Report Number: WUCSE-2004-12

2004-03-25

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

Sarang Dharmapurikar, Michael Attig, and John Lockwood

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, of-ten 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 solutions have been introduced. In this paper we show how Bloom filters can be used... **Read complete abstract on page 2.**

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

Sarang Dharmapurikar, Michael Attig, and John Lockwood

Complete 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, of-ten 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 solutions have been introduced. In this paper we show how Bloom filters can be used effectively to perform string matching for thousands of strings at wire speed. We describe how Bloom filters can be implemented feasibly on commodity FPGA. Our analysis shows that this approach for string matching is more effective than the current FPGA-based solutions which use Deterministic or Non-deterministic Finite Automata (DFA or NFA). Fi-nally, we give the details of our implementation of string matching technique on Xilinx XCV 2000E FPGA.

2004-12

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

Authors:

March 25, 2004

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 solutions have been introduced.
In this paper we show how Bloom filters can be used effectively to perform string matching for thousands of strings at wire speed. We describe how Bloom filters can be implemented feasibly on commodity FPGA. Our analysis shows that this approach for string matching is more effective than

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

Sarang Dharmapurikar
sarang@arl.wustl.edu

Michael Attig
mea1@arl.wustl.edu

John Lockwood
lockwood@arl.wustl.edu

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 solutions have been introduced.

In this paper we show how Bloom filters can be used effectively to perform string matching for thousands of strings at wire speed. We describe how Bloom filters can be implemented feasibly on commodity FPGA. Our analysis shows that this approach for string matching is more effective than the current FPGA-based solutions which use Deterministic or Non-deterministic Finite Automata (DFA or NFA). Finally, we give the details of our implementation of string matching technique on Xilinx XCV 2000E FPGA.

1 INTRODUCTION

Network Intrusion Detection Systems (NIDS) scan the payload of the Internet packets to look for the presence of any of the predefined signature strings. Such signature strings can indicate the presence of an Internet worm or a computer virus, signify unauthorized login to a server or access to illegal websites. It is critical for NIDS to scan each and every byte of the payload and take appropriate action on the packets that contain the predefined keywords. 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. Moreover, such a NIDS should be able to process the packets at the wire speed in order to avoid becoming a network bottleneck. With the networking speeds doubling every year, it is becoming increasingly difficult for software based packet scanners 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.

Recently, FPGAs have been used to perform high speed pattern matching [9, 6, 15]. The techniques proposed in this literature implements DFA or NFA on an FPGA for the strings to be detected. Reconfigurable logic gives the ability to add and delete the strings of interest to or from the FPGA. Thus, due to a high degree of parallelism, such techniques can detect the predefined patterns at very high speeds. However, a drawback in these schemes is that the logic gate consumption is proportional to the total number of characters in the strings of interest. Hence such schemes do not scale very well with a large number of strings.

A Bloom filter offers a very attractive choice for string matching. It is a randomized technique to test membership of a string in a group of given strings. Using this technique, first a group of strings is compressed by calculating multiple hash functions over each string. This compressed set of strings is stored using a small amount of memory. This set can be queried to find out if a given strings belongs to it. The two important properties of a Bloom filter that make it a viable solution for string matching are the following:

1] Scalability: Bloom filters use a constant amount of memory to compress each string irrespective of the length of the original string. Thus, very large strings can be stored with very little memory. This makes it highly scalable in terms of memory usage.

2] Speed: The amount of computation involved in detecting a string using Bloom filters is constant. This computation in fact is calculation of hash functions and the corresponding memory lookups. Efficient hash functions can be implemented in a hardware very easily with very little resource consumption. Hence, a hardware implementation of Bloom filter can do string matching at very high speeds. Our end result shows that we can look for a set of 10,000 strings at the rate of more than 2.4 Gbps (OC-48).

As mentioned above, Bloom filters use a small amount of memory to store the compressed strings. The amount of memory depends on the number of strings being compressed and typically is a few megabits. For instance, to store 10,000 strings, around 200k bits are required. Almost all the modern FPGAs come with multi-port embedded memory blocks which can be utilized for constructing Bloom filters. However, the real reason for using FPGAs stems from the requirement of memory reconfiguration for Bloom filters. As will be clear later, we maintain one Bloom filter for detecting strings of one particular length. If the database of strings to be detected has non-uniform number of strings for each unique string-length then the Bloom filters need to be tuned to accommodate this non-uniformity and achieve the optimal performance. Moreover, since the string length distribution can change over time, Bloom filters need to be re-tuned to maintain optimality which involves reallocation of the Block memories and hash functions. While doing this, the underlying hardware needs to change. Hence, the FPGAs prove to be extremely effective in such a scenario.

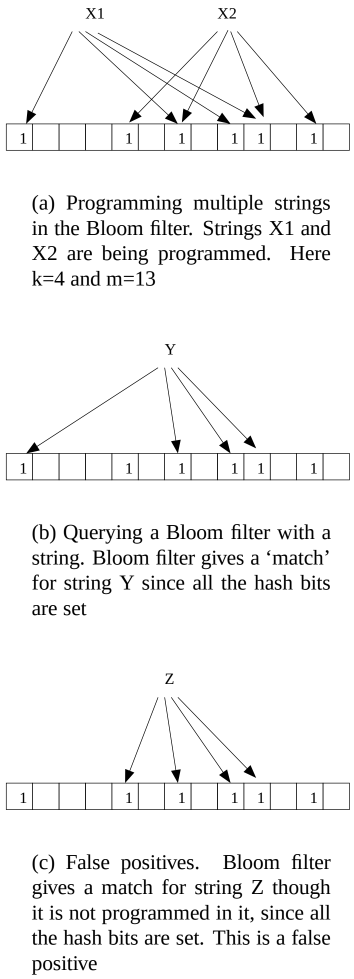
The rest of the paper is organized as follows. Section 2 presents an introduction to the Bloom filter. Section 3 gives an overview of the system. The detailed implementation of the Bloom filters in FPGA is presented in Section 4. The implementation results are discussed in 5. A brief summary of the related work is given in 6 followed by conclusions and future work in 7.

2 Theory of Bloom Filters

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 [3]. The theory behind Bloom filters is described in this section. First we describe the ordinary Bloom filter and then explain the enhancement to it, known as Counting Bloom filter.

2.1 Bloom Filters

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. Figure 1(a) illustrates this concept.



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

(b) Querying a Bloom filter with a string. Bloom filter gives a ‘match’ for string Y since all the hash bits are set

(c) 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

Figure 1. Illustration of Bloom filter

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 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 1(b) and 1(c).

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. (A further investigation will reveal that *Y* is the same as *X1*). 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]

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

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 $\frac{m}{n}$, 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 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.

2.2 Counting Bloom Filters

One property of Bloom filters is that it is not possible to delete a member stored in 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 that hash to any of these bits. In order to solve this problem, the idea of the *Counting Bloom filters* was proposed in [5]. 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

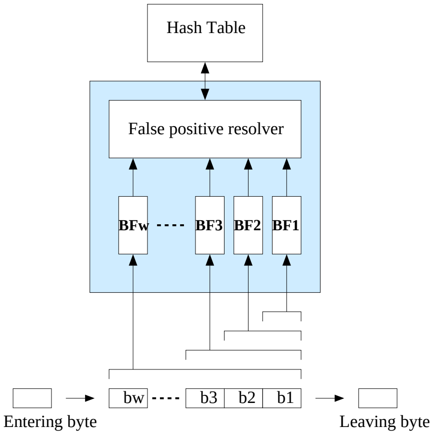


Figure 2. A single packet scanning engine consisting of multiple Bloom filters each of which detects strings of a unique length. The longest string of interest is ‘w’

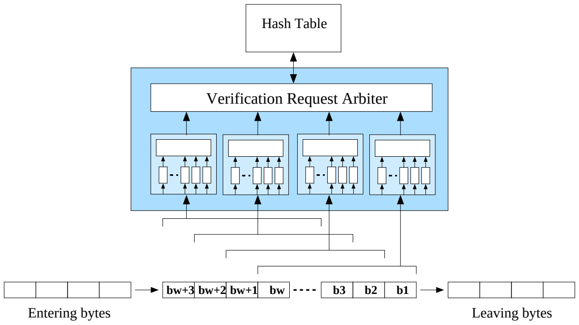


Figure 3. Multiple engines used in parallel each monitoring a window of bytes shifted by a single byte. In this example, the throughput will be 4x

We use Bloom filters to store the signature strings to be detected and query these Bloom filters using all possible strings in the streaming data to check if any of these strings is a string of interest. If the Bloom filter does not give a match then the data can be ignored since a false negative is

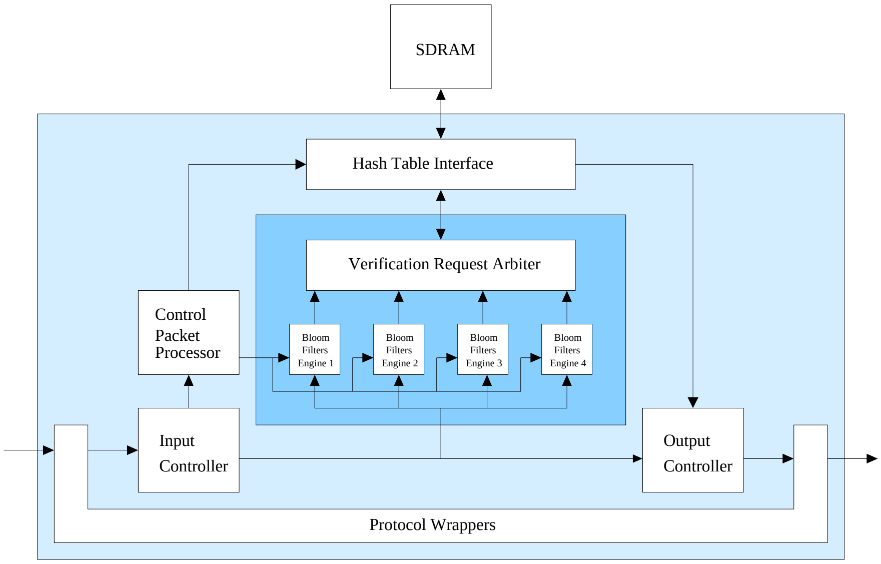


Figure 4. Block diagram of the architecture.

not possible. When the filter gives a match for the input then it can be a potential string of interest with a high probability.

To make this possible, we group the strings to be detected by their lengths and store all the strings of one particular length in one Bloom filter. Depending on how many different unique lengths exist in the database of strings of interest, we instantiate an array of parallel Bloom filters each of which stores strings of a given length and checks the strings of the same length in the streaming network data.

Figure 3 illustrates this concept. It shows that there are strings of length ranging from 1 byte to ‘w’ bytes and all the strings with the same length are stored in the corresponding Bloom filter. This set of Bloom filters monitors a window of ‘w’ bytes of network data. This window contains strings of length one to w bytes. Each of these strings is input to the corresponding Bloom filter to test for the membership. When none of the Bloom filters give a match, the data window is progressed by a single byte and the same procedure is repeated. 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 checked against a *hash table* which determines if a string is indeed a member of the set or a false positive. Thus, the hash table acts as a false positive eliminator. When a true string is found, an appropriate action (drop, forward, log) can be taken on the packet.

Since the false positive rate can be reduced to a desired small value using ample memory, the probes to the hash table due to false positives can be reduced. Therefore, the probes in the hash table will almost always be due to the true positives. Since the true positive probability is typically low, very few probes need to be performed in the hash table and hence normally the data window is progressed every clock cycle.

Clearly, with only one set of Bloom filters, the data stream can progress by only one byte per clock cycle. In order to get more throughput, multiple such engines need to be deployed in parallel. This is depicted in Figure 3 where four identical engines containing multiple Bloom filters are used. Each of these engines scans a data window with an offset of one byte. Hence all the strings are scanned and the window can be advanced by four bytes in a single clock cycle giving a throughput 4x that of the single engine. Since all the Bloom filters in all the engines share the access to the same hash table, one more level of arbitration needs to be added above the four engines. In case of multiple matches within the same window, the arbitration logic will start the hash table probe with the longest matching string from the first engine and end with the shortest matching string in the last engine.

Figure 3 shows the detailed block diagram of the system as implemented on the FPX. The main components of the system are Bloom filter engines and a hash table. The architecture of these blocks is explained in the next section. Packets on the link are parsed by the protocol wrappers [2] and the application layer data is presented to the scanner module. The Control Packet Processor is responsible for decoding the control packets and making updates to the hash table as well as Bloom filter. Whenever a string is added or deleted from the database, these updates are done. These updates essentially involve setting or resetting some bits in the Bloom filter and storing or deleting the strings from the hash table.

The Input Controller controls the flow of input byte stream to the Bloom filter. It can receive 4 bytes of an Internet packet per clock cycle. Hence four parallel Bloom filter engines are used to maintain the throughput. This data is buffered in a 2 KB FIFO. When the Bloom filters can accept data, the Input Controller streams bytes through a series of flip-flops, composing the *window* that the Bloom filters monitor. Whenever Bloom filter finds a matching string in the streaming data, it asks the Input Controller to pause the data stream and dispatches a request to the hash table to verify the presence of the matching string. Once the hash table gives a result, the Bloom filter asks the Input Controller to resume the data stream.

The Output Controller reassembles the input stream. Once the packet is received completely, it outputs the packet to the protocol wrappers. However, when the hash table finds a matching string, it instructs the output controller to output a warning packet instead of original packet. In that case the original packet is dropped.

4 Implementation Details

In order to reduce the false positive probability considerably, a long bit vector is required. Even for supporting the programmability for hundreds of strings, many thousand bits are required. The cost of using the on-chip flip-flops for this purpose is too high. However, modern FPGAs have on-chip RAMs with more than one port (typically dual port) that can be exploited to create the Bloom Filter vector. For instance, the Xilinx XCV 2000E chip on the FPX board has 160 on-chip RAMs each with two ports. Each Block RAM can be configured as a single bit wide and 4096 bits long vector, giving a bit lookup throughput of 2 bits per clock cycle. We now describe how these on-chip RAMs can be used for building the basic Bloom Filter.

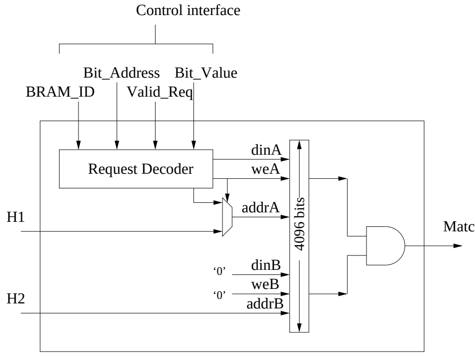


Figure 5. A *Partial* Bloom filter (PBF) with $k = 2$ hash functions and $m = 4096$ bits. This filter can accommodate $n = 1419$ strings with false positive probability $(1/2)^2$.

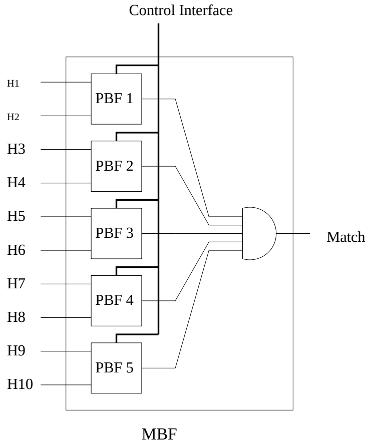


Figure 6. A *mini* Bloom filter (MBF) constructed from five PBFs. This can support, 1419 strings with a false positive probability $(1/2)^{10}$.

4.1 Implementing Bloom Filter

Figure 4.1 shows how a Block RAM can be used to construct a Bloom filter. The Block RAM is configured as a single bit wide and 4096 bits long bit array. It has two read/write ports, both of which can be used to lookup the bit values corresponding to two distinct hash functions. Thus, this Bloom filter can support $k = 2$ hash functions, $m = 4096$ bits in the array and hence can support $n = (m/k)ln2 = 1419$ strings in it. Apart from doing the lookups corresponding to the hash values, a control interface is needed through which the bits in the Bloom filter can be set or reset whenever a string is added or deleted. Hence one of the two ports of the Block RAM is shared between the hash lookup interface and control interface as shown in the figure. When the control interface is not in the use, the port is used for doing the lookup corresponding to the hash value. The control interface of this basic Bloom filter consists of four inputs: *BlockRAM_ID*, *Bit_Address*, *Bit_Value* and *Valid_Req*. *Valid_Req* is simply a qualifier bit, i.e. all the other control inputs are valid only if this input is high. The $BRAM\_ID$ input specifies which Block RAM is chosen and $Bit\_Address$ specifies which bit to program in that Block RAM. Each Block RAM in the system is assigned an ID. If the $BRAM\_ID$ input matches with the ID, the bit specified by the $Bit\_Address$ input is programmed to the value specified by the Bit_Value input. Since the other port is used only for lookup of the hash value and never for writing to the bits, the write enable input (weB) is always disabled. Likewise, the data input (dinB) too is reset permanently.

We refer to this Bloom filter as a *Partial Bloom filter (PBF)* since multiple such Bloom filters are needed to create

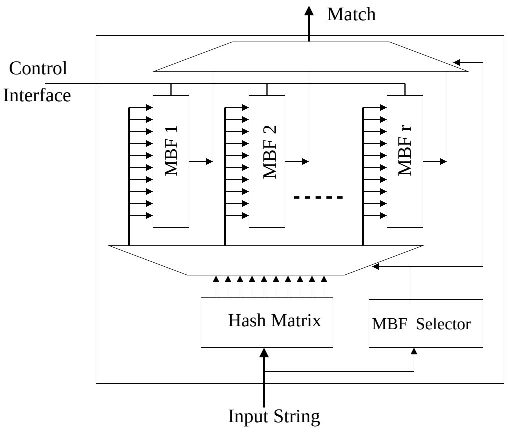


Figure 7. A large Bloom filter (LBF) constructed by using multiple mini-Bloom filters (MBF). The signatures are distributed uniformly randomly among all the MBFs. With r MBFs, $r \times 1419$ strings can be stored in this LBF

a Bloom filter with smaller false positive probability.

By using multiple PBF to store the same set of strings, the false positive probability can be reduced. For instance, if two PBFs are used then the false positive probability will be $((1/2)^2)^2 = (1/2)^4$, if three PBFs are used then the false positive probability will be $((1/2)^2)^3 = (1/2)^6$ and so on. For this work, five PBFs are used to achieve a false positive probability of $(1/2)^{10} = 0.00097$ and a storage capacity of 1419 strings. This set of five PBFs is called a *mini-Bloom filter (MBF)*. The PBFs in a MBF receive the same control signals coming from a controller. This MBF is shown in Figure 4.1. Note that the hash function matrix is not shown in the figure yet.

A MBF can support only a small number of strings. To create a Bloom filter with a bigger capacity, multiple MBFs are used in parallel. This set of MBFs is called *Large Bloom Filter (LBF)*. Figure 4.1 shows the schematic of the LBF constructed using r MBFs. This LBF has a capacity of $r \times 1419$. A string is stored in only one of the MBFs. The MBF for storing a particular string is chosen randomly by calculating a hash value over the input string. During the query process, the same hash value is will be calculated and hence the same MBF will be probed for the presence of the string. For probing a particular MBF, k hash values are calculated and they are routed through a MUX to the MBF chosen for probing. The output of the same MBF is routed out.

4.2 Hash Functions

A class of universal hash functions described in [12] were found to be suitable for hardware implementation. It should be recalled that k hash functions are generated. Following is the description of how this hash matrix is calculated. For any bit string X with b bits represented as

$$X =< x_1, x_2, x_3, ..., x_b >$$

the i^{th} hash function over X, $h_i(X)$ is calculated as,

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

where '.' is a bitwise AND operator and '$\oplus$' is a bitwise XOR operator. d_{ij} is a predetermined random number in the range [0…m-1]. *Note that the hash value can be out of the range [0…m-1] if m is not a power of 2. Hence m must be a power of 2. For this implementation m= 4096, which is a power of 2.* It can be observed that the hash functions are calculated cumulatively and hence the results calculated over the first i bits can be used for calculating the hash function over the first $i+1$ bits. This property of the hash functions results in a regular and less resource consuming hash function matrix.

4.3 Hash Table

A hash table is used to resolve false positives from Bloom filters. Two versions of the hash table have been implemented. The first uses Synchronous Dynamic Random Access Memory(SDRAM), and the second uses Synchronous Random Access Memory (SRAM). More strings may be stored in the SDRAM version. However, SDRAM has more latency for a hash probe than SRAM. With 64 Megabytes commodity SDRAM there are $2^{20} = 1048576$ slots to store 32 byte signatures. With merely 10,000 signatures stored randomly in these slots, the occupancy of the hash table is 1/100, making the hash collision probability negligible. In our current implementation, the latency in probing a hash slot in the SDRAM (with a 8 byte wide data bus) based hash table is 20 clock cycles. Likewise, using a 2 Megabytes commodity ZBT SRAM (with a 4 byte wide data bus), there are $2^{16} = 65536$ slots to store 32 Byte signatures with a latency of 14 clock cycles.

A block diagram of the hash table is shown in Figure 4.3. The hash value calculator is similar to that used for each hash function for the Bloom filters. In order to resolve hash collisions, quadratic probing is implemented. Two extra bits make this possible. An *occupied bit* is maintained to signify that an entry exists at a specific location. A *deleted bit* is maintained to specify that an entry existed at one time in this location, but it has since been deleted. This allows the search to continue in case an entry has been deleted.

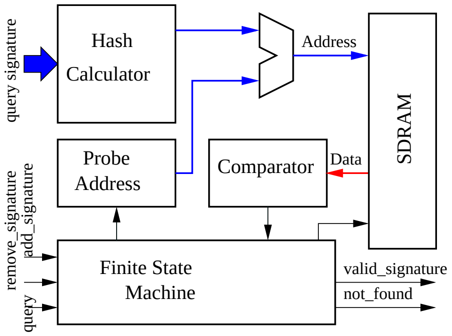


Figure 8. The hash table consists of a hash calculator, a finite state machine, and a comparator. The hash table resolves collisions by performing quadratic probing.

When a signature is added, the hash address is calculated. This address is checked for occupancy. If an entry does not exist at that location (the *occupied bit* is cleared) or the *deleted bit* has been set, the new string is added to the address. If an entry exists at that location (the *occupied bit* is set and the signature stored does not match the query signature), the next location will be checked. The next address is calculated by adding an offset equal to a power of 2 in the base address. Thus, the locations to be checked upon collisions will have offsets of $1, 2, 4, 8, 16, \ldots$. This is continued until an empty slot is found. As can be seen, offset calculation is easily implemented in hardware via a left shift addition term.

When a signature is queried, the same process is repeated, stopping when a signature match occurs or the *occupied bit* and *deleted bit* are found to both be cleared (indicating that the string was not in the hash table). Deleting a signature consists of performing a query, followed by setting the *deleted bit* if the signature was found.

5 Implementation Results

A single Bloom filter engine and the rest of the components as shown in the Figure 3 have been implemented and tested on the FPX [] board. The system was designed to easily change the number of MBFs and PBFs to accommodate the particular database of signatures. In addition, different length Bloom filters may also be instantiated. This allowed several different systems to be built for testing purposes. All of the following systems were implemented in the Xilinx Virtex 2000E FPGA and tested in the test environment to be described.

Component	LUTs	Flip Flops	Block RAMs
Single 10-byte Bloom Filter	1495 (3%)	1297	5
Single 32-byte Bloom Filter	4438 (11%)	3372	5
24 to 32 byte Bloom Filters	? (?%)	?	?
CPP	144 (0%)	315	0
Hash Table	1540 (4%)	1484	0
Input Processor	34 (0%)	81	5
Output Processor	230 (0%)	251	2
Protocol Wrappers	3487 (7%)	2694	22

Table 1. A summary of the resources consumed by each component in a single engine system. Three variations were implemented. First with a single 10-byte Bloom filter. Second with a single 32-byte Bloom filter and third with multiple Bloom filters for 24 to 32 bytes signatures inclusive

5.1 Resource Consumption and Throughput

A single engine consisting of just one Bloom filter which scans strings of only one length was built. Such a system is useful for copyright protection applications where the files to be protected can be characterized by a randomly chosen fixed length string.

Two separate single length engines were built to observe the LUT resource consumption due to hash functions. The first system monitored a 10-byte window. The second system used a 32-byte window instead. The Bloom filter in both the systems contained five PBFs allowing 1419 10-byte signatures to be stored.

The 10-byte system operates at 73.513 MHz, corresponding to a throughput of 588 Megabits per second. and the 32-byte system operates at 66.353 MHz, corresponding to a throughput of 531 Megabits per second. A summary of the resource usage by component is given in Table 5.1. The increase in LUT consumption and slice utilization are due to the 176 new bits that are included in hash function calculation and the need to extend the byte-sized flip-flops in the byte stream.

A system has been built and tested that searches for signatures of lengths 24 bytes to 32 bytes, inclusive. Each Bloom filter in this engine had five PBFs. Since each Bloom filter can support 1419 signatures, the collective capacity of this engine is 12771 signatures. This circuit operates at 64.75 MHz, corresponding to a throughput of 518 Megabits per second. With a larger FPGA, this system could use a

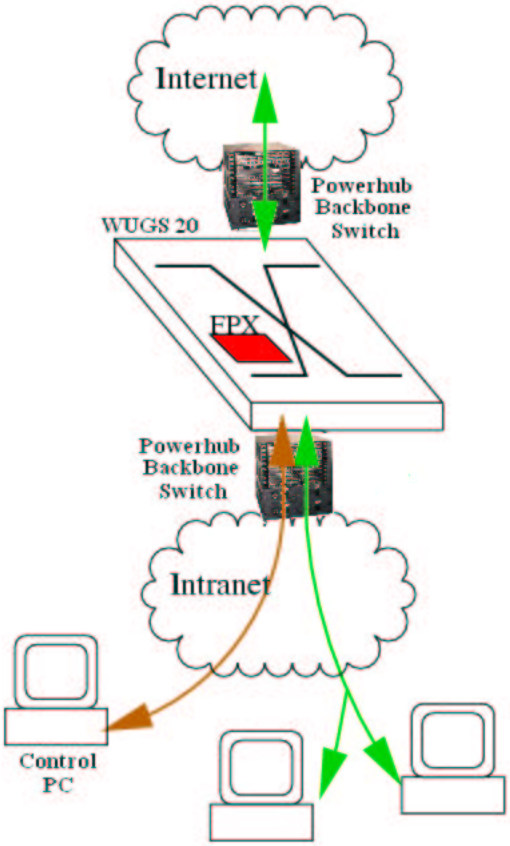


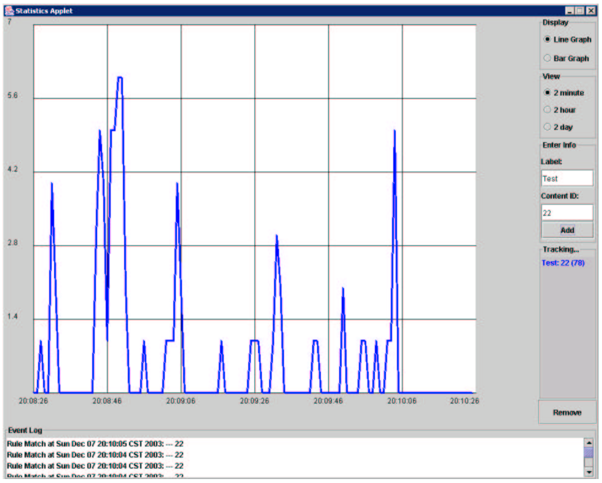
Figure 9. Laboratory test configuration.

quad-engine to improve the throughput to 2.12 Gigabits per second.

5.2 Experiments and Results

The three systems described above were implemented and tested in the lab configuration shown in Figure 5.2. First, the FPGA on the FPX board is programmed with the Bloom filter circuit. Next, a control PC writes UDP control packets to set bits in the Bloom filter(s) and add the specific signature to the hash table. These UDP packets travel from the control PC through the local Intranet to a Powerhub multilayer backbone switch. The Powerhub transforms the UDP packets to ATM cells, which transverse through the WUGS 20 ATM switch. The ATM cells enter the Protocol Wrappers, are decoded as UDP packets, and are fed to Control Packet Processor for programming the Bloom filter(s). Signatures are added and deleted via these control packets at any time.

The lab configuration was set up such that the Bloom filter circuit was placed on the border of the local Intranet and the Internet. In this way, Internet traffic coming from and going to PCs on the local network went through the Bloom



Display
● Line Graph
○ Bar Graph

View
● 2 minute
○ 2 hour
○ 2 day

Enter Info
Label: Test
Content ID: 22

Tracking...
Test: 22 (78)

Event Log

```
Rule Match at Sun Dec 07 20:10:05 CST 2003: --- 22
Rule Match at Sun Dec 07 20:10:04 CST 2003: --- 22
Rule Match at Sun Dec 07 20:10:04 CST 2003: --- 22
Rule Match at Sun Dec 07 20:10:04 CST 2003: --- 22
```

Figure 10. A alert message graphing application which plots the number of occurrences per second of a specified signature.

filter circuit. Several test signatures, associated with known signatures on web pages and in files, were programmed into the circuit. These web pages were viewed and files were accessed from the laboratory computers. Upon detection of these signatures, the Bloom filter circuit created an alert message. The alert message was sent to the control PC, informing it that a specified signature was found. For example, in the 10-byte system, the 10-byte signature of ???? was programmed from the SoBigF virus that recently hit. This signature was programmed into the Bloom filter, scanned for, and detected. As another example, the signature “Double, double toil and trouble” from Macbeth was programmed into the Bloom filter system that scans 24 to 32 byte windows. One of the laboratory PCs then went to download Macbeth from a web page. Three alert messages were received by the control PC, since this signature occurs three times in Macbeth. Alert messages received at the control PC can be sent to a simple graphing application, as can be seen in Figure 5.2. This application plots the number of occurrences per second of a specified signature.

6 Related Work

Jason et al. in [4] 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 [7] 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 [8, 10, 11] are available, however, few details about these proprietary systems was available.

Advent of the modern reconfigurable hardware technology, particularly FPGAs, has added a new dimension to hardware based packet inspection techniques. Literature [9, 6, 15] 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 Conclusion

An FPGA-based technique for detecting predefined signature strings in packet payload at wire speed has been presented. This technique uses Bloom filter which is a randomized algorithm for storing and matching strings. Since Bloom filters consume a small amount of memory and can perform string matching with a constant amount of computation, it can be used effectively for real-time deep packet inspection. Bloom filters can be implemented easily in FPGAs using the embedded memory blocks. For optimal performance, the memory blocks need to be allocated to Bloom filters in proportion to the number of strings they contain. Since this string distribution can change over time, block memories need to be reallocated. Hence FPGA is an attractive choice for the implementation of Bloom filters. An implementation in a Xilinx Virtex 2000E FPGA on the FPX platform can support packet scanning at 2.4 Gbps for 10,000 strings.

The components of this system can be used to implement a complete Snort-like NIDS on a FPGA. Since Snort rules involve header rules and payload rules, we need to integrate a scalable, hardware-based technique for header rule matching with this payload scanner. These two coupled with the TCP/IP processor developed in Washington University [14] can give the ability to do *stateful* packet inspection, in which rule matching can be done on per-flow basis in addition to per-packet basis.

References

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

[2] F. Braun, J. Lockwood, and M. Waldvogel. Reconfigurable router modules using network protocol wrappers. In *Proceedings of Field-Programmable Logic and Applications*, pages 254–263, Belfast, Northern Ireland, Aug. 2001.

[3] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey.

[4] 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.

[5] 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.

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

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

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

[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] D. V. Schuehler, J. Moscola, and J. W. Lockwood. Architecture for a hardware based, tcp/ip content scanning system. In *Hot Interconnects*, pages 89–94, Stanford, CA, Aug. 2003.

[15] 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.