An Enhanced Bloom Filter for Longest Prefix Matching

Gahyun Park  
SUNY-Geneseo  
Email: park@geneseo.edu

Minseok Kwon  
Rochester Institute of Technology  
Email: jmk@cs.rit.edu

Abstract—A Bloom filter is a succinct data structure for membership queries. While the filter enables a compact storage, it allows false positives when queried and exhibits an inherent tradeoff between the false positive rate and space complexity. Among many applications, IP address lookup shows promise for improvement using on-chip Bloom filters; however, high false positive rates may cause more off-chip memory access and degrade the performance significantly. We introduce a new Bloom filter called the length-aware Bloom filter (LABF) for multiple pattern matching problems. The primary idea is to explore the discrepancy in length distribution between the set of patterns and the set of prefixes of input text that are examined against the patterns. While maintaining the simplicity, LABFs outperform the standard Bloom filter when the pattern lengths are non-uniformly distributed in a wide range. Such pattern length distributions frequently occur in multiple pattern matching, e.g., longest prefix matching in IP address lookup. We derive a simple formula to configure parameters for the construction of an LABF and provide a provable guarantee for the average number of false positives. Our experimental results show that LABFs reduce the average false positive rates by a factor of 4 and 16, for IPv4 and IPv6, respectively.

Keywords—Bloom filters, probabilistic data structure, multiple pattern matching, longest prefix matching, IP lookup

I. INTRODUCTION

A Bloom filter (BF) is a randomized data structure for membership queries [1]. Due to their simplicity and memory efficiency, Bloom filters have found a myriad of applications including IP lookup and packet inspection [2], [10], [5], [4]. While numerous variants of Bloom filters have been proposed for different applications, not much work has been done to improve the essential performance parameter of Bloom filters, namely the false positive probability with a given memory constraint. It is indeed well-recognized that a Bloom filter is a highly optimized data structure and there is almost no room to improve it. We often call the original Bloom filter standard BF in this paper to distinguish it from other Bloom filter variants.

We present a new data structure called a length-aware Bloom filter (LABF) for multiple pattern matching that can optimize a standard BF further. LABFs exploit the discrepancy in length distributions between the set of patterns and the set of prefixes of a queried input text. We observe that prefixes of an input text \( T \) have a uniform length distribution with exactly one prefix for each length \( 1 \leq i \leq |T| \). In contrast, many well-known sets of patterns such as IPv4 and IPv6 address prefixes and the snort database in networking intrusion and detection system are characterized by highly non-uniform lengths distributed over a wide range. While a standard BF uses the same number of hash functions for all patterns, an LABF uses different numbers of hash functions depending on pattern length leading to significant reduction in false positive rates without incurring extra memory space. The idea of using different numbers of hashes for each data to minimize the false positive rate is not new [3], [15]. Their schemes, however, require the popularity distribution of data rather than length distributions.

It is important to note that the length distribution was also considered in prior work [5], [11], [6] for different reasons. In those schemes, the skewed distribution was not a desirable characteristic in constructing Bloom filters; however, LABFs take advantage of the property to improve the performance of the filter. We provide a provable guarantee on the expected number of false positives. Our experimental results show that LABFs reduce the average false positive rates by a factor of 4 and 16, for IPv4 and IPv6, respectively.

The rest of the paper is organized as follows. We discuss Bloom filters and multiple pattern matching in Section II and discuss LABFs in Section III. Section IV evaluates the performance of LABFs. We summarize related work in Section V and conclude in Section VI.

II. BACKGROUND

A. Bloom Filters

A Bloom filter that represents a set \( S = \{x_1, \ldots, x_n\} \) of \( n \) elements is an array of \( m \) bits, initially all set to 0. The elements of the set are hashed to the filter using \( k \) hash functions, \( h_1, \ldots, h_k \), with a range of \( \{1, \ldots, m\} \). We make the natural assumption that these hash functions map each item in the universe to a random number uniformly distributed over the range \( \{1, \ldots, m\} \). As an element \( x \in S \) is inserted, all the bits of \( h_i(x) \) are set to 1 for \( 1 \leq i \leq k \). Note that a bit in the filter may be overwritten to 1 more than once as the hash values of subsequent insertions are mapped to the same bit.

A membership query to test whether item \( y \) is in \( S \) is performed as follows. If all of \( h_i(y) \) are set to 1, \( y \) is a member of \( S \) with some probability; otherwise, \( y \) is clearly not a member of \( S \). That is, a Bloom filter may yield a false positive where the filter indicates that an element \( x \) is in \( S \) although it is not.

Since the mathematics behind a Bloom filter has been well studied [2], we briefly summarize the main results. The essential performance parameter of Bloom filters is the
accuracy measured by a false positive probability. The false positive rate, denoted by \( p \), can be computed in terms of \( m \), \( k \), and \( n \) as follows.

\[
p = \left(1 - e^{-\frac{kn}{m}}\right)^k.
\]  

(1)

Given \( n \) elements in \( S \) and memory amount \( m \), the false positive probability is minimized by selecting the value of \( k \) as \( \frac{m}{n \log_2 e} \). Then about a half of the bits \((m/2)\) in the filter are set to 1 and the other half are 0’s, and the false positive probability becomes the minimum, \( \frac{1}{2} \). In this paper, we only consider the optimally configured Bloom filters which use the optimal number of hash functions and hence the optimal performance of the standard Bloom filter is to be compared to that of our length-aware Bloom filter.

**B. Multiple Pattern Matching**

Multiple pattern matching is one of the well-known classical problems in computer science. Given a set of patterns/strings/elements, one wants to find all occurrences of those patterns in an input text. Let \( S = \{x_1, \ldots, x_n\} \) be a set of patterns. By \( w \) and \( w' \), we denote the maximum and minimum lengths of patterns in \( S \). That is, \( w = \max_{1 \leq i \leq n} \{|x_i|\} \) and \( w' = \min_{1 \leq i \leq n} \{|x_i|\} \), where \(|x_i|\) is the length of \( x_i \). Without loss of generality, we assume that \( w' = 1 \). Let \( T \) be an input text that is to be examined against \( S \). In general, the length of input text \( T \) is longer than \( w \), i.e., \(|T| \geq w\).

While many well-known methods exist for (exact) multiple pattern matching such as the Aho-Corasick and Commentz-Walter algorithms, we consider the most basic algorithm using the brute-force approach that scans \( T \) from left to right. The process consists of \((|T| - w + 1)\) rounds, and a substring \( y \) of \( T \) in a \( w \)-bit sliding window is considered in each round. Let \( y = z_1 \cdots z_w \), where each \( z_i \) is the \( i \)-th bit, and let \( y_i = z_1 \cdots z_i \) be a prefix of \( y \) with length \( i \). Each \( y_i \) is tested whether or not it is in \( S \). After examining all the prefixes of \( y \), the sliding window moves one bit to right to begin the next round.

The standard Bloom filters have been used for multiple pattern matching problems [6, 12, 4]. Once all of the patterns in \( S \) are inserted, each prefix \( y_i \) is queried to the filter for its membership: \( y_i \in S \)? All prefixes \( y_i \) of \( y \) are probed (possibly in parallel) in the same round. The performance of a standard Bloom filter used in the multiple pattern matching context can be measured in terms of the number of false positives. Let \( m \) be the filter size in number of bits. Recall that the optimal number of hash functions is \( k = \frac{m}{n \log_2 e} \) and the false positive rate is \( (\frac{1}{2})^k \). Each string \( x_i \) is hashed using \( k \) independent hash functions \( h_1, \ldots, h_k \). In each round, all \( w \) prefixes are probed and the false positive rate for each prefix is \( (\frac{1}{2})^k \).

Using the mean of binomial random variables, the expected number of false positives in one round is

\[
w \frac{1}{2}^k.
\]  

(2)

And hence the expected number of false positives for an input text \( T \) is \((|T| - w + 1)w \frac{1}{2}^k\).

We observe that the false positive probability itself may be very small; for example, using \( k = 10 \) hash functions, the false positive rate is only \((\frac{1}{2})^{10} \approx 0.001\). However, the number of false positives in an instance of multiple pattern matching increases in proportion to \(|w|T|\), and Bloom filters are tested against a large number of input strings in practice. Thus, reducing the false positive probability (for a fixed memory) is one of the most important goals in any Bloom filter applications.

**III. LENGTH-AWARE BLOOM FILTERS**

We first discuss the main idea for of length-aware Bloom filters, and then describe our algorithm with its properties more formally.

**A. Bloom Filters with Non-Uniform Distributions**

A standard Bloom filter does not always yield the minimum number of false positives in multiple pattern matching. Assume a specific pattern distribution in which the range of pattern lengths in \( S \) is wide, i.e., \( w \) is large, and a half of those patterns are of the same length \( l \), for some \( 1 \leq l \leq w \). Let \( S_1 \subseteq S \) be the set of the patterns with length \( l \). Unlike the standard Bloom filter that uses the same number of hash functions for all patterns, we use less hash functions for the patterns in \( S_1 \). The patterns in \( S \setminus S_1 \) then use more hash functions while maintaining the invariance that about a half of the Bloom filter is filled, which guarantees the minimum number of false positives. Suppose that one uses \( k - 1 \) hash functions \( (h_1, \ldots, h_{k-1}) \) for \( x \in S_1 \) and \( k + 1 \) hash functions \( (h_1, \ldots, h_{k+1}) \) for \( x \in S \setminus S_1 \). Since the total number of hash functions used to fill the filter is still \( kn \), only a half of the filter is filled with 1. In this case, the false positive probability for some \( x \notin S \) for the former case becomes \((\frac{1}{2})^{k-1} \), which is higher than that of the standard Bloom filter; the false positive probability for the latter becomes \((\frac{1}{2})^{k+1} \), which is lower than that of the standard Bloom filter. In a \( w \)-bit wide sliding window, however, there is only one prefix with length \( l \) which has a higher false positive rate and the rest \( w - 1 \) prefixes have lower false positive rates than \((\frac{1}{2})^k \). Thus the expected number of false positives becomes

\[
(\frac{1}{2})^{k-1} + (w - 1)(\frac{1}{2})^{k+1} = \frac{w + 3}{2} \left(\frac{1}{2}\right)^k
\]  

(3)

In comparison to the standard Bloom filter (refer to Equation \((2)\)), the number of false positives is reduced nearly in half.

The number of false positives given in Equation \((3)\) can be further reduced by choosing a different number of hash functions. Let \( c = \frac{\log w}{2} \) and assume that \( c \) is an integer. Now a filter is constructed using \( k - c \) hash functions for the elements in \( S_1 \) and \( k + c \) hash functions for the elements in \( S \setminus S_1 \). The expected number of false positives is then reduced by a factor...
of $\sqrt{m/2}$ as follows:

$$
(\frac{1}{2})^{k-c} + (w-1)(\frac{1}{2})^{k+c} = \left(\frac{2^c + \frac{w-1}{2}}{2^c}\right) \left(\frac{1}{2}\right)^k
$$

$$
= \left(\frac{\sqrt{w} + \frac{w-1}{\sqrt{w}}}{\sqrt{w}}\right) \left(\frac{1}{2}\right)^k
$$

$$
\leq \left(\frac{w}{2}\right)^{\frac{1}{2}} w^\left(\frac{1}{2}\right)^k
$$

This approach leads to a lower expected number of false positives due to the following two reasons: 1) Most of the prefixes (except of length $l$) produce low false positive rates due to using more hash functions $(k+c)$, nevertheless, 2) the fill-up factor, i.e., the fraction of enabled bits in the filter, does not increase because the other half of the patterns with length $l$ lowers the fill-up factor using only $k-c$ hash functions. Note that we aim to reduce the expected number of false positives, from $w^{(\frac{1}{2})k}$ to $w^{(\frac{1}{2})k}$, rather than the false positive rate of each string. With the standard Bloom filter, each string has the same false positive rate, and hence reducing the number of false positives is equivalent to reducing each string’s false positive rate.

### B. Length-Aware Bloom Filters

We now describe the length-aware Bloom filter (LABF) formally. The basic idea is to partition $S$ into subsets depending on the lengths of elements and use different numbers of hash functions for each subset while maintaining the invariance that a half of the cells in the filter are set to 1. Unlike standard Bloom filters in which the optimal number of hash functions is uniquely determined, an LABF may have more than one option to select the number of hash functions for each subset. While we partition $S$ into two subsets for simplicity, one may use more than two subsets in practice to optimize even further. Here we introduce two different constructions, namely symmetric and asymmetric LABFs. All of the parameters used in our LABF description are summarized in Table I.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>Set of patterns represented by a Bloom filter</td>
</tr>
<tr>
<td>$m$</td>
<td>Bloom filter size (standard or length-aware)</td>
</tr>
<tr>
<td>$k$</td>
<td>Optimal number of hash functions for a standard Bloom filter</td>
</tr>
<tr>
<td>$w$</td>
<td>Number of different pattern lengths in $S$</td>
</tr>
</tbody>
</table>

#### 1) Symmetric LABFs: Let $L = \{1, \ldots, w\}$ be the set of distinct lengths for the elements in $S$. For any $l \in L$, let $S(l) = \{x \in S \mid |x| = l\}$ be a subset of $S$ whose elements are of length $l$. Let $\{l_1, \ldots, l_w\}$ be a permutation of $\{1, \ldots, w\}$ such that $|S(l_1)| \geq |S(l_2)| \geq \ldots \geq |S(l_w)|$. That is, $l_1$ is the most popular length, $l_2$ is the second most popular, and so on. Let $i^*$ be the smallest $i$ such that

$$
\sum_{j=1}^{i} |S(l_j)| \geq \frac{1}{2} n.
$$

Next, we partition $L$ into two subsets $L_1 = \{l_1, \ldots, l_c\}$ and $L_2 = \{l_{c+1}, \ldots, l_w\}$. Also, partition $S$ into two subsets $S_1$ and $S_2$ where $S_1 = S(l_1) \cup \ldots \cup S(l_c)$ and $S_2 = S(l_{c+1}) \cup \ldots \cup S(l_w)$. Let $\epsilon = \frac{1}{w}$, and assume that $\epsilon \leq \frac{1}{2}$, which reflects the condition that pattern lengths are non-uniformly distributed over a wide range. Finally, choose $c = \lfloor \log_2(2\epsilon)^{-1} \rfloor$. An LABF is constructed such that if $x \in S_1$, hash $x$ with $h_1, \ldots, h_{k-c}$, and if $x \in S_2$, hash $x$ with $h_1, \ldots, h_{k+c}$.

When an input string $x$ is queried, one needs to probe the locations at $h_1(x), \ldots, h_{k-c}(x)$ in the filter if $|x| \in L_1$, and probe the locations at $h_1(x), \ldots, h_{k+c}(x)$ otherwise. The false positive rate (assuming $x \not\in S$) is either $(\frac{1}{2})^{k-c}$ if $|x| \in L_1$ or $(\frac{1}{2})^{k+c}$ if $|x| \in L_2$. The expected number of false positives in a $w$-bit sliding window is then

$$
e w(\frac{1}{2})^{k-c} + (1-\epsilon)w(\frac{1}{2})^{k+c} = \alpha_c w(\frac{1}{2})^k
$$

where $\alpha_c = 2\epsilon + 1 - \frac{1}{2}$.

#### 2) Asymmetric LABFs: We introduce an alternative construction of LABFs which can be more effective than the symmetric counterpart where the length distribution is highly non-uniform over a relatively small range. With a fixed constant $c \geq 1$, let $i^* = i^*(c)$ be the smallest $i$ such that

$$
\sum_{j=1}^{i} |S(l_j)| \geq \frac{c}{c+1} n.
$$

As in the symmetric LABFs, let $S_1 = S(l_1) \cup \ldots \cup S(l_c)$ and $S_2 = S(l_{c+1}) \cup \ldots \cup S(l_w)$. To insert $x$ into the filter, use $h_1, \ldots, h_{k-1}$ if $x \in S_1$, use $h_1, \ldots, h_{k+c}$ otherwise. The expected number of false positives is

$$
e w(\frac{1}{2})^{k-1} + (1-\epsilon)w(\frac{1}{2})^{k+c} = \alpha_c w(\frac{1}{2})^k
$$

where $\alpha_c = 2\epsilon + 1 - \frac{1}{2}$.

If $\alpha_c < 1$, then the expected number of false positives is less than that of the standard Bloom filter. Note that $\alpha_c$ is

---

1In fact, we can relax the assumption as $\epsilon \leq \frac{1}{2}$. If $\frac{1}{2} \leq \epsilon \leq \frac{1}{4}$, then the gain of LABF is small. Thus we do not consider the $\epsilon$ values in this range.
Algorithm 2 Asymmetric LABF construction

1: Partition $S = S(l_1) \cup \ldots \cup S(l_w) \\
2: \text{Fix } c \geq 1 \\
3: i^* \leftarrow \min\{i \mid \sum_{j=1}^{i} |S(l_j)| \geq \frac{c}{c+1} n\} \\
4: \epsilon \leftarrow \frac{c}{i^*} \\
5: \alpha_c \leftarrow 2\epsilon + \frac{1-\epsilon}{2} \\
6: \text{if } \alpha_c < 1 \text{ then} \\
7: \text{use } h_1, \ldots, h_{k-1} \text{ for } x \in S_1 \\
8: \text{use } h_1, \ldots, h_{k+c} \text{ for } x \in S_2 \\
9: \text{else} \\
10: \text{use } h_1, \ldots, h_k \text{ for all } x \in S \\
11: \text{end if}

IV. LONGEST PREFIX MATCHING USING LABFs

A. LABFs for Longest Prefix Matching

Longest Prefix Matching (LPM) can be viewed as a multiple pattern matching problem to which Algorithms 1 and 2 are directly applicable. In applications like IP lookup or deep packet inspection, the prefixes that result in (both true and false) positive matches are checked against the raw data starting from longest until a true match is found. Thus, false positives shorter than the longest matched prefix are disregarded. Formally, let $T$ be an incoming string (e.g., IP address) and the true longest match of $T$ is of length $w_T \leq w$. The average number of false positives is then

$$w - w_T \left(\frac{1}{2}\right)^k$$

for a standard Bloom filter. Unlike (2), the number of false positives in LPM depends on $w_T$ which is not known a priori. Since only those prefixes with length $> w_T$ contribute to the number of false positives in (7), we recommend to use more hash functions for long prefixes and less hash functions for short prefixes.

B. IP Address Lookup

IPv4 addresses are known to have a skewed prefix length distribution as presented in [5]. This skewed distribution makes IP address lookup a relevant candidate for length-aware Bloom filters. Their analysis of IPv4 BGP tables shows that more than 55% of the prefixes are 24 bit long, most others are between 15 to 23 bits, and very few prefixes are less than 8 bits. Our own analysis confirms this result as 55% of the prefixes are 24 bits long, and the other 45% are spread over different prefix lengths. In their paper, initially on-chip memory is divided into 32 filters equally in support of parallel access for 32 prefix lengths. Unfortunately, the unbalanced length distribution leads to higher false positive rates since the filter of some prefix lengths (e.g., 24) results in having much more 1’s than 0’s. To remedy this problem, asymmetric Bloom filters are proposed allocating filter memory in proportion to the fair share of each prefix in the distribution yielding the same false positive rates as one standard Bloom filter.

LABFs are not directly comparable to asymmetric Bloom filters since the former is designed for general sequential access (like Bloom filters) while the latter is specifically designed for parallel access. It is thus sufficient to compare LABFs against standard Bloom filters in terms of false positives, and we can infer the performance of LABFs in comparison to asymmetric Bloom filters because asymmetric Bloom filters are optimized to give the same false positive rates as standard Bloom filters—a half of the bits in the filter are set to 1.

LABFs are 1) simple and 2) flexible to dynamic changes of the prefix distribution as LABFs use a single filter and allocate a prefix’s fair share by using different numbers of hash functions instead of manipulating memory allocation. In addition, LABFs are not deterministic with respect to the prefix length distribution and the number of hash functions. As network conditions change dynamically, a router needs to update its own routing table causing updated prefix length distributions. To reflect this change, LABFs need to recompute the numbers of hash functions periodically. Considering each router issues route updates not so frequently (e.g., 30 seconds in RIPv), this overhead is not unbearable in practice.

We run IP address lookup experiments based on our analysis of prefix length distribution. We generate prefixes randomly in proportion to the distribution, and insert them to Bloom filters. We then generate IP addresses again randomly and search the filters for the prefix of a target IP address. The experiments are conducted for both the BF and LABF with two different numbers of input addresses (100,000 and 1,000,000) as the number of hash functions ranges from 7 to 14. For the LABF, we use $k+2$ hash functions when the prefix length is less than 24, $k-2$ hash functions when the length is 24, and $k+10$ hash functions otherwise. Note that an IP address may be matched to more than one prefix, and we simply selected the longest prefix in such a case.

In Figure 1 we compare the false positive rates of these filters as the number of hash functions changes. We test two different cases–Figure 1(a) shows the results when the number of hash functions is in the range of (7, 10), $m = 1.44kn$ (filter size), $n = 1,000$ (the number of prefixes) for 100,000 IP addresses, and $n = 10,000$ for 1,000,000 addresses; Figure 1(b) shows the results with the same parameters except the number of hash function that changes from 11 to 14. The filter size is set to be in proportion to $n$. The results show that the false positive rates of LABFs are lower than those of BF in all of the cases by a factor of 2 to 4. For example, the false positive rates of the LABF and BF are 0.000319 and 0.001153, respectively, when the number of trials is 1,000,000 and $k = 10$. 
We run similar experiments for IPv6 as well and analyze their false positive rates for different filter sizes. The IPv6 addresses are 128 bits long and their prefix lengths vary from 16 to 128 bits [9]. We use the latest snapshot of the IPv6 BGP table from Telecom Italia Lab (TILAB) measured in January, 2012 [14]. We analyze the distribution of prefix lengths for IPv6 addresses that contains 36 unique prefix lengths from 16 to 128 bits out of 7,520 prefixes. The most popular prefix length is 32 accounting for nearly 50.4% (3,787/7,520), and the second most popular one is 48, approximately 33.5% (2,516/7,520) of all the prefix lengths. This means that around 84% of the prefixes belong to one of these two prefix lengths. As discussed earlier, this lopsided distribution strongly indicates that IPv6 also has potential for improved false positive rates if a length-aware Bloom filter is used.

We present the false positive rates computed from both the standard and our length-aware Bloom filters as the number of hash functions used changes from 7 to 14 in Figure 2. We use the same parameter values as the IPv4 experiments for the number of IP addresses, the filter size, and the number of prefixes. In the LABF, we use $k - 2$ and $k - 1$ hash functions for the prefix lengths of 32 and 48, respectively, while we use $k + 9$ hash functions for all the other prefix lengths. Of course, the BF uses $k$ hash functions for all prefix lengths. The results again confirm that the false positive rates of the LABFs are reduced by a factor of approximately 15-16 compared to those of the BFs. For example, the false positive rates of the BF and LABF are 0.000119 and 0.000007, respectively, when the number of trials is 100,000 and $k = 13$.

V. RELATED WORK

Bloom filters and their variations are used extensively for networking and other applications [2], [10], [5], [4], [7]. While the performance of a standard Bloom filter has not been improved much since its invention in 1970, there are approaches that aim at improving the performance by reducing the fill factor of the filter [8] or by adopting the power of two choices [11]. Although the idea of reducing the fill factor is natural and the proposed schemes are applicable to general membership queries, the performance gains are relatively low compared to the overhead cost. Bruck et al. [3] and Zhong et al. [15] propose using different numbers of hashes to reduce the false positive rate in Bloom filters. Their schemes use data popularity distributions while LABFs depend on data length distributions. Additionally, their schemes require the popularity distribution in both set and membership queries, whereas LABFs need the length distribution in the set only, not requiring the query distribution that is hard to compute in many contexts.

Directly related applications of LABFs include IP address lookup and multiple pattern matching based on Bloom filters [5], [4], [6], [13]. Bloom filters are implemented on embedded on-chip memory for fast string matching and real data are stored on off-chip memory. Positive matches (true or false) indicated by Bloom filters are verified by examining them against the off-chip table. LABFs enhance the lookup perfor-
performance by minimizing the number of off-chip memory accesses caused by false positives. Song et al. [13] later propose distributed and load-balanced Bloom filters that do not depend on the length distribution and enhance parallelism as well. LABFs can easily be modified in a distributed and load balanced manner, and are amenable to hardware implementation.

VI. CONCLUSIONS

We have designed a length-aware Bloom filter that reduces the false positive rate significantly when pattern lengths are non-uniformly distributed in multiple pattern matching. A set of patterns with popular lengths are hashed into the filter using less hash functions while the rest patterns of unpopular lengths are inserted using more hash functions. We observe that IP address prefixes (both IPv4 and IPv6) and the Snort database in network intrusion and detection system exhibit this skewed length distribution characteristic as relevant candidate for LABFs. Unlike prior work that requires considerable preprocessing, LABFs need simple preprocessing in order to configure the optimal number of hash functions, and the complexity of insertion and lookup is comparable to a standard Bloom filter. Our experimental results show that the average false positive rates are decreased by a factor of 4 and 16 for IPv4 and IPv6 longest prefix matching, respectively.

REFERENCES