International Journal Of Computer Science And Applications Vol. 1, No. 3, December 2008
ISSN 0974-1003

Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based low power multiple-hashing Bloom Filters

Arun M
Lecturer – ECE / Research Scholar
KSR College of Engineering
Tiruchengode – 637 215
+91 9942533393
aruninvlsi@gmail.com

Dr. A. Krishnan
Dean
KSR College of Engineering
Tiruchengode – 637 215
+91 4288 274757
amasikrishnan@hotmail.com

Prof.PS.Periasamy
Professor/ECE
KSR College of Engineering
Tiruchengode – 637 215
+91 4288 2274757
psp_03@yahoo.co.in

ABSTRACT
Modern Network Intrusion Detection Systems (NIDS) inspect the network packet payload to check if it conforms to the security policies of the given network. This process, often referred to as deep packet inspection, involves detection of predefined signature strings or keywords starting at an arbitrary location in the payload. String matching is a computationally intensive task and can become a potential bottleneck without high-speed processing. Since the conventional software-implemented string matching algorithms have not kept pace with the increasing network speeds, special purpose hardware, Field Programmable Gate Arrays (FPGAs), have been introduced. A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support string matching of Network Intrusion Detection System (NIDS). Bloom filters allow false positives but the space savings often outweigh this drawback when the probability of an error is controlled. FPGAs have achieved sufficient capability to performing complex network processing in programmable hardware. Network devices utilizing FPGAs show a desirable balance between performance and flexibility, which makes FPGA preferable to pure software and ASIC solutions. We present an implementation of low power multiple hashing bloom filter using FPGAs. We describe how multiple hashing Bloom filters can be implemented feasibly on Xilinx XCV2000E FPGA.

Categories and Subject Descriptors
B.8.2 [**Hardware**]: Performance and Reliability - *Performance Analysis and Design Aids*; C.2.M [**Computer Communications Networks**]: Miscellaneous;

General Terms
Algorithm, Performance, Design

Keywords
Bloom filter, false positive, false negative, FPGA, NDIS, String Matching, Hash function

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© Copyright 2008 Research Publications, Chikhli, India

1 INTRODUCTION
Network Intrusion Detection System is one of the security application can be made use of multi-hashing schemes. 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 [1]. A detailed survey of Bloom filters for networking applications can be found in [2]. A hardware system, FPGAs, consisting of Bloom filters to detect malignant content, is described 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 5W [5]. To decrease the power consumption of Bloom filter, a new lookup technique is proposed [5], 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 filter is presented in [5], in which mathematical analysis carried out clearly states the efficiency of the new lookup technique in terms of power. This paper presents hardware architectures for implementation of the different classes of the hash functions utilized in programming and lookup operations of Bloom filter. The implementation results using Xilinx XCV2000E FPGA with different hashing functions in varying configurations of the Bloom filter is discussed.

2 BLOOM FILTER THEORY
Bloom filter was formulated by Burton H. Bloom in 1970 [1] and is used widely today for different purposes including web caching, intrusion detection, content based routing [2]. The theory behind Bloom filters is described in this section. Given a string *X*, the Bloom filter computes k hash functions on it producing hash values ranging from 1 to *m*. It then sets *k* bits in a m bit long vector at the addresses corresponding to the *k* hash values. The same procedure is repeated for all the members of the set. This process is called "programming" of the filter. Figure 1 illustrates this concept. In this figure, two messages, *X1, X2* are being programmed in the Bloom filter which has *k=4* hash functions and *m=13* bits in the array. Note that different strings can have overlapping bit patterns as shown in this figure. The query

process is similar to programming, where a string whose membership is to be verified is input to the filter. The Bloom filter generates *k* hash values using the same hash functions it used to program the filter. The bits in the *m* - bit long vector at the locations corresponding to the *k* hash values are looked up. If at least one of these *k* bits is found not set then the string is declared to be a non-member of the set. If all the bits are found to be set then the string is said to belong to the set with a certain probability. This uncertainty in the membership comes from the fact that those *k* bits in the *m*-bit vector can be set by any of the *n* members. Thus finding a bit set does not necessarily imply that it was set by the particular string being queried. However, finding a bit not set certainly implies that

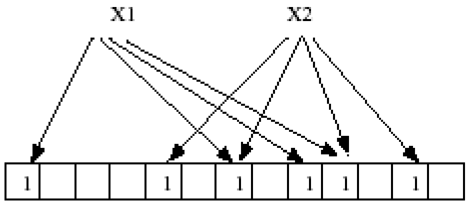

Figure 1.Programming multiple strings in the Bloom filter. Strings X1 and X2 are being programmed. Here k=4 and m=13

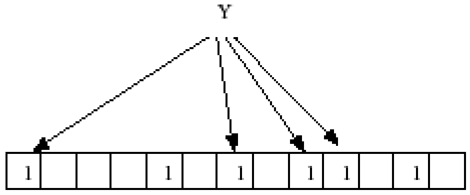

Figure 2. Querying a Bloom filter with a string. Bloom filter gives a 'match' for string Y since all the hash bits are set

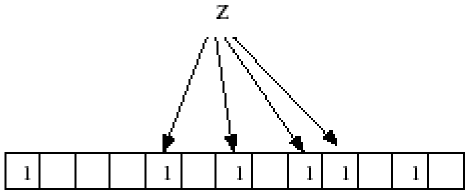

Figure 3. False positives. Bloom filter gives a match for string Z though it is not programmed in it, since all the hash bits are set. This is a false positive

the string does not belong to the set, since if it did then all the *k* bits would definitely have been set when the Bloom filter was programmed with that string. This explains the presence of false positives in this scheme, and the absence of any false negatives. The concept is illustrated in Figures 2 and 3. A string *Y* is input for verifying its membership. The same hash functions calculate *k* hash values over *Y* and all the bits corresponding to these hash values are found to be set. Similarly when another string *Z* is input for membership verification, all the corresponding bits in the bit array are found to be set although there is no such string programmed in the filter, i.e. neither *X1* nor *X2* has the same bit pattern as *Z*. Hence, clearly it is a false positive. The false positive rate, *f*, is expressed as [1]

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

Where, *n* is the number of strings programmed into the Bloom filter. The value of *f* can be reduced by choosing appropriate values of m and *k* for a given size of the member set, *n.* It is clear that the value of *m* needs to be quite large compared to the size of the string set i.e., *n*. Also, for a given ratio of *m/n,* the false positive probability can be reduced by increasing the number of hash functions *k.* In the optimal case, when false positive probability is minimized with respect to *k,* we get the following relation

$$f = \left(\frac{m}{n}\right) \ln 2 \qquad (2)$$

This corresponds to a false positive probability of

$$f = \left(\frac{1}{2}\right)^{k} \qquad (3)$$

The ratio *m/n* can be interpreted as the average number of bits consumed by a single member of the set. It should be noted that this space requirement is independent of the actual size of the member. In the optimal case, the false positive probability decreases exponentially with a linear increase in the ratio *m/n.* Secondly, this also implies that the number of hash functions, *k,* and hence the number of random lookups in the bit vector required to query one membership is proportional to *m/n.*

2.1 Typical Bloom Filter

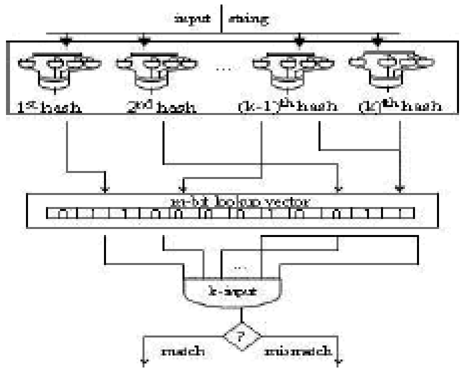

Figure 4. Block Diagram of typical Bloom filter

A block diagram of a typical Bloom filter is illustrated in Fig. 4. 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. 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 e signature set with a *false positive probability*, which is called a *match*.

3 LOW POWER LOOK UP

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 [4], the false positive probability f is calculated by (1). In order to minimize the false positive probability, the value of m must be quite larger than n. For a fixed value of 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*. Ilhan Kaya [5] calls 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 Figure 5.

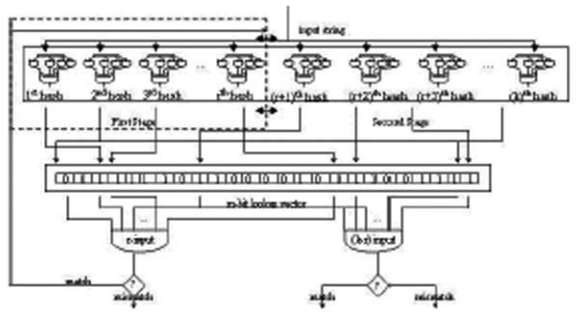

Figure 5. Block Diagram of low power look up Bloom filter

4 PRACTICAL HASHING FUNCTIONS

This section explains three different types of practical hashing functions. Performances of different hash functions in hardware are investigated in [6]. 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 device utilization of multi-hashing schemes on Xilinx XCV2000E FPGA.

4.1 H_3 class of universal hash functions

Universal class of hash functions are first introduced by Carter J.L. [3]. 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 = <x1*, x2, x3, . . . , xb> ith* hash function over the string *X* is defined as

$$h_i(x) = d_{i1} \cdot x_1 \oplus d_{i2} \cdot x_2 \oplus d_{i3} \cdot x_3 \oplus \ldots\ldots d_{ib} \cdot x_b \qquad (4)$$

where *dij* 's are random coefficients uniformly distributed between *1* to size of the lookup vector, *m*, and *xk* is the *kth* bit of the input string. • is a bit by bit AND operation, and is a logical exclusive OR (XOR) operation. A block diagram of the *H*3 class of hash functions implemented is given in Figure 6.

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.

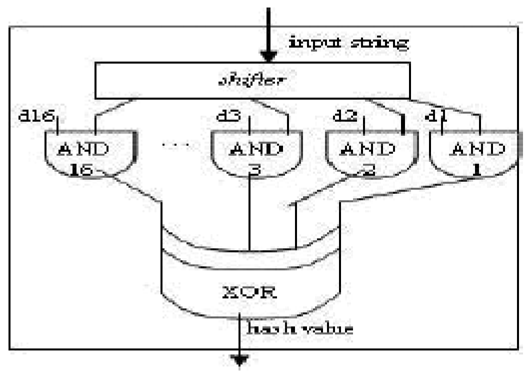

Figure 6. A block diagram of a *H*3 class of universal hash function

4.2 Bit Extraction hashing functions

This 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 = <x1*, x2, x3, . . . , xb> ith* hash function over the string *X* is defined as

$$h_i(x) = <x_{l1}, x_{l2}, x_{l3}, \ldots, x_{lj}> \quad (5)$$

where *lj* 's are random bit positions uniformly distributed between one to size of the input signature in bits, *b*, and *xlj* is the input bit located at *lj* . A block diagram of randomized bit extraction hash functions implemented is illustrated in Figure 7.

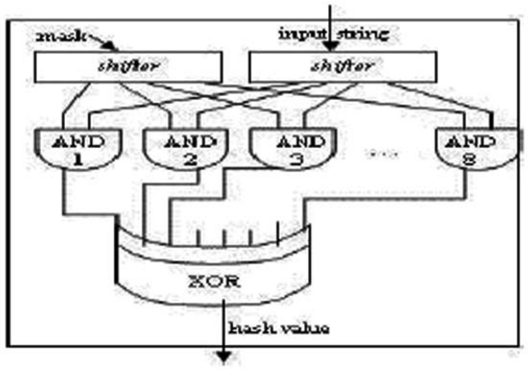

Figure 7. A block diagram of a bit extraction hash function

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 speci.ed by random number, *lj* . These types of hash functions produce key values shorter in bits than the size of the signature.

4.3 Hashing functions from XOR 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 = <x1*, x2, x3, . . . , xb>* *ith* hash function over the string *X* is defined as

$$hi(x) = (xs1 \oplus xs2)(xs3 \oplus xs4) \ldots, (xsj-1 \oplus xsj) \qquad (6)$$

where *sj* 's are the uniformly distributed random bit positions in input string. *xsj* are the bits at the position specified by *sj* . There are two segments of length *j*-bits are formed and XOR-ed. Figure 8 illustrates a block diagram of a hash function from XOR method.

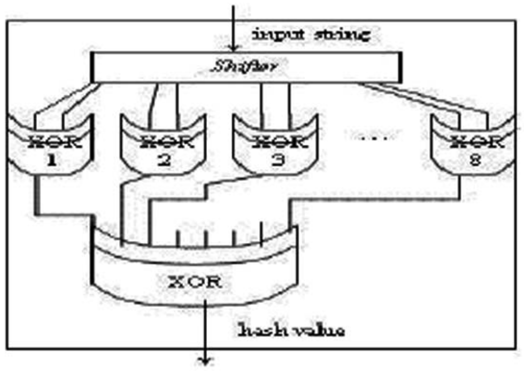

Figure 8. A block diagram of hash function using XOR method

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-inputXOR gate. The length of the resulting hash value is smaller in bits than the input.

5 IMPLEMENTATION RESULTS

Logical designs of low power look up bloom filter with respect to types of hashing functions were implemented on Xilinx XCV2000E FPGA and utilization of LUTs, Flip Flops and Block RAMs are summarized in the following table. Device utilization is higher in the type of bit extraction hashing function.

Table 2. Table captions should be placed above the table

Hashing Function	LUTs	Flip Flops	Block RAMs
Universal	2990 (4.4%)	2295	6
Bit Extraction	4550 (9%)	3998	7
XOR Method	3050 (4.5%)	2567	6

6 CONCLUSION

When compare to the device utilization of Bloom filter for string matching proposed by [4], low power look up Bloom filter of different hash functions [5] consumes double the devices on Xilinx XCV2000E FPGA. Power consumption of Bloom filter is reduced when adopting low power look up and proven theoretically [5]. Implementation and its observation reflect the tradeoff between the space and power.

7 REFERENCES

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

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

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

[4] Dharmapurikar, S., Krishnamurthy, P., Sproull, T.S. and Lockwood, J. W. "Deep Packet Inspection Using Parallel Bloom Filters", *IEEE Micro*, vol. 24, no. 1, pp. 52-61, 2004.

[5] Ilhan Kaya, Taskin Kocak, "A Low Power Lookup Technique for Multi-Hashing Network Applications", IEEE Computer Society, Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI'06)

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