Cardinality Estimation Adaptive Cuckoo Filters (CE-ACF): Approximate Membership Check and Distinct Query Count for High-Speed Network Monitoring

Pedro Reviriego, Jim Apple, Alvaro Alonso, Otmar Ertl, and Niv Dayan

***Abstract*— In network monitoring applications, it is often beneficial to employ a fast approximate set-membership filter to check if a given packet belongs to a monitored flow. Recent adaptive filter designs, such as the Adaptive Cuckoo Filter, are especially promising for such use cases as they adapt fingerprints to eliminate recurring false positives. In many traffic monitoring applications, it is also of interest to know the number of distinct flows that traverse a link or the number of nodes that are sending traffic. This is commonly done using cardinality estimation sketches. Therefore, on a given switch or network device, the same packets are typically processed using both a filter and a cardinality estimator. Having to process each packet with two independent data structures adds complexity to the implementation and limits performance. This paper shows that adaptive cuckoo filters can also be used to estimate the number of distinct negative elements queried on the filter. In flow monitoring, those distinct queries correspond to distinct flows. This is interesting as we get the cardinality estimation for free as part of the normal adaptive filter's operation. We provide (1) a theoretical analysis, (2) simulation results, and (3) an evaluation with real packet traces to show that adaptive cuckoo filters can accurately estimate a wide range of cardinalities in practical scenarios.**

***Index Terms*— Flow monitoring, cardinality estimation, approximate membership checking, adaptive cuckoo filters.**

I. Introduction

IN MANY networking applications, there is a need to check whether an element belongs to a set. For example, each packet received by a switch or a router has to be checked against a potentially large routing table to determine the best matching entry (i.e. the longest matching prefix for IP and the exact matching destination MAC address for Ethernet). In network monitoring, it is of interest to track subsets of the flows[1] that traverse a link at a given time. This is done for example to count the number of bytes or packets belonging to flows with certain characteristics (e.g., subject to significant traffic or originating in a given set of IP addresses [1]). Determining set membership can be done by storing the full keys in a hash table [2], [3], yet this can be costly in terms of memory footprint and bandwidth.

Switching ASICs have a limited amount of on-chip memory (e.g. SRAM) that is fast and a much larger external memory (e.g. DRAM) that is slower and has bandwidth limitations [4]. At the same time, each key (i.e., a 5-tuple) consists of 104 and 296 bits in IPv4 and IPv6, respectively. Hence, storing a hash table with the full keys on-chip is too costly in terms of memory. On the other hand, storing it in the slower external memory and checking it for every packet (including ones that are not being tracked) can create a performance bottleneck. This problem is also commonly found in database applications, where the keys that identify data entries are often too large and numerous to be stored and quickly checked against in faster memory (i.e., in this case, DRAM). On the other hand, having to search storage (i.e., disk or SSD) for each queried key can lead to a performance bottleneck [5], [6], [7], [8].

A common solution to this problem is to first perform a quick approximate check if the element is in the set [9]. Only on a positive outcome, a full check for the existence of the key in external memory is needed. Hence, when the application queries for non-existing keys, many accesses to external memory are avoided [10], [11]. This significantly alleviates the bottleneck on the external memory interface and reduces the use of on-chip memory. This quick approximate membership check is implemented with probabilistic data structures commonly referred to as filters. Many different types of filters have been proposed over the years from the classical Bloom filter [12] to the more recent cuckoo [13], xor [14], quotient [15], [16], [17], and ribbon [18] filters. These filters cannot return false negatives but do return false positives.

Manuscript received 9 January 2023; revised 3 July 2023; accepted 29 July 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Caesar. The work of Pedro Reviriego was supported by the Spanish Agencia Estatal de Investigacion (AEI) 10.13039/501100011033 through the ACHILLES Project under Grant PID2019-104207RB-I00 and the 6G-INTEGRATION-3 Project under Grant TSI-063000-2021-127. *(Corresponding author: Pedro Reviriego.)*

Pedro Reviriego and Alvaro Alonso are with the Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, 28040 Madrid, Spain (e-mail: pedro.reviriego@upm.es; alvaro.alonso@upm.es).

Jim Apple is with Engineering, Voltron Data, Mountain View, CA 94041 USA (e-mail: jima@voltrondata.com).

Otmar Ertl is with Dynatrace Research, 4040 Linz, Austria (e-mail: otmar.ertl@dynatrace.com).

Niv Dayan is with the Department of Computer Science, University of Toronto, Toronto, ON M5S 1A1, Canada (e-mail: nivdayan@cs.toronto.edu).

Digital Object Identifier 10.1109/TNET.2023.3302306

[1]A flow is defined as a group of packets that have the same 5-tuple formed by the source and destination IP addresses and ports and the protocol used. This typically corresponds to an exchange of information between two endpoints.

Therefore, they can be used to perform a fast and low-cost check when many queries are negative to safely discard them from further processing [10]. The probability of a false positive depends on the amount of memory allocated to the filter and the number of elements it stores. It can typically be estimated analytically. The different filters offer different trade-offs in terms of their false positive probability, performance, and memory footprint, among other features [19].

Recently, adaptive filters have been proposed to react to a false positive when querying for some element x by modifying the filter such that subsequent queries for x return a negative [20]. Such adaptation is useful in skewed query distributions, whereby the same key may be repeatedly queried for and result in a false positive each time. In flow processing, for instance, each flow consists of many packets, and it is therefore desirable to prevent future false positives on the rest of the packets once the first false positive has occurred. Adaptive filters based on quotient [21], Bloom [22] and cuckoo filters [23] have been proposed and shown to dramatically reduce the false positive rate in packet processing applications and beyond.

In network monitoring, another feature that is of interest is knowing the number of distinct flows on a link or the number of distinct active nodes in a network [24], [25]. This is useful, for example, in detecting early Distributed Denial of Service Attacks (DDoS) or other anomalies in the traffic [26]. Similarly, in database workload monitoring, it is desirable to measure the size of the set of distinct queries. This is useful not only for security and anomaly detection [27] but also for deriving business insight. For instance, in online search, it is desirable to monitor the number of distinct search terms to gauge user behavior over time [28].

Computing the exact number of distinct elements on a large set can be costly in terms of memory and computation. To reduce the cost, many cardinality estimation algorithms have been proposed that provide good estimates while using a small memory footprint and simple operations [29]. For example, the HyperLogLog (HLL) estimator is widely used in both computing [28] and networking applications [30] and can provide estimates that deviate by less than 1% from the real value using only a few kilobytes of memory [31]. Cardinality estimators typically hash each key to an array of approximate counters, which are used to compute the cardinality estimate.

Overall, we observe that many network devices have to apply 1) approximate membership checking and 2) cardinality estimation on the same set of packets. If independent data structures are used for each function, each packet has to be processed twice. Furthermore, enough memory has to be allocated to both data structures to provide low error guarantees. This poses a challenge for high-speed switches that have limited on-chip memory and have to process hundreds of millions of packets per second [4]. For such applications, reducing the number of memory accesses needed per packet and the memory footprint is highly desirable.

This paper shows that carefully configured adaptive cuckoo filters can surprisingly be used to estimate the cardinality of the negative elements queried on the filter as part of their normal operation. In particular, a cardinality estimate can be computed from the contents of the filter without introducing any modification to its structure, memory footprint, or operations. The cardinality estimate is accurate except for very small or very large values, thus covering the cardinality values of many practical use cases. This means that adaptive cuckoo filters provide cardinality estimation for free. This paper shows how to derive cardinality estimations from an adaptive cuckoo filter and evaluate the accuracy of the results. Specifically, we make the following contributions:

1) Showing that the contents of adaptive cuckoo filters in some configurations contain information about the cardinality of the set of negative queries done on the filter.
2) Presenting a cardinality estimator that uses the filter contents to estimate cardinality.
3) Analyzing theoretically the proposed filter-based cardinality estimator.
4) Evaluating by simulation the proposed filter-based cardinality estimator.
5) Discussing how cardinality estimation may also be implemented in other adaptive filters.

The rest of the paper is organized as follows. Section II provides the background on adaptive cuckoo filters and cardinality estimators. The proposed cardinality estimation approach using an adaptive cuckoo filter is presented and analyzed in section III and evaluated using simulation in section IV. We provide a guideline for how to generalize the approach to other adaptive filters in section V.

II. Preliminaries

This section briefly describes adaptive cuckoo filters and discusses existing cardinality estimation algorithms.

A. Adaptive Cuckoo Filters (ACFs)

In many applications for which approximate membership checking is used, the workload is skewed with a few elements being checked repeatedly. In networking, for instance, a significant part of the traffic is concentrated on a few flows [32]. This means that when a false positive occurs for one of those frequently checked elements, many more false positives are likely to follow as subsequent checks are made to the same element. This has led to the development of adaptive filters [20] such as the adaptive cuckoo filter [23]. Such filters, upon detecting a false positive, adapt their contents so that future queries to the same element do not result in more false positives.

An adaptive cuckoo filter stores a fingerprint for each key within a cuckoo filter in fast memory while storing the full keys within a cuckoo hash table in slower memory. The cuckoo filter and cuckoo hash table have the same number of buckets and cells per bucket, and there is a one-to-one correspondence between their contents: if cell j within bucket i of the cuckoo filter stores a fingerprint for key x, then cell j within bucket i of the cuckoo hash table stores the full key x. Insertions and removals are done using cuckoo hashing in the filter, while full keys are moved in the cuckoo hash table to mirror the cuckoo filter's contents. This mirrored design allows to retrieve the key associated with any fingerprint. When a false positive occurs,

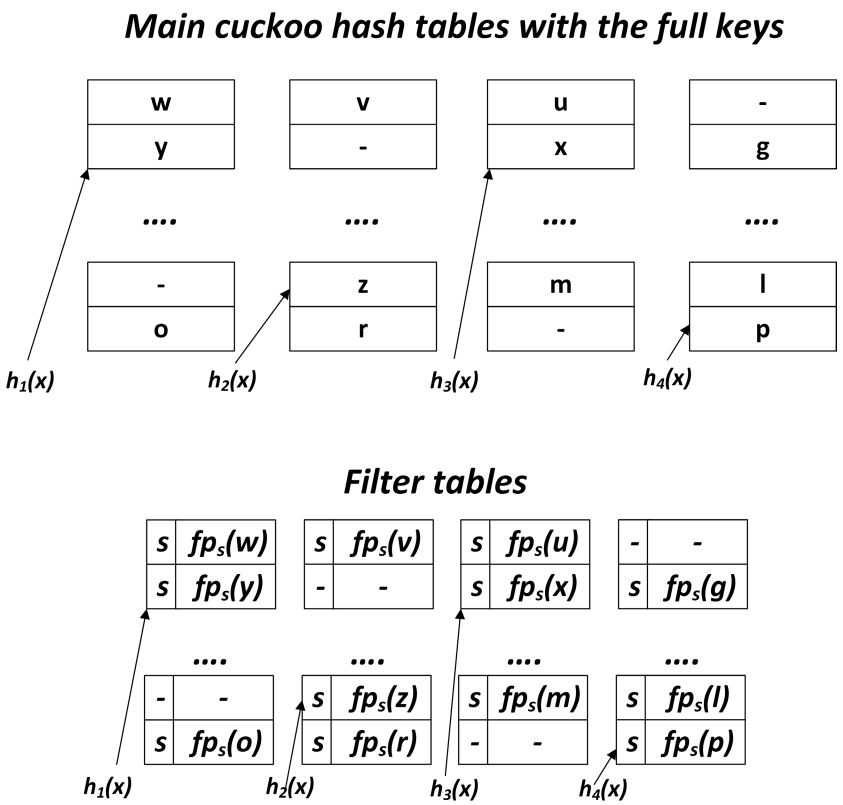

Fig. 1. Illustration of the four table ACF configuration with one bit fingerprint selector. A lookup for element x is being done on the ACF. The four hash functions h_1,h_2,h_3,h_4 are computed for x and those positions are read on each table respectively. Then the selector bit s is used to select the fingerprint $fp_0(x)$ or $fp_1(x)$ to use in the comparison with the fingerprint stored on each table. The entry on the third table will match the fingerprint and subsequently the main tables will be accessed. In this case x is stored in the main tables and thus is a true positive.

the full key associated with the falsely matching fingerprint is retrieved from the cuckoo hash table, and an alternative hash function is used to generate a new fingerprint for it. The filter maintains some extra metadata to record which hash function was used to generate each fingerprint. During a query, this allows comparing a key in question to a fingerprint using the correct hash function. After adapting a fingerprint, it is very unlikely that the new fingerprint will still match the queried key after a different hash function is applied. In this way, adaptive cuckoo filters eliminate recurring false positives for frequently queried keys.

Adaptive cuckoo filters are a natural fit for flow monitoring as the full key must be retrieved whenever a positive occurs in the filter. Hence, the overhead of retrieving the full key from slower memory in order to adapt a fingerprint does not constitute additional overhead. Furthermore, in flow monitoring and many other networking applications, the size of each full key is uniform (e.g., a 5-tuple) and so the keys align perfectly within the cells of the cuckoo hash table. Finally, an additional benefit is that cuckoo hashing is widely used in networking applications and implementable in hardware [3].

Adaptive Cuckoo filters are also a good match for modern database applications. For example, recent systems such as Aerospike [33] and FASTER [34] store data in a log-structured manner in storage (e.g., disk or flash), and they index the locations of data entries using a space-efficient filter in memory (e.g., DRAM). This filter contains a fingerprint for each entry and a pointer to the location for the entry in storage. Hence, in case of a positive fingerprint match, a query follows the pointer to storage to check if it points to the target entry. In this context, mirroring is unnecessary as the pointers to storage allow retrieving the original key for a falsely matching fingerprint so that it can be adapted. Since databases are often subject to heavy-tailed query distributions, having the filter adapt to remove recurring false positives reduces tail latency.

Two adaptive cuckoo filter variants have been proposed. In the first one, the filter consists of two sub-tables and each bucket contains four cells. In the second design, the filter consists of four sub-tables and each bucket contains one cell [23]. In the first configuration, a different hash function is used for each of the cells and adaptation is implemented by moving elements from one cell to another within the same bucket. In the second design, additional metadata in the form of one or more *selector bits* record which hash function generated each fingerprint. When a false positive occurs, the value of the selector bits is modified to reflect the new hash function being used.

In this paper, only the four-table adaptive cuckoo filter configuration[2] is used as our cardinality estimation method uses the selector bits that are only applicable to this configuration. We illustrate it in Figure 1. It is composed of the main cuckoo

[2]It may seem that this configuration has worse false positive probability than the two-table configuration, as it requires selector bits for adaptation. However, the false positive probability when no adaptation is used is lower (approximately half) in the four table configuration, so when using one selector bit both configurations have similar performance.

hash tables (top) that store the full keys and the filter itself (bottom). Both of these components are partitioned into four sub-tables, and the cuckoo hash table mirrors the contents of the cuckoo filter. Each sub-table is accessed using a different hash function $h_i()$ in the filter. We denote the number of selector bits per cell as s_b. Each cell of the filter stores a selector value s and the fingerprint $fp_s(z)$ of an element z is computed using the hash function that corresponds to the selector value s.

To search for key x in the filter, positions $h_i(x)$ are read and the stored fingerprints are compared with $fp_s(x)$ using the value of s stored on each position. If there is a match, a positive is returned. Otherwise, the result of the check is negative. Figure 1 illustrates the mappings for element x that would match on the third table of the filter. If the filter returns a positive, the cuckoo hash table is checked. If the key is not found, then a false positive is detected. To prevent it from recurring, the value of s and the fingerprint are changed in the filter for that key.

The false positive probability for a non-existing key that has not been queried before can be approximated by $\frac{d \cdot o}{2^f}$ where d is the number of tables, o the filter occupancy, and f the fingerprint length in bits. This is the same as for the plain cuckoo filter. In contrast, the false positive rate on a set of recurring queries to non-existing keys is lower as the filter will adapt to remove false positives when they are detected [23].

When inserting a key, we may find that the positions it maps to on the four tables are already occupied. When that occurs, one of the stored elements is displaced to one of its three alternative locations. This process continues recursively until an empty cell is found as in cuckoo hashing. In the ACF, the movements are done in both the filter and the full cuckoo hash tables to keep the one to one relationship between cells in both data structures. This enables the adaptive cuckoo filter to reach an occupancy of 95% with very high probability. In networking applications, the number of insertions is typically much lower than the number of queries. For example, in flow monitoring, the number of insertions is equal to the number of flows being monitored. In contrast, the number of queries is equal to the number of packets across all flows that traverse the link. Hence, the number of queries is typically several orders of magnitude larger than the number of insertions. For this reason, swapping entries across buckets to find space has a comparatively lower overall overhead, and the performance bottleneck tends to be querying.

For a ACF with four sub-tables, the best performance on real packet traces is obtained when using a single selector bit [23]. Interestingly, the proposed cardinality estimation only is applicable to a single selector bit as will be discussed in section III. Therefore, in the rest of the paper we also assume that a single selector bit $s_b = 1$ is used. In this case, two hash functions are used and thus two different fingerprints are supported for each key. In this case, the selector bit can be changed from 0 to 1 or from 1 to 0. Note that as many queries are processed, several false positives can occur on a cell so that the value of the selector can switch back and forth. This configuration has a similar performance to that of the two table configurations when evaluated using packet traces. We refer the reader to the original ACF paper [23] for further details.

B. Cardinality Estimation Algorithms

Estimating the number of distinct elements or cardinality of a set is needed in many computing [28] and networking applications [30]. In network monitoring, it is of interest to know the number of flows on a link, or in an online service to know the number of unique users. In database systems, it is often desirable to measure the number of distinct queries to non-existing keys, e.g., to ascertain whether data is missing and if so how much. Over the years, many cardinality estimation algorithms have been proposed [29]. These algorithms can drastically reduce the memory footprint and computing effort compared to an exact calculation. For example, estimates that are accurate to a few percent of the exact value over a wide range of cardinalities are achieved using only a few kilobytes of memory and hash computations per element. The general approach used in cardinality estimation algorithms is to map the elements to pseudo-random values and compute some statistics on those values. We describe the most popular cardinality estimation approaches below.

One of the simplest methods is the Linear Probabilistic Counting Array (LPCA) [35], also known as linear counting [29]. This approach maps each key to one bit in a bit vector, setting it from 0 to 1 or keeping it set to 1. Cardinality is estimated based on the number of ones in the array. An LPCA is simple to implement. To achieve a good accuracy, however, some knowledge of the expected maximum cardinality is needed. Furthermore, the memory requirement is linear in the number of keys. This is a limitation when memory is scarce and the cardinality can take large values.

This has led to the development of more sophisticated algorithms that require a sublinear or even constant memory size for practical cardinality values. The k minimum values sketch [36], [37] maps elements to hash values using a hash function and keeps the k minimum hash values seen in the stream of data. Cardinality is estimated based on the largest of the k hash values seen so far. This means that the memory footprint is independent of the number of elements in the set.

The HyperLogLog [31] and its predecessors (e.g., the LogLog [38]) map each element to multiple slots in an array of counters. Each counter stores the maximum number of leading zeroes across all hashes that have been mapped to it. Such designs significantly reduce the memory requirement. The HyperLogLog can estimate cardinality values for up to billions of elements using a constant array size with five bit counters. As a result, HyperLogLog is widely used in computing [28] and networking [30] applications. HyperLogLog has been implemented, for instance, in programmable data planes using P4 [39], [40] and is used by most major cloud providers. HyperLogLog is considered the state-of-the-art for cardinality estimation when reducing the memory footprint is the priority. In the rest of the paper, HyperLogLog is used as the baseline for evaluating our proposed CE-ACF in terms of cardinality estimation. Comparing against simpler algorithms such as LPCAs would show larger benefits. By comparing to

HyperLogLog, the results presented in the paper represent a lower bound in terms of memory savings.

Recently, the use of machine learning has also been proposed for cardinality estimation, but it remains to be seen if it will become as widely used and achieves the accuracy and cost of existing solutions [41], [42].

III. Cardinality Estimation With Adaptive Cuckoo Filters (CE-ACF)

This section first describes the cardinality estimation problem and the intuition of why adaptive cuckoo filters can be used to address it. Then, the proposed filter-based cardinality estimation algorithm is presented in detail and analyzed theoretically for the single selector bit s_b configuration. For larger values of s_b, it is shown that the adaptation state depends on the number of elements and not only on the number of distinct elements and therefore the values of the selector bits cannot be used for cardinality estimation. Finally, we discuss (1) the potential of integrating the cardinality estimation with the filter versus the use of independent data structures, and (2) the implementation of the proposed scheme in different hardware platforms. The terms used throughout the paper are summarized in Table I.

A. Problem Statement and Intuition

Consider a multiset of elements S that is queried on an adaptive cuckoo filter. Let N be the subset of S formed by elements that have not been inserted into the filter. In this setting, we want to estimate the cardinality C (i.e., the number of distinct elements) of the set N. This corresponds to a scenario in which an adaptive cuckoo filter is used to monitor a small fraction of the flows (that are thus inserted into the filter) that traverse the link (e.g., originating in a set of source IP addresses) and we also want to know the total number of active flows on the link. In this case, S corresponds to all packets flowing through the link while N corresponds to packets belonging to non-monitored flows. Since the number of monitored flows is known,[3] we want to estimate the number of flows in N.[4]

The intuition for why an adaptive filter can estimate cardinality is that the filter state contains information about the number of distinct negative elements that had been queried for. For example, consider a cell in the filter that stores an f bit fingerprint and one selection bit (initialized to zero) that determines the hash function used to compute the fingerprint. If no negative elements that map to this cell are queried, the selection bit would be zero. However, if one negative element is queried for, there would be approximately a probability of $\frac{1}{2^f}$ that it matches the stored fingerprint. In this case, adaptation would be triggered setting the selection bit to one. Note that querying again for the same element will not change the state of the filter. As more distinct negative elements are queried, the probability of adaptation increases. Therefore, the values of the selector bits contain information about the number of distinct negative elements queried on the filter. A larger number of cells that have adapted would correspond to a larger cardinality.

[3]Note that there is no need to estimate the number of elements that have been inserted into the filter as we can easily compute the exact value by just counting the number of flow insertions.

[4]Note that typically when a filter is used, the number of negative elements is larger than the number of positive elements, as this is the case where filters eliminate many external memory accesses and are therefore attractive.

B. Estimating the Cardinality

Let us consider an adaptive cuckoo filter with $d = 4$ tables, each consisting of b buckets with one cell per bucket. Each cell is formed by a selector bit s and a fingerprint $fp_s(x)$ of f bits. The filter is first loaded with a set of elements T to reach an occupancy of o. Finally, a set of S elements is queried on the filter of which C are distinct negative elements belonging to the set N. At the end of this process, the probability that a fingerprint with a given cell has been adapted can be estimated as follows. The number of distinct elements that map to each cell can be approximated by $\frac{C}{b}$, which would typically be much larger than one. An example of the mappings to one of the cells is illustrated in Figure 2. In this case, adaptation is triggered when querying for element y. The more queries to non-existing elements that map to this cell, the larger is the probability of adaptation. In this case, no other negative elements match $fp_1(x)$, so the cell's selector bits will store $s = 1$.

On average, the expected number of elements in N that match a fingerprint in a given cell is $\frac{C}{b \cdot 2^f}$. When using the Poisson approximation [31], which assumes that the number of distinct elements is Poisson distributed, the probability of having k distinct elements that match the initially stored fingerprint $fp_i(x)$ can be modeled as:

$$P_i(k) \approx \frac{(\frac{C}{b \cdot 2^f})^k}{k!} \cdot e^{-\frac{C}{b \cdot 2^f}} \tag{1}$$

The probability that at least one element $y \in N$ matches a fingerprint $fp_0(x)$ in a given cell and triggers an adaptation is approximated as $1 - P_i(0)$ in Equation 2.

$$P_{adapt} \approx 1 - e^{-\frac{C}{b \cdot 2^f}} \tag{2}$$

Let us consider a cell for which there is a matching element y_0 for $fp_0(x)$ and another matching element y_1 for $fp_1(x)$. In this case, the final value of s depends on which of the two elements is queried last on the filter. If the last one is y_0, then $s = 1$. Conversely, if it is y_1, then $s = 0$. This is illustrated in Figure 3 for different sequences of queries to elements y_0 and y_1.

As a result, the probability of having each value of s would be approximately 0.5. When there are l_0 and l_1 matching elements (not necessarily distinct) for the fingerprints $fp_0(x)$ and $fp_1(x)$, respectively, the probability of each value of s would be different, being larger for $s = 1$ when $l_0 > l_1$. However, the probability of having l_0 and l_1 matching elements to $fp_0(x)$ and $fp_1(x)$, respectively, is the same as that of having matching l_1 and l_0 elements to $fp_0(x)$ and $fp_1(x)$, respectively. Therefore, considering both cases, the probabilities of having $s = 0$ and $s = 1$ are both approximately 0.5. The only exception to this analysis is when both l_0, l_1 are zero. In this case, $s = 0$. This further implies that the number

TABLE I
SUMMARY OF NOTATION

Symbol	Meaning
b	Number of buckets per table in the filter
$d = 4$	Number of tables in the filter
h_i	Hash function for table i
o	Occupancy of the filter (fraction)
f	Number of fingerprint bits
fp_s	Fingerprint function for selector bit value s
s_b	Number of selector bits
s	Value of the selector bits
T	Set of elements stored in the filter
S	Elements queried on the filter
$N \subseteq S$	Negative elements queried on the filter
C	Cardinality of the negative elements queried on the filter
$P_i(j)$	Probability of having j distinct negative elements on N that match $f_i(x)$ for element x stored in a particular cell
l_0, l_1	Number of elements matching the fingerprint when the selector bit is 0,1 in a given cell.
p_1	Probability that the selector bit takes a value of 1 in a given the cell after a set of queries to the filter
$\hat{p}_1$	Fraction of the selector bits with a value of 1 in the cells of the filter after a set of queries to the filter
M	Number of counters in the HLL estimator
c	Number of counter bits in each cell of the HLL estimator

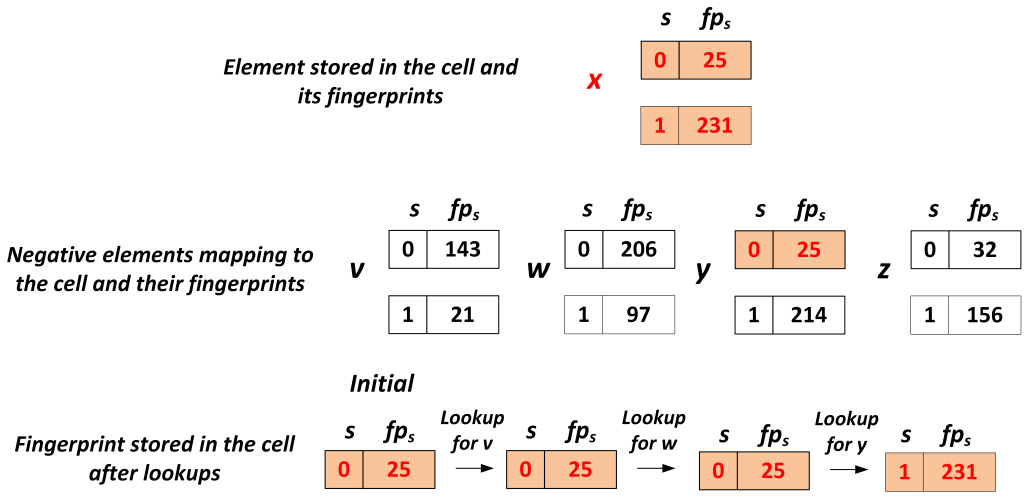

Fig. 2. Illustration of a cell that stores the fingerprint of element x (top) and to which negative elements v, w, y, z map to (middle). Element y has the same fingerprint as x when the selector bit is zero ($s = 0$). Therefore, after y is queried (bottom right), the cell is adapted by setting $s = 1$ and the fingerprint to 231, for which there is no matching fingerprint on v, w, y, z.

of distinct elements mapping to those fingerprints must be zero as well. Therefore, $P(l_0 = 0 \wedge l_1 = 0) = P_0(0) \cdot P_1(0) \approx e^{-\frac{2 \cdot C}{b \cdot 2^f}}$ using equation 1 and the probability $p_1 = P(s = 1)$ that a selector bit is set can be approximated by

$$p_1 = \frac{1}{2} \cdot (1 - P(l_0 = 0 \wedge l_1 = 0)) \approx \frac{1}{2} \cdot (1 - e^{-\frac{2 \cdot C}{b \cdot 2^f}}) \quad (3)$$

By solving Equation 3 for C and estimating p_1 as the fraction of selector bits that are set to 1 (denoted as $\hat{p}_1$), we obtain Equation 4.

$$\hat{C} := -b \cdot 2^{f-1} \cdot \ln(1 - 2 \cdot \hat{p}_1) \quad (4)$$

We obtain a value for the parameter $\hat{p}_1$ using Equation 5 by traversing the filter, counting the number of occupied cells with $s = 1$, and dividing by the number of occupied cells $d \cdot b \cdot o$.

$$\hat{p}_1 := \frac{\text{number of occupied cells with } s = 1}{d \cdot b \cdot o} \quad (5)$$

This analysis is similar to the one of the Odd Sketch [43], which estimates similarity between sets rather than cardinality. Indeed, each cell of the filter is equivalent to a position of an Odd Sketch except that only approximately $\frac{1}{2^f}$ of the elements change the selector bit. Just as for the Odd Sketch, the approximations derived are not valid when p_1 approaches 0.5 (i.e., for very large cardinalities). Since p_1 approaches 0.5 as $C \to \infty$ and the estimator is only defined as long as $\hat{p}_1 < 0.5$, our estimation approach fails for very large cardinalities C. At the same time, for very small cardinalities, it is unlikely that any selector bit would be set. A large estimation error is therefore also expected also in this case. The next subsection analyzes the estimation error as a function of the cardinality.

C. Accuracy

The estimator $\hat{p}_1$ corresponds to an estimator of the probability of a binomial distribution with $d \cdot b \cdot o$ trials for which the variance is known to be

$$\text{Var}(\hat{p}_1) = \frac{p_1 \cdot (1 - p_1)}{d \cdot b \cdot o} \approx \frac{1 - e^{-\frac{4 \cdot C}{b \cdot 2^f}}}{4 \cdot d \cdot b \cdot o}. \quad (6)$$

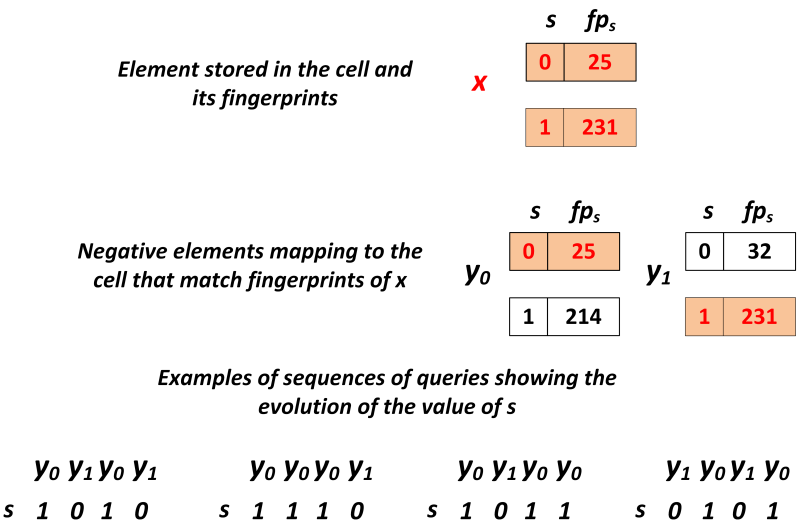

Fig. 3. Illustration of a cell that stores the fingerprint of element x and to which two negative elements y_0 and y_1 map to. Element y_0 has the same fingerprint as x when the selector bit is zero ($s = 0$) and y_1 has the same fingerprint as x when the selector bit is one ($s = 1$). Therefore, the value of the selector bit depends on the sequence of queries as shown in the figure. The last element queried of the pair y_0 and y_1 determines the final value of s for the given cell.

Using the delta method [44], the variance of $\hat{C}$ can be approximated by Equation 7.

$$\mathrm{Var}(\hat{C}) \approx \mathrm{Var}(\hat{p_1}) \left(\frac{dp_1}{dC}\right)^{-2} \approx \frac{1 - e^{-\frac{4 \cdot C}{b \cdot 2^f}}}{4 \cdot d \cdot b \cdot o} \left(\frac{e^{-\frac{2C}{b2^f}}}{b2^f}\right)^{-2} = \frac{4^{f-1} \cdot b \cdot (e^{\frac{4C}{b2^f}} - 1)}{d \cdot o}. \quad (7)$$

As a consequence, the expected relative standard error (RSE) is roughly given by

$$\mathrm{RSE} = \frac{\sqrt{\mathrm{Var}(\hat{C})}}{C} \approx \frac{\phi(\frac{C}{b2^f})}{\sqrt{b \cdot d \cdot o}}. \quad (8)$$

with $\phi(x) := \frac{\sqrt{e^{4x}-1}}{2x}$. Since the function $\phi(x)$ is convex and has its minimum at $x_{\min} \approx 0.398$ with $\phi(x_{\min}) \approx 2.49$, the lowest estimation error is achieved for $C \approx 0.398 \cdot b \cdot 2^f$ where RSE $\approx 2.49/\sqrt{b \cdot d \cdot o}$. The values of $\phi(x)$ are shown in Figure 4. It can be observed how accuracy degrades as we move away from $x_{\min} \approx 0.398$ to either larger or lower values (cardinalities).

For $x \ll 1$ we can approximate $e^{4x} - 1 \approx 4x$ and therefore $\phi(x) \approx \frac{1}{\sqrt{x}}$, which yields

$$\mathrm{RSE} \approx \sqrt{\frac{2^f}{d \cdot o \cdot C}} \qquad \text{for } C \ll b \cdot 2^f. \quad (9)$$

To better understand the accuracy of the proposed estimator, we compare it with HLL that has an estimation error of approximately RSE $\approx 1.04/\sqrt{M}$ where M is the number of counters [31]. This means that on the optimal point, our estimator has the same accuracy as an HLL with $\frac{b \cdot d \cdot o}{5.73}$ counters, and the equivalent number of counters decreases as we move towards smaller or larger cardinalities. This is illustrated in Figure 5, which shows the number of counters of an HLL structure that achieves the same estimation error as our estimator as a function of the cardinality for a filter with $b = 65536, d = 4, f = 8$ and 95% occupancy. It can be seen that our estimator is equivalent to an HLL with more than 5,000 counters over a large range of cardinality values.

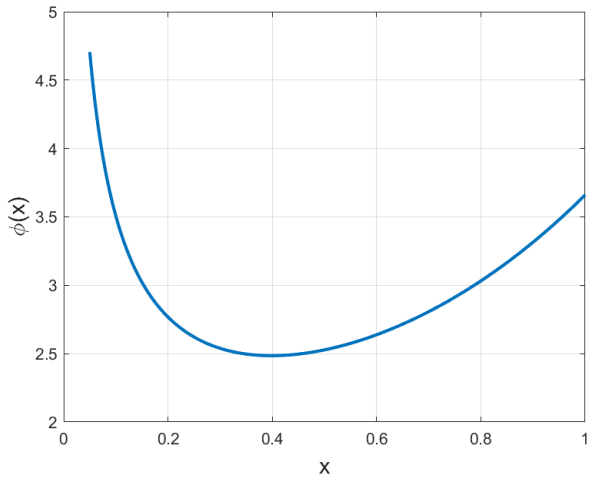

Fig. 4. The Function $\phi(x)$. As we move away from the minimum in either direction, the function grows for both smaller and larger values of $x = \frac{C}{b2^f}$.

In summary, our estimator achieves a good accuracy unless the cardinality is too small or too large compared to $b \cdot 2^f$. The first case is unusual for a filter that is used precisely when the negative queries are more frequent than the positives such that the cardinality of the negative queries is typically much larger than the number of elements stored in the filter. As for the second case, when the cardinality is very large, the benefits of adaptation are smaller as there are so many negatives per cell that the filter keeps adapting and does not reach a state where false positives can be eliminated. Therefore, our estimator covers the range of cardinalities in settings on which adaptive filters are of interest.

D. Applicability to More Than One Selector Bit $s_b > 1$

Although it would seem that similar schemes could be used to estimate the cardinality for ACFs that use more than one

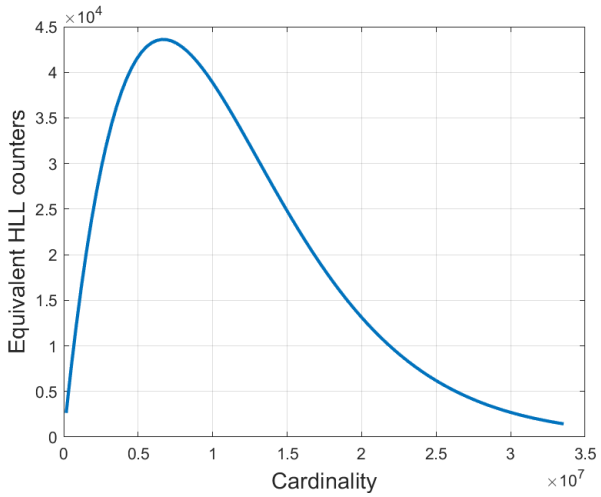

Fig. 5. Number of counters of an HLL structure with the same estimation error for a filter with $b = 65536, d = 4, f = 8$ and 95% occupancy as a function of the cardinality.

selector bit $s_b > 1$, that is not the case. The key insight is that when there are multiple selector bits, their values depend on the number of elements. Thus sets with the same distinct count but different numbers of elements would have different distributions of the selector bit values. This is best illustrated with an example. Suppose $s_b = 2$ and that we have a given query set such that there is exactly one query falsely matching each of the four possible fingerprints (i.e., $l_0 = l_1 = l_2 = l_3 = 1$). The probability of having a value of zero for the selector bits after this particular set has been queried in the filter would be that of those elements being queried in the order l_0, l_1, l_2, l_3 which is one of twenty-four possible permutations. Instead, if having the same distinct count, the number of elements is $l_0 = l_1 = l_2 = l_3 = 1000$ or any large number, the probability would be closer to one-fourth, and as the number of elements is increased when the number of distinct elements is kept constant, all selector states will become asymptotically equally likely. Therefore, the adaptation state depends not only on the distinct count of the elements but also on the number of elements and thus cannot be used to provide a meaningful estimator of the cardinality. This means that cardinality estimation can only be implemented when $s_b = 1$.

E. Benefits of Integrating Cardinality Estimation on the Filter

Table II models the speed of queries and cardinality estimations in terms of memory accesses across two baselines: an independent filter and Hyperloglog versus our integrated approach[5]. Integration reduces the cost of queries (which are the most frequent operations in our target application) as no Hyperloglog needs to be accessed. On the other hand, the cost of cardinality estimation is higher as all the filter cells have to be read compared to reading all the counters of a Hyperloglog. However, cardinality estimates are typically computed less frequently. Flow monitoring, for example, entails performing tens of millions of queries per second while the number of distinct flows may be estimated and reported only once or a few times per second. Therefore, the cost of computing the cardinality estimate is not an issue. The bottleneck is rather the query path. The last row of Table II further shows that our integrated approach saves memory as there is no Hyperloglog. In summary, the integration has benefits in terms of memory bandwidth and memory footprint compared to using an independent filter and cardinality estimator.

[5]We assume the filter to be also an ACF to facilitate the comparison.

TABLE II

COMPARISON OF INTEGRATED AND INDEPENDENT FILTERING AND CARDINALITY ESTIMATION. A FILTER WITH d TABLES OF b BUCKETS OF SIZE f AND A CARDINALITY ESTIMATOR WITH M COUNTERS OF SIZE c ARE USED IN THE COMPARISON

Parameter	Independent	Integrated
Memory Accesses per Query	$d+1$	d
Memory Accesses per Estimate	M	$b \cdot d$
Memory Footprint (bits)	$b \cdot d \cdot f + M \cdot c$	$b \cdot d \cdot f$

F. Implementation on Hardware Platforms

The proposed CE-ACF is not tied to any particular implementation platform, though additional benefits can arise on certain platforms. For example, on programmable data planes, implementing HyperLogLog requires additional Ternary Content Addressable Memory (TCAM) blocks to count the number of leading zeros. This entails additional complexity [39], which our design obviates as it can perform cardinality estimation without the use of a Hyperloglog.

Furthermore, our design is well aligned with the increasing use of cuckoo hashing in modern switches [2], [3] as it is also based on cuckoo hashing. The implementation of cuckoo hashing in hardware can be done efficiently both in ASICs [3] and in FPGAs [45]. In P4 programmable switches, swapping of elements during insertions may introduce a significant bandwidth overhead. However, recent works suggest that new higher level approaches to code the programs can reduce the swapping overhead so that its impact on the switch bandwidth is marginal [46]. Even without those optimizations, cuckoo filter variants have been successfully implemented in P4 programmable data planes [47], and so our design is applicable for such platforms as well.

IV. EVALUATION

The proposed scheme has been implemented[6] and tested in different configurations both with randomly generated data and with packet traces. The use of random data enables us to explore how different parameters affect the cardinality estimation while the packet traces are used as a case study to validate the proposed scheme for a flow monitoring application. Finally, the proposed CE-ACF is compared to an alternative solution using an ACF for filtering and a HyperLogLog for cardinality estimation.

A. Synthetic Data

In the first experiment, filters with $d = 4, b = 1024, f = 7, 11, s = 1$ have been created and filled up to 95% occupancy.

[6]The code is available at https://github.com/aalonsog/ACF/tree/cestimate/reverse.

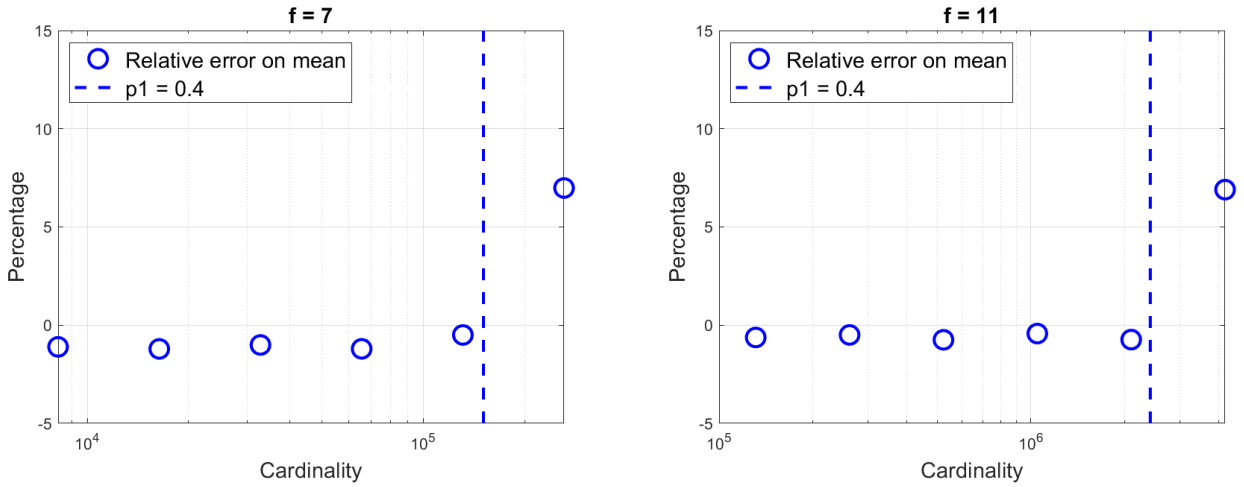

Fig. 6. Results for the first experiment in terms of relative deviation from the true cardinality value of the mean estimate over all runs for $f = 7$ (left) and $f = 11$ (right). The cardinality that corresponds to $p_1 = 0.4$ is also shown for reference as a dotted vertical line. The estimate is much worse when that value is exceeded.

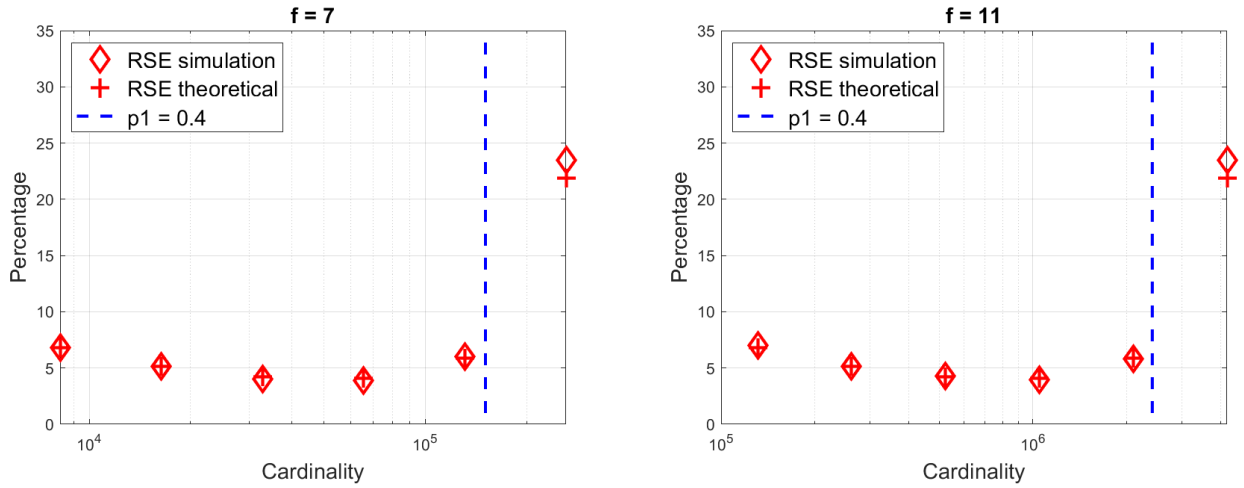

Fig. 7. Results for the first experiment in terms of the relative standard error. The relative standard error given by equation 8 is also shown as well as the cardinality that corresponds to $p_1 = 0.4$ as a dotted vertical line. The estimate is much worse when that value is exceeded.

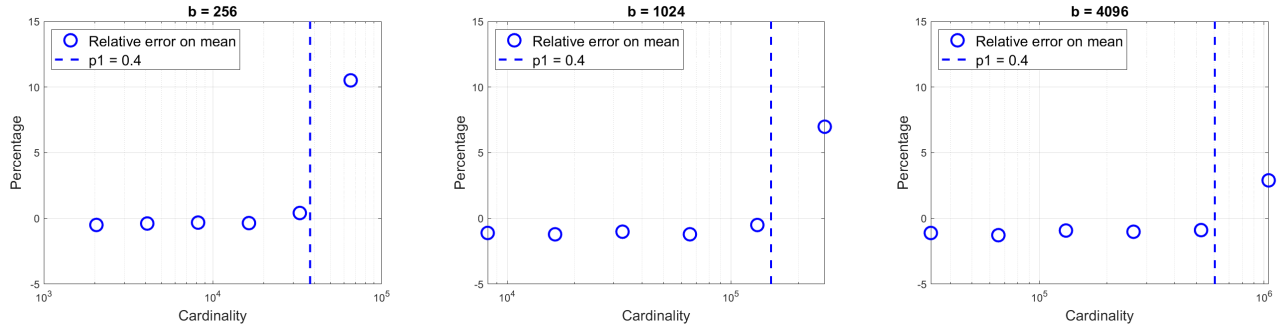

Fig. 8. Results for the second experiment in terms of relative deviation from the true cardinality value of the mean estimate over all runs for $b = 256$ (left), $b = 1024$ (middle) and $b = 4096$ (right).

Then, a set of N elements taken randomly and uniformly from a set of C unique elements is queried on the filter. Finally, cardinality is estimated using equation 4. This process is repeated across 1000 experimental runs. The mean and relative standard error are then computed. This has been done for different values of C. In each run, each negative element is queried on average five times.[7] The results are shown in Figures 6 and 7. The vertical line in these figures corresponds to the cardinality for which $p_1 = 0.4$ (i.e., getting close to 0.5 of the cells having a selector bit with a value of one). These measurements, therefore, mark the start of the range of cardinalities for which the proposed estimator exhibits diminishing accuracy. It can be seen that the cardinality estimate has low relative deviation on the mean so it seems to have a small bias as discussed in [43]. Figure 7 shows that the relative standard error is small for a wide range of cardinality values. Finally, the theoretical estimate is accurate

[7]Note that the number of times that elements are queried on the filter should not affect the cardinality estimate.

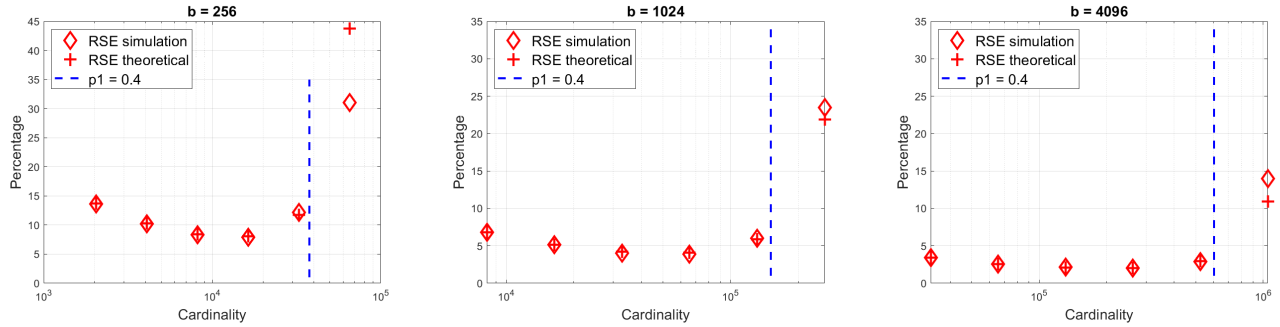

Fig. 9. Results for the second experiment in terms of the relative standard error (the theoretical value of the relative standard error given by equation 8 is also shown as well as the cardinality that corresponds to $p_1 = 0.4$ for reference) for $b = 256$ (left), $b = 1024$ (middle) and $b = 4096$ (right).

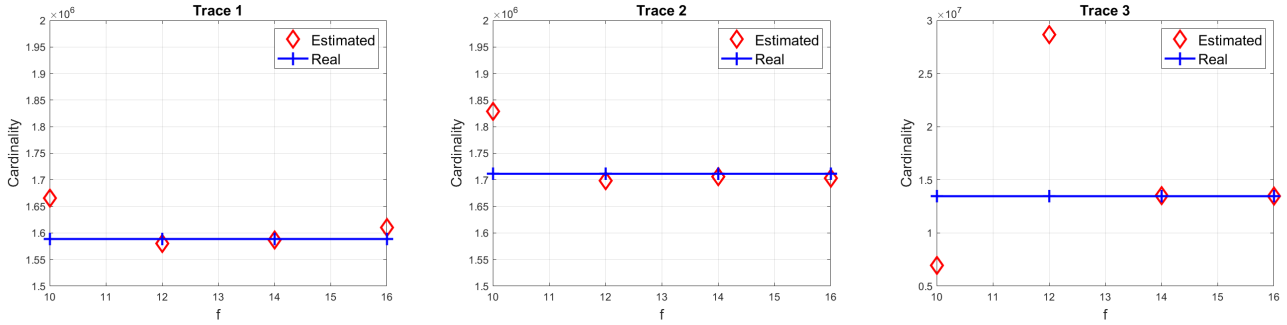

Fig. 10. Results for the CAIDA traces in terms of the average of the cardinality estimates. The true cardinality is also shown for reference. Trace 1 (left), Trace 2 (middle) and Trace 3 (right).

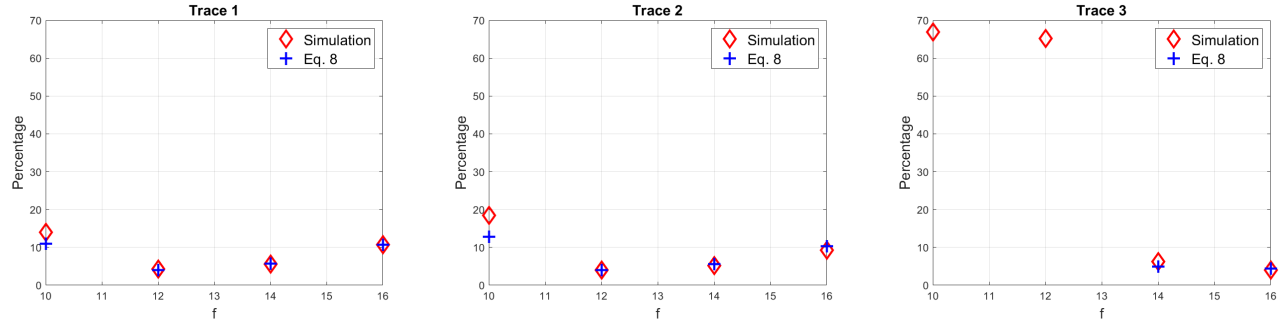

Fig. 11. Results for the CAIDA traces in terms of the relative standard error (the theoretical value of the relative standard error given by equation 8 is also shown). Trace 1 (left), Trace 2 (middle) and Trace 3 (right).

when the cardinality is below the cardinality that corresponds to $p_1 = 0.4$ as expected.

The second experiment explores the impact of the filter size on the accuracy of the estimator. To do so, filters with $d = 4, b = 256, 1024, 4096, f = 7, s = 1$ have been created and filled up to 95% occupancy. Then, the same procedure as that of the first experiment is carried out. The results are summarized in Figures 8 and 9. It can be clearly seen that as b and thus the filter size increases, so does the accuracy of the estimator as predicted by equation 8.

B. Packet Traces

To validate the proposed cardinality estimation in a realistic flow monitoring scenario, packet traces from CAIDA have been used. The traces have approximately 33.4, 37.8 and 32.6 million packets and 1,588,650, 1,711,354, 13,459,419 flows respectively.

Filters with $d = 4, b = 1024, f = 10, 12, 14, 16, s = 1$ have been constructed for each of the traces and filled with the first flows of each trace until a 95% occupancy is reached. Then, all packets on the trace are checked on the filter and the cardinality is estimated based on the final filter state. The process has been repeated one hundred times using different hash functions.

The results are presented in Figures 10, 11. It can be observed that the estimates are accurate for the first two traces when the fingerprints have at least 12 bits. Instead for the third trace, 14 bits are needed to get accurate estimates as it has many more flows. This is consistent with the analysis presented in the previous section. These results confirm that the ACF can be used to estimate the cardinality of the negative elements queried on the filter provided that it is in the range estimated theoretically in the previous section, which depends on both b and f.

C. Comparison With Alternative Solutions

Finally, it is of interest to compare the proposed CE-ACF with the use of an independent filter and cardinality estimator. As discussed before, for the filter we use the same ACF and for the cardinality estimation a HyperLogLog. In more detail,

TABLE III
COMPARISON OF THE PROPOSED CE-ACF WITH AN INDEPENDENT ACF AND HLL FOR DIFFERENT METRICS

	Memory accesses	Hash functions	Memory
Independent ACF and HLL	5	8	68 Kb
Proposed CE-ACF	4	6	61 Kb
Savings	20%	25%	9.1%

TABLE IV
TIME (IN SECONDS) NEEDED TO PROCESS THE PACKET TRACES WITH THE CE-ACF VS AN INDEPENDENT ACF AND HLL

Trace	ACF+HLL	CE-ACF	Savings
Trace 1	1.82	1.53	15.9%
Trace 2	1.76	1.44	18.1%
Trace 3	1.96	1.68	14.3%

we use $M = 1024$ counters of $c = 6$ bits. The main metrics for comparison are summarized in Table III. The ACFs have in both cases four tables of 1024 buckets and fingerprints of 14 bits. The memory accesses and hash functions are for negative queries, which are dominant in filter workloads. The use of the CE-ACF provides significant savings in both the memory bandwidth needed and the number of hash functions. The savings in memory footprint are smaller but not negligible.

Additionally, the three packet traces have been processed using the proposed CE-ACF and an independent ACF and HLL from Apache DataSketches.[8] All packets are queried on the filter and the estimation of the cardinality is run every second. The samples were run 10 times and the average speed was taken.[9] The run times for each trace are shown in Table IV. It can be observed that the proposed CE-ACF reduces the time needed significantly. This confirms that the savings in the per packet processing when using a single data structure are much larger than the increase in computing the cardinality estimates. In summary, these results confirm that the CE-ACF is attractive for efficient implementation in high-speed flow monitoring applications.

V. CONCLUSION AND FUTURE WORK

In this paper, we have shown that adaptive cuckoo filters can be used to estimate cardinality as part of their normal operation. In more detail, the values of the selector bit in the filter contain information on the cardinality of the set of negative elements that have been checked in the filter. A cardinality estimator that uses that information has been proposed, analyzed, and evaluated, showing that it can provide accurate estimates over a wide range of cardinality values. This is of interest for applications on which both approximate membership check and cardinality estimation are needed as both functions can be implemented in a single data structure, reducing memory and computing overheads.

The intuition that led to this paper, namely that the state of an adaptive cuckoo filter can be used to estimate cardinality of the negative elements queried on the filter, is applicable to adaptive filters in general. This is because an adaptation happens when a false positive is detected and removed, and thus adaptations can be used to estimate the number of distinct false positives and thus of elements. Therefore, deriving cardinality estimators for other adaptive filters like the Telescoping [21] or adaptive one memory access Bloom filters [22] is an interesting direction for future work.

[8]https://datasketches.apache.org/

[9]The experiments were run on an Intel Core i7-7800X running Ubuntu 18.04.6 LTS and Clang version 16.0.0.

REFERENCES

[1] R. Hofstede et al., "Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 2037–2064, 4th Quart., 2014.

[2] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance Ethernet forwarding with CuckooSwitch," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol.*, New York, NY, USA, Dec. 2013, pp. 97–108, doi: 10.1145/2535372.2535379.

[3] G. Levy, S. Pontarelli, and P. Reviriego, "Flexible packet matching with single double cuckoo hash," *IEEE Commun. Mag.*, vol. 55, no. 6, pp. 212–217, Jun. 2017.

[4] D. Kim et al., "TEA: Enabling state-intensive network functions on programmable switches," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 90–106.

[5] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. ACM Int. Conf. Manage. Data*, May 2017, pp. 79–94.

[6] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 505–520.

[7] N. Dayan and S. Idreos, "The log-structured merge-bush & the Wacky continuum," in *Proc. Int. Conf. Manage. Data*, Jun. 2019, pp. 449–466.

[8] N. Dayan and M. Twitto, "Chucky: A succinct cuckoo filter for LSM-tree," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 365–378.

[9] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," *IEEE/ACM Trans. Netw.*, vol. 14, no. 2, pp. 397–409, Apr. 2006.

[10] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing Bloom filter: Challenges, solutions, and comparisons," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1912–1949, 2nd Quart., 2019.

[11] S. Pontarelli, P. Reviriego, and M. Mitzenmacher, "EMOMA: Exact match in one memory access," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 11, pp. 2120–2133, Nov. 2018.

[12] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[13] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2014, pp. 75–88.

[14] T. M. Graf and D. Lemire, "Xor filters," *ACM J. Experim. Algorithmics*, vol. 25, pp. 1–16, Mar. 2020, doi: 10.1145/3376122.

[15] M. A. Bender et al., "Don't thrash: How to cache your hash on flash," in *Proc. 3rd Workshop Hot Topics Storage File Syst.*, 2011, pp. 1627–1637.

[16] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proc. ACM Int. Conf. Manage. Data*, May 2017, pp. 775–787.

[17] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. Johnson, "Vector quotient filters: Overcoming the time/space trade-off in filter design," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 1386–1399.

[18] P. C. Dillinger, L. Hübschle-Schneider, P. Sanders, and S. Walzer, "Fast succinct retrieval and approximate membership using ribbon," 2021, *arXiv:2109.01892*.

[19] P. C. Dillinger and S. Walzer, "Ribbon filter: Practically smaller than Bloom and Xor," 2021, *arXiv:2103.02515*.

[20] M. A. Bender, M. Farach-Colton, M. Goswami, R. Johnson, S. McCauley, and S. Singh, "Bloom filters, adaptivity, and the dictionary problem," in *Proc. IEEE 59th Annu. Symp. Found. Comput. Sci. (FOCS)*, Oct. 2018, pp. 182–193.

[21] D. J. Lee, S. McCauley, S. Singh, and M. Stein, "Telescoping filter: A practical adaptive filter," in *Proc. ESA*, 2021, pp. 60:1–60:18. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14641/

[22] P. Reviriego, A. Sánchez-Macián, O. Rottenstreich, and D. Larrabeiti, "Adaptive one memory access Bloom filters," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 2, pp. 848–859, Jun. 2022.

[23] M. D. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," in *Proc. Workshop Algorithm Eng. Exp. (ALENEX)*, 2018, pp. 1–20.

[24] L. Zheng, D. Liu, W. Liu, Z. Liu, Z. Li, and T. Wu, "A data streaming algorithm for detection of superpoints with small memory consumption," *IEEE Commun. Lett.*, vol. 21, no. 5, pp. 1067–1070, May 2017.

[25] Q. Xiao et al., "Cardinality estimation for elephant flows: A compact solution based on virtual register sharing," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3738–3752, Dec. 2017.

[26] D. Ding, M. Savi, F. Pederzolli, M. Campanella, and D. Siracusa, "In-network volumetric DDoS victim identification using programmable commodity switches," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 2, pp. 1191–1202, Jun. 2021.

[27] A. Sallam, D. Fadolalkarim, E. Bertino, and Q. Xiao, "Data and syntax centric anomaly detection for relational databases," *Wiley Interdiscipl. Rev., Data Mining Knowl. Discovery*, vol. 6, no. 6, pp. 231–239, 2016.

[28] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proc. 16th Int. Conf. Extending Database Technol.*, New York, NY, USA, Mar. 2013, pp. 683–692, doi: 10.1145/2452376.2452456.

[29] H. Harmouch and F. Naumann, "Cardinality estimation: An experimental survey," *Proc. VLDB Endowment*, vol. 11, no. 4, pp. 499–512, Dec. 2017, doi: 10.1145/3186728.3164145.

[30] M. Kadosh, G. Levy, Y. Shpigelman, O. Shabtai, Y. Piasetzky, and L. Mula, "Cardinality-based traffic control," U.S. Patent 17/335 312, Dec. 1, 2022.

[31] P. Flajolet, E. Fusy, and O. Gandouet, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. Int. Conf. Anal. Algorithms (AOFA)*, 2007, pp. 127–146.

[32] L. Guo and I. Matta, "The war between mice and elephants," in *Proc. 9th Int. Conf. Netw. Protocols*, 2001, pp. 180–188.

[33] V. Srinivasan et al., "AeroSpike: Architecture of a real-time operational DBMS," *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1389–1400, Sep. 2016.

[34] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "FASTER: A concurrent key-value store with in-place updates," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 1930–1933.

[35] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 208–229, Jun. 1990, doi: 10.1145/78922.78925.

[36] F. Giroire, "Order statistics and estimating cardinalities of massive data sets," *Discrete Appl. Math.*, vol. 157, no. 2, pp. 406–427, Jan. 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166218X08002813

[37] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *Proc. 6th Int. Workshop Randomization Approximation Techn.* Berlin, Germany: Springer-Verlag, 2002, pp. 1–10.

[38] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Proc. Eur. Symp. Algorithms (ESA)*, 2003, pp. 605–617.

[39] D. Ding, M. Savi, F. Pederzolli, and D. Siracusa, "INVEST: Flow-based traffic volume estimation in data-plane programmable networks," in *Proc. IFIP Netw. Conf.*, Jun. 2021, pp. 1–9.

[40] H. Namkung, D. Kim, Z. Liu, V. SekaR, and P. Steenkiste, "Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations," in *Proc. ACM SIGCOMM Symp. SDN Res. (SOSR)*, Oct. 2021, pp. 176–182.

[41] K. Kim, J. Jung, I. Seo, W.-S. Han, K. Choi, and J. Chong, "Learned cardinality estimation: An in-depth study," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2022, pp. 1214–1227, doi: 10.1145/3514221.3526154.

[42] R. Cohen and Y. Nezri, "Cardinality estimation in a virtualized network device using online machine learning," *IEEE/ACM Trans. Netw.*, vol. 27, no. 5, pp. 2098–2110, Oct. 2019.

[43] M. Mitzenmacher, R. Pagh, and N. Pham, "Efficient estimation for high similarities using odd sketches," in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 109–118.

[44] G. Casella and R. L. Berger, *Statistical Inference*, 2nd ed. Pacific Grove, CA, USA: Duxbury, 2002.

[45] S. Pontarelli et al., "FlowBlaze: Stateful packet processing in hardware," in *Proc. 16th USENIX Symp. Networked Syst. Design Implement.* Boston, MA, USA: USENIX Assoc., Feb. 2019, pp. 531–548. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pontarelli

[46] J. Sonchack, D. Loehr, J. Rexford, and D. Walker, "Lucid: A language for control in the data plane," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 731–747, doi: 10.1145/3452296.3472903.

[47] Q. Xiao, H. Wang, and G. Pan, "Accurately identify time-decaying heavy hitters by decay-aware cuckoo filter along kicking path," in *Proc. IEEE/ACM 30th Int. Symp. Quality Service (IWQoS)*, Jun. 2022, pp. 1–10.

Pedro Reviriego received the M.Sc. and Ph.D. degrees in telecommunications engineering from the Technical University of Madrid, Madrid, Spain, in 1994 and 1997, respectively. From 1997 to 2000, he was an Engineer with Teldat, Madrid, where he is working on router implementation. In 2000, he joined Massana to work on the development of 1000BASE-T transceivers. From 2004 to 2007, he was a Distinguished Member of Technical Staff with the LSI Corporation, where he is working on the development of ethernet transceivers. From 2007 to 2018, he was with Nebrija University and with Universidad Carlos III de Madrid from 2018 to 2022. He is currently with Universidad Politcnica de Madrid, where he is working on probabilistic data structures, high-speed packet processing, and machine learning.

Jim Apple received the M.S. degree in computer and information science from the University of Oregon. His current research interests include hashing and hashing-based data structures.

Alvaro Alonso received the M.Sc. and Ph.D. degrees in telecommunications engineering from Universidad Politécnica de Madrid, Spain, in 2013 and 2016, respectively. He is currently an Associate Professor with Universidad Politcnica de Madrid. His current research interests include public open data, security management in smart context environments, and multi-videoconferencing systems.

Otmar Ertl received the Ph.D. degree in technical sciences from the Vienna University of Technology in 2010. He is currently a Lead Researcher with Dynatrace Research. His current research interests include probabilistic algorithms and data structures.

Niv Dayan received the M.Sc. and Ph.D. degrees from the IT University of Copenhagen. He is currently an Assistant Professor with the University of Toronto. Prior to that, he was a Research Scientist with Pliops and a Post-Doctoral Researcher with Harvard. His current research interests include the design and analysis of storage engines and their core data structures.