Low Computational Cost Bloom Filters

Jianyuan Lu, Tong Yang, *Member, IEEE*, Yi Wang, *Member, IEEE*, Huichen Dai, *Member, IEEE*, Xi Chen, Linxiao Jin, Haoyu Song, *Senior Member, IEEE*, and Bin Liu, *Senior Member, IEEE*

***Abstract*— Bloom filters (BFs) are widely used in many network applications but the high computational cost limits the system performance. In this paper, we introduce a low computational cost Bloom filter named One-Hashing Bloom filter (OHBF) to solve the problem. The OHBF requires only one base hash function plus a few simple modulo operations to implement a Bloom filter. While keeping nearly the same theoretical false positive ratio as a Standard Bloom filter (SBF), the OHBF significantly reduces the computational overhead of the hash functions. We show that the practical false positive ratio of an SBF implementation strongly relies on the selection of hash functions, even if these hash functions are considered good. In contrast, the practical false positive ratio of an OHBF implementation is consistently close to its theoretical bound. The stable false positive performance of the OHBF can be precisely derived from a proved mathematical foundation. As the OHBF has reduced computational overhead, it is ideal for high throughput and low-latency applications. We use a case study to show the advantages of the OHBF. In a BF-based FIB lookup system, the lookup throughput of OHBF-based solution can achieve twice as fast as the SBF-based solution.**

***Index Terms*— Bloom filter, hash function, modulo operation.**

I. Introduction

THE recent trends of Software-defined Networking (SDN) and Network Function Virtualization (NFV) [1] increasingly demand implementing and deploying network functions in software appliance such as commodity servers for flexibility and cost efficiency. Hash Table is an indispensable and powerful tool to realize a wide range of network functions. Bloom filter, as a memory efficient hashing scheme, has found its applications in different layers of network stack [2]. Extensive research has been conducted to improve this classical data structure in the past few years. Various novel network applications are made possible by the use of Bloom filter and its variants [3]–[5].

While the memory efficiency is a given benefit, the successful use of Bloom filter does come with a cost. A Bloom filter needs to use a relatively large number of hash functions (*e.g.*, 35 as in [6]) and perform the same number of memory accesses. For systems using multiple Bloom filters (*e.g.*, 32 Bloom filters are used in [3]), the required number of hash functions can be prohibitively large. To achieve the theoretical performance bound of a Bloom filter, these hash functions need to be strong (*i.e.*, presenting good randomness and uniformity) and mutually independent. Unfortunately, good hash functions (*e.g.*, MD5 and SHA-1) are known to be computation-intensive. While the memory access latency can be hidden with the well established mechanisms such as caching and banking, hash computations alone consume a lot of CPU cycles and introduce excessive latency which can become the system performance bottleneck. For example, a moderate 10GE interface requires 15Mpps throughput. This leaves less than 300 clock cycles for a state-of-the-art 4GHz CPU to process a packet. This limited clock budget simply cannot afford to compute a large number of independent and strong hash functions. Therefore, it is a critical and essential system requirement to reduce the cost of hash computation while retaining the desired hashing properties. Although simple hash functions may be used to speed up the Bloom filter query operations [7]–[9], we show that simple hash functions can not pass the randomness and independence tests most of the time, which causes a Bloom filter to significantly deviate its false positive ratio from the theoretical bound. In this paper, we address the hash computational cost issue by introducing a novel algorithm that requires only one strong hash function to realize a Bloom filter. To the best of our knowledge, this is by far the most efficient approach with a proved performance bound. The resulting data structure, called One-Hashing Bloom Filter (OHBF), presents nearly the same theoretical false positive ratio as a standard Bloom filter. Actually, we find that the practical false positive ratio of a standard Bloom filter implementation strongly relies on the selection of hash functions, even if these hash functions are considered good. Our experiments show that OHBF outperforms many existing practical Bloom filter implementations with more than one strong hash function. To test the effectiveness of OHBF in practical applications, we showcase a Bloom filter-based FIB lookup system. In the system, the lookup throughput of OHBF-based scheme is two times faster than the standard Bloom filter-based scheme.

Strictly speaking, in OHBF, the hashing process still generates k hash values as if we have k independent hash functions. The difference is, all the k hash values originate from a single hash function plus a few simple modulo operations. It is proved that the generated k hash values are pairwise independent. While the computational cost of a

Manuscript received December 17, 2016; revised September 24, 2017 and May 8, 2018; accepted August 13, 2018; approved by IEEE/ACM Transactions on Networking Editor M. Li. Date of publication September 20, 2018; date of current version October 15, 2018. This work was supported in part by the Huawei Innovation Research Program (HIRP); in part by the NSFC under Grant 61373143, Grant 61432009, and Grant 61872213. *(Corresponding author: Bin Liu.)*

J. Lu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with Alibaba Cloud, Beijing 100102, China (e-mail: lu-jy11@mails.tsinghua.edu.cn).

T. Yang, Y. Wang, H. Dai, L. Jin, and B. Liu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: liub@mail.tsinghua.edu.cn).

X. Chen is with the School of Information and Communication Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China.

H. Song is with Network Research, Huawei Technologies, Santa Clara, CA 95050 USA.

Digital Object Identifier 10.1109/TNET.2018.2869851

modulo operation is relatively fixed in modern CPUs, the computational cost of a hash function is proportional to the element size [10]. As we replace hash function computation with modulo computation in OHBF, the computational cost is approximately reduced to $1/k$. The gain is significant when the element size is large. The tradeoff is that the OHBF consumes more auxiliary memory and takes more time in Bloom filters' construction process in order to reduce the hash computational cost at runtime, *i.e.*, in the element lookup process. Moreover, in the construction process, OHBF uses some extra memory to store a prime table, and implements more instructions than traditional Bloom filters to build a specified partitioned bit array.

While we leave finding an optimal partition algorithm as future work, we extensively explore the design space covering a wide range of realistic application scenarios. It shows our algorithm, although simple, can satisfy all application requirements under different design constraints, such as memory size, element set size, and target false positive ratio.

The remaining of the paper is organized as follows. Section II surveys the related work. Section III details the OHBF scheme and theoretical analysis. Section IV analyses the performance of different practical Bloom filter implementations. Section V presents the experimental results for performance comparison. Section VI shows a case study for a Bloom filter-based FIB lookup system. Section VII concludes the paper.

II. Related Work

A. Standard Bloom Filter and Its Variants

A Standard Bloom Filter (SBF) [11] uses a bit vector of size m to represent a set S of n elements. All the bits in the Bloom filter are initialized to zero. When an element $x \in S$ is added to the Bloom filter, we use k different hash functions $h_i(x), 1 \leq i \leq k$ to map the element to k integer numbers in the range $[0, m-1]$. Then the corresponding bits are set to be one. We repeat the above process for each element in set S. After all the elements are hashed to the Bloom filter bit vector, the Bloom filter has been successfully established.

A membership query determines whether an element y belongs to the set S or not. If all the k corresponding bits indexed by $h_i(y)$ are ones, then $y \in S$; otherwise, $y \notin S$. But the answer to the querying process can be false positive. Suppose that $y \notin S$, but all the k hashed bits happen to be ones. In this case the query falsely concludes that $y \in S$. The false positive ratio for SBF is

$$f_s = \left(1 - (1 - 1/m)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k \tag{1}$$

The false positive ratio decreases as the size of the Bloom filter, m, increases. It increases as more elements, n, are added. By taking the derivative of f_s with respect to k and equalizing it to zero, we can get the optimal k which minimizes the false positive ratio for the given values of m and n.

$$k_{opt} = (m/n) ln2 \approx 9m/13n \tag{2}$$

A Bloom filter can be optimized and enhanced in different aspects:

1) Dynamic Updates. While SBF only supports element insertions, Counting Bloom Filters [12] substitute each bit by a small counter to support element deletions. Obviously, this method increases the memory space cost. Other similar dynamic update extensions of Bloom filter can be found in [13]–[17].

2) Counting. When an SBF answers that an element belongs to the set, we do not know the concrete frequencies of this item in this set. This problem is amended in [12] and [18]–[20] at the cost of more memory space or hashing computation.

3) Scalability. An SBF only supports static membership queries. In case the set cardinality is unknown prior to the Bloom filter construction, the Bloom filter variants in [21] and [22] can scale its capacity dynamically based on current set cardinality.

4) Multiple-set. [23], [24] extend SBF to support multiple-set membership test, which groups elements to different sets. In addition to false positive, classification failure happens with a probability. More recent related work can be found in [16], which uses novel coding techniques.

5) Cache Efficiency. A Bloom filter uses k hash functions and performs the same number of memory accesses. In the worst case, k cache misses happen in one element query. Blocked Bloom Filter [25] improves the cache efficiency by cutting the Bloom filter into blocks, each of which fits into one cache line. However, it increases the space cost. A similar work can be found in [26].

6) Generalization. [27], [28] introduce false negatives to Bloom filters. A tradeoff between false positives and false negatives makes the applications more flexible. The Bloomier filter [29] generalizes the SBF to support arbitrary function queries. Reference [30] generalizes the SBF to support approximate state machines.

7) Substitution. Cuckoo filters, proposed in [31], have lower space overhead than space-optimized Bloom filters when n is large and the target false positive ratio is very low. Cuckoo filter does not have the basic Bloom filter structure. It is a substitution of Bloom filters in many cases.

Different from the above studies, our work in this paper improves the Bloom filters from another aspect, *i.e.*, reducing the hash function computational cost. Several previous works have attempted to achieve this goal. Kirsh and Mitzenmacher use two base hash functions $h_1(x)$ and $h_2(x)$ to construct additional hash functions in the form of $g_i(x) = h_1(x) + ih_2(x)$ [32]. We call this scheme Less Hashing Bloom Filter (LHBF). Since this technique cannot guarantee the independence of the synthetic hash functions,[1] the false positive ratio in practice could be much higher than the theoretical expectation [8]. Song *et al.* [33] introduce a simple method to produce k hash values using $O(log\, k)$ seed hash functions. However, the paper does not analyze the correlation of the k hash values. In [34], Skjegstad and Maseng also use one hash function to implement Bloom filters for set reconciliation between two nodes. Studies in [12], [26], and [35] propose the

[1]An example to illustrate the correlation of the simulated hash functions is: consider $g_2(x) = h_1(x) + 2h_2(x)$ and $g_4(x) = h_1(x) + 4h_2(x)$; apparently, $g_2(x)$ and $g_4(x)$ have the same parity.

method that produces k hash values with a single hash function by dividing the generated hash bits into k hash segments. However, this method has several limitations. For example, the length of a Bloom filter needs to be a power of two, required by the hash segment's range. Meanwhile, the supported number of hash functions is not scalable, restricted by the Bloom filter's length and the hash segment's bit length. Our work is different with it mainly in two aspects. First, our proposed OHBF is an alternative to SBF and generalized for wider application scenarios. Second, due to strict independence of modulo operations, the OHBF has nearly the same false positive probability as SBF with the same memory constraint. More importantly, we formally analyze a hash value generation process, and propose a Bloom filter construction algorithm which results in nearly the same false positive ratio as a standard Bloom filter.

B. Practical Hash Functions for Bloom Filter

A Bloom filter needs k uniform and independent hash functions. If the hash function properties are compromised, the actual false positive ratio can be much worse than the theoretical analysis. The hash functions used for Bloom filters mainly fall in three groups:

1) Cryptographic Hash Functions. Cryptographic hash functions have good randomness assurance, so they are the popular choices for implementing Bloom filters. For example, MD5 is used in Bloom filter implementations [9], [12]. However, the complexity of MD5 is high. The cost of MD5 is proportional to key size. It requires 6.8 CPU cycles per byte on average [10]. The cost on hashing long keys can be prohibitive for some applications.

2) Non-cryptographic Hash Functions. Several relatively simple hash functions, such as CRC32, FNV and BKDR, are often used to implement Bloom filters [7]–[9]. Similarly, the computational complexity of these hash functions is proportional to the key size. While these hash functions are less computation-intensive than the cryptographic hash functions, their randomness is not as good, which translates to higher Bloom filter false positive ratios.

3) Universal Hash Functions. Hash functions can be selected from a family of hash functions with a certain mathematical property [36]. The Bloom filter implementations with these hash functions can approach the ideal false positive ratio [37]. Since the universal hash functions need to be "randomly" selected from a family, the practical implementation still needs the aid of traditional hash functions (*i.e.*, cryptographic and non-cryptographic hash functions). Therefore, in the latter part of this paper, we do not consider universal hash functions when implementing Bloom filters.

III. Design and Theoretical Analysis

In this section, we will first divide the hashing process into two stages for ease of discussion. Second, we will describe the One-Hashing Bloom Filter (OHBF) design. Then we will prove that hashings in the modulo stage of OHBF are mutually independent, which mathematically guarantees the OHBF's performance. The false positive probability of OHBF will be analyzed and formulated. At last, we will introduce an algorithm to determine the size of partitions for OHBF.

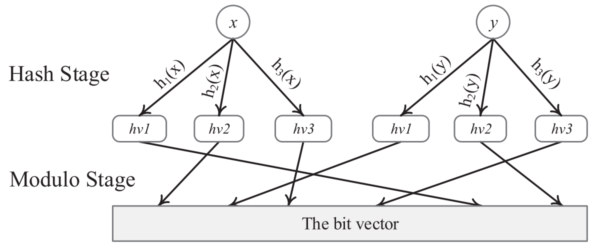


Fig. 1. A schematic view of SBF with two elements ($n = 2$). $k = 3$ hash functions are used in the hash stage. The output of hash functions will modulo the Bloom filter size in the modulo stage.

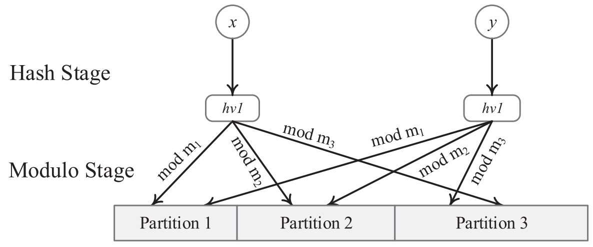


Fig. 2. A schematic view of OHBF with two elements ($n = 2$). Only one base hash function is used in the hash stage. The output of the base hash function will modulo $k = 3$ partition sizes in the modulo stage.

A. Two Stages of Bloom Filter Hashing

In a Bloom filter, hash functions are used to compute the filter entry index. The hashing process is essentially a mapping from $\mathcal{U} \rightarrow \mathcal{V}$, where $\mathcal{U}$ is the space of elements and $\mathcal{V}$ is the space of the Bloom filter. The hashing process is often conducted in two stages:

1) Hash Stage: $\mathcal{U} \rightarrow \mathcal{M}$, mapping $\mathcal{U}$ to a machine word $\mathcal{M}$ (*e.g.*, 32-bit or 64-bit), using a hash function $h(x)$.
2) Modulo Stage: $\mathcal{M} \rightarrow \mathcal{V}$, mapping $\mathcal{M}$ to target $\mathcal{V}$, by modulo $|\mathcal{V}|$ (*i.e.*, $h(x)\ mod\ m$). This is needed because $h(x)$ usually covers a larger space than the Bloom filter size m.

Therefore, in our OHBF scheme, the hashing process becomes $\mathcal{U} \rightarrow \mathcal{M} \rightarrow \mathcal{V}$. People usually treat the hash mapping as an integral process and do not distinguish these two stages explicitly. Because in most cases, the modulo stage will always modulo the Bloom filter size. But we show that the two stage separation of hash function can be taken advantage of to significantly simplify the Bloom filter implementation. In the latter part of this paper, we use the notation $h(x)$ to represent the hash stage, and $h(x)\ mod\ m$ to represent the modulo stage.

B. One-Hashing Bloom Filter Design

Figure 2 shows the structure of OHBF. Instead of treating the entire filter memory as one bit vector as in SBF (shown in Figure 1), OHBF divides the bit vector into k partitions, where k is the number of hash functions in SBF. Note that SBF first uses different hash functions to get k machine words and then uses these machine words to modulo the Bloom filter size m. Each result can address the entire filter space. In contrast, OHBF first uses just one base hash function to get a machine word and then uses it to modulo each partition's size. Each result can only address one bit in the corresponding partition. Different from the previous even partitioning methods, the partitions in OHBF are purposely made uneven. The reason is explained in Section III-C. We use an example to illustrate the mechanism of OHBF. Let $m_i, 1 \le i \le k$ denotes the i^{th} partition size of OHBF. We have $m = \Sigma_{i=1}^{k} m_i$. Suppose $k = 3$, $m_1 = 11$, $m_2 = 13$, and $m_3 = 15$. When an element e comes, we apply the base hash function $h(\cdot)$ and suppose $h(e) = 4201$. As $h(e)\ mod\ m_1 = 10$, $h(e)\ mod\ m_2 = 2$, and $h(e)\ mod\ m_3 = 1$, the corresponding $10th$, $2nd$, and $1st$ bit of each partition is set.

OHBF has a simple structure and is easy to be implemented. OHBF uses extra modulo operations instead of traditional hash functions in SBF. As CPU's ALU has optimized modulo execution units, the modulo operation has smaller computational cost than hash computing, which consumes considerable CPU clock cycles per element. OHBF reduces the hash computational cost to nearly $1/k$ of SBF. However, to make OHBF have the similar false positive probability with SBF, it has to meet the following requirements:

- The partition algorithm needs to ensure the modulo operation to generate independent values. This is proved by theoretical analysis. With this property, we only need to find one good base hash function. (Section III-C)
- The resulting false positive ratio of OHBF should be close enough to the equivalent SBF. The false positive ratio change due to partitions should be small enough. (Section III-D)
- The sum of the partition sizes is close enough to the filter memory constraint. As the final size probably has a deviation from the target size, we expect the gap between them is minor enough, without noticeably affecting the target memory consumption. (Section III-E)

All these requirements can be satisfied by our scheme. Actually, we expect the OHBF scheme to be a practical substitute for SBF. All the parameters of OHBF should be nearly the same as SBF. Thus, we could use OHBF, anywhere Bloom filters are needed, to reduce the computational cost and improve the system performance.

C. Proof of Hashing Independence in Modulo Stage

The uneven partitioning method in OHBF would produce independent modulo results if partition sizes are pairwise relatively prime. Let $g_i(x) = h(x)\ mod\ m_i,\ 1 \le i \le k$. We claim that in the modulo stage of OHBF hashing, $g_1(x), g_2(x) \ldots, g_k(x)$ are pairwise independent if the size of each partition satisfies:

$$(m_i, m_j) = 1, \quad 1 \le i < j \le k \tag{3}$$

where (m_i, m_j) means the greatest common divisor of two integers m_i and m_j. Such m_i and m_j are also called relatively prime.

Before we prove our claim, we need to prove two lemmas first. To facilitate the proof, we borrow some notations from *Number Theory*, as shown in Table I. The symbols a, b, q represent non-negative integers.

TABLE I
NOTATIONS

$a \vert b$	a divides b
(a,b)	the greatest common divisor of a and b
$a \equiv b\ mod\ q$	a and b are congruent modulo q
$a \not\equiv b\ mod\ q$	a and b are incongruent modulo q

Lemma 1: If two integers $a, b \in [0, q-1]$, where $a \neq b$, $(p, q) = 1$, then $ap \not\equiv bp\ mod\ q$.

Proof: [Proof by Contradiction] Suppose that $ap \equiv bp\ mod\ q$. Let us assume $a < b$. Then $q|(bp - ap)$, which means $q|(b-a)p$. By definition $(p, q) = 1$, we can derive $q|(b-a)$. But this is impossible because $1 \le b - a < q$. Therefore, the supposition does not hold and the statement is true. □

Lemma 2: Let $\mathcal{Z}$ denote a uniformly distributed non-negative integer random variable over range $[0, rpq-1]$, where $r, p, q \in \mathbb{Z}^+$. Let $\mathcal{X} = (\mathcal{Z}\ mod\ p)$ and $\mathcal{Y} = (\mathcal{Z}\ mod\ q)$, where $(p, q) = 1$. Then $\mathcal{X}$, $\mathcal{Y}$ are mutually independent random variables.

Proof: Obviously, $\mathcal{X} \in [0, p-1]$, $\mathcal{Y} \in [0, q-1]$. Let us assume $\mathcal{X} = a$, $a \in [0, p-1]$. Then, by definition, $\mathcal{Z} = cp + a$, where $c \in [0, rq-1]$. Let $\mathcal{Z}_a$ denote $\{\mathcal{Z} | \mathcal{Z} = cp + a, c \in [0, rq-1]\}$, $\mathcal{Z}_a^d$ denote $\{\mathcal{Z} | \mathcal{Z} = cp + a, c \in [dq, dq+q-1], 0 \le d \le r-1\}$. Then $\mathcal{Z}_a = \bigcup_{d=0}^{r-1} \mathcal{Z}_a^d$.

We first consider $c \in [0, q-1]$. By Lemma 1, we know that the q remainders $\mathcal{Z}_a^0\ mod\ q$ are not equal to each other. Note that the q remainders range in $[0, q-1]$ and they are not equal to each other, then we can say that $\mathcal{Z}_a^0\ mod\ q$ is uniformly distributed in the range $[0, q-1]$.

Then we consider $c \in [dq, dq+q-1]$, $1 \le d \le r-1$. Because $(cp + a) \equiv ((c\ mod\ q)p + a)\ mod\ q$, so the q remainders $\mathcal{Z}_a^d\ mod\ q$ are equal to $\mathcal{Z}_a^0\ mod\ q$. Therefore, $\mathcal{Z}_a^d\ mod\ q$ are also uniformly distributed in the range $[0, q-1]$.

Consequently, we can conclude that $\mathcal{Z}_a\ mod\ q$ are uniformly distributed in $[0, q-1]$. That is, $Pr(\mathcal{Y} = b | \mathcal{X} = a) = Pr(\mathcal{Y} = b) = 1/q$, where $a \in [0, p-1]$, $b \in [0, q-1]$. Similarly, we can obtain that $Pr(\mathcal{X} = a | \mathcal{Y} = b) = Pr(\mathcal{X} = a) = 1/p$. Successfully, we prove that $\mathcal{X}$, $\mathcal{Y}$ are mutually independent random variables. □

Theorem 1: Suppose that the machine word output, $\mathcal{M} = h(x)$, is uniformly distributed over $[0, rm_1m_2\ldots m_k - 1]$. If the partition sizes $m_1, m_2, \ldots, m_k$ are pairwise relatively prime, then $g_1(x), g_2(x), \ldots, g_k(x)$ are pairwise mutually independent random variables.

Proof: Let us assume an arbitrary pair (i, j) which satisfies that $1 \le i < j \le k$. Let $s = rm_1m_2\ldots m_k/m_im_j$, then we know that $\mathcal{M}$ is uniformly distributed over $[0, sm_im_j - 1]$. Since $(m_i, m_j) = 1$ by definition, we get that $g_i(x)$ and $g_j(x)$ are mutually independent random variables by Lemma 2. As (i, j) is selected arbitrarily, we can conclude that the modulo stage results $g_1(x), g_2(x), \ldots, g_k(x)$ are pairwise mutually independent random variables. □

In Theorem 1, we assume that $\mathcal{M}$ covers a range which is a multiple of the product $m_1m_2\ldots m_k$. However, in practice, $\mathcal{M}$ usually covers the range a power of 2, *i.e.*, $|\mathcal{M}| = 2^L$, where L is the machine word bit width. The result that $|\mathcal{M}|$ modulo $m_1m_2\ldots m_k$ may not be zero, which makes Theorem 1 inapplicable. There exists a simple method to solve the problem. We can discard the redundant numbers by restricting the range to $[0, c]$, where $c = |\mathcal{M}| - |\mathcal{M}|\%(m_1 m_2 \dots m_k)$. This implies that $|\mathcal{M}| > m_1 m_2 \dots m_k$. If $|\mathcal{M}|$ is far greater than the product $m_1 m_2 \dots m_k$, *i.e.*, $|\mathcal{M}| \gg m_1 m_2 \dots m_k$, then the modulo part can be ignored in practice.

We have proved that the outputs of the modulo stage are mutually independent, on one condition that the partition sizes are pairwise relatively prime. This result guarantees that we eliminate the correlation of hash functions theoretically. With this property, we only need to find one good base hash function to implement OHBF.

If the partition sizes are not pairwise relatively prime, the modulo results in the modulo stage would have some degree of correlation. For example, assume the OHBF has two partitions with length $m_1 = 2, m_2 = 4$, then we can get $g_1(x) = 0$ when $g_2(x) = 0$ or 2, $g_1(x) = 1$ when $g_2(x) = 1$ or 3. An extreme example is that when $m_1 = m_2 = \cdots = m_k$, the OHBF would degrade to a Bloom filter with effective length m_i and only one effective hash function, due to $g_1(x) = g_2(x) = \cdots = g_k(x)$. Therefore, the correlation of hashing results in the modulo stage leads to memory waste and will degrade the Bloom filter's performance. In the following sections, we assume the partition sizes are pairwise relative prime if not specifically defined.

D. False Positive Analysis

The false positive of OHBF is caused by two factors. The first factor is the hashing collision in the hash stage, denoted as event $\mathcal{E}$. If the machine words collide, it will definitely cause false positive. The second factor is the modulo collision in the modulo stage. If the machine words from the hash stage do not collide but all the modulo remainders happen to collide, this will also cause false positive. Then the total false positive probability is:

$$\begin{aligned} f_o = Pr(\mathcal{F}) &= Pr(\mathcal{F}|\mathcal{E})Pr(\mathcal{E}) + Pr(\mathcal{F}|\neg\mathcal{E})Pr(\neg\mathcal{E}) \\ &= Pr(\mathcal{E}) + Pr(\mathcal{F}|\neg\mathcal{E})(1 - Pr(\mathcal{E})) \end{aligned} \tag{4}$$

Suppose the machine word has L bits and L^e effective bits, where $L^e = log_2(2^L - 2^L\%m_1 m_2 \dots m_k)$ according to Theorem 1. A specific machine word will be selected with probability $\frac{1}{2^{L^e}}$, and not be selected with probability $1 - \frac{1}{2^{L^e}}$. After n elements are inserted, the probability that a specific machine word has not been hashed is $\left(1 - \frac{1}{2^{L^e}}\right)^n$, which implies that the machine word collision probability is:

$$Pr(\mathcal{E}) = 1 - \left(1 - \frac{1}{2^{L^e}}\right)^n \tag{5}$$

The analysis of the second factor is similar to the machine word collision analysis. We can conclude that the false positive ratio caused by the second factor is:

$$Pr(\mathcal{F}|\neg\mathcal{E}) = \prod_{i=1}^{k} (1 - (1 - 1/m_i)^n) \tag{6}$$

Typically the machine word range (*e.g.* 2^{32} or 2^{64}) is far greater than the size of Bloom filter, *i.e.*, $2^L \gg m$. A collision of machine word does not likely happen as long as the machine word space is large enough, so $Pr(\mathcal{E})$ in practice is nearly 0. Therefore, the false positive probability of OHBF is simplified to be:

$$Pr(\mathcal{F}) \approx Pr(\mathcal{F}|\neg\mathcal{E}) = \prod_{i=1}^{k} (1 - (1 - 1/m_i)^n) \tag{7}$$

Because the function $\left(1 - \left(1 - \frac{1}{x}\right)^n\right)$ with respect to x is a monotonically decreasing function, we have

$$(1 - (1 - M_1)^n)^k \le f_o \le (1 - (1 - M_2)^n)^k \tag{8}$$

where $M_1 = \frac{1}{\max_i \{m_i\}}$ and $M_2 = \frac{1}{\min_i \{m_i\}}$.

Further, we can obtain the following theorem.

Theorem 2: The false positive ratio of OHBF can be estimated by the following inequality,

$$\begin{aligned} f_o &\le \left(1 - \left(\sqrt[k]{\prod_{i=1}^{k} (1 - 1/m_i)}\right)^n\right)^k \\ &\approx \left(1 - \sqrt[k]{\prod_{i=1}^{k} e^{-\frac{n}{m_i}}}\right)^k \end{aligned} \tag{9}$$

Proof: Making use of the well-known mathematical property that the *arithmetic mean* is greater than or equal to the *geometric mean*, we can derive:

$$\begin{aligned} f_o &= \prod_{i=1}^{k} (1 - (1 - 1/m_i)^n) \\ &\le \left(\frac{1}{k}\sum_{i=1}^{k} (1 - (1 - 1/m_i)^n)\right)^k \\ &= \left(1 - \frac{1}{k}\sum_{i=1}^{k} (1 - 1/m_i)^n\right)^k \\ &\le \left(1 - \sqrt[k]{\prod_{i=1}^{k} (1 - 1/m_i)^n}\right)^k \\ &= \left(1 - \left(\sqrt[k]{\prod_{i=1}^{k} (1 - 1/m_i)}\right)^n\right)^k \\ &\approx \left(1 - \sqrt[k]{\prod_{i=1}^{k} e^{-\frac{n}{m_i}}}\right)^k \end{aligned}$$

□

f_o has the similar form as SBF's false positive probability $f_s = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{n}{m/k}}\right)^k$. Note that $\sqrt[k]{\prod_{i=1}^{k} e^{-\frac{n}{m_i}}}$ is the *geometric mean* of $e^{-\frac{n}{m_i}}$. This suggests that the partition sizes should be very close to each other. If the distribution of m_i is near m/k and m is large enough, f_o will be very close to f_s.

Table II shows the theoretical false positive probability comparison between SBF and OHBF. It can be seen that the OHBF's false positive ratio is very close to SBF's. As OHBF's partition size must satisfy Equation 3, the total size of OHBF may have a difference from the target filter size. If the difference is minor enough, the OHBF mechanism still works. Algorithm 1 described in the next subsection is used to divide the partitions.

Algorithm 1 Determine the Sizes of Partitions

```
Input: $m_p,\ k,\ pTable$
Output: $m_f,\ partLen$
1: scan $pTable$ to find the prime closest to $\lfloor m_p/k \rfloor$ and
      denote its index in $pTable$ as $pdex$
2: $sum \leftarrow 0$; $m_f \leftarrow 0$
3: for $i \leftarrow pdex - k + 1$ to $pdex$ do
4:     $sum \leftarrow sum + pTable[i]$
5: end for
6: $dif1 \leftarrow m_p - sum$; $j \leftarrow pdex + 1$
7: $sum \leftarrow sum + pTable[j] - pTable[j-k]$
8: $dif2 \leftarrow m_p - sum$
9: while $dif2 < dif1$ do
10:     $dif1 \leftarrow dif2$; $j \leftarrow j + 1$
11:     $sum \leftarrow sum + pTable[j] - pTable[j-k]$
12:     $dif2 \leftarrow abs(sum - m_p)$
13: end while
14: for $i \leftarrow 1$ to $k$ do
15:     $partLen[i] \leftarrow pTable[j - k + i]$
16:     $m_f \leftarrow m_f + partLen[i]$
17: end for
```

TABLE II
THEORETICAL FALSE POSITIVE PROBABILITY COMPARISON BETWEEN SBF AND OHBF

m	$n = 1000,\ k = 3$		
	f_s	f_o	$\lvert f_o - f_s\rvert / f_s$ (%)
10003	1.7399 e-2	1.7404 e-2	0.026
19993	2.7054 e-3	2.7058 e-3	0.014
29989	8.6273 e-4	8.6281 e-4	0.010
39995	3.7740 e-4	3.7743 e-4	0.007
49991	1.9761 e-4	1.9762 e-4	0.006
m	$n = 1000,\ k = 10$		
	f_s	f_o	$\lvert f_o - f_s\rvert / f_s$ (%)
10012	1.0118 e-2	1.0149 e-2	0.308
19986	8.9441 e-5	8.9612 e-5	0.192
30034	3.3187 e-6	3.3238 e-6	0.154
39994	2.8084 e-7	2.8116 e-7	0.113
49988	3.8390 e-8	3.8424 e-8	0.086

E. Determine the Sizes of Partitions

From the previous analysis, we have known that the partition algorithm should meet the following two requirements:

- The sizes of partitions must satisfy $(m_i, m_j) = 1,\ 1 \le i < j \le k$. Only by ensuring this, can we guarantee that the outputs of the modulo stage in OHBF are mutually independent.
- The sizes of partitions are close to each other with small deviations. According to the false positive analysis of OHBF, this has a direct impact to f_o.

We provide a simple algorithm to satisfy these requirements: just pick k consecutive primes as the size of the partitions. We first build a prime table. The maximum prime in the table can be determined by demand. Our experience shows that in most cases it should be around $m/k + \delta$, where $\delta < 300$. Note that taking consecutive primes as the partition sizes is not necessary. We can find more simple partition method in practice.

The sums of these consecutive primes are discrete. Given a planned overall size m_p for a Bloom filter, we usually cannot

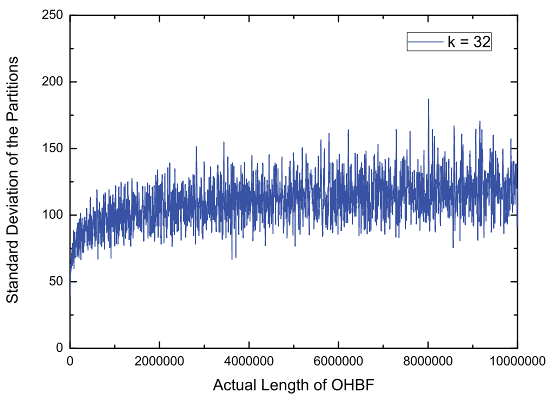


Fig. 3. Standard deviation of the partition sizes when $k = 32$.

get k prime numbers to make their sum m_f be exactly m_p. As long as the difference between m_p and m_f is small enough, it neither causes any trouble for the software implementation nor noticeably shifts the false positive ratio.

We refer to $pTable$ as our prime table, and $pTable[i]$ is the i^{th} prime number in $pTable$. The primes in $pTable$ are consecutive primes in ascending order. Refer k as the number of hash functions in standard Bloom filter and refer $partLen[i]$ as the i^{th} partition size. We determine the size of each partition and the overall size by Algorithm 1.

Table III shows some partition examples using Algorithm 1. It can be seen that the difference ($\frac{|m_f - m_p|}{m_p}$) between the planned size and the actual size is very small, and the size of each partition is very close. We also scan the difference between m_p and m_f when $k = 32$ and $m_p < 10{,}000{,}000$, the result shows that the biggest difference between them is only 315. The standard deviation of OHBF partition sizes when $k = 32$ is shown in Figure 3. These results illustrate that our partition algorithm can meet all the OHBF design requirements.

The partition sizes, as the metadata of OHBF, incur some space cost, denoted as C. We can get:

$$C = \sum_{i=1}^{k} \log m_i \approx k \log \frac{m}{k} \tag{10}$$

In general cases, the space cost of partition sizes is negligible, compared to the Bloom filter space cost m. For example, as the configurations in Table II, $C = 12$ when $m = 10003, k = 3$ and $C = 100$ when $m = 10012, k = 10$. Only if the space cost of C is comparable to m, the false positive probability of OHBF should be reconsidered.

IV. PRACTICAL BLOOM FILTER ANALYSIS

An SBF needs k hash functions, an LHBF needs 2 hash functions, and an OHBF needs just one hash function. Through the theoretical analysis of OHBF, people may get the impression that we reduce the number of hash functions at the cost of higher false positive ratio. However, in practice, OHBF is most likely to present lower false positive ratio than SBF and LHBF. This can be explained by the fact that the practical hash functions are far worse than truly random hash functions.

All the Bloom filter analysis is based on two main assumptions on the hash functions: *1) randomness*, all the hash

TABLE III
EXAMPLES OF PARTITIONS USING ALGORITHM 1

m_p	m_f	$\frac{\lvert m_f - m_p \rvert}{m_p}$	Size of Each Partition for m_f $(k = 10)$									
10000	10012	1.2 e-3	971	977	983	991	997	1009	1013	1019	1021	1031
20000	19986	7.0 e-4	1973	1979	1987	1993	1997	1999	2003	2011	2017	2027
40000	39994	1.5 e-4	3947	3967	3989	4001	4003	4007	4013	4019	4021	4027
80000	80044	5.5 e-4	7949	7951	7963	7993	8009	8011	8017	8039	8053	8059
160000	159990	6.3 e-5	15937	15959	15971	15973	15991	16001	16007	16033	16057	16061
320000	319984	5.0 e-5	31957	31963	31973	31981	31991	32003	32009	32027	32029	32051
640000	640024	3.8 e-5	63929	63949	63977	63997	64007	64013	64019	64033	64037	64063
1280000	1280084	6.6 e-5	127931	127951	127973	127979	127997	128021	128033	128047	128053	128099

functions used in Bloom filters map data elements uniformly to the range, and *2) independence*, all the hash functions used in Bloom filters map data elements to the range independently. In this section, we test some representative practical hash functions on their randomness and independence. The *chi-squared test*, a well-known method in *hypothesis testing*, will be used. The details of *chi-squared test* can be found in [38]. Next, we will model the randomness test and independence test to their corresponding *chi-squared test*.

A. Randomness Test of Single Hash Function

The hash functions used in Bloom filters will first map an element to a machine word of L bits. If we directly evaluate whether the hash value is uniform distribution or not, the sample space size will be 2^L. The huge size sample space will make the evaluation process complex. Alternatively, to simplify the problem, we convert the hash value's uniform distribution test into the hash value bits sum's *binomial* distribution test.

If a hash function is truly random, then each bit of the hash value X should follow *0-1* distribution, *i.e.*, $Pr(x_i = 0) = 1/2, Pr(x_i = 1) = 1/2, 1 \leq i \leq L$. And the sum of all bits, $H_X = \sum_{i=1}^{L} x_i$, should follow *binomial* distribution. The probability distribution of H_X is:

$$Pr(H_X = i) = \frac{C(L, i)}{2^i}, \quad 0 \leq i \leq L \tag{11}$$

So the null hypothesis in this hypothesis test is:

H0: *The sum of hash value bits, H_X, follows a binomial distribution, $Pr(H_X = i) = \frac{C(L,i)}{2^i}$*

The test statistic S_X in the randomness test is:

$$S_X = \sum_{i=0}^{L} \frac{\left(O(h_X^i) - E(h_X^i)\right)^2}{E(h_X^i)} \tag{12}$$

where $O(h_X^i)$ is the observed frequency when $H_X = i$, $E(h_X^i)$ is the expected frequency when $H_X = i$.

As the binomial distribution has $L+1$ values, the degrees of freedom of χ^2 distribution are L. Hence, the rejection region in randomness test is:

$$S_X \geq \chi^2_{\alpha,L} \tag{13}$$

B. Independence Test Between Different Hash Functions

Now we test the independence between two hash functions. We call every two hash functions a hash function *pair*.

Similar to the last subsection, we convert the independence test into the binomial goodness-of-fit test. If a pair of hash functions are independent, then the corresponding hash values X and Y would be independent. Therefore, the *exclusive-or* operation result, $Z = X \oplus Y$, would be a uniformly distributed variable. So each bit of Z would be 0-1 distribution with $Pr(z_i = 0) = 1/2, Pr(z_i = 1) = 1/2$, where $z_i = x_i \oplus y_i, 1 \leq i \leq L$. The sum of each bit of Z, $H_Z = \sum_{i=1}^{L} z_i$, would follow the binomial distribution.

$$Pr(H_Z = i) = \frac{C(L, i)}{2^i}, \quad 0 \leq i \leq L \tag{14}$$

So the null hypothesis in this independence test is:

H0: *The sum of all bits of Z, H_Z, follows a binomial distribution, $Pr(H_Z = i) = \frac{C(L,i)}{2^i}$*

The test statistic S_Z in the independence test is:

$$S_Z = \sum_{i=0}^{L} \frac{\left(O(h_Z^i) - E(h_Z^i)\right)^2}{E(h_Z^i)} \tag{15}$$

And, the rejection region in independence test is:

$$S_Z \geq \chi^2_{\alpha,L} \tag{16}$$

TABLE IV
COLLECTED HASH FUNCTIONS (REFERENCES [39]) AND THEIR NUMBER

APHash h1	BKDR h2	BOB h3	CRC32 h4	DEKHash h5
DJBHash h6	FNV32 h7	Hsieh h8	JSHash h9	OCaml h10
OAAT h11	PJWHash h12	RSHash h13	SBOX h14	SDBM h15
Simple h16	SML h17	STL h18	MD5 h19	SHA-1 h20

C. Hash Function Collection and Test

We collect a total of 20 hash functions, consisting of 18 non-cryptographic hash functions and 2 cryptographic hash functions (MD5 and SHA-1). The hash functions are shown in Table IV. The name and source of these hash functions mainly refer to [39]. All the hash values are 32 bits, *i.e.*, $L = 32$. As the standard MD5 and SHA-1 hash values are 128 and 160 bits respectively, we convert them to 32 bits by *exclusive-or* the hash values every 32 bits. The significance level is set to be $\alpha = 0.05$. So the rejection region in both the randomness test and the independence test is $S \geq \chi^2_{0.05,32} = 46.194$.

We use the real-world Internet traces, obtained from CAIDA [40], to evaluate the hash functions. The trace is extracted from an OC-192 link and lasts 60 minutes. It contains 2G packets, 5M different destination IP addresses,

TABLE V
RANDOMNESS TEST OF SINGLE HASH FUNCTION

Function	h1	h2	h3	h4	h5	h6	h7	h8	h9	h10	h11	h12	h13	h14	h15	h16	h17	h18	h19	h20
key_len = 4	−	−	+	+	−	−	−	−	−	−	−	−	−	+	−	−	−	−	+	+
key_len = 13	−	+	+	+	−	−	+	−	−	−	+	−	+	+	−	−	+	−	−	+

TABLE VI
INDEPENDENCE TEST BETWEEN DIFFERENT HASH FUNCTIONS, THE UPPER TRIANGULAR MATRIX CORRESPONDING TO KEY_LEN=4, THE LOWER TRIANGULAR MATRIX CORRESPONDING TO KEY_LEN=13

Function	h1	h2	h3	h4	h5	h6	h7	h8	h9	h10	h11	h12	h13	h14	h15	h16	h17	h18	h19	h20
h1	○	−	+	+	−	−	−	+	−	−	+	−	−	+	−	−	−	−	+	+
h2	+	○	+	+	−	−	−	−	−	−	−	−	−	+	−	−	−	−	+	+
h3	+	+	○	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
h4	+	+	+	○	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
h5	+	+	+	+	○	−	−	+	−	−	+	−	−	+	−	−	−	−	+	+
h6	+	−	+	+	+	○	−	−	−	−	+	−	−	+	−	−	−	−	+	+
h7	+	−	+	+	+	−	○	+	−	−	−	−	−	+	−	−	−	−	+	+
h8	+	+	+	+	+	+	+	○	−	−	+	−	+	+	+	−	−	−	+	+
h9	+	+	+	+	+	+	+	+	○	−	+	−	−	+	−	−	−	−	+	+
h10	+	−	+	+	−	−	−	+	+	○	−	−	−	+	−	−	−	−	+	+
h11	+	+	+	+	+	+	+	+	−	+	○	−	−	+	−	−	+	−	+	+
h12	+	+	−	+	−	+	+	−	+	+	+	○	−	+	−	−	−	−	+	+
h13	+	−	+	+	+	−	−	+	+	−	+	+	○	+	−	−	−	−	+	+
h14	+	+	+	+	+	+	+	+	+	+	+	+	+	○	+	+	+	+	+	+
h15	−	−	+	−	+	−	−	+	+	−	+	+	−	+	○	−	−	−	+	+
h16	+	−	+	−	+	−	−	+	+	−	+	+	−	+	−	○	−	−	+	+
h17	+	−	+	−	+	−	−	+	+	−	−	+	−	+	−	−	○	−	+	−
h18	+	−	+	+	+	−	−	+	+	−	+	+	−	+	−	−	−	○	+	+
h19	+	+	+	+	+	+	+	+	+	−	+	−	+	+	−	+	+	+	○	+
h20	+	+	−	+	+	+	+	−	+	+	+	+	+	+	+	+	+	+	−	○

and 50M flows. Because the input key length can affect the evaluation of hash functions, we use two kinds of keys to evaluate these hash functions. The first kind of keys is the 4-byte destination IP address and the second kind of keys is the 13-byte 5-tuple IP header.

Table V is the randomness test result of each single hash function. The notation '$+$' represents that we accept the assumption and the notation '$-$' represents that we reject the assumption. We can see that most of the non-cryptographic hash functions cannot pass through the chi-squared test, and some hash functions, such as BKDR, FNV32 and OAAT, perform better randomness property when the input keys are longer. The two cryptographic hash functions present good randomness property, just in accordance with our instinct.

Table VI is the independence test result of hash function pairs. The notation '$+$' and '$-$' represent that we accept and reject the assumption respectively. The notation '$\circ$' means the test does not apply. It can be concluded that many hash function pairs demonstrate some correlation relationship, especially for two hash functions both with poor randomness property. The correlation could result in large deviation from desired false positive probability. It also makes the choice for hash function combinations difficult.

D. Discussion

The hash functions used by Bloom filters are not truly random in practice. First, many hash functions cannot be regarded as uniformly distributed. Second, many hash function pairs have some degree of correlation. Therefore, hash function selection when implementing a Bloom filter is difficult, especially for large k. A poor selection may lead to big false positive deviation from the theoretical value, just as the statement in [37]. Although the same good hash function with different initial seeds mostly show good independence property, the use of hash functions in this way still incurs the multiplied computational cost. On the contrary, OHBF requires only one good base hash function to implement a Bloom filter. The fewer practical hash functions we need, the easier we can make the right selection. Moreover, the actual false positive ratio would also be closer to the theoretical value since the possible correlation of hash functions is eliminated in OHBF.

V. BLOOM FILTER EVALUATIONS

We evaluate the OHBF scheme from the following three aspects: *1)* the cost of modulo operation, *2)* practical false positive ratio and *3)* querying time.

We compare different Bloom filter implementations on a commodity server with an Intel Xeon CPU E5645$\times$2 (6 cores$\times$2 threads, 2.4GHz) and 48GB DDR3 (1,333MHz, ECC) memory. The server runs an OS Linux 2.6.43 kernel (x86_64). we use the C++ Programming Language to implement the programs. The experiments use the same real-world trace as that used in Section IV-C. We use the well-known *Sieve of Eratosthenes* algorithm to get the pTable(prime number table). The pTable takes 2.6 Mbytes.

A. The Cost of Modulo Operation

The cost of the modulo operation cannot be ignored. However, it only applies on the (fixed-size) output of hash function. We test the modulo operation on our server and find each operation needs 7.3 clock cycles on average. In contrast, the cost of hash functions is directly proportional to the element size. For example, CRC32, MD5, SHA-1 needs 6.9, 6.8, 11.4 clock cycles per byte, respectively [10].

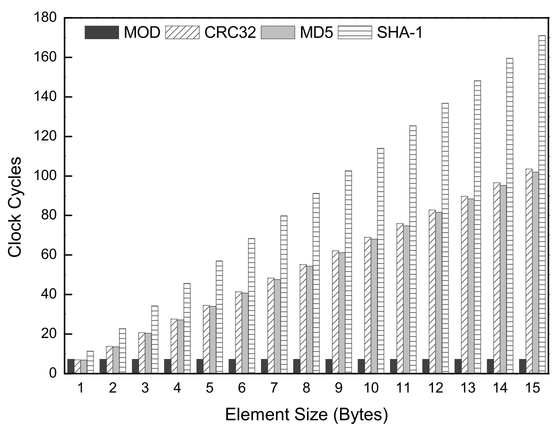


Fig. 4. The cost comparison between fixed-size-input modulo operation and variable-size-input hash functions.

TABLE VII
FALSE POSITIVE RATIO COMPARISON ON KEY_LEN= 4 bytes, $k = 3$, $n = 1000$

m	SBF Theory	Difference Compared to SBF Theory (%)		
		OHBF Theory	OHBF Sim	LHBF Sim
10003	1.740 e-2	0.026	0.029	0.046
11003	1.359 e-2	0.024	0.057	0.021
11993	1.084 e-2	0.022	0.016	0.057
13003	8.747 e-3	0.021	0.001	0.014
13993	7.186 e-3	0.019	0.038	0.036
14995	5.962 e-3	0.018	0.014	0.169
16003	4.996 e-3	0.017	0.105	0.171
17011	4.227 e-3	0.017	0.012	0.138
18005	3.616 e-3	0.016	0.006	0.147
19009	3.112 e-3	0.015	0.117	0.049

The cost comparison between the (fixed-size-input) modulo operation and the (variable-size-input) hash functions is shown in Figure 4. Compared to common hash functions, one modulo operation is fast enough. Intuitively, OHBF will have better performance gain over SBF as the element size increases or the number of hash functions increases. This is confirmed by our experiments in the following subsections.

B. False Positive Evaluation

We compare three Bloom filter implementations: SBF, LHBF, and OHBF. In the experiments, we set $n = 1000$. All the hash functions are selected from Table IV, and their outputs are 64 bits. We do not run the experiments on SBF but use its ideal case as benchmark for comparison. LHBF employs the extended double hashing cube scheme discussed in [32]. The two hash functions for LHBF are MD5 and SHA-1. The base hash function for OHBF is MD5. We implement the experiments on both destination IPs (key_len=4) and 5-tuple flow identifiers (key_len=13).

The results are shown in Table VII to X. The '*difference*' in the tables means a difference ratio. For example, the '*OHBF theory*' column is calculated by $\frac{|OHBF\ theory - SBF\ theory|}{SBF\ theory}$. From the four tables, we can see that both the theoretical and practical false positive ratios of OHBF are very close to SBF. The small instability of OHBF comes from the high requirement of the base hash function. Since OHBF heavily relies on the randomness of the base hash function, the pseudo-random hash functions in practice would lead to the jitter of false positive performance. In [32], LHBF is claimed to have the same asymptotic false positive ratio as SBF when m/n is constant and $n \to \infty$. However, in practice, the false positive ratio difference between LHBF and SBF can be significant. Table X shows that the practical false positive ratio of LHBF is much higher than SBF's theoretical false positive ratio, especially when the theoretical false positive probability is low. The reason for its poor false positive ratio is twofold. First, the synthetic hash functions for LHBF have some degree of correlation (discussed in Section II). Second, the element number usually is not very large in practice (*e.g.*, $n = 1000$ in our settings).

According to the analysis of hashing independence in Section III-D, we know that the hash function needs at least 38, 40, 109, and 115 hash bits for experiments corresponding

TABLE VIII
FALSE POSITIVE RATIO COMPARISON ON KEY_LEN= 13 bytes, $k = 3$, $n = 1000$

m	SBF Theory	Difference Compared to SBF Theory (%)		
		OHBF Theory	OHBF Sim	LHBF Sim
10003	1.740 e-2	0.026	0.089	0.119
11993	1.084 e-2	0.022	0.027	0.116
13993	7.188 e-3	0.019	0.017	0.101
16003	4.996 e-3	0.017	0.058	0.009
18005	3.616 e-3	0.016	0.035	0.005
19993	2.706 e-3	0.014	0.014	0.138
22013	2.068 e-3	0.013	0.057	0.173
24013	1.620 e-3	0.012	0.005	0.192
26009	1.293 e-3	0.011	0.092	0.205
28001	1.049 e-3	0.010	0.063	0.134

TABLE IX
FALSE POSITIVE RATIO COMPARISON ON KEY_LEN= 4 bytes, $k = 10$, $n = 1000$

m	SBF Theory	Difference Compared to SBF Theory (%)		
		OHBF Theory	OHBF Sim	LHBF Sim
10012	1.012 e-2	0.308	0.165	0.489
11018	5.706 e-3	0.287	0.520	0.863
11978	3.379 e-3	0.294	0.267	0.678
12990	1.991 e-3	0.250	0.356	1.609
14002	1.200 e-3	0.294	0.270	0.900
14968	7.554 e-4	0.244	0.495	1.351
16000	4.701 e-4	0.227	0.211	1.715
16974	3.059 e-4	0.243	0.017	2.282
18008	1.974 e-4	0.240	0.377	2.584
18974	1.331 e-4	0.217	0.451	2.721

TABLE X
FALSE POSITIVE RATIO COMPARISON ON KEY_LEN= 13 bytes, $k = 10$, $n = 1000$

m	SBF Theory	Difference Compared to SBF Theory (%)		
		OHBF Theory	OHBF Sim	LHBF Sim
10012	1.012 e-2	0.308	0.212	0.489
11978	3.379 e-3	0.294	0.063	0.629
14002	1.200 e-3	0.294	0.507	1.096
16000	4.701 e-4	0.227	0.127	1.627
18008	1.974 e-4	0.240	0.457	2.668
19986	8.944 e-5	0.191	0.476	3.969
21956	4.295 e-5	0.229	0.057	5.915
24010	2.104 e-5	0.171	0.054	9.452
25998	1.102 e-5	0.185	0.222	14.93
28014	5.946 e-6	0.154	0.033	23.21

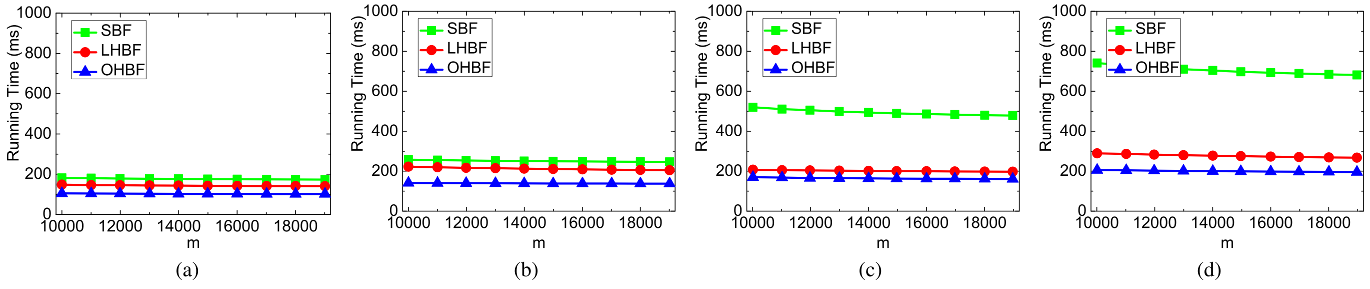


Fig. 5. Querying time of the three Bloom filters, with $n = 1000$, the size m varying in each subfigure. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment. (a) key_len = 4 bytes, $k = 3$. (b) key_len = 13 bytes, $k = 3$. (c) key_len = 4 bytes, $k = 10$. (d) key_len = 13 bytes, $k = 10$.

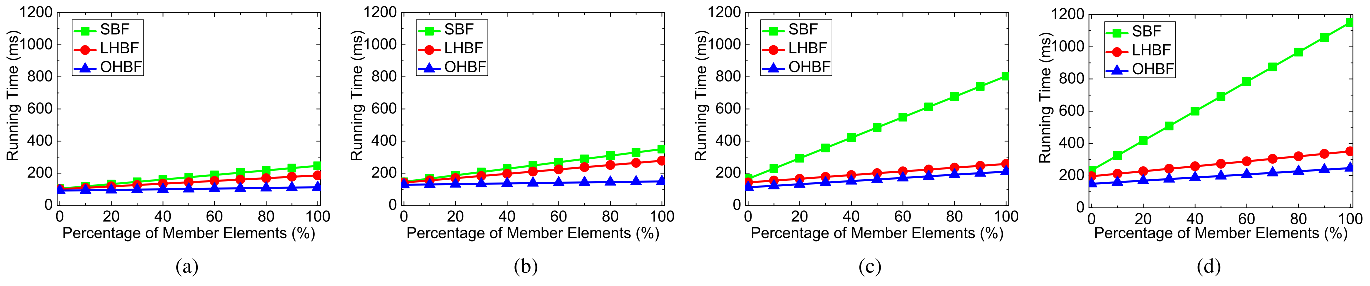


Fig. 6. Querying time of the three Bloom filters, with $m = 16000$, $n = 1000$, the percentage of member elements varying in each subfigure. Each point in this figure is the mean of 1,000 experiments. We implement 1,000,000 queries in each experiment. (a) key_len = 4 bytes, $k = 3$. (b) key_len = 13 bytes, $k = 3$. (c) key_len = 4 bytes, $k = 10$. (d) key_len = 13 bytes, $k = 10$.

to Table {VII, VIII, IX, X} respectively for OHBF. Note that the hash outputs are all set to be 64 bits, which means that the latter two experiments (Table {IX, X}) do not strictly satisfy the independence requirement for OHBF. The results in Table {IX, X} show that the practical false positive ratios of OHBF are still very close to their theoretical value. Despite the lack of theoretical analysis, these results tell us that OHBF has good scalability for large k, where large number of hash bits are needed.

C. Querying Time Evaluation

In this subsection, we evaluate the querying speed of different Bloom filter implementations. To give a fair comparison for different Bloom filter implementations, all the hash functions are BOB-based hash function with different seeds. In the following discussion, a member element represents that the element is in the programmed element set, and a non-member element represents the opposite.

Figure 5 shows the querying time comparison when the size of Bloom filters varies. The search elements are composed of 50% member elements and 50% non-member elements. We can see from the four figures that OHBF takes the least time for querying. As the key length increases, the querying time gap between OHBF and SBF (or LHBF) increases. This is because when the key becomes longer, more time is consumed for hash function computation in SBF (or LHBF). As the hash function number k increases, the querying time gap between OHBF and LHBF increases slowly, but the querying time gap between OHBF and SBF increases quickly. The reason is that LHBF reduces the hash computational cost to (approximate) $2/k$, and OHBF reduces the hash computational cost to (approximate) $1/k$. As the Bloom filter size m increases, the querying time of the same Bloom filter presents a slightly decreasing trend. This is because the non-member elements tend to terminate the search earlier when the false positive ratio is lower.

Note that not all the hash functions are needed when querying a non-member element. If the current querying bit is 0, we do not need to query the subsequent bits. Therefore, the composition of the querying elements will affect the querying time. Figure 6 shows the querying time comparison when the composition percentage of member elements varies. It can be concluded that the querying time of OHBF increases slower than SBF (or LHBF) as the percentage of member elements increases. Moreover, the longer the keys are or the larger the hash function number is, the less querying time OHBF spends.

VI. Case Study

In this section, we use a representative network application to test the actual performance of OHBF. In addition to reducing hash computation for Bloom filters, OHBF can achieve other benefits in practical implementations. We choose the Forward Information Base (FIB) lookup, which is a key performance bottleneck in backbone networks, as our case study. We will guide how to use and configure OHBF in this application.

Song *et al.* [4] propose using Bloom filters (SBFs) to accelerate FIB lookups. Essentially, FIB lookup is a Longest Prefix Matching (LPM) problem. Dharmapurikar *et al.* decompose the LPM to several Exact Matchings (EMs). While each EM can be easily accomplished by using hash table with just 1 (> 1 if hash collision happens) off-chip memory access per lookup, n EMs need n off-chip memory accesses. To avoid many high-latency off-chip memory accesses, they use low-latency on-chip SBFs to filter out unnecessary

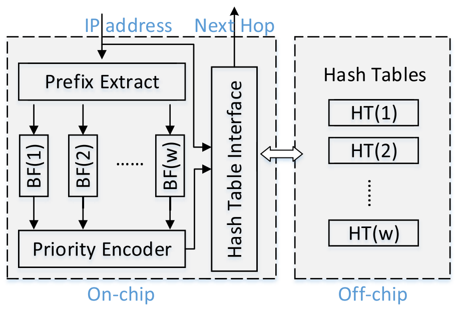


Fig. 7. FIB lookup with Bloom filters.

invalid off-chip hash probes. Although the SBFs are proposed to use hardware-based implementation, we show that a software-based implementation with our optimization is also viable to accelerate the FIB lookup. The on-chip SBFs with a large number of hash functions (*e.g.*, more than 100 hash functions in [3]) consume too much CPU cycles. If we replace SBF with OHBF, two times speedup can be achieved.

A. Bloom Filter-Based FIB Lookup

The proposed approach in [3] is shown in Figure 7. Before we describe the FIB lookup process, we need to introduce the construction process of the system. First, the prefixes (or <prefix, next-hop> pairs) of a FIB are grouped into W sets according to prefix length. Then, the system builds W Bloom Filters (BFs) and W Hash Tables (HTs). Each BF is associated with one set. All the prefixes in one set are inserted into an associate BF. Similarly, each HT is associated with one set. All the <prefix, next-hop> pairs in one set are inserted into an associate HT. The input of a BF is a prefix, and the output of a BF is 1-bit 0 (negative) or 1 (positive). The input of an HT is a prefix. If matched, the output is a next-hop; if not, the output is NULL. The W outputs of BFs form the inputs of Priority Encoder, which determines the search order of HTs. Note that BFs are stored in on-chip SRAM memory and HTs are stored in off-chip DRAM (or SRAM) memory. The placement policy implies that the BFs must be small enough to reside in on-die on-chip memory.

Now we describe the FIB lookup process. After an IP address arrives at the search engine, all the possible W prefixes are extracted from the destination IP address. Then these prefixes are tested in BFs. Note that each BF is used to test the membership of a fixed-length prefix. Due to the false positive effect of BFs, the positive output of a BF implies that the associate HT may contain the lookup prefix with high probability. As BFs do not have false negative, the negative output of a BF means that the associate HT does not contain the lookup prefix definitely. Therefore, positive output of a BF needs (at least) one off-chip hash probe to make sure whether the associate HT contains the lookup prefix or not. i positive outputs of BFs need (at most) i off-chip hash probes. Since the primary goal of the system is to minimize the number of high-latency off-chip hash probes, we need to probe HTs in an optimized order. The Priority Encoder tells the Hash Table Interface the probe order of off-chip HTs. Due to the LPM rule in FIB lookup, the probe order is from the longest prefix length to the lowest one. For example, if {BF(8), BF(16), BF(24)} show positive outputs, the probe order of HTs is <24, 16, 8>. Once an HT returns a valid next-hop, the probe process terminates. If all the possible HTs return NULL, a default next-hop may be used (the packet will be discarded if a default next-hop does not exist).

TABLE XI
COLLECTED TRACES FROM CAIDA

Name	Time	Util Rate
tr1	20140320.1300	14.5%
tr2	20140619.1300	15.0%
tr3	20140918.1300	39.3%
tr4	20141218.1300	27.2%
tr5	20150219.1300	19.7%
tr6	20150521.1300	18.9%
tr7	20150917.1300	16.2%
tr8	20151217.1300	22.4%

TABLE XII
COLLECTED FIBS FROM ROUTEVIEWS

Name	Location	# of prefixes
fib1	Ashburn, VA	617,554
fib2	Palo Alto, CA	640,484
fib3	London, GB	622,777
fib4	Portland, Oregon	617,421
fib5	Tokyo, Japan	604,371
fib6	Sydney, Australia	621,568
fib7	Eugene, Oregon	648,384
fib8	Atlanta, Georgia	613,243

B. Experimental Settings and Basic Configuration

We use a software method to simulate the Bloom filter-based FIB lookup approach. Note that, in the experiments, we emphasize the FIB lookup performance comparison with different Bloom filter implementations (SBF, LHBF and OHBF). The configurations of our experiments are as follows.

Platform: We implement the experiments on a commodity server with an Intel CPU Core i7-4790 (4 cores $\times$ 2 threads, 3.6 GHz) and 16GB DDR3 (800 MHz) memory. Each core of the CPU has independent an L1 D-Cache (32 KBytes), an L1 I-Cache (32 KBytes), and an L2 Cache (256 KBytes). The 4 cores share an L3 Cache (8 MBytes). In total, This server has more than 64 Mbits on-chip cache. The cache line size is 64-byte.

Traces: We use 8 real-world Internet traffic traces, shown in Table XI. These traces were collected by CAIDA [40] on backbone 10 Gbps links, with monitors located in Chicago. Each trace lasts 60 minutes.

FIBs: We collect 8 BGP FIBs from *routeviews.org* [41], as shown in Table XII. The collection time is 2016/08/01.00:00 UTC. The FIB size ranges from 600K to 650K.

System Settings: The false positive probability of Bloom filters is minimized to $f = \left(\frac{1}{2}\right)^k$, when the number of hash functions k satisfies $k = \frac{m}{n} ln2$. That is to say, if k is not the optimized value, we need to sacrifice more memory to lower the false positive probability. Due to limited on-chip memory, we always use the optimized number of hash functions to reduce the on-chip memory requirement. In implementation, all BFs and all HTs are implemented on one Bloom filter

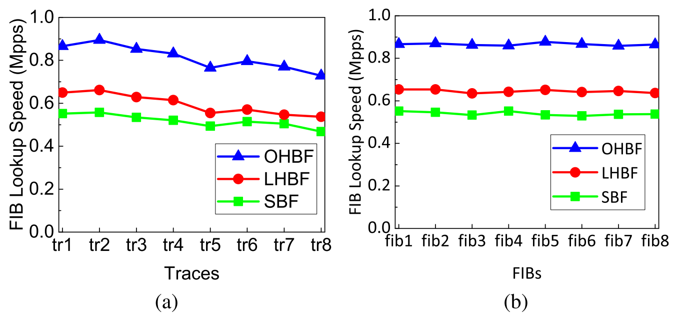


Fig. 8. Lookup speed with basic configuration. (a) Different Traces with fib1. (b) Different FIBs with tr1.

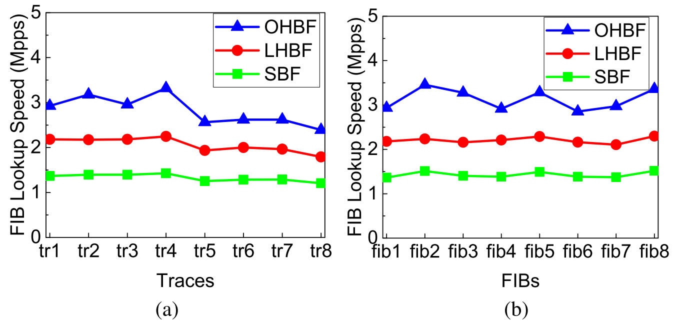


Fig. 9. Lookup speed with optimized configuration. (a) Different Traces with fib1. (b) Different FIBs with tr1.

and one hash table respectively. We extend all prefixes to 32-bit keys by filling up zero in high bits. Thus all keys have the same length. A prefix's length is recorded in the corresponding hash table entry. This simplifies the system parameter settings. n is the total prefix number, m is the total Bloom filter bit vector size.

According to [3], the expected number of hash probes per FIB lookup is:

$$E_{exp} = Wf + 1 = W\left(\frac{1}{2}\right)^{k} + 1 \tag{17}$$

In our BGP IPv4 FIBs, $W = 25$, as the minimum length of prefixes in actual IPv4 FIBs is 8. We set $k = 10$ in our system. Then we can get $E_{exp} = 1.02$, which is small enough for expected off-chip hash probes.

As the previous settings, for a 650K-prefix FIB ($n = 650$K), we need $m = 9.4$ Mbits on-chip memory (cache). The memory cost for partition sizes C is 200 bits, which is negligible for the overall Bloom filter's space cost. Because the size of all the FIBs in Table XII is smaller than 650K, our platform satisfies the on-chip memory requirement (the platform has more than 64 Mbits cache).

C. Evaluations

1) Basic Configuration: Section VI-B describes the basic configuration of our system. Because the commodity server does not provide the interface to operate the on-chip memory (cache), we need a warm-up stage for the Bloom filters. The warm-up stage reads the Bloom filters 100 times to ensure them stored in cache as much as possible, before we start to measure the FIB lookup performance.

As shown in Figure 8, in the basic configuration, the OHBF-based system shows consistently better performance than the LHBF and SBF-based systems, either using different traces to query one FIB (Figure 8(a)) or using one trace to query different FIBs (Figure 8(b)). In this figure, we find that the OHBF-based system does not improve the lookup speed manyfold, although OHBF reduces the hash computational cost to nearly $1/k$ of SBF. The reason is twofold. First, it is the off-chip hash table probes, rather than the on-chip BF tests, that dominate the system lookup speed. Second, not all BFs in the system are tested. The lookup process terminates when a BF returns a positive output and the associate HT returns a valid next-hop, which results in a smaller fraction of hash computation in Bloom filters for most lookups.

2) Optimized Configuration: In the worst case, we need to check on-chip BFs and probe off-chip HTs 25 times with the basic configuration. In this section, we use a technique, called leaf-pushing [42], to optimize the worst case. In the new scheme, only prefixes with length {8, 16, 24, 32} are reserved. Other prefixes are pushed to the four levels according to algorithms introduced in [42]. After leaf-pushing, we only need 4 on-chip BFs, thus reducing the on-chip checking overhead. Another optimization is that we can use IP address as the hash value in OHBF. Note that IP address has 32-bit and our OHBF only needs one hash value. Then the hash computation in OHBF only needs several CPU *div* instructions to perform. This advantage comes from the reduced hash function requirement of OHBF.

As shown in Figure 9, in the optimized configuration, OHBF-based system also shows consistently better performance than LHBF and SBF-based system, either using different traces to query one FIB (Figure 9(a)) or using one trace to query different FIBs (Figure 9(b)). Compared to the basic configuration (Figure 8), the optimized configuration with three different Bloom filter implementations all have more than two times lookup speed improvement. The performance improvement comes from a decreased number of BF tests. We can also find that, in the optimized configuration, OHBF-based system has two times faster lookup speed than SBF-based system. However, in the basic configuration, it is only (approximate) 1.6 times faster. The reason is twofold. First, it has less BF tests on average in the optimized configuration, as the total BF number decreases significantly. Second, OHBF uses IP address as the only hash value, which reduces hash computational overhead.

D. Discussion

In this section, we realize a Bloom filter-accelerated FIB lookup system. The system employs three different Bloom filter implementations for performance comparison. Apparently, the OHBF-based system has better lookup performance than the LHBF and SBF-based systems. This is because the hash computation cost is high in such systems and the OHBF reduces it to a large extent. Note that the system needs tens of Bloom filters, and each Bloom filter needs multiple hash functions. In the basic configuration, 250 hash functions are needed for Bloom filters in total. Hash computation in Bloom filters becomes a system performance bottleneck as shown in our case study. The reduced hash computational cost in OHBF results in considerable system performance improvement.

VII. Conclusions

OHBF requires only one base hash function and a set of k consecutive prime numbers as modulo operands. The composite hash functions have the strength of the desired property of k strong and independent hash functions, yet the overall computational complexity is limited to the base hash function. The Bloom filter vector is conditioned to the selected prime numbers and each hash value addresses one of the partitions respectively. With proved false positive performance, OHBF is ideal for applications which need both low latency and high throughput. Both performance evaluations and the case study show that OHBF outperforms standard Bloom filters and other Bloom filter variants with faster implementing speed and practical lower false positive ratio. The proposed OHBF is a fundamental optimization for Bloom filters and retains its generality. Therefore, we can easily extend the OHBF technique to other Bloom filter variants, such as Counting Bloom Filter [12], One Memory Access Bloom Filter [26], and Space-Code Bloom Filter [19].

References

[1] M. Chiosi, "Network functions virtualisation—Introductory white paper," in *Proc. SDN OpenFlow World Congr.*, 2012, pp. 1–16.

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

[3] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," in *Proc. ACM SIGCOMM*, 2003, pp. 201–212.

[4] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended Bloom filter: An aid to network processing," in *Proc. ACM SIGCOMM*, 2005, pp. 181–192.

[5] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter forwarding architecture for large organizations," in *Proc. ACM CoNEXT*, 2009, pp. 313–324.

[6] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel Bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, Jan. 2004.

[7] F. Hao, M. Kodialam, and T. Lakshman, "Building high accuracy Bloom filters using partitioned hashing," in *Proc. ACM SIGMETRICS*, 2007, pp. 277–288.

[8] A. A. Iqbal, M. Ott, and A. Seneviratne, "Simplistic hashing for building a better Bloom filter on randomized data," in *Proc. 13th Int. Conf. Netw.-Based Inf. Syst. (NBiS)*, Sep. 2010, pp. 325–331.

[9] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 1, pp. 131–155, 1st Quart., 2012.

[10] *Crypto++ 5.6.0 Benchmarks*. Accessed: Apr. 24, 2015. [Online]. Available: http://www.cryptopp.com/benchmarks.html

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

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

[13] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable Bloom filters," in *Proc. ACM SIGMOD*, 2006, pp. 25–36.

[14] H. Shen and Y. Zhang, "Improved approximate detection of duplicates for data streams over sliding windows," *J. Comput. Sci. Technol.*, vol. 23, no. 6, pp. 973–987, 2008.

[15] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, and M. F. Magalhaes, "The deletable Bloom filter: A new member of the Bloom family," *IEEE Commun. Lett.*, vol. 14, no. 6, pp. 557–559, Jun. 2010.

[16] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li, "Noisy Bloom filters for multi-set membership testing," in *Proc. ACM SIGMETRICS*, 2016, pp. 139–151.

[17] H. Dai, L. Meng, and A. X. Liu, "Finding persistent items in distributed datasets," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 1–9.

[18] S. Cohen and Y. Matias, "Spectral Bloom filters," in *Proc. ACM SIGMOD*, 2003, pp. 241–252.

[19] K. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-code Bloom filter for efficient per-flow traffic measurement," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 1762–1773.

[20] Y. Matsumoto, H. Hazeyama, and Y. Kadobayashi, "Adaptive Bloom filter: A space-efficient counting algorithm for unpredictable network traffic," *IEICE Trans. Inf. Syst.*, vol. 91, no. 5, pp. 1292–1299, 2008.

[21] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable Bloom filters," *Inf. Process. Lett.*, vol. 101, no. 6, pp. 255–261, 2007.

[22] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic Bloom filters," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 120–133, Jan. 2010.

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

[24] M. K. Yoon, J. Son, and S.-H. Shin, "Bloom tree: A search tree based on Bloom filters for multiple-set membership testing," in *Proc. IEEE INFOCOM*, Apr. 2014, pp. 1429–1437.

[25] F. Putze, P. Sanders, and J. Singler, "Cache-, hash- and space-efficient Bloom filters," in *Proc. Int. Workshop Exp. Efficient Algorithms*, 2007, pp. 108–121.

[26] Y. Qiao, T. Li, and S. Chen, "One memory access Bloom filters and their generalization," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1745–1753.

[27] B. Donnet, B. Baynat, and T. Friedman, "Retouched Bloom filters: Allowing networked applications to trade off selected false positives against false negatives," in *Proc. ACM CoNEXT*, 2006, p. 13.

[28] R. P. Laufer *et al.*, "Towards stateless single-packet IP traceback," in *Proc. IEEE LCN*, Oct. 2007, pp. 548–555.

[29] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: An efficient data structure for static support lookup tables," in *Proc. ACM SODA*, 2004, pp. 30–39.

[30] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom filters: From approximate membership checks to approximate state machines," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 315–326, 2006.

[31] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *Proc. ACM CoNEXT*, 2014, pp. 75–88.

[32] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better Bloom filter," *Random Struct. Algorithms*, vol. 33, no. 2, pp. 187–218, Sep. 2008.

[33] H. Song, F. Hao, M. Kodialam, and T. Lakshman, "IPv6 lookups using distributed and load balanced Bloom filters for 100 Gbps core router line cards," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 2518–2526.

[34] M. Skjegstad and T. Maseng, "Low complexity set reconciliation using Bloom filters," in *Proc. ACM FOMC*, 2011, pp. 33–41.

[35] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 3, pp. 52–66, Jul. 2015.

[36] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *J. Comput. Syst. Sci.*, vol. 18, no. 2, pp. 143–154, Apr. 1979.

[37] M. Ramakrishna, "Practical performance of Bloom filters and parallel free-text searching," *Commun. ACM*, vol. 32, no. 10, pp. 1237–1239, 1989.

[38] J. L. Devore, *Probability and Statistics for Engineering and the Sciences*. North Scituate, MA, USA: Duxbury Press, 2012.

[39] C. Henke, C. Schmoll, and T. Zseby, "Empirical evaluation of hash functions for multipoint measurements," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 39–50, 2008.

[40] C. Walsworth, E. Aben, K. Claffy, and D. Andersen. (2016). *The CAIDA Anonymized Internet Traces*. [Online]. Available: http://www.caida.org/data

[41] University of Oregon. (2016). *Route Views Project*. [Online]. Available: http://www.routeviews.org/

[42] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, 1999.

Jianyuan Lu received the B.S. degree in information and computing science from the Beijing University of Posts and Telecommunications, Beijing, China, in 2011, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, in 2017.

He is currently a joint Postdoctoral Research Fellow with Tsinghua University and Alibaba Cloud. His research interests include cloud computing, software-defined networking, network measurements, high-performance network algorithm, and power-proportional network design.

Tong Yang received the Ph.D. degree in computer science from Tsinghua University in 2013.

He visited the Institute of Computing Technology, Chinese Academy of Sciences, China, from 2013 to 2014. He is currently a Research Assistant Professor with the Computer Science Department, Peking University. He published papers in SIGCOMM, SIGKDD, SIGMOD, SIGCOMM CCR, VLDB, ATC, ToN, ICDE, and INFOCOM. His research interests include network measurements, sketches, IP lookups, Bloom filters, sketches, and KV stores.

Yi Wang received the Ph.D. degree in computer science and technology from Tsinghua University in 2013.

He is currently a Research Associate Professor with the SUSTech Institute of Future Networks, Southern University of Science and Technology, Shenzhen, China. His research interests include router architecture design and implementation, software-defined networks, greening the Internet, fast packet forwarding, information-centric networking, and time-sensitive networks.

Huichen Dai received the B.S. degree from the Xian University of Electronic Science and Technology, Xi'an, China, in 2010, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, in 2016.

He was a Post-Doctoral Research Fellow with the Department of Computer Science and Technology, Tsinghua University. He is currently a Senior Engineer at Huawei Technologies Co., Ltd, where he is currently working on congestion control algorithms for RDMA. He used to be interested in research topics in computer networks, including router architecture, fast packet processing, and future Internet architecture, such as named-data networking and software-defined networking.

Xi Chen received the B.S. degree from the School of Information and Communication Engineering, Beijing University of Posts and Telecommunications, Beijing, China, in 2016.

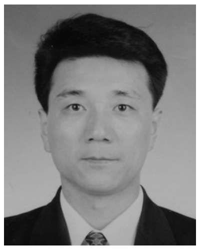
His research interests include routing, network security, and software-defined networking.

Linxiao Jin received the B.S. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2014.

He is currently a Software Engineer at Oracle Public Cloud. His research interests include routing, network security, and software-defined networking.

Haoyu Song received the B.E. degree in electronics engineering from Tsinghua University, Beijing, China, in 1997, and the M.S. and D.Sc. degrees in computer engineering from Washington University in St. Louis, St. Louis, MO, USA, in 2003 and 2006, respectively.

He was an MTS Researcher with Bell Labs, Alcatel-Lucent, Holmdel, NJ, USA, and a Research Assistant with the Applied Research Laboratory, Washington University in St. Louis. He is currently a Senior Principal Network Architect with Huawei Technologies, Santa Clara, CA, USA. He has published over 30 peer-reviewed papers and has filed over 20 patents for his work on network algorithm and architecture. His research interests include network virtualization and cloud computing, high-performance networks, algorithms for network packet processing and security, network chip architecture, and ASIC/FPGA design and verification.



Bin Liu received the M.S. and Ph.D. degrees in computer science and engineering from Northwestern Polytechnical University, Xi'an, China, in 1988 and 1993, respectively.

He is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His current research areas include high-performance switches/routers, network processors, high-speed network security, and greening the Internet. He has received numerous awards from China, including the Distinguished Young Scholar of China and the inaugural Applied Network Research Prize sponsored by ISOC and IRTF in 2011.