Kaya, I., & Kocak, T. (2006). A low power lookup technique for multi-hashing network applications. In IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures 2006, Karlsruhe, Germany. (pp. 179 - 184). Institute of Electrical and Electronics Engineers (IEEE). 10.1109/ISVLSI.2006.3

A Low Power Lookup Technique for Multi-Hashing Network Applications

Ilhan Kaya and Taskin Kocak
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816, U.S.A
{ikaya, tkocak}@cs.ucf.edu

Abstract

Many network security applications require large virus signature sets to be maintained, retrieved, and compared against the network streams. Software applications frequently fail to identify so many signatures through comparisons at very high network speeds. Bloom filters are one of the main multi-hashing schemes utilized in hardware to support this level of security. Nevertheless Bloom filters consume significant power to store, retrieve and lookup virus signatures owing to many hash function computations required to index to the memory. We present a novel lookup technique and architecture to decrease the power consumption of multi-hashing schemes, predominantly Bloom filters, in hardware. The theoretical analysis has shown that power gain achieved through new lookup technique can go up to 90%. Simulation results with three different classes of the hash functions embedded into the Bloom filter have indicated that power consumption of the Bloom filters can be considerably decreased by employing the low power lookup technique.

1 Introduction

Many network security applications make use of multi-hashing schemes. Either they require functionalities offered by hash tables or hash functions. Not only the software applications but also some hardware systems depend upon the properties of a high performing multi-hashing scheme. Such a multi-hashing scheme generally appears in the form of a Bloom filter [3]. Bloom filters have been used for many network applications like resource routing [6], string matching [1, 7], and packet filtering [2]. They are also used to improve lookup operations in hash tables [8]. A hardware system, consisting of Bloom filters to detect malignant content, is described in [7]. A detailed survey of Bloom filters for networking applications can be found in [4].

Although Bloom filters have found wide spread usage in networking applications, they are not conservative in terms of power. A network intrusion detection system (NIDS) consists of 4 Bloom filter engines can dissipate up to 5 W.To decrease the power consumption of Bloom filters, we propose a new lookup technique which basically makes use of less number of hash function computations to determine the maliciousness of the network stream. The architecture to implement this new lookup technique in Bloom filters is presented in this paper. Furthermore, a comparative power analysis of the Bloom filter architecture which realizes the new lookup operation is given. A mathematical analysis carried out in this paper clearly states the efficiency of the new lookup technique in terms of power.

In spite of the importance of the hashing techniques in Bloom filters, hashing analysis of the Bloom filters is somewhat overlooked sofar. This paper also presents hardware architectures for implementation of the different classes of the hash functions utilized in programming and lookup operations of Bloom filters. The simulation results with the different hashing functions in varying configurations of the Bloom filters is also discussed.

2 Low power lookup technique

Before describing the low power lookup technique for multi-hashing schemes, it is important to consider what a multi-hashing scheme stands for. In this paper, we use a Bloom filter as a multi-hashing scheme for network applications. 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. The first operation is called *programming* of the Bloom filter, and the second operation is *lookup*. A block diagram of a typical Bloom filter is illustrated in Fig. 1. 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

Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI'06)
0-7695-2533-4/06 $20.00 © 2006 IEEE

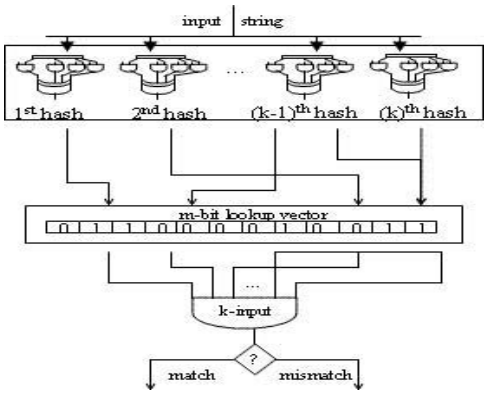

Figure 1. Block diagram of a typical Bloom filter

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.

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. 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. It may conclude that the input is a member of the signature set, although in reality the input may not be a member of the set. Following the analysis of [7], the false positive probability f is calculated by,

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

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 the 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. Since the number of hash functions in Bloom filters is large to reduce the false positive probability, it is intuitive that their total power consumptions are large. During the programming phase of the Bloom filter, not much can be done to reduce the power consumption, otherwise Bloom filter will produce many false positives. However, while performing lookups over the Bloom filter, the number of hash functions used to produce a decision can be reduced significantly. This is because Bloom filter never makes false negatives, and it is enough to find a zero on the *m-bit* long lookup vector to conclude that there is a *mismatch*. We call this type of lookup operation as *low power lookup technique*. The architecture to support such a lookup operation for a multi-hashing scheme is illustrated in Fig. 2.

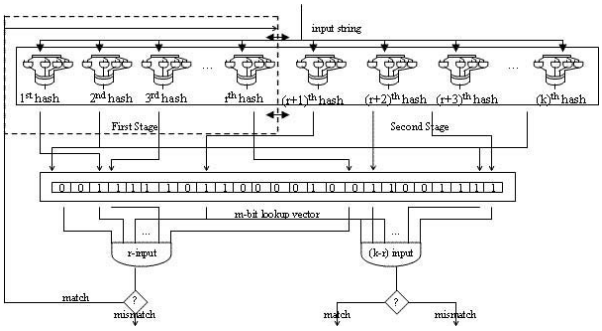

Figure 2. Block diagram of a two stage Bloom filter supporting low power lookup

At the core of the proposed architecture supporting low power lookup technique lies the division of the hash functions into two groups. These two groups are clearly identifiable on the Fig. 2. The first stage of hash functions 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. The result produced at the end of the first stage will be used as a select signal to start computing the second stage of hash functions. In the worst case, the new lookup operation will make use of all the hash functions in both groups, nonetheless most of the time the first group of hash functions will be enough to make a decision, which is a *mismatch*, claiming input is free of malicious content. Instead of computing k many hash functions, now it is enough to compute r many indices. This results in computational power savings.

3 A comparative power analysis

In this section, a theoretical approach is followed to analyze and compare the power consumptions of the different lookup operations available through two Bloom filter architectures presented in Fig. 1, and Fig. 2 respectively. A single Bloom filter shown in Fig. 1 uses k many hash functions in order to make a decision on the input given. Hence, the power consumption of a Bloom filter when performing a regular lookup operation is a summation of the power consumptions of each of the hash function computations, P_{H_i}, plus the power consumed accessing the memory for each

hash value computed, P_Q, plus the power consumed by an AND gate.

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_i} + P_Q) + P_{AND} \quad (2)$$

Power consumption of an AND gate is ignored hereafter, since it is minimal compared to the power used by the hash functions. We assume that the power required to query *m-bit* vector is approximately constant for each index calculated by any of the hash functions. The power consumption equation for a single Bloom filter simply becomes the total power used up by the hash functions and the power consumed by querying the m-bit vector for each hash value calculated.

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_i} + P_Q) \quad (3)$$

In order to compare the power consumption of a regular lookup operation to that of the low power lookup proposed, we use 16-bit implementation of hash functions. For comparison reasons, we do not consider different class of hash function implementations till the next section. We assume all of the hash functions implemented are from the universal class of hash functions called H_3 [5]. Hence, all of the *k* many hash functions are of type 16-bit H_3 class of hash functions, so Equ. 3 becomes

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_{i(H_{16})}} + P_Q) = k.(P_{H_{16}} + P_Q) \quad (4)$$

To derive the power consumption of the new lookup operation proposed, we follow an mathematical analysis similar to the analysis done in [9].Let us first derive the probability of match in the first stage. The probability that a bit is still unset after all the signatures are programmed into the the Bloom filter by using *k*-many independent hash functions is α.

$$\alpha = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} (for\ large\ m) \quad (5)$$

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

$$(1 - \alpha) \approx 1 - e^{\frac{-kn}{m}} \quad (6)$$

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*,

$$p = \prod_{i=1}^{r} (1 - \alpha) = (1 - \alpha)^r \approx (1 - e^{\frac{-kn}{m}})^r \quad (7)$$

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

$$1 - (1 - e^{\frac{-kn}{m}})^r \quad (8)$$

With a probability of *(1-p)* the first stage of the hash functions in the Bloom filter will produce a mismatch when performing a lookup operation. Otherwise, the first stage produces a match, then the second stage is used to compare the input with the signature sought as it is suggested by the architecture proposed. Therefore the power consumption of a Bloom filter shown in Fig. 2 is given by

$$\begin{aligned} P_{BF_{lowpower}} &= P_{1st-stage} + P\{match\} \times P_{2nd-stage} \\ P_{BF_{lowpower}} &= \sum_{i=1}^{r} (P_{H_i} + P_Q) + p \times \sum_{j=r+1}^{k} (P_{H_j} + P_Q) \\ &\quad + P_{AND} \end{aligned} \quad (9)$$

As we stated previously, for comparison purposes, $P_{H_{i,j}}$ are of type 16-bit H_3 class of universal hash functions. By substituting Equ. 7 into Equ. 9 power consumption of a Bloom filter shown on Fig. 2 becomes

$$\begin{aligned} P_{BF_{lowpower}} &= \sum_{i=1}^{r} (P_{H_{i(H_{16})}} + P_Q) \\ &\quad + (1 - e^{\frac{-kn}{m}})^r \times \sum_{j=r+1}^{k} (P_{H_{j(H_{16})}} + P_Q) \\ &= r.(P_{H_{16}} + P_Q) + \\ &\quad (1 - e^{\frac{-kn}{m}})^r (k - r)(P_{H_{16}} + P_Q) \end{aligned} \quad (10)$$

The power saving ratio, *PSR*, in a single Bloom filter implemented based on the architectures presented functioning on two different lookup techniques can be calculated as

$$PSR = \frac{\left(P_{BF_{regular}} - P_{BF_{lowpower}}\right)}{\mathrm{P}_{BF_{regular}}} \quad (11)$$

By substituting Equ. 4 and Equ. 10 into Equ. 11, the average power saving ratio, *PSR*, is found out to be

$$PSR = \frac{k - r + (r - k)\,(1 - e^{\frac{-kn}{m}})^r}{k} \quad (12)$$

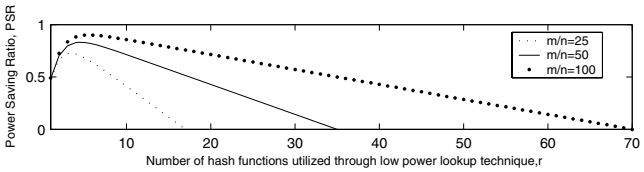

Figure 3. Power saving ratio w.r.t. number of hash functions in the first stage

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.

As it is seen in the Fig. 3, the number of bits per signature, $\frac{m}{n}$, increases, the amount of power conserved in the system increases. In other words, the power saving ratio becomes larger as $\frac{m}{n}$ increases. This is because, as $\frac{m}{n}$ increases, although probability of mismatch in the first stage stays the same for all configurations for a fixed value of *r* (See Equ. 8), the number of hash functions deployed in the first stage becomes a smaller portion of the overall hash functions deployed in each configuration. For a fixed value of *r*, $\frac{r}{k}$ decreases. This explains the reduction in power consumption. The less are the number of hash functions utilized through low power lookup technique, the more the power is saved. Another observation from Fig. 3 is that as the number of hash functions utilized through low power lookup technique increases, the power saving ratio, *PSR*, first increases to an optimum *PSR* value, thereafter it drops gradually.

4 Practical hashing functions utilized in Bloom filters

In this section, we will analyze the effects of utilizing different hashing functions in Bloom filters. Performance of different hash functions in hardware are investigated in [10]. We utilized three different types of hash functions in Bloom filters to examine the effects of them on the performance of low power lookup technique and possible power savings on multi-hashing schemes.

4.1 H_3 class of universal hash functions

Universal class of hash functions are first introduced by Carter et al. [5]. They defined a special class of hash functions and called them as class H_3. The definition is as follows. Given any string *X*, consisting of *b* bits, X = $<x_1, x_2, x_3, \ldots, x_b>$

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

$$h_i(x) = d_{i1} \bullet x_1 \oplus d_{i2} \bullet x_2 \oplus d_{i3} \bullet x_3 \oplus ...d_{ib} \bullet x_b \quad (13)$$

where d_{ij}'s are random coefficients uniformly distributed between *1* to size of the lookup vector, *m*, and x_k is the k^{th} bit of the input string. $\bullet$ is a bit by bit AND operation, and $\oplus$ is a logical exclusive OR (XOR) operation. A block diagram of the H_3 class of hash functions implemented is given in Fig. 4.

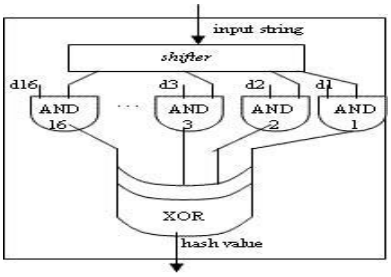

Figure 4. A block diagram of a H_3 class of universal hash function

Input is shifted one bit left till 16 bits are handled. Each bit is logically AND-ed with the random number. At the end, all AND results XOR-ed together to get a hash value. This type of hash functions are linear transformations, as a result they distribute the index values randomly. Implementation of these type of hash function requires 16 2-input AND gates and a single 16-input XOR gate for a 16 bit signature. They produce key values as the same size of the input. Pseudocode to implement H_3 class of hash functions is given below.

Pseudocode 1 A H_3 class of universal hash function

```
for each signature:
    - generate as many random numbers
    as the bits in the signature
    - left shift the signature to
    get to the specified bit
    - AND each shifted signature
    with the random number
    - XOR all the results of AND's
```

4.2 Bit extraction hashing functions

These type of hashing functions consists of selecting *j* bits out of *b* bits of the signature. Depending on the selection fashion of these bits out of input signature, they are classified as *regular* and *randomized* bit extraction hash functions. Since regular bit extraction hashing functions are constrained in number by the input length, we have used randomized bit extraction hash functions. Definition of a randomized bit extraction hashing function is as

follows. Given any string *X*, consisting of *b* bits, X = $<x_1, x_2, x_3, \ldots, x_b>$

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

$$h_i(x) = <x_{l_1}, x_{l_2}, x_{l_3}, \ldots, x_{l_j}> \quad (14)$$

where l_j's are random bit positions uniformly distributed between one to size of the input signature in bits, *b*, and x_{l_j} is the input bit located at l_j . A block diagram of randomized bit extraction hash functions implemented is illustrated in Fig. 5.

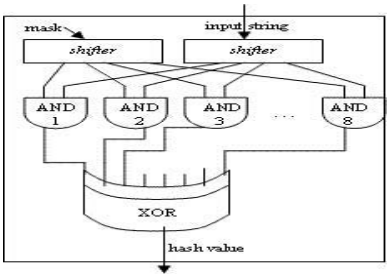

Figure 5. A block diagram of a bit extraction hash function

Implementation of these types of hash functions requires 8 2-input AND gates and a single 8-input XOR gate for a 16 bit signature. Shifter is necessary to left shift the bits in input as specified by random number, l_j. These type of hash functions produce key values shorter in bits than the size of the signature. They distribute keys randomly since the bit positions to extract the bits based on random numbers. Pseudocode to simulate this hash function is given below.

Pseudocode 2 A bit extraction hash function

```
for each signature:
    - generate as many random numbers
    as the bits in the indices
    - right shift  the signature to
    get to random bit position
    -adjust the bit at random position to
    the correct position at index
    by left or right shifting
    - XOR all the results of shifting
```

4.3 Hashing functions from exclusive OR method

These types of hash functions partition the *b* bit long input signature into *j* bits of segments. The segments are XOR-ed to get the hash value. The segments can be formed either in a regular manner or randomly like bit extraction hash functions. Since we want to have random indices, we have used random segment forming hash functions. The definition of the hashing functions from XOR method is as follows. Given any string *X*, consisting of *b* bits, X = $<x_1, x_2, x_3, \ldots, x_b>$

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

$$h_i(x) = (x_{s_1} \oplus x_{s_2})(x_{s_3} \oplus x_{s_4}) \ldots, (x_{s_{j-1}} \oplus x_{s_j}) \quad (15)$$

where s_j's are the uniformly distributed random bit positions in input string. x_{s_j} are the bits at the position specified by s_j. There are two segments of length *j*-bits are formed and XOR-ed. Fig. 6 illustrates a block diagram of a hash function from XOR method.

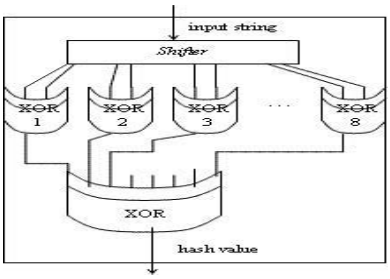

Figure 6. A block diagram of a hash function using the XOR method

Implementation of these types of hash functions requires a shifter to get to the bit at the random position, plus 8 2-input XOR gates, and a 8-input XOR gate. The length of the resulting hash value is smaller in bits than the input. However they map the inputs to the hash values in a completely random manner due to the random selection of bits from input. Following is a pseudocode to implement these type of hash functions.

Pseudocode 3 A hash function using the XOR method

```
for each signature:
    - generate twice as many random numbers
    as the bits in the indices
    - right shift  the signature to get the
    random bit positions for two segments
    - XOR the bits at each segment
    - right shift the XOR result to get
    correct position
```

5 Simulations

We have simulated the low power lookup technique presented in Fig. 2 with three different hash functions described in the previous section by using our custom-written

C program. In the simulation, parameters such as the size of the signature set, the type of signatures, *m*-bit long lookup vector size are varied to observe the power saving ratio. The length of the signatures are 16 bits. Indices, however, depend on the type of hash function utilized since both bit extraction and hash functions using the XOR method produce 8 bit hash values when the signature length is 16 bits. Table 1 depicts the results of the simulations illustrating large power gains through low power lookup operation.

Table 1. Power Savings through low power lookup operation for different hash functions

Simulation number	hash function type	number of signatures	lookup vector length	Power Saving (%)
Simul. 1	H_3	101	65536	97
Simul. 2	H_3	1000	65536	82
Simul. 3	EXT	6	256	88
Simul. 4	HXOR	6	256	85
Simul. 5	EXT	10	256	71
Simul. 6	EXT	20	256	60

The most important observation from the simulation results presented on Table 1 is that the low power lookup operation indeed provides significant power savings. The type of hash functions does not effect the power savings ratio as long as the bits allocated to the per signature, $\frac{m}{n}$, stays constant. However the type of the hash function affects the number of signatures that can be programmed, *n*, and size of the lookup vector, *m*. This is an expected result, since bit extraction and hash functions from XOR method produces 8-bit long indices whilst H_3 type of universal hash functions produce 16-bit indices. Consequently, the size of the lookup vector is limited by 256 or 65536 respectively. Hence, the type of the hash functions determines the size of the lookup vector *m*. As the number of signatures programmed in Bloom filter, *n*, increases, the number of hash function computations required to generate a decision on the maliciousness of the input rises. As a result, the power gain that is achieved in the Bloom filter through new lookup technique drops.

6 Conclusions

In this paper, we have proposed a low power lookup technique for multi-hashing schemes. Furthermore, an architecture supporting this new low power lookup technique for Bloom filters is described. Mathematical power analysis as well as simulations are carried out to show the effectiveness of the proposed method. In addition to that, three different types of hash functions are examined to observe the effects on the power savings and the operation of the multi-hashing scheme. The simulations performed revealed that the new lookup technique drastically decreases the power consumption of the Bloom filter. Simulations have also shown that the type of the hash function utilized in Bloom filter does not largely affect the power savings by low power lookup technique. However the type of the hash functions determines the size of the lookup vector, which in turn affects the number of allowed signatures programmed in to Bloom filter.

References

[1] M. Attig, S. Dharmapurikar, and J.L. Lockwood. "Implementation Results of Bloom Filters for String Matching", *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 322-323, 2004.

[2] M. Attig, and J.L. Lockwood."SIFT: Snort Intrusion Filter for TCP", *IEEE Symp. on High-Performance Interconnects*, Stanford, CA, 2005.

[3] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors", *Commun. ACM*, vol. 13, no. 7, pp. 422-426, July 1970.

[4] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey", *Internet Mathematics*, vol. 1, no. 4, pp. 485-509, July 2003.

[5] J. L. Carter and M. Wegman, "Universal classes of hash functions", *Journal of Computer and System Sciences*, vol. 18, pp. 143-154, 1978.

[6] S. Czerwinski, B. Y. Zhao, T. Hodes, A. D. Joseph, and R. Katz. "An Architecture for a Secure Service Discovery Service", *Proc. ACM/IEEE Int'l Conf. on Mobile Computing and Networking*, pp. 24-35, 1999.

[7] 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, 2004.

[8] S. Dharmapurikar, H. Song, J. Turner, and J. W. Lockwood, "Fast Hash Table Lookup Using Extended bloom Filter: An Aid to Network Processing", *ACM/SIGCOMM*, pp. 181-192. Philadelphia, 2005.

[9] M. Mitzenmacher, "Compressed Bloom filters", *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604-612, October, 2002.

[10] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient Hardware Hashing Functions for High Performance Computers", *IEEE Trans. on Computers*, vol. 48, no. 12, pp. 1378-1381, 1997.