Deep Packet Inspection using Parallel Bloom Filters

Sarang Dharmapurikar Praveen Krishnamurthy Todd Sproull John Lockwood
Computer Science and Engineering Department
Washington University in Saint Louis
One Brookings Drive, Saint Louis, MO, 63130
{sarang,praveenk,todd,lockwood}@arl.wustl.edu
http://www.arl.wustl.edu/arl/projects/fpx

Abstract

Recent advances in network packet processing focus on payload inspection for applications that include content-based billing, layer-7 switching and Internet security. Most of the applications in this family need to search for predefined signatures in the packet payload. Hence an important building block of these processors is string matching infrastructure. Since conventional software-based algorithms for string matching have not kept pace with high network speeds, specialized high-speed, hardware-based solutions are needed. We describe a technique based on Bloom filters for detecting predefined signatures (a string of bytes) in the packet payload. A Bloom filter is a data structure for representing a set of strings in order to support membership queries. We use hardware Bloom filters to isolate all packets that potentially contain predefined signatures. Another independent process eliminates false positives produced by Bloom filters.

We outline our approach for string matching at line speeds and present the performance analysis. Finally, we report the results for a prototype implementation of this system on the FPX platform. Our analysis shows that with the state-of-the-art FPGAs, a set of 10,000 strings can be scanned in the network data at the line speed of OC-48 (2.4 Gbps).

1 Introduction

There is a class of packet processing applications which need to inspect packets on the link deeper than the protocol headers and analyze its payload. For instance, network security applications require that the packets containing certain malicious strings in the payload should be dropped. Content-based billing techniques analyze media files and bill the receiver based on the material transferred over the network. Content forwarding applications look at the HTTP headers and direct the requests to predetermined servers for load balancing.

Most payload scanning applications have a common requirement for string matching. For example, a media file can be characterized by the presence of a string of bytes (for the rest of the paper, a string is synonymous to a *signature*) and its transmission across a link can be monitored by looking out for the presence of this string on the network. Well known Internet worms like Nimda, Code Red and Slammer contain a string of bytes as signature. Since the location of such strings in the packet payload is not deterministic, such applications need the ability to detect strings of different lengths starting at the arbitrary locations in the packet payload.

Such packet inspection applications, when deployed at router ports, must be able to operate at wire speeds. With the networking speeds doubling every year, it is becoming increasingly difficult for software based packet monitors to keep up with the line rates. This has underscored the need for the specialized hardware-based solutions which are portable and operate at wire speeds.

We describe a hardware-based technique using Bloom filters [1] which can perform string detection in streaming data without degrading network throughput. A Bloom filter is a data structure that stores a set of signatures compactly by computing multiple hash functions on each member of the set. With this technique, a database of strings is queried for the membership of a particular string. The answer to this query can be false positive but never a false negative. An important property of this data structure is that the computation time involved in performing the query is independent of the number of strings in the database provided the memory used by the data structure scales linearly with the number of strings stored in it. Furthermore, the amount of storage required by the Bloom filter for each string independent of its length.

Signatures are grouped according to their length (in bytes) and stored in parallel Bloom filters. Each of these Bloom filters scans the streaming data for the presence of strings of its corresponding length. Whenever a Bloom filter detects any suspicious string, an *analyzer* is probed with this string to decide if it indeed belongs to the given set of strings or if it is a false positive.

The focus of this paper is to present the architecture of hardware implementation of Bloom filters and analyze its performance. The analysis for the Field Programmable Gate Arrays (FPGAs) shows that OC-48 line speed scanning for more than 10,000 strings can be achieved with this technique. The rest of the paper is organized as follows. Section 2 gives the background on Bloom filters. Section 3 describes the approach for signature detection at line speeds and quantifies the trade-offs between resources and system performance. Section 4 highlights some of the important challenges in the hardware implementation and describes how to deal with them. Section 5 gives an analysis of the performance of our prototype system implemented using Xilinx XCV2000E FPGA. Section 6 discusses recent related work and point out some of the important differences between this work and that of others. Section 7 summarizes the contributions of this paper and discusses directions for future work.

2 Theory of Bloom Filters

Bloom filters use a randomized technique to test membership queries on a set of strings. Given a string X, the Bloom filter computes k hash functions on it producing k 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. 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 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 false positive rate, *f*, is expressed as

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

where, n is the number of strings programed 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 $\frac{m}{n}$, 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

$$k = (m/n)ln2 \tag{2}$$

This corresponds to a false positive probability ratio of

$$f = (1/2)^k \tag{3}$$

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 Counting Bloom Filters

One property of Bloom filters is that it is not possible to delete a member stored into the filter. Deleting a particular entry requires that the corresponding k hashed bits in the bit vector be set to zero. This could disturb other members programmed into the filter which hash to any of these bits. In order to solve this problem, the idea of the *Counting Bloom filters* was proposed in [4]. A Counting Bloom filter maintains a vector of counters corresponding to each bit in the bit-vector. Whenever a member is added to or deleted from the filter, the counters corresponding to the k hash values are incremented or decremented, respectively. When a counter changes from zero to one, the corresponding bit in the bit-vector is set. When a counter changes from one to zero, the corresponding bit in the bit-vector is cleared.

It is important to note that the counters are changed only during addition and deletion of strings in a Bloom filter. For applications like network intrusion detection, these updates are relatively less frequent than the actual query process itself. Hence, counters can be maintained in software and the bit corresponding to each counter is maintained in hardware. Thus, by avoiding counter implementation in hardware, memory resources can be saved.

3 System Description

From the previous section it can be seen that, the amount of computation required to detect the membership of a string in a Bloom filter is independent of the size of set of

strings. Secondly, the computation, which requires generation of hash values, can be performed in special purpose hardware (such as FPGAs). Hence, hardware implementation of Bloom filters is quite attractive for quickly isolating the potentially matching strings from the streaming network data.

3.1 Overview of the System

A predefined set of signatures, grouped by their lengths, is stored in a set of parallel Bloom filters in hardware. Each of these Bloom filters contains the signatures of a particular length. These Bloom filters are used to monitor network traffic and operate on strings of the corresponding length from network data, as shown in Figure 1. Each string is tested for its membership in the Bloom filters. If a string is found to be a member of any Bloom filter then it is declared as a possible matching signature. Such strings are probed into an $analyzer$ which determines if a string is indeed a member of the set or a false positive. Stated differently, the analyzer is a deterministic string matching algorithm which verifies if the input string is a member of given set or not. When a string of interest is found, an appropriate action (drop, forward, log) can be taken on the packet.

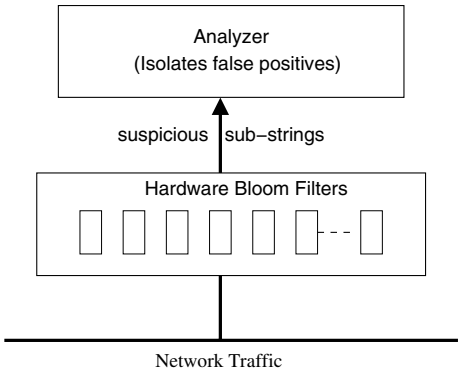

Figure 1. Bloom filters scanning all traffic on multi-gigabit network for predefined signatures

Let the length of the signatures range from L_{min} to L_{max}. The Bloom filter engine reads as input a data stream that arrives at the rate of one byte per clock cycle. It monitors a window of L_{max} bytes as shown in Figure 2.

When this window is full, it contains $L_{max} - L_{min}$ different sub-strings which are potential signatures. Membership of each of these sub-strings is verified using the corresponding Bloom filter. Each of the hardware Bloom filters in our design gives one query result per clock cycle. In this way, memberships of all the $L_{max} - L_{min}$ strings can be verified in a single clock cycle. If none of the sub-strings shows a match, the data stream can be advanced by a byte.

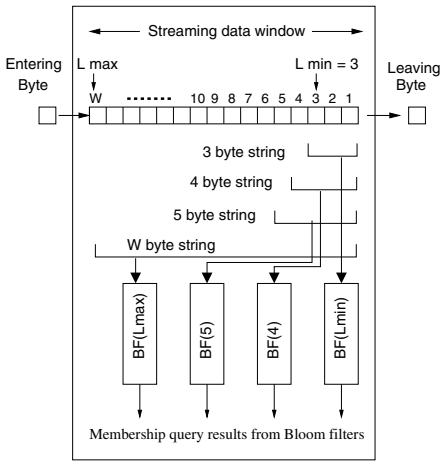

Figure 2. A window of streaming data containing strings of length from $L_{min} = 3$ to $L_{max} = W$. Each string is examined by a Bloom filter

By monitoring a window in this way, eventually all the possible strings of length from L_{min} bytes to L_{max} bytes in every packet are scanned. In the case of multiple sub-strings matching within a single window, the longest sub-string among them is considered as the string of interest. This policy is called the Longest Sub-string First (LSF). Thus, in the case of multiple matches at the same time in the array of Bloom filters, the analyzer is probed with the longest sub-string down to the shortest sub-string. The search stops as soon as a sub-string is first confirmed by the analyzer. After the search is over, the window is advanced by a byte and the same procedure is repeated.

3.2 Throughput of the System

From the previous section, it should be clear that the Bloom filters do in fact accelerate the string matching by isolating most of the strings from the network data and processing just those strings to the analyzer which have a very high probability of matching. It is important to note that a string of interest never goes unnoticed since Bloom filter never gives false negatives. An expression that gives the statistical throughput of the system can derived. The following notations are used:

- τ: time (in seconds) required to check the presence of a string using the analyzer.
- f: false positive probability of each Bloom filter
- B: the total number of Bloom filters in the system
- F: the clock frequency (in Hz) at which the system operates

Within a window, it is possible that multiple Bloom filters show matches corresponding to their sub-strings. For a

search that ends at the l^{th} Bloom filter, let B_l denote the number of Bloom filters filters for lengths higher than l. The probability that exactly i filters associated with string lengths greater than l will generate false positives is given by

$$P_l = \binom{B_l}{i} f^i (1-f)^{B_l - i} \quad (4)$$

For each value of i, i additional probes into the analyzer would be required. Hence, the expected number of additional probes in the analyzer required can be expressed as

$$E_l = \sum_{i=1}^{B_l} i \binom{B_l}{i} f^i (1-f)^{B_l - i} \quad (5)$$

which is the mean for a binomial distribution with B_l elements and a probability of success f. Hence,

$$E_l = B_l f \quad (6)$$

The equation above shows that the expected number of $additional$ probes into the analyzer, when the search ends at l^{th} Bloom filter, is equal to the number of Bloom filters for the longer string lengths times the false positive probability (which is the same for all the filters). In the worst case, $B_l = B$, hence value of E_l is upper bounded Bf. This upper bound on the expected number of additional probes in the analyzer is used for further calculations. Since each of these probes requires time τ in the worst case, The expected additional time spent in probes can be expressed as

$$T_{add} = Bf\tau \quad seconds \quad (7)$$

Since the search ends at Bloom filter l, if it shows a match then it means a $true$ match has been found, otherwise it means there are no Bloom filters for string lengths less than l that show a match in the given window. In the earlier case, again, time T will be spent to probe the analyzer for the confirmation of true match. In the latter case, time equal to the clock period, $(1/F)$, will be spent. If the frequency of occurrence of a true string in the data stream is denoted by p then, on an average, the time spent during the end of the search within a window is

$$T_{end} = p\tau + (1-p)(1/F) \quad seconds \quad (8)$$

Thus, on an average, a total of T_{add} + T_{end} is spent in examining a window, after which the window is advanced by a byte. Hence the throughput of the system, R, can be expressed as

$$R = \frac{1}{T_{add} + T_{end}} bytes/s = \frac{1}{Bf\tau + p\tau + (1-p)(1/F)} bytes/s \quad (9)$$

The system as shown in the Figure 2 processes one byte every clock cycle. If the set of Bloom filters is grouped in a single scanner engine then multiple such engines can be instantiated to monitor the data stream starting with an offset of a byte. Thus, if three such engines are used then the byte stream can be advanced by three bytes at a time, as shown in Figure 3.

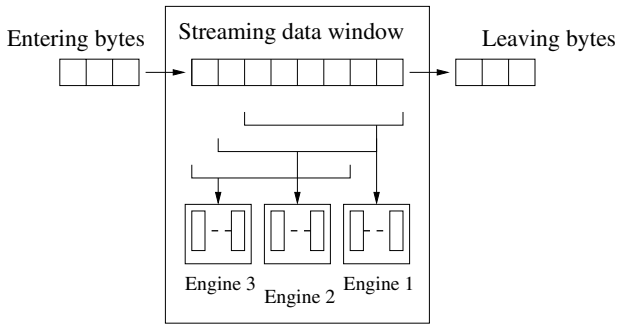

Figure 3. Instantiating multiple parallel engines for better throughput

If each of the G parallel engines is coupled with an independent analyzer circuit, then the throughput is simply GR. Alternatively if they share the same analyzer then the throughput expressed in equation 9 needs to be recalculated since there is more contention for accessing the analyzer. In this case, the throughput, R_G, becomes

$$R_G = \frac{G}{GT_{add} + T_{end}} bytes/s = \frac{G}{GBf\tau + p\tau + (1-p)(1/F)} bytes/s \quad (10)$$

with the assumption that only one of the G engines finds a true match in a given window.

3.3 Tuning the System Parameters

Equation 10 can be simplified by considering realistic values of different parameters. Recall that the analyzer is assumed to require a constant time, τ, to check the input string in the database. Such an analyzer can be easily designed as a hash table. A set of strings can be inserted into a hash table with collisions resolved by chaining the colliding strings together in a linked list. Such a hash table has an average of constant search time [15]. This hash table can be stored in an off-chip commodity SRAM. Although the average search time in such a hash table can be made independent of the number of strings by using ample of memory, the string retrieval time from the memory depends on the length of the string. For a sub-string of length 32 bytes that is probed into the hash table maintained in an SRAM

with a data bus width of 4 bytes, 8 clock cycles are required to retrieve the string and compare it against the input. With L_{max} set to 32, even with an assumption of one collision and accounting for memory access latencies, a hash probe should require no more than 20 clock cycles. Hence, $\tau = 20/F$ i.e. 20 times the system clock period, is used.

Since the frequency of occurrence of the strings being looked for in the streaming data is typically very low, small values of p can be assumed. The values, $p = 0.001$ (i.e. on an average for every thousand characters scanned, one string of interest is found) is assumed for this example. Considering the values, $B = 24$ (hence, signatures of 24 distinct lengths can be scanned), $F = 100MHz$ (which is typically the speed at which FPGAs and commodity SRAMs operate), and $G = 4$ (i.e. 4 Bloom filter engines are used in parallel), and substituting these values in equation 10 we obtain the following expression for the throughout

$$R_4 = \frac{3.2}{1920 \times f + 1.019} Gigabits/s \qquad (11)$$

In order to express the value of f, the following notations are used

- f_i: false positive probability of the i^{th} Bloom filter within an engine
- m_i: memory allocated to Bloom filter i
- n_i: number of strings stored in Bloom filter i
- M: the total amount of on-chip memory available for Bloom filters of all G engines. Hence, each engine is allocated M/G amount of memory, which is shared by B Bloom filters in it.
- N: the total number of strings being stored in the B Bloom filters of an engine. Thus, $N = \sum_{i=1}^{B} n_i$

Since the false positive probability of all the Bloom filters of an engine is engineered to be the same, say f, using equation 3,

$$f_i = f = \left(\frac{1}{2}\right)^{\left(\frac{m_i}{n_i}\right) \ln 2} \quad \forall i \in [1 \ldots B] \qquad (12)$$

This implies that

$$\frac{m_i}{n_i} = \frac{m_{i+1}}{n_{i+1}} = \frac{\sum m_i}{\sum n_i} = \frac{M/G}{N} \forall i \in [1 \ldots B] \qquad (13)$$

Therefore,

$$f = \left(\frac{1}{2}\right)^{\left(\frac{M/G}{N}\right) \ln 2} \qquad (14)$$

After substituting the value of f in expression 11 and plotting the value of the throughput R_G for a total of $N = 10,000$ strings, the graph shown in figure 4 is obtained.

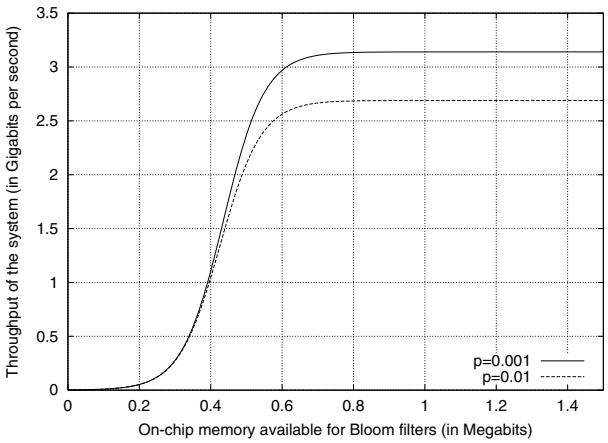

Figure 4. The throughput of the system as a function of the available on-chip memory. Two different values of p, the probability of true occurrence of strings, are considered. The system is tuned for a total of N=10,000 strings of B=24 distinct lengths. The maximum string probing time in the analyzer is 20 times the clock period of the system with the clock frequency, F, being 100MHz

As the figure shows, the effect of false positives is dominant for small vales of memory, which results in a lower throughput. However, as the amount of memory increases, the throughput increases rapidly and saturates to over 3 Gbps. Thus, with merely 1 Megabit of on-chip memory, 10,000 strings can be scanned at the line rate of OC-48 (i.e. 2.4 Gbps). Moreover, the number of strings can be increased with a proportional increase in the memory.

4 Hardware Design Considerations

The effect of increasing on-chip memory on the system throughput was analyzed in Section 3.3. Equation 2 shows that for a fixed number of strings in a Bloom filter, the number of bits allocated to a member in a Bloom filter also decides the number of hash functions needed for that Bloom filter. For example, if 50 bits per member on an average (i.e., $m/n = 50$) are allocated, then number of hash functions need to be $k \approx 50 \times 0.7 = 35$ and the false positive probability is $(1/2)^{35} \approx 3 \times 10^{-11}$. This section describes how to support these hash functions in hardware and perform the corresponding random lookups in on-chip memories.

4.1 Hash Functions

Although this scheme uses a considerable number of hash functions, implementing these in hardware is relatively

inexpensive. A class of universal hash functions called H_3 described in [12] were found to be suitable for hardware implementation. It should be recalled that k hash functions are generated for each filter. Hence, the total number of distinct hash functions needed is $k \times B$ for one engine. Following is the description of how this hash matrix is calculated.

For any i^{th} byte represented as

$$byte_i =< b_1^i, b_2^i, b_3^i ... b_8^i >$$

first the l^{th} hash function h_i^l on it is calculated as follows

$$h_i^l = d_{i1}^l.b_1^i \oplus d_{i2}^l.b_2^i \oplus d_{i3}^l.b_3^i \oplus ..... \oplus d_{i8}^l.b_8^i \qquad (15)$$

where d_{ij}^l is a predetermined random number in the range [1...m]. Then the l^{th} hash function over all the i bytes is calculated as

$$H_i^l = H_{i-1}^l \oplus h_i^l \quad \forall i \in [1 \ldots W], \quad \forall l \in [1 \ldots k] \quad (16)$$

with $H_0^l = 1$. It can be observed that the hash functions are calculated cumulatively and hence the results calculated over the first i bytes can be used for calculating the hash function over the first $i+1$ bytes. This property of the hash functions results in a regular and less resource consuming hash function matrix.

4.2 Using Multi-port Embedded Memories

Each hash function corresponds to one random lookup in the m-bit long memory array. Thus, for 35 hash functions, the Bloom filter memory should be able to support 35 random lookups every clock cycle. Figure 5a illustrates these requirements graphically. Memories with such density and lookup capacity are realized by making use of the embedded Random Access Memories (RAMs) in the VLSI chip.

With today's state-of-the-art VLSI technology, it is easy to fabricate memories that hold a few million bits. A concise summary of the recent embedded memory technologies can be found in [3].

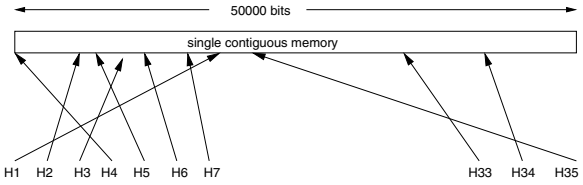

Figure 5a. A Bloom filter with single memory vector which allows 35 random lookups at a time

For embedded memories limited in their lookup capacity, a desired lookup capacity can be realized by employing multiple memories with smaller lookup capacity. For instance, Lucent's memory core [3] has five read-write ports. Hence, using this memory core, five random memory locations can be read in a single clock cycle. In order to perform 35 concurrent memory operations, seven parallel memory cores, each with $1/7^{th}$ the required array size, are needed (See Figure 5b). Since the basic Bloom filter allows any hash function to map to any bit in the vector, it is possible that for some member, more than 5 hash functions map to the same memory segment, thereby exceeding the lookup capacity of this memory core. This problem can be solved by restricting the range of each hash function to a given memory. Thus, memory contention can be prevented.

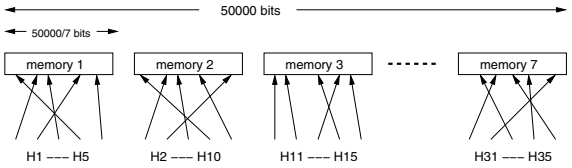

Figure 5b. Bloom filter implemented using multiple smaller memories with smaller lookup capacity (e.g. 5) to realize the desired lookup capacity (e.g. 35)

In general, if h is the maximum lookup capacity of a RAM as limited by the technology then k/h such memories, each of size $\frac{m}{(k/h)}$ can be combined to realize the desired capacity of m bits and k hash functions. Only h hash functions are allowed to map to a single memory. The false positive probability can be expressed as

$$f' = [1 - (1 - \frac{1}{\frac{m}{(k/h)}})^{hn}]^{(k/h)h} \approx [1 - e^{-nk/m}]^k \quad (17)$$

Comparing equation 17 with equation 1, it can be seen that restricting the number of hash functions mapping to a particular memory has negligible effect on the false positive probability.

5 Prototype Implementation and Results

A prototype of the system was implemented in a Field Programmable Gate Array (FPGA), a Xilinx XCV2000E, on the Field Programmable Port Extender (FPX) [8]. This particular application used single size signatures (hence $B = 1$) of 32 bytes to detect the transfer of media files over the network.

The XCV2000E FPGA has 160 embedded block memories, each of which can be configured as single bit wide, 4096 bit long array that can perform two read operations using dual ports in a single clock cycle. This memory is used to construct a Bloom filter, with $m = 4096$ and $k = 2$. Using equations 2 and 3, it can be seen that this block RAM

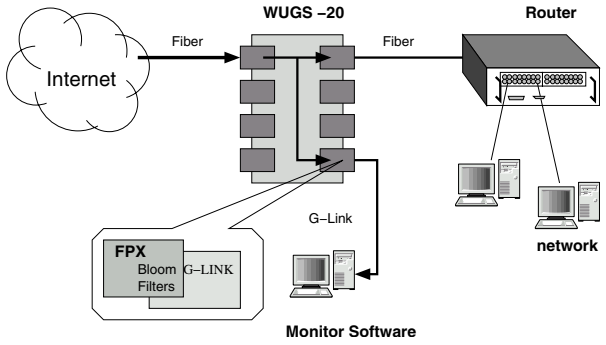

Figure 6. Deployment setup for the prototype

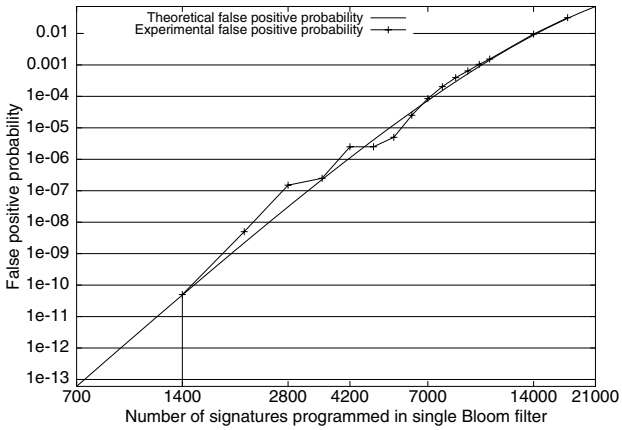

Figure 7. The false positive probability as a function of the number of signatures stored into one Bloom filter engine.

can support $n = (m/2) \times ln2 \approx 1434$ signatures with a false positive probability $1/2^2 = 0.25$. By employing 5 such block RAMs, a mini-Bloom filter with string capacity 1434 and false positive probability of $f = 1/2^{10}$ can be constructed. Using 35 block RAMs, 7 such mini-Bloom filters can be constructing giving an aggregate capacity of $1434 \times 7 = 10038$ strings. These mini-Bloom filters constitute one engine. Four parallel engines can be instantiated (which together consume $35 \times 4 = 140$ block RAMs) to push 4 bytes in a single clock cycle (hence, $G = 4$). Substituting these values in equation 10 it can be seen that the throughput of over $2.46Gbps$, which corresponds to a line rate of OC-48, can be achieved.

A functional prototype with a single Bloom filter engine has been implemented. It consumes, 35 block RAMs and only 14% of the avialble logic resources on the FPGA. The system operates at 81 MHz. Figure 6 shows a deployment setup for this system. Traffic from the Internet passes through WUGS-20, a gigabit switch, where the data is multicast to an FPX and to a router. The router contains a Fast Ethernet blade to which the workstations connect. Data from the workstations pass to the router then to the Internet through the WUGS-20. Traffic coming from the Internet to the router is processed in the FPX. The analyzer was replaced by a software process in a standalone workstation that checks all packets marked as a possible match by the Bloom filters in the FPX.

Experiments were performed to observe the practical performance of Bloom filters in terms of the false positive rate. The filter was programmed with different number strings and the false positives were measured. Figure 7 shows the result of this experiment along with the theoretical value. The figure shows that the experimental results are consistent with the theoretical predictions. Note that in our experiments the system did not produce any false positives for strings less than 1400 (with approximately 200 strings in each mini-Bloom filter) and hence a dip can be seen in the curve.

To determine throughput for this particular prototype configuration, traffic was sent to the WUGS-20 switch at a fixed rate and then recycled in the switch to generate traffic at speeds above 1Gbps. Using a single match engine, the circuit scanned data at the rates up to 600 Mbps.

6 Related Work

Jason et al. in [2] explore the benefits of using Aho-Corasick Boyer-Moore (AC_BM) algorithm to improve the performance of SNORT [13]. This algorithm is faster than the Boyer-Moore algorithm currently used by the current version of SNORT engine. Varghese and Fisk in [6] analyze set-wise implementation of Boyer-Moore algorithm which has average-case performance that is better than the Aho-Corasick algorithm. These algorithms are primarily geared toward software implementation. Commercial hardware implementations of some packet content inspection [7, 10, 11] are available, however, few details about these proprietary systems available.

Advent of the modern reconfigurable hardware technology, particularly FPGAs, has added a new dimension to hardware based packet inspection techniques. Literature [9, 5, 14] shows that new approaches using reconfigurable hardware essentially involve building an automaton for a string to be searched, generating a specialized hardware circuit using gates and flip-flops for this automaton, and then instantiating multiple such automata in the reconfigurable chip to search the streaming data in parallel. The common characteristic of these approaches is that the on-chip hardware resource consumption (gates and flip-flops) grows linearly with the number of characters to be searched. Secondly, these methods require the FPGA to be reprogrammed to add or delete individual strings from the database. Any

change in the database requires the recompilation, regeneration of the automaton, re-synthesis, re-place and route of the circuits.

In contrast, the Bloom filter-based system is able to a handle a larger database with reasonable resources, and supports fast updates to the database. The latter is an important feature in network intrusion detection system which require immediate action to certain attacks like an Internet-worm outbreak.

7 Conclusions

A technique has been presented to detect for the presence of predefined strings in packet payload at wire speeds. The technique is based on the hardware implementation of Bloom filters. Constant time computation of the algorithm along with the scalability of Bloom filters makes it an attractive choice for applications such as network intrusion detection which require real time processing. An analysis of the trade-offs between the memory resources and system performance has been presented along with the analysis tailored for the FPGA-based implementation. An FPGA-based implementation in a Xilinx Virtex 2000E FPGA on the FPX platform could support 10,000 strings. Our future work involves using multiple Bloom filter engines in parallel to handle line speeds of $2.4\ Gbps$ (OC-48) with the FPX infrastructure.

8 Acknowledgement

The authors would like to acknowledge the contributions of David Lim and Ramaprabhu Janakiraman of Washingtom University. This work is supported by a grant from Global Velocity.

References

[1] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *ACM*, 13(7):422–426, May 1970.

[2] J. Coit, S. Staniford, and J. McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of snort. In *Proceedings of DISCEX II*, June 2001.

[3] B. Dipert. Special Purpose SRAMs smooth the ride. *EDN*, Jun 1999.

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

[5] R. Fanklin, D. Caraver, and B. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings from Field Programmable Custom Computing Machines*, 2002.

[6] M. Fisk and G. Varghese. Fast content-based packet handling for intrusion detection. Technical Report CS2001-0670, University of California, San Diego, 2001.

[7] Integrated Device Technology, Inc. Pax.port, Classification and Content Inspection Co-Processors, 2002.

[8] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, Feb. 2001.

[9] J. Moscola, J. Lockwood, and R. P. Loui. Implementation of a Content-Scanning Module for an Internet Firewall. Submitted to IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Apr. 2003.

[10] NetScreen Technologies Inc. NetScreen-5000 Series, 2003.

[11] PMC Sierra Inc. PM2329 ClassiPi Network Classification Processor Datasheet, Issue 4, 2001.

[12] M. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. In *Proc. of Int. Conf. on Computing and Information*, pages 1621–1636, 1994.

[13] M. Roesch. SNORT - lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference*, 1999.

[14] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Rohnert Park, CA, USA, Apr. 2001.

[15] Thomas H. Corman, Charles E. Leiserson, Ronald D. Rivest, Clifford Stein. *Introduction to Algorithms*. Prentice Hall, 2002.