Bloom Filter with a False Positive Free Zone

Sándor Z. Kiss, Éva Hosszu, János Tapolcai, Lajos Rónyai and Ori Rottenstreich

***Abstract*—Bloom filters and their variants are widely used as space-efficient probabilistic data structures for representing sets 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. We then generalize the filter to support the listing of the elements through the use of counters rather than bits. We detail networking applications of the filter and discuss potential generalizations. We evaluate the performance of the filter in comparison with the traditional Bloom filters. We also evaluate the price in terms of memory that needs to be paid to guarantee real false positive-free operations for having a deterministic Bloom filter-like behavior. Our data structure is based on recently developed combinatorial group testing techniques.**

***Index Terms*—Computer networks, Data Structures, Software defined networking.**

Sándor Z. Kiss is with Department of Algebra, Budapest University of Technology and Economics (BME), Hungary (email: kisspest@cs.elte.hu). Éva Hosszu and János Tapolcai are with MTA-BME Future Internet Research Group, Faculty of Electrical Engineering and Informatics (VIK), BME, Hungary (emails: {hosszu, tapolcai}@tmit.bme.hu). Lajos Rónyai is with the Institute of Computer Science and Control, Eötvös Loránd Research Network, and BME (email: ronyai@sztaki.hu). Ori Rottenstreich is with the Department of Computer Science and the Viterbi Department of Electrical Engineering, Technion, Haifa, Israel (e-mail: or@technion.ac.il).

Part of this work has been supported by the Hungarian Scientific ResearchFund (grant No. OTKA K115288, K129335, K128062, FK134604, K124171), and the Hungarian Ministry of Innovation and the National Research, Development and Innovation Office within the framework of the Artificial Intelligence National Laboratory Programme. The research reported in this paper and carried out at the BME was supported by the "TKP2020, Institutional Excellence Program" of the National Research Development and Innovation Office in the field of Artificial Intelligence (BME IE-MI-SC TKP2020). The research leading to these results was partially supported by the High Speed Networks Laboratory (HSNLab). S.Z.K. was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences, by the ÚNKP-18-4 New National Excellence Program of the Ministry of Human Capacities, by the ÚNKP-19-4 New National Excellence Program of the Ministry for Innovation and Technology, and by the ÚNKP-20-5 New National Excellence Program of the Ministry for Innovation and Technology from the source of the national research, development and innovation fund. This work of was also partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate, by the Alon fellowship, by German-Israeli Science Foundation (GIF) Young Scientists Program, by the Taub Family Foundation as well as by the Polak Fund for Applied Research, at the Technion.

I. Introduction

Bloom filter [1] and its variants [2]–[8] 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 and 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 [9]–[14].

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 $\{1, \dots, m\}$. In an insertion of an element x, the hash values $h_1(x), h_2(x), \dots, h_k(x)$ 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 $h_1(y), h_2(y), \dots, h_k(y)$ 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)* [15] allows for **listing the items** through looking for entries with a single element and extracting them one by one.

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 $h_1(y), h_2(y), \dots, h_k(y)$ 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 $\boldsymbol{p}$. Given these, the number of hash functions k can be computed, and more importantly, the required filter length m can be determined. 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

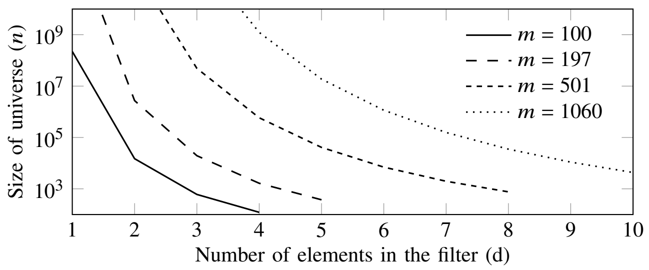


Fig. 1: The boundaries of the false positive free zone (FPFZ, below the curves) of the EGH filter depending on the size of the universe n and number of elements in the filter d. Data structure size is m bits.

application. With that motivation, we explore the idea: what if we would like to have an actually false positive free structure? Does it really require just a little extra space (and implementation overhead) over a BF with a very small false positive probability, or much more? 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 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 finite memory cannot distinguish between infinitely many elements). 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 $\{1, \ldots, n_d\}$. Different 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 how a false positive free data structure could be devised while having a similarly simple construction and the same lookup-per-operation performance in the bit-probe model, assuming that the universe $U := \{1, \ldots, n\}$ from which elements are taken is finite. 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. As for hash functions, we use the traditional division method. It is based on just division with remainder by a prime number and is known to have modest computational cost. 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 [16]. 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 allow deleting elements. By maintaining counters in a modified construction of the filter, we propose a fast algorithm for listing the elements in the false-positive free zone. 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' coefficients are obtained as residues of elementary symmetric polynomials, and the roots can be found with the Bisection Method and the Sturm Sequence. Finally, we evaluate the false positive free zone of the EGH filters of practical sizes. Space and running-time analysis are provided to measure the EGH filter's performance compared to a traditional BF. Furthermore, we show the large marginal cost overhead that has to be paid in a transition from filters with a small false positive probability to filters guaranteed to have no false positives.

The rest of the paper is organized as follows. Section II details use cases focusing on networking applications. Section III overviews related work. Next, Section IV describes the model. Then, in Sections V and VI we propose the solutions that have a false positive free zone. In Section VII, we evaluate the performance of the proposed constructions. Section VIII discusses flexibility and implementation. Finally Section IX concludes the paper. For completeness the Appendix overviews existing methods used in implementing the solutions.

II. Network Applications and Use Cases

This section discusses four application scenarios to illustrate the price in terms of memory needs to be paid to guarantee real false positive-free operations for having a similarly simple Bloom filter-like construction. Note that EGH of [16] implements the best known (deterministic) CGT construction.

A. Discovering Neighboring Cells in Radio Access Networks

In a Radio Access Network, the field is covered by base stations and moving User Equipment (UE). Each UE connects to one or multiple base stations. Two base stations are defined *overlapping* if they have at least some minimum number of H common UEs. When executing handover processes, each base station (cell) should maintain the list of User Equipment (UE), which overlaps with other cells to which such handovers may be made. The problem is called the neighboring cell list (NCL), where the goal is to continuously monitor the overlapping base station pairs in a decentralized way with minimal messages between them. Typically there are $100 - 1000$ base stations and roughly $10K - 100K$ moving UEs. On average, each base station is connected to ≈ 100 moving UEs, and H can equal $H = 10$, for instance.

In a naive approach, every base station sends its list of UEs to every other base station. Note that the management overlay, where the messages are sent, is a spanning tree. To reduce the protocol overhead, the study [17] suggests implementing the process in two steps: Step 1 gives a fast probabilistic guess on which cells may be neighboring, while in Step 2 only these candidate neighboring base stations pairs exchange their list of UEs to compute the intersections correctly.

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

In Step 1, the spanning tree established by the management overlay is also used to aggregate the messages containing the list of UEs. Here Bloom Filters are used because they have bounded size and allow efficient computation of set unions and intersections. More precisely, instead of sending the list of UEs between every base station pairs, each base station that is a leaf of the spanning tree sends a Bloom Filter containing its list of UEs to its adjacent node in the spanning tree. If a base station is an internal node of the spanning tree, it computes all incoming Bloom Filters' union and forwards the union Bloom filter as an aggregated monitoring messages to their neighbors in the spanning tree. This is repeated until every base station receives information about every other base station through these aggregated monitoring messages.

In Step 2, each node takes the received Bloom filter and computes the intersection with the list of UEs it is connected to. It gives an upper-bound on the number of common UEs. It is only an upper-bound because Bloom Filters are probabilistic, and there might be false-positives. If the obtained upper bound is at least H, the list of UEs is exchanged with all the base stations that inserted UEs in the Bloom filter.

Note that we can use an EGH filter instead of a Bloom filter in the monitoring message, thus Step 2 might be omitted for many base station pairs where the received EGH filter is in the false-positive free zone. The EU ID is 16-bit long; thus, the size of the universe is $n_d = 2^{16} = 65536$, and for $d = 100$ users, the required filter length is $m = 10^5$, implying a message of $\approx$ 10KB, which was medium message size in an early UMTS system. Note that, because of the block structure of the EGH filter, their size can be different, depending on the number of UEs, keeping EGH filter in the false-positive free zone. Furthermore, EGH filters can be aggregated in the same way as Bloom Filters, i.e., computing their Boolean OR. Note that the union of two EGH filters will have many more items (the size may double) and probably will not be in the false-positive free zone anymore. Note that, unlike regular Bloom filters, it is possible to compute the intersection of EGH filters of different lengths.

The key benefit is that if the leaf nodes send EGH filters in the false-positive free zone, Step 2 can be ignored between a leaf node and its neighbor. Note that, at intermediate nodes, the aggregated EGH filters provide a probabilistic data structure in the same way and performance as Boom filters. Therefore, by properly selecting the aggregation tree, many messages can be saved in Step 2.

B. Encoding of Flow Attributes in SDN Switches

A recent study [18] describes Software-Defined Networking (SDN) scenarios in which the exact encoding of small sets is necessary to distinguish between classes of traffic with the different required treatment. Each such traffic class is encoded as a unique attribute carrying tag in the packet header. The desired property is the ability to test whether the represented set includes some queried attributes.

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 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 prevent wrong forwarding of a packet, the appliance of a redundant network function or 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. When forwarding each packet, 9 bit-positions of the header need to be checked (even for 3 attributes). This is itself a smaller value than reading a binary representation of a single attribute ($10 = \lceil \log_2 606 \rceil$). If a new attribute or new traffic appears with more than d attributes, we need to increase the size of the filter with new blocks to guarantee no false positives as described in Sec. VIII-A.

C. Multicast Addressing

Another application for the EGH filter can be the in-packet Bloom filter [19]. 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. 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 [20] overviews the forwarding anomalies caused by false positives, such as packets storms, forwarding loops, and flow duplication.

In [20] the AS-level topology graph was considered for $m = 800, 1024$. It has $n \leq 10^5$ links today (fixed universe), which can be in the FPFZ of an EGH filter for $d = 6, 7$. It is important that forwarding each packet can be performed by examining only 22-25 bit-positions (even if 7 links are included in the filter). As a comparison, reading a binary representation of a single link involves $17 = \lceil \log_2 10^5 \rceil$ bits.

D. Distributed Storage

Suppose we store multiple copies of the same dataset so that each dataset is modified independently. Such a dataset can be a general indexed database or version control to manage the changes to documents, computer programs, large web sites, etc. It is necessary to identify with low communication overhead conflicts among the copies, which are the modified parts' intersection in the various copies.

To achieve such a reconciliation, each computer with a copy of the dataset can send an EGH filter to each other. The EGH filter contains the indices of the modified items, for example, the line numbers in version control. The intersection of the EGH filters describes the conflicted items, which require special care. In this case, the universe is usually fixed, for example, composed of the source code lines or regions in the code. A false positive answer in the identification can cause unnecessary extension of the process and redundant communication. Furthermore, with frequent synchronizations, the number of conflicted or even modified elements is typically small. Note that it is a typical application of IBLT [15]; however, unlike IBLT, the EGH filter cannot efficiently list the elements outside the FPFZ. If too many elements are inserted, the only way to list them is to test the EGH filter for every element of the universe.

E. 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) [21]. 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 succinctly storing 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 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 the wrong IP address is identified.

For example, assume that 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 the intersection has $O(i)$ time complexity.

III. 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$(\boldsymbol{p}, m, k)$: 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 $\boldsymbol{p}$.

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 $U = \{1, \dots, n\}$, 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.

Adopting the notation of prior work [22]–[24], 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 [25], [26], for a survey we refer the reader to [24].

Bloom filters and their variants [1]–[6], [27], [28] are by far the most popular data structures allowing an approximate representation of S. In Bloom filters to achieve an optimal false positive rate $\boldsymbol{p}$ the number of hash functions k is proportional to $\log \frac{1}{\boldsymbol{p}}$. In [29] Bloom filters were improved to make k a constant number independent of $\boldsymbol{p}$.

Other solutions that use hashing for the static membership problem have been proposed, including hash compaction [30], cuckoo hashing [31] and multiset-representation [29].

The functions used by the EGH filter were previously investigated in [32] 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 counters [2], called Counting Bloom Filter (CBF). By incorporating extra cells to accompany each counter, one can also achieve **listing of the items** [15].

C. Previous Results in Static Membership Problem

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 [33] and [34]. A summary of most of these results can be found in the survey [24].

There exist several deterministic schemes solving the static membership problem. The most famous is the Fredman-Komlós-Szemerédi scheme [35], which can perform queries in an optimal $O(1)$ time in the word-RAM model. However, it requires $O(n)$ space, which 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 [36]. The idea of group testing dates back to World War II when millions of blood samples were analyzed to detect syphilis in the US military. It was suggested to pool the blood samples to reduce the number of tests. 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* [37]. The problem to find exactly d defectives is to design *d-separable* matrices [36]. A dual notion in combinatorics is called *d-cover-free families* [38], [39], *superimposed codes* or ZFD_r codes [40]. Finding up to d items is related to the design of *d-disjunct* matrices [36]. Recent works described the Shifting Bloom Filter (ShBF), a data structure relying on the encoding of auxiliary information in set representation for allowing membership, association, and multiplicity queries [41]. For instance, auxiliary information can be used as an offset for the bits selected to be set in the filter. In another recent work, a technique to reduce the number of hash functions in Bloom filter constructions was proposed [42]. With some similarity to our approach, the method relies on computing a single hash value for an element. Then, bits in the filter are associated with the element based on computing the remainder in dividing the hash value modulo some prime numbers.

D. Previous Works on Reducing False Positives in Bloom Filters

Schemes have been suggested to improve the error-memory trade-off of the Bloom filter and the CBF. The regular CBF simply represents counters with a fixed number of bits per counter (typically four). More efficient representations have been suggested, benefitting from typically low counter values, not using the counters' most significant bits. The ML-HCBF (MultiLayer Hashed CBF) [43] relies on a hierarchical compression where more bits are assigned to the least significant bits of the counters. Similarly, the VL-CBF (Variable Length CBF) [44] relies on Huffman coding to describe counters with a variable number of bits. Another approach is to rely on more than a single Bloom filter to improve accuracy. Lim et al. [45] proposed for the case of a finite universe maintaining two Bloom filters, representing a set S and its complement S^c. Since Bloom filters have no false negatives, it is necessarily correct if only one of the filters provides a positive answer. If both provide a positive answer, both options should be examined. The Cross-checking Bloom filter [46] is an architecture for reducing false positives that includes a main filter and cross-checking part with two filters. Cross-checking is accessed to validate a positive answer to the main part. The two cross-checking filters are programmed each with a subset of two disjoint sets that together compose the represented set.

Other studies [47], [48] suggested reducing the false positives by allocating a different number of hash functions to elements based on the query popularity. While the approach can be helpful for a skewed query distribution, maintaining the used number of hash functions per element can be challenging. Another technique makes use of fingerprints [49], [50]. Upon element insertion, a fingerprint is stored in hash locations. This enables a careful counter examination counter upon a query.

While the above approaches to reduce false positives maintain the property of no false negatives, other schemes allow false negatives to achieve this reduction in false positives. The Retouched Bloom filter [51] does so by clearing some of the bits that have earlier been set. Approaches are suggested to select the bits to be cleared, such as selecting randomly or focusing on those that do not imply many false negatives. Likewise, the Generalized Bloom filter [52] maintains two groups of hash functions. Upon an element insertion, bits pointed by the first group are set, and those by the second are cleared. A query of an element requires matching some bits that have to be 0s and others 1s.

IV. Problem Definition: Identifying Elements through Group Testing

In this paper we deal with two functionality variants: the *basic EGH filter* should support **insert** and **query**; 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 $h_i : U \to \{1, \dots, m\}$ for $i = 1, \dots, k$ are used to represent each element x.

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

Querying whether an element y is in S means returning $y \in S$ if bits at positions $h_1(y), h_2(y), \dots, h_k(y)$ 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 $i = 1, \dots, k$. 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 $i = 1, \dots, k$. 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 $\{h_1, h_2, \dots, h_k\}$ with functions $\{\hat{h}_1, \hat{h}_2, \dots, \hat{h}_k\}$ such that there is no false positive in the membership testing for a given finite universe $U_d = \{1, \dots, n_d\}$ as long as the number of elements stored in the filter is at most a pre-defined threshold d. Formally:

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

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 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. This should 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 $U = \{1, \dots, n\}$ 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. 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 $\{1,\dots,n\}$ 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 [53] 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 [54].

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. The first asymptotically optimal d-disjunct matrix construction was given by Hwang and Sós [55], while the shortest real-world problem size non-adaptive CGT codes were developed by Eppstein, Goodrich and Hirschberg [16], which we utilize in the EGH filter.

V. 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, Goodrich, and Hirschberg [16, Section 2]. The essence of their solution is to use the Chinese Remainder Theorem [56] 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 $[1,\dots,n]$. Let d be the maximal number of inserted elements for which the false positive free zone is guaranteed. The first k primes are selected $\{p_1 = 2, p_2 = 3, \dots, p_k\}$ (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 $i = 1,\dots,k$ are defined as

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

[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.

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 V-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 [56]. Let $p_1,\dots,p_k$ be pairwise-coprime integers and $a_1,\dots,a_k$ be arbitrary integers. The theorem states that the following system of simultaneous congruences

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

has a unique solution for x modulo $P = \prod_{i=1}^{k} p_i$. The solution can be found through the following method [57]. 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 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: $p_1,\dots,p_k$, and $a_1,\dots,a_k$
begin
1   for $i = 1$ to $k$ do
2       $N_i = \prod_{j\neq i} p_j$
3       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 $U = \{1,\dots,n\}$ if

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

which can be written as

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

Proof [16]: Recall that the EGH filter consists of k blocks, each of them assigned to a prime p_j, where $j =$

$1, \dots, k$. For all k blocks, the bit that corresponds to the remainder of $x \pmod{p_j}$ is set to 1 in the EGH filter. 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 $j = 1, \dots, k$. For items $x, y \in U$ let us define the function $P(x, y)$ as follows:

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

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. 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

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

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

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

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 $\leq n$, we obtain $x = y'$ which contradicts the fact that $x \notin S$. ■

We consider the space and time requirements of the EGH filter. We can rely on a result from [16], 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 estimate the number of primes needed. This number implies the number of required memory accesses k upon element insertion of a query.

Claim 3: The number of bit access at membership testing is at most

$$k \leq \frac{\lfloor 2d \ln n \rfloor}{\ln \lfloor 2d \ln n \rfloor} \left(1 + \frac{1.2762}{\ln \lfloor 2d \ln n \rfloor} \right) + 1 \ , \tag{8}$$

for $n \geq 7$.

Proof: The number of bit accesses in the EGH filter equals to the number of primes such that Eq. (6) is met. Taking the logarithm of both sides of Eq. (6) we have

$$\ln n \leq \frac{1}{d} \sum_{j=1}^{k} \ln p_j \ .$$

Note that $\sum_{j=1}^{k} \ln p_j$ can be expressed with the Chebyshev function, $\theta(x) = \sum_{p_j \leq x} \ln p_j$, where $x = p_k$ is the k-th largest prime. Since are searching for the smallest k such that the above inequality holds, we have

$$d \ln n \geq \theta(p_{k-1}) \ .$$

Next, we use $\theta(x) \geq \frac{x}{2}$ for $x > 4$ [16]. Note that, since $n \geq 7$, the prime p_{k-1} is at least 5. Putting the above two together for the largest prime we have

$$p_{k-1} \leq \lfloor 2d \ln n \rfloor \ .$$

To complete the proof we apply the prime counting function, $\pi(x)$, which is the number of primes less than or equal to x. For $\pi(x)$ we use the bound of [58] that states for $x \geq 2$

$$\pi(x) < \frac{x}{\ln x} \left(1 + \frac{1.2762}{\ln x} \right) \ . \tag{9}$$

Finally we substitute $x = \lfloor 2d \ln n \rfloor$ to get the bound on $k - 1$. The above argument is a variant of Thm. 1 of [16]. ■

B. Illustrative example of EGH Filter

Now let us construct an EGH Filter that 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 $\hat{h}_5(x) = x \pmod{11} + 17$.

Table I shows the codes obtained, which are composed of 5 blocks. The matrix is of size 28×48 such that the columns refer to the $n_2 = 48$ elements $U = \{1, \dots, 48\}$ and each column describes the $2 + 3 + 5 + 7 + 11 = 28$ bits for each element with a single set bit in each of the five blocks, with the lengths of the primes $p_1 = 2, p_2 = 3, p_3 = 5, p_4 = 7$ and $p_5 = 11$, respectively. A block of length p_i refers to the hash values $0, 1, \dots, p_i - 1$. The first (leftmost) column refers to the element $x = 1$. For $i \in [1, 5]$ we have $x \pmod{p_i} = 1$, thus along this column the first bit is set in each of the blocks. Similarly, the last (rightmost) column corresponds to the element $x = 48$ implying $x \pmod 2 = 0, x \pmod 3 = 0, x \pmod 5 = 3, x \pmod 7 = 6, x \pmod{11} = 4$, that results in a bit-vector 10 100 00010 0000001 00001000000. Accordingly, for this column in the five blocks, the bits with a value of one are the first, first, fourth, seventh, and fifth. More generally, for $U = \{1, \dots, n\}$ the matrix is of size $\left(\sum_{j=1}^{k} p_j \right) \times n$ such that column i describes the values of i modulo each of the primes

TABLE I: Example of a 2-disjunct matrix with 48 columns and 28 rows. The column vectors are the codes of the elements. 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 $\hat{h}_5(x) = x \pmod{11} + 17$.

	$n_2 = 48$																																																
item ID:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	p_1
	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	
	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	$p_2 = 3$
	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	
	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	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	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	$p_3 = 5$
	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	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	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	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	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	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	
	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	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	
	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	$p_4 = 7$
	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	
	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	
	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	
	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	
	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	
	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	
	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	$p_5 = 11$
	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	
	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	
	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	
	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	
	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	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	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	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	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	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	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	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	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	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	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	

$p_1, \ldots, p_k$ through k blocks with a single bit set in each of them.

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 $(n_1)^1 = 2310, (n_3)^3 = 2197 \leq 2310$.

C. *False Positives Outside the False Positive Free Zone for a Large Universe*

So far in the paper, we showed that there would be no false positives if the universe is the set of integers between $1, \ldots, n$, and there are at most d elements in the filter. In this section, we will investigate what is the false probability if these conditions do not hold. To have an analogy to Bloom filters and give simple formulas we will assume a situation when we are way out of this false positive free zone, namely the universe is $1, \ldots, n' = \prod_{j=1}^{k} p_j$, and d' the number of elements in the filter is an arbitrary positive number.

The false positive rate of the Bloom filter is [1], [10]

$$\boldsymbol{P}_{false}^{BF} = \left(1 - \left(1 - \frac{1}{m}\right)^{kd'}\right)^k \approx \left(1 - e^{-kd'/m}\right)^k \quad , \tag{10}$$

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

As a direct consequence of the Chinese Remainder Theorem we have (see also Lemma 2 in [42] for two primes):

Property 1: Let $p_1, \ldots, p_k$ be a set of pairwise co-prime numbers, and let Z denote uniformly distributed non-negative integer random variable over the range $[1, \prod_{j=1}^{k} p_j]$. The variables $X_i := (Z \mod p_i)$ $(1 \leq i \leq k)$ are mutually independent.

The probability of a false positive over universe $1, \ldots, n'$ given d' elements in the filter satisfies

$$\boldsymbol{P}_{false}^{EGH} < \prod_{i=1}^{k} \left(1 - \left(1 - \frac{1}{p_i}\right)^{d'}\right). \tag{11}$$

To be more precise, the right-hand side is the probability that an arbitrary element u of the universe will give a positive answer for membership testing of an EGH filter composed as follows: we previously inserted into an empty filter d' elements that are uniformly distributed and mutually independent elements of the universe. Note that, here we can precisely quantify the difference between the left- and the right-hand side of the inequality, that is a typically very small value representing the probability of a positive answer because the last random element is actually in the filter, i.e., $1 - \left(1 - \frac{1}{n'}\right)^{d'}$.

Next, we describe an upper estimation of the right-hand side of (11), and thus, also for the probability $\boldsymbol{P}_{false}^{EGH}$. Assume that we choose the first k primes for $p_1, \ldots, p_k$. In the next step by applying (twice) the well known inequality between the arithmetic and geometric mean we obtain

$$\prod_{i=1}^{k} \left(1 - \left(1 - \frac{1}{p_i}\right)^{d'}\right) \leq \left(\frac{\sum_{i=1}^{k} \left(1 - \left(1 - \frac{1}{p_i}\right)^{d'}\right)}{k}\right)^k$$
$$\leq \left(1 - \left(\sqrt[k]{\prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)}\right)^{d'}\right)^k.$$

Using also the well known estimation [59, formula (3.27)] that if $x \geq 3$ then

$$\prod_{p \leq x} \left(1 - \frac{1}{p}\right) \geq \frac{0.09}{\log x},$$

we obtain that

$$\left(1 - \left(\sqrt[k]{\prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)}\right)^{d'}\right)^k < \left(1 - \left(\sqrt[k]{\frac{0.09}{\log p_k}}\right)^{d'}\right)^k$$
$$< \left(1 - \left(\sqrt[k]{\frac{1}{12 \log p_k}}\right)^{d'}\right)^k$$
$$< \left(1 - \left(\frac{1}{\sqrt[k]{12(\log k + \log(\log k + \log\log k + 8))}}\right)^{d'}\right)^k.$$

For the last inequality we are using the fact that [60] for $k > 1$ we have $p_k < k(\log k + \log\log k + 8)$.

VI. Accurate Listing of Elements

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]. Here, d is the maximum number of elements inserted in the filter. This makes the EGH structure taking $O(d^2 \log n \log d)$ space. In this variant, inserting an element x is done by incrementing the counters $\hat{h}_i(x)$ by 1 for $i = 1, \ldots, k$. Note that Claim 3 gives an upper bound on k, which is the number of counter values modification at element insertion. Deletion of an item y that had previously been inserted is carried out by decrementing the k corresponding counters by 1.

The key benefit is that if the EGH filter is in the False Positive Free Zone, listing the elements can be done in a deterministic way. This section will provide an efficient algorithm for accurately listing the elements; however, it requires maintaining counters instead of bits. The main idea is to define a system of equations whose roots in modular arithmetic $x_1, \ldots, x_d$ are exactly the elements in the filter. Unfortunately, we need counters to define such a system of equations, because roughly speaking we need to know how many times $x_1, \ldots, x_d$ has a residue of a given prime. Note that an accurate listing of the elements of a basic EGH filter (without counters) is also possible; however, we are not aware of any efficient algorithm. One obvious algorithm is to iterate through the universe $\{1, \ldots, n\}$ and perform membership testing for each entry. We can also try to guess the counter values with a brute-force searching all possible counter values. It is trivial for $d = 1, 2$, and can be efficient for small d.

In the rest of the section, we explain how to define a system of equations where the roots are the elements in modular arithmetic. We show how to solve the equations through algebraic computations to list the elements in the filter. Our algorithm that runs in $O(\mathrm{poly}(d \log(n)))$ steps for listing the elements, where d is the number of elements in the filter.

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 remainder of x_1 and x_2 for the prime p_i for $i \in [1, k]$ respectively, where $p_1 \cdot p_2 \cdots p_k > n^2$. The task is to compute two integers $x_1,\ x_2 \in [1, n]$ 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 fact is valid for $x_1 \times x_2$ and for $x_1 - x_2$. 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. As a result $(y_{i,1} - y_{i,2})^2 \pmod{p_i}$ is congruent to $z_i := (x_1 - x_2)^2 \pmod{p_i}$ for every $1 \le i \le k$. Even if we swap $y_{i,1}$ and $y_{i,2}$ we get the same values of z_i after squaring. In other words, this symmetric function is invariant for swapping the remainders of x_1 and x_2. On the other hand from the residues z_i we can obtain $z := (x_1 - x_2)^2 \pmod{p_1 \cdot p_2 \cdots p_k}$ by solving the corresponding system of congruences by using the Chinese Remainder Theorem, and we know that z is a square of an integer because x_1 and x_2 are both in $[1, n]$, thus their difference cannot be more than n, hence $z \le n^2$, thus we have an equality in the previous congruence i.e., $z = (x_1 - x_2)^2$. Next we need to find the square root of the integer z. This can be done with Newton-iteration or binary search for large n. Let u be the positive square root of z, and assuming that $x_1 > x_2$ we have $x_1 - x_2 = u$. On the other hand it is clear that $x_1 + x_2 \equiv y_{i,1} + y_{i,2} \pmod{p_i}$ for $1 \le i \le k$. 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 provided by the Chinese Remainder method.

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_i's are $0, 1, 4, 4$. By using the Chinese Remainder Theorem (Algorithm 1) we obtain that $z \equiv 4 \pmod{210}$ thus $u = 2$.

By (Algorithm 1) we get that $x_1 - x_2 \equiv 2 \pmod{210}$. Similarly $x_1 + x_2 \equiv 10 \pmod{210}$. 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 make use of the theory of elementary symmetric polynomials. We use the following property of polynomials [61]: given a polynomial $p(z)$, where α_i denotes its coefficients and x_is are the roots of $p(z)$, i.e.,

$$p(z) = z^d + \ldots + \alpha_{d-1} z + \alpha_d = (z - x_1) \ldots (z - x_d), \quad (12)$$

then we have

$$\alpha_i = (-1)^i \sigma_i(x_1, \ldots, x_d), \quad (13)$$

where $\sigma_i(x_1, \ldots, x_d)$ for $i \in [1, d]$ is called the ith elementary symmetric polynomial of $x_1, \ldots, x_d$ and can be computed by Algorithm 2 [62].

The obtained elementary symmetric polynomial for $m \in [1, d]$ is

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

The following algorithm computes the elementary symmetric polynomials all together.

For example for $d = 3$ we have

$$\sigma_1(x_1, x_2, x_3) = x_1 + x_2 + x_3 \ , \quad (14)$$

$$\sigma_2(x_1, x_2, x_3) = x_1 x_2 + x_1 x_3 + x_2 x_3 \ , \quad (15)$$

$$\sigma_3(x_1, x_2, x_3) = x_1 x_2 x_3 \ . \quad (16)$$

Next, we explain how to define for a general d, a system of equations in modular arithmetic whose roots are the elements

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
1   $\sigma_0^{(i)} := 1$, for $i = 1, \ldots d-1$
2   $\sigma_j^{(i)} = 0$, for all $j > i$
3   $\sigma_1^{(1)} = x_1$
4   for $i = 2$ to $d$ do
5       for $j = 1$ to $i$ do
6           $\sigma_j^{(i)} = \sigma_j^{(i-1)} + x_i \sigma_{j-1}^{(i-1)}$
```

in the filter. Let $p_1, \ldots, p_k$ be distinct primes. 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:

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

for all $i \in [1, k]$.

Note that in the advanced filter we have counters instead of bits, thus we know how many elements lie in each residue class, which is necessary to generate vector $y_{i,1}, \ldots, y_{i,d}$ if $d > 2$. 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 hold

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

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

$$a_i^{(j)} \equiv \sigma_j(y_{i,1},\ldots,y_{i,d}) \pmod{p_i} \tag{17}$$

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, applying Chinese Remainder Theorem for each row of the above equations to obtain

$$A_j := \text{ChineseRemainder}(a_1^{(j)}, \ldots, a_k^{(j)}, \quad p_1, \ldots, p_k) , \tag{18}$$

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

$$\begin{aligned}\sigma_j(x_1,\ldots,x_d) \leq \binom{d}{j} n^j = \binom{d}{d-j} n^j &\leq d^{d-j} n^j\\ &\leq n^{d-j} n^j = n^d.\end{aligned} \tag{19}$$

From $P \geq n^d$, $\sigma_j(x_1, \ldots, x_d) \leq n^d$ and the uniqueness part of the Chinese Remainder Theorem it follows that the above congruences for the A_j can be stated as equalities, 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. (12) and (13) 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

$$f(z) \ = \ z^d - A_1 z^{d-1} + A_2 z^{d-2} - \ldots. + (-1)^d A_d. \tag{20}$$

It can be done with standard mathematical algorithms, such as the Root Finder method of Heindel [63] based on the Bisection Method [64] and the Sturm Sequence [61]. 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. We overview these known methods in the Appendix.

Algorithm 3: ListEghFilter

```
Input: $y_{i,1}, \ldots, y_{i,d}$ for all $1 \leq i \leq k$, $p_1, \ldots, p_k$
begin
1   for $j = 1$ to $d$ do
2       for $i = 1$ to $k$ do
3           $a_i^{(j)} := \sigma_j(y_{i,1}, \ldots, y_{i,d}) \pmod{p_i}$
4       $A_j :=$
          ChineseRemainder$(a_1^{(j)}, \ldots, a_k^{(j)}, p_1, \ldots, p_k)$
5   Set $f(z) = z^d + \sum_{t=1}^d (-1)^t A_t z^{d-t}$
6   Compute $(x_1, x_2, \ldots, x_d) =$ RootFinder$(f(z), 0, P)$
```

To summarize the above we have the following ListEghFilter algorithm shown in Algorithm 3. In the inner loop we compute $a_i^{(j)}$ by Eq. (17) 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 $y_{i,1}, \ldots, y_{i,d}$. In the outer loop this gives the A_js with the Chinese remaindering process as in Eq. (18). Then we build up our polynomial $f(z)$ and find its roots by using the RootFinder method.

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

[3]Theorem 3.1 in [65] provides a faster but more complex algorithm than the Sturm Sequence that finds the items at the Boolean cost $\tilde{O}(d^3 \log^3 n)$.

Proof: The algorithm contains three steps. The first step 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 $x_1, \ldots, x_d \leq n$, one addition needs $O(\log n)$ bit operations, and one multiplication requires $O(\log^2 n)$ bit operations. Accordingly, the total cost of Algorithm 2 is $O(d^2 \log^2 n)$ bit operations.

In the next step, we compute the residues of the elementary symmetric polynomials $\sigma_j(y_{i,1}, \ldots, y_{i,d}) \pmod{p_i}$, $(1 \leq i \leq k)$ $(1 \leq j \leq d)$. We have $k \times d$ such residues. Obviously, $k \leq d \log n$. As noted before the p_is are (the first) distinct prime numbers. It follows from [16] that $p_k = O(d \log n)$. By (19), to compute all residues one requires at most $O(d \times k \times \log(n^d) \times \log(d \log n)) = O(d^3 \log^2 n(\log d + \log\log n)) = O(d^{3+\varepsilon}(\log n)^{2+\varepsilon})$ bit operations.

We analyze the Chinese remaindering process (Algorithm 1). Chinese remaindering requires $O(\log^2 P)$ bit operations [66]. Obviously, $p_i \leq n$, it follows that

$$\begin{aligned}\log P = \sum_{i=1}^{k} \log p_i &< k \log p_k \\ &= O((d \log n \cdot (\log d + \log\log n)) \\ &= O((d \log n)^{1+\varepsilon}).\end{aligned}$$

Since the number of systems of congruences is d, computing the A_js in the ListEghFilter needs $O(d \cdot (d \log n)^{2+\varepsilon})$ 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 $f(z)$.

Coefficients of our polynomial are at most n^d, which implies that $K \leq dn^d$. By [67] the running time of RootFinder is $O(d^{10} + d^7 \log^3 K)$ (see Theorem 8 and the remark following it in [67]). 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 $O(d^2 \log^2 n + d^{3+\varepsilon} \log^{2+\varepsilon} n + d^{10} \log^3 n) = O(d^{10} \log^3 n)$ bit operations. ■

VII. Numerical Evaluation

We perform experiments to examine the performance of the EGH filter under different scenarios. We have released the `C++` source code of our implementation for the EGH filter construction and the algorithm for listing elements[4]. 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. While we focus on small d, we also evaluate the performance for larger d values to further understand the solution's cost in practice.

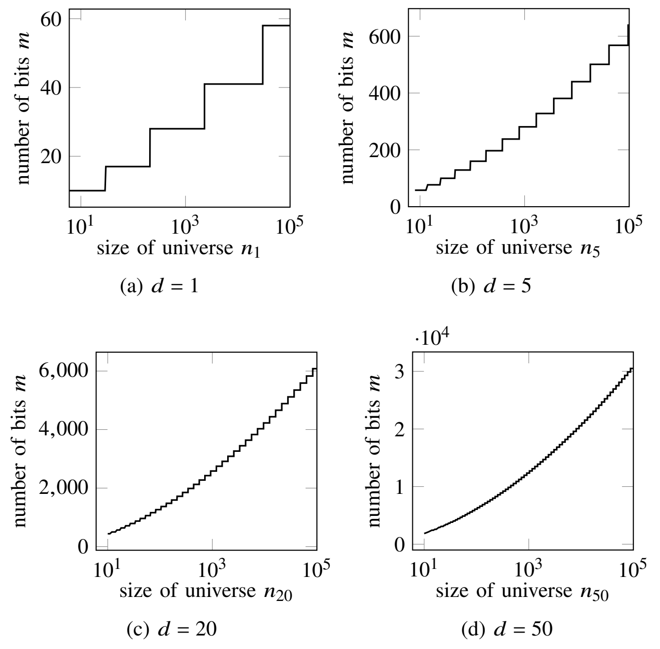


Fig. 2: The EGH filter's length depending on the size of the false-positive free zone $[1, n_d]$.

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 V. 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 II, 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 first 17 primes will sum up to at least 440 and are required to allow such false positive free zone. As a comparison, reading such 5 elements in binary representation would require 75 bit-access $(5 \cdot \lceil \log_2 18100 \rceil)$. The table also shows the ability of larger filters (with their corresponding overhead) to allow larger universe size n and set size d. For instance, access to 20 positions in a filter of 639 bits, would allow representing $d = 6$ elements from a universe size of $n_6 = 28700$ or $d = 9$ among $n_9 = 937$.

Fig. 2 shows the number of bits needed in the EGH filter to have a false positive free zone for $d \in \{1, 5, 20, 50\}$ and a universe size that ranges from 10 to 10^5. 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. For example, for $d = 5$ as many as 281 bits are required for a universe of size 1670, and 440 bits for a universe of size

[4]https://github.com/jtapolcai/egh_filter

TABLE II: 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. See also Fig. 1 as a graphical representation.

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	$n_1 = 6.5{\cdot}10^9$ $n_2 = 80400$ $n_3 = 1860$ $n_4 = 283$

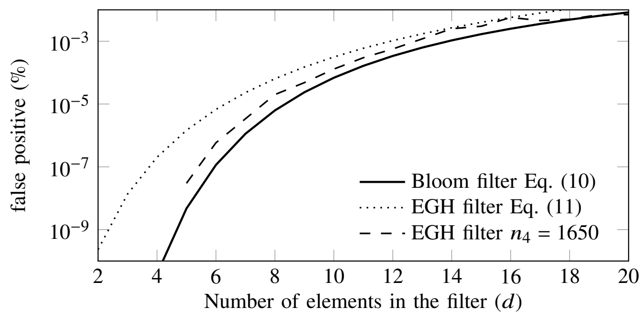
p_k	m	FPFZ
$p_{11} = 31$	160	$n_1 = 2.01{\cdot}10^{11}$ $n_2 = 448000$ $n_3 = 5850$ $n_4 = 669$ $n_5 = 182$
$p_{12} = 37$	197	$n_1 = 7.42{\cdot}10^{12}$ $n_2 = 2.72{\cdot}10^6$ $n_3 = 19500$ $n_4 = 1650$ $n_5 = 375$
$p_{13} = 41$	238	$n_1 = 3.04{\cdot}10^{14}$ $n_2 = 1.74{\cdot}10^7$ $n_3 = 67300$ $n_4 = 4180$ $n_5 = 788$ $n_6 = 259$
$p_{14} = 43$	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	$n_1 = 1.92{\cdot}10^{21}$ $n_2 = 4.38{\cdot}10^{10}$ $n_3 = 1.24{\cdot}10^7$ $n_4 = 209000$ $n_5 = 18100$ $n_6 = 3530$ $n_7 = 1100$ $n_8 = 458$

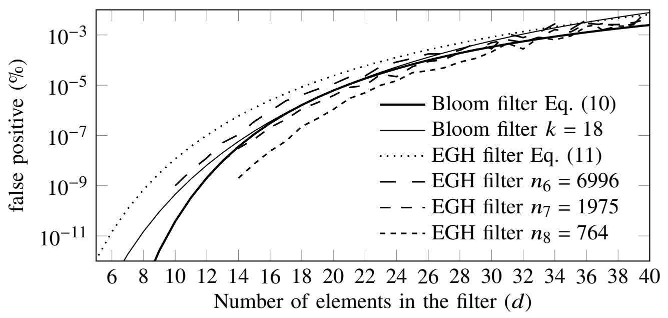
p_k	m	FPFZ
$p_{18} = 61$	501	$n_2 = 3.42{\cdot}10^{11}$ $n_3 = 4.89{\cdot}10^7$ $n_4 = 585000$ $n_5 = 41100$ $n_6 = 7000$ $n_7 = 1980$ $n_8 = 765$
$p_{19} = 67$	568	$n_2 = 2.80{\cdot}10^{12}$ $n_3 = 1.99{\cdot}10^8$ $n_4 = 1.67{\cdot}10^6$ $n_5 = 95300$ $n_6 = 14100$ $n_7 = 3600$ $n_8 = 1290$ $n_9 = 584$
$p_{20} = 71$	639	$n_3 = 8.23{\cdot}10^8$ $n_4 = 4.86{\cdot}10^6$ $n_5 = 224000$ $n_6 = 28700$ $n_7 = 6620$ $n_8 = 2200$ $n_9 = 937$

p_k	m	FPFZ
$p_{21} = 73$	712	$n_3 = 3.44{\cdot}10^9$ $n_4 = 1.42{\cdot}10^7$ $n_5 = 527000$ $n_6 = 58700$ $n_7 = 12200$ $n_8 = 3770$ $n_9 = 1510$ $n_{10} = 726$
$p_{22} = 79$	791	$n_4 = 4.24{\cdot}10^7$ $n_5 = 1.26{\cdot}10^6$ $n_6 = 122000$ $n_7 = 22800$ $n_8 = 6510$ $n_9 = 2450$ $n_{10} = 1120$
$p_{23} = 83$	874	$n_4 = 1.28{\cdot}10^8$ $n_5 = 3.06{\cdot}10^6$ $n_6 = 254000$ $n_7 = 42900$ $n_8 = 11300$ $n_9 = 4010$ $n_{10} = 1750$

p_k	m	FPFZ
$p_{24} = 89$	963	$n_5 = 7.50{\cdot}10^6$ $n_6 = 536000$ $n_7 = 81400$ $n_8 = 19800$ $n_9 = 6600$ $n_{10} = 2740$ $n_{11} = 1330$
$p_{25} = 97$	1060	$n_6 = 1.15{\cdot}10^6$ $n_7 = 157000$ $n_8 = 35100$ $n_9 = 11000$ $n_{10} = 4330$ $n_{11} = 2020$ $n_{12} = 1070$
$p_{26} = 101$	1161	$n_5 = 4.71{\cdot}10^7$ $n_6 = 2.48{\cdot}10^6$ $n_7 = 303000$ $n_8 = 62500$ $n_9 = 18300$ $n_{10} = 6870$ $n_{11} = 3080$ $n_{12} = 1570$



(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. 3: 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.

18100. The theoretical lower bound on the size m of the d-disjunct matrices by Claim 2 (from Section IV) is also shown on the charts with dashed lines. Filling this relatively large gap between the lower bound and the constructions is a long-standing open problem in CGT.[5]

In our C++ implementation of the list operation for advanced EGH filter, we have used large precision numbers for the ROOTFINDER algorithm, although the 64-bit `long long` should also be enough in practice. See Fig. 4 for the running time of listing d elements of universe n on a commodity laptop with Intel i5 2.5Ghz commodity processor.

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

[5]Note that a randomized CGT construction cannot be transformed into simple $\{\hat{h}_1, \hat{h}_2, \ldots, \hat{h}_k\}$ functions in the context of the EGH filter.

Fig. 3 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. (10) for an optimal number of hash functions, and 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. (11). The false positive is slightly larger for EGH outside of the FPFZ, especially for small d, which we pay to have an 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 $U = \{1, \ldots, 6996\}$ ($n_6 = 6996$) has an 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 BF as more and more elements are inserted into 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

TABLE III: 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 correspond to the Bloom filter with optimal number of hash functions.

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

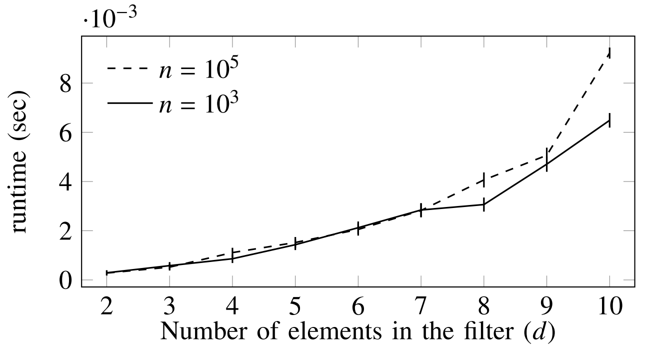


Fig. 4: Runtime of the decoding algorithm with 95% confidence intervals of 1000 measurements.

and larger d can also meet the same bound of $\prod_{i=1}^{k} p_i \geq n^d$.

VIII. 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 they are. This allows great flexibility because the Bloom filter size can be dynamically changed without recomputing the filter. To reduce the length, we need to erase the last blocks. Sometimes filter length has to be increased, allowing a false positive free zone for a larger set. We can increase length by adding blocks that refer to new primes values while maintaining existing filter blocks. The computation of the new filter can be simplified by computing only its value for the added blocks and relying on previous values of the existing blocks.

B. Tradeoff between Memory and False Positive Guarantees

There is a clear tradeoff between the implied false positive free zone to the amount of memory allocated to the filter. An example for this tradeoff was shown early through the various curves in Fig. 1. The memory consumption $O(d^2 \log n \log d)$ bits of the EGH filter has a monotonically increasing cost (in bits) for adding one to the upper bound d on the set size. Memory should be allocated based on the typical size of the represented set when membership queries are performed. Consider a distribution $(\phi_0, \phi_1, \phi_2, \ldots)$ such that $\sum_{i \geq 0} \phi_i = 1$ and $\phi_i \in [0, 1]$ describes the probability of the set S to be of size $|S| = i$ upon a query. The distribution can be implied by typical set size. Clearly, there is no point in increasing d from some i to $i + 1$ if there are no queries when $|S| = i + 1$ (namely $\phi_{i+1} = 0$). In addition to this distribution, the selection of the filter size should take into account the false positive probability beyond the false positive free zone as analyzed in Section V-C as a function of the filter length. Let $P^{EGH}_{false}(m, i)$ denote the implied false positive probability with a construction of m bits (with some first k primes) representing a set of size i. Consider a fixed universe of size n and assume the guarantee of no false positives for a query has some value W implied by the particular system. One possible guideline can be selecting d such the value $(W \cdot \sum_{i \in [0,d]} \phi_i) + \sum_{i \geq d+1} \phi_i \cdot (1 - P^{EGH}_{false}(m(d, n), i))$ is large in comparison with the memory cost $m(d, n)$.

C. 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 a low false-positive rate. In EGH, we need to perform only a simple division with the remainder operation. Moreover, the same function $\hat{h}_i$ in an EGH filter is used if $i \leq k$. This means, the functions $\{\hat{h}_1, \hat{h}_2, \ldots, \hat{h}_k\}$ can be hardware-implemented, or in assembly for software implementations. The EGH filter size defines the required number of functions.

IX. Conclusion

In this paper, we described the EGH filter to represent 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 counters, supporting deletions, and efficient listing of the elements. The fast listing of the elements is performed by finding the roots of a system of equations in modular arithmetic. Our approach is based on traditional number-theoretical techniques such as the Chinese Remainder Theorem, the Bisection Method, and the Sturm Sequence. We examined the performance of the new approach. The paper aims to connect two heavily researched fields: Bloom filters and Combinatorial Group Testing. While we showed how such a connection could allow false-positive free zones with their unique properties, we demonstrated that there could be a high memory overhead for the guarantee to prevent false positives while preserving the basic Bloom filter access properties.

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, and X. Luo, "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] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," *IEEE/ACM Trans. Netw.*, vol. 22, no. 4, pp. 1092–1105, 2014.

[8] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing Bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys & Tutorials*, 2019.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] L. Luo, D. Guo, J. Wu, O. Rottenstreich, Q. He, Y. Qin, and X. Luo, "Efficient multiset synchronization," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1190–1205, 2017.

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

[16] 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.

[17] J. Baliosian and R. Stadler, "Distributed auto-configuration of neighboring cell graphs in radio access networks," *IEEE Transactions on Network and Service Management*, vol. 7, no. 3, pp. 145–157, 2010.

[18] 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.

[19] 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.

[20] 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.

[21] 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 *IFIP Networking*, 2017.

[22] 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.

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

[24] 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.

[25] 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.

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

[27] O. Rottenstreich and I. Keslassy, "The Bloom paradox: When not to use a Bloom filter," *IEEE/ACM Trans. Netw.*, vol. 23, no. 3, pp. 703–716, 2015.

[28] P. Reviriego and O. Rottenstreich, "The tandem counting bloom filter - it takes two counters to tango," *IEEE/ACM Trans. Netw.*, vol. 27, no. 6, pp. 2252–2265, 2019.

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

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

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

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

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

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

[35] 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.

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

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

[38] 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.

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

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

[41] T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li, "A shifting framework for set queries," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3116–3131, 2017.

[42] J. Lu, T. Yang, Y. Wang, H. Dai, X. Chen, L. Jin, H. Song, and B. Liu, "Low computational cost bloom filters," *IEEE/ACM Trans. Netw.*, vol. 26, no. 5, pp. 2254–2267, 2018.

[43] D. Ficara, A. D. Pietro, S. Giordano, G. Procissi, and F. Vitucci, "Enhancing counting Bloom filters through huffman-coded multilayer structures," *IEEE/ACM Trans. Netw.*, vol. 18, no. 6, pp. 1977–1987, 2010.

[44] L. Li, B. Wang, and J. Lan, "A variable length counting Bloom filter," in *International Conference on Computer Engineering and Technology*, 2010.

[45] H. Lim, J. Lee, and C. Yim, "Complement Bloom filter for identifying true positiveness of a Bloom filter," *IEEE Communications Letters*, vol. 19, no. 11, pp. 1905–1908, 2015.

[46] H. Lim, N. Lee, J. Lee, and C. Yim, "Reducing false positives of a Bloom filter using cross-checking Bloom filters," *Applied Mathematics & Information Sciences*, vol. 8, no. 4, p. 1865, 2014.

[47] J. Bruck, J. Gao, and A. Jiang, "Weighted Bloom filter," in *IEEE ISIT*, 2006.

[48] M. Zhong, P. Lu, K. Shen, and J. I. Seiferas, "Optimizing data popularity conscious Bloom filters," in *ACM PODC*, 2008.

[49] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond Bloom filters: From approximate membership checks to approximate state machines," in *ACM SIGCOMM*, 2006.

[50] S. Pontarelli, P. Reviriego, and J. A. Maestro, "Improving counting Bloom filter performance with fingerprints," *Inf. Process. Lett.*, vol. 116, no. 4, pp. 304–309, 2016.

[51] B. Donnet, B. Baynat, and T. Friedman, "Retouched Bloom filters: Allowing networked applications to trade off selected false positives against false negatives," in *ACM CoNEXT*, 2006.

[52] R. P. Laufer, P. B. Velloso, and O. C. M. B. Duarte, "A generalized Bloom filter to secure distributed network applications," *Computer Networks*, vol. 55, no. 8, pp. 1804–1819, 2011.

[53] 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.

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

[55] F. Hwang and V. Sós, "Non-adaptive hypergeometric group testing," *Studia Sci. Math. Hungar*, vol. 22, pp. 257–263, 1987.

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

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

[58] P. Dusart, “The k-th prime is greater than $k(\ln k + \ln\ln k - 1)$ for $k \geq 2$,” *Mathematics of Computation*, vol. 68, no. 225, pp. 411–415, 1999.
[59] J. Rosser and L. Schoenfeld, “Approximate formulas for some functions of prime numbers,” *Illinois J. Math.*, vol. 1, pp. 64–94, 1962.
[60] J. Rosser, “The n-th prime is greater than $n \log n$,” *Proc. Lond. Math. Soc.*, vol. 2, pp. 21–44, 1938.
[61] V. V. Prasolov and D. Leites, *Polynomials*. Springer, 2004, vol. 11.
[62] H. Jiang, S. Graillat, and R. Barrio, “Accurate and fast evaluation of elementary symmetric functions,” in *IEEE Symposium on Computer Arithmetic*, 2013.
[63] L. E. Heindel, “Integer arithmetic algorithms for polynomial real zero determination,” *Journal of the ACM*, vol. 18, no. 4, pp. 533–548, 1971.
[64] B. Faires, *Numerical Analysis 6th*. Brooks/Cole Pub., 1997.
[65] I. Z. Emiris, V. Y. Pan, and E. P. Tsigaridas, “Algebraic algorithms,” *Computer Science Technical Reports, Paper 361*, 2012.
[66] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.
[67] G. E. Collins and R. Loos, “Polynomial real root isolation by differentiation,” in *ACM symposium on Symbolic and algebraic computation*, 1976.

Appendix

We give a short overview of the main known tools we make use of. First, we describe the well-known Bisection method [64], 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 predefined threshold ε. Pseudo-code is given in Algorithm 4.

Algorithm 4: BisectionMethod [64]

```
  Input: Polynomial $f$, range $[a, b]$, accuracy $\varepsilon$
1 while $\frac{b-a}{2} > \varepsilon$ do
2     Compute $c = \frac{a+b}{2}$ and $d = f(c) \cdot f(a)$
3     if $d < 0$ then
4         $b := c$
5     if $d > 0$ then
6         $a := c$
7     if $d = 0$ then
8         break
  return $c$
```

We also need some facts about Sturm sequences [61]. 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 $f_{i+1}(z)$.

Algorithm 5: SturmSequence [61]

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

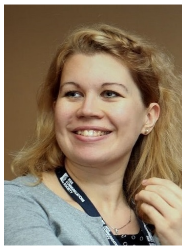
Let $f_0(z) = f(z)$, $f_1(z), \ldots, f_m(z)$ be a Sturm sequence, where $f(z)$ has no repeated roots, and let $\omega(z)$ denote the number of sign changes (ignoring zeroes) in the sequence $f(z), f_1(z), \ldots, f_m(z)$. Sturm's theorem states that if $f(a) \neq 0$ and $f(b) \neq 0$, then the number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

Algorithm 6: RootFinder [63]

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

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 halving point, and calculate the number of roots in both of these smaller intervals. Repeat these steps until one of 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.

Sándor Kiss received his Ph.D. degree in Mathematics from Eötvös Loránd University Budapest in 2010. He worked as a junior research fellow at the Institute of Computer Science and Control for one year. He is an Associate Professor at the Budapest University of Technology and Economics. His main research areas involve number theory, discrete mathematics and algebraic algorithms.



Éva Hosszu earned her MSc in Applied Mathematics in 2012 from the Budapest University of Technology and Economics. She is currently pursuing her PhD in Computer Science. Her research interests include efficient failure localization in optical networks and combinatorial optimizatiion. Her current research focuses on the applications of combinatorial search in telecommunication networks.

János Tapolcai received the MSc degree in technical informatics and the PhD degree in computer science from the Budapest University of Technology and Economics (BME), Budapest, in 2000 and 2005, respectively, and the D.Sc. degree in engineering science from the Hungarian Academy of Sciences (MTA) in 2013. He is currently a Full Professor with the High-Speed Networks Laboratory, Department of Telecommunications and Media Informatics, BME. He has authored over 150 scientific publications. He was a recipient of several Best Paper Awards, including ICC'06, DRCN'11, HPSR'15, and NaNa'16. He is a winner of the MTA Lendület Program and the Google Faculty Award in 2012, Microsoft Azure Research Award in 2018. He is a TPC member of leading conferences, e.g., IEEE INFOCOM 2012-, and the general chair of ACM SIGCOMM 2018.

Lajos Rónyai is a research professor with the Institute of Computer Science and Control, Eötvös Loránd Research Network, Budapest, Hungary. He leads a research group there which focuses on theoretical computer science and discrete mathematics. He is also a full professor at the Mathematics Institute of the Budapest University of Technology and Economics. He received his PhD in 1987 from the Eötvös Loránd University Budapest. His research interests include efficient algorithms, complexity of computation, algebra, and discrete mathematics. He is a member of the Hungarian Academy of Sciences and a recipient of the Count Széchenyi Prize.

Ori Rottenstreich is an assistant professor at the department of Computer Science and the department of Electrical Engineering of the Technion, Haifa, Israel. Previously, he was a Postdoctoral Research Fellow at Princeton university. Ori received his B.Sc. degree in Computer Engineering and Ph.D. degree in Electrical Engineering from Technion.