relative to the A/S ratio. However, its use should be carefully considered, as varying fingerprint ranges can lead to worse performance.

4.2 Simulations with Packet Traces

We also utilized packet traces taken from the CAIDA datasets [CAIDA 2018] to validate the ACF with real-world data. For each trace and size set, the first $|S|$ flows were selected to be in the set S and placed on the ACF. Then all the packets were queried and the false-positive rate was logged. To simulate several values of the A/S ratio, the size of the ACF was changed. The two variants of the ACF and the cuckoo filter were evaluated as in the previous subsection. The ACF was able to

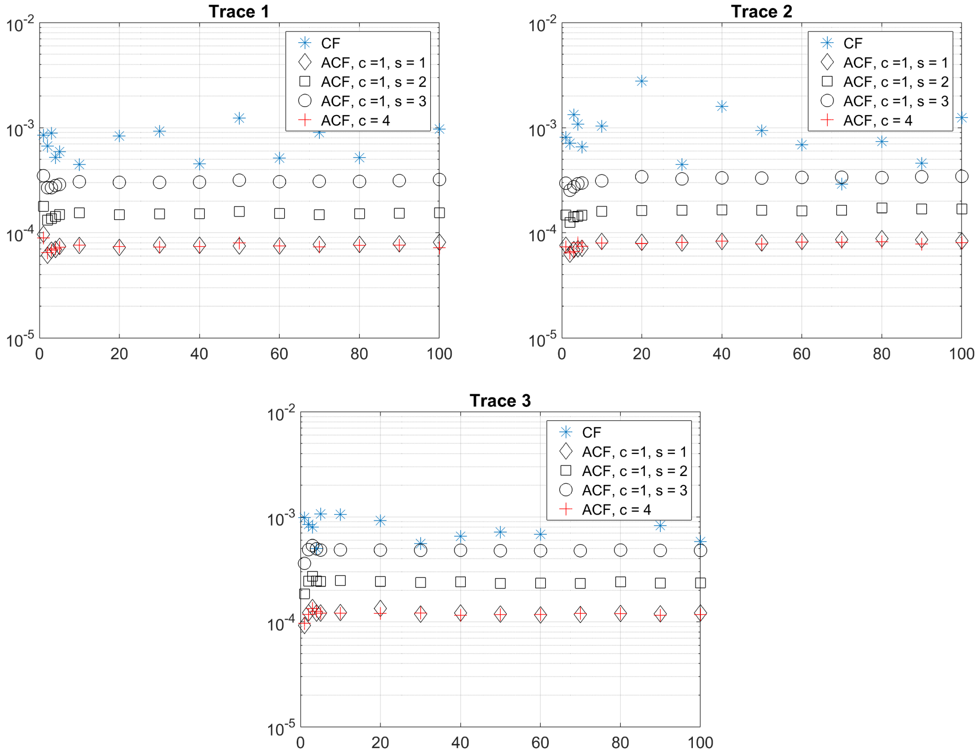

Fig. 6. False-positive rate versus A/S ratio for the three CAIDA traces when using 12 bits per cell.

consistently reduce the false-positive rate of the cuckoo filter. We selected the 5-tuple as the key to insert in the ACF, and the set of 5-tuples in each trace as the $A \cup S$ set. Then, we select the same A/S ratios used in the previous experiment and set the size of the S set, and consequently the size of the ACF so that it reaches 95% load when the elements in S are inserted. Then, the ACF is loaded by picking the first $|S|$ 5-tuples in the trace. The remaining flows were considered as potential false positives for the ACF.

We provide results for three distinct traces,[5] with each corresponding to 60 seconds of traffic. The first trace has approximately 18.5 million packets and 691,371 different flows, giving an average value of $n_e \approx 26.7$ packets per flow. The distribution of n_e is highly skewed with the largest flow having 130,675 packets and the largest one thousand flows each having more than ten thousand packets. The other two traces have 14.6 and 37.4 million packets and 632,543 and 2,313,092 flows, respectively, and the number of packets per flow is also highly skewed.

The false-positive rates for the three traces are shown in Figures 5, 6, and 7 for 8, 12, and 16 bits per cell, respectively. The results are similar for the three traces. The best ACF configurations are the first variant ($d = 4, c = 1$) with $s = 1$ and the second variant ($d = 2, c = 4$). Both provide similar false-positive rates and clearly outperform the cuckoo filter for all the values of the bits per cell tested. An interesting observation is that the ACF reduces the false-positive rate for all A/S ratios even when the number of bits per cell is only eight. This was not the case with simulated queries, and the empirical result can be explained by how the ACF benefits from the skewness

[5] The traces used were taken from the CAIDA 2014 dataset and are equinix-chicago.dirA.20140619-130900, equinix-chicago.dirB.20140619-132600, and equinix-sanjose.dirA.20140320-130400.

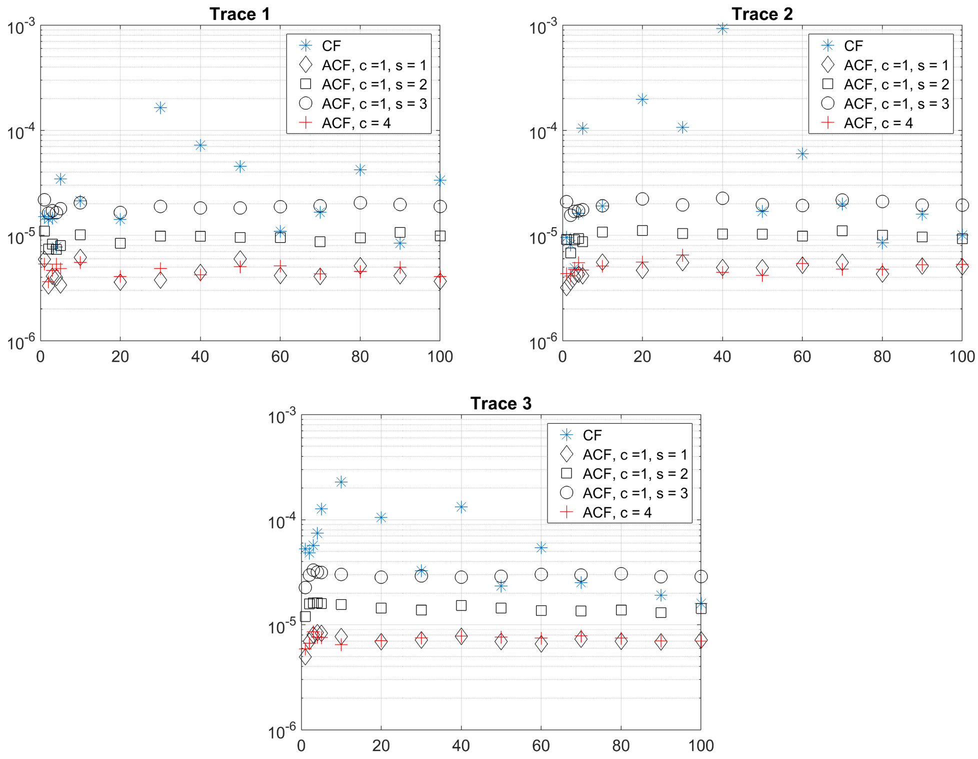

Fig. 7. False-positive rate versus A/S ratio for the three CAIDA traces when using 16 bits per cell.

of the traffic. For example, let us consider the first ACF first variant ($d = 4, c = 1$) with $s = 1$, and consider a case where there are two flows that create false positives on a bucket, one on each of the hash selector values. If one of the flows has 1,000 packets and the other has only 10, then in the worst case, the ACF gives only 20 false positives on the bucket, because after 10 packets the hash selector value that gave a false positive for the small flow will give no more false positives, and the ACF will stop on that hash selector. Also, when the number of bits per cell increases, the results for the cuckoo filter show high variability. This is again due to the traffic skewness, because as the false-positive probability gets smaller, only a few flows contribute to the false-positive rate, and the rate therefore depends heavily on the number of packets of those flows. This variability does not appear on the ACF as false positives on flows with many packets are effectively removed by the ACF as explained before.

5 CONCLUSIONS AND FUTURE WORK

This article has presented the adaptive cuckoo filter (ACF), a variant of the cuckoo filter that attempts to remove false positives after they occur so that the following queries to the same element do not cause further false positives, thereby reducing the false-positive rate. The performance of the ACF has been studied theoretically and evaluated with simulations. The results confirm that when queries have a temporal correlation the ACF is able to significantly reduce the false-positive rate compared to the cuckoo filter, which itself offers improvements over the original Bloom filter.

The main novelty of the ACF is its ability to adapt to false positives by identifying the element that caused the false positive, removing it and re-inserting it again in a different way so that the false positive is removed, but the element can still be found when searched for. This article has studied two variants of the ACF, in both of which the elements remain in the same bucket when false positives are found; other more complex schemes could move elements to different buckets as well, but we leave studying such variants to future work. We believe the simple variants we have proposed, because of their simplicity and strong performance, can find uses in many natural applications.

APPENDIX: ALGORITHMS

We provide pseudocode for our algorithms.

ALGORITHM 1: ACF for buckets with a single cell: Lookup for an element x

```
Require: Element $x$ to search for
Ensure: Positive/negative
for $i \leftarrow 1$ to 4 do
    Access bucket $h_i(x)$
    Read $\alpha$ and $f_\alpha(y)$ stored on the cell
    Compute $f_\alpha(x)$
    Compare $f_\alpha(y)$ in the cell with $f_\alpha(x)$
    if Match then
        return positive
    end if
end for
return negative
```

ALGORITHM 2: ACF for buckets with a single cell: Adaptation to remove a false positive

```
Require: False positive on bucket $h_i(y)$
Ensure: update $\alpha$ and $f_\alpha(y)$
Access bucket $h_i(y)$
Read $\alpha$ stored on the cell
Retrieve element $y$ stored on bucket $h_i(y)$ from the main table
Increment $\alpha$ modulo $2^s$
Compute $f_\alpha(y)$
Store the new value of $\alpha$ and $f_\alpha(y)$ on bucket $h_i(y)$ in the ACF
```

ALGORITHM 3: ACF for buckets with multiple cells: Lookup for an element x

```
Require: Element $x$ to search for
Ensure: Positive/negative
Compute $f_1(x), f_2(x), f_3(x), f_4(x)$
for $i \leftarrow 1$ to 2 do
    Access bucket $h_i(x)$
    for $j \leftarrow 1$ to 4 do
        Compare fingerprint stored in cell $j$ with $f_j(x)$
        if Match then
            return positive
        end if
    end for
end for
return negative
```

ALGORITHM 4: ACF for buckets with multiple cells: Adaptation to remove a false positive

```
Require: False positive in cell $j$ on bucket $h_i(y)$
Ensure: update bucket $h_i(y)$
Select randomly a cell $k$ different from $j$ on bucket $h_i(y)$
Retrieve elements $y, z$ stored in cells $j, k$ on bucket $h_i(y)$ from the main table
Compute $f_j(z)$ and $f_k(y)$
Write $f_j(z)$ and $f_k(y)$ in cells $j$ and $k$ on bucket $h_i(y)$ of the ACF
Write $z$ in cell $j$ and $y$ in cell $k$ on bucket $h_i(y)$ in the main table
```

REFERENCES

Michael A. Bender, Martín Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom filters, adaptivity, and the dictionary problem. In *Proceedings of the IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS'18).* 182–193. DOI: https://doi.org/10.1109/FOCS.2018.00026

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

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

Jehoshua Bruck, Jie Gao, and Anxiao Jiang. 2006. Weighted bloom filter. In *Proceedings of the IEEE International Symposium on Information Theory.* IEEE, 2304–2308.

CAIDA. 2018. Realtime Passive Network Monitors. Retrieved from http://www.caida.org/data/realtime/passive.

Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04).* Society for Industrial and Applied Mathematics, Philadelphia, PA, 30–39. Retrieved from http://dl.acm.org/citation.cfm?id=982792.982797.

Martin Dietzfelbinger and Christoph Weidling. 2007. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.* 380, 1–2 (2007), 47–68.

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 ACM CoNEXT Conference.* ACM, 13.

Ulfar Erlingsson, Mark Manasse, and Frank McSherry. 2006. A cool and practical alternative to traditional hash tables. In *Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS'06).*

Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies.* ACM, 75–88.

Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. 2005. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.* 38, 2 (2005), 229–248.

Intel. 2011. 64 and IA-32 Architectures Software Developer's Manual, Volume 2.

Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.* 39, 4 (2009), 1543–1561.

Daniel Lemire and Owen Kaser. 2016. Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptogr. Eng.* 6, 3 (2016), 171–185.

Michael Mitzenmacher and Eli Upfal. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis.* Cambridge University Press.

Michael Mitzenmacher and Salil Vadhan. 2008. Why simple hash functions work: Exploiting the entropy in a data stream. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms.* Society for Industrial and Applied Mathematics, 746–755.

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

Salvatore Pontarelli, Pedro Reviriego, and Juan Antonio Maestro. 2016. Parallel d-pipeline: A cuckoo hashing implementation for increased throughput. *IEEE Trans. Comput.* 65, 1 (2016), 326–331.

Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. 2012. Leveraging Zipf's law for traffic offloading. *ACM SIGCOMM Comput. Commun. Rev.* 42, 1 (2012), 16–22.

Ming Zhong, Pin Lu, Kai Shen, and Joel Seiferas. 2008. Optimizing data popularity conscious bloom filters. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing.* ACM, 355–364.

Received August 2018; revised May 2019; accepted June 2019