High Performance Pattern Matching using Bloom–Bloomier Filter

Nguyen Duy Anh Tuan, Bui Trung Hieu, Tran Ngoc Thinh
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology
268 Ly Thuong Kiet, Ho Chi Minh City, Vietnam

***Abstract*-In this paper, we propose a high performance architecture based on the combination of Bloom Filter and Bloomier Filter (BBF) to enhance the speed of pattern matching process on Clam Antivirus (ClamAV) database. BBF maintains small on-chip memory, low number of fault positives and can indicate which patterns are the candidate matches. The implementation results on low-cost Altera Cyclone II show that our architecture can handle 43,491-characters of ClamAV pattern set with only 9.5 bits per character and achieve a throughput of 1 gigabit per second (Gbps). As compared with previous systems, our memory utilization is far better up to 73%.**

I. INTRODUCTION

Nowadays, along with the development of internet, information security is becoming more and more critical. One of the most important aspects in this field is antivirus. Although there are many improvements in antivirus programs, they still have to match a file stream with static patterns of known viruses. This process occupies a noticeable amount of resource and slows down entire system due to the growing number of viruses. In addition, software-based solution can not catch up with the gigabit networks.

This limitation leads to a demand of hardware solution to speed up this process. FPGA-based system is one of popular hardware technologies because of its high speed operation and flexibility in changing application. There are many FPGA-based solutions have been developed in this field, but those systems [2, 3, 4, 5] need quite large on-chip memory to operate (14.1 – 34.6 bits per character) or consume lots of logic elements [6], up to 1.12 logic elements per character. Our Bloom-Bloomier Filter (BBF) is a flexible, storage-efficient FPGA-based system that boosts the speed of pattern matching. The BBF relies on hash-based Bloom Filter and Bloomier Filter algorithm, thus the usage of logic element (currently 0.27 logic elements per character) is independent of the quantity of implemented patterns. We use up to ten hash functions to decrease the false positive probability as well as the off-chip memory access rate. The on-chip memory density of BBF is only 9.5 bits per character not as much as previous systems of which on-chip memory density are 14.1 to 34.6 bits per character.

This paper is organized into 5 sections. ClamAV Database, Bloom Filter and Bloomier Filter are introduced in Section 2. Section 3 describes the combination of Bloom and Bloomier Filter as well as the architecture of BBF system. Synthesis and simulation results are presented in Section 4. Finally, conclusions and future work are given in Section 5.

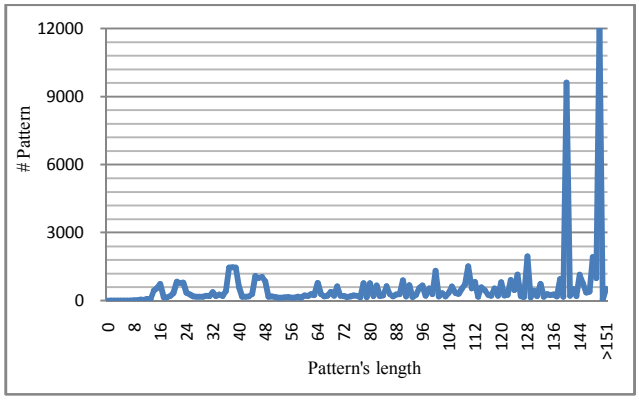
II. BACKGROUND

A. Clam Antivirus Database

Clam Antivirus (ClamAV) [1], acquired by Sourcefire, is one of the most popular open-source antivirus applications. It is mainly used in mail server of mid-size organizations to detect malwares. We use ClamAV virus database for our recognized pattern set. The database updated on 14 May 2009 has a total of 545,035 patterns and there are 3 types of pattern: regular expression, simple matching and MD5. 83.5% of those patterns are MD5 and regular expression patterns. Ref. [7] stated that the most consuming time in virus scanning process is simple pattern matching task (73.4% of scan time), so we only concentrate on improving this task. There are 89,904 simple matching patterns, the lengths of those patterns are distributed as in Fig. 1.a. Due to the limitation of memory capacity of FPGA Cyclone II chip, we only carry out patterns of which length ranges from 10 to 20 characters. The number of pattern in each length is shown in Fig. 1.b.

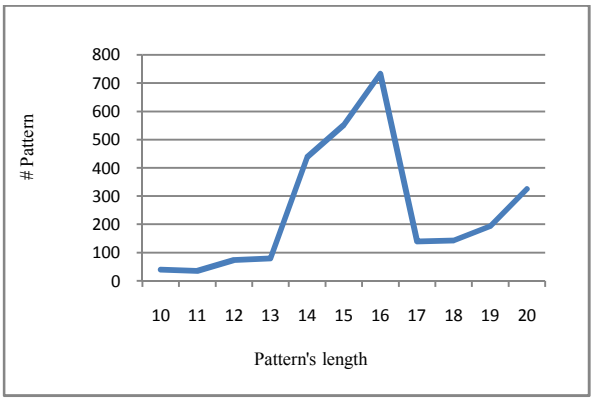
B. Bloom Filter

The basic idea of Bloom Filter [8] is to use an index table to check the existence of a given string in a pre-defined set. Initially, all entries in the 1-bit-array index table are set to '0'. Each member of pre-defined set is then hashed to k positions in the index table by k hash functions, entries corresponding with those positions are set to '1'. This process is repeated until all members of pre-defined set are hashed to index table.



a)

Figure 1. a). All pattern's length distribution



b)

b). There are 2751 patterns which are 10-to-20 character long

Membership of a string is checked in similar method. At first, that string is fed to k hash functions above to get k entries in the index table. If one of these entries is '0', this string is not member of the set, otherwise, the existence of this string in the set is uncertain and further check is required. This uncertainty is caused by "false positive" problem in hash-based system. Probability of false positive depends on number of hash functions (k), size of set (n) and length of index table (m). Equation (1) is used to tune those variables to get desired false positive probability.

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

C. Bloomier Filter

In Bloom Filter's worst case, the entire pre-defined set is scanned to confirm the uncertain-match result from hashing operation. [9] introduces better approach by using secondary hash function and index table to reduce scope of scanning but it still has to scan more than one pattern. This task consumes lots of time because the patterns are usually stored in low speed off-chip memory.

Bloomier Filter [10] is developed to solve this weakness. It can show exactly which pattern in the set is the best match with the searched string so the query time is constant. Bloomier Filter's algorithm is similar to Bloom Filter but its index table is constructed in a different method. Instead of using one bit for each index table entry, Bloomier Filter stores more information in one entry, as a result, size of each entry depends on which information is encoded. Because of this extra information, Bloomier's index table is built in a more complex way as compared with Bloom Filter [11].

Equation (2) describes how to decode the encoded information. Just as Bloom Filter, the searched string is hashed by k hash functions resulted in k positions in the index table. Then, all values of these positions are XORed together, at this stage, the information is recovered.

$$INFO = \bigwedge_{i=1}^{k} Data(Hash_i(String)) \quad (2)$$

INFO: the actual information of Bloom – Bloomier Filter
$Hash_i(String)$: hash result of String
$Data(Hash_i(String))$: data of entry in the index table at position $Hash_i(String)$
$\bigwedge_{i=1}^{k}$ *: denote* $n-ary\ XOR$ *operation*

III. SYSTEM ARCHITECTURE

A. Overview

Our BBF system in Fig. 2 includes a Character Scanning Unit, a Comparison Unit and an Off-chip Memory to store original patterns. There are 3 main components in Character Scanning Unit: Standalone Hash Unit (SHU), Bloom – Bloomier Unit (BBU) and On-chip Memory to store suspected strings. If one of BBUs signals a match, the address of correlated original pattern and current scanning string (suspected string) are passed to Comparison Unit to determine whether that string is identical with original pattern. When the exact match is confirmed, our system reports this match together with ID of the pattern.

To improve throughput, we organize SHUs and BBUs in parallel and pipeline structure as shown in Fig. 3. Each SHU or BBU uses previous hash results from their preceding neighbor and input character to calculate its own hash. This structure allows us to scan multiple-length substrings at the same time [12].

B. Character Scanning Unit

a. Bloom – Bloomier Filter

The advantage of Bloomier Filter over Bloom Filter in latter stage of searching (compare the searched string with original pattern) is also the biggest disadvantage when implemented in our system. We have to access off-chip memory to do comparison for every searched string. At this point, Bloomier Filter is

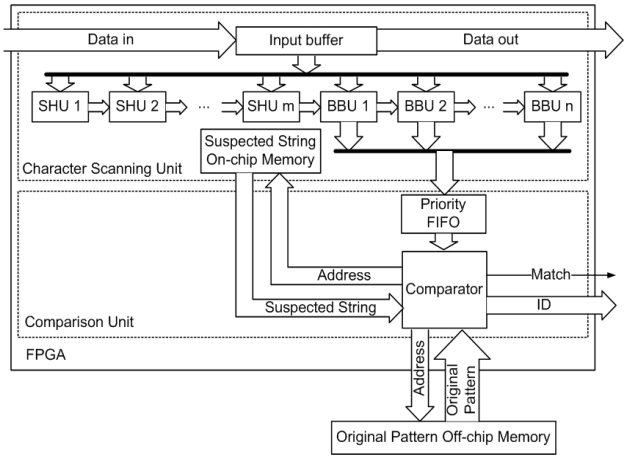


Figure 2. BBF System architecture overview

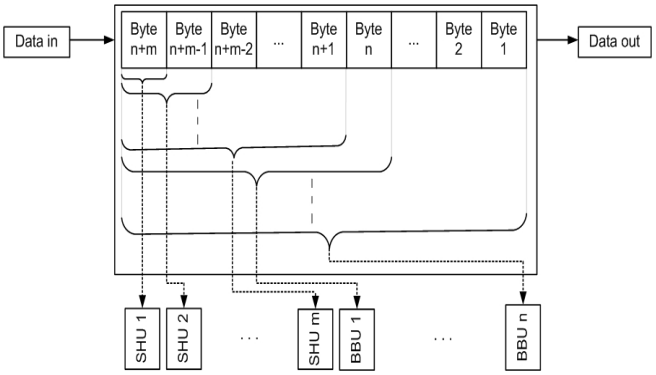


Figure 3. Parallel Character Scanning Units and Stream Window. The Stream Window, which is a shift register, stores current (n+m) scanning characters.

worse than Bloom Filter which just performs comparison when all of bits in hashed-positions are '1'.

Our solution is to combine Bloomier Filter with Bloom Filter in order to minimize the comparison process which requires accessing to low-speed off-chip memory. We use one more bit in the index table entry, called "Bloom bit" besides the Bloomier encoded bits. This Bloom bit is not part of information we want to encode in the entry as original Bloomier Filter. All of the Bloom bits in index table act as an independent Bloom Filter. We use this Bloom Filter to check whether the query string may be in the set. If all of Bloom bits in hashed-positions are '1', we decode the information from Bloomier encoded bits then start comparing the string.

Another problem with this structure is the limitation of SRAM-based FPGA chips. k hash functions require k random lookups to the memory (index table) in a single clock cycle, whereas SRAM-based FPGA only supports 2 queries at a time. To solve this problem, we break Bloomier bits into (k/2) parts, encode them into (k/2) separated index tables. Hence k hash functions are used, each pair of them corresponds to one index table. Example 1 demonstrates the operation of Bloom – Bloomier Filter.

Example 1:

• *Stage 1*: examine the Bloom Filter's results from (k/2) index tables.

Given k = 4, we have 2 index tables. After hashing the searched string, we get 4 entries in 2 index tables: A, B from index table 1 and C, D from index table 2.

Suppose: A = 1101 → bloom bit is: 1
B = 1010 → bloom bit is: 1
C = 1100 → bloom bit is: 1
D = 1001 → bloom bit is: 1

So the searched string maybe in set, go to stage 2.

• *Stage 2*: calculate (k/2) information parts.

part_1 = Bloomier(A) ^ Bloomier(B) = 101 ^ 010 = 111

part_2 = Bloomier(C) ^ Bloomier(D) = 100 ^ 001 = 101

• *Stage 3*: concatenate these parts together to form actual information.

Information = {part_1, part_2} = 111_101

b. Hash Function Consideration

Due to parallel and pipeline structure of Character Scanning Unit, we choose Shift-add-xor (SAX) as system hash function [3]. Additionally, the BBF will compare the searched string with the original pattern from off-chip memory when there is a match signal from BBUs, thus the encoded information in index tables is the address of corresponding pattern in off-chip memory.

The number of hash functions used in BBF has major impact on system performance because it affects the false positive rate of filter. High false positive rate means there will be more suspected strings need to be checked, the overflow possibility of BBU-FIFOs will also increase, when this happens, the system will terminate and wait for the Comparison Unit to read out BBU – FIFOs. The diagram in Fig. 4 shows the number of hash functions and its correlative false positive rate. Base on the quantity of implemented patterns and the Cyclone II FPGA

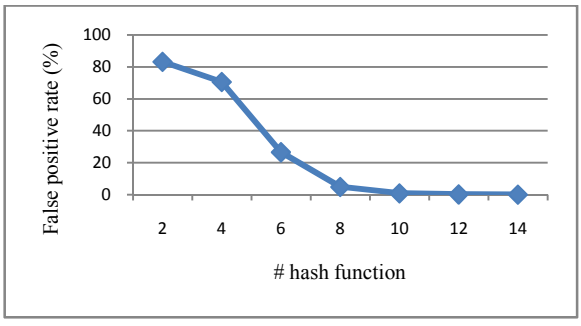


Figure 4. The number of hash functions and its correlative false positive rate. False positive rate drops dramatically with the increasing of hash functions: from 83% (2 hash functions) to 0.02% (14 hash functions)

chip's resources, we employ 10 hash functions for the filter. Hence, there are 5 index tables, each index table is configured to be 4-bit wide, which are 1 Bloom bit and 3 Bloomier bits.

c. Bloom – Bloomier Unit Structure

Fig. 5 is the structure of our Bloom – Bloomier Unit (BBU). Hash Modules described in Fig. 6 performs hashing and index table lookup operation, feeds result to Address Decoder to determine whether a match may occur. The Address Decoder will examine the "Bloom bit" in all index entries, if all are "1," current streaming window data will be saved to on-chip memory and the encoded address will be put into BBU-FIFO. This information is used by Comparison Module to check the identity of the searched string with original pattern. The differences between SHU and BBU are the omitted index table and Address Decoder. SHU simply calculates its hash then feeds this result to the next SHU or BBU.

C. Comparison Unit

Our Comparison Unit is composed of 2 parts: the Priority FIFO Module and Comparator Module. We connect all output data of BBU-FIFOs to a common bus and selectively fetch BBU-FIFOs' content to Priority FIFO. We follow the longest-match-first policy to choose how BBU-FIFOs will be loaded. As the result, the Priority FIFO module should load the BBU-FIFOs in a round-trip longest to shortest substring order.

Comparator Module reads data from Priority FIFO, uses this information to compare original pattern from off-chip memory and corresponding suspected string from on-chip memory. Whenever an exact match is detected, the comparator will report the pattern's ID, terminate the system till the next stream arrives.

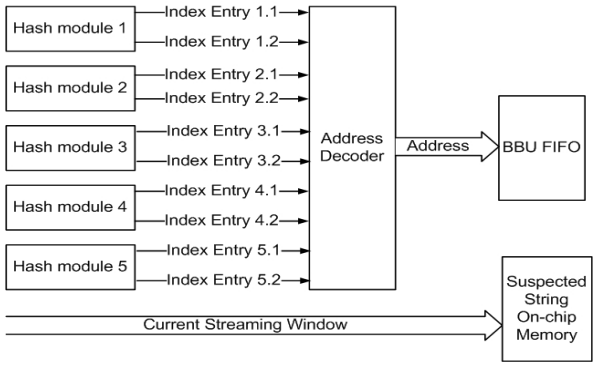


Figure 5. Bloom – Bloomier Unit (BBU). It consists of 5 Hash Modules, an Address Decoder and a BBU-FIFO.

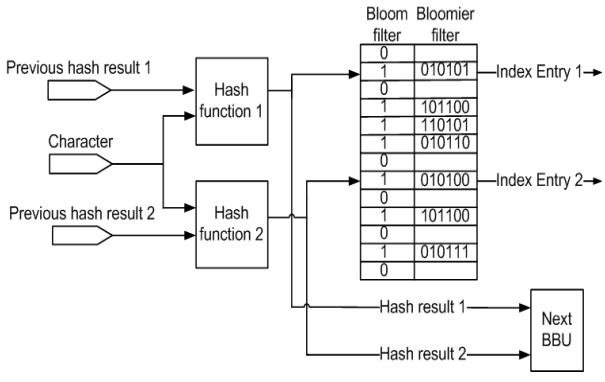


Figure 6. Hash Module in BBU. It uses previous hash results from preceding Hash Module and current character.

D. Database update

Our system mostly relies on memory: on-chip memory stores index tables and off-chip memory stores original patterns content, thus it can be updated quite easily without having to reconfigure entire system. To remove a pattern out of database, we simply change the value of pointer in off-chip memory to *null*, when the Comparator notices this invalid value, it drops current suspected string, proceeds to examine the next string. Adding new patterns is not easy as removing. The software running on PC has to re-construct the index table then transmit new index table's value via Communication Module in BBF system to replace old index table.

IV. FPGA IMPLEMENTATION RESULTS

Our system is implemented on the Altera DE2 Development and Education board. The FPGA chip in DE2 is Cyclone II EP2C35F672C6. We use Quartus II 9.1 Web Edition for hardware synthesis, mapping, placing and routing.

As mentioned above, we only implement the patterns of lengths from 10 to 20 characters. There are 2751 patterns with total of 43,951 characters in this range. Because of the limitation of low-cost FPGA chip and of synthesis tool, our system can only operate at 128 MHz (1Gbps). Table 1 is the list of hardware consumption for all components in the system which consists of 9 SHUs, 11 BBUs with their index tables and FIFO, Comparator Module, Priority FIFO and on-chip memory to save suspected strings. Our system also uses 54.5 kilobytes available off-chip memory on DE2 board to store all original patterns. Therefore, our system does not require many hardware resources and fit easily in low-cost Altera Cyclone II FPGA chip. Table 2 shows the comparison of our system with previous systems in on-chip memory usage.

V. CONCLUSIONS AND FUTURE WORK

Our system is an effective solution to accelerate the performance of pattern matching in ClamAV. The first novel aspect

TABLE 1
LOGIC AND MEMORY COST ON CYCLONE II EP2C35F672C6

Component	*Quantity*	*Block RAMs M4K*	*Logic Elements*	*Note*
SHU	9	0	4,140	Standalone Hash Unit
BBU	11	0	5,401	Bloom – Bloomier Unit
Index table	55	85	0	Each BBU has 5 index tables
BBU-FIFO	11	11	2,244	Each BBU has 1 FIFO
Priority FIFO	1	1	204	Implemented in Comparison Unit
On-chip memory	1	5	0	Stores suspected strings
Comparator Module	1	0	507	
Total		**102**	**11,989**	

TABLE 2
ON-CHIP MEMORY DENSITY COMPARISON

System	*Number of Chars*	*Memory (Kbits)*	*Bits per Char*
Our system	**43,951**	**417**	**9.5**
PH-Mem [2]	20,911	288	14.1
Cuckoo Hashing [3]	68,266	1,116	16.7
B-FSM [4]	25,200	656	26.4
HashMem [5]	18,636	630	34.6

of this system is the combination of Bloom Filter and Bloomier Filter to minimize the off-chip memory access times for exact pattern comparison. Another enhancement is the usage of ten SAX hash functions to reduce the false positive probability.

In near future, our system will support all ClamAV simple patterns and some kinds of wildcard. We also intend to create a system called Hybrid Antivirus Solution, which is a combination of hardware and software to take full advantage of high-speed FPGA-based system as well as flexibility of PC application. An anti-virus application running on PC uses some heuristic algorithms to discover unknown viruses while FPGA-based system scans the file stream in order to detect known viruses by their signatures.

REFERENCES

[1] ClamAV official website, "http://www.clamav.net."

[2] I. Sourdis, D. Pnevmatikatos, S. Wong and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," Proc. International Conference FPL, 2005, 644-647.

[3] T. N. Thinh, S. Kittitornkun and S. Tomiyama, "Applying cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," ICFPT 2007, 2007, 121-128.

[4] J. van Lunteren, "High-performance pattern-matching for intrusion detection," IEEE Int'l. Conf. on Comp. Comm., 2006, 1-13.

[5] G. Papadopoulos and D. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching," Proc. International Conference FPL, 2005.

[6] Tran Ngoc Thinh, Surin Kittitornkun, "Systolic Array for String Matching in NIDS," 4th IASTED Asian Conference Communication System and Networks, April 2-4, 2007.

[7] Xin Zhou, Bo Xu, Yaxuan Qi and Jun Li, "MRSI: A Fast Pattern Matching Algorithm for Anti-Virus Applications," Seventh International Conference on Networking, pp.256-261.

[8] B.Bloom, "Space/Time Tradeoffs in Hash Coding with Allowance Errors," Comm., ACM, vol. 13, no. 7, May 1970, p.422-426.

[9] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, John Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," ACM SIGCOMM, vol. 35, no. 4, October 2005, p.181-192.

[10] Bernard Chazelle, Joe Kilian, Ronitt Robinfeld, Ayellet Tal, "The Bloomier Filter: an Efficient Data Structure for Static Support Lookup Table," Society for Industrial and Applied Mathematics, 2004, p.30-39.

[11] J. Hasan, S. Cadambi, V. Jakkula and S. Chakradhar, "Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture," 33rd International Symposium on Computer Architecture, p.203-215.

[12] S. Dharmapurikar, P. Krishnamurthy, T. Sproull and J. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," Micro IEEE, vol. 24, no. 1, January 2004, p.52-61.