Bloom Filters: Design Innovations and Novel Applications

Yi Lu, Balaji Prabhakar
Dept. of Electrical Engineering
Stanford University
Stanford, CA 94305
yi.lu,balaji@stanford.edu

Flavio Bonomi
Cisco Systems
175 Tasman Dr
San Jose, CA 95134
flavio@cisco.com

Abstract

Bloom filters have been very interesting in networking because they enable the high speed, low cost implementation of various hardware algorithms. The paper introduces the idea of variable-length signatures, as opposed to the current practice of using fixed-length signatures. This idea naturally enables Bloom filters to perform flow deletions, a well-known problem with standard Bloom filters. Other uses of this idea are also presented and explored. A second contribution of the paper is the use of a bank of Bloom filters to identify the action that must be applied to the packets of a flow, or to dynamically record the state a flow is in. Our work shows that this approach is a promising alternative to expensive CAM or hash table lookups, and suggests a method of building cheap "fuzzy" flow memories.

1 Introduction

Bloom filters have recently become popular within the networking community because they are suited for high-speed implementations and because they enable novel algorithmic solutions to key networking problems, such as packet forwarding, measurements and security. The primary use of a standard Bloom filter is for determining set membership: does an element x belong to a given set S? Its probabilistic nature makes it produce false positives; that is, it may declare that x belongs to S even when this is not true. However, by sizing the Bloom filter appropriately, the false positive probability can be made small enough for many applications. These applications include web caching [8], address lookup [3], network measurement [11, 12, 6], intrusion and anomaly detection [7] and deep packet inspection [4]. The survey paper [1] contains other recent work and useful references. In addition to their simplicity, this rather wide applicability have made Bloom filters very interesting objects of study.

Two exciting fronts are open in Bloom filter research: (i) innovations to the basic design, and (ii) creating novel data structures using Bloom filters. This paper contributes to both categories. We shall state our contributions after formally describing a standard Bloom filter.

1.1 A Standard Bloom Filter

Let U denote the universe of finite binary numbers and let $S = \{x_1, x_2, \ldots, x_n\}$ be a subset of U. We shall say that an element $x \in U$ is "valid" if $x \in S$; else, we shall say that x is "invalid."

A Standard Bloom Filter (denoted SBF) is an m-bit vector, B. Available to us are k (random hash) functions $h_1(\cdot), ..., h_k(\cdot)$ each of which maps an $x \in U$ to a randomly chosen element of the set $\{e_1, ..., e_m\}$, where e_i is an m-bit vector with only its i^{th} bit set to 1. Let $h(x)$ be the logical

OR of $h_1(x), ..., h_k(x)$. We refer to $h(x)$ as the "signature" of x.[1] Notation: For two binary words a and b of equal length $a \ll b$ denotes that b has a 1 in each location where a has a 1.

The use of an SBF involves the following two operations.

Training. Given S and $h(\cdot)$, the vector B is set equal to the logical OR of $h(x_1), ..., h(x_n)$. Equally, the bits corresponding to the signatures of elements in S are set to 1 in the bitmap vector B.

Querying. To determine whether a y in U belongs to S, compute $h(y)$. If $h(y) \ll B$, declare $y \in S$, else declare $y \notin S$. Clearly, the declaration $y \notin S$ can never be false; however, the declaration that $y \in S$ can be false sometimes.

1.2 Our Contributions

VBF: A Bloom filter with Variable-length Signatures. Consider an SBF and a set S of elements. As before, there are k hash functions used for computing $h(x)$, $x \in S$. The key difference between a VBF and an SBF is in the training and querying phases.

Training. During training we shall set only t $(\leq k)$ bits of $h(x)$ to 1 in the bitmap B; thus, we shall allow for the setting of a "partial signature."

Querying. We shall declare that $x \in S$ if *at least* q $(\leq k)$ bits of $h(x)$ are set to 1 in B.

These modifications afford great flexibility and lead to interesting new uses of Bloom filters. We mention three.

First, it allows us to test membership when S is time-varying; that is, when elements are continually inserted into and deleted from S. This requires updating the Bloom filter so that, at any time, it contains the signatures of the currently valid elements. Updating at insertion is straightforward: simply add the new element's signature to B. Deleting an element requires that its signature bits be set to 0 in B. This could partially erase the signatures of other valid elements, rendering them to be declared not in S; that is, a false negative. Variable-length signatures present an elegant and natural solution to the deletion problem.

The next two applications of the VBF take advantage of differing flow sizes; for example, flow sizes in the Internet are heavy-tailed (Pareto distributed). The essential idea here is to increase or decrease the signature lengths of long flows so that they become more or less easy to identify in the VBF. This feature can be used in the following two ways.

We can adaptively reduce false positives. Suppose an element $y \notin S$ has enough of its signature bits set in B so as to be falsely declared as valid. If y is a long flow and if it known is that it is a false positive (through means described later), then by unsetting just a few (but not all) of its signature bits, y can be "removed" from the Bloom filter. In an SBF such an operation would cause other valid flows to be removed (become false negative) as they will lose some of their signature bits. However, a VBF recognizes partial signatures and this helps reduce the occurrence of false negatives. We explore this in Section 2.

Next, we can gauge flow lengths from signature lengths. Let the first packet of a flow, F, set t bits in B. Suppose each further packet of F sets an additional bit with some probability p so long as F is deemed to be valid. Then, the signature length at any time is a good indicator of the number of packets sent by F until that time. This idea will be explored in other work.

Approximate Action Classification. Whereas a Bloom filter can answer the question "does $x \in S$?", it cannot identify which element of S is x. However, using a bank of $\lceil \log_2 |S| \rceil$ filters, one might determine which element of S is x.[2] A generalization of this question is this: The

[1] Observe that $h(x)$ is an m-bit vector with at most k bits set to 1.

[2] We do not elaborate on the method here, but rather mention the well-known puzzle related to this question. There are n people, exactly one of whom has a virus which can be detected by a blood test. What is the minimum number

set S of flows is partitioned into disjoint subsets S_i. Which subset does x belong to? This latter problem, the action classification problem, is very common in practice.

Each element of S is associated with an *action*; for example, when S is a flow table in a router or a switch, actions can be a combination of tasks like: admit its packets, forward them to egress port 13, encrypt before transmitting, etc. Arriving packets must be assigned an action and this is done by consulting a flow table. The current practice is to maintain a flow table as a hash table or to store it in a CAM (Content Addressable Memory). However, due to large flow table sizes, high operating speeds and stringent heat dissipation constraints, hash tables and CAMs are becoming very expensive and difficult to implement at high speeds.[3]

A number of recent papers propose the use of Bloom filters to build more efficient hash tables; notably [14] and [10]. The idea is to reduce the number of lookups and space requirement of hash tables dramatically by using a Bloom filter-based summary in conjunction with a hash table.

Our approach is different. We use Bloom filters for minimizing (ultimately, avoiding) accesses to hash tables or CAMs; that is, to directly determine the action that must be applied to a packet without identifying the flow to which it belongs. Our goal is to replace the hash table or the CAM by smaller, cheaper, on-chip memories. In this sense our work is closer in spirit to [2], which proposes using a bank of Bloom filters for *approximate* action classification, essentially trading accuracy for cost. As will be clear in Section 3, our work is a significant advance over the naive scheme proposed in [2] and suggests a method of building "fuzzy" flow memories using a bank of Bloom filters.

2 Variable-length Signatures

Bloom filters using variable-length signatures (VBFs) have the option of setting and looking up a "partial signature," and this feature enables novel uses of the Bloom filter. In this section, we use VBFs for tracking a time-varying set[4] of flows and for adaptively reducing false positives.

In what follows we shall say that an item's signature has been "deleted" if it is removed from the Bloom filter when instructed (by some central processor) to do so. We shall say that the signature has been "aged" if it has been removed by some lazy background process. For example, aging can be achieved by unsetting the Bloom filter bits in a round robin fashion.

Current solutions for tracking a time-varying set include the counting Bloom filter [3, 8] and double buffering [2]. A counting Bloom filter deletes an element by decrementing the counters corresponding to its signature bits. False negatives occur when a counter overflows. In practice, an acceptable false negative rate requires 3- or 4-bit counters, and the resultant increase in space makes the use of counting Bloom filters a concern. Double buffering uses two bit-maps, only one of which is active at any time. When the active bit-map is half-full, new signatures are placed in both the active and inactive bit-maps, although only the active bit-map is queried. When the inactive bit-map gets half-full, it becomes active and the previously-active bit-map becomes inactive and is cleared. This alternating cycle *ages* signatures and there is no option for continually retaining a signature over two cycles without corresponding arrivals. As a result, the false negative

---

of blood tests needed to identify the infected person? Answer: $\lceil \log_2 n \rceil$, obtained by mixing bloods.

[3] Hash tables, usually stored in SRAMs or DRAMs, require a lot more space (buckets) than the number of flows in order to reduce collisions. Moreover, they may require multiple memory accesses (usually a random number, depending on the flow), requiring a high memory bandwidth. While CAMs have the same size as the flow table and need only a single memory access, they dissipate a lot of power, require more space per item and have a higher access time when compared to RAMs (Random Access Memories).

[4] By a "time-varying set" we mean a set whose membership changes with time; for example, a table holding currently active flows. Since elements are inserted into and deleted from the set over time, their signatures must concurrently be inserted into and deleted from the corresponding Bloom filter.

rate is considerably higher in the double-buffering scheme than in the counting Bloom filter; we shall see some comparisons below.

We propose a variable-length signature enhancement that trades an increase in false positives for a much larger reduction in false negatives. We will see that variable-length signatures and counters complement each other nicely to provide a better solution for tracking time-varying sets than just using counters.

The rest of the paper: There are two sections for dealing with each of the ideas mentioned above. Due to a shortage of space, we have chosen to present the basic algorithmic ideas and their enhancements, deferring the theoretical analysis and extensive simulations for a longer publication.

2.1 The VBF

We shall first describe the operation of a VBF in general terms, specifying several options. Consider a bit-map of size m and k hash functions $h_1(x), ..., h_k(x)$ constituting the signature $h(x)$ as in an SBF.

Train/insert: When a flow x is to be inserted, set $t \leq k$ bits of its signature $h(x)$ to 1.

Query/lookup: A flow x is declared to be valid if at least q $(q \leq t \leq k)$ bits in $h(x)$ are set to 1 in the bit-map B. (Clearly, such a declaration results in a false positive if x is not valid at the time of the query.)

Delete: When flow x is to be deleted, at least $k - d$ of its signature bits are set to zero. Of course, we shall insist that $d < q$ so that a deleted flow is no longer declared as valid.

Recover/increment: Any number of the missing bits of a positive flow (true or false) may be recovered by setting them to 1; thus, recovery strengthens or lengthens signatures. Recovery can be triggered by packet arrivals and may be preceded by a coin toss whose outcome can decide if and by how much the signature should be increased.

Age/scrub: This operation sets a 1-bit of the bit-map B to 0. The 1-bit can either be obtained from a round-robin pointer cycling through the bit-map or be chosen uniformly at random from amongst all the bits.

Counting VBFs

As in a counting SBF, each location on the bit-map B of a counting VBF contains a c-bit counter. The operations are exactly as described for a VBF, except that setting (unsetting) a bit corresponds to incrementing (decrementing) the associated counter. A counter which overflows is set at saturation.

When there is a choice of counters to increment or decrement, the implementor may choose to do so according to some suitable policy; e.g., random, longest counter, shortest counter, etc.

Remarks: Note that we allow recovery only for valid flows; i.e., flows which have at least q of their signature bits present. The difference between aging and deletion is that aging affects all flows equally, whereas deletion can be targeted at a specific flow.

2.2 Applications of the VBF

1. Flow deletion: It is immediately obvious that a VBF permits deletion of flows, while deletion is not defined for an SBF. Further, the recover operation allows flows to regain the signature bits they lost accidentally when other flows were being deleted. Similarly, the combination of the age and recover operations with the use of variable-length signatures provides an elegant solution for tracking time-varying sets.

Having said this, it is still possible for a valid flow to lose enough of its signature due to the deletion of other flows so as to become a false negative. This is studied in the next subsection.

There is an important point to mention in connection with the deletion operation. When a flow is being deleted, we set several (at least $k - d$) of its bits to 0. Due to the random nature of signatures, the bits of a flow which are set to 0 are *most likely* to belong to *different* valid flows. [5] Thus, deletion affects valid flows in a minimal fashion.

2. False positive/negative rate vs probability: The false *probability* (positive or negative) is usually defined on a per *flow* basis, whereas the corresponding false *rate* is defined on a per *packet* basis. Thus, the false positive probability equals the probability that a flow not in S is falsely declared as valid. The false positive rate equals the *fraction of packets* which are falsely declared as positive. When the flow size distribution is heavy-tailed, it is possible to achieve a very small false rate for a given false probability by making fewer errors on large flows which bring lots of packets if we could identify such flows and take advantage of this in the Bloom filter. We briefly explore this in the next subsection.

2.3 Trace-driven Comparison

Whereas a static and Markovian dynamic analysis of the VBF is possible, due to a lack of space we do not present it here. Instead, we present performance results from simulations using an Internet packet trace. This allows us to compare the counting VBF to counting SBFs and to the double buffering scheme. The focus of the simulations is on a key performance trade-off: By how much does a VBF reduce the false negative rate, and how much false positive rate does it trade in the bargain?

Trace details: The comparison is performed on a 5 million packet CAIDA trace collected at 9:10am, Aug 14, 2002. There are a total of 168640 flows. The number of concurrently active flows reaches a maximum of 46953. We, therefore take $n = 470004$ and size our Bloom filters. We designate the longest 10% of the flows (which bring 80% of the traffic) as "long," and the shortest 60% of the flows as "short," and the rest as "medium." We designate a randomly chosen 90% of flows as valid and the remaining 10% as invalid. The large proportion of flows designate as valid allows us to observe the deletion process thoroughly. The relatively small proportion of invalid flows does not affect the accuracy of the false positive rate because an invalid flow becomes a false positive independent of other flows.

Bloom filter details: We use $\frac{m}{n} = 16$ and $k = 11$ for all Bloom filters. We use $t = 11, q = 10, d = 0$ for all VBFs; and a flow which has 10 signature bits is incremented to 11 signature bits when its next packet arrives. A flow's signature is inserted into the Bloom filter when its first packet arrives, subsequent packets generate queries, and the signature is deleted when the last packet is processed.

Basic version: VBF vs. SBF. An SBF does not deal with deletion as a part of its design. Nevertheless, to compare it with a VBF, we simply remove all 11 signature bits of a flow from the SBF when it is deleted. We count all flows whose signature bits fall short of 11 at queries as false negatives.

A simple VBF with $t = 11, q = 10, d = 0$ is used to demonstrate the trade-off between false positives and false negatives. The parameters chosen here are by no means optimal; they are chosen mainly for consistency with the next section where we shall see that a counting VBF provides a much better solution to the deletion problem. The comparison is in Table 1.

[5] Essentially, this statement follows from the fact (see [5, 9]) that when two signatures overlap, they are most likely to do so at one bit.

		SBF	VBF
Space (Mbits)		0.679	0.679
FP ($\times 10^{-4}$)	Large	0	6
	Medium	0.2203	11
	Small	0	0
	Average	0.0353	6.250
FN ($\times 10^{-4}$)	Large	9076	1906
	Medium	6001	3010
	Small	4456	2242
	Average	8522	2082

Table 1: Comparison of performance on the packet trace: Basic version. FP is the false positive rate (i.e., per packet) and FN is the false negative rate.

We see that the VBF is able to reduce the overall false negative rate by approximately 4-folds. The effect is especially conspicuous with the long flows. The strikingly low false positive rate for the SBF is mainly due to its massive loss of bits in deletions; and unlike the VBF, no recovery is performed. The correct false positive rate for this trace can be seen in the counting 2-bit and 3-bit SBFs in Table 2, which loose a lot fewer bits to deletion. Comparing the false positive rate of the VBF with those numbers, we see that the increase in false positive rates is less than 3-fold.

Counting VBF vs. counting SBF. We now introduce two improvements. First, instead of a basic VBF, we use a 2-bit counting VBF with the same parameter as above. The VBF is compared with 2-bit and 3-bit counting SBFs. We leave a counter at saturation once it overflows. Note that a 3-bit counting Bloom filter uses 50% more space than a 2-bit filter.

Second, we introduce the enhanced 2-bit VBF, which adaptively reduces false positive probabilities for long flows. Long flows are identified by sampling each arriving packet with probability p. For a flow with l packets, the probability that at least one of its packet is sampled is $1-(1-p)^{l}$. Hence the longer the flow is, the more likely it will be sampled. [6]

Once a flow has been identified as long (i.e., one of its packet has been sampled), it is then looked up in an off-chip table to determine if it is a false positive. If yes, then we *attenuate* its signature down to a bits; that is, we set at least $k-a$ bits of the flow's signature to 0. In the simulations below we choose $p = 0.25$ and $a = 8$ for the enhanced 2-bit VBF.

Since 80% of the work is brought by the very few long flows, only a small number of signatures need to be attenuated. However, the improvement in the false positive probability affects 80% of the packets. This idea, mentioned above when contrasting false positive probability and rate, alone is potentially interesting for other applications involving Bloom filters where power laws need to be exploited.

The comparison results are tabulated in Table 2. The results for double buffering are included for completeness.

We see that the 2-bit VBF causes a 386-fold reduction in the false negative rate by trading a 24-fold increase in the false positive rate. The corresponding numbers for the enhanced 2-bit VBF are 384 and 4.8, with the most pronounced reduction occurring for the long flows. Hence, with two-thirds the space, the performance of the enhanced 2-bit VBF compares favorably with that of the 3-bit counting SBF.

[6] An algorithm, called SIFT, which is based on this sampling idea is studied in [13], where it is used to reduce flow delays in the Internet.

		2-bit SBF	3-bit SBF	2-bit VBF	Enhanced 2-bit VBF	Double buffering
Space (Mbits)		1.353	2.030	1.353	1.353	1.353
FP ($\times 10^{-4}$)	Large	1.849	1.849	61	7	0
	Medium	4.846	4.846	34	22	8.59
	Small	0	0	50	50	0
	Average	2.295	2.295	56	11	1.377
FN ($\times 10^{-4}$)	Large	323	0	0.733	0.733	1435
	Medium	157	0	0.691	0.715	3265
	Small	78	0	2.596	2.596	2124
	Average	293	0	0.759	0.763	1729

Table 2: Comparison of performance on the packet trace: Improved version. FP is the false positive rate (i.e., per packet) and FN is the false negative rate.

3 Approximate Action Classification

In this section we show how a bank of SBFs (or VBFs) can be used to classify arriving packets according to the action that must be applied to them. Another use of such a bank is to record flow data (such as the state it is in). Both problems use some kind of a hash table or a CAM. Our goal is to use a bank of SBFs to reduce the total space required and to use cheaper memories. The main drawback of our approach is errors: false positives and failures, and we describe one way of coping with them. However, if successful, our approach would obviate the need for a costly hash table or CAM lookup. A comparison of costs shows that our solution is cost-effective and encourages further work.

Now consider the problem. There is a set, A, of actions exactly one of which must be applied to each flow in a set S. Partitioning S according to the action corresponding to each flow, we obtain subset-action pairs $\{S_i, A_i\}$.

One approach for performing lookups is to have an SBF for each A_i and to load the signatures of flows in S_i into it. Arriving packets are looked up in all filters and their action will be determined if only one filter returns a positive. By sizing the filters appropriately, the probability that a flow tests positive on the wrong filter can be made very small. This approach, well-known in practice, has been explicitly stated and developed in [2]. However, there are two problems with this approach.

First, when there are many actions,[7] a large number of lookups are required. Second, this approach is very sensitive to the distribution of flows per action; i.e., to the sizes of the A_i. It is quite likely that in practice the A_i will have widely-varying sizes. Recognizing this, [2] suggests a clever load-balancing idea (described later). Unfortunately, this does not suffice because the resulting fluctuation in the loading is high relative to the mean, yielding a poor performance. We propose a different arrangement which addresses both these problems. For ease of reference, let us call the scheme in [2] the *linear-lookup* scheme because it uses as many filters as there are actions.

One way to reduce the number of look-ups is by encoding the action indices. For example, consider a bank of three SBFs. It can encode a maximum of $2^3 - 1 = 7$ actions, reserving "000" code for elements not in S. Each bit in the 3-bit encoding of an action corresponds to each of the three filters. If action "101" is to be applied to flow F, then we load F's signature in the first and third filters, but not in the second. Since only $\log |A|$ lookups are performed for each query, we

[7] The number of actions can be in the order of a few hundred, and the number of states a flow might be in could be more than a thousand.

will call this the *log-lookup* scheme.

Notice that in the log-lookup scheme a flow's signature is inserted into multiple SBFs. Therefore, to obtain the same false positive probability, the total number of bits used across all the SBFs in log-lookup is much higher than in linear-lookup. This trade-off between space and the number of lookups leads us to other ways of encoding actions.

Codes: Note that the linear-lookup scheme corresponds to a mapping $M_1 : A \rightarrow \mathcal{S}_1$; i.e., a mapping of actions into the set $\mathcal{S}_1$ of binary words of length $|A|$ with *exactly one 1* in them. We consider a family of encodings $M_i : A \rightarrow \mathcal{S}_i$, where $\mathcal{S}_i$ is the set of binary words of length L_i which have *exactly i 1s* in them. The L_i is chosen such that $\binom{L_i}{i} \geq |A|$. We shall call codes corresponding to M_i as the "i-encoding scheme." In particular, the linear-lookup scheme and the 1-encoding scheme are the same.

Let us specialize to $i = 2$ for constructing the bank of SBFs corresponding to the encoding M_2. We take a number L_2 of filters in a bank, and associate with each action A_j a distinct codeword in $\mathcal{S}_2$.[8] Each codeword specifies exactly two filters in the bank; for example, if $M_2(A_j)$ has 1s in positions a_j and b_j, then filters numbered a_j and b_j are specified by A_j.

Signatures of flows corresponding to A_j are loaded into its two filters.[9] It is possible that when all signatures are loaded, that a flow might be in "error;" that is, its signature shows up in 3 or more filters. Let P_e denote the probability that a flow is in error. We make a second pass and test all flows to see if they are in error. All error flows are placed in an overflow CAM, whose size we would like to keep small. This can be done by sizing the filters so that P_e is as small as desired. Since P_e is upper bounded by the bank false positive probability (i.e., the probability that a flow registers as a false positive in any filter in the bank), we obtain that

$$\begin{aligned} P_e &\leq 1 - (1 - (1 - e^{-\frac{n'k}{m'}})^k)^{L_2} \\ &\approx L_2(1 - e^{-\frac{n'k}{m'}})^k, \end{aligned} \tag{1}$$

where n' is the number of flows and m' is the size of the bit-map *per filter*, and $k = \frac{m'}{n'} \ln 2$ is the length of a flow's signature. (Note that we have assumed a uniform distribution of flows per filter; due to load balancing, this assumption is nearly correct.)

Formula (1) can be used to size the filters and we obtain the comparison table, Table 3, below. We note that there is also the probability that a flow not in the set S of interest is falsely classified as belonging to some action A_j. Of course, in order to do this, such a flow would have to test positive in filters corresponding to valid codewords. The table below also gives the number of false positive flows.

Table 3 shows how different action encoding schemes work for 100,000 flows and 1000 actions. Each flow was assigned one of the actions uniformly at random. The false positive probability was obtained using another 100,000 flows. The following features stand out: (1) The space increases sublinearly (in fact, logarithmically, according to formula (1)) as P_e decreases; for example, in the worst case, it increases by less than a factor of 1.5 when P_e decreases from 10% to 1%. (2) The 1-encoding scheme commits a lot more errors (and hence requires a much

[8]The encoding of actions into codewords can either be obtained via a simple formula or via a table lookup; we do not get into that here.

[9]More precisely, we use the neat load balancing scheme introduced in [2]: For each flow F, we generate a uniform random hash, U_F, in $\{1, 2, ..., L_2\}$. If A_j is to be applied to F, then load F's signature into filters numbered $(a_j + U_F) \mod L_2$ and $(b_j + U_F) \mod L_2$. When a packet belonging to F tests positive in these filters, subtract U_F from their indices to deduce that A_j must be applied to F.

	Desired P_e (%)	Signature length: k	Space (Kbits)	Error flows	F.P. flows
1-encoding (1000 filters)	1	17	2396	1730	1711
	2	16	2250	3243	3205
	4	15	2104	6089	5854
	8	14	1955	11428	10701
	10	13	1906	13631	12627
2-encoding (45 filters)	1	12	3491	985	6.6
	2	11	3200	2011	23
	4	10	2907	3999	96
	8	9	2610	7939	373
	10	9	2513	9971	569
3-encoding (20 filters)	1	11	4642	1001	0.25
	2	10	4206	1969	1
	4	9	3767	3945	6.6
	8	8	3322	7769	41
	10	7	3177	9693	76

Table 3: Comparison of different encoding schemes. All results are the average of 20 simulation runs. The standard deviation is less than 5×10^{-3} of the mean.

larger CAM) than the 2- and 3-encoding schemes. Indeed, the size of the CAM needed by the 1-encoding scheme is much larger than intended by the choice of P_e; that is, the number of fbws in error, and hence the size of the CAM, should roughly equal 100,000$\times P_e$. This is clearly not the case for the 1-encoding scheme, while it is essentially true for the 2- and 3-encoding schemes. The deviation from the designed-for error is due to the high variance of the load relative to the mean load in the 1-encoding scheme. (3) Not surprisingly, the false positive probability decreases dramatically as we increase the number of 1s in the codewords, with most of the gain obtained in going from 1-encoding to 2-encoding. (4) Finally, we note that the 2-encoding scheme has just 45 fi lters in the bank, requiring a lot fewer lookups than the 1-encoding scheme and not much more than the 3-encoding scheme. We conclude that the 2-encoding scheme strikes the best trade-off between performance and cost (space and number of lookups) and compare it with the CAM.

Comparison with CAM. For $P_e = 1\%$, the 2-encoding scheme uses 35 bits/fbw and requires an auxiliary CAM of size 1000. It allows a false positive probability of 6.6×10^{-5}. By contrast, a single CAM for holding 100,000 entries will admit 0 false positives if we stored the entire address of a fbw (which is unique to it). However, IPv4 or Ethernet fbw addresses can be very large, upwards of 100 bits.[10] This comparison is hardly in favor of the CAM. Therefore, let us consider hashing fbw addresses down to a smaller size. To have the collision probability be on the order of 10^{-9} when hashing 100,000 fbws, we need to use 61 bits/fbw. With this we obtain a false positive probability (for the other 100,000 fbws) which is essentially 0. With 10 bits needed for specifying the action corresponding to a fbw, we get a total of 71 bits/fbw, almost double the size of the 2-encoding scheme.

To summarize, the upside of the 2-encoding scheme is that it requires half the total bits that a CAM needs. (Recall that a CAM is more expensive, consumes a lot more power and uses more space in silicon than a DRAM or SRAM of the same size.) The downside of the 2-encoding scheme is that it is error-prone, admits false positives and requires extra work (inserting/deleting signatures, querying, etc). However, the upside is very encouraging and warrants further investigation into the usefulness of a bank of SBFs or even counting VBFs.

[10] Simply taking a source-destination pair in IPv4 (Ethernet) as the fbw address requires 64 (96) bits/fbw.

4 Conclusion

The paper introduced two main ideas: variable-length signatures for Bloom filters (or VBFs), and the use of a bank of Bloom filters for building fuzzy flow memories. Due to a shortage of space, we were able to present only preliminary analyses and comparisons of these ideas; a more in-depth study will be presented in forthcoming publications. Our results are, nevertheless, quite encouraging and suggest that both the ideas can lead to better Bloom filters for tracking time-varying flow tables and for building flow memories. In addition, several research avenues have been mentioned as worth pursuing further and we are embarked on this program.

Acknowledgment: We thank Paul Cuff for his timely help with simulations that led to the numbers in Table 3.

References

[1] A. Z. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Accepted to Internet Mathematics*, 2005.

[2] Francis Chang, Wu-Chang Feng, and Kang Li. Approximate caches for packet classification. *In Proceeding of IEEE Infocom 2004, Hongkong*, 2004.

[3] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using bloom filters. *SIGCOMM, (Karlsruhe, Germany), Aug*, 2003.

[4] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. *HOTI'03: Hot Interconnects 11:2003, Stanford*, 8/03.

[5] P.C. Dillinger and P. Manolios. Bloom filters in probabilistic verification. *FMCAD, Formal Methods in Computer-Aided Design*, 2004.

[6] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *Proceedings of ACM SIGCOMM*, 2002.

[7] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. *Proceedings of the USENIX/ACM Internet Measurement Conference*, October 2003.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transaction on Networking*, 8(3):281–293, 2000.

[9] A. Kirsch and M. Mitzenmacher. Building a better bloom filter. *Technical Report of Computer Science Group, Harvard University*, (TR-02-05), 2005.

[10] A. Kirsch and M. Mitzenmacher. Simple summaries for hashing with multiple choices. *43rd Annual Allerton Conference on Communication, Control and Computing, Sep*, 2005.

[11] A. Kumar, M. Sung, J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 177–188, 2004.

[12] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code bloom filter for efficient per-flow traffic measurement. *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 167–172, 2003.

[13] K. Psounis, A. Ghosh, B. Prabhakar, and G. Wang. A simple algorithm for tracking elephant flows, and taking advantage of power laws. *Proceedings of Allerton Conference*, 2005.

[14] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: An aid to network processing. *SIGCOMM, (Philadelphia), Aug*, 2005.