Telescoping Filter: A Practical Adaptive Filter

David J. Lee ✉
Cornell University, Ithaca, NY, USA

Samuel McCauley ✉ 🏠
Williams College, Williamstown, MA, USA

Shikha Singh ✉ 🏠
Williams College, Williamstown, MA, USA

Max Stein ✉
Williams College, Williamstown, MA, USA

Abstract

Filters are small, fast, and approximate set membership data structures. They are often used to filter out expensive accesses to a remote set $\mathcal{S}$ for negative queries (that is, filtering out queries $x \notin \mathcal{S}$). Filters have one-sided errors: on a negative query, a filter may say "present" with a tunable false-positive probability of ε. Correctness is traded for space: filters only use $\log(1/\varepsilon) + O(1)$ bits per element.

The false-positive guarantees of most filters, however, hold only for a single query. In particular, if x is a false positive, a subsequent query to x is a false positive with probability 1, not ε. With this in mind, recent work has introduced the notion of an **adaptive filter**. A filter is adaptive if each query is a false positive with probability ε, regardless of answers to previous queries. This requires "fixing" false positives as they occur.

Adaptive filters not only provide strong false positive guarantees in adversarial environments but also improve query performance on practical workloads by eliminating repeated false positives.

Existing work on adaptive filters falls into two categories. On the one hand, there are practical filters, based on the cuckoo filter, that attempt to fix false positives heuristically without meeting the adaptivity guarantee. On the other hand, the broom filter is a very complex adaptive filter that meets the optimal theoretical bounds.

In this paper, we bridge this gap by designing the **telescoping adaptive filter** (TAF), a practical, provably adaptive filter. We provide theoretical false-positive and space guarantees for our filter, along with empirical results where we compare its performance against state-of-the-art filters. We also implement the broom filter and compare it to the TAF . Our experiments show that theoretical adaptivity can lead to improved false-positive performance on practical inputs, and can be achieved while maintaining throughput that is similar to non-adaptive filters.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Filters, approximate-membership query data structures (AMQs), Bloom filters, quotient filters, cuckoo filters, adaptivity, succinct data structures

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.60

Related Version *Full Version*: `https://arxiv.org/abs/2107.02866`

Supplementary Material *Software*: `https://github.com/djslzx/telescoping-filter`
archived at `swh:1:dir:76164f338679e3e058408d6a5572d400e2f03977`

Funding *David J. Lee*: This author's research is supported in part by NSF CCF 1947789.
Samuel McCauley: This author's research is supported in part by NSF CCF 2103813
Shikha Singh: This author's research is supported in part by NSF CCF 1947789.
Max Stein: This author's research is supported in part by NSF CCF 1947789.

29th Annual European Symposium on Algorithms (ESA 2021).
Editors: Petra Mutzel, Rasmus Pagh, and Grzegorz Herman; Article No. 60; pp. 60:1–60:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A filter is a compact and probabilistic representation of a set $\mathcal{S}$ from a universe $\mathcal{U}$. A filter supports insert and query operations on $\mathcal{S}$. On a query for an element $x \in \mathcal{S}$, a filter returns "present" with probability 1, i.e., a filter guarantees no false negatives. A filter is allowed to have bounded false positives – on a query for an element $x \notin \mathcal{S}$, it may incorrectly return "present" with a small and tunable false-positive probability ε.

Filters are used because they allow us to trade correctness for space. A lossless representation of $\mathcal{S} \subseteq \mathcal{U}$ requires $\Omega(n \log u)$ bits, where $n = |\mathcal{S}|$, $u = |\mathcal{U}|$, and $n \ll u$. Meanwhile, an optimal filter with false-positive probability ε requires only $\Theta(n \log(1/\varepsilon))$ bits [11].

Examples of classic filters are the *Bloom filter* [7], the *cuckoo filter* [18], and the *quotient filter* [6]. Recently, filters have exploded in popularity due to their widespread applicability – many practical variants of these classic filters have been designed to improve upon throughput, space efficiency, or cache efficiency [8, 16, 19, 22, 29, 32].

A filter's small size allows it to fit in fast memory, higher in the memory hierarchy than a lossless representation of $\mathcal{S}$ would allow. For this reason, filters are frequently used to speed up expensive queries to an external dictionary storing $\mathcal{S}$.

In particular, when a dictionary for $\mathcal{S}$ is stored remotely (on a disk or across a network), checking a small and fast filter first can avoid expensive remote accesses for a $1 - \varepsilon$ fraction of negative queries. This is the most common use case of the filter, with applications in LSM-based key-value stores [12, 23, 27], databases [13, 15, 17], and distributed systems and networks [9, 31].

False positive guarantees and adaptivity. When a filter is used to speed up queries to a remote set $\mathcal{S}$, its performance depends on its false-positive guarantees: how often does the filter make a mistake, causing us to access $\mathcal{S}$ unnecessarily?

Many existing filters, such as the Bloom, quotient and cuckoo filters, provide poor false-positive guarantees because they hold only for a single query. Because these filters do not *adapt*, that is, they do not "fix" any false positives, querying a known false positive x repeatedly can drive their false-positive rate to 1, rendering the filter useless.

Ideally, we would like a stronger guarantee: even if a query x has been a false positive in the past, a subsequent query to x is a false positive with probability at most ε. This means that the filter must "fix" each false positive x as it occurs, so that subsequent queries to x are unlikely to be false positives. This notion of adaptivity was formalized by Bender et al. [5]. A filter is **adaptive** if it guarantees a false positive probability of ε for every query, *regardless of answers to previous queries.* Thus, adaptivity provides security advantages against an adversary attempting to degrade performance, e.g., in denial-of-service attacks.

At the same time, fixing previous false positives leads to improved performance. Many practical datasets do, in fact, repeatedly query the same element – on such a dataset, fixing previous false positives means that a filter only incurs one false positive per *unique* query. Past work has shown that simple, easy-to-implement changes to known filters can fix false positives heuristically. Due to repeated queries, these heuristic fixes can lead to reduction of several orders of magnitude in the number of incurred false positives [10, 21, 25].

Recent efforts that tailor filters to query workloads by applying machine learning techniques to optimize performance [14, 24, 30] reinforce the benefits achieved by adaptivity.

Adaptivity vs practicality. The existing work on adaptivity represents a dichotomy between simple filters one would want to implement and use in practice but are not actually adaptive [21, 25], or adaptive filters that are purely theoretical and pose a challenge to implementation [5].

Mitzenmacher et al. [25] provided several variants of the *adaptive cuckoo filter* (ACF) and showed that they incurred significantly fewer false positives (compared to a standard cuckoo filter) on real network trace data. The data structures in [25] use simple heuristics to fix false positives with immense practical gains, leaving open the question of whether such heuristics can achieve worst-case guarantees on adaptivity.

Recently, Kopelowitz et al. [21] proved that this is not true even for a non-adversarial notion of adaptivity. In particular, they defined **support optimality** as the adaptivity guarantee on "predetermined" query workloads: that is, query workloads that are fixed ahead of time and not constructed in response to a filter's response on previous queries. They showed that the filters in [25] fail to be adaptive even under this weaker notion – repeating $O(1)$ queries n times may cause them to incur $\Omega(n)$ false positives.

Kopelowitz et al. [21] proposed a simple alternative, the *cuckooing ACF*, that achieves support optimality by cuckooing on false positives (essentially reinserting the element). Furthermore, they proved that none of the cuckoo filter variants (including the cuckooing ACF) are adaptive. They showed that a prerequisite to achieving adaptivity is allocating a variable number of bits to each stored element – that is, maintaining variable-length fingerprints. All of the cuckooing filter variants use a bounded number of bits per element.

The only known adaptive filter is the **broom filter** of Bender et al. [5], so-named because it "cleans up" its mistakes. The broom filter achieves adaptivity, while supporting constant-time worst-case query and insert costs, using very little extra space – $O(n)$ extra bits in total. Thus the broom filter implies that, *in theory*, adaptivity is essentially free.

More recently, Bender et al. [4] compared the broom filter [5] to a static filter augmented with a comparably sized top-k cache (a cache that stores the k most frequent requests). They found that the broom filter outperforms the cache-augmented filter on Zipfian distributions due to "serendipitous corrections" – fixing a false positive eliminates future false positives in addition to the false positive that triggered the adapt operation. They noted that their broom filter simulation is "quite slow," and left open the problem of designing a practical broom filter with performance comparable to that of a quotient filter.

In this paper, we present a practical and efficient filter which also achieves worst-case adaptivity: the **telescoping adaptive filter**. The key contribution of this data structure is a practical method to achieve worst-case adaptivity using variable-length fingerprints.

Telescoping adaptive filter. The telescoping adaptive filter (TAF) combines ideas from the heuristics used in the adaptive cuckoo filter [25], and the theoretical adaptivity of the broom filter [5].

The TAF is built on a rank-and-select quotient filter (RSQF) [29] (a space- and cache-efficient quotient filter [6] variant), and inherits its performance guarantees.

The telescoping adaptive filter is the first adaptive filter that can take advantage of *any* amount of extra space for adaptivity, even a fractional number of bits per element. We prove that if the TAF uses $\left(\frac{1}{e} + \frac{b}{(1-b)^2}\right)$ extra bits per element in expectation, then it is is provably adaptive for any workload consisting of up to $n/(b\sqrt{\varepsilon})$ unique queries (Section 4). Empirically, we show that the TAF outperforms this bound: with only 0.875 of a bit extra per element for adaptivity, it is adaptive for larger query workloads. Since the RSQF uses 2.125 metadata bits per element, the total number of bits used by the TAF is $(n/\alpha)(\log_2(1/\varepsilon)+3)$, where α is the load factor.

The TAF stores these extra bits space- and cache-efficiently using a practical implementation of a theoretically-optimal compression scheme: *arithmetic coding* [20, 33]. Arithmetic coding is particularly well-suited to the exponentially decaying probability distribution of repeated false positives. While standard arithmetic coding on the unit interval can be slow, we implement an efficient approximate integer variant.

The C code for our implementation can be found at `https://github.com/djslzx/telescoping-filter`.

Our contributions. We summarize our main contributions below.

- We present the first provably-adaptive filter, the telescoping adaptive filter, engineered with space, cache-efficiency and throughput in mind, demonstrating that adaptivity is not just a theoretical concept, and can be achieved in practice.
- As a benchmark for TAF, we also provide a practical implementation of the broom filter [5]. We call our implementation of the broom filter the **extension adaptive filter** (exAF). While both TAF and exAF use the near-optimal $\Theta(n)$ extra bits in total to adapt on $\Theta(n)$ queries, the telescoping adaptive filter is optimized to achieve better constants and eke out the most *adaptivity per bit.* This is confirmed by our experiments which show that given the same space for adaptivity (0.875 bits per element), the TAF outperforms the false-positive performance of the exAF significantly on both practical and adversarial workloads. Meanwhile, our experiments show that the query performance of exAF is factor 2 better than that of the TAF . Thus, we show that there is a trade off between throughput performance and how much adaptivity is gained from each bit.
- We give the first empirical evaluation of how well an adaptive filter can fix positives in practice. We compare the TAF with a broom filter implementation, as well as with previous heuristics. We show that the TAF frequently matches or outperforms other filters, while it is especially effective in fixing false positives on "difficult" datasets, where repeated queries are spaced apart by many other false positives. We also evaluate the throughput of the TAF and the exAF against the vacuum filter [32] and RSQF, showing for the first time that adaptivity can be achieved while retaining good throughput bounds.

2 Preliminaries

In this section, we provide background on filters and adaptivity, and describe our model.

2.1 Background on Filters

We briefly summarize the structure of the filters discussed in this paper. For a more detailed description, we refer the reader to the full version. All logs in the paper are base 2. We assume that ε is an inverse power of 2.

The quotient filter and cuckoo filter are both based on the single-hash function filter [28]. Let the underlying hash function h output $\Theta(\log n)$ bits. To represent a set $\mathcal{S} \subseteq \mathcal{U}$, the filter stores a fingerprint $f(x)$ for each element $x \in \mathcal{S}$. The fingerprint $f(x)$ consists of the first $\log n + \log(1/\varepsilon)$ bits of $h(x)$, where $n = |\mathcal{S}|$ and ε is the false-positive probability.

The first $\log n$ bits of $f(x)$ are called the **quotient** $q(x)$ and are stored implicitly; the remaining $\log(1/\varepsilon)$ bits are called the **remainder** $r(x)$ and are stored explicitly in the data structure. Both filters consist of an array of slots, where each slot can store one remainder.

Quotient filter. The quotient filter (QF) [6] is based on linear probing. To insert $x \in \mathcal{S}$, the remainder $r(x)$ is stored in the slot location determined by the quotient $q(x)$, using linear probing to find the next empty slot. A small number of metadata bits suffice to recover the original slot for each stored element. A query for x checks if the remainder $r(x)$ is stored in the filter – if the remainder is found, it returns "present"; otherwise, it returns "absent." The rank-and-select quotient filter (RSQF) [29] implements such a scheme using very few metadata bits (only 2.125 bits) per element.

Broom filter. The **broom filter** of Bender et al. [5] is based on the quotient filter. Initially, it stores the same fingerprint $f(x)$ as a quotient filter. The broom filter uses the remaining bits of $h(x)$, called **adaptivity bits**, to extend $f(x)$ in order to adapt on false positives.

On a query y, if there is an element $x \in \mathcal{S}$ such that $f(x)$ is a prefix of $h(y)$, the broom filter returns "present." If it turns out that $y \notin \mathcal{S}$, the broom filter adapts by extending the fingerprint $f(x)$ until it is no longer a prefix of $h(y)$.[1] Bender et al. show that, with high probability, $O(n)$ total adaptivity bits of space are sufficient for the broom filter to be adaptive on $\Theta(n)$ queries.

Cuckoo filters and adaptivity. The cuckoo filter resembles the quotient filter but uses a cuckoo hash table rather than linear probing. Each element has two fingerprints, and therefore two quotients. The remainder of each $x \in \mathcal{S}$ must always be stored in the slot corresponding to one of x's two quotients.

The Cyclic ACF [25], Swapping ACF [25], and Cuckooing ACF [21][2] change the function used to generate the remainder on a false positive. To avoid introducing false negatives, a filter using this technique must somehow track which function was used to generate each remainder so that the appropriate remainders can be compared at query time.

The Cyclic ACF stores s extra bits for each slot, denoting which of 2^s different remainders are used. The Swapping ACF, on the other hand, groups slots into constant-sized bins, and has a fixed remainder function for each slot in a bin. A false positive is fixed by moving some $x \in \mathcal{S}$ to a different slot in the bin, then updating its remainder using the function corresponding to the new slot. The Cuckooing ACF works in much the same way, but both the quotient and remainder are changed by "cuckooing" the element to its alternate position in the cuckoo table.

2.2 Model and Adaptivity

All filters that adapt on false positives [5, 21, 25] have access to the original set $\mathcal{S}$. This is called the *remote representation*, denoted $\mathbf{R}$. The remote representation does not count towards the space usage of the filter. On a false positive, the filter is allowed to access the set $\mathbf{R}$ to help fix the false positive.

The justification for this model is twofold. (This justification is also discussed in [5,21,25].) First, the most common use case of filters is to filter out negative queries to $\mathcal{S}$ – in this case, a positive response to a query accesses $\mathbf{R}$ anyway. Information to help rebuild the filter can be stored alongside the set in this remote database. Second, remote access is necessary to achieve good space bounds: Bender et al. [5] proved that any adaptive filter without remote access to $\mathcal{S}$ requires $\Omega(n \log\log u)$ bits of space.

Our filter can answer queries using only the local state $\mathbf{L}$. Our filter accesses the remote state $\mathbf{R}$ in order to fix false positives when they occur, updating its local state. This allows our filter to be adaptive while using small (near optimal) space for the local state.

Adaptivity. The **sustained false positive** rate of a filter is the probability with which a query is a false positive, *regardless of the filter's answers to previous queries.*

[1] These additional bits are stored separately in the broom filter: groups of $\Theta(\log n)$ adaptivity bits, corresponding to $\log n$ consecutive slots in the filter, are stored such that accessing all the adaptivity bits of a particular element (during a query operation) can be done in $O(1)$ time.

[2] We use the nomenclature of [21] in calling these the Cyclic ACF, Swapping ACF, and Cuckooing ACF.

The sustained false positive rate must hold even if generated by an adversary. We use the definition of Bender et al. [5], which is formally defined by a game between an adversary and the filter, where the adversary's goal is to maximize the filter's false positive rate. We summarize this game next; for a formal description of the model see Bender et al. [5].

In the **adaptivity game**, the adversary generates a sequence of queries $Q = x_1, x_2, \ldots, x_t$. After each query x_i, both the adversary and filter learn whether x_i is a false positive (that is, $x_i \notin S$ but a query on x_i returns "present"). The filter is then allowed to adapt before query x_{i+1} is made by the adversary. The adversary can use the information about whether queries $x_1, \ldots, x_i$ were a false positive or not, to choose the next query x_{i+1}.

At any time t, the adversary may assert that it has discovered a special query $\tilde{x}_t$ that is likely to be a false positive of the filter. The adversary "wins" if $\tilde{x}_t$ is in fact a false positive of the filter at time t, and the filter "wins" if the adversary is wrong and $\tilde{x}_t$ is not a false positive of the filter at time t.

The **sustained false positive** rate of a filter is the maximum probability ε with which the adversary can win the above adaptivity game. A filter is **adaptive** if it can achieve a sustained false positive rate of ε, for any constant $0 < \varepsilon < 1$.

Similar to [5], we assume that the adversary cannot find a never-before-queried element that is a false positive of the filter with probability greater than ε. Many hash functions satisfy this property, e.g., if the adversary is a polynomial-time algorithm then one-way hash functions are sufficient [26]. Cryptographic hash functions satisfy this property in practice, and it is likely that even simple hash functions (like Murmurhash used in this paper) suffice for most applications.

Towards an adaptive implementation. Kopelowitz et al. [21] showed that the Cyclic ACF (with any constant number of hash-selector bits), the Swapping ACF, and the Cuckooing ACF are not adaptive. The key insight behind this proof is that for all three filters, the state of an element – which slot it is stored in, and which fingerprint function is used – can only have $O(1)$ values. Over $o(n)$ queries, an adversary can find queries that collide with an element on all of these states. These queries can never be fixed.

Meanwhile, the broom filter avoids this issue by allowing certain elements to have more than $O(1)$ adaptivity bits – up to $O(\log n)$, in fact. The broom filter stays space-efficient by maintaining $O(1)$ adaptivity bits per element *on average*.

Thus, a crucial step for achieving adaptivity is dynamically changing how much space is used for the adaptivity of each element based on past queries. The telescoping adaptive filter achieves this dynamic space allocation (hence the name "telescoping") using an arithmetic coding.

3 The Telescoping Adaptive Filter

In this section, we describe the high-level ideas behind the telescoping adaptive filter.

Structure of the telescoping adaptive filter. Like the broom filter, the TAF is based on a quotient filter where the underlying hash function h outputs $\Theta(\log n)$ bits. For any $x \in \mathcal{S}$, the first $\log n$ bits of $h(x)$ are the quotient $q(x)$ (stored implicitly), and the next $\log(1/\varepsilon)$ bits are the initial remainder $r_0(x)$, stored in the slot determined by the quotient. We maintain each element's original slot using the strategy of the rank-and-select quotient filter [29], which stores 2.125 metadata bits per element.

The TAF differs from a broom filter in that, on a false positive, the TAF *changes* its remainder rather than lengthening it, similar to the Cyclic ACF.

For each element in the TAF, we store a **hash-selector value**. If an element x has hash-selector value i, its remainder r_i is the consecutive sequence of $\log(1/\varepsilon)$ bits starting at the $(\log n + i \log(1/\varepsilon))$th bit of $h(x)$. Initially, the hash-selector values of all elements are 0, and thus the remainder $r(x)$ is the first $\log 1/\varepsilon$ bits immediately following the quotient. When the hash-selector value of an element $x \in S$ is incremented, its remainder "slides over" to the next (non-overlapping) $\log 1/\varepsilon$ bits of the hash $h(x)$, as shown in Figure 1. Thus, the fingerprint of x is $f(x) = q(x) \circ r_i(x)$, where $\circ$ denotes concatenation and i is the hash-selector value of x.

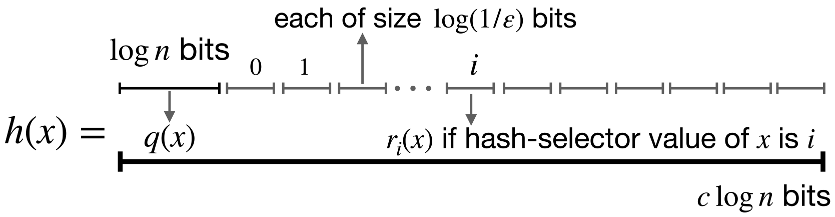
each of size log(1/ε) bits
log n bits
0
1
i
h(x) =
q(x)
r_i(x) if hash-selector value of x is i
c log n bits

Figure 1 The fingerprint of $x \in S$ is its quotient $q(x)$ followed by its remainder $r_i(x)$, where i is the hash-selector value of x.

On a false positive query $y \notin \mathcal{S}$, there must be some $x \in \mathcal{S}$ with hash-selector value i, such that $q(x) = q(y)$ and $r_i(x) = r_i(y)$. To resolve this false positive, we increment i. We update the hash-selector value and the stored remainder accordingly.

We describe below how to store hash-selector values using an average 0.875 bits per element. This means that the TAF with load factor α uses $(n/\alpha)(\log(1/\varepsilon) + 3)$ bits of space.

Difference between hash-selector and adaptivity bits. Using hash-selector bits, rather than adaptivity bits (as in the broom filter), has some immediate upsides and downsides.

If fingerprint prefixes $p(x)$ and $p(y)$ collide, they will still collide with probability 1/2 after each prefix has been lengthened by one bit. But adding a bit also reduces the probability that x will collide with any *future* queries by a factor of 1/2. Such false positives that are fixed (without being queried) are called **serendipitous false positives** [4].

On the other hand, incrementing the hash-selector value of an element $x \in \mathcal{S}$ after it collides with an element $y \notin \mathcal{S}$ reduces the probability that y will collide again with x by a factor of $\varepsilon \ll 1/2$. Thus, the TAF is more aggressive about fixing repeated false positives. However, the probability that x collides with future queries that are different from y remains unchanged. Thus, on average the TAF does not fix serendipitous false positives.

Our experiments (Section 6) show that the gain of serendipitous false positive fixes is short-lived; aggressively fixing false positives leads to better false-positive performance.

Storing hash selectors in blocks. The TAF does not have a constant number of bits per slot dedicated solely to storing its hash-selector value. Instead, we group the hash-selector values associated with each $\Theta(\log n)$ contiguous slots (64 slots in our implementation) together in a block. We allocate a constant amount of space for each such block. If we run out of space, we **rebuild** by setting all hash-selector values in the block to 0. (After a rebuild, we still fix the false positive that caused the rebuild. Therefore, there will often be one non-zero hash-selector value in the block after a rebuild.)

Encoding hash-selector bits. To store the hash selectors effectively, we need a code that satisfies the following requirements: the space of the code should be very close to optimal; the code should be able to use < 1 bits on average per character encoded; and the encode and decode operations should be fast enough to be usable in practice.

In Section 5, we give a new implementation of the arithmetic coding that is tailored to our use case, specifically encoding characters from the distribution given in Corollary 3. Our implementation uses only integers, and all divisions are implemented using bit shifts, leading to a fast and reliable implementation while still retaining good space bounds.

4 Telescoping Adaptive Filter: Analysis

In this section, we analyze the sustained false-positive rate, the hash-selector probabilities, and the space complexity of the telescoping adaptive filter.

We assume the TAF uses a uniform random hash function h such that the hash can be evaluated in $O(1)$ time. In our adaptivity analysis of the TAF (Theorem 1), we first assume that the filter has sufficient space to store all hash-selector values; that is, it does not rebuild. Then, in Theorem 4, we give a bound on the number of unique queries that the TAF can handle (based on its size) without the need to rebuild, thus maintaining adaptivity.

Adaptivity. We first prove that the telescoping adaptive filter is adaptive, i.e., it guarantees a sustained false positive rate of ε.

We say a query x has a **soft collision** with an element $y \in \mathcal{S}$ if their quotients are the same: $q(x) = q(y)$. We say a query x has a **hard collision** with an element $y \in \mathcal{S}$ if both their quotients and remainders are the same: $q(x) = q(y)$ and $r_i(x) = r_i(y)$, where i is the hash-selector value of y at the time x is queried (see Section 3).

▶ **Theorem 1.** *Consider a telescoping adaptive filter storing a set $\mathcal{S}$ of size n. For any adaptively generated sequence of t queries $Q = x_1, x_2, \ldots, x_t$ (possibly interleaved with insertions), where each $x_i \notin \mathcal{S}$, the TAF has a sustained false-positive rate of ε; that is,* $\Pr[x_i \textit{ is a false positive}] \le \varepsilon$ *for all* $1 \le i \le t$.

Proof. Consider the i-th query $x_i \in Q$. Query x_i is a false positive if there exists an element $y \in \mathcal{S}$ such that there is hard collision between them. Let $h_i(y) = q(y) \circ r_k(y)$ denote the fingerprint of y at time i, where y has the hash-selector value k at time i. Then, x_i and y have a hard collision if and only if $h_i(x_i) = h_i(y)$.

We show that for any y, regardless of answers to previous queries, x_i and y have a hard collision with probability ε/n; taking a union bound over all elements gives the theorem.

We proceed in cases. First, if x_i is a first-time query, that is, $x_i \notin \{x_1, \ldots, x_{i-1}\}$, then the probability that $h_i(x_i) = h_i(y)$ is the probability that both their quotient and remainder match, which occurs with probability $2^{-(\log n + \log 1/\varepsilon)} = \varepsilon/n$.

Next, suppose that x_i is a repeated query, that is, $x_i \in \{x_1, \ldots, x_{i-1}\}$. Let $j < i$ be the largest index where $x_i = x_j$ was previously queried. If x_j did not have a soft collision with y, that is, $q(x_j) \ne q(y)$, then x_i cannot have a hard collision with y. Now suppose that $q(x_j) = q(y)$. We have two subcases.

1. y's hash-selector value has not changed since x_j was queried. Note that, in this case, x_j must not have had a hard collision with y, as that would have caused y's hash-selector value, and thus its remainder, to be updated. Thus, $h_j(y) = h_i(y) \neq h_j(x_j) = h_i(x_i)$.
2. y's hash-selector value has been updated since x_j was queried. Such an update could have been caused by a further query to x_j having a hard collision with y, or some other query $x_k \in x_j, x_{j+1}, \ldots, x_i$ having a hard collision with y. In either case, the probability that the new remainder matches, i.e., $r_i(y) = r_i(x_i)$, is $2^{-\log 1/\varepsilon} = \varepsilon$.

Therefore, the probability that x_i has a hard collision with y is at most $\varepsilon \cdot \Pr[q(x_j) = q(y)] = \varepsilon/n$. Finally, by a union bound over n possibilities for $y \in \mathcal{S}$, we obtain that $\Pr[x_i \text{ is a false positive}] \leq \varepsilon$ for all $1 \leq i \leq t$, as desired. ◀

Hash-selector probabilities. The telescoping adaptive filter increments the hash-selector value of an element $y \in \mathcal{S}$ whenever a false positive query collides with y. Here we analyze the probability of an element having a given hash-selector value.

▶ **Lemma 2.** *Consider a sequence $Q = x_1, x_2, \ldots, x_t$ of queries (interleaved with inserts), where each $x_i \notin \mathcal{S}$ and Q consists of cn unique queries (with any number of repetitions), where $c < 1/\varepsilon - 1$. Then for any $y \in \mathcal{S}$, if $v(y)$ is the hash-selector value of y after all queries in Q are performed, then:*

$$\Pr[v(y) = k] \begin{cases} = (1 - \frac{\varepsilon}{n})^{cn} & \text{if } k = 0 \\ \leq \varepsilon^k (1 - \varepsilon) \sum_{i=1}^{k} \binom{cn}{i} \frac{1}{n^i} & \text{if } k > 0 \end{cases}$$

Proof. First, consider the case $k = 0$: the hash-selector value of y stays zero after all the queries are made if and only if none of the queries have a hard collision with y. Since there are cn unique queries, and the probability that each of them has a hard collision with y is ε/n, the probability that none of them collide with y is $(1 - \varepsilon/n)^{cn}$.

Now, consider the case $k \geq 1$. Given that the hash selector value of y is k, we know that there have been exactly k hard collisions between queries and y (where some of these collisions may have been caused by the same query). Suppose there are i unique queries among all queries that have a hard collision with y, where $1 \leq i \leq k$. Let k_j be the number of times a query j collides with y causing an increment in its hash-selector value, where $1 \leq j \leq i$. Thus, $\sum_{j=1}^{i} k_j = k$.

For a query x_j, the probability that x_j collides with y, the first time x_j is queried, is ε/n. Then, given that x_j has collided with y once, the probability of any subsequent collision with y is ε. (This is because the $\log 1/\varepsilon$ bits of the remainder of y are updated with each collision.) Thus, the probability that x_j collides with y at least k_j times is $\frac{\varepsilon}{n} \cdot \varepsilon^{k_j - 1}$.

The probability that a query x_j collides with y at least k_j times, is given by $\prod_{j=1}^{i} \frac{\varepsilon}{n} \cdot \varepsilon^{k_j - 1} = \frac{\varepsilon^k}{n^i}$. There are $\binom{cn}{i}$ ways of choosing i unique queries from cn, for $1 \leq i \leq k$, which gives us

$$\Pr[v(y) \geq k] = \varepsilon^k \sum_{i=1}^{k} \binom{cn}{i} \frac{1}{n^i} \tag{1}$$

Finally, using Inequality 1, we can upper bound the probability that a hash-selector value is exactly k.

$$\begin{aligned}
\Pr[v(y) = k] &= \Pr[v(y) \geq k] - \Pr[v(y) \geq k+1] \\
&= \varepsilon^k \left[\sum_{i=1}^{k} \binom{cn}{i} \frac{1}{n^i} - \varepsilon \sum_{i=1}^{k+1} \binom{cn}{i} \frac{1}{n^i} \right] \\
&= \varepsilon^k \cdot (1-\varepsilon) \left[\sum_{i=1}^{k} \binom{cn}{i} \frac{1}{n^i} - \frac{\varepsilon}{1-\varepsilon} \binom{cn}{k+1} \frac{1}{n^{k+1}} \right] \\
&\leq \varepsilon^k (1-\varepsilon) \sum_{i=1}^{k} \binom{cn}{i} \frac{1}{n^i}
\end{aligned}$$ ◀

We simplify the probabilities in Lemma 2 in Corollary 3. The probability bounds in Corollary 3 closely match the distribution of hash-selector frequencies we observe experimentally.

▶ **Corollary 3.** *Consider a sequence $Q = x_1, x_2, \ldots, x_t$ of queries (interleaved with inserts), where each $x_i \notin \mathcal{S}$ and Q consists of cn unique queries (with any number of repetitions), where $c < 1/\varepsilon - 1$. For any $y \in \mathcal{S}$, if $v(y)$ is the hash-selector value of y after all queries in Q are performed, then:*

$$\Pr[v(y) = 0] < \frac{1}{e^{c\varepsilon}}, \text{ and } \Pr[v(y) = k] < \varepsilon^k \sum_{i=1}^{k} \frac{c^i}{i!} \text{ for } k \geq 1.$$

Proof. To upper bound $\Pr[v(y) = 0]$, we use the inequality $(1 - 1/x)^x \leq 1/e$ for $x > 1$. To upper bound $\Pr[v(y) = k]$, we upper bound:

$$\binom{cn}{i} \frac{1}{n^i} \leq \frac{cn \cdot (cn-1) \cdots (cn-i)}{i!} \frac{1}{n^i} \leq \frac{c^i n^i}{i! n^i} = \frac{c^i}{i!}$$ ◀

Space analysis. Up until now, we have assumed that we always have enough room to store arbitrarily large hash selector values. Next, we give a tradeoff between the space usage of the data structure and the number of unique queries it can support.

We use the hash-selector probabilities derived above to analyze the space overhead of storing hash-selector values. Theorem 4 assumes an optimal arithmetic encoding: storing a hash-selector value k that occurs with probability p_k requires exactly $\log(1/p_k)$ bits. In our implementation we use an approximate version of the arithmetic coding for the sake of performance.

▶ **Theorem 4.** *For any $\varepsilon < 1/2$ and $b \geq 2$, given a sequence of $n/(b\sqrt{\varepsilon})$ unique queries (with no restriction on the number of repetitions of each), the telescoping adaptive filter maintains a sustained false-positive rate of ε using at most $\left(\frac{1}{e} + \frac{b}{(b-1)^2}\right)$ bits of space in expectation per element.*

Proof. Let $c = 1/(b\sqrt{\varepsilon})$; thus, there are cn unique queries. Consider an arbitrary element $y \in \mathcal{S}$. The expected space used to store the hash-selector value $v(y)$ of y is $\sum_{k=0}^{\infty} p_k \log 1/p_k$, where p_k is the probability that $v(y) = k$.

We separate out the case where $k = 0$, for which p_k is the largest, and upper bound the $p_0 \log 1/p_0$ term below, using the probability derived in Lemma 2.

$$\begin{aligned}
p_0 \log 1/p_0 &= (1 - \varepsilon/n)^{cn} \log \frac{1}{(1-\varepsilon/n)^{cn}} \leq \frac{1}{e^{c\varepsilon}} \cdot \log(1 + \frac{\varepsilon}{n})^{cn} \\
&= \frac{1}{e^{c\varepsilon}} \cdot cn \log(1 + \frac{\varepsilon}{n}) \leq \frac{1}{e^{c\varepsilon}} \cdot cn \frac{\varepsilon}{n} = \frac{c\varepsilon}{e^{c\varepsilon}} < \frac{1}{e} \qquad (2)
\end{aligned}$$

In step (2) above we use the fact that $x/e^x < 1/e$ for all $x > 0$.

We now upper bound the rest of the summation, that is, $\sum_{k=1}^{\infty} p_k \log 1/p_k$ for $k \geq 1$. When upper bounding this summation we will be using upper bounds on p_k – but this is a *lower* bound on $\log 1/p_k$. To deal with this, we observe that the function $x \log 1/x$ is monotonically increasing for $x < 1/e$. Therefore, if we show that the bounds in Corollary 3 never exceed $1/e$, we can substitute both terms in $p_k \log 1/p_k$ in our analysis. We start by showing this upper bound. In the following, we use $b \geq 2$ and $\varepsilon < 1/2$.

$$p_k < \varepsilon^k \sum_{i=1}^{k} \frac{c^i}{i!} < \varepsilon^k c^k \cdot k < \varepsilon^k \cdot \left(\frac{1}{b\sqrt{\varepsilon}}\right)^k \cdot k = \frac{k}{b^k} \cdot \varepsilon^{k/2} < k \cdot \frac{1}{2^{3k/2}} < \frac{1}{e}.$$

We now upper bound the sum $\sum_{k=1}^{\infty} p_k \log 1/p_k$ by replacing p_k with its upper bound $\varepsilon^k c^k \cdot k$ (this replacement is an upper bound because we showed $\varepsilon^k \sum_{i=1}^{k} \frac{c^i}{i!} < 1/e$ above).

$$\sum_{k=1}^{\infty} p_k \log 1/p_k \leq \sum_{k \geq 1} k\varepsilon^k c^k \log \frac{1}{\varepsilon^k c^k k} = \sum_{k=1}^{\infty} \frac{k}{b^k} \cdot \left(\varepsilon^{k/2} \cdot \log 1/\varepsilon^k\right) \tag{3}$$

$$< \sum_{k=1}^{\infty} \frac{k}{b^k} = \frac{b}{(b-1)^2}. \tag{4}$$

We simplify step (3) above using the fact that $\sqrt{x} \log 1/x < 1$ for all $x \leq 1$; step (4) is a known identity.

Thus, $\sum_{k=0}^{\infty} p_k \log 1/p_k < 1/e + b/(b-1)^2$, which is the expected number of bits used to store the hash-selector value of y. ◀

Theorem 4 implies that if the TAF is using a certain number of bits per element in expectation to store hash-selector values, then there is a precise bound on the number of unique queries it can handle in any query workload while being provably adaptive. For example, if $\varepsilon = 1/2^8$ and we set $b = 4$ in Theorem 4, then a telescoping adaptive filter that uses $4/9 + 1/e \approx 0.812$ bits per element in expectation can handle $4n$ unique queries without running out of space and having to rebuild. In Section 6, the TAF outperforms this bound, retaining good performance with 0.812 bits per element for $A/S \leq 20$.

5 Implementation

In this section, we describe the implementation of the TAF and our implementation of the broom filter [5], which we call the *extension adaptive filter* (exAF).

Recall that adaptive filters have a local state **L** and a remote representation **R**.

Rank-and-select quotient filter. The local state **L** of both the TAF and exAF is implemented as a rank-and-select quotient filter (RSQF) [29]. The RSQF stores metadata bits – one **occupied** bit and one **runend** bit for each slot. The occupied bit associated with slot i indicates whether any elements with the quotient i have been inserted into the filter. The runend bit associated with slot i tracks whether the remainder placed in slot i is the last remainder in a contiguous run of remainders with the same quotient. These metadata bits are sufficient to find the original slot of an element, but processing them bit-by-bit can be slow. The RSQF cleverly uses *rank* and *select* operations to quickly jump to the original slot [6]. These operations are efficiently implemented using x86 instructions on 64-bit words.

To improve cache efficiency, the RQSF stores remainders (along with their 2 metadata bits) in 64-element *blocks*. In particular, each block stores 64 contiguous remainders and two 64-bit metadata arrays. To search through the blocks efficiently, an **offset** (stored using at most 8 bits) is stored for each block. The offset of a location i is the distance between i and i's associated runend. Each block stores the offset of its first slot. In total, the RSQF stores 2.125 metadata bits per element in the filter.

Arithmetic coding on integers. Arithmetic coding can give theoretically optimal compression, but the standard implementation that recursively divides the unit interval relies on floating point operations. These floating point operations are slow in practice, and involve precision issues that can lead to incorrect answers or inefficient representations. In our implementation, we avoid these issues by applying arithmetic coding to a range of integers, $\{0, \ldots, 2^k - 1\}$ for the desired code length k, instead of the unit interval. We set $k = 56$, encoding all hash-selector values for a block in a 56-bit word. When multiplying or dividing integral intervals by probabilities in $[0, 1]$, we approximate floating point operations using integer shifts and multiplications.

Remote representation. We implement $\mathbf{R}$ for both filters as an array storing elements in the set $\mathcal{S}$, along with their associated hashes. We keep $\mathbf{R}$ in sync with $\mathbf{L}$: if the remainder $r(x)$ is stored in slot s in $\mathbf{L}$, then x is stored in slot s in $\mathbf{R}$. This leads to easy lookups: to lookup an element x in $\mathbf{R}$, we simply check the slot $\mathbf{R}[s]$ where $r(x) = \mathbf{L}[s]$. Insertions that cause remainders to shift in $\mathbf{L}$ are expensive, however, as we need to shift elements in $\mathbf{R}$ as well.

TAF implementation. The local state of TAF is an RSQF where each block of 64 contiguous elements stores the remainders of all elements, all metadata bits (each type stored in a 64-bit word), an 8-bit offset, and a 56-bit arithmetic code storing hash-selector values.

TAF's inserts are similar to the RSQF, which may require shifting remainders. The TAF updates the hash-selector values of all blocks that are touched by the insertion.

Our implementation uses MurmurHash [3] which has a 128-bit output. We partition the output of MurmurHash into the quotient, followed by chunks of size $\log(1/\varepsilon)$, where each chunk corresponds to one remainder. Each time we increment the hash-selector value, we just slide over $\log(1/\varepsilon)$ bits to obtain the new remainder.

On a query x, the TAF goes through each slot s corresponding to quotient $q(x)$ and compares the remainder stored in s to $r_i(y)$, where i is the hash-selector value of s, retrieved by decoding the blocks associated with each s. If they match, the filter returns "present" and checks $\mathbf{R}$ to determine if $x \in \mathcal{S}$. If $x \notin \mathcal{S}$, the filter increments the hash-selector i of x and updates the arithmetic code of the block containing x.

If the 56-bit encoding fails, we **rebuild**: we set all hash-selector bits in the block to 0, and then attempt to fix the false positive again.

exAF implementation. Our implementation of the broom filter, which we call the exAF, maintains its local state as a blocked RSQF, similar to the TAF . The main difference between the two filters is how they adapt. The exAF implements the broom filter's adapt policy of lengthening fingerprints. To do this efficiently, we follow a strategy similar to the TAF . We divide the data structure into blocks of 64 elements, storing all extensions for a single block into an arithmetic code that uses at most 56 bits.

The exAF's insertion algorithm resembles the that of the RSQF and broom filter. However, while the broom filter adapts on inserts to ensure that all stored fingerprints are unique, the exAF does not adapt on inserts, and may have duplicate fingerprints.

During a query operation, the exAF first performs an RQSF query: it finds if there is a stored element whose quotient and remainder bits match, without accessing any extension bit. Only if these match does it decode the block's arithmetic code, allowing it to check extension bits. This makes queries in the exAF faster compared to TAF, which must perform decodes on all queries. If the full fingerprint of a query y collides with an element $x \in \mathcal{S}$, the filter returns "present" and checks $\mathbf{R}$ to determine if $x \in \mathcal{S}$. If $x \notin \mathcal{S}$, the exAF adapts by adding extension bits to $f(x)$ by decoding the block's arithmetic code, updating x's extension bits, and re-encoding.

As in the TAF, if the 56-bit encoding fails, the exAF rebuilds by setting all adaptivity bits in the block to 0, and then attempts to fix the false positive again.

6 Evaluation

In this section, we empirically evaluate the telescoping adaptive filter and the exAF.

We compare the false-positive performance of these filters to the Cuckooing ACF, the Cyclic ACF (with $s = 1, 2, 3$ hash-selector bits), and the Swapping ACF. The Cyclic ACF and the Cuckooing ACF use 4 random hashes to choose the location of each element, and have bins of size 1. The Swapping ACF uses 2 location hashes and bins of size 4.

We compare the throughput of the TAF and exAF against the vacuum filter [32], our implementation of the RSQF, and a space-inefficient version of the TAF that does not perform arithmetic coding operations.

Experimental setup. We evaluate the filters in terms of the following parameter settings.

- *Load factor.* For the false-positive tests, we use a load factor of .95. We evaluate the throughput on a range of load factors.
- *Fingerprint size:* We set the fingerprint size of each filter so that they all use the same amount of space. We use 8-bit remainders for the TAF. Because the TAF has three extra bits per element for metadata and adaptivity, this corresponds to fingerprints of size 11 for the Swapping and Cuckooing ACF, and size $11 - s$ for a Cyclic ACF with s hash-selector bits.
- *A/S ratio.* The parameter A/S (shorthand for $|A|/|\mathcal{S}|$) is the ratio of the number of unique queries in the query set A and the size of the filter's membership set $\mathcal{S}$. Depending on the structure of the queries, a higher A/S value may indicate a more difficult workload, as "fixed" false positives are separated by a large number of interspersed queries.

All experiments were run on a workstation with Dual Intel Xeon Gold 6240 18-core 2.6 Ghz processors with 128G memory (DDR4 2666MHz ECC). All experiments were single-threaded.

6.1 False Positive Rate

Firehose benchmark. We measure the false positive rate on data generated by the Firehose benchmark suite [1, 2] which simulates a real-world cybersecurity workload. Firehose has two generators: **power law** and **active set**; we use data from both.

The active set generator generates 64-bit unsigned integers from a continuously evolving "active set" of keys. The probability with which an individual key is sampled varies in time according to a bell-shaped curve to create a "trending effect" as observed in cyberstreams [2]. We generated 10 million queries using the active set generator. We set the value `POW_EXP` in the active set generator to 0.5 to encourage query repetitions. (Each query is repeated approximately 57 times on average in our final dataset.)

We then generated 50 million queries using the power-law generator, which generates queries using a power-law distribution. This dataset had each query repeated many times; each query was repeated 584 times on average.

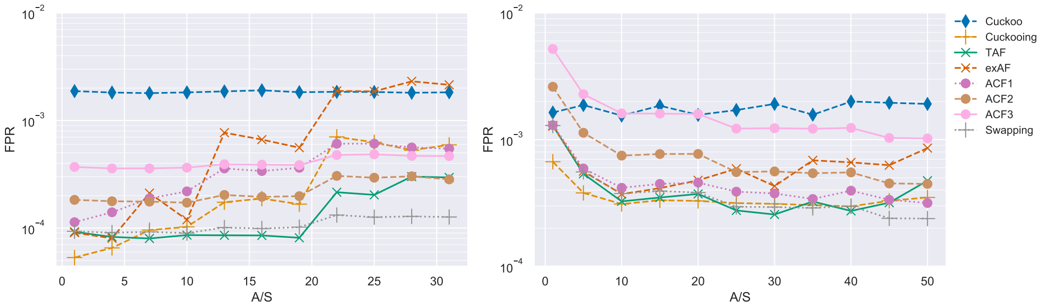

Figure 2 False positive rates on the firehose benchmarks. The plot on the left uses the active set generator; the plot on the right uses the power-law generator.

In our tests we vary the size of the stored set $\mathcal{S}$ (each uses the same input, so $|A|$ is constant). The results are shown in Figure 2; all data points are the average of 10 experiments. ACF1, ACF2, and ACF3 represent the Cyclic ACF with $s = 1, 2, 3$ respectively.

For the active set generated data, the TAF is the best data structure for moderate A/S. Above $A/S \approx 20$, rebuilds become frequent enough that TAF performance degrades somewhat, after which its performance is similar to that of the Cyclic ACF with $s = 2$ (second to the Swapping ACF). This closely matches the analysis in Section 4.

For the power law data, the TAF is competitive for most A/S values, although again it is best for moderate values.

Notably, in both cases (and particularly for the active set data), the exAF performs substantially worse than the TAF. This shows that given the space amount of extra bits per element on average, the TAF uses them more effectively towards adaptivity than the exAF.

Network Traces. We give experiments on three network trace datasets from the CAIDA 2014 dataset, replicating the experiments of Mitzenmacher et al. [25]. We use three network traces from the CAIDA 2014 dataset, specifically `equinix-chicago.dirA.20140619` ("Chicago A", Figure 3) `equinixchicago.dirB.20140619-432600` ("Chicago B", Figure 3), and `equinix-sanjose.dirA.20140320-130400` ("San Jose", Figure 4).

On network trace datasets, most filters are equally effective at fixing false positives, and their performance is determined mostly by their **baseline false positive rate**, that is, the probability with which a first-time query is a false positive. If s bits are used for adaptivity, that increases the baseline FP rate by 2^s, compared to when those bits are used towards remainders. This gives the Cuckooing ACF an advantage as it uses 0 bits for adapting.

The TAF and exAF perform similarly to the Swapping ACF and ACF1 (Cyclic ACF with $s = 1$) on these datasets.

Adversarial tests. The main advantage of the TAF and exAF is that both are adaptive in theory – even against an adversary. Adversarial inputs are motivated by security concerns, such as denial-of-service attacks, but they may also arise in some situations in practice. For example, it may be that the input stream is performance-dependent, and previous false positives are more likely to be queried again.

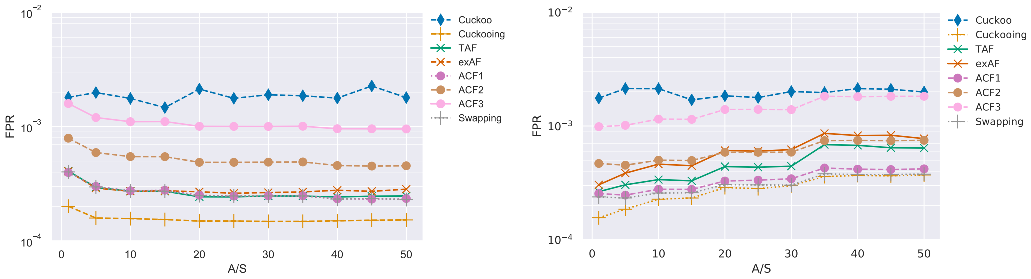

Figure 3 False positive performance of the filters on network trace data. The Chicago A dataset is used on the left, and the Chicago B dataset is on the right.

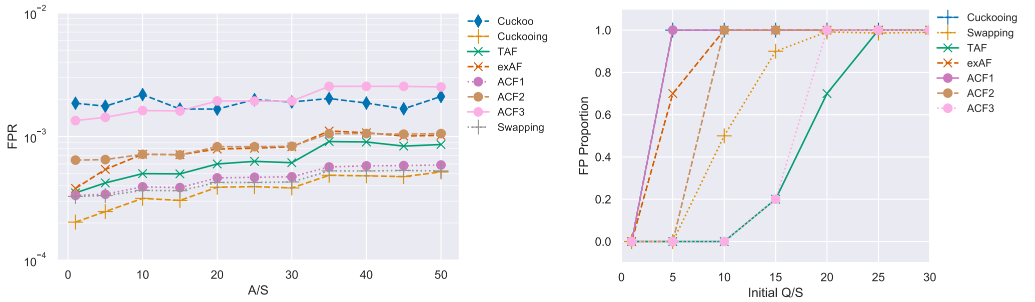

Figure 4 On the left is the network trace San Jose dataset. On the right is adversarial data, where we vary the size of the initial query set, and plot the proportion of elements in the final set that are false positives.

We test our filter against an "adversarial" stream that probabilistically queries previous false positives. This input is significantly simpler than the lower bounds given in [21] and [5], but shares some of the basic structure.

Our adversarial stream starts with a set of random queries $|Q|$. The queries are performed in a sequence of rounds; each divided into 10 subrounds. In a subround, each element of Q is queried. After a round, any element that was never a false positive in that round is removed from Q. The filter then continues to the next round. The test stops when $|Q|/|\mathcal{S}| = .01$, or a bounded number of rounds is reached.

The x-axis of our plot is $|Q|/|\mathcal{S}|$, and the y-axis is the false positive rate during the final round (after the adversary has whittled Q to only contain likely false positives). We again see that the TAF does very well up until around $|Q|/|\mathcal{S}| \approx 20$. After this point, the adversary is successfully able to force false positives. This agrees closely with the analysis in Section 4.

The Cyclic ACF with $s = 3$ (ACF3) does well on adversarial data even though it is known to not be adaptive. This may be in part because the constants in the lower bound proof [21] are very large (the lower bound uses $1/\varepsilon^8 \approx 2^{64}$ queries). However, this adaptivity comes at the cost of a worse baseline FP rate, as this filter struggles on network trace data.

6.2 Throughput

In this section, we compare the throughput of our filters to other similar filters.

For the throughput tests, we introduce several new filters as a point of comparison. The vacuum filter [32] is a cuckoo filter variant designed to be space- and cache-efficient. We compare to the "from scratch" version of their filter [34]. We also compare to our implementation of the RSQF [29]. The RSQF does not adapt, or perform remote accesses.

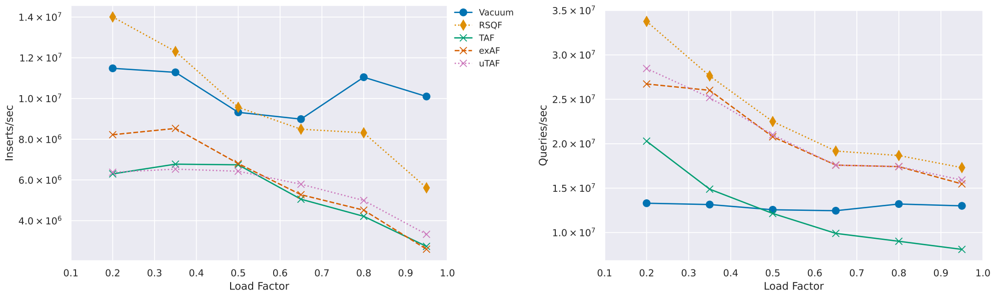

Figure 5 The throughput for inserts (left) and queries (right) on the active set Firehose data.

Finally, to isolate the cost of the arithmetic coding itself, we compare to our implementation of an **uncompressed telescoping adaptive filter (uTAF)**. The uTAF works exactly as the TAF, except it stores its hash-selector values explicitly, without using an arithmetic coding. This means that the uTAF is very space-inefficient.

For the throughput tests, we evaluated the performance on the active set Firehose data used in Figure 2. Our filters used 2^{24} slots. We varied the load factor to compare performance. All data points shown are the average of 10 runs.

The throughput tests show that the TAF achieves similar performance in inserts to the other filters, though it lags behind in queries at high throughput. The exAF performs significantly better for queries, likely due to skipping decodes as discussed in Section 5.

The uTAF is noticeably faster than the TAF, but is similar in performance to exAF. This highlights the trade-offs between the two ways to achieve adaptivity: the exAF scheme of lengthening remainders has better throughput but worse adaptivity per bit; while the TAF scheme of updating remainders has better adaptivity per bit but worse throughput. Overall, while the query-time decodes of TAF do come at a throughput cost, they stop short of dominating performance.

7 Conclusion

We provide a new provably-adaptive filter, the telescoping adaptive filter, that was engineered with space- and cache-efficiency and throughput in mind. The TAF is unique among adaptive filters in that it only uses a fractional number of extra bits for adaptivity (0.875 bits per element). To benchmark the TAF, we also provide a practical implementation of the broom filter. To effectively compress the adaptivity metadata for both filters, we implement arithmetic coding that is optimized for the probability distributions arising in each filter.

We empirically evaluate the TAF and exAF against other state-of-the-art filters that adapt, on a variety of datasets. Our experiments show that TAF outperforms the exAF significantly on false-positive performance, and frequently matches or outperforms other heuristically adaptive filters. Our throughput tests show that our adaptive filters achieve a comparable throughput to their non-adaptive counterparts.

We believe that our technique to achieve adaptivity through variable-length fingerprints is universal and can be used alongside other filters that stores fingerprints of elements (e.g., a cuckoo or vacuum filter). Thus, there is potential for further improvements by applying our ideas to other filters, taking advantage of many years of filter research.

References

1 Karl Anderson and Steve Plimpton. Firehose streaming benchmarks. Technical report, Sandia National Laboratory, 2015.

2 Karl Anderson and Stevel Plimpton. FireHose streaming benchmarks. `www.firehose.sandia.gov`. Accessed: 2018-12-11.

3 Austin Appleby. Murmurhash. `https://github.com/aappleby/smhasher`, 2016. Accessed: 2020-08-01.

4 Michael A Bender, Rathish Das, Martín Farach-Colton, Tianchi Mo, David Tench, and Yung Ping Wang. Mitigating false positives in filters: to adapt or to cache? In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 16–24. SIAM, 2021.

5 Michael A Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *Symposium on Foundations of Computer Science (FOCS)*, pages 182–193. IEEE, 2018.

6 Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: how to cache your hash on flash. *Proc. VLDB Endowment*, 5(11):1627–1637, 2012.

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

8 Alex D Breslow and Nuwan S Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endowment*, 11(9):1041–1055, 2018.

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

10 J Bruck, Jie Gao, and Anxiao Jiang. Weighted bloom filter. In *Symposium on Information Theory*. IEEE, 2006.

11 Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Symposium on Theory of Computing (STOC)*, pages 59–65. ACM, 1978.

12 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *Transactions on Computer Systems*, 26(2):4, 2008.

13 Saar Cohen and Yossi Matias. Spectral bloom filters. In *International Conference on Management of Data (SIGMOD)*, pages 241–252. ACM, 2003.

14 Kyle Deeds, Brian Hentschel, and Stratos Idreos. Stacked filters: learning to filter by structure. *Proc. VLDB Endowment*, 14(4):600–612, 2020.

15 Fan Deng and Davood Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *International Conference on Management of Data (SIGMOD)*, pages 25–36. ACM, 2006.

16 Peter C Dillinger and Stefan Walzer. Ribbon filter: practically smaller than bloom and xor. *arXiv preprint arXiv:2103.02515*, 2021.

17 David Eppstein, Michael T Goodrich, Michael Mitzenmacher, and Manuel R Torres. 2-3 cuckoo filters for faster triangle listing and set intersection. In *Principles of Database Systems (PODS)*, pages 247–260. ACM, 2017.

18 Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Conference on emerging Networking Experiments and Technologies (CoNEXT)*, pages 75–88. ACM, 2014.

19 Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)*, 25:1–16, 2020.

20 Paul G. Howard and Jeffrey Scott Vitter. *Practical Implementations of Arithmetic Coding*, pages 85–112. Springer US, Boston, MA, 1992. `doi:10.1007/978-1-4615-3596-6_4`.

21 Tsvi Kopelowitz, Samuel McCauley, and Eli Porat. Support optimality and adaptive cuckoo filters. In *Proc. 17th Algorithms and Data Structures Symposium (WADS)*, 2021. To appear.

22 Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proc. VLDB Endowment*, 12(5):502–515, 2019.

23 Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: LSM-tree database storage engine serving Facebook's social graph. *Proc. VLDB Endowment*, 13(12):3217–3230, 2020.

24 Michael Mitzenmacher. A model for learned bloom filters, and optimizing by sandwiching. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 462–471, 2018.

25 Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018.

26 Moni Naor and Eylon Yogev. Bloom filters in adversarial environments. In *Annual Cryptology Conference*, pages 565–584. Springer, 2015.

27 Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

28 Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. An optimal bloom filter replacement. In *Symposium on Discrete Algorithms (SODA)*, pages 823–829. ACM-SIAM, 2005.

29 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *International Conference on Management of Data (SIGMOD)*, pages 775–787. ACM, 2017.

30 Jack Rae, Sergey Bartunov, and Timothy Lillicrap. Meta-learning neural bloom filters. In *International Conference on Machine Learning (ICML)*, pages 5271–5280. PMLR, 2019.

31 Sasu Tarkoma, Christian Esteve Rothenberg, Eemil Lagerspetz, et al. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials*, 14(1):131–155, 2012.

32 Minmei Wang and Mingxun Zhou. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proc. VLDB Endowment*, 2019.

33 Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

34 Mingxun Zhou. Vacuum filter. `https://github.com/wuwuz/Vacuum-Filter`, 2020. Accessed: 2020-12-01.