# Enhancing Counting Bloom Filters Through Huffman-Coded Multilayer Structures

Domenico Ficara, *Member, IEEE*, Andrea Di Pietro, *Student Member, IEEE*, Stefano Giordano, *Senior Member, IEEE*, Gregorio Procissi, *Member, IEEE*, and Fabio Vitucci, *Member, IEEE*

***Abstract*—Bloom Filters are efficient randomized data structures for membership queries on a set with a certain known false positive probability. Counting Bloom Filters (CBFs) allow the same operation on dynamic sets that can be updated via insertions and deletions with larger memory requirements. This paper first presents a simple tight upper bound for counters overflow probability in CBFs, which is adopted in the design of more efficient CBFs. On the basis of such theoretical achievements, we introduce the idea of a hierarchical structure as well as the use of Huffman code to improve standard CBFs in terms of fast access and limited memory consumption (up to 50% of memory saving). The target could be the implementation of the compressed data structures in the small (but fast) local memory or "on-chip SRAM" of devices such as network processors. As an application of our algorithms, an anti-evasion system is finally proposed.**

***Index Terms*—Counting Bloom Filter (CBF), evasion, hashing, Huffman coding, multilayer structure, network processor.**

## I. Introduction

NOWADAYS, streamed data processing is a basic problem in many areas related to computer applications. In particular, detecting whether an item belongs to a set is one of the most challenging tasks, especially when the amount of data to be processed per unit of time is very large and rapidly changes.

A Bloom Filter (BF) is a simple data structure for information representation and query processing. It is a randomized method based on hash functions. Thus, it allows for false positives, but the space savings often outweigh this drawback. BFs were introduced by Burton Bloom [1] in the 1970s for database applications, but recently they have received great attention also in the networking area [2] for collaborating in overlay and peer-to-peer networks, packet routing, and measurements. BFs are also proposed for many distributed networking protocols. For example, in order to share Web cache, a proxy periodically broadcasts BFs that represent the contents of their cache. In this situation, BFs are not only data structures, but also messages being transmitted in a network. Thus, several performance parameters have to be taken into account: the probability of false positives, memory size, number of items to be managed, and transmission size.

BFs do not address the issues of inserting and deleting items in the set. For example, a set may change over time, with elements being inserted and deleted. Deletion cannot be done by simply reversing the insertion operation because of the collisions created by the hash functions. In order to allow these operations, Counting Bloom Filters (CBFs) have been designed [3]. They are based on the same idea as BFs, but they use fixed size counters (also called bins) instead of single bits of presence. When an item is inserted, the corresponding counters are incremented; deletions can then be safely done by decrementing the counters. CBFs present the problem of counters overflow, which has to be considered in the design.

This paper adopts a simple upper bound for the CBF overflow probability, which is functional to the design of new efficient solutions.

The central idea of the paper is that, by leveraging on the bound, a novel paradigm in CBF design can be adopted. Such a paradigm involves *compression*—to improve CBFs in terms of fast access and limited memory consumption (up to 50% of memory saving in comparison with the standard solutions)—and the introduction of layer *hierarchy* in the CBF data structure.

The target could be to take advantage of the built-in memory hierarchy of many systems (such as network processors, NPs) to implement compressed data structures in the small but fast local memory or "on-chip SRAM" of such devices. As an example of the advantages of our compressed CBFs, we propose a compact solution to the detection of evasion attacks to intrusion prevention systems (IPSs).

In detail, the main contributions of this paper (which is an extended version of the work in [4]) are the following:

- the use of Huffman code in CBF, which is optimal for independent symbols (such as the bins of a CBF);
- the idea of a hierarchical multilayer structure;
- the proposal of an efficient CBF for systems with limited memory such as NPs and programmable routers;
- the adoption of these efficient structures in the solution of a difficult task such as recognizing evasion attacks.

This paper is organized in two main parts. First, we describe the proposed algorithms, and then we show a brief example of their application, which is shown in more detail in [5].

The next section presents the most important works on CBFs. In Section III, the theoretical results at the basis of our research

Manuscript received March 25, 2009; revised April 04, 2010; accepted May 28, 2010; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Kodialam. Date of publication August 03, 2010; date of current version December 17, 2010. This work was supported in part by the European Project FP7-ICT PRISM under Contract 215350.

The authors are with Dipartimento di Ingegneria dell'Informazione, Università di Pisa, 56122 Pisa, Italy (e-mail: domenico.ficara@iet.unipi.it; andrea.dipietro@iet.unipi.it; stefano.giordano@iet.unipi.it; gregorio.procissi@iet.unipi.it; fabio.vitucci@iet.unipi.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2010.2055243

are shown, while Section IV introduces the idea of Huffman compression coming from the theoretical achievements previously obtained. The data structure described here represents an intermediate step forward to the central proposal of the paper, which is presented in Section V, in terms of algorithms, properties, and simulation results.

A comparison among our algorithms and the algorithms defined in literature is performed in Section VI, by adopting NP Intel IXP2800 as a referential hardware platform. Finally, the application of our structures to an anti-evasion system is proposed in Section VII, and Section VIII gives the conclusions of our work.

## II. Background on Bloom Filters

A Bloom Filter represents a set $S$ of $n$ elements from a universe $U$ by using an array of $m$ bits, denoted by $B[1], \dots, B[m]$, initially all set to 0. The filter uses $k$ independent hash functions $h_1, \dots, h_k$ with $\log_2(m)$-bits-long output that independently map each element in the universe to a random number uniformly distributed over the range. For each element $x$ in $S$, the bits $B[h_i(x)]$ are set to 1, for $1 \leq i \leq k$ (a bit can be set to 1 multiple times).

To answer a query of the form "*Is y in S?*", we check whether all $B[h_i(y)]$ are set to 1. If not, $y$ is not a member of $S$, by construction. If all $B[h_i(y)]$ are set to 1, it is assumed that $y$ is in $S$, hence a BF may yield a false positive. The probability of a false positive $f$ can be tuned by choosing the proper values for $m$ and $k$. It is a well-known result [3] that the minimum $f$ is obtained for $k = (m/n) \ln 2$. In this configuration, all bits $B[1], \dots, B[m]$ are set or cleared with probability $p = 1/2$ (thus, roughly, the same number of ones and zeros are present in the BF).

Many works about BFs have been presented, and the major improvements are compressed BFs [6], distance-sensitive BFs [7], dynamic BFs [8], and space-code BFs [9].

As previously stated, BFs do not allow insertion and deletion of an item in the set. Therefore, CBFs have been introduced, which use $m$ fixed size bins instead of $m$ single bits of presence. When an item is inserted (or deleted), the corresponding counters are incremented (or decremented).

However, CBFs present the problem of counters overflow, which has to be considered in the design. Although for most network applications 4-bits-long counters are sufficient [2], the distribution of counters load across bins changes dramatically (according to Poisson arrivals [3]), suggesting that 4 bits per bin is a safe choice, and that a certain amount of compression is achievable. Moreover, by using a fixed number of bits, the problem of counters overflow in CBFs is not completely solved. It results in a lack of adaptiveness and inaccuracy of stored information.

In order to waive these limitations and achieve better performance, many improvements to CBFs have been done. Mitzenmacher [6] shows that unbalancing the number of ones and zeros in a standard BF can help achieve a good compression ratio before transmission (e.g., for Web-caching application). This way, by keeping the same amount of bits of the uncompressed case, it is possible to either reduce the false positive probability or use a lower number of hash functions.

Spectral Bloom Filters (SBFs) [10] are an extension of standard BFs to multisets, allowing estimates of the multiplicities of individual items with small error probabilities. The word "spectral" means that SBFs allow only filtering of elements whose multiplicities are within a requested spectrum (therefore they do not preserve bins from overflow in a conclusive way). The main goal of SBFs is the optimal counter space allocation, so they dynamically vary the size of their counters in order to minimize the number of necessary bits. To achieve this flexibility, SBFs include additional slack bits among the counters and complex index structures, which increase both memory needs and access time as compared to standard CBFs. Finally, SBFs introduce techniques for filter compression based on Elias code that reduce the transmission size of data structures but increase again the processing load.

Dynamic count filters (DCFs) [11] are data structures designed for speed and adaptiveness in a very simple way. They do not require the use of indexes, thus obtaining a fast access time, and permanently avoid counters overflow. DCFs consist of two different vectors: The first one is a basic CBF with counters of fixed size, the second one is the Overflow Counter Vector, which has a counter for each element of first vector that keeps track of the number of overflow events. The size of counters in the Overflow Counter Vector changes dynamically to avoid saturation. This implies that, for each update, a structure rebuilding is required. Moreover, the decision of having the same size for all these counters (for direct access) entails that many bits are not used. Therefore, this solution can be improved, especially in terms of memory consumption.

The d-left CBFs (dlCBFs) [12] are simple alternatives based on d-left hashing and fingerprints of bins. They do not rely on the principles of Bloom Filters, but they offer the same functionalities. The dlCBFs use less space, generally saving a factor of two or more for the same fraction of false positives, and the construction is very simple and practical, much like the original Bloom Filter construction. Indeed, the simplicity in constructing and maintaining data structures is maybe the greatest contribution of [12] as compared to previous works. Moreover, even dlCBFs have the limitation of potential counters overflow and the need for an additional fingerprint for each bin in the data structure.

A successive proposal [13] advocates the use of rank indexing to achieve compact representations of BFs and CBFs through a hierarchical construction. The main idea of this proposal is to implement a CBF as a hash table where a fingerprint (hash value) for each key is stored. Even if the data structure does not actually perform any counting operation, dynamic insertions and deletions from the set are supported. The authors use several layers of bitmaps to avoid the overhead associated to the canonical pointer based implementation. While the use of multilayer bitmaps suggests a similarity with our work, its focus is significantly different, as it does not really provide "counting" functionalities and it cannot support multisets.

The memory utilization is the parameter that is better taken into account in this work. As previously mentioned, there are several cases where network bandwidth is still expensive and transmission size becomes a fundamental parameter (e.g., Web cache sharing or P2P applications). Moreover, although memory appears plentiful today, there are many hardware architectures used in network devices (e.g., network processors) that may take advantage of using very space-efficient data structures, in terms of both performance and costs. Indeed, memory saving can greatly speed up a device by requiring rare access to slower off-chip memory. Furthermore, while ordinarily DRAM memory is cheap, fast SRAM memory and especially on-chip SRAM continue to be comparatively scarce. All these issues have led our research, which had the target of an efficient and practical data structure for CBF.

## III. Theoretical Results

In this section, we present the main theoretical results on the CBF counter overflow probability and on Huffman coding of bin counters that will be the basis of the data structures proposed in the rest of the paper.

The following classical result [2] on CBF gives a bound on the overflow probability $P(\varphi \geq j)$ that is widely adopted to design the bin size:

$$P(\varphi \geq j) \leq \left(\frac{enk}{jm}\right)^j. \tag{1}$$

However, (1) is pretty loose. Theorem 1 presents a tighter bound for $P(\varphi \geq j)$.

*Lemma 1:* Let $\varphi$ be a CBF counter value, and $\alpha = nk/m - 1$. If $\alpha < 1$, the function $\chi(j) = P(\varphi = j)$ is a monotonically decreasing function.

*Proof:* The probability of the event $\{\varphi = j\}$, for $j \geq 1$, is given [2] by

$$\chi(j) = \binom{nk}{j}\left(\frac{1}{m}\right)^j\left(1-\frac{1}{m}\right)^{nk-j}.$$

The ratio between two consecutive values is

$$\frac{\chi(j+1)}{\chi(j)} = \frac{nk-j}{j+1}\frac{1}{m-1} < \frac{\alpha}{j+1} < 1 \tag{2}$$

which gives the proof. ■

For $k = (m/n)\ln 2$, $\alpha = (m \times \ln 2)/(m-1)$. $\alpha$ is less than 1 for $m > (1-\ln 2)^{-1} \approx 3.26$. In the CBFs, the previous condition is always satisfied since $m \gg 1$.

*Theorem 1:* Let $\varphi$ be a CBF counter value, and $\alpha = nk/m - 1$. If the number of hash functions is chosen so as to minimize the probability $f$ of false positive (i.e., $k = (m/n)\ln 2$), then

$$P(\varphi \geq j) < \frac{\alpha(j+1)}{j(j+1-\alpha)}P(\varphi = j-1).$$

*Proof:* By repeatedly applying (2)

$$P(\varphi \geq j) = \sum_{i=j}^{+\infty} P(\varphi = i) < P(\varphi = j)\sum_{i=0}^{+\infty}\frac{j!\alpha^i}{(j+i)!}. \tag{3}$$

The right-hand sum of (3) can be bounded as

$$\sum_{i=0}^{+\infty}\frac{j!\alpha^i}{(j+i)!} < \sum_{i=0}^{+\infty}\left(\frac{\alpha}{j+1}\right)^i = \frac{j+1}{j+1-\alpha}$$

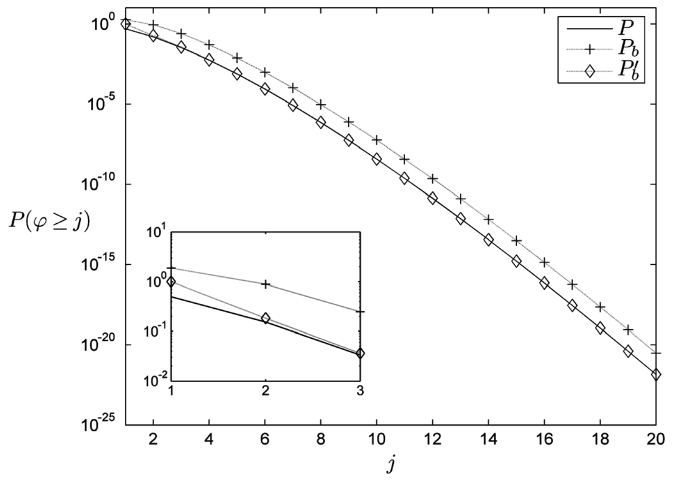

Fig. 1. Bounds comparison for $n = 1000, k = 10$, and $m = nk/\ln 2$. $P$ is the actual $P(\varphi \geq j)$, $P_b$ is the well-known (1), while $P_b'$ is that provided by Theorem 1. In the smaller graph, a zoom on the contour of $j = 2$. $P_b'$ is always tighter than $P_b$.

to finally obtain

$$P(\varphi \geq j) < \frac{\alpha(j+1)}{j(j+1-\alpha)}P(\varphi = j-1). \tag{4}$$

■

*Corollary 1:* Under the previous results, if $\alpha < 1$

$$P(\varphi > j) < P(\varphi = j-1). \tag{5}$$

*Proof:* From (3), by changing the lower limit of the series from 0 to 1, we obtain

$$P(\varphi > j) < \frac{\alpha^2}{j(j+1-\alpha)}P(\varphi = j-1). \tag{6}$$

Then, considering that $j \geq 1$, $\alpha^2/(j(j+1-\alpha)) < 1$. ■

Lemma 1 allows to approximate $P(\varphi = 0)$ and $\mathbb{E}[\varphi]$

$$1 = \sum_{j=0}^{+\infty} P(\varphi = j) \simeq P(\varphi = 0)\sum_{j=0}^{\infty}\frac{\alpha^j}{j!} = P(\varphi = 0)e^{\alpha}. \tag{7}$$

Then, $P(\varphi = 0) \simeq e^{-\alpha}$. As for the expectation of $\varphi$, we get

$$\mathbb{E}[\varphi] = \sum_{j=0}^{+\infty} jP(\varphi = j) \simeq P(\varphi = 0)\sum_{j=0}^{\infty} j\frac{\alpha^j}{j!} = \alpha. \tag{8}$$

If the CBF minimizes $f$, $\mathbb{E}[\varphi] \simeq \ln 2 = 0.693$, which is a very tight approximation in several cases.

It is interesting to see that, as shown in Fig. 1, the previous bound can be much tighter than the widely used (1). For instance, if $n = 1000$, $k = 10$ and $m = nk/\ln 2$, (1) yields $P(j > 15) \leq 1.37 \times 10^{-15}$, while our bound produces $P(j > 15) < 1.51 \times 10^{-16}$, with a gain of an order of magnitude. Moreover, the results of this first theorem are the basis for the following one.

*Observation 1:* Let $H(\sigma)$ be the Huffman coding of $\sigma$, $len(\cdot)$ the "bit-length" operator, and $\varphi$ a CBF counter value. Then

$$len(H(\varphi)) = \varphi + 1.$$

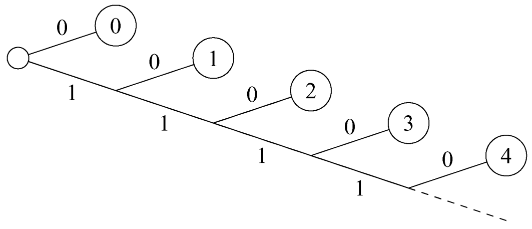

Fig. 2. Huffman tree for the CBF bin counters.

Indeed, Huffman codes can be obtained by using a binary tree. The tree is constructed from a list of $N$ nodes (symbols) whose weights correspond to the symbol probabilities.

The whole procedure is the following.

- Let $x$ and $y$ be the two nodes with the lowest weight.
- $x$ and $y$ are aggregated into a parent node whose weight is set to the sum of the two nodes.
- The parent node replaces $x$ and $y$ in the list.

These steps are repeated until the list contains one node only.

To perform Huffman coding of CBF bin counters, we first construct a tree whose nodes $X_0, \dots, X_N$ correspond to the possible values of the counters $j = 0, \dots, N$; the weight of the $j$th node is set to $P(\varphi = j)$. Let $L_\tau$ be the list of nodes at step $\tau$ and let $X_\tau$ be the parent node to be created at this step. Suppose we have $L_\tau = \{X_0, X_1, \dots, X_{N-\tau-1}, X_{\tau-1}\}$. The weight of the parent node $X_{\tau-1}$ created at the previous step is $P(X_{\tau-1}) = P(j > N - \tau - 1)$.

By using the result of corollary 1, we obtain

$$P(X_{\tau-1}) < P(j = N - \tau - 2).$$

Moreover, the previous inequality also implies that $P(X_{\tau-1})$ is smaller than any of the values in the set $\{P(X_0), \dots, P(X_{N-\tau-2})\}$. Then, at step $\tau$, the nodes with the smallest weights are $X_{\tau-1}$ and $X_{N-\tau-1}$, and they shall be aggregated into the parent node $X_\tau$. Thus

$$\begin{aligned} L_\tau &= \{X_0, \dots, X_{N-\tau-2}, X_{N-\tau-1}, X_{\tau-1}\} \Rightarrow \\ \Rightarrow L_{\tau+1} &= \{X_0, \dots, X_{N-\tau-2}, X_\tau\} \end{aligned}$$

The resulting tree turns out to be completely unbalanced (i.e., the depth of all $N$ nodes is given by the sequence of the first $N$ naturals) such as the one of Fig. 2. Therefore, the depth of node $\varphi$ is $\varphi + 1$, i.e., the encoding of the value $\varphi$ of a CBF counter is $\varphi + 1$ bits long. This result, which comes in turn from the results of Theorem I, is one of the basic principles of our structures.

## IV. Huffman Counting Bloom Filters

The target of our data structures is an improvement of CBFs by avoiding counters overflow and reducing memory needs. The drawback, as we will see below, is a very slight increase of complexity for the insertion/deletion of an element.

The first step toward the above-mentioned target (that will be fully accomplished through the layered structure presented in the next section) involves the use of Huffman coding in CBF. The result is Huffman counting Bloom Filter (HCBF). In order to introduce this data structure, we begin by recalling spectral Bloom Filters [10].

They use a memory-efficient structure that encodes any bin with Elias coding. This way, bins do not have a fixed position and, for all $k$ hash functions, we have to find the right bin it points to by looking up a certain amount of words. Lookup implies to decode a number of bins until the right one is found. Moreover each insertion and deletion imply a potential shift of the whole structure.

To simplify these operations, SBFs divide the entire structure in subsegments and use a set of tables in aid to the lookup. In addition, a certain number $\varepsilon$ of empty bits (called slack bits) are inserted to reduce shifts operations for insertions and deletions. Elias compression scheme is a perfect choice when dealing with large numbers, such as those of multiset membership query applications. However, for smaller values (recall that in a regular CBF, 16 is widely considered as a high loose bound), other codings can perform better. By leveraging on Observation 1 of the previous section, our proposal is to encode a number $\sigma$ with $\sigma$ consecutive ones and a trailing zero (Fig. 2). This way, the encoding produces $\sigma + 1$ bits for symbol $\sigma$: it is a Huffman coding, as shown in Section III. This is a major advantage since Huffman is the minimum redundancy coding for independent symbols such as the bins of a CBF.

Moreover, our coding scheme allows an easy lookup since most processors provide an instruction that counts the number of bits set to 1 in a word (*popcount*). By taking advantage of such an instruction, we do not have to decode each value we find during lookup, but simply count the number of cleared bits in a word. The number of cleared bits is the number of symbols encoded in that word (see example in Fig. 3). Clearly, we still have to perform a shift for each insertion or deletion, and we need a table to speed up lookup, but the total size of the structure is very close to the minimum (given by the entropy of all symbols).

## A. Size

In order to simplify the operations and reduce the cost of lookups and insertions/deletions, we group the bins in $B$ blocks of $D$ bins (with few slack bits), and we address the blocks with the table. The average size of the HCBF is

$$\mathbb{E}[S] = m\,(1 + \mathbb{E}[\varphi]) + B\,(\varepsilon + \log_2[(m - D)(\varphi_{\max} + 1)])$$

where $\varepsilon$ is the number of slack bits kept at the end of each block. The last part of the above formula takes into account the table size. The table is addressed by the first $\log_2 B$ bits of the hash, and the remaining bits represent the bin index. Each entry of the

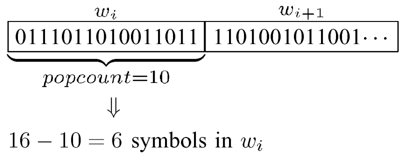

Fig. 3. Example of fast lookup through popcount.

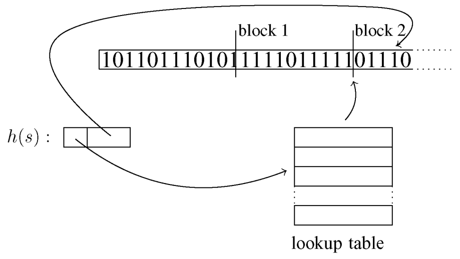

Fig. 4. Example of HCBF.

table represents the starting address of the corresponding block thus requiring less than $(m-D)(\varphi_{\max}+1)$ bit.

## B. Lookup

As for operation complexity, a lookup requires $k$ hash functions and, for each of them, a check in the table and a search in the corresponding block for the bin we need (see Fig. 4). Thus, on average, $D/2$ bins have to be looked, and $W/(\mathbb{E}[\varphi]+1)$ bins will be found in a word of $W$ bits. The overall average number of operations for a lookup is then

$$\omega = k\left(\frac{D\left(\mathbb{E}[\varphi]+1\right)}{2W}\right).$$

As shown in Section III, $\mathbb{E}[\varphi] \simeq \ln 2$. Therefore, the average number of operations for a lookup is constant and its complexity is $O(1)$.

## C. Insertion/Deletion

In order to insert a new element, we need to perform a lookup and to add a "1" digit for each bin in the code. This corresponds to shifting all the bits at the bin's right by one position and a table update. Thus, for all insertions, the number of operations is

$$\omega = k\left(\frac{D\left(\mathbb{E}[\varphi]+1\right)}{W}\right).$$

It is straightforward to see that, since even deletion requires a lookup and a shift, the overall cost is the same as insertion. The complexity of these operations is $O(1)$, as for lookup.

## V. Multilayer Compressed CBF

The drawbacks of the algorithm described in Section IV, as well as SBF, are related to the memory wastage due to slack bits and to the complexity of a searching based lookup (even if aided by index tables).

In the following, the multilayer compressed counting Bloom Filter (ML-CCBF) is presented, which is a CBF that reduces the memory requirements and the complexity of lookup. The idea is to explode the CBF along another dimension, hence creating a *multilayer* structure, where, for each encoded symbol, a bit per layer is stored. This construction, in conjunction with the Huffman coding defined in Section IV, provides a stack of bitmaps $(L_0, \ldots, L_N)$, where the first layer $L_0$ is a standard BF. The other layers are built and modified dynamically when needed. The relationship between ML-CCBF and the previously described HCBF can be expressed in a few words by saying that ML-CCBF is somewhat the rotated version of HCBF, with

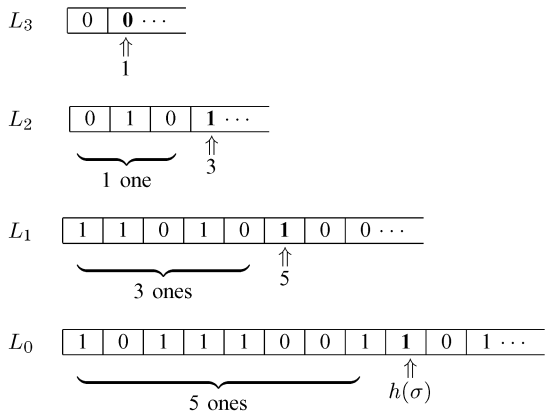

Fig. 5. ML-CCBF example. The resulting Huffman code for $\varphi$ is 1110.

all bits representing the Huffman-coded values of counters in HCBF placed in ascending layers.

To the best of our knowledge, although a limited degree of hierarchy is sometimes obtained by adding a CAM [14] or another counter [11], this is the first attempt to introduce the idea of a hierarchy of arrays in CBFs, which results in a *multilayer* structure where counters may span over different levels.

Let $popcount(u)$ be the number of 1's in the bitmap $(0, \ldots, u-1)$. The construction is as follows.

- $L_i$ keeps all the $i$th binary digits of our Huffman-encoded counters.
- On $L_i$, the $j$th bit belongs to the counter whose $popcount$ on $L_{i-1}$ is $j$.

Fig. 5 shows an example of a ML-CCBF. In the example, we are counting a bin $\varphi$ for symbol $\sigma$. The bin at layer 0 is pointed by the hash function $h(\sigma)$. The number of ones before $h(\sigma)$ is computed (i.e., $popcount(h(\sigma)) = 5$) and used as an index for layer 1. The procedure is repeated until we find a "0" digit (that is the end of the code). Therefore, the resulting Huffman code for the counter is 1110, which corresponds to value 3.

## A. Complexity and Properties

One of the most significant advantages of our algorithm is that it is an extension of a standard BF. Thus, the lookup is as simple and fast as in a standard BF since we need to check only bits at layer 0. Therefore, the lookup complexity is $O(1)$.

Instead, for insertions and deletions, we need to explore different layers in the structure. We refer to $m_i$ as the number of bits in layer $i$. The size of layer $i$ can be obtained as

$$m_i = m_0 P(\varphi \geq i).$$

The above formula provides a useful mean for dimensioning the overall data structure. As the (binomial) distribution of counters is known, the maximum length of each layer can be estimated, and the corresponding memory allocated accordingly. Also, the formula allows to allocate the number of levels as well by selecting the number for which the probability of overflow is negligible. In addition, when multiset has to be supported, the maximum cardinality for a key has to be taken into account.

Since jumping one layer up requires a *popcount* on a potentially large number of bits, we divide all layers in blocks of the same bit-size $D$ and add a table for each level. When computing

$popcount(u_j)$ at layer $j$, the first $\log_2(m_j/D)$ bits of $u_j$ are used as the index to table $j$. Each entry of the table represents the number of ones preceding the start of the block. Thus, if $W$ is the number of bits in a word, the actual *popcount* operation works only on less than $D/W$ words. Therefore, the average cost of a *popcount* is $1 + D/2W$.

Algorithms 1 and 2 show the pseudocode for insertion and deletion procedures in a ML-CCBF. Both operations require, for all $k$ bins, the complete lookup of multiplicity (by exploring a certain amount of layers), a shift by one position, and the update of the last explored table. Such an update simply consists of an increment or a decrement on a limited number of entries. Therefore, the average number of operations for insertion and deletion is given by

$$\omega = k\left[\mathbb{E}[\varphi]\left(1+\frac{D}{2W}\right)+2\right].$$

Once again, $\mathbb{E}[\varphi] \simeq \ln 2$, thus the average amount of operations is fixed and the complexity for insertion/deletion is $O(1)$.

**Algorithm 1** The insertion of an element in a ML-CCBF

```
1: for $i \leftarrow 1, k$ do
2:     $j \leftarrow 0$
3:     $u_0 \leftarrow h_i(s)$
4:     while $(L_j(u_j) = 1)$ do
5:         $u_{j+1} \leftarrow popcount(u_j)$
6:         $j \leftarrow j + 1$
7:     end while
8:     $L_j(u_j) \leftarrow 1$
9:     $u_{j+1} \leftarrow popcount(u_j)$
10:    $j \leftarrow j + 1$
11:    $L_j(u_j + 1, \ldots, m_j + 1) \leftarrow L_j(u_j, \ldots, m_j)$
12:    $m_j \leftarrow m_j + 1$
13:    $L_j(u_j) \leftarrow 0$
14:    UpdateTable $(L_j)$
15: end for
```

**Algorithm 2** The deletion of an element in a ML-CCBF

```
1: for $i \leftarrow 1, k$ do
2:     $j \leftarrow 0$
3:     $u_0 \leftarrow h_i(s)$
4:     while do $(L_j(u_j) = 1)$
5:         $u_{j+1} \leftarrow popcount(u_j)$
6:         $j \leftarrow j + 1$
7:     end while
8:     $L_j(u_j, \ldots, m_j) \leftarrow L_j(u_j + 1, \ldots, m_j + 1)$
9:     $m_j \leftarrow m_j - 1$
10:    $L_{j-1}(u_{j-1}) \leftarrow 0$
11:    UpdateTable $(L_j)$
12: end for
```

A major advantage of ML-CCBF over HCBF and SBF comes from having update and lookup operations decoupled: All insertions/deletions work with higher layers or may flip some bits in the bottom layer 0, requiring no shift nor enlargement of layer 0. This means that we can still perform lookups during data set updates if we just take precautions (by means of mutexes) when dealing with changes in the bottom layer (the BF).

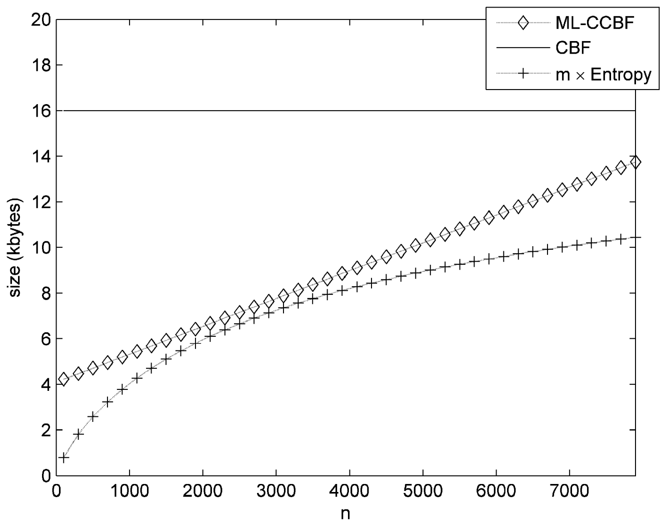

Fig. 6. Size comparison among ML-CCBF, CBF, and $m \times \mathrm{Entropy}$.

## B. Size

ML-CCBF is a multilayer transposition of the algorithm shown in Section IV, with no need for slack bits. Hence, it results in a lower memory requirement

$$S = m_0 + \sum_{i=1}^{m_0} \varphi_i + \sum_{i=1}^{n_{\mathrm{tab}}} TS_i.$$

$TS_i$ is the size of the table required for layer $i$, which needs $n_i = \lceil m_i/D \rceil$ entries of size $\log_2(m_i)$, thus resulting in

$$TS_i = n_i \log_2(m_i) = \left\lceil \frac{m_0}{D} \right\rceil P(\varphi \geq i) \log_2\left[m_0 P(\varphi \geq i)\right].$$

The average amount of required memory is then

$$\mathbb{E}[S] = m_0\left(1 + \mathbb{E}[\varphi]\right) + TS.$$

A closed-form expression for $TS = \sum_{i=1}^{n_{\mathrm{tab}}} TS_i$ is not simple to obtain in a general case. However, we use the results of Theorem 1 to compute a bound for $TS$.

If $\alpha = \ln 2$ to minimize the false positive probability, then

$$TS \leq \left\lceil \frac{m_0}{D} \right\rceil (2\log_2(m_0) - 1.85).$$

Clearly, as updates occur, upper layers may change in size, thus requiring some extent of overprovisioning or memory dynamic allocation schemes. However, those layers are designed to be placed in memories whose sizes are not critical (as opposed to layer 0), and this does not affect the overall scheme properties.

Fig. 6 shows the comparison among the sizes of ML-CCBF, standard CBF, and the minimum amount of bits for independent symbols (BF Entropy $= m \times \mathrm{Entropy}$), for $k = 10$ and $m = 32\,768$ (notice that $m$ is fixed regardless of $n$; therefore, the probability of false positives $f$ is not minimized). The memory

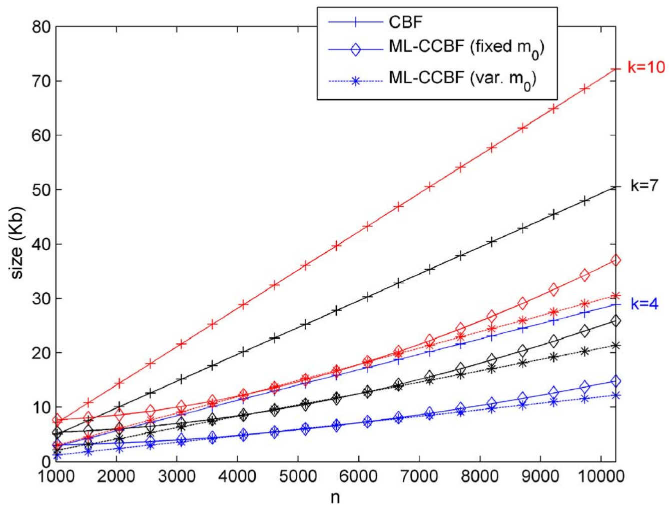

Fig. 7. Size comparison between CBF and ML-CCBF (for fixed and variable number of bits $m_0$ for layer 0).

saving of our method is clear as it approaches the minimum value. Note that the optimal number of elements $n = 2270$ (i.e., the value that minimizes $f$) minimizes the distance from the BF entropy as well.

Fig. 7 instead reports, for $k = 4, 7, 10$, the curve of the structure size (in kilobytes) for various numbers of elements ($n = 1024, \ldots, 10\,240$) between a standard CBF (constructed so as to minimize false positives) and a ML-CCBF constructed with a fixed layer 0 or with a variable layer 0. For the latter, $m_0$ has been set as minimizing false positives (i.e., $m_0 = n \times k/\ln 2$), while for ML-CCBFs with a fixed layer 0, $m_0$ has been set to $n'k/\ln 2$ (with $n' = 4096$). Setting the size $m_0$ of the bottom layer of ML-CCBF (basically the corresponding BF) as fixed is an easy and fast way to construct the structure, but it does not provide the best results in terms of false positives and memory efficiency. However, the figure shows that even for $n \simeq 2n'$ (i.e., for twice the optimal number of elements), the size penalty for a ML-CCBF with fixed $m_0$ is limited to less than 20% with respect to an optimal (i.e., with variable $m_0$) construction. Moreover, it is noteworthy that the difference between the two types of construction (in terms of size) is minimal, thus showing that the choice of $n'$ (and hence of $m_0$) in the scenario with a fixed layer 0 is not critical.

## VI. Comparative Analysis

For the evaluation of the algorithms proposed in this paper and the comparison to others known in literature, the network processor Intel IXP2800 has been taken as referential hardware architecture. NPs are platforms that offer very high packet processing capabilities (e.g., for gigabit networks) and combine the programmability of general-purpose processors with the high performance typical of hardware-based solutions. The IXP2800 is designed to perform a wide range of functionalities, including multiservice switches, routers, and broadband access devices. It is a fully programmable network processor, characterized by a hierarchy of processing units (an XScale core and 16 32-bit microengines MEv2) and memory devices (4 kB of local memory, 16 kB of scratchpad memory, besides external memories of the host card). The bigger the memory, the slower the access to it. For more details about Intel IXP2800, we refer to [15].

TABLE I
Number of Clock Cycles for Operations in the IXP2800

| Operations | Number of cycles |
|---|---|
| hash | 10 |
| popcount | 1 |
| shift | 1 |
| read/write in local memory | 2 |
| read/write in scratchpad memory | 60 |

The hierarchy of memory devices in the IXP2800 reflects the memory architecture of many systems, which present small fast memories and slower big ones. Therefore, although referred to a certain hardware platform, the results of our research are very general.

As shown in Table I, we have weighted, according to the IXP2800 hardware reference manual [16], the operations of the algorithms in terms of clock cycles for microengines (which are the processors designed to handle fast data path).

In the analysis, we always considered a few clock cycles for emptying the pipeline from all operations. Indeed, all costs reported in the tables are *average costs*. They are not the minimal costs (those that can be achieved when the pipeline is full and no additional costs are paid for switching among operations). For example, to access local memory, if pointers are all set, only one clock cycle is needed, while the cost of setting pointers is three clock cycles. We always conservatively estimated two clock cycles penalty. In fact, generally once the pointer is set, one can access long words (LWs) around with no extra costs (in terms of clock cycles). This is actually a very common case, particularly when working with ML-CCBF and HCBF, where typically LWs to be processed are next to each other.

Each algorithm has been simulated, and its performance has been measured in terms of memory consumption and processing load for lookup and insertion/deletion. In simulation runs, the total number of data elements is $n = 2000$, $k = 10$, and the number of bins for the main vector is $2.8 \times 10^4$, thus minimizing the probability of false positives. For the algorithms that divide data structure in subsegments, the number of blocks is $B = 64$. All other parameters are set to obtain about the same probability of false positives among the different algorithms and to be able to manage the same number $n$ of elements. Moreover, for the algorithms which present a hierarchical structure, we have located each substructure in the fastest memory as possible (see Table II).

Concerning ML-CCBF, the main BF vector $L_0$ and index tables are stored in local memory, while the remaining vectors in scratchpad. A lookup only requires checking the first vector, therefore only local memory is accessed. For insertion and deletion, we still need to explore different layers in the structure, thus both memories are accessed.

For a standard CBF, built with 4 bits for bin, the overall structure has been located in scratchpad. Therefore lookup, insertion, and deletion require accesses to this memory.

With the data of our simulation, DCF (see Section II) does not experiment any overflow of counters in CBF vector. Therefore, Overflow Counter Vector is not necessary, and DCF exhibits

TABLE II
PERFORMANCE ALGORITHMS COMPARISON

| | ML-CCBF | ML-CCBF | CBF | DCF | HCBF | SBF | dlCBF |
|---|---|---|---|---|---|---|---|
| Size (KB) | 6.13 | 6.13 | 14.1 | 14.1 | 6.42 | 12.12 | 5.2 |
| Main structure (KB) | 3.52 (local) | 6.13 (scratch.) | 14.1 (scratch.) | 14.1 (scratch.) | 5.92 (scratch.) | 8.12 (scratch.) | 5.2 (scratch.) |
| Secondary structures (KB) | 2.4 (scratch.) | - | - | - | - | - | - |
| Index tables (KB) | 0.21 (local) | - | - | - | 0.5 (local) | 4 (local) | - |
| Probability of false positives | $10^{-3}$ | $10^{-3}$ | $10^{-3}$ | $10^{-3}$ | $10^{-3}$ | $10^{-3}$ | $1.5 \times 10^{-3}$ |
| Lookup (clock cycles) | 120 | 700 | 700 | 700 | 780 | 801 | 800 |
| Insertion/Deletion (clock cycles) | 1064 | 1893 | 710 | 710 | 1058 | 1217 | 810 |

exactly the same behavior of CBF in terms of both size and complexity.

Regarding HCBF, we have stored the main structure in scratchpad and the index tables in local memory. As mentioned in Section IV, a lookup requires, for each hash function, to check the table in local memory, to search for the corresponding block in scratchpad for the bin we need, and to compute a popcount. The same number of operations are required for inserting/deleting an element, with the addition of shifting by one position the bits in the bin to increment/decrement a counter. Remember that HCBF is a simple alternative version of SBF, which is a structure optimized for multiset. SBFs use, for values greater than 2, Elias code instead of Huffman code and several more index tables, thus resulting in higher memory consumption and operational complexity.

Finally, the overall unique structure of dlCBF has been located in scratchpad. A lookup requires $k$ hashes, $k$ permutations, and $k$ accesses to scratchpad, while an insertion or a deletion needs the same operations for locating the candidate bins, $k$ accesses to scratchpad to find the right bin, and finally, depending on the counter value, either one incrementing (or decrementing) operation or the insertion (removal) of a new fingerprint and its associated counter.

As for rank-indexed hashing, the functionality it provides is somehow different from those offered by the other CBFs. Although it does support dynamic insertion or deletion of elements and it certainly provides good compression, it does not really support counting functionalities, and it cannot keep track of multisets. Therefore, it is not suitable for all of the applications that use CBFs. Moreover, as the lookup operation in this data structure is equivalent to walking through a list, it cannot be parallelized as in the case of the other solutions, where the locations specified by multiple hash functions can be accessed independently by different cores at the same time. For this reason, we believe that a direct comparison (in terms of plain clock cycles) is not fair to the other algorithms, which can be simply sped up through parallelization.

From results in Table II, it is clear that the solutions proposed in this paper show a significant memory saving in comparison to standard CBF and DCF (savings of 56% for ML-CCBF and 54% for HCBF) and also SBF. Instead, there is a memory consumption increase in comparison to dlCBF (from 0.93 up to 1.22 kB). Hovewer, our methods, inspired by dynamic approaches (e.g., DCF), avoid in a conclusive way the problem of counters overflow, thus preserving the accuracy of stored information. This makes our data structure suitable for keeping track of multisets: Inserting the same element several times on a CBF can rapidly lead to overflow, especially with architectures that, like dlCBF, put a lot of effort into reducing the size of the counters (counters can be as small as 2 bits). With our solution, adding keys just requires adding more layers and results in an increased memory footprint.

Moreover, the introduction of a hierarchical structure allows in ML-CCBF a remarkable decrease of clock cycles for the lookup operation. Indeed, the main structure is stored in local memory, thus enabling lookup by accessing local memory only. Naturally, keeping the whole state required by ML-CCBF in the same cache level reduces the performance boost that is provided by the structure layerization, thus negatively impacting the overall performance: The lookup time is the same as a standard BF, and the insertion/deletion time is increased. The membership query is the most frequent operation for these data structures. Therefore, the reduction of about 83% of clock cycles for lookup is a great outcome. It outweighs the drawback of an increase of 50% of processing for inserting/deleting an element.

Performance results indeed show that ML-CCBF cannot be the best solution when high update rates are requested. This, indeed, is the cost to pay for flexibility that, if on one hand guarantees no overflow, on the other hand requires a few extra operations for inserting (and deleting) entries. Clearly, this suggests the use of ML-CCBF in applications that require very fast lookup but reasonably frequent updates (as in the next section application)

Finally, note that our HCBF outperforms SBF in terms of memory consumption and operational complexity. This is an expected result due to the simplicity of our method and to the use of Huffman code (SBFs are optimized for multisets). If compared to the complexity of standard algorithms, HCBF shows a reduction of 13% for lookup and an increase of 45% for insertion/deletion. The different frequency of operations allows to claim that the tradeoff is advantageous.

## VII. APPLICATIONS OF ML-CCBF

ML-CCBF can be adopted for many purposes. In this section, we propose a possible scheme based on such a filter that addresses the issue of evasion attacks.

### A. *Anti-Evasion*

The recent techniques of pattern matching involve the use of finite automata (FA) [17]–[20], hybrid schemes such as Aho–Corasick–Boyer–Moore [21], or hardware architectures that use field programmable gate arrays (FPGAs) or TCAMs. Recently, BFs and CBFs have also been used for pattern matching [22], [23].

TABLE III
PERFORMANCE OF OUR SYSTEM IN TERMS OF DETECTED ATTACKS AND FALSE POSITIVES

| Trace | Size (MB) | Normal attacks | | | Small attacks | | | Total attacks | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | generated | detected | false pos. | generated | detected | false pos. | generated | detected | false pos. |
| Tr1 | 224 | 4240 | 99.8% | 0.23% | 312 | 100% | 4.8% | 4552 | 99.8% | 0.5% |
| Tr2 | 190 | 3800 | 98.9% | 0.1% | 310 | 100% | 4.8% | 4110 | 99% | 0.46% |
| Tr3 | 160 | 1418 | 99.9% | 0.35% | 200 | 100% | 3.5% | 1618 | 99.9% | 0.74% |
| Tr4 | 100 | 1213 | 100% | 0.32% | 185 | 100% | 4.3% | 1398 | 100% | 0.85% |
| Tr5 | 50 | 789 | 99.2% | 0.25% | 108 | 100% | 6.4% | 897 | 99.2% | 1% |

However, some research works [24], [25] show how to evade standard pattern matching techniques by splitting the attack into several packets. Currently, the only way to deal with this problem is to reassemble the overall flow and afterwards apply standard pattern matching algorithms. However, this dramatically increases requirements for security systems in terms of both memory and processing power.

Some works try to avoid the need for flow reassembly. In [26], a combination of parallel BFs is used; each of them "performs" the search for a certain string length. When a packet arrives, a complete check is performed on all the filters. If a match is detected, the flow state becomes suspicious, and the flow is passed to an analyzer for a further deterministic check.

Also, the basic idea of [27] is to split the signatures to be searched by pattern matching into small substrings. In this way, if a sufficiently large piece is completely inserted by an attacker in a packet, it is easily detected. Otherwise, the attacker is forced to use several very small or out-of-order packets, and such abnormal behaviors are revealed by adopting proper heuristics.

## B. Using ML-CCBF for Anti-Evasion

As mentioned, BFs and CBFs provide interesting features for pattern matching and anti-evasion. In particular, CBFs provide the capabilities of quickly updating the set they represent and counting the occurrences of elements. The first property can be used to rapidly take into account each new virus definition with no need to rebuild the overall structure. Instead, counting the occurrences of elements allows CBF to represent the different substrings constituting a string: The arrival of any pieces belonging to the string triggers a decrease of the proper bins, and when the filter is completely reset to zero, the overall match is detected.

ML-CCBF provides further interesting features for fast pattern matching. For instance, its multilayer structure allows to speed up a processing engine. The first layer, which is used for the frequent lookups (all of the packets have to be checked and, at line speed, the time budget is strict), can be put in a fast and small memory, while the other layers, useful for string set updates, can be stored in a slower memory. As most of the traffic is benign and will be discarded by the first stage, the performance requirements of this second layer of filters are much looser. Therefore ,we propose an anti-evasion system that takes advantage of ML-CCBF properties.

As shown in Fig. 8, it is composed of several modules. At first, traffic flows are divided by a classifier according to transport protocols and forwarded to different engines, named substring detectors (SDs). Such a first division allows to balance the load among the SDs and decrease the size of their relative filters.

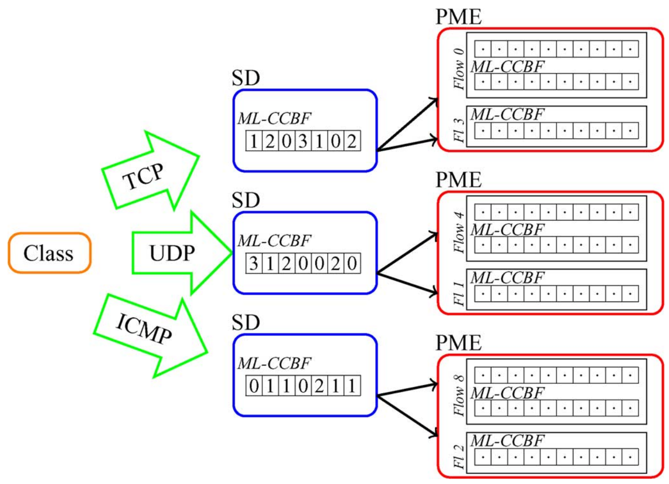

Fig. 8. Scheme of our system.

Each SD performs a pattern matching on the overall content of packets by using specific ML-CCBFs. Each filter represents the set of substrings constituting the strings that identify the valid attacks for that protocol. More precisely, ML-CCBFs represent all the substrings of 3 bytes in order to also reveal the shortest strings, which are 6 bytes long (as pointed out from the analysis of SNORT data sets). The use of ML-CCBFs in this phase allows for a fast update of a string set.

After a malicious substring is detected by one of the SDs, a further block of ML-CCBFs is set in the so-called pattern matching engines (PMEs) in order to determine if such an alert actually corresponds to a real attack. More precisely, a PME sets a filter for each string to which the detected substring belongs. Such a CBF represents all the remaining characters: Whenever a filter is completely reset to zero, it is assumed that the string has been detected and the packets must be dropped in order to nullify the attack.

However, not all the attacks can be detected in this way. For instance, a string split in several very small packets (less than 3 bytes) is not revealed. Fortunately, packets of 1–2 bytes are very rare in real traffic,[1] except for certain applications such as telnet and ssh, and therefore we can use their presence as an alert and divert all the small packets to a slow path engine. In order to face denial-of-service attacks, we can select a threshold on the maximum number of flows to be diverted or use the queue system proposed in [28].

The system can be improved, in terms of both functioning and performance, by adopting a series of refinements. For instance, deleting the most frequent substrings from the filters allows to

[1]As shown, for example, by data at http://netflow.internet2.edu/.

save memory (smaller filters) and processing load (fewer substrings that generate an alert). The potential drawbacks are a lower detection capability (since fewer substrings signal an attack) and a bigger number of false positives (since filters can be more easily reset to zero).

### C. Experimental Results

For the experimental runs, a cluster of PCs that generate traffic toward a LAN is used. One of them runs FTester, which is a software tool capable of generating evasion attacks, while the other ones generate background traffic. The lengths of substrings in our runs are alternated in order to have both "normal" evasion attacks (with substrings of almost 3 bytes) and attacks with "small packets" (less than 3 bytes). A general-purpose PC running our anti-evasion system is placed before the LAN to protect it.

In Table III, for traces of different sizes, the number of attacks that are generated, the percentage of real attacks we detect, and the percentage of false positives are reported, respectively, for "normal" and "small" attacks and their sum. These results exhibit high percentages of detection, while the number of false positives remains small. As foreseen, the technique used against the attacks performed with "small packets" generates the largest number of false alerts since each small packet is signaled as a potential attack.

We want once more to remark that this example is proposed as a reasonable use case for ML-CCBF. Therefore, the purpose is not to present a thorough evaluation of the proposed IPS architecture. For this reason, only a few macroscopic performance parameters are presented, while some others (like throughput) are not investigated, as they would depend on too many factors and implementing choices that are outside the scope of the paper.

## VIII. Conclusion

The target of this paper is to propose an efficient structure for data representation in systems with limited memory such as programmable routers and network processors.

A tighter upper bound for counter overflow probability in counting Bloom Filters has been presented: Its specific expression and the properties that come from it allow to introduce a novel approach to CBF design. On the basis of such a theoretical achievement, the paper presents the following.

- HCBF, a first attempt to take advantage of Huffman code, which is optimal for independent symbols, such as the bins in CBFs. This approach represents an intermediate step to the central proposal of the paper.
- ML-CCBF, which combines the advantages of HCBF with the idea of a multilayer structure for CBF (in order to exploit the memory hierarchy of many systems).

A comparison among such data structures and several solutions defined in literature has been performed by using Intel IXP2800 as referential platform. The outcomes show clear memory savings of our solutions in comparison to standard CBFs (up to 50%) and a great reduction of the lookup time in ML-CCBF, even with respect to the more advanced algorithms.

Finally, we have shown a possible application of our structures in network security. In particular, we adopt ML-CCBF in a scheme that allows to detect evasion attacks in the fast data path with no need of reassembling the flows.

## References

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

[2] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Math.* vol. 1, no. 4, 2005 [Online]. Available: http://www.internetmathematics.org/volumes/1/4/Broder.pdf

[3] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 254–265, 1998.

[4] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Multilayer compressed counting Bloom filters," in *Proc. 27th IEEE INFOCOM*, 2008, pp. 311–315.

[5] G. Antichi, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Counting Bloom filters for pattern matching and anti-evasion at the wire speed," *IEEE Netw.*, vol. 23, no. 1, pp. 30–35, Jan.–Feb. 2009.

[6] M. Mitzenmacher, "Compressed bloom filters," in *Proc. 20th ACM PODC*, New York, 2001, pp. 144–150.

[7] A. Kirsch and M. Mitzenmacher, "Distance-sensitive Bloom filters," in *Proc. ALENEX*, 2006, pp. 41–50.

[8] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network applications of dynamic Bloom filters," in *Proc. 25th IEEE INFOCOM*, 2006, vol. 1.

[9] A. Kumar, J. J. Xu, L. Li, and J. Wang, "Space-code Bloom filter for efficient traffic flow measurement," in *Proc. ACM IMC*, New York, 2003, pp. 167–172.

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

[11] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey, "Dynamic count filters," *SIGMOD Rec.*, vol. 35, no. 1, pp. 26–32, 2006.

[12] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting Bloom filters," in *Proc. 4th Ann. Eur. Symp. Algor.*, 2006, vol. LNCS 4168, pp. 684–695.

[13] N. Hua, H. Zhao, B. Lin, and J. Xu, "Rank-indexed hashing: A compact construction of Bloom filters and variants," in *Proc. IEEE ICNP*, 2008, pp. 73–82.

[14] 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*, New York, 2005, pp. 181–192.

[15] E. J. Johnson and A. R. Kunze*, IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Santa Clara, CA: Intel Press, 2003.

[16] *"Intel IXP2800 Hardware Reference Manual,"* Intel, 2004.

[17] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. ACM SIGCOMM*, pp. 339–350.

[18] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. ANCS*, 2007, pp. 145–154.

[19] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *IEEE Symp. Security Privacy*, May 2008, pp. 187–201.

[20] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, A. DiPietro, and G. Antichi, "An improved DFA for fast regular expression matching," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 29–40, 2008.

[21] C. J. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of snort," *discex*, vol. 01, p. 0367, 2001.

[22] 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.–Feb. 2004.

[23] M. Nourani and P. Katta, "Bloom filter accelerator for string matching," in *Proc. 16th ICCCN*, 2007, pp. 185–190.

[24] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," Secure Networks, Inc., Calgary, AB, Canada, T2R-0Y6, Tech. Rep., 1998.

[25] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," in *Proc. 10th. USENIX SSYM*, Berkeley, CA, 2001, p. 9.

[26] N. S. Artan and H. J. Chao, "Multi-packet signature detection using prefix Bloom filters," in *Proc. IEEE GLOBECOM*, 2005, vol. 3, pp. 1811–1816.

[27] G. Varghese, J. A. Fingerhut, and F. Bonomi, "Detecting evasion attacks at high speeds without reassembly," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 327–338, 2006.

[28] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. ACM ANCS*, , pp. 155–164.

**Domenico Ficara** (M'10) received the Ph.D. degree in information engineering from the Department of Information Engineering, University of Pisa, Pisa, Italy.

During his Ph.D. studies, he collaborated with Cisco Systems, San Jose, CA, on deep packet inspection research and development projects. His main research interests are deep packet inspection and network topology discovery techniques.

**Andrea Di Pietro** (S'10) received the Master's degree in telecommunication engineering from the University of Pisa, Pisa, Italy, in April 2007. He is currently pursuing the Ph.D. degree with the NetGroup of the University of Pisa.

From May 2007 to December 2008, he was a Research Assistant with the NetGroup of the University of Pisa. His research interests are in network tomography and network performance measurement.

**Stefano Giordano** (SM'10) received the Master's degree in electronics engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in 1990 and 1994, respectively.

He is an Associate Professor with the Department of Information Engineering, University of Pisa, where he is responsible for the telecommunication networks laboratories. His research interests are telecommunication networks analysis and design, simulation of communication networks and multimedia communications.

Dr. Giordano is Secretary of the Communication Systems Integration and Modeling (CSIM) Technical Committee. He is Associate Editor of the *International Journal on Communication Systems* and of the *Journal of Communication Software and Systems* technically cosponsored by the IEEE Communication Society. He is member of the Editorial Board of the *IEEE Communication Surveys and Tutorials*. He is one of the referees of the European Union, the National Science Foundation, and the Italian MIUR and MAP Ministries.

**Gregorio Procissi** (M'10) received the graduate degree in telecommunication engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in 1997 and 2002, respectively.

From 2000 to 2001, he was a Visiting Scholar with the Computer Science Department, University of California, Los Angeles. In September 2002, he became a Researcher with Consorzio Nazionale Inter-Universitario per le Telecomunicazioni (CNIT) in the Research Unit of Pisa. Since 2005, he has been an Assistant Professor with the Department of Information Engineering, University of Pisa. His research interests are measurements and performance evaluation of IP networks.

**Fabio Vitucci** (M'09) received the Master's degree in telecommunication engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in October 2004 and June 2008, respectively.

He currently conducts research with the Department of Information Engineering, University of Pisa, in the areas of packet classification, pattern matching, and network processors.