Constructions and Applications for Accurate Counting of the Bloom Filter False Positive Free Zone

Ori Rottenstreich
Technion
or@cs.technion.ac.il

Pedro Reviriego
Universidad Carlos III de Madrid
revirieg@it.uc3m.es

Ely Porat
Bar Ilan University
poratelY@cs.biu.ac.il

S. Muthukrishnan
Rutgers University
muthu@cs.rutgers.edu

ABSTRACT

Bloom filters are used in many networking applications to answer set membership queries at low cost but suffer from false positives. We study Bloom filter constructions that when representing a set of size up to d taken from a finite universe of size n, completely avoid false positives. We suggest memory-efficient Bloom filters constructions with a false positive free zone to allow representations of larger sets through linear memory dependency in the set size. Our first construction relies on Orthogonal Latin Square (OLS) codes and the second relies on the representation of elements through values of polynomials defined modulo primes. Beyond Bloom filters supporting set membership, we also consider sketches allowing a more general functionality such as flow size estimation. In particular, we show the applicability of the false positive free zone for accurate size estimation in the famous Count-Min sketch. We compare the new constructions to existing approaches through analytical and experimental evaluations for showing their superiority.

CCS CONCEPTS

• **Networks → Network algorithms**;

KEYWORDS

Measurement, Flow size estimation, Bloom filter, Count-Min Sketch

ACM Reference Format:
Ori Rottenstreich, Pedro Reviriego, Ely Porat, and S. Muthukrishnan. 2020. Constructions and Applications for Accurate Counting of the Bloom Filter False Positive Free Zone. In *Symposium on SDN Research (SOSR '20), March 3, 2020, San Jose, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373360.3380845

1 INTRODUCTION

1.1 Background

The Bloom filter is a popular data structure widely used in many networking algorithms [3, 6, 27], in fields as diverse as packet classification, monitoring, routing, filtering, caching and set synchronization [7, 10, 11, 13, 28, 39]. The Bloom filter is used for set representation, supporting element insertion and answering membership queries. There are two kinds of errors for membership queries in the representation of a set S: a false positive (when an element $x \notin S$ is reported as a member of S) and a false negative (when an element $x \in S$ is reported as a non member of S).

The Bloom filter encounters false positives and has no false negatives. The probability for an error (ratio of non-member elements reported as members) decreases when more memory is allocated for the data structure and increases when a larger set S is represented. The Bloom filter stores an array of bits, where a set of hash functions is used to map elements to locations in the bit array. With initial values of zero bits, the elements of S are first inserted to the filter, setting all bits pointed by the hash functions. Upon a query, the bits mapped by the queried element are examined and a positive answer is given in case all these bits are set.

For some applications potential occurrence of false positives can result in severe performance degradation. For instance, when a Bloom filter represents flow attributes in SDN switches such as the network functions a flow has to go through [29], false positives imply additional overhead on middleboxes and extend flow latency. Additional applications with a similar behavior are multicast addressing [37] and detection of botnet attacks [1].

A recent approach [21] presented the EGH Bloom filter which ensures that there are no false positives when the number of set elements taken from a finite universe is at most a given bound d. The construction is applicable when the universe U is by nature small and makes use of simple carefully-designed hash functions while relying on prime numbers. The required filter length to completely avoid false positives is affected by the universe size and the maximal set size that still avoids false positives. Namely, *a limited size filter implies a tradeoff and can describe without false positives larger sets of elements from a small universe or alternatively smaller sets selected from a large universe.*

The construction described in [21] allocates an amount of memory affected by the universe size and the maximal set size for which false positives should be avoided. The construction was shown to require a memory of $O(d^2 \log n)$ bits such that false positives are avoided whenever $|S| \leq d$. In particular, for a given universe size, when d increases and larger sets have to be represented without false positives, the required memory increases quadratically, restricting in practice the size of sets that can be represented.

SOSR '20, March 3, 2020, San Jose, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380845

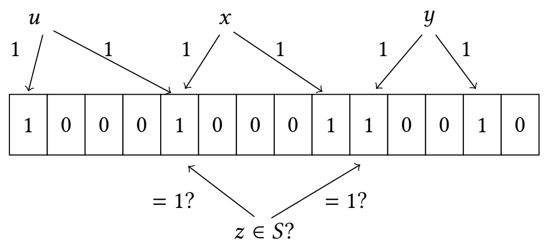


Figure 1: The Bloom filter (representing a set $S = \{u, x, y\}$): An element is mapped by hash functions to bits in the array. Bits are set upon element insertion and are examined upon a membership query.

In some key networking applications, like the encoding of flow attributes in SDN switches [29] or the multicast addressing [37], the universe is by nature small and the size of the set is not negligible. This leads to scenarios where the d^2 factor has a large impact on the required memory. In those applications, since the filter is added to each packet, minimizing its size becomes critical. Therefore, finding alternative constructions of filters with a FPFZ that scale better with the maximal set size d would be of high interest.

In our target networking applications, we want to support fast queries and set updates and also allow the set to become eventually larger than d. The first requirement is needed as in networking, packets have to be processed at high speed and thus checking operations must be fast. The second is because, even if the system is dimension for sets of d elements, in some cases the set can be larger. This can occur for example, in multicast applications where the filter encodes the links to send a packet, when the network is large or the packet has to reach many nodes.

A naïve set representation can rely on a simple list of the elements. Since $\log_2 n$ bits are required to represent an element uniquely, such simple representation requires $d \log_2 n$ memory to represent a set of size d. In fact, unique representations can be derived even with $\log_2 \binom{n}{d}$ bits as the element order is not important. However, such representations do not support answering queries or set updates with the reading of a fixed and small amount of memory positions. A list would require going over it all to eliminate membership of a non-member. Even a sorted list which is harder to maintain, would require a number of accesses which grows logarithmically with the set size. We restrict the solutions to require a number of accesses which is constant and in particular does not grow linearly with the set size. A model formalizing such restrictions was described in [34] where a (d, n, m, k)-scheme was defined as a storage scheme that stores any d elements of an n-sized universe using m bits such that membership queries can be answered using k bit accesses.

Beyond answering set membership queries, network management often requires more advanced measurements capabilities. Tasks include counting the amount of traffic per flow, identifying those flows of high volume, counting the number of distinct flow or identifying undesired network behavior [2, 24, 26, 38, 41]. Traditionally, tasks are fulfilled through lossy data structures called sketches, fulfilling these tasks in only statistical accuracy.

Table 1: Summary of main notations

Symbol	Meaning
S	represented set
U	universe from which elements are selected
n	universe size $\|U\|$
d	maximal set size in the false positive free zone
m	required filter length
k	number of hash functions
M	binary matrix representing the hash functions

Table 2: Complexity of the suggested OLS and POL filters allowing memory complexity which is linear in the maximal set size in comparison with the existing EGH filter. t is a controllable parameter.

filter	memory complexity	# probes
EGH [14]	$O(d^2 \cdot \log n)$	$O(d \cdot \log n)$
OLS	$(d+1)\sqrt{n}$	$d+1$
POL	$((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$	$(t-1) \cdot d + 1$

1.2 Contributions

As the first contribution of this paper, we suggest two constructions of Bloom filters with a false positive free zone that allow filter memory complexity which is *linear* in the size of the represented set. The existing EGH filter construction required quadratic memory dependency in the set size. The new constructions achieve an advantage for larger sets. Following the parameters implied by a given application, a specific construction can be more efficient than others based on the values of the universe set n and the maximal allowed set size d. The notations used in the paper are summarized in Table 1 and parameters of the new suggested constructions named OLS and POL filters are summarized in Table 2.

The second main contribution of this work is showing how constructions for Bloom filter with a false positive free zone can be used to guarantee accuracy in additional sketches with functionality beyond membership queries such as the Count-Min sketch [8], performing flow size estimation. We believe this work can serve as the basis towards the design of new type of measurement data structures for a wider range of measurement tasks having guarantees (which are not statistical) on the accuracy.

2 PRELIMINARIES

This section provides an overview of the Bloom filter data structure, a formal definition of its false positive free zone and describes the existing construction of filters with a false positive free zone. It also gives a brief overview of linear block codes. This provides the reader the background needed to follow the proposed constructions presented in Section 3 and Section 4.

2.1 Bloom filter and its false positive free zone

A Bloom Filter is a well-known simple data structure used to represent a set elements $S = \{x_1, \ldots, x_l\}$ from a universe U using an array of m bits [3].

As illustrated in Fig. 1, the Bloom filter uses k uniformly-distributed hash functions over the range $1, \ldots, m$ of its m- bit filter. For each element in the set S, k hash entries are calculated using the hash functions and the corresponding bits are set to one. For instance, in the figure, two hash functions are used and the bits pointed by u, x and y are set to one. To test whether an element $z \in U$ is in S, we check whether all of its k corresponding bit locations mapped by z are set to one. If this is not the case, we know that $z \notin S$. If all of them are set, the Bloom filter states that $z \in S$, although this might be a false positive error. For each $z \notin S$, the probability of a false positive error, can be approximated by $(1 - p_0)^k$, where $p_0 = (1 - 1/m)^{nk}$ is the probability that a specific bit is still zero after the insertion of n elements.

While the above probability of a Bloom filter for a false positive depends on its number of elements, it still remains positive while holding a small number of elements and even a single element. For some applications, it is required to guarantee the avoidance of false positives implying that any positive membership indication is correct. For a finite universe, the false positive free zone guarantees the avoidance of false positives as long as the number of set elements remains small. We follow the definition of the false positive free zone of a Bloom filter from [21].

Definition 2.1. The **false positive free zone** of a filter allows a universe $U = \{1, \ldots, n\}$ of size n and a maximal set size d, if for any filter $S \subseteq U$ satisfying $|S| \leq d$, the query operator of an element $y \in U$ always returns the true answer and in particular avoids false positives.

Note that we do not restrict the set size from temporarily being larger than d but allow the occurrence of false positives when this happens.

The construction of a filter of size m is described through a binary matrix M of size $m \times n$. The columns refers to the n universe elements, each described through a binary vector of length m showing the bits it maps to the k hash functions. The requirements on M for enabling the FPFZ are as follows.

Let M be a $m \times n$ binary matrix such that for $i \in [1, n]$ the i^{th} column describes the bits mapped by an element $j \in U$ where $|U| = n$ through the set of hash functions. Let S be a set of elements of size $|S| \leq d$ and let $j \notin S$ be another element. The matrix M implies a FPFZ of size d if for each such S and $j \notin S$, there is an index $i \in [1, m]$ such that $M_{i,j} = 1$ and $\forall j' \in S, M_{i,j'} = 0$. Namely, the membership of j in the filter describing S can be eliminated based on the index i.

2.2 The EGH Filter

The EGH filter was described as the first construction of a Bloom filter with a false positive free zone (Definition 2.1) in [21]. For a universe $U = \{1, \ldots, n\}$ it defines the mapping of elements into a filter of m bits through a matrix of size $m \times n$. The filter is organized in blocks such that each block length is a prime integer, starting from the smallest primes $p_1 = 2, p_2 = 3, p_3 = 5$, etc. To provide the false positive free zone property for all represented sets of size $|S| \leq d$, it is required to use k primes such that $\prod_{i=1}^{k} p_i \geq n^d$. The required filter length is then given by the sum $m = \sum_{j=1}^{k} p_j$. For an element $x \in U$ the mapping of x into the filter is simply given by computing $x \bmod p_i$ for all k primes and storing the values.

The filter for a set S is given by the cumulative OR of the columns that correspond to the $|S|$ elements. A membership element query returns positive if all k corresponding bits are set in the filter. The false positive free zone property guarantees that a negative answer is returned to the query of all elements in $U \setminus S$, namely each such element is mapped to at least one bit not set by any of S elements. Moreover, as in the traditional Bloom filter, false negatives do not occur. The asymptotic memory required filter is $O(d^2 \log n)$.

2.3 Linear Block Codes

In communications or storage, errors can occur and alter the data transmitted or stored. If those errors are not corrected they could lead to performance degradation or even to system failure. Therefore, many communications and storage systems use error detection and correction codes [25]. One of the most common types of codes used are systematic binary linear block codes that protect a data block of v bits using a codeword of w bits formed by the original v data bits and $w - v$ additional bits used to store parity checks on a subset of the v bits. The code can be defined by its generator matrix G and parity check matrix H. Both matrices are binary. The first one is used to obtain the encoded data by multiplying the generator matrix with the data to encode such that additions are done modulo 2. Conversely, the parity check matrix H detects errors. This is done again by multiplication, with the w bits transmitted or stored. The resulting vector is commonly known as the syndrome and when it is not the zero vector, an error is detected.

As an example, let us consider a simple Hamming code that can correct single bit errors [16] and a block of size $v = 4$ for which $w - v = 3$ additional parity check bits are needed. The generator matrix for the code (of size $w \times v$) is shown in Fig. 2(a). The corresponding parity check matrix H (of size $(w - v) \times w$) can be found in Fig. 2(b). Each row corresponds to a parity check equation and each column to a data or parity bit. If data bits are b_1, b_2, b_3, b_4 from left to right and parity bits are p_1, p_2, p_3 also from left to right, we would have: $p_1 = b_1 \texttt{ xor } b_3 \texttt{ xor } b_4$ where `xor` is used to implement modulo-2 addition. The leftmost data bit b_1 for instance participates on parity checks p_1 and p_2.

We could think of the parity check matrix H (excluding the columns of the parity check bits) as our M matrix of size $m \times n$. In that case, the size of the filter m would correspond to the number of parity check bits $w - v$ and the number of data bits v to the false positive free universe size n. To provide a false positive free zone, we need that the union or Boolean sum (or bitwise OR) of any up to d columns does not contain any other column. Unfortunately, this is not generally the case for an arbitrary matrix. For example, going back to the Hamming code, the column that corresponds to b_4 (fourth column in Fig. 2(b)) contains the other three data columns. The same applies to various error correction codes, for example to Bose-Chaudhuri-Hocquenghem (BCH) codes that have some columns that contain other columns. There is however an exception for which the desired properties hold: Orthogonal Latin Square (OLS) codes. In the next section, those codes are described and we explain how they can be used to build filters with a free positive zone.

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{vmatrix}$$

(a) Generator matrix G.

$$\begin{vmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{vmatrix}$$

(b) Parity check matrix H.

Figure 2: Linear codes: Generator matrix and parity check matrix for a linear code, encoding $v = 4$ data bits through $w = 7$ bits.

3 ORTHOGONAL LATIN SQUARE BLOOM FILTERS WITH A FALSE POSITIVE FREE ZONE

In this section we present our first construction: the OLS filter. A first subsection discusses the features of the Orthogonal Latin Square (OLS) codes and how they can be used to construct the filter. Then a practical filter example is given followed by a formal proof regarding the filter false positive free zone. The rest of the section presents an analytical comparison with the existing EGH filter.

3.1 Description of the proposed filters

As we mentioned, the design of the matrices required for the false positive free Bloom filters is to some extent similar to the design of binary linear block codes for error detection and correction. Those codes are defined by their generator or by their parity check matrices. In the case of the error correction codes, the operations are XOR (modulo-2 addition) instead of binary OR. In particular, a class of codes known as Orthogonal Latin Square (OLS) Codes [19], has been widely studied to protect memories from errors as they have a modular construction and can be decoded in parallel with simple circuitry [35].

An Orthogonal Latin Square (OLS) Code is constructed using Latin Squares. A Latin square of order s is a matrix of size s by s where elements $0, 1, \ldots, s-1$ are placed in the matrix so that each of them appears only once in each row and each column [20]. We refer to s as the order of the Latin square. An example of two Latin squares is given in Fig. 3(a).

Two Latin squares are called orthogonal if when superimposing the squares, every ordered pair of elements appears exactly once. Taking the two Latin squares of Fig. 3(a), we obtain the matrix shown in Fig. 3(b) when they are superimposed. For the superimposing, in each pair of values the left value is taken from the first Latin square and the right value from the right Latin square. It can be seen that each ordered pair of values appears only once. A set of Latin squares is mutually orthogonal if all the pairs of Latin squares in the set are orthogonal. This means that when all the Latin squares in the set are superimposed, a pair of ordered sequences of elements has at most one position with the same value as if there are two, then the Latin squares that correspond to those positions would not be orthogonal.

The construction of orthogonal Latin square codes [19] makes use of two additional special matrices (which are not Latin squares). These two matrices, shown in Fig. 3(c), have fixed values along all

$$\begin{array}{|ccccc|} \hline 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 0 \\ 2 & 3 & 4 & 0 & 1 \\ 3 & 4 & 0 & 1 & 2 \\ 4 & 0 & 1 & 2 & 3 \\ \hline \end{array} \qquad \begin{array}{|ccccc|} \hline 0 & 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 0 & 1 \\ 4 & 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 & 0 \\ 3 & 4 & 0 & 1 & 2 \\ \hline \end{array}$$

(a) Two Latin squares of order 5.

$$\begin{Vmatrix} 0,0 & 1,1 & 2,2 & 3,3 & 4,4 \\ 1,2 & 2,3 & 3,4 & 4,0 & 0,1 \\ 2,4 & 3,0 & 4,1 & 0,2 & 1,3 \\ 3,1 & 4,2 & 0,3 & 1,4 & 2,0 \\ 4,3 & 0,4 & 1,0 & 2,1 & 3,2 \end{Vmatrix}$$

(b) Superposition of the two Latin squares.

$$\begin{array}{|ccccc|} \hline 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ \hline \end{array} \qquad \begin{array}{|ccccc|} \hline 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \\ \hline \end{array}$$

(c) Additional matrices for OLS code construction.

Figure 3: Latin squares and superposition: In (a), two Latin squares or order 5. Their superposition is shown in (b). (c) shows two additional matrices used for the construction of the OLS code.

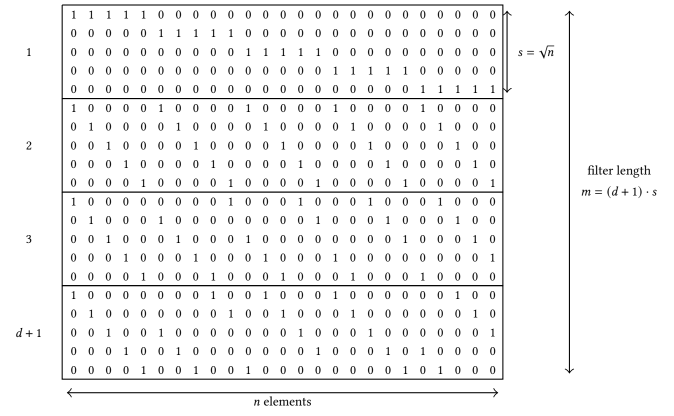
rows or columns. By the definition of Latin squares, if one such matrix is superimposed with a Latin square (and in particular those from Fig. 3(a)), every ordered pair of elements appears exactly once.

Orthogonal Latin Square (OLS) codes are constructed from a set of mutually orthogonal Latin squares. The code covers s^2 data bits and each Latin square adds s parity bits that check the bits that have the corresponding value. To do so, each value in each Latin square represents a parity bit that covers the bits on which it appears. For example, in Fig. 3(a), the first parity check bit of the first Latin square (that corresponds to value 0) would check bits 0, 9, 13, 17 and 21. As each data bit corresponds to a position in the Latin square, it participates exactly in one of the s parity checks associated with a Latin square. Additionally, since the Latin squares are mutually orthogonal, each pair of parity checks has at most a bit in common.

Therefore, by construction, an OLS code built using $d + 1$ Latin squares has a parity check matrix H that satisfies:

- Each pair of columns only has at most a single position with a bit of one in common.
- Each column that corresponds to a data bit has the same number of ones: $d + 1$.
- Each column that corresponds to a data bit has size $(d+1) \cdot s$ and has a single bit of one in the first s bits, another on the second group of s bits and so on.

Based on the properties of the matrices, we can see that an OLS code with $d + 1$ ones in the columns can be used to implement a Bloom filter with a zero false positive universe given by the size of the data block of the code for up to d elements (at least $d + 1$ elements are needed to create a false positive).



$$
\begin{array}{c|ccccccccccccccccccccccccc|}
 & 1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
 & 0&0&0&0&0&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1 & 0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0\\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&0&0&0\\
 & 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1\\
\hline
 & 1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0\\
 & 0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0\\
2 & 0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0\\
 & 0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0\\
 & 0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1\\
\hline
 & 1&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0\\
 & 0&1&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0\\
3 & 0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&1&0\\
 & 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&1\\
 & 0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0\\
\hline
 & 1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0\\
 & 0&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0&0&0&1&0\\
d+1 & 0&0&1&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&0&1\\
 & 0&0&0&1&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&0&1&0&0&0&0\\
 & 0&0&0&0&1&0&0&1&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&0&0\\
\hline
\end{array}
$$

Figure 4: Parity check matrix H of size $(d+1) \cdot s \times s^2 = 20 \times 25$ for an Orthogonal Latin Square code with universe size $|U| = n = s^2 = 25$ for $s = 5$ and maximal set size $d = 3$. The required filter length is $m = (d+1) \cdot s = (d+1) \cdot \sqrt{n} = 4 \cdot 5 = 20$.

In general, the Bloom filters based on OLS codes built from a Latin Square of size s has the following parameters:

- Universe size: $n = s^2$.
- Bloom filter size: $(d+1) \cdot s$.
- Free positive zone: d.

Elements insertion and membership queries can be performed in the OLS filter simply in a way similar to these operations in the original Bloom filter with specific mappings of the hash functions described by the matrix where each element in the universe is associated with a column. An insertion of an element is performed by setting in the filter the bits set in the column. A query is performed by checking that all bits set in the column are set also in the filter. Similarly, membership of multiple elements can be done by checking that all bits that should be set for one of them are set in the filter.

3.2 Illustrative Example

As an example, the $20 \times 25 = (d+1) \cdot s \times s^2$ matrix M of an OLS filter with $d = 3$, $n = 25$ and $s = \sqrt{n} = 5$ is shown in Fig. 4. The filter length is $m = (d+1) \cdot s = 20$ defined for a universe of size $n = 25$ and guarantees no false positives for sets of to size $d = 3$. It includes $d + 1 = 4$ parts of s bits. In this case, the first two sub-matrices used are these from Fig. 3(c) and the third and fourth correspond to the Latin squares on Fig. 3(a). In case the maximal set size is smaller $d = 2$ a shorter filter of length $(2+1) \cdot 5 = 15$ can be satisfying. Such a filter is simply obtained by considering only the first $d + 1 = 2 + 1 = 3$ first sub-matrices, each of $s = 5$ rows.

3.3 False positive free zone property

In the following main theorem, we prove the OLS filter completely avoids false positives in a well-defined zone.

Theorem 3.1. *Consider an OLS filter defined for a universe $U = \{1, \ldots, n = s^2\}$. For $d \geq 1$, an OLS filter of size $(d+1) \cdot s$ bits completely avoids false positives for any represented set of size $|S| \leq d$.*

Proof. The proof relies on the structure of the partial parity check matrix of the OLS code for which an example is shown in Fig. 4. We refer to some combination of at most d columns from the matrix and explain that their OR does not (fully) cover any other column. A column is divided into $d + 1$ parts of s bits with exactly a single bit of one in each part. Let j be a column index and Q be a set of column indexes satisfying $|Q| \leq d, j \notin Q$. We explain that column j cannot be covered by the cumulative OR for the columns of Q. By contradiction: To be fully covered, all the $d + 1$ bits of one of the columns j have to be covered by the (at most) d columns that refer to Q. By the pigeonhole principle, one column in Q covers at least two bits of one of column j. This contradicts the fact that each pair of columns in the matrix has at most a single bit of one in common. Thus at least one of bits of one in column j is not set in the cumulative OR for the columns in Q. Accordingly, a query of a general element in $U \setminus S$ would not lead to a false positive. ◻

3.4 Analytical Comparison with previously proposed filters

We compare the performance of the proposed OLS filter with the only previously known construction with a false positive free zone [21] known as the EGH filter. To do so, we compare the minimal filter size required for a given universe size and a maximal allowed set size such that false positives are guaranteed to be avoided. We denote by $m(n, d)$ the minimal required filter size for a universe $U = \{1, \ldots, n\}$ of size n and a maximal set size d. Clearly, the function $m(n, d)$ is (weakly) monotonically increasing in both parameters.

We first refer to the memory complexity of the EGH filter of [21]. Recall that the filter is composed of k segments that refer to the k smallest primes that their product equals at least n^d. The filter length equals the sum of the primes.

The memory complexity of the EGH filter of [21] satisfies $m(n, d) = O(d^2 \cdot \log_2 n)$.

The property was shown in [21]. The minimal filter length $m(n, d)$ is given as the sum of the first k primes $m(n, d) = \sum_{j=1}^{k} p_j$ where k is determined as the minimal value for which the inequality $\prod_{j=1}^{k} p_j \geq n^d$. holds. By relying on [14], it was concluded that the bound can be satisfied with $p_k \leq \lceil 2d \log(n) \rceil$ such that the bound on the sum of the first k primes follows.

We now refer to the suggested OLS based filters with its described construction.

Theorem 3.2. *The memory complexity of the OLS filter satisfies: $m(n, d) = (d+1) \cdot \sqrt{n}$.*

Proof. The value $m(n, d)$ simply expresses the Bloom filter length, given as the length of a column in a given construction of the OLS filter. To allow a maximal set size d, a column of the code generator matrix has $d + 1$ parts, each part of s bits where the universe size is $n = s^2$. The OLS filter value for a set is simply given as the OR of multiple columns and its length equals the length of a column $(d+1) \cdot \sqrt{n}$. ◻

The above behaviors demonstrate us that the OLS should be used when the represented set d is large while the universe size is still restricted at some degree. Intuitively, this is described as the following relationship between the two complexities.

Corollary 3.3. *For an asymptotically increasing maximal set size d the OLS filter requires improved memory than the EGH filter.*

This analysis shows that depending on the application the proposed filters can provide an advantage in some settings over the filters proposed in [21].

4 POLYNOMIAL BASED FILTERS WITH A FALSE POSITIVE FREE ZONE

4.1 Motivation

We have been discussing so far two approaches for the design of a Bloom filter with a false positive free zone. Let again n describe the size of the universe $U = \{0, \ldots, n-1\}$ and d the maximal set size for which false positives should be avoided. The existing EGH filter requires memory of $O(d^2 \cdot \log n)$ bits [21] and our suggested OLS filter from Section 3 achieves complexity of (exactly) $(d+1)\sqrt{n}$ bits. We saw that the EGH filter becomes expensive for larger d while the OLS becomes less attractive for large n. In this section, we would like to allow even larger flexibility in the design, trying to avoid the expensive components of d^2 in the complexity of the EGH filter as well as the $\sqrt{n}$ in that of the OLS filter. Accordingly, we describe a construction that requires a complexity of $((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$ where t is an integer parameter we control. This construction is based on the use of Polynomials and we refer to it as the *POL filter*.

4.2 Construction based on Polynomials

Let t satisfy $((t-1) \cdot d + 1) < \sqrt[t]{n}$ and assume for simplicity that $\sqrt[t]{n}$ is an integer. We associate an element $y \in U$ with a polynomial P_y of degree $t-1$ such that each of its t coefficients $a_0, \ldots, a_{t-1}$ belong to $[0, \sqrt[t]{n}-1]$. The coefficients are selected such that $P_y(x) = \sum_{i=0}^{t-1} a_i \cdot x^i$ satisfies $P_y(\sqrt[t]{n}) = \sum_{i=0}^{t-1} a_i \cdot (\sqrt[t]{n})^i = y$. Note that $a_0, \ldots, a_{t-1}$ are well defined given $y, \sqrt[t]{n}$ as the coefficients in a representation of y in base $\sqrt[t]{n}$. For instance a_0 equals $y \bmod \sqrt[t]{n}$ and $a_1 = ((y - a_0)/\sqrt[t]{n}) \bmod \sqrt[t]{n}$, etc. Let R_t denote the range $[0, \sqrt[t]{n}-1]$ following an assumption that $n = |U|$ is fixed.

We characterize each element $y \in U$ by $((t-1) \cdot d + 1)$ values for $x \in [0, (t-1) \cdot d]$ of the polynomial $P_y(x)$ such that calculations are performed modulo $\sqrt[t]{n}$. We describe each such value in $|R_t| = \sqrt[t]{n}$ bits such that a single bit that corresponds to the value equals one while all others are zero. Similarly, we describe the ordered list of values in a binary vector of length $((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$.

We refer to a generator matrix with $n = |U|$ columns. For $y \in [1, n]$ the y^{th} column keeps the $((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$ bits describing the element $y \in U$. The Bloom filter in that construction is of length $((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$. Equivalently, a column in the matrix is composed of $((t-1) \cdot d + 1)$ groups, each of $\sqrt[t]{n}$ bits. For $y \in U$, the j^{th} group (for $j \in [0, (t-1) \cdot d]$), refers to the value $P_y(j)$.

A pseudocode for the POL filter construction for a given set appears in Algorithm 1. Earlier to the insertion of elements, the filter is initialized as an array of zero bits. The filter for a set S is (as in other constructions) simply given by the cumulative OR for the

```
Input: Set $S$ with $x \in U = \{0, \ldots, n-1\}$, parameters $d, t$
Result: POL filter $A_S$ for the set $S$
begin
1   $A_S = (0) \times ((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$
2   for $y$ in $S$ do
3       $A_y = (0) \times ((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$
4       $y' = y$
5       for $i \in [0, (t-1)]$ do
6           $a_i = y' \bmod \sqrt[t]{n}$
7           $y' = (y' - a_i)/\sqrt[t]{n}$
        end
8       $P_y(x) = \sum_{i=0}^{t-1} a_i \cdot x^i$
9       for $j \in [0, ((t-1) \cdot d]$ do
10          $A_y[j \cdot \sqrt[t]{n} + (P_x(j) \bmod \sqrt[t]{n})] = 1$
        end
11      $A_S = A_S$ OR $A_y$
    end
12  Return $A_S$
end
```

Algorithm 1: POL Filter construction

corresponding $|S|$ columns of the matrix. Its length is $((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$ and we can see it as composed of $((t-1) \cdot d + 1)$ groups.

The insertion and query operations work as follows.

Element Insertion: Upon an insertion of an element $y \in S$, the vector is set as the OR of the previous vector with the column for y. Equivalently, upon such an insertion of an element y, in each of group $j \in [0, (t-1) \cdot d]$ of bits, we set the bit that refers to $P_y(j)$ (if it has not been set before).

Element Query: Upon a query of an element $y \in U$, $((t-1) \cdot d + 1)$ bits are accessed. The bits that correspond to $P_y(j)$ for $j \in [0, (t-1) \cdot d]$, one in each group, are examined. A positive answer to the query is returned if all bits are set. If at least one of them is zero, a negative answer is returned for the query.

4.3 Illustrative Example

For $n = 7^3 = 343$ for instance, we have $\sqrt{n} = 7^{1.5} \approx 18.52$ and setting $t = 3$ implies $\sqrt[t]{n} = 7$. Here, the length of the OLS filter is $(d+1)\lceil\sqrt{n}\rceil = 19(d+1)$. The POL filter length is $((t-1) \cdot d + 1) \cdot \sqrt[t]{n} = (2d+1) \cdot 7 = 14d+7$, shorter than that of the OLS filter for any $d \geq 1$. The generator matrix of the POL filter has $n = 343$ columns and when $d = 2$ each column is of $14d+7 = 35$ bits. We demonstrate some of the columns of the matrix, each composed of $(t-1) \cdot d + 1 = 5$ groups of $\sqrt[t]{n} = 7$ bits. For instance the element $y = \sqrt[t]{n} = 7$ corresponds to polynomial $P_y(x) = x$, and we have a column with the following 35 bits (1000000 0100000 0010000 0001000 0000100), where spaces are shown just to simplify the presentation and do not exist. For instance in the second group the bit that corresponds to $P_y(1) = 1$ is set. Likewise for $y = (\sqrt[t]{n})^2 + 1 = 50$ the polynomial is $P_y(x) = x^2 + 1$ and we have a column of the following 35 bits (0100000 0010000 0000010 0001000 0001000), where for instance in the fifth group we have $P_y(4) = 4^2 + 1 = 17 = 3$ (modulo $\sqrt[t]{n} = 7$).

4.4 Guarantees and Analytical Cost

We show a guarantee on the false positive zone holds.

Theorem 4.1. *Consider a filter with the described construction defined for a universe $U = \{1, \ldots, n\}$. For $d \geq 1$ let t be an integer satisfying $((t-1) \cdot d + 1) < \sqrt[t]{n}$ such that $\sqrt[t]{n}$ is prime. The filter of size $((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$ bits completely avoids false positives for any represented set of size $|S| \leq d$.*

Proof. Consider an element $y \in U$ for which the filter provides a positive indication. This happens when for $j \in [0, (t-1) \cdot d]$, the corresponding bit of y was set by an element in S. Namely, for all the $(t-1) \cdot d + 1$ values of j, the polynomial value of $P_y(j)$ equals to one of the $|S|$ polynomial values computed for the same j. Since $|S| \leq d$, by the pigeonhole principal there is necessarily a member $w \in S$ such that $|\{j | P_w(j) = P_y(j)\}| \geq \lceil ((t-1) \cdot d + 1)/d \rceil = t$. The two polynomials $P_w(j), P_y(j)$, each of degree t collide on at least t distinct values. By $\sqrt[t]{n}$ being a prime, it implies that P_w exactly equals P_y. Accordingly, $y = w$ and $y \in S$. □

The following theorem follows from the construction details and desribes the memory complexity of the proposed POL filters.

Theorem 4.2. *The memory complexity of the POL filter can be selected as $m(n, d) = ((t-1) \cdot d + 1) \cdot \sqrt[t]{n}$ for an integer t satisfying $((t-1) \cdot d + 1) < \sqrt[t]{n}$ and in addition $\sqrt[t]{n}$ is prime.*

5 ACCURATE MEASUREMENT BASED ON THE FPFZ

We demonstrate how the concept of the false positive free zone can be applicable beyond set representation with Bloom filters. We refer to additional counter based sketches that typically also encounter a tradeoff between memory efficiency and accuracy. We explain how the ideas of this work are applicable to eliminate error in such sketches. Interestingly, we are the first to introduce the notion of estimation error free zone in such sketches. This connection is not limited to constructions from our work and applies also to that of [21] that has not been discussed beyond Bloom filters.

The Count-Min Sketch Estimating flow size is a required capability in many networking applications, in fields as diverse as accounting, monitoring, load balancing, routing and filtering and even beyond networking. Counting the exact size for every flow is often challenging due to a typically large number of active flows on a specific time, making it difficult to maintain a counter-per-flow within a memory accessible at line rate. There can be two possible errors in an estimation of a flow size: Overestimations and underestimations. The state-of-the-art data structure for flow size estimation is the Count-Min Sketch (CM) suggested by Cormode and Muthukrishnan in 2005 [8].

The CM relies on a two-dimensional array of counters initialized to values of 0s. A set of k hash functions are used to map a flow to k counters, one in each of its rows. The structure naturally fits that of the OLS filter with a single bit set in each subset of $\sqrt{n}$ bits of the filter as well as that of the POL filters with a single bit set in each subset of $\sqrt[t]{n}$ bits. Upon the arrival of a flow, each of these counters is incremented by the length of the flow sequence. To estimate the size of a flow, its k counters are considered and the size is estimated as the minimal value among the k counters. Since multiple flows can contribute to the same counter, the computed value can be larger than the exact one in case other flows contributed to all k counters implying overestimations. On the other hand, the CM

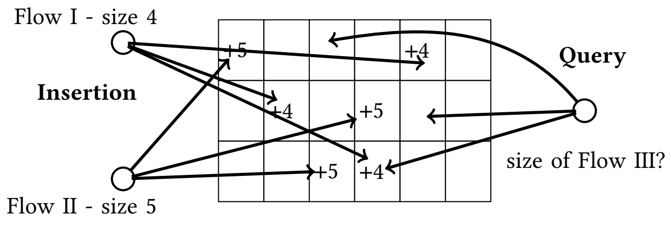


Figure 5: The Count-Min sketch (CM) [8], allowing flow size estimation. A flow size is estimated as the minimum among the counters it is mapped to by a set of hash functions.

completely avoids underestimations. A tradeoff exists between the level of accuracy and the amount of allocated memory such that more allocated memory reduces collisions among flows. Similarly, reducing the number of active flows improves accuracy. However, to the best of our knowledge there are no known results on how to completely avoid flow size overestimations even when the number of flows is small.

The CM is illustrated in Fig. 5. Flows I, II of size 4 and 5, respectively, are recorded in the sketch (shown on the left side). Each flow increases the value of $k = 3$ counters by its own size. The size of Flow III (right side) is estimated by querying the CM. Its size is estimated as the minimal among the k counters it is mapped to.

We describe how we can avoid errors such as the typical overestimation of flow size when constraints on the number of estimated flows are satisfied. This can be achieved using a mapping of flows to CM counters given as a mapping implying a false positive free zone for the Bloom filter. We assume a finite number of possible flows such that the number of flows with non-zero amount of traffic is bounded. We formalize that in the following theorem.

Theorem 5.1. *Consider a CM sketch defined for up to n flows using an array of m counters. Let M be a $m \times n$ binary matrix implying a Bloom filter false positive free zone for sets $|S| \leq d$ selected from a universe of size n. Assume a mapping of flows to counters in the CM sketch that follows the matrix M. In case at most d of the flows have a non-zero flow size, the flow size reported by the CM is exactly its right value for all n flows. Moreover, with $d+1$ non-zero flows, the size of each of these is reported correctly.*

Proof. We show that flow size is computed correctly for a flow regardless whether it is of size zero or has a positive size. Consider an arbitrary flow. By the definition of M, it follows that in the CM the flow maps to one counter such that none of the flows in S is mapped to this counter. Besides this estimated flow there are at most d other non-zero flows. Accordingly, the counter has a value which exactly equals the right flow size. Since the CM reports a flow size based on the minimal among several counters the existence of one counter with the correct value implies an exact size estimation. □

6 IMPLEMENTATION CONSIDERATIONS

In this section we study implementation of the OLS and POL filters in comparison with the traditional Bloom filter. We also discuss implementation in the emerging programmable switches.

The OLS filter In a traditional Bloom filter, the positions accessed upon insertions and queries are determined by computing hash functions for the element [3]. Instead, in the proposed OLS filters, those positions are obtained from the matrix that defines the filter. In more detail, for a given element x, the positions are those that have a one in the column of the matrix that corresponds to x. Therefore, to implement the proposed OLS filters, those positions should be stored and used for lookups and insertions. For the OLS filter, it might be difficult to present a simple formula for the bits selected for a particular element.

Two scenarios can be considered for implementation. The first for small filter, e.g. of the memory word length allowing the matrix to be stored in memory. To insert an element, the corresponding matrix column is read and ORed with the filter. To check if an element is stored in the filter, the corresponding column from the matrix is read and ANDed with the filter. Then the number of ones is compared to $d + 1$ and only upon an equality a positive answer is given. This implementation is well suited for small filters as it requires just two memory accesses for lookups: one to retrieve the column and one to retrieve the filter and some simple operations that are efficiently supported by modern processors. A drawback is that the full matrix needs to be stored in memory. This is not an issue when many filters are used in the system all sharing the matrix. That would be the case for example when the filters are included in the packet headers and used to convey information for multicast [37] or SDN attributes [29].

The second scenario is when the filter is much larger than the memory word. In that case, an efficient alternative is to store only the positions to check for each element instead of the full matrix. Then, to insert (check) an element, the positions associated with that element are retrieved and then set (read). This would require several memory accesses as in a traditional Bloom filter. The OLS filter overhead is storing the bit positions to for each element.

The POL filter In the case of the POL filter (unlike the OLS filter), we can present a simple formula for the selection of bits that correspond to a particular element. This allows an implementation of the filter without maintaining the matrix representing the mapping of elements to bits. In more detail, given an element y we first compute the coefficients of the polynomial $P_y(x)$ and then evaluate it for $x \in [0, (t-1) \cdot d]$ to obtain the positions that correspond for the particular element. This requires computing $t - 1$ polynomial coefficients based on modulo operations and evaluating the polynomial $(t-1) \cdot d + 1$ times. This would for most implementations be slightly more complex than computing the hash functions of the traditional Bloom filter. The detailed formulas appear in the pseudocode of Algorithm 1. Another option is to maintain the matrix for speedup of the mapping process. This can be especially useful when multiple filters share the same matrix.

Programmable Switches The emergence of P4-based programmable data planes [5] is an exciting opportunity to push the implementation of data structures and network algorithms to programmable switches. In fact, both Bloom filters [18] and count min sketches [32] have been implemented in programmable switches. Therefore, it is of interest to discuss the implementation of the proposed filters and sketches in P4. For the POL filter, arithmetic operations could be used to compute the bits to set (check) in the filter during insertions (queries). The P4 specification supports the operations needed including modulo on integer numbers. However, performance may not be optimal due to the large number of arithmetic operations needed and the fact that the switch pipelines are typically not optimized for a large number of arithmetic operations. Another alternative that is applicable to both OLS and POL filters is to use a first table to obtain the positions to check on the filter. This would require an additional access and memory for the table of positions to check. This cost may be acceptable when several filters are used in parallel so that the cost of the first table is shared among them. However, such sharing may imply contraints.

The implementation can take advantage of the special structure of the bits accessed by an element in the two proposed filters. The filter is composed of groups of bits such that a single bit is accessed in each group. See for instance the OLS filter from Fig. 4 where each group is of length $\sqrt{n}$ bits. For the POL filter each part is of length $\sqrt[t]{n}$. In both filters the number of groups is fixed. We can set the number of pipeline stages based on the number of groups so one bit is accessed in each stage. The number of required stages is $d + 1$ for the OLS filter and $(t-1) \cdot d + 1$ for the POL filter. The parameter t can be selected based on the number of available stages. A similar reasoning applies when the structures are used for counting sketches here with a counter in each position. In summary, it seems that the proposed schemes could be implemented in programmable data planes but with additional overheads in terms of memory and/or speed compared to the original Bloom filters and count min sketches. Full implementation and evaluation of the schemes in programmable switches is left for future work.

7 PRACTICAL CONFIGURATIONS AND EVALUATION

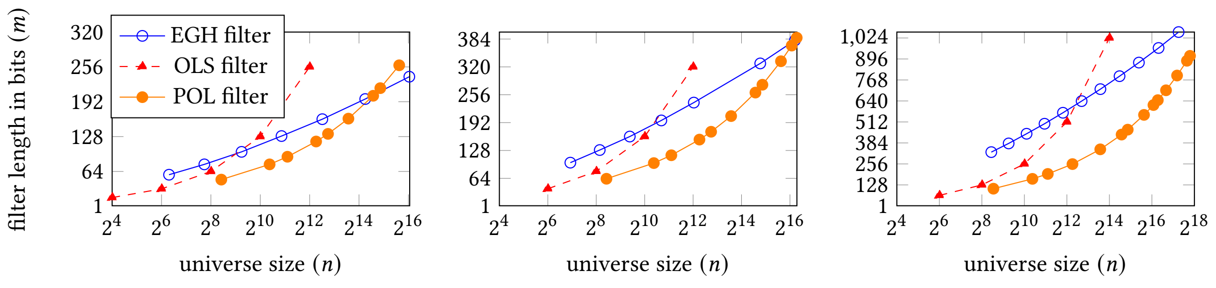
This section first discusses practical configurations of the proposed OLS and POL filters and the EGH filter of [21] giving numerical examples that complement the asymptotic analysis. Then, the false positive rate of the proposed filters when they operate beyond their FPFZ is evaluated by simulation and compared to that of a traditional Bloom filter. Last, accurate flow size estimation with the Count-Min Sketch is also demonstrated.

7.1 Practical Configurations - FPFZ Size

Configurations for the existing EGH filter and the proposed OLS and POL filters for some practical values of the universe size and filter length are summarized in Fig. 6. Note that the values of the maximal set size d shown are the worst case to compare the proposed filters against the EGH filters as they become more competitive as d increases. From the plots it can be seen, that for example, when $d = 2, 3$, the OLS filters provide better parameters when n is smaller than 256. For $d = 4$ the range increases to n being smaller than 1024. In the case of the POL filter with $t = 3$, the ranges are extended to over 2000, 16000 and 64000 for $d = 2, 3, 4$, respectively.

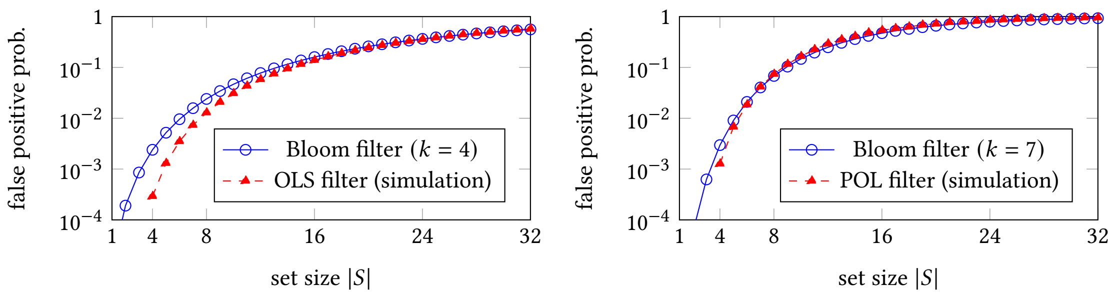
The last plot shows, for $d = 7$ and filter size smaller than 512 bits, the benefits of the proposed filters when the maximal set size increases. The OLS and POL filters can reduce the filter size or extend the universe size.

These examples show how the proposed filters can be used to either extend the universe size for the same filter size or to reduce the filter size for a given universe size when filter size is small. Note



(a) maximal set size $d = 3$ (b) maximal set size $d = 4$ (c) maximal set size $d = 7$

Figure 6: Illustrative comparison of the length in bits m vs. universe size n for the suggested and existing filters.



(a) $n = 256, d = 3, m = 64$ (b) $n = 343, d = 3, m = 49$

Figure 7: False positive probability beyond the false positive free zone for the OLS and POL filters.

that this is the case, for example when the filter is embedded as a field of the packets in applications like SDN attribute encoding [29] or multicast addressing [37]. Recall (Table 2) that the memory complexity of the OLS and POL filters grows with the universe size n and $\sqrt{n}$ or $\sqrt[3]{n}$, respectively, unlike the logarithmic dependency of the EGH filter. Accordingly, for large enough universe size both OLS and POL filters require more memory than the EGH filter.

7.2 Bloom Filter False Positive Probability

To check that the proposed filters have the expected false positive free zone, they have been implemented and then the entire universe has been tested for one thousand combinations of d elements in the set. The false positive rate when the filter stores more than d elements was also measured.

We considered the following OLS filters and POL filters:

- Filter OLS O: $n = 256, d = 3, m = 64$.
- Filter POL P: $n = 343, d = 3, m = 49$.

Those values correspond to universes of a few hundred elements which are sufficient for applications using a Bloom filter in packet headers to convey information to the network nodes [29],[37].

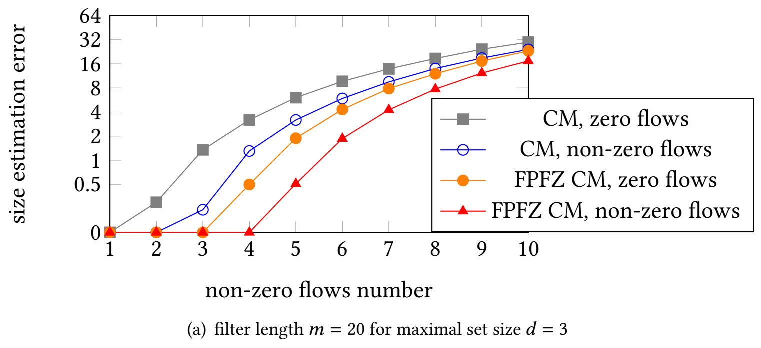
The simulation results are shown in Fig. 7 and compared with the approximated false positive probability of a traditional Bloom filter [3, 6]. While no false positives have been observed for sets of at most size d, the false positive rates of the filters match reasonably well that of a Bloom filter outside the false positive free zone where for the OLS filter a modest improvement was observed.

7.3 Count-Min Sketch Accuracy

We compare the accuracy of the Count-Min Sketch (CM) in estimating flow size in two possible variants. The first is based on random mapping of elements into counters (with distinct mappings to any pair of elements). The second is based on Section 5 and relies on a mapping of elements that implies a false positive free zone where bits are replaced by the counters of the CM. In particular we refer to a mapping of the OLS filter as illustrated in Fig. 4.

We assume a universe of $n = 25$ flows. We vary in the range $[1, 10]$ the number of flows with a positive size. The size of a non-zero flow is uniformly-distributed in the range $[1, 100]$. We track flow size in the CM and estimate a flow size as the typical CM's estimation by taking the minimum among the counters a flow maps to. Recall the CM either computes a flow size accurately or overestimates its size. We are interested in learning the cases where an exact estimation is guaranteed as well as in the average flow size overestimation. As indicated by Theorem 5.1, we can observe a difference in the accuracy of the estimation of the flows of size zero and those of the non-zero flows. Intuitively, with a fixed number of non-zero flows, flows of size zero are potentially affected by other flows. Fig. 8 presents the overestimation in logarithmic scale.

First, Fig. 8(a) refers to a filter of length $m = 20$ allowing a false positive free zone with $d = 3$ for the a universe with a number of



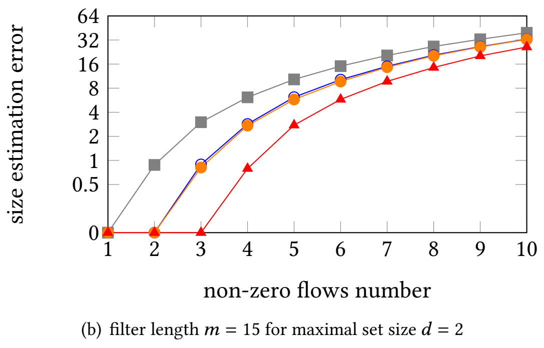


Figure 8: Comparison of the accuracy of the Count Min sketch (CM) with a mapping based on the false positive free zone vs. random mapping. Construction is based on the OLS filter from Fig. 4.

elements that equals the total flow number $n = 25$. Consider the CM with the FPFZ mapping illustrated in the first two right curves. With up to four non-zero flows, each of these flows is evaluated correctly. Likewise, with at most three non-zero flows, all flows of size zero also observe exact size estimation and do not encounter overestimation. This exactly follows Theorem 5.1. On the other hand, with the traditional random mapping the original CM, we can observe a overestimation in the size of a flow of size zero even when there are only two non-zero flows. Flows of size zero are sometimes overestimated when there are two or more non-zero flows while due to the uniqueness of the hashing of the elements cannot be overestimated by a single non-zero flow. In addition, the graph shows an improvement in the average overestimation. For instance, with five non-zero flows, the overestimation of these flows reduces from 3.18 in the traditional CM to only 0.51 in the suggested FPFZ CM. Likewise, on such a case the overestimation of the zero flows reduces from 6.06 to 1.88.

Fig. 8(b) shows the results for the corresponding filter of length $m = 15$, implying a false positive free zone for $d = 2$. This refers to the first three bit groups in the OLS filter from Fig. 4. As implied by the analysis, using a filter with $d = 2$ further limits the number of non-zero flows for which accurate size estimations are guaranteed. E.g., for the new CM variant three non-zero flows are sufficient to imply an overestimation in their size and two are sufficient to imply an overestimation in the size of zero flows. Similarly, a larger overestimation is observed for both the new CM variant based on the FPFZ and the traditional CM. However, the higher accuracy of the new variant is still meaningful. For instance, again with five non-zero flows the overestimation of zero flows is 10.29 in the traditional CM and equals 5.77 in the FPFZ CM, a reduction of 43.9% in the overestimation.

8 RELATED WORK

The work of [21] presented the notion of the false positive free zone and suggested the EGH filter (discussed in Section 2) as its construction. On the contrary, previous works tried to reduce the probability for false positives in Bloom filters rather than to eliminate them completely. We overview such works. Likewise, the careful mapping of elements into the filter bits in our proposed constructions of the false positive free zone is related to broader approaches to reduce and avoid collisions in hashing.

Reducing the Bloom filter false positive probability For a given set size, the false positive probability is a function of the allocated memory. The tradeoff of the Bloom filter itself was shown to be close to an ideal one implied by properties of Information theory. To support deletions, often counters are used instead of bits, requiring more memory for a given false positive probability. A line of work focused on data structure constructions for improving this tradeoff while supporting deletions. A work by Bonomi et al. [4] describes a scheme where a fingerprint of an element is stored in a memory location allowing more accurate answer to queries. The use of fingerprints stored in several locations was considered in [33] to further reduce the false positive rate. The VI-CBF [36] uses variable counter increments for encoding some element information to reduce false positives. Other works refer to tradeoff between the false positive probability to that of the false negative probability. In the (traditional) Bloom filter, false negatives are completely avoided. An approach suggested by Donnet et al. [12] is to randomly select bits of ones and reset them to zero. Reducing the ratio of 1s clearly reduces the false positive probability but also implies false negatives. A similar idea was described by Laufer et al. as the Generalized Bloom Filter [23]. Finally, [9], proposes the use of asymmetric error correction codes to reduce the false positive rate of filters used to perform multi-set membership checking. The cuckoo filter [15] allows set representation with low false positive probability by keeping element fingerprints. The adaptive cuckoo filter exploits correlation in the queries to avoid having the same false positive repeating many times [30].

Collision resolution and avoidance techniques Hash tables and other data structures with hashing-based mapping often suffer from hash collisions. Similar to the Cuckoo filter, the *cuckoo hash table* [31] relies on flexibility to insert an element to one of multiple entries (as indicated by multiple hash functions) in order to reduce element collisions and achieve high occupancy of tables earlier to the detection of an insertion failure. *Open addressing* [22] solves collisions in hashing by computing prioritized alternative locations for an element. A sequential location can be computed in various ways, e.g. by adding a constant to the address or rehashing. Chaining simply allows an entry to keep a list of located elements. *Hopscotch hashing* [17] allows storing an element within a small neighborhood around a counter mapped by a hash function. *Coalesced hashing* [40] combines chaining and open addressing.

9 CONCLUSIONS AND FUTURE WORK

Restricting the negative impact of false positives is a desired property of Bloom filters. A false positive free zone guarantees their avoidance when a restriction on the universe size and the represented set size holds. Our first contribution is the description of two novel constructions for the Bloom filter while allowing the existence of the zone in memory complexity which is linear in the size of the represented set. The constructions are designed for scenarios of larger sets and suggest new tradeoffs between the universe size and the set size, improving the known tradeoff for cases of increasing set size. Our main contribution is the applicability of the false positive free zone to provide accuracy in supporting functionality beyond set membership such as flow size estimation in the Count-Min sketch. Future work will focus on trying to find additional sketches for which the Bloom filter free zone can guarantee exact counting. A particular interest will be in those recently suggested generic data structures that support multiple measurement tasks.

10 ACKNOWLEDGEMENTS

Ori Rottenstreich was partially supported by the Taub Family Foundation, by the German-Israeli Foundation for Scientific Research and Development (GIF), by the Gordon Fund for System Engineering as well as by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate. Pedro Reviriego would like to acknowledge the support of the TEXEO project TEC2016-80339-R funded by the Spanish Ministry of Economy and Competitivity and of the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496. This work is also partially supported by the European Commission in the frame of the Horizon 2020 project PASSION (grant #780326).

REFERENCES

[1] Zainab Abaid, Mohamed Ali Kâafar, and Sanjay Jha. 2017. Early detection of in-the-wild botnet attacks by exploiting network communication uniformity: An empirical study. In *IFIP Networking*.

[2] Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. 2018. A Comparison of Performance and Accuracy of Measurement Algorithms in Software. In *ACM SOSR*.

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

[4] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters. In *ESA*.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Comput. Commun. Rev.* 44, 3 (2014), 87–95.

[6] Andrei Z. Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1, 4 (2003), 485–509.

[7] Tao Chen, Deke Guo, Yuan He, Honghui Chen, Xue Liu, and Xueshan Luo. 2013. A Bloom filters based dissemination protocol in wireless sensor networks. *Ad Hoc Networks* 11, 4 (2013), 1359–1371.

[8] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The Count-Min sketch and its applications. *J. Algorithms* 55, 1 (2005), 58–75.

[9] Haipeng Dai, Yuankun Zhong, Alex X. Liu, Wei Wang, and Meng Li. 2016. Noisy Bloom Filters for Multi-Set Membership Testing. In *ACM SIGMETRICS*.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *ACM SIGMOD*.

[11] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. 2006. Longest prefix matching using Bloom filters. *IEEE/ACM Trans. Netw.* 14, 2 (2006), 397–409.

[12] Benoit Donnet, Bruno Baynat, and Timur Friedman. 2006. Retouched Bloom filters: Allowing networked applications to trade off selected false positives against false negatives. In *ACM CoNEXT*.

[13] Benoit Donnet, Bamba Gueye, and Mohamed Ali Kaafar. 2012. Path similarity evaluation using Bloom filters. *Computer Networks* 56, 2 (2012), 858–869.

[14] David Eppstein, Michael T. Goodrich, and Daniel S. Hirschberg. 2007. Improved Combinatorial Group Testing Algorithms for Real-World Problem Sizes. *SIAM J. Comput.* 36, 5 (2007), 1360–1375.

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

[16] Richard W Hamming. 1950. Error detecting and error correcting codes. *Bell System technical journal* 29, 2 (1950), 147–160.

[17] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *International Symposium Distributed Computing (DISC)*.

[18] J. Hill, M. Aloserij, and P. Grosso. 2018. Tracking Network Flows with P4. In *IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*.

[19] MY Hsiao, DC Bossen, and RT Chien. 1970. Orthogonal Latin square codes. *IBM Journal of Research and Development* 14, 4 (1970), 390–394.

[20] A Donald Keedwell and József Dénes. 2015. *Latin squares and their applications.* Elsevier.

[21] Sándor Z. Kiss, Éva Hosszu, János Tapolcai, Lajos Rónyai, and Ori Rottenstreich. 2018. Bloom Filter with a False Positive Free Zone. In *IEEE INFOCOM*.

[22] Donald Knuth. 1973. *The Art of Computer Programming. Part 3, Sorting and Searching.* Addison-Wesley.

[23] Rafael P. Laufer, Pedro B. Velloso, and Otto Carlos Muniz Bandeira Duarte. 2011. A Generalized Bloom Filter to Secure Distributed Network Applications. *Computer Networks* 55, 8 (2011), 1804–1819.

[24] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*.

[25] Shu Lin and Daniel J Costello. 2004. Error Control Coding. 2nd Edition. *Englewood Cliffs, NJ: Prentice-Hall* (2004).

[26] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM*.

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

[28] Lailong Luo, Deke Guo, Jie Wu, Ori Rottenstreich, Qian He, Yudong Qin, and Xueshan Luo. 2017. Efficient Multiset Synchronization. *IEEE/ACM Trans. Netw.* 25, 2 (2017), 1190–1205.

[29] Robert MacDavid, Rüdiger Birkner, Ori Rottenstreich, Arpit Gupta, Nick Feamster, and Jennifer Rexford. 2017. Concise Encoding of Flow Attributes in SDN Switches. In *ACM SOSR*.

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

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

[32] F. Pereira, N. Neves, and F. M. V. Ramos. 2017. Secure network monitoring using programmable data planes. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*.

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

[34] Jaikumar Radhakrishnan, Smit Shah, and Saswata Shannigrahi. 2010. Data structures for storing small sets in the bitprobe model. In *Springer ESA*.

[35] Pedro Reviriego, Salvatore Pontarelli, Alfonso Sánchez-Macián, and Juan Antonio Maestro. 2014. A Method to Extend Orthogonal Latin Square Codes. *IEEE Trans. VLSI Syst.* 22, 7 (2014), 1635–1639.

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

[37] Mikko Särelä, Christian Esteve Rothenberg, Tuomas Aura, András Zahemszky, Pekka Nikander, and Jörg Ott. 2011. Forwarding anomalies in Bloom filter-based multicast. In *IEEE INFOCOM*.

[38] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SOSR*.

[39] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2012. Theory and practice of Bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2012), 131–155.

[40] Jeffrey Scott Vitter. 1982. Implementations for Coalesced Hashing. *Commun. ACM* 25, 12 (1982), 911–926.

[41] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM*.