Kocak, T., & Kaya, I. (2006). Low-power bloom filter architecture for deep packet inspection. IEEE Communications Letters, 10(3), 210 - 212. 10.1109/LCOMM.2006.03028, 10.1109/LCOMM.2006.1603387

Link to published version (if available):
10.1109/LCOMM.2006.03028
10.1109/LCOMM.2006.1603387

Link to publication record in Explore Bristol Research

PDF-document

Low-Power Bloom Filter Architecture for Deep Packet Inspection

Taskin Kocak and Ilhan Kaya

Abstract—**Bloom filters are frequently used to identify malicious content like viruses in high speed networks. However, architectures proposed to implement Bloom filters are not power efficient. In this letter, we propose a new Bloom filter architecture that exploits the well-known pipelining technique. Through power analysis we show that pipelining can reduce the power consumption of Bloom filters up to 90%, which leads to the energy-efficient implementation of intrusion detection systems.**

Index Terms—**Bloom filters, low-power design, network intrusion detection.**

I. Introduction

NETWORK intrusion detection systems (NIDS) are used to identify the malicious content such as internet worms and viruses in network packets. They typically employ a deep packet inspection functionality, which checks the payload of the packet against a set of known virus and worm signatures. Bloom filters [1] are used in such filtering applications to match strings such as Snort rules [7]. Although Bloom filters have found wide spread usage in networking applications [2], they are not conservative in terms of power. An NIDS consists of 4 Bloom filter engines can dissipate up to 5 W. In order to reduce the power consumption of Bloom filters, we propose to employ a pipelining technique in the architecture of Bloom filters. We call this new type of Bloom filters as *pipelined Bloom filters*. In this letter, first, we propose a hardware system consists of pipelined Bloom filters as an energy-efficient NIDS. Then, we provide a mathematical analysis to show that the proposed system is more energy-efficient than the regular Bloom filter-based architectures used so far.

II. Bloom Filters

A Bloom filter is a data structure that stores a given set of signatures, by first computing multiple hash functions on each of the members of the set, and then it queries the database for a given input string, by again computing many hash functions of the input with a lookup operation followed over the database. First operation is called *programming* of the Bloom filter, and the second operation is *querying*. A typical Bloom filter is illustrated in Fig. 1.

In the programming stage, given a string *X*, which is a member of the signature set, a Bloom filter computes *k* many hash values on the input *X* by using *k* different hash functions. Then it uses these hash values as index to the *m-bit* long lookup vector. It sets the bits corresponding to the index given by the hash values computed. It repeats this procedure for each member of the signature set.

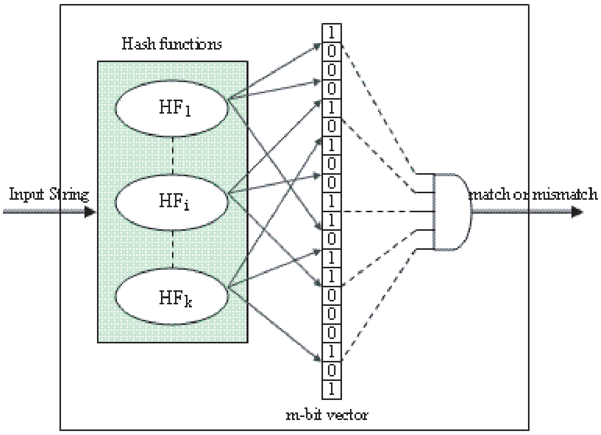

Fig. 1. A regular Bloom filter with k hash functions.

In the query stage, for an input string *Y*, Bloom filter computes *k* many hash values by utilizing the same hash functions used in programming of the Bloom filter. Bloom filter looks up the bit values located on the offsets (computed hash values) on the bit vector, and,

* If it finds any bit unset at those addresses, it declares the input string to be a nonmember of the signature set, which is called a *mismatch*.

* Otherwise, it finds all the bits are set, it concludes that input string may be a member of the signature set with a certain probability (*false positive probability*), which is called a *match*.

A Bloom filter never produces *false negatives*, which means if it decides that an input is a nonmember, input certainly does not belong to the signature set. However, it may produce false positives.

Following the analysis of [4], the false positive probability f is calculated by,

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

where n is the number of signatures programmed into the Bloom filter, m is the length of the lookup vector, and k is the number of hash functions used to implement the Bloom filter. In order to minimize the false positive probability, the value of m must be quite larger than n. For a fixed value of $\frac{m}{n}$, k must be large enough such that f gets minimized. The number of hash functions that minimizes the false positive probability

Manuscript received September 29, 2005. The associate editor coordinating the review of this letter and approving it for publication was Prof. Iakovos Venieris.

The authors are with the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816 USA (e-mail: {tkocak, ikaya}@cs.ucf.edu).

Digital Object Identifier 10.1109/LCOMM.2006.03028.

is given by

$$k = \left(\frac{m}{n}\right)\ln 2 \tag{2}$$

If we take the average number of bits allocated to a single signature, $\frac{m}{n} = 50$, the number of hash functions per Bloom filter is calculated as

$$k = ({}^{m}\!/_{n})\ln 2, \quad k = (50)\ln 2, \quad k = 35 \tag{3}$$

Substituting $\frac{m}{n} = 50$, and $k = 35$ into the false positive probability equation, Equ. 1, yields a small probability with which Bloom filter produces false positives.

$$f = 0.5^{35} < 3.10^{-11} \tag{4}$$

Hash functions used in the Bloom filters are generally of type *universal hash functions* [3]. The performance of universal hash functions are explored by Ramakrishna et al [6]. The hash function, being a member of a universal hash function class, maps the input string to an output string, such that collision probability of given any two input strings is small. Given any string *X*, consisting of *b* bits,

$$X =< x_1, x_2, x_3 ... x_b > \tag{5}$$

i^{th} hash function over the string *X* is defined as

$$h_i(x) = r_{i1} \bullet x_1 \oplus r_{i2} \bullet x_2 \oplus r_{i3} \bullet x_3 \oplus ... r_{ib} \bullet x_b \tag{6}$$

where r_{ij}'s are random coefficients ranging from *1* to *m*, and x_i's are the bits in the input string.

A single Bloom filter uses *k* many hash functions in order to make a decision on the input. Hence the power consumption of a Bloom filter shown in Fig. 1 is a summation of the power consumptions of each of the hash functions, P_{H_i}, with the lookup operation, P_L, followed, plus an AND operation:

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_i} + P_L) + P_{AND} \tag{7}$$

Power consumption of the AND gate is ignored hereafter, since it is minimal compared to the power used by the hash functions. We also assume that the lookup power over a m-bit vector is approximately constant for each index calculated by any of the hash functions. Since hash functions with the same number of input bits will be implemented with the same number of components and will consume approximately the same amount of power. We can write the power consumption of a regular Bloom filter as follows

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_i} + P_L) = k.(P_H + P_L) \tag{8}$$

III. Pipelined Bloom Filters

Since the number of hash functions required to minimize the false positive probability of a Bloom filter is large, it is better, in terms of power, to implement these hash functions in a pipelined manner. We call this new type of Bloom filters *pipelined Bloom filters*.

Basically, a pipelined Bloom filter, as shown in Fig. 2, consists of two groups of hash functions. The first stage always computes the hash values. By contrast, the second stage of hash functions only compute the hash values if in the first stage there is a match between the input and the signature sought.

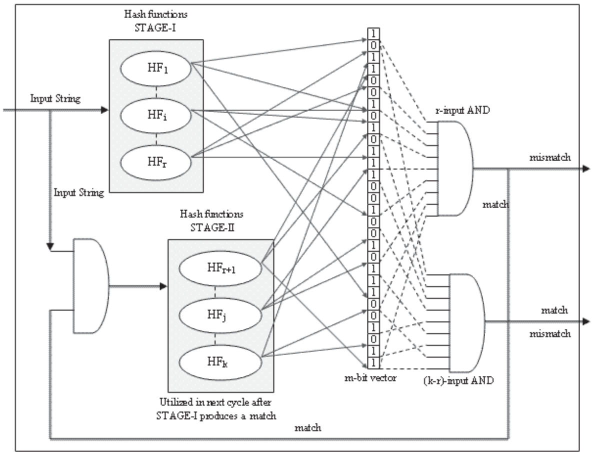

Fig. 2. A 2-stage pipelined Bloom filter.

The pipelined Bloom filters will have the same number of hash functions as the regular Bloom filters. Hence the false positive probability is the same. A pipelined Bloom filter exploits the virus free nature of the network traffic in most of the time. At worst, it will operate like a regular Bloom filter, which uses all of the hash functions before making a decision on the type of the input. The advantage of using a pipelined Bloom filter is if the first stage produces a mismatch, there is no need to use the second stage in order to decide whether the input string is a member of the signature set. This is simply because a Bloom filter never produces a false negative.

Let us first derive the probability of match in the first stage. By following a similar analysis of [5], we assume that the hash functions used in each Bloom filter are perfectly random. This is a reasonable assumption since each hash function coefficients are selected randomly in range 1 to *m*. In the first stage, *r*-many of the hash functions are utilized. The probability that a bit is still unset after all the signatures are programmed into the pipelined Bloom filter by using *k*-many independent hash functions is α.

$$\alpha = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} (for\, large\, m) \tag{9}$$

where $\frac{1}{m}$ represents any one of the *m* bits set by a single hash function operating on a single signature. Then $\left(1 - \frac{1}{m}\right)$ is the probability that the bit is unset after a single hash value computation with a single signature. For it to remain unset, it should not be set by any of the *k*-many hash functions each operating on all of the *n*-many signatures in the signature set. Consequently, the probability that any one of the bits is set is

$$(1 - \alpha) \approx 1 - e^{\frac{-kn}{m}} \tag{10}$$

In order for the first stage to produce a match, the bits indexed by all *r* of the independent random hash functions should be set. So the match probability of the first stage is, represented as *p*,

$$p = \prod_{i=1}^{r} (1 - \alpha) = (1 - \alpha)^r \approx (1 - e^{\frac{-kn}{m}})^r \tag{11}$$

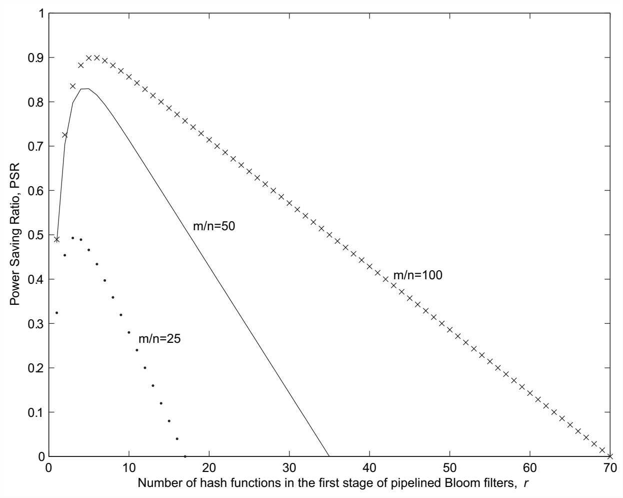

Fig. 3. Power saving ratio in pipelined Bloom filters w.r.t. number of hash functions utilized in the first stage.

The mismatch probability of the first stage is simply *1-p*,

$$1-(1-e^{\frac{-kn}{m}})^r \tag{12}$$

With a probability of *(1-p)* the first stage of the hash functions in the pipelined Bloom filter will produce a mismatch. Otherwise, the first stage produces a match, then the second stage is used to compare the input with the signature sought. Therefore the power consumption of a pipelined Bloom filter is given by

$$\begin{aligned} P_{BFpipeline} &= P_{1st-stage} + P\{match\} \times P_{2nd-stage} \\ P_{BFpipeline} &= \sum_{i=1}^{r}(P_{H_i}+P_L) + p \times \sum_{j=r+1}^{k}(P_{H_j}+P_L) \\ &\quad +P_{AND} \end{aligned} \tag{13}$$

Again, P_{AND} can be neglected, since it is very small with respect to the power consumption of hash functions. The power consumption of a pipelined Bloom filter simplifies to

$$\begin{aligned} P_{BF_{pipeline}} &= \sum_{i=1}^{r}(P_{H_i}+P_L) \\ &\quad +(1-e^{\frac{-kn}{m}})^r \times \sum_{j=r+1}^{k}(P_{H_j}+P_L) \\ &= r.(P_H+P_L)+ \\ &\quad (1-e^{\frac{-kn}{m}})^r(k-r)(P_H+P_L) \end{aligned} \tag{14}$$

The power saving ratio, *PSR*, in a single Bloom filter by deploying pipelining technique can be calculated as

$$PSR = \frac{(P_{regular}-P_{pipelined})}{\mathrm{P}_{regular}} \tag{15}$$

By substituting Equ. 8 and Equ. 14 into Equ. 15, the average power saving ratio, *PSR*, is given by

$$PSR = \frac{\Big(k \times A - \Big[r + (1-e^{\frac{-kn}{m}})^r \times (k-r)\Big] \times A\Big)}{k \times A} \tag{16}$$

where $A = (P_H + P_L)$, which is the power consumption of a single hash function with a single lookup operation. After simplifying *A*s, average power saving ratio, *PSR*, is found out to be

$$PSR = \frac{k-r+(r-k)\,(1-e^{\frac{-kn}{m}})^r}{k} \tag{17}$$

For different values of the number of bits allocated to per signature, $\frac{m}{n}$, power savings over the number of hash functions utilized in the first stage are illustrated in Fig. 3.

The amount of power conserved in the system increases as $\frac{m}{n}$ increases. This is because the number of hash functions deployed in the first stage becomes a smaller portion of the overall hash functions deployed in each configuration. The increase in the PSR value at first stems from the fact that increasing the number of hash functions in the first stage increases the probability of mismatch, thus the second stage is not utilized. After the optimum value, PSR decreases steadily. This is again because, the more hash functions are deployed in the first stage, the more power that they consume. If we increase the number of hash functions used in the first stage to such a degree that all hash functions in the system deployed in the first stage, there remains no power gain at all (i.e., the system behaves just like a regular Bloom filter.)

IV. Conclusion

In this paper, we exploited the fact that the most of the current network traffic is not malicious and proposed to pipeline the hash functions in the Bloom filters that are used in network intrusion detection systems. Analytical results show that the pipelining technique significantly decreases the total power consumption of a Bloom filter. It is shown that the lesser the number of hash functions implemented in the fist stage of a pipelined Bloom filter, the more the power saving is. The number of bits allocated to per signature, $\frac{m}{n}$, affects the power saving ratio in a pipelined Bloom filter. Analysis performed for feasible values of $\frac{m}{n}$ revealed up to 90% power savings. The selection of the hash functions to be deployed in the first stage of a pipelined Bloom filter is not considered in the current analysis. Our future work will include this as well as experimental evaluation of these novel pipelined Bloom filter architectures.

References

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

[2] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: a survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485-509, July 2003.

[3] J. L. Carter and M. Wegman, "Universal classes of hash functions," *J. Computer and System Sciences*, vol. 18, no. 2, pp. 143-154, Apr. 1979.

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

[5] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Trans. Networking*, vol. 10, no. 5, pp. 604-612, Oct. 2002.

[6] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Trans. Computers*, vol. 46, no. 12, pp. 1378-1381, Dec. 1997.

[7] The Sourcefire Vulnerability Research Team, "Official Snort Ruleset," Sourcefire, Inc., Columbia, MD, Aug. 2005 (web: http://www.snort.org/pub-bin/downloads.cgi).