Bloom Filter with a False Positive Free Zone

Sándor Z. Kiss*, Éva Hosszu†, János Tapolcai†, Lajos Rónyai‡, Ori Rottenstreich§
* Department of Algebra, Budapest University of Technology and Economics (BME), Hungary, kisspest@cs.elte.hu
† MTA-BME Future Internet Research Group, High-Speed Networks Laboratory (HSN*Lab*), {hosszu, tapolcai}@tmit.bme.hu
‡ Computer and Automation Research Institute Hungarian Academy of Sciences and BME, ronyai@sztaki.hu
§ Department of Computer Science, Princeton University, Princeton, USA, orir@cs.princeton.edu

***Abstract*—Bloom filters and their variants are widely used as space efficient probabilistic data structures for representing set systems and are very popular in networking applications. They support fast element insertion and deletion, along with membership queries with the drawback of false positives. Bloom filters can be designed to match the false positive rates that are acceptable for the application domain. However, in many applications a common engineering solution is to set the false positive rate very small, and ignore the existence of the very unlikely false positive answers. This paper is devoted to close the gap between the two design concepts of *unlikely* and *not having* false positives. We propose a data structure, called EGH filter, that supports the Bloom filter operations and besides it can guarantee false positive free operations for a finite universe and a restricted number of elements stored in the filter. We refer to the limited universe and filter size as the false positive free zone of the filter. We describe necessary conditions for the false positive free zone of a filter and generalize the filter to support listing of the elements. We evaluate the performance of the filter in comparison with the traditional Bloom filters. Our data structure is based on recently developed combinatorial group testing techniques.**

I. Introduction

Bloom filter [1] and its variants [2]–[6] are widely used data structures allowing for an approximate representation of a set S to answer membership queries of the form: is an element x in S? Their immense popularity is due to enabling highly versatile and seemingly endless application opportunities for membership testing, along with a nice trade-off among running time, space, error probability and implementation complexity. Their many computer and networking applications include caching, filtering, monitoring, data synchronization [7]–[12].

A traditional *Bloom filter (BF)* is a binary array of length m used to represent a set S, offering **insertions** and **queries**, both of which are carried out by setting/checking only a small number k of the m bits, where $k \ll m$ [1]. The BF is initialized with all bits set to zero. It has k hash functions, all of which hash elements uniformly and independently in the range $\{1, \ldots, m\}$. In an insertion of an element x, the hash values $h_1(x), h_2(x), \ldots, h_k(x)$ are computed and the corresponding bits are set to 1. If a bit is already set to 1 then it must remain set. Querying whether an element y is in S is carried out by computing the hash values $h_1(y), h_2(y), \ldots, h_k(y)$ and checking if they are all set to 1. If so, then the query returns that $y \in S$, otherwise it returns $y \notin S$. The functionality can be extended to support **deletions** by trading the bits for appropriately sized counters in a variant called the *Counting Bloom Filter (CBF)* [2]. By incorporating extra KEYSUM and VALUESUM fields to accompany each counter, a scheme named the *Invertible Bloom Lookup Table (IBLT)* [13] allows for **listing the items** through looking for entries with a single element and extracting them one by one.

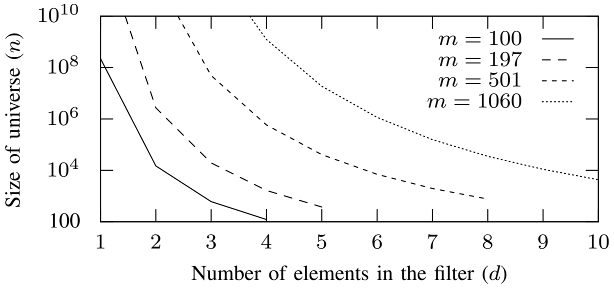


Fig. 1: The boundaries of the false positive free zone (FPFZ, below the curves) of the EGH filter depending on the size of the universe n and number of elements in the filter d. Data structure size is m bits.

By their very nature Bloom filters may give a *false positive answer* to a query operation, becoming probabilistic in this sense. A false positive occurs when all the hash values $h_1(y), h_2(y), \ldots, h_k(y)$ for some element y are set to 1 due to some other elements, even though y itself has not been previously inserted. Generally speaking, when tuning a Bloom filter one estimates the number of items n to be stored in the filter and chooses an appropriately low false positive probability p. Given these the number of hash functions k can be computed and more importantly, the required filter length m. While storing a fixed number of elements, increasing the filter length reduces the possible false positive probability obtained for the corresponding optimal number of hash functions.

In practice, focusing on its great space savings and easy computation, the *very small false positive probability* of the Bloom filter is often ignored and simply regarded as *none*, making the Bloom filter a *practically* false positive free structure. However, it is only *almost* false positive free, and false positive can occur and might cause difficulties in the application. With that motivation we explore the idea: could we **define some conditions, under which the filter is guaranteed to avoid false positives**?

Generally, Bloom filters can cope with a finite or infinite

The work is partially supported by the Hungarian Scientific Research Fund (grant No. OTKA K124171 and K115288).

universe through using hash functions that map elements to positions in the range $\{1, \ldots, m\}$. Clearly, a strict requirement to avoid false positives must restrict the universe to be finite (a limited size memory cannot distinguish between infinite subsets of elements of an infinite universe). Moreover, the possibility to satisfy this requirement is affected by the number of elements being held in the filter. For simplicity the universe is restricted to $U = \{1, \ldots, n_d\}$ for the case when false positives are guaranteed to be avoided until at most d elements are in the filter. In other words, if at most d elements from $\{1, \ldots, n_d\}$ are inserted in the filter we can be sure there are no false positives for queries of elements from $\{1, \ldots, n_d\}$. Various values of d allow different maximal universe size n_d. We refer to it as the **false positive free zone** of the filter (see also Fig. 1). Note that d is assumed to be a small number, e.g. $O(\log n)$.

In this paper we describe necessary conditions for the false positive free zone of a filter, and propose an alternative hash-based scheme for Bloom filters which can guarantee a false positive free zone. The main idea is to show the analogy between the BF and the widely studied problem of *non-adaptive Combinatorial Group Testing (CGT)*, where the goal is to identify up to d defective elements among a given range of items $\{1, \ldots, n_d\}$ through as few group tests as possible. Our hash function alternatives require less computational cost than traditional hash functions, as they are just a simple modulo division by a prime number. We call the resulting data structure the *EGH filter (*or shortly *EGHF)*, as it is an adaptation of the combinatorial group testing method described by Eppstein, Goodrich and Hirschberg [14]. First we investigate the basic version of the filter, which supports insertions and queries only, and we focus later on the more general Counting Bloom Filters that can also delete elements. We propose a fast algorithm for listing the elements in the false positive free zone through a more advanced construction. It is based on some advanced algebraic computations and runs in $O(\text{poly}(d \log(n_d)))$ steps[1], where d is the number of elements in the represented set. Its main idea is to define a system of equations where the roots will be the elements in the filter. The equations are the residues of elementary symmetric polynomials, and the roots can be found with the Bisection method and the Sturm chain. Finally, we evaluate the false positive free zone of the EGH filters of practical sizes. A space and running-time analysis is provided to measure the performance of the EGH filter compared to a traditional BF.

The rest of the paper is organized as follows. Section VI details use cases focusing on networking applications. Next, Section III defines the model of the work. Then in Sections IV and V we propose the solutions that have a false positive free zone. Section II overviews related work. In Section VIII we evaluate the performance of the proposed constructions and finally Section IX concludes the paper.

II. Related Work

A. Background

This paper focuses on a data structure that supports probabilistic membership testing, similar to Bloom filters, and has a false positive free zone with a restriction on the number of elements in the filter. In order to describe the novelty let us define the two widely investigated problems our data structure jointly solves. First, Bloom filters consider the following problem.

PROBABILISTIC MEMBERSHIP(p, m, k): Given a set S which is a subset of a (finite or infinite) universe U, design a data structure on m bits such that membership queries of the form "$x \in S$" can be answered using k bitprobes with the probability of false answers p.

Second, static membership testing is a deterministic data structure on a finite number of elements in the universe. This subproblem we are facing in the false positive free zone.

STATIC MEMBERSHIP(d, n, m, k): Given a set S with at most d elements, where S is a subset of a finite universe $U = \{1, \ldots, n\}$, design a data structure on m bits such that membership queries of the form "$x \in S$" can be answered using k bitprobes without giving false answers if $|S| \leq d$.

Adapting the notation of prior work [16]–[18], a (d, n, m, k)-*scheme* is a storage scheme that stores any d elements of an n-bit-sized universe using m bits such that membership queries can be answered using k probes. Such a scheme can be either adaptive or non-adaptive, depending on whether during the execution of a query the results of previous bit probes can be taken into account or not while determining the later probes, respectively. In this work we consider non-adaptive schemes. For an arbitrary deterministic non-adaptive scheme we denote the minimum space m needed for a (d, n, m, k)-scheme to exist by $m(d, n, k)$, where false positives are not allowed.

B. Previous Results in Probabilistic Membership Problem

First let us mention randomized schemes dealing with the static membership problem. A number of papers consider this problem [19], [20], for a survey we refer the reader to [18].

Bloom filters and their variants [1]–[6] are by far the most popular data structures allowing an approximate representation of S. In Bloom filters to achieve an optimal false positive rate p the number of hash functions k is proportional to $\log \frac{1}{p}$. In [21] Bloom filters were improved to make k a constant number independent of p.

Other solutions that use hashing for the static membership problem have been proposed, including hash compaction [22], cuckoo hashing [23] and multiset-representation [21].

The functions used by the EGH filter were previously investigated in [24] for a fundamentally different goal of reducing the computation time of the hash functions at lookup.

The functionality of a Bloom filter can be extended to support **deletions** by trading the bits for appropriately sized

[1] Throughout this paper log denotes logarithm of base 2.

counters [2], called Counting Bloom Filter (CBF). By incorporating extra cells to accompany each counter one can also achieve **listing of the items** [13].

C. Previous Results in Static Membership Problem

The related solutions based on the above characteristics are: In recent years a lot of work has been focused on the special cases when either d or k is small. The capabilities of very few bit probes are explored in [25] and [26]. A summary of most of these results can be found in the survey [18].

There exist a number of deterministic schemes solving the static membership problem. The most famous is the Fredman-Komlós-Szemerédi scheme [27], that can perform queries in a clearly optimal $O(1)$ time in the word-RAM model. However, it requires $O(n)$ space, that can be much larger than $O(d^2 \log n)$ for small d.

In general, this design problem is also often called *combinatorial group testing* (CGT) in the literature [28]. The idea of group testing dates back to World War II when millions of blood samples were analyzed to detect syphilis in US military. In order to reduce the number of tests it was suggested to pool the blood samples. The problem is called *non adaptive* CGT if the probing is performed simultaneously without knowing the result of other tests. The goal is to identify defective items among a given set of items through as few tests as possible. The special case $d = 1$ is called a *separating system* [29]. The problem to find exactly d defectives is to design *d-separable* matrices [28]. A dual notion in combinatorics is called *d-cover-free families* [30], [31], *superimposed codes* or ZFD_r codes [32]. Finding up to d items is related to the design of *d-disjunct* matrices [28].

III. Problem Definition: Identifying Elements through Group Testing

In this paper we deal with two variants of functionality: the *basic EGH filter* should support **insert** and **query**; while the *advanced EGH filter* should support **insert, query, delete** and **list**.

Definition 1: The data structure **filter** can store a set of elements of the universe U in a binary array of m bits, where a set of functions $h_i : U \to \{1, \ldots, m\}$ for $i = 1, \ldots, k$ are used to represent each element x.

Inserting an element $x \in U$ in a filter S means setting the bits at positions $h_1(x), h_2(x), \ldots, h_k(x)$ to one.

Querying whether an element y is in S means returning $y \in S$ if bits at positions $h_1(y), h_2(y), \ldots, h_k(y)$ are all set to 1, otherwise returning $y \notin S$.

The *code* of the element x is an m bit long binary vector with ones only in positions $h_i(x)$ for $i = 1, \ldots, k$. We say that a code of element y is *contained in the filter* if the filter has bit 1 at positions $h_i(y)$ for $i = 1, \ldots, k$. Filters can provide $O(1)$ lookup-per-operation complexity in the bit-probe model. In the traditional Bloom filter the functions h_is are pseudo-random hash functions. In the EGH filter having a false positive-free zone we replace $\{h_1, h_2, \ldots, h_k\}$ with functions $\{\hat{h}_1, \hat{h}_2, \ldots, \hat{h}_k\}$ such that there is no false positive in the membership testing for a given finite universe $U_d = \{1, \ldots, n_d\}$ as long as the number of elements stored in the filter is not greater than a pre-defined threshold d. Formally:

Definition 2: The **false positive free zone** of a filter allows a universe of size n_d for $d = 1, \ldots, d_{max}$, if for any filter $S \subseteq U_d$ and $|S| \leq d$ the query operator of an element $y \in U_d$ always returns the true answer, where $U_d = \{1, \ldots, n_d\}$.

For simplicity we refer to n_d as n. For a filter with n elements in the universe we define a *code matrix* M. It is an $m \times n$ binary matrix, where each column corresponds to a code of an element in the universe. The binary array of the filter S is going to be the Boolean sum (bitwise OR) of the columns of M corresponding to the elements of S. A false positive occurs when the Boolean sum of d columns contains another column. Such a case has to be avoided.

This problem was widely investigated in the context of **non-adaptive Combinatorial Group Testing (CGT)**. The primary goal of a CGT construction is to identify up to d defective elements among a given set through as few group tests as possible. Formally,

Given: a finite universe $U = \{1, \ldots, n\}$ and a (positive integer) maximum number of defective elements d.

Find: an $m \times n$ binary matrix M, where the union or Boolean sum (or bitwise OR) of any up to d columns does not contain any other column.

Note that, in the matrix M the rows correspond to the group tests and the columns to the elements. An entry of the matrix indexed (i, j) is equal to 1 if the i^{th} test contains the j^{th} element, and 0 otherwise. Such matrices are called d-disjunct matrices, and they are sufficient to unambiguously identify all d faulty elements and constitute the basis for non-adaptive combinatorial search algorithms and binary d-superimposed codes. In other words, to avoid false positives when having at most d elements in the EGH filter, we need to ensure that the code matrix is d-disjunct[2]. Formally we have the following.

Claim 1: A necessary and sufficient condition to avoid false positives in a filter having at most d elements from the universe $\{1, \ldots, n\}$ is that the corresponding $m \times n$ code matrix is d-disjunct.

Namely false positive-free operations require the codes assigned to each element to be d-disjunct non-adaptive CGT codes. Ruszinkó [33] gave a lower bound on the size of the d-disjunct matrices which can be applied to our scenario. Later it was improved by Füredi [34].

Claim 2: For any false positive free filter

$$m(d, n) \geq 0.25 \frac{d^2}{\log(d)} \log(n) \ , \tag{1}$$

where $m(d, n)$ denotes the space m needed for n elements in the false positive free zone and at most d elements in the filter.

[2] Note that there is a weaker CGT construction, called d-separable, where the bitwise OR of up to arbitrary d codes are to be distinct from each other. Note that distinct codes are not enough to avoid false positives, but we also need the property that the codes do not contain each other.

IV. Basic EGH Filter with False Positive Free Zone

A. Data Structure Construction

The proposed EGH filter data structure is based on the combinatorial group testing method described by Eppstein *et al.* [14, Section 2]. The essence of their solution is to use the Chinese Remainder Theorem [35] and solve a CGT problem by finding a solution to a system of linear congruences.

Let U be the set of the integers in the interval $[1, \ldots, n]$. Let d be the maximal number of inserted elements for which the false positive free zone is guaranteed. A number of k first primes are selected $\{p_1 = 2, p_2 = 3, \ldots, p_k\}$ (e.g., by the sieve of Eratosthenes), such that their product P is at least n^d, i.e.,

$$n^d \leq P = \prod_{i=1}^{k} p_i \ , \tag{2}$$

while their sum

$$m = \sum_{i=1}^{k} p_i \ ,$$

denotes the length of the codes. In the EGH filter the simple functions $\hat{h}_i$ for $i = 1, \ldots, k$ are defined as

$$\hat{h}_i(x) = x \pmod{p_i} + \sum_{j=1}^{i-1} p_j \ . \tag{3}$$

Note that the code consists of k blocks, where the i^{th} block has p_i bits all zero except for one position, which is $x \pmod{p_i}$ for an element x. In other words, the code is a radix block representation of the remainders after division with p_i (an example appears in Section IV-B). The codes generated by the construction were proved to be d-disjunct, meaning that the bitwise OR of any up to d codes does not contain any other code. In order to better understand the solution, we present the proof for that property with our terminology and notations. First we summarize the well known Chinese Remainder Theorem [35]. Let $p_1, \ldots, p_k$ be pairwise coprime integers and $a_1, \ldots, a_k$ be arbitrary integers. The theorem states that the following system of simultaneous congruences

$$x \equiv a_i \pmod{p_i}, \qquad i \in \{1, \ldots, k\} \tag{4}$$

has a unique solution for x modulo $P = \prod_{i=1}^{k} p_i$. The solution can be found through the following method [36]. For each $1 \leq i \leq k$ the integers p_i and $\prod_{j \neq i} p_j$ are necessarily coprime. In the first step for each $1 \leq i \leq k$, the modular multiplicative inverse of $\prod_{j \neq i} p_j$ modulo p_i is found. Namely, for each $1 \leq i \leq k$ the following congruences are solved:

$$q_i \cdot \prod_{j \neq i} p_j \equiv 1 \pmod{p_i}.$$

By using the extended Euclidean algorithm the integers r_i and q_i satisfying $r_i \cdot p_i = 1 + q_i \cdot \prod_{j \neq i} p_j$ can be found.

Then, choosing $e_i = q_i \prod_{j \neq i} p_j$, x can be constructed as

$$x = \sum_{i=1}^{k} a_i e_i \pmod{P}, \tag{5}$$

which satisfies the congruences (4). Algorithm 1 provides a more formal description of this key method.

Algorithm 1: ChineseRemainder

```
Input: $p_1, \ldots, p_k$, and $a_1, \ldots, a_k$
begin
    for $i = 1$ to $k$ do
        $N_i = \prod_{j \neq i} p_j$
        Find the modular multiplicative inverse:
            $q_i = N_i^{-1} \pmod{p_i}$
    return $x = \sum_{i=1}^{k} a_i q_i N_i \pmod{p_1 p_2 \cdots p_k}$.
```

The following lemma shows the correctness of the above construction.

Lemma 1: The EGH filter has a false positive free zone with at most d elements in the filter for universe $U = \{1, \ldots, n_d = n\}$ if

$$n \leq \sqrt[d]{\prod_{j=1}^{k} p_j} \ , \tag{6}$$

which can be written as

$$d \leq \frac{\log \prod_{j=1}^{k} p_j}{\log n} = \frac{\sum_{j=1}^{k} \log p_j}{\log n} \ . \tag{7}$$

See the proof in Appendix A.

We consider the space and time requirements of the EGH filter. We can rely on a result from [14], showing that for given d and n, the inequality of (7) can be satisfied with $\sum_{j=1}^{k} p_j = O(d^2 \log n)$ and $p_k \leq \lceil 2d \log(n) \rceil$. Note that EGH filter memory size is given by the sum of prime values. Namely, to have a false positive-free zone over n elements in the universe and maximum d elements in the filter, we have $m(d, n) = O(d^2 \log n)$. We can also evaluate the required time.

Corollary 1: The computation time of constructing an EGH filter with a false positive zone over n elements in the universe and maximum d elements in the filter is $O(d \log(n))$.

Proof: To construct an EGH filter we need to find a set of primes (or prime powers), for which $\prod_{i=1}^{k} p_i \geq n^d$ holds, where p_i denotes the i^{th} prime number (or prime power) and k is the number of primes found. This can be done by generating the sequence of primes till $\prod_{i=1}^{k} p_i \geq n^d$ holds. The fastest implementation of prime number sieves requires $O(p_k)$ operations [37], which leads to $O(d \log(n))$ operations in total. ■

B. Illustrative example of EGH Filter

Now let us construct an EGH Filter which has a false positive free zone over a universe of size $n_2 = 48$ when at most $d = 2$ elements can be in the filter. First, a set of prime integers should be selected such that their product is at least $n^d = 48^2 = 2304$. Multiplying the first five primes 2, 3, 5, 7 and 11 we get $P = 2310$, which results in codes of length $2+3+5+7+11 = 28$ bits. We have five simple functions by Eq. (3), namely $\hat{h}_1(x) = x \mod 2$, $\hat{h}_2(x) = x \pmod 3 + 2$,

$\hat{h}_3(x) = x \pmod 5 + 5$, $\hat{h}_4(x) = x \pmod 7 + 10$ and $\hat{h}_5(x) = x \pmod{11} + 17$.

With the use of the above codes of 28 bits, the allowed universe size is determined by the number of allowed elements d. While for $d = 2$ we explained that the size is $n_2 = 48$, it increases to $n_1 = 2310$ for $d = 1$ and decreases to $n_3 = 13$ for $d = 3$, as $(n_1)^1 = 2310, (n_3)^3 = 2197 \leq 2310$. Note that for $d = 3$, the EGH representation is not efficient since it stores subsets of 13 elements in 28 bits. Instead we could assign a bit dedicated for each of the 13 elements, that would result in a trivial 13 bit long false positive free representation for any d. We can also calculate the false positive rate when more than d elements are stored in the set, as studied in the next subsection. For instance, with $n_2 = 48$ for $d = 2$, the false positive rate over universe $\{1, \ldots, 48\}$ while keeping $3 > d$ elements in the filter would be 0.55%.

C. False Positives Outside the False Positive Free Zone

The false positive rate of the Bloom filter is

$$P_{false}^{BF} = \left(1 - \left(1 - \frac{1}{m}\right)^{kd'}\right)^k \approx \left(1 - e^{-kd'/m}\right)^k \quad , \quad (8)$$

which is minimal if $k \approx \frac{m}{d'} \ln 2$, where d' is the number of inserted elements.

The probability of false positives for infinite universe and arbitrary number of elements in the filter satisfies

$$P_{false}^{EGH} = \prod_{i=1}^{k} \left(1 - \left(1 - \frac{1}{p_i}\right)^{d'}\right) \quad . \quad (9)$$

V. Advanced EGH Filters with False Positive Free Zone

The EGH filter data structure can be easily extended to support deletions by using an array of counters (rather than bits), of $\lceil \log d \rceil$ bits each, as in the Counting Bloom Filter (CBF) [2]. This makes the EGH structure take $O(d^2 \log n \log d)$ space, where again Eq. (7) holds. In this variant, inserting an element x is done by incrementing the counters $\hat{h}_i(x)$ by 1 for $i = 1, \ldots, k$. Deletion of an item y that had previously been inserted, is carried out by decrementing the corresponding counters by 1.

As for listing, the problem is much more challenging. In CGT the obvious solution is to iterate through the universe $\{1, \ldots, n_d = n\}$ and perform membership testing for each entry. In our case we intend to have an algorithm that runs in $O(\text{poly}(d \log(n)))$ steps for listing the elements, where d is the number of elements in the filter. The main idea is to define a system of equations where the roots are the elements in modular arithmetic. The equations are then solved to obtain the list of elements. This requires algebraic computations described in the rest of this section.

Before we explain our approach for general d, let us first explain the special cases of $d = 1$ and $d = 2$.

A. Algorithms for Listing $d = 1, 2$ Elements in the EGH Filter

The situation for $d = 1$ is simple, because Algorithm 1 (The Chinese Remainder) solves the problem based on the remainders of the single element for each of the primes.

For $d = 2$ let $y_{i,1}$ and $y_{i,2}$ be the remainders for the prime p_i for $i \in [1, k]$. The task is to compute two integers x_1, x_2 resulting in these remainders. The method is based on the fact that the Chinese remainders provide a ring homomorphism. In other words, the operations $+, -, \times$ can be swapped with forming remainders. More precisely, let x_1, x_2, satisfying $x_1 \pmod{p_i} = y_{i,1}$ and $x_2 \pmod{p_i} = y_{i,2}$, be two elements in the filter. Then the remainder of $x_1 + x_2 \pmod{p_i}$ is $y_{i,1} + y_{i,2} \pmod{p_i}$. A similar argument is valid for $x_1 \times x_2$ and for $x_1 - x_2$.

As a result $(y_{i,1} - y_{i,2})^2 \pmod{p_i}$ is congruent to $z := (x_1 - x_2)^2 \pmod{p_i}$. Even if we swap $y_{i,1}$ and $y_{i,2}$ we get the same value of z after squaring. In other words, this symmetric function is invariant for swapping the remainders of x_1 and x_2. On the other hand, z can be obtained by solving the corresponding set of congruences with the Chinese Remainder Theorem, and we know it is a square number because x_1 and x_2 are both in $[1, n]$, thus their difference cannot be more than n, hence $z \leq n^2 \leq P = p_1 \cdot p_2$.

Next we need to find the square root of integer z in modular arithmetic. This can be done with Newton-iteration or binary search for large n. Let u be the positive square root of z, and assume that $x_1 \geq x_2$. Let u_i be the remainder of u modulo p_i. Please observe that in advance we know the remainders $\{y_{i,1}, y_{i,2}\}$ only as a set, and cannot associate one of the numbers with a specific remainder. The equation $u_i = y_{i,1} - y_{i,2}$ helps us to find that association, and hence identify x_1 and x_2 by Algorithm 1. It is clear from the properties of the congruences that $x_1 - x_2 \equiv u_i \pmod{p_i}$ for $1 \leq i \leq k$. On the other hand it is clear that $x_1 + x_2 \equiv y_{i,1} + y_{i,2} \pmod{p_i}$ for $1 \leq i \leq k$. We solve this system of congruences by applying Algorithm 1. As x_1 and x_2 are both in $[1, n]$, we get that both $x_1 - x_2$ and $x_1 + x_2$ are at most $2n < n^2$, thus we have an equality in our congruences.

B. An Illustrative Example of the Algorithm for Listing two Elements in the EGH Filter

To illustrate how this idea works for $d = 2$ we give an example. Assume that we have $n = 14$ items and we would like to describe a set of two of them (x_1 and x_2). Our task is to identify these items. To do this we have to choose coprime integers say $p_1 = 2$, $p_2 = 3$, $p_3 = 5$ and $p_4 = 7$, which clearly satisfy $P = 210 > 196 = n^2$. The remainders are $y_{1,1} = 0$, $y_{2,1} = 0$, $y_{3,1} = 1$, $y_{4,1} = 6$ and $y_{1,2} = 0$, $y_{2,2} = 1$, $y_{3,2} = 4$, $y_{4,2} = 4$. The values of z are $0, 1, 4, 4$, which solved by using the Chinese Remainder Theorem (Algorithm 1) we obtain that $z \equiv 4 \pmod{210}$ thus $u = 2$. Obviously $u_1 = 0, u_2 = 2, u_3 = 2, u_4 = 2$, thus by (Algorithm 1) we get that $x_1 - x_2 \equiv 2 \pmod{210}$. Similarly we get that $x_1 + x_2 \equiv 10 \pmod{210}$. As $210 > n^2 = 196$ it follows that $x_1 - x_2 = 2$ and $x_1 + x_2 = 10$. Solving this system of linear equations we obtain that $x_1 = 6$ and $x_2 = 4$ as desired.

C. Algorithm to List d Elements in the EGH Filter

In this subsection we explain how to define for a general d, a system of equations whose roots are the elements of the filter. Then we provide an approach to solve the system for obtaining the list of elements. We strongly on the theory of elementary symmetric polynomials. We use the following facts about polynomials [38], that if we have a polynomial $p(z)$, where α_i denotes its coefficients and x_is are the roots of $p(z)$, i.e.,

$$p(z) = z^d + \ldots + \alpha_{d-1}z + \alpha_d = (z - x_1)\ldots(z - x_d), \quad (10)$$

then we have

$$\alpha_i = (-1)^i \sigma_i(x_1, \ldots, x_d), \quad (11)$$

where $\sigma_i(x_1, \ldots, x_d)$ for $(1 \leq i \leq d)$ is called the i^{th} elementary symmetric polynomial of $x_1, \ldots, x_d$ and can be computed by using Algorithm 2 [39]. The obtained elementary symmetric polynomial for $1 \leq m \leq d$ is

$$\sigma_m(x_1, \ldots, x_d) = \sum_{1 \leq j_1 < j_2 < \ldots < j_m \leq d} x_{j_1} \cdot \ldots \cdot x_{j_m}.$$

Algorithm 2: ELEMENTARYSYMMETRICPOLYNOMIALS

```
Input: $x_1, \ldots, x_d$
Result: $\sigma_1(x_1, \ldots, x_d), \ldots, \sigma_d(x_1, \ldots, x_d)$
begin
    $\sigma_1^{(1)} := 1$, for $i = 1, \ldots d - 1$
    $\sigma_j^{(i)} = 0$, for all $j > i$
    $\sigma_2^{(1)} = x_1$
    for $i = 2$ to $d$ do
        for $j = 1$ to $i$ do
            $\sigma_j^{(i)} = \sigma_j^{(i-1)} + x_i \sigma_{j-1}^{(i-1)}$
```

For example for $d = 3$ we have

$$\sigma_1(x_1, x_2, x_3) = x_1 + x_2 + x_3 \;, \quad (12)$$

$$\sigma_2(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2x_3 \;, \quad (13)$$

$$\sigma_3(x_1, x_2, x_3) = x_1x_2x_3 \;. \quad (14)$$

Next we explain how to define for a general d, a system of equations in modular arithmetic whose roots are the elements in the filter. Let $p_1, \ldots, p_k$ be pairwise coprime prime powers, i.e., $p_i = q_i^{\alpha_i}$, where q_i is a prime and $\alpha_i \geq 0$ is an integer. We choose the p_is such that Eq. (7) holds, i.e. $n^d \leq P = \prod_{i=1}^k p_i$. Let $y_{i,1}, \ldots, y_{i,d}$ be the remainders modulo p_i of the $d \leq n$ elements $x_1, \ldots, x_d$ from S. The task is to find numbers $x_1, \ldots, x_d$ which satisfy the following systems of congruences:

$$x_1 \equiv y_{i,1} \pmod{p_i}, \;\ldots,\; x_d \equiv y_{i,d} \pmod{p_i}$$

for all $1 \leq i \leq k$.

Note that in the advanced filter we have counters instead of bits, thus we know how many elements lie in each residue class. It follows that with Algorithm 2 we can calculate the residues of the elementary symmetric polynomials of $x_1, \ldots, x_d$ modulo all the p_is. For $j = 1, \ldots, d$ let $\sigma_j(x_1, \ldots, x_d)$ denote the j^{th} elementary symmetric polynomial of $x_1, \ldots, x_d$. It follows from the properties of the congruences that the following holds

$$\begin{aligned} \sigma_j(x_1, \ldots, x_d) &\equiv \sigma_j(y_{1,1}, \ldots, y_{1,d}) \pmod{p_1} \;, \\ &\vdots \\ \sigma_j(x_1, \ldots, x_d) &\equiv \sigma_j(y_{k,1}, \ldots, y_{k,d}) \pmod{p_k} \;, \end{aligned}$$

for every $j = 1, \ldots, d$. Note that on the right hand side we have constants. We define

$$a_i^{(j)} \equiv \sigma_j(y_{i,1}, \ldots, y_{i,d}) \pmod{p_i} \quad (15)$$

so that we can substitute it to have the following $d \times k$ system of equations

$$\begin{aligned} &\sigma_1(x_1, \ldots, x_d) \equiv a_1^{(1)} \pmod{p_1}, \ldots, \sigma_1(x_1, \ldots, x_d) \equiv a_k^{(1)} \pmod{p_k}, \\ &\vdots \\ &\sigma_d(x_1, \ldots, x_d) \equiv a_1^{(d)} \pmod{p_1}, \ldots, \sigma_d(x_1, \ldots, x_d) \equiv a_k^{(d)} \pmod{p_k}. \end{aligned}$$

We can run Algorithm 1, the Chinese Remainder Theorem for each row of the above equations to obtain

$$A_j := \textsc{ChineseRemainder}(a_1^{(j)}, \ldots, a_k^{(j)}, \quad p_1, \ldots, p_k) \;, \quad (16)$$

for $j = 1, \ldots d$. Next we have the following system of equations

$$\begin{aligned} \sigma_1(x_1, \ldots, x_d) &\equiv A_1 \pmod{P} \;, \\ &\vdots \\ \sigma_d(x_1, \ldots, x_d) &\equiv A_d \pmod{P} \;. \end{aligned}$$

It is clear from the definition of $\sigma_j(x_1, \ldots, x_d)$ that

$$\begin{aligned} \sigma_j(x_1, \ldots, x_d) < \binom{d}{j} n^j = \binom{d}{d-j} n^j < d^{d-j} n^j \\ < n^{d-j} n^j = n^d. \quad (17) \end{aligned}$$

It follows that the following congruences hold without $\pmod{P}$, because $P \geq n^d$ and $\sigma_j(x_1, \ldots, x_d) \leq n^d$ for $j = 1, \ldots d$ i.e.,

$$\sigma_1(x_1, \ldots, x_d) = A_1, \ldots, \sigma_d(x_1, \ldots, x_d) = A_d.$$

Recall that according to Eq. (10) and (11) the roots of the polynomial

$$\begin{aligned} f(z) = z^d &- \sigma_1(x_1, \ldots, x_d) z^{d-1} \\ &+ \sigma_2(x_1, \ldots, x_d) z^{d-2} - \cdots + (-1)^d \sigma_d(x_1, \ldots, x_d) \end{aligned}$$

are actually $x_1, \ldots, x_d$. This means that in order to list the elements we need to find the roots of the polynomial

$$f(z) = z^d - A_1 z^{d-1} + A_2 z^{d-2} - \ldots. + (-1)^d A_d. \quad (18)$$

It can be done with standard mathematical algorithms, such as the root finder method of Heindel [40] based on the Bisection method [41] and the Sturm chain [38]. Roughly speaking, this technique first isolates the roots of the polynomial with the help of a theorem of Sturm and then finds them by the Bisection method. See the details in Appendix C.

To summarize the above we have the following algorithm. In the inner loop we compute $a_i^{(j)}$ by Eq. (15) for $i = 1, \ldots, k$ and $j = 1, \ldots, d$, which is the p_i remainder of the j^{th} elementary symmetric polynomials after substituting $y_{i,1},\dots,y_{i,d}$. In the outer loop this gives the A_js with the Chinese remaindering process as in Eq. (16). Then we build up our polynomial $f(z)$ and find its roots by using the ROOTFINDER method.

Algorithm 3: LISTEGHFILTER

```
Input: $y_{i,1},\dots,y_{i,d}$ for all $1 \le i \le k$, $p_1,\dots,p_k$
begin
    for $j = 1$ to $d$ do
        for $i = 1$ to $k$ do
            $a_i^{(j)} := \sigma_j(y_{i,1},\dots,y_{i,d}) \pmod{p_i}$
        $A_j$ := CHINESEREMAINDER($a_1^{(j)},\dots,a_k^{(j)},p_1,\dots,p_k$)
    Set $f(z) = z^d + \sum_{t=1}^{d}(-1)^t A_t z^{d-t}$
    Compute $(x_1,x_2,\dots,x_d)$ = ROOTFINDER($f(z),0,P$)
```

Theorem 1: The LISTEGHFILTER finds the elements stored in the EGH filter using $O(d^{10}\log^3 n)$ bit operations[3].

See the proof in Appendix B.

VI. Network Applications and Use Cases

A. Encoding of Flow Attributes in SDN Switches

A recent study [43] describes Software-Defined Networking (SDN) scenarios in which exact encoding of small sets is necessary to distinguish between classes of traffic with different required treatment. Each such traffic class is encoded as a unique attribute carrying tag in the packet header. A desired property is the ability to test whether the represented set includes some queried attribute.

They deal with three scenarios. The first corresponds to Internet Exchange Point (IXP), where multiple autonomous systems (ASes) exchange traffic and interdomain routing information. Here the tag encodes the set of advertising peers used in the forwarding decision. The second is related to service chaining, where the tag represents the set of middleboxes which must be traversed by the traffic flow. The third scenario is in the context of network policies where each traffic class is allowed to access different network resources.

In all of these three applications, false positives should be avoided, e.g., to avoid wrong forwarding of a packet, the appliance of a redundant network function or an illegal access to a resource. With the EGH filter, if the tag is for instance $m = 100$ bit long, with a variety of $n = 606$ pre-defined attributes (fixed universe), false positives can be fully avoided if each traffic class has at most $d = 3$ attributes.

B. Multicast Addressing

Another application for the EGH filter can be the in-packet Bloom filter [44]. It is a new forwarding mechanism developed for information centric networking, where Bloom filters are used to encode multicast trees in the packet header in a stateless manner. Placed in packet headers, the in-packet Bloom filters can effectively represent a set of node or link IDs along the expected path. Paths are often short. The study [45] overviews the forwarding anomalies caused by false positives, such as packets storms, forwarding loops and flow duplication.

In [45] the AS-level topology graph was considered for $m = 800, 1024$. It has $n \le 10^5$ links today (the universe is fixed), which can be in the FPFZ of an EGH filter for $d = 6, 7$.

C. Early Detection of Botnet Attacks

In early detection of botnet attacks the goal is to identify communication patterns as a sign of communication between the bots and the botnet controllers (called C&C servers) [46]. For example, a common technique is to hide C&C servers behind an hourly-changing domain name. Bots algorithmically generate and try to resolve a number of domains (with domain generation algorithms - DGA), only one of which is registered as the C&C server. Thus DGA behavior is characterized by many, often repeating, failed DNS queries at multiple DNS servers form the same IP address.

The application requires to succinctly store a set of suspicious IP addresses at each DNS server, which is sent periodically to each other, and list the items in the possible intersections of the sets. In this case the universe is the set of 32 bit long IP addresses (i.e. $n = 2^{32}$ and fixed universe), and because of the short monitoring period the number of newly infected IP addresses are typically small. A false positive answer in this case means wrong IP address is identified.

For example there are $i = 1000$ suspicious IP addresses in each monitoring period, which is $i \cdot 32$bit= 4KB to send as a blacklist, while to find the intersection of two lists has $O(i \log i)$ time complexity. On the other hand, an EGH filter of $m = 1161$ counters can detect up to $d = 4$ infected items, with constant time element insertion in the filter, and intersection has $O(i)$ time complexity.

VII. Discussion

A. Flexibility

Compared to the traditional BF an important advantage of the EGH filter is that the function $\hat{h}_i$ is the same for any filter not depending on its length. In other words the EGH filter has a block structure; and for a longer EGH filter we need to add more blocks, while keeping the previous blocks as is. This allows a great flexibility, because the size of the Bloom filter can be dynamically changed without recomputing the filter. To reduce the length we just need to erase the last blocks.

B. Implementation Issues

Another advantage of the EGH filter in comparison with Bloom filters is the reduced hash computation cost. Typically the hash functions are either computationally intensive (like the cryptographic hash functions such as MD5) or have good randomness (e.g., CRC32, FNV, BKDR). The randomness is important to have small a false positive rate. In EGH, we need to perform only a simple modulo operation. Moreover, the same function $\hat{h}_i$ in a EGH filter is used if $i \le k$. This means, the functions $\{\hat{h}_1, \hat{h}_2, \dots, \hat{h}_k\}$ can be hardware implemented, or in assembly for software implementations. The EGH filter size then defines the number of functions we need to use.

[3] Theorem 3.1 in [42] provides a faster but more complex algorithm than the Sturm chain that finds the items at the Boolean cost $\tilde{O}(d^3 \log^3 n)$.

TABLE I: The size of the false positive free zone n_d of the EGH filter with up to d elements for different memory size (of m bits). The filter makes use of k primes, such that p_k is the last of them.

k	p_k	m	FPFZ
1	2	2	$n_1 = 2$
2	3	5	$n_1 = 6$
3	5	10	$n_1 = 30$
4	7	17	$n_1 = 210$
5	11	28	$n_1 = 2310$ $n_2 = 48$
6	13	41	$n_1 = 30000$ $n_2 = 173$
7	17	58	$n_1 = 511000$ $n_2 = 714$ $n_3 = 79$
8	19	77	$n_1 = 9.7{\cdot}10^6$ $n_2 = 3110$ $n_3 = 213$
9	23	100	$n_1 = 2.2{\cdot}10^8$ $n_2 = 14900$ $n_3 = 606$ $n_4 = 122$
10	29	129	$n_1 = 6.5{\cdot}10^9$ $n_2 = 80400$ $n_3 = 1860$ $n_4 = 283$

p_k	m	FPFZ
$p_{11} = 31$	160	$n_1 = 2.01{\cdot}10^{11}$ $n_2 = 448000$ $n_3 = 5850$ $n_4 = 669$ $n_5 = 182$
$p_{12} = 37$	197	$n_1 = 7.42{\cdot}10^{12}$ $n_2 = 2.72{\cdot}10^6$ $n_3 = 19500$ $n_4 = 1650$ $n_5 = 375$
$p_{13} = 41$	238	$n_1 = 3.04{\cdot}10^{14}$ $n_2 = 1.74{\cdot}10^7$ $n_3 = 67300$ $n_4 = 4180$ $n_5 = 788$ $n_6 = 259$
$p_{14} = 43$	281	$n_1 = 1.31{\cdot}10^{16}$ $n_2 = 1.14{\cdot}10^8$ $n_3 = 236000$ $n_4 = 10700$ $n_5 = 1670$ $n_6 = 485$

p_k	m	FPFZ
$p_{15} = 47$	328	$n_1 = 6.15{\cdot}10^{17}$ $n_2 = 7.84{\cdot}10^8$ $n_3 = 850000$ $n_4 = 28000$ $n_5 = 3610$ $n_6 = 922$ $n_7 = 348$
$p_{16} = 53$	381	$n_1 = 3.26{\cdot}10^{19}$ $n_2 = 5.71{\cdot}10^9$ $n_3 = 3.19{\cdot}10^6$ $n_4 = 75600$ $n_5 = 7990$ $n_6 = 1790$ $n_7 = 613$
$p_{17} = 59$	440	$n_1 = 1.92{\cdot}10^{21}$ $n_2 = 4.38{\cdot}10^{10}$ $n_3 = 1.24{\cdot}10^7$ $n_4 = 209000$ $n_5 = 18100$ $n_6 = 3530$ $n_7 = 1100$ $n_8 = 458$

VIII. Numerical Evaluation

We perform experiments to examine the performance of the EGH filter under different scenarios. We compare it with existing Bloom-filter based solutions and focus on the amount of memory, the universe size, the obtained probability for false positives (if exist) and the number of memory accesses.

First, we evaluate the universe size that allows the false positive free zone (FPFZ) of the EGH filter for different number of stored items d. We implemented the Eppstein-Goodrich-Hirschberg (EGH) filter, as described in Section IV. The sequence of prime numbers (i.e. $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, etc.) is generated via the sieve of Eratosthenes. To have a FPFZ of a given d and n_d we add prime numbers until $\prod_{i=1}^{k} p_i \geq (n_d)^d$ holds. This gives us an EGH filter length of $m = \sum_{i=1}^{k} p_i$ bits where k is the number of prime numbers.

In Table I, we describe, for various values of d, the universe size n_d that allows keeping up to d elements from the universe without false positives for $m \leq 440$. For example, an EGH filter of length $m = 440$ has a false positive free zone for universe of $\{1, \ldots, n_5 = 18100\}$ with at most $d = 5$ elements in the filter. To perform a membership query, we need to test 17 positions in the filter, as the number of (first) primes that sum up to 440. The number of bits increases in a logarithmic fashion for a fixed d as a function of the order of magnitude of the universe size.

Next we compared the EGH filter the with Bloom filter (BF). To illustrate the benefits of the EGH filter we computed the false positive probability of the Bloom filter with the same size m and same number of hash functions k as in the EGH filter with the FPFZ. The results are shown in Table II. The false positive probability of the BF is not negligible, especially for small d, where EGH in the FPFZ is guaranteed to avoid false positives. We also added the minimum value of false positives of the BF obtained when an optimal number of hash functions are used.

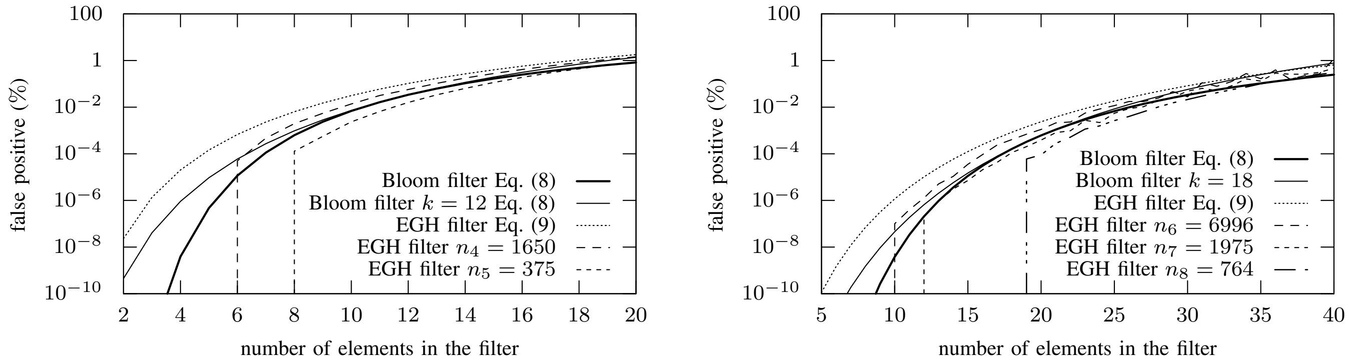
TABLE II: The false positive probability p of the Bloom filter for the same size m and the number of bit lookups k and the number of elements in the filter d as the EGH filter allowing a universe size of at least $n_d = 100, 200, 500$. The last two columns corresponds to the Bloom filter with optimal number of hash functions to achieve minimal false positives.

input			EGH	BF	Optimal BF	
d	m	k	n_d	p	k_{OPT}	p
1	17	4	209	.00215	12	.000364
2	41	6	173	.00028	15	.00006
3	77	8	213	.000028	18	.000004
4	100	9	122	.000022	18	.000006
5	160	11	182	$1.30{\cdot}10^{-6}$	23	$2.22{\cdot}10^{-7}$
10	440	17	134	$4.03{\cdot}10^{-9}$	31	$6.77{\cdot}10^{-10}$
20	1264	27	104	$4.13{\cdot}10^{-13}$	44	$6.58{\cdot}10^{-14}$
2	58	7	714	.000022	21	.000001
3	77	8	213	.000028	18	.000004
4	129	10	283	$1.88{\cdot}10^{-6}$	23	$1.19{\cdot}10^{-7}$
5	197	12	375	$1.10{\cdot}10^{-7}$	28	$6.33{\cdot}10^{-9}$
10	501	18	202	$4.39{\cdot}10^{-10}$	35	$3.61{\cdot}10^{-11}$
20	1593	30	211	$8.03{\cdot}10^{-16}$	56	$2.44{\cdot}10^{-17}$
1	28	5	2310	.000127	20	.0000018
3	100	9	606	$2.41{\cdot}10^{-6}$	24	$1.21{\cdot}10^{-7}$
4	160	11	669	$1.60{\cdot}10^{-7}$	28	$4.79{\cdot}10^{-9}$
5	238	13	788	$8.50{\cdot}10^{-9}$	33	$1.23{\cdot}10^{-10}$
10	712	21	726	$3.61{\cdot}10^{-13}$	50	$1.43{\cdot}10^{-15}$
20	2127	34	562	ε	74	ε

Fig. 2 shows the false positive rate of EGH and Bloom filters for two filter lengths $m = 197$ and $m = 501$ bits. The solid curves are the false positive rate of the BF computed by Eq. (8) for optimal number of hash functions, and for the same number as for the EGH filter. The dotted curve shows the false positive rate outside the false positive free zone (FPFZ), computed by Eq. (9). The false positive is slightly larger for EGH outside of the FPFZ, especially for small d, which is the price we pay to have a FPFZ. We also measured the false positives for a fixed universe for different values of d. We did it by generating 10^9 filters with d elements and selected a random element not in the filter. For each such instance, we tested if the filter gives a false positive or not. Recall the false positive rate is guaranteed to be zero in the FPFZ; however, surprisingly a larger zone was actually free of false positives. For example in the $m = 501$ bit long filter the universe $U = \{1, \ldots, n_6 = 6996\}$ has a FPFZ till $d = 6$, while in 10^9 randomly generated queries, there were no false positives when d was at most 9. In general the false positive rate increases in a similar way for EGH and in BF as more and more elements are inserted to the filter. On the charts we can also see how the false positive free zone depends on the size of the universe. It is because smaller n and larger d can also meet the same bound of $\prod_{i=1}^{k} p_i \geq n^d$.

IX. Conclusion

In this paper we described the EGH filter for the representation of sets, while avoiding false positives when constraints on the universe size and the represented set size hold. The proposed approach is an adaptation of a known non-adaptive combinatorial group testing scheme. The used functions are deterministic, fast and simple to calculate, enabling a superior lookup performance compared to Bloom filters. We also extended the model through the use of counters, supporting deletions and efficient listing of the elements. The fast listing of the elements is performed by finding the roots of a system



(a) Filter length $m = 197$. In the EGH filter the primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 and 37.

(b) Filter length $m = 501$. In the EGH filter the primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59 and 61.

Fig. 2: Beyond the false positive free zone: The false positive rate of the EGH and the Bloom filters for various filter length m, and size of universe n.

of equations in modulo arithmetic. Our approach is based on traditional number theoretical techniques such as the Chinese Remainder Theorem, the Bisection Method and the Sturm chain. Through numerical evaluations we demonstrated the performance of the new approach.

References

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

[2] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[3] M. Mitzenmacher, "Compressed Bloom filters," *IEEE/ACM Trans. on Networking*, vol. 10, no. 5, pp. 604–612, 2002.

[4] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended Bloom filter: an aid to network processing," *ACM SIGCOMM CCR*, vol. 35, no. 4, pp. 181–192, 2005.

[5] D. Guo, J. Wu, H. Chen, X. Luo *et al.*, "Theory and network applications of dynamic Bloom filters." in *IEEE INFOCOM*, 2006.

[6] S. Xiong, Y. Yao, Q. Cao, and T. He, "kBF: a Bloom filter for key-value storage with an application on approximate state machines," in *IEEE INFOCOM*, 2014.

[7] W.-C. Feng, K. G. Shin, D. D. Kandlur, and D. Saha, "The blue active queue management algorithms," *IEEE/ACM Trans. on Networking*, vol. 10, no. 4, pp. 513–528, 2002.

[8] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[9] F. Hao, M. Kodialam, T. Lakshman, and H. Song, "Fast dynamic multiple-set membership testing using combinatorial Bloom filters," *IEEE/ACM Trans. on Networking*, vol. 20, no. 1, pp. 295–304, 2012.

[10] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[11] B. Donnet, B. Gueye, and M. A. Kaafar, "Path similarity evaluation using Bloom filters," *Computer Networks*, vol. 56, no. 2, pp. 858–869, 2012.

[12] T. Chen, D. Guo, Y. He, H. Chen, X. Liu, and X. Luo, "A Bloom filters based dissemination protocol in wireless sensor networks," *Ad Hoc Networks*, vol. 11, no. 4, pp. 1359–1371, 2013.

[13] M. T. Goodrich and M. Mitzenmacher, "Invertible Bloom lookup tables," in *IEEE Allerton Conference on Communication, Control, and Computing*, 2011.

[14] D. Eppstein, M. T. Goodrich, and D. S. Hirschberg, "Improved combinatorial group testing algorithms for real-world problem sizes," *SIAM Journal on Computing*, vol. 36, no. 5, pp. 1360–1375, 2007.

[15] "Bloom filter with a false positive free zone (complete version)," 2017. [Online]. Available: https://www.dropbox.com/s/ipejqixq27qg2rp/avoid-false-positives-full.pdf?dl=0

[16] A. Brodnik and J. I. Munro, "Membership in constant time and almost-minimum space," *SIAM Journal on Computing*, vol. 28, no. 5, pp. 1627–1640, 1999.

[17] J. Radhakrishnan, S. Shah, and S. Shannigrahi, "Data structures for storing small sets in the bitprobe model," in *Springer ESA*, 2010.

[18] P. K. Nicholson, V. Raman, and S. S. Rao, "A survey of data structures in the bitprobe model," in *Space-Efficient Data Structures, Streams, and Algorithms*. Springer, 2013.

[19] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh, "Are bitvectors optimal?" *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1723–1744, 2002.

[20] A. Ta-Shma, "Storing information with extractors," *Information Processing Letters*, vol. 83, no. 5, pp. 267–274, 2002.

[21] A. Pagh, R. Pagh, and S. S. Rao, "An optimal Bloom filter replacement," in *ACM-SIAM symposium on Discrete algorithms*, 2005.

[22] U. Stern and D. L. Dill, "A new scheme for memory-efficient probabilistic verification," in *Springer Formal Description Techniques IX*, 1996, pp. 333–348.

[23] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Elsevier Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[24] J. Lu, T. Yang, Y. Wang, H. Dai, L. Jin, H. Song, and B. Liu, "One-hashing Bloom filter," in *IEEE IWQoS*, 2015.

[25] N. Alon and U. Feige, "On the power of two, three and four probes," in *ACM-SIAM Symposium on Discrete Algorithms*, 2009.

[26] M. Garg and J. Radhakrishnan, "Set membership with a few bit probes," in *ACM-SIAM Symposium on Discrete Algorithms*, 2015.

[27] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with $o(1)$ worst case access time," *Journal of the ACM*, vol. 31, no. 3, pp. 538–544, 1984.

[28] D. Z. Du and F. Hwang, *Combinatorial group testing and its applications*. World Scientific, 1993.

[29] G. Katona, "On separating systems of a finite set," *Journal of Combinatorial Theory*, vol. 1, no. 2, pp. 174–194, 1966.

[30] P. Erdős, P. Frankl, and Z. Furedi, "Families of finite sets in which no set is covered by the union of others," *Israel Journal of Mathematics*, vol. 51, no. 1-2, pp. 79–89, 1985.

[31] M. Ruszinkó, "On the upper bound of the size of the r-cover-free families," in *IEEE ISIT*, 1993.

[32] W. Kautz and R. Singleton, "Nonrandom binary superimposed codes," *IEEE Trans. on Information Theory*, vol. 10, no. 4, pp. 363–377, 1964.

[33] M. Ruszinkó, "On the upper bound of the size of the r-cover-free families," *Journal of Combinatorial Theory, Series A*, vol. 66, no. 2, pp. 302–310, 1994.

[34] Z. Füredi, "On r-cover-free families," *Journal of Combinatorial Theory, Series A*, vol. 73, no. 1, pp. 172–173, 1996.

[35] C. Ding, D. Pei, and A. Salomaa, *Chinese Remainder Theorem: applications in computing, coding, cryptography*. World Scientific Publishing Co., Inc., 1996.

[36] K. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory (2nd ed.)*. Cambridge, MA: Springer-Verlag, 1990.

[37] P. Pritchard, "Fast compact prime number sieves (among others)," *Journal of algorithms*, vol. 4, no. 4, pp. 332–344, 1983.

[38] V. V. Prasolov and D. Leites, *Polynomials.* Springer, 2004, vol. 11.

[39] H. Jiang, S. Graillat, and R. Barrio, "Accurate and fast evaluation of elementary symmetric functions," in *IEEE Symposium on Computer Arithmetic*, 2013.

[40] L. E. Heindel, "Integer arithmetic algorithms for polynomial real zero determination," *Journal of the ACM*, vol. 18, no. 4, pp. 533–548, 1971.

[41] B. Faires, *Numerical Analysis 6th.* Brooks/Cole Pub., 1997.

[42] I. Z. Emiris, V. Y. Pan, and E. P. Tsigaridas, "Algebraic algorithms," *Computer Science Technical Reports, Paper 361*, 2012.

[43] R. MacDavid, R. Birkner, O. Rottenstreich, A. Gupta, N. Feamster, and J. Rexford, "Concise encoding of flow attributes in SDN switches," in *ACM Symposium on SDN Research (SOSR)*, 2017.

[44] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "Lipsin: line speed publish/subscribe inter-networking," in *ACM SIGCOMM CCR*, vol. 39, no. 4, 2009, pp. 195–206.

[45] M. Sarela, C. E. Rothenberg, T. Aura, A. Zahemszky, P. Nikander, and J. Ott, "Forwarding anomalies in Bloom filter-based multicast," in *IEEE INFOCOM*, 2011.

[46] Z. Abaid, M. A. Kaafar, and S. Jha, "Early detection of in-the-wild botnet attacks by exploiting network communication uniformity: An empirical study," in *Proc. IFIP Networking*, 2017.

[47] G. Tenenbaum, *Introduction to Analytic and Probabilistic Number Theory*, 1st ed. Cambridge University Press, 1995.

[48] E. Bach and J. O. Shallit, *Algorithmic Number Theory: Efficient Algorithms.* Cambridge, MA: MIT press, 1996, vol. 1.

[49] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography.* CRC Press, Inc., 1996.

[50] H. Iwaniec and E. Kowalski, *Analytic number theory.* American Mathematical Society Providence, 2004, vol. 53.

[51] G. E. Collins and R. Loos, "Polynomial real root isolation by differentiation," in *Proc. of the third ACM symposium on Symbolic and algebraic computation.* ACM, 1976, pp. 15–25.

Appendix

A. *Proof of Lemma 1*

The EGH filter has a false positive free zone with at most d elements in the filter for universe $U = \{1, \ldots, n_d = n\}$ if

$$n \leq \sqrt[d]{\prod_{j=1}^{k} p_j} \quad . \tag{19}$$

Proof 1: Recall that the EGH filter consists of k blocks, each of them assigned to a prime p_j, where $j = 1, \ldots, k$. For all k blocks the bit that corresponds to the remainder of $x \pmod{p_j}$ is set to 1 in the EGH filter. The proof is indirect and we assume there is a set of codes S belonging to no more than d elements and the EGH filter composed of the bitwise OR of the corresponding codes has bit 1 for the remainders of $x \pmod{p_j}$ for every prime $j = 1, \ldots, k$.

For items $x, y \in U$ let us define the function $P(x, y)$ as follows:

$$P(x, y) = \prod_{j=1,\ldots,k \mid x \equiv y \mod p_j} p_j \quad .$$

In other words $P(x, y)$ is the product of all the generator primes p_j in which x and y cannot be distinguished, as both have the same remainder. Intuitively, $P(x, y)$ shows the similarity between the codes of x and y. By the Chinese Remainder Theorem we have $x \equiv y \mod P(x, y)$.

Assume by contradiction that the EGH filter wrongly indicates on the membership of an element $x \notin S$. Note that every 1 bit of x is covered and thus every p_j appears at least once in one of these products $P(x, y)$ for some $y \in S$, and because of Eq. (2) we have

$$\prod_{y \in S} P(x, y) \geq \prod_{j=1}^{k} p_j \geq n^d \quad .$$

Moreover, there are at most d elements in S leading to the fact

$$\max_{y \in S} P(x, y) \geq \sqrt[d]{n^d} = n \quad .$$

Therefore, there exists at least one element in the EGH filter (denoted as y') for which $P(x, y') \geq n$. By construction, y' is congruent to the same values to which x is congruent modulo each of the p_j's in $P(x, y')$. By the Chinese Remainder Theorem, the solution to these common congruences is unique modulo the least common multiple of these p_j's, which is $P(x, y')$ itself, since the p_j's are relatively prime to each other. Therefore, x must be equal to y' modulo a number that is at least n, and since both x and y' are positive integers $\leq n$. Thus we have $x = y'$; which contradicts the fact that $x \notin S$.

B. *Proof of Theorem 1*

The ListEghFilter finds the elements stored in the EGH filter using $O(d^{10} \log^3 n)$ bit operations[4].

Proof 2:

[4] Theorem 3.1 in [42] provides a faster but more complex algorithm than the Sturm chain that finds the items at the Boolean cost $\widetilde{O}(d^3 \log^3 n)$.

The ListEghFilter algorithm contains three steps. In the first step, it computes the elementary symmetric polynomials, in the second step it uses the Chinese Remainder Theorem, and finally it determines the roots of the corresponding polynomial.

In Algorithm 2 we compute the elementary symmetric polynomials recursively. To get the r^{th} elementary symmetric polynomial one needs $r-1$ additions and $r-1$ multiplications in Algorithm 2, thus we can compute all elementary symmetric polynomials by using $1 + \ldots + (d-1)$ additions and multiplications. As $x_1, \ldots, x_d < n$, one addition needs $O(\log n)$ bit operations, and one multiplication requires $O(\log^2 n)$ bit operations, thus the total cost of Algorithm 2 is $O(d^2 \log^2 n)$ bit operations.

In this paragraph we analyze the Chinese remaindering process (Algorithm 1). It is well known [49] that Chinese remaindering requires $O(\log^2 P)$ bit operations. As noted before the p_is are some power of primes, i.e., $p_i = q_i^{\alpha_i}$, where q_i is a prime and $\alpha_i \geq 1$. Since $p_i \leq n$, it follows that $q_i^{\alpha_i} \leq n$, thus we have $\alpha_i \leq \log_2 n$ ($\log_2 n$ denotes the logarithm of n to the base 2). In view of this observation it is easy to see [47] that if we select the first k primes for $q_1, \ldots, q_k$ then

$$\log P = \sum_{i=1}^{k} \log p_i = \sum_{i=1}^{k} \alpha_i \log q_i \leq \log_2 n \sum_{i=1}^{k} \log q_i$$
$$< (\log n)\pi(q_k) \log q_k = k(\log n)(\log q_k), \quad (20)$$

where $\pi(x)$ denotes the number of primes up to x. It is well known [50] that the k^{th} prime number is $O(k \log k)$, thus we have $\log P = O((\log n)k(\log k + \log\log k)) = O((\log n)k \log k)$. We know from [14] that

$$k = O\Big(\frac{d \log n}{\log(2d \log n)}\Big),$$

which implies $k \log k = O(d \log n)$. It follows that the total cost is $O(d^2 \log^4 n)$. Since the number of systems of congruences is d, computing the A_js in the ListEghFilter needs $O(d^3 \log^4 n)$ bit operations.

In the last step we have to determine the roots of the polynomial $f(z)$. For a polynomial $f(z) = a_d z^d + \ldots + a_1 z + a_0$ let $K = \sum_{i=0}^{d} |a_i|$, which is called the 1 - norm of the polynomial $f(z)$.

It is clear that all coefficients of our polynomial are at most n^d, which implies that $K \leq dn^d$. It follows from [51] that the running time of Heindel's algorithm is $O(d^{10} + d^7 \log^3 K)$ (see Theorem 8. and the remark after that and see also in [51]). We have to use the Bisection method at most $d-1$ times, which requires $O(d \log n)$ operations, because the length of each interval is at most n. Thus the total cost to determine all roots requires at most $O(d^{10} + d^{10} \log^3 n + d \log n) = O(d^{10} \log^3 n)$ bit operations. This implies that the total cost of the ListEghFilter algorithm is $O(d^2 \log^2 n + d^2 \log^4 n + d^3 \log^4 n + d^{10} \log^3 n) = O(d^{10} \log^4 n)$ bit operations.

C. The Root Finder Algorithm

In this section we give a short survey about the main tools we make use of. First we describe the well known Bisection method [41], which is useful to find a root of a polynomial (with some required level of accuracy). Let f be a polynomial for the real variable z and consider the equation $f(z) = 0$. Let $[a, b]$ be an interval and assume that $f(a)$ and $f(b)$ have opposite signs. Since f is continuous on $[a, b]$, the Intermediate Value Theorem implies that there exists an $r \in [a, b]$, such that $f(r) = 0$. At each step we divide the interval into two smaller intervals by computing the midpoint $c = (a + b)/2$ of the interval and the value $f(c)$. If $f(c) = 0$ then c is a root of f. Otherwise either $f(a)$ and $f(c)$ have opposite signs and $[a, c]$ contains a root, or $f(c)$ and $f(b)$ have opposite signs and $[c, b]$ contains a root. We select the subinterval which contains a root as the new interval to be used in the next step. Note that the size of the interval that contains a root of f is reduced by half at each step. The above process is continued until the interval is sufficiently small, smaller than some pre-defined threshold ε. Pseudo-code is given in Algorithm 4.

Algorithm 4: BisectionMethod

```
Input: Polynomial $f$, range $[a, b]$, accuracy $\varepsilon$
begin
    Repeat the following process while $\frac{b-a}{2} > \varepsilon$ do
        Compute $c = \frac{a+b}{2}$ and $d = f(c) \cdot f(a)$
        if $d < 0$ then
            $b := c$
        if $d > 0$ then
            $a := c$
        if $d = 0$ then
            break
return $c$
```

We also need some facts about Sturm sequences [38]. By applying the Euclidean algorithm to compute the greatest common divisor of $f_0(z) = f(z)$ and $f_1(z) = f^{'}(z)$ it is easy to obtain a Sturm sequence. In particular, by taking successively the remainders with polynomial division and change their signs. See also Algorithm 5 for the pseudo-code. The algorithm terminates, because the degree sequence of $f_i^{'}$s is decreasing. Finally, the obtained sequence of polynomials have the following property

$$\begin{aligned}
f_0(z) &= q_1(z)f_1(z) - f_2(z),\\
f_1(z) &= q_2(z)f_2(z) - f_3(z),\\
&\vdots\\
f_{m-2}(z) &= q_{m-1}(z)f_{m-1}(z) - f_m(z),\\
f_{m-1}(z) &= q_m(z)f_m(z).
\end{aligned}$$

where $q_{i+1}(z)$ is the quotient of the polynomial long division of $f_i(z)$ by $f_{i+1}(z)$.

Let $f_0(z) = f(z)$, $f_1(z), \ldots, f_m(z)$ be a Sturm sequence, where $f(z)$ has no repeated roots, and let $\omega(z)$ denote the number of sign changes (ignoring zeroes) in the sequence $f(z), f_1(z), \ldots, f_m(z)$. Sturm's theorem states that if $f(a) \neq 0$ and $f(b) \neq 0$, then the number of distinct roots of f in the interval $(a, b]$ is equal to $\omega(a) - \omega(b)$.

Algorithm 5: SturmSequence

```
Input: A polynomial $f(z)$ with integral coefficients
Result: A Sturm sequence of polinomials
        $f_0(z), f_1(z), \ldots, f_m(z)$
begin
    $f_0(z) = f(z)$
    $f_1(z) = f^{'}(z)$
    for $i = 2$ to $m$ do
        $f_i(z) :=$ the remainder of the polynomial long division
        of $f_{i-2}(z)$ by $f_{i-1}(z)$ .
```

If there is given a polynomial with integral coefficients and with no repeated integral roots which lie in the interval $[a, b]$, then we can find all the roots of this polynomial by combining the Bisection method and Sturm theorem in the following way. Consider the intervals $[a, (a + b)/2]$ and $[(a + b)/2, b]$. Apply Sturm's Theorem and calculate the number of roots in both intervals. Choose one in which there is at least one root, e.g., $[a, (a+b)/2]$ and divide it into two equal parts by taking its halve point, and calculate the number of roots in both of these smaller intervals. Repeat these steps until one the subintervals will contain only one root. We can find this root by the Bisection method. Dividing the original polynomial by the corresponding linear polynomial. We can repeat this process and obtain all the roots of the original polynomial. More formally we have the following algorithm.

Algorithm 6: RootFinder

```
  Input: polynomial $f(z)$ with integral coefficients, range $[a, b]$
  begin
1     $f_0(z), \ldots, f_m(z) :=$ SturmSequence($f(z)$)
2     Calculate $\omega(a) - \omega(b)$.
3     if $\omega(a) - \omega(b) = 0$ then
          There is no root in $(a, b)$
4     if $\omega(a) - \omega(b) = 1$ then
          Find the only root by the BisectionMethod
5     if $\omega(a) - \omega(b) > 1$ then
          Calculate $c = \frac{a+b}{2}$ and goto step 2 and repeat this
          process to the intervals $(a, c)$ and $(c, b)$
```