An Intrusion Detection System for Embedded Network Security Based on Pattern Matching

Shinumol B. M.
PG student, VLSI and Embedded Systems
Dept. of Electronics and Communication
TKM Institute Of Technology
Kollam, Kerala, India
sshinu007@gmail.com

Rafeek S.
Asst. Professor
Dept. of Electronics and Communication
TKM Institute Of Technology
Kollam, Kerala, India
rafeekcep@gmail.com

***Abstract*— Network security has always been an important issue. To ensure a secure network environment, firewalls were first introduced to block unauthorized internet users from accessing resources in a private network .This method significantly reduces the probability of being attacked. However, attacks such as spam, spyware, worms and viruses, target the application layer rather than the network layer. Therefore, traditional firewalls no longer provide enough protection. Firewall routers also require several time-consuming steps to provide a secure connection and hence the virus detection speed is reduced. Fortunately, most security-guaranteed programs use rule-based designs. Therefore, a pattern matching processor, based on some algorithm is developed. This accelerates the detection speed, and can be called as a virus detection processor. The purpose of pattern matching is to check whether a text contains at least one of a given set of patterns. There are many algorithms for fast pattern matching. The goal of this work is to provide a systematic virus detection solution for network security for embedded systems. The programming language using is VHDL and is simulated using Xilinx ISE design suite 13.2 and Modelsim.**

***Keywords*—Network security, Virus, Firewall, Virus detection, Pattern matching, Embedded systems.**

I. INTRODUCTION

Virus has become a major threat to today's Internet communications. Virus detection has become an essential part in today's network communication security. The end users are vulnerable to virus attacks, spams and Trojan horses, for example. They may visit malicious websites or hackers may gain entry to their computers and use them as zombie computers to attack others. To ensure a secure network environment, firewalls were first introduced to block unauthorized Internet users from accessing resources in a private network by simply checking the packet head (MAC address/IP address/port number). This method significantly reduces the probability of being attacked. However, attacks such as spam, spyware, worms, viruses, and phishing target the application layer rather than the network layer. Therefore, traditional firewalls no longer provide enough protection. Many solutions, such as virus scanners, spam-mail filters, instant messaging protectors, network shields, content filters, and peer-to-peer protectors, have been effectively implemented. Initially, these solutions were implemented at the end-user side but tend to be merged into routers/firewalls to provide multi-layered protection. As a result, these routers stop threats on the network edge and keep them out of corporate networks.

When a new connection is established, the firewall router scans the connection and forwards these packets to the host after confirming that the connection is secure. Because firewall routers focus on the application layer of the OSI model, they must reassemble incoming packets to restore the original connection and examine them through different application parsers to guarantee a secure network environment. For instance, suppose a user searches for information on web pages and then tries to download a compressed file from a web server. In this case, the firewall router might initially deny some connections from the firewall based on the target's IP address and the connection port. Then, the fire-wall router would monitor the content of the web pages to prevent the user from accessing any page that connects to malware links or inappropriate content, based on content filters.

When the user wants to download a compressed file, to ensure that the file is not infected, the firewall router must decompress this file and check it using anti-virus programs. In summary, firewall routers require several time-consuming steps to provide a secure connection. Most security-guaranteed programs use rule-based designs. Therefore, a pattern matching processor to accelerate the detection speed is developed and it can be called as a virus detection processor because the database size it supports has reached the antivirus software level. The purpose of pattern matching is to check whether a text contains at least one of a given set of patterns.

II. LITERATURE SURVEY

A.Automata-Based Architecture

Most automata-based approaches are based on the algorithm proposed by Aho and Corasick in 1975 called AC algorithm [10] which describes a linear-time algorithm for multi-pattern search with a large finite-state machine. Aho and Corasick proposed an algorithm for concurrently matching multiple strings. Aho–Corasick (AC) algorithm used the structure of a finite automation that accepts all strings in the set. The automation processes the input characters individually and tracks partially matching patterns. The AC algorithm has proven linear performance, making it suitable for searching a large set of rule. Two different implementations exist for Snort, implemented by Mike Fisk and Marc Norton, respectively. This work tested both implementations and employed the latter in the present experiments because of its superior performance. However, the Norton implementation requires considerably more. Its performance is not affected by the size of a given pattern set (the sum of all pattern lengths), but it requires a significant amount of memory due to state explosion

B. Filtering Based Architecture

Filtering based approach is also known as heuristic based approach which is mainly based on Boyer-Moore algorithm and Bloom filters. The Boyer–Moore algorithm [Boyer and Moore 1977] is widely used because of its efficiency in single-pattern matching problems. This algorithm uses two heuristics to reduce the number of character comparisons required for pattern matching. Both heuristics are triggered by mismatches. The first heuristic is commonly referred to as the bad character heuristic, works as follows: if the search pattern contains the mismatching character, the pattern is shifted so that the mismatching character is aligned with the rightmost position at which it appears in the pattern. Meanwhile, if the mismatching character does not appear in the search pattern, the pattern is shifted so that the first character in the pattern is one position later than that of the mismatching character in the given text. The second heuristic commonly referred to as the good suffixes heuristic, works as follows: if a mismatch is found in the middle of the pattern, the search pattern is shifted to the next occurrence of the suffix in the pattern. The Boyer–Moore algorithm [9] was designed for searching for a single pattern from a given text and performs well in this role. However, the current implementation of Boyer–Moore in Snort is not efficient in seeking multiple patterns from given payloads. In Sarang et al. (2004) presented a pattern-matching processor based on Bloom filters. They used multiple Bloom filters to check different-length prefixes of the pattern in parallel. This design needs 32 memory read ports because it uses 32 hash functions. However, most commonly used memory modules only have two ports: a read port and a write port. To lower the memory read port requirements, they divided a bit vector into several smaller vectors implemented by 140-block RAM of FPGA. The total memory size, then, is 70 kB for 10038 patterns. The design also includes an analyzer that isolates false positives. The performance of this design can reach 2.46 Gb/s.

C. Bit Split method

In 2005, Lin Tan introduced a bit-split method by splitting an 8-bit character into four 2-bit characters to construct the automaton. Their state machines are smaller than the original, and they have fewer fan-out states for each transaction. However, the bit-split method reads several memory blocks in parallel when matching patterns. Thus, it can only be implemented by on-chip memory because of its high memory read port requirements. Piti Piyachon and Yan Luo extended this concept to the Intel IXP2855 network processor. For increasingly large pattern sets, an IBM team implemented an optimized AC algorithm on the cell processor, and they discovered that the memory gap was the bottleneck. As a result, they modified the algorithm and used DMA to reduce the effect on the memory system. Some designs take advantage of field-programmable gate array's (FPGA's) ability to be reconfigured to improve performance. Some of the designs are even based on non-deterministic finite automata (NFA) to handle complex regular expressions. These methods provide high throughput, but the maximum number of patterns they support is limited by the FPGA comparators. The Xilinx Virtex2-8000 FPGAs only support about 781 ClamAV rules. In 2004, to support an unlimited pattern count, Cho presented the idea of a two-phase architecture that implements a front-end filter with an FPGA and stores its full pattern database in a large memory. Later, Sourdi implemented a perfect hashing function on an FPGA to remove redundant memory accesses caused by address collisions. Some designs have used content-addressable memory (CAM) to improve engine filtering rates and to store the entire pattern database in a large external memory. Since then, pattern-matching designs have tended to use a two-phase architecture, in which one phase finds suspicious positions and the other phase precisely identifies patterns. All of these designs provide more than 1-Gb/s performance, and some support even more than 10 Gb/s. However, with increasing pattern sets, it becomes more difficult to implement these designs in on-chip memory or dedicated circuits. Some designs attempt to store pattern sets in external memory, which is typically implemented by SDRAM or DDR, for their space requirements. Although DRAM technology has greatly improved over the last few decades, DRAM-based memories still require initial cycles before pumping out their first non-consecutive data. The gain only appears in consecutive readings of various sectors. A non-consecutive read operation of DDR memory still typically costs 25 - 40 ns, compared to the 1-3 ns working cycle of existing processors. Unfortunately, most pattern matching designs have irregular access to their memories. Thus, even though the kernels of these works are well designed, their performances are slowed dramatically by these long memory access processes especially for filtering-

based designs. For this reason, improving the filter rate and overlapping the access time are the two major trends.

III. Proposed Method

The goal of this work is to provide a systematic virus detection solution for network security for embedded systems. Instead of placing entire matching patterns on a chip, our solution is a two-phase dictionary-based antivirus processor that works by condensing as much of the important filtering information as possible onto a chip and infrequently accessing off-chip data to make the matching mechanism scalable to large pattern sets. The proposed efficient design of a two-phase pattern matching virus detection processor [1] mostly comprises the filtering engine and the exact matching engine. The filtering engine is a front-end module responsible for filtering out secure data efficiently and indicating to candidate positions that patterns possibly exist at the first stage. The exact-matching engine is a back-end module responsible for verifying the alarms caused by the filtering engine. The two stages can be designed with the help of some proposed algorithms. Both engines have individual memories for storing significant information. The filtering engine's on-chip memory can store only a small amount of significant information regarding the patterns. Conversely, the exact-matching engine not only stores the entire pattern in external memory but also provides information to speed up the matching process. Our exact-matching engine is space-efficient and requires only four times the memory space of the original size pattern set. The size of a pattern set is the sum of the pattern length for each pattern in the given pattern set; in other words, it is the minimum size of the memory required to store the pattern set for the exact-matching engine.

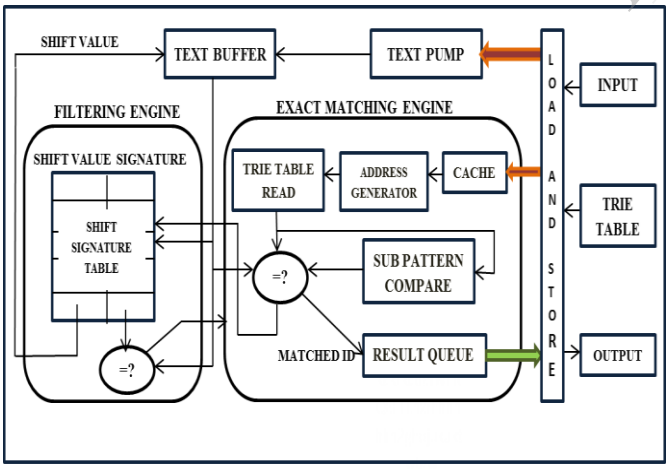


Fig.1 Virus Detection processor

The proposed exact-matching engine also supports data prefetching and caching techniques to hide the access latency of the off-chip memory by allocating its data structure well. The other modules include a text buffer and a text pump that prefetches text in streaming method to overlap the matching progress and text reading. A load/store interface was used to support bandwidth sharing. The architecture of virus detection processor is shown in Figure 1.

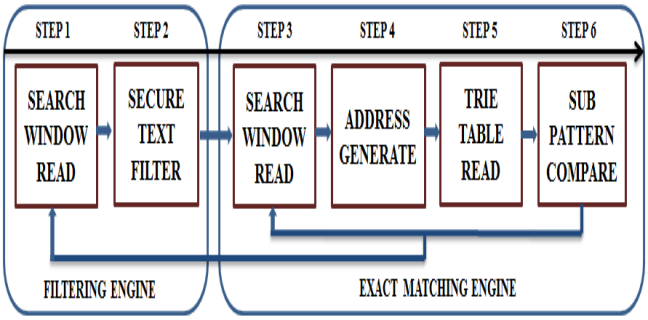


Fig.2 Two Phase Execution Flow

This proposed architecture has six steps shown in Figure 2 for finding patterns. Initially, a pattern pointer is assigned to point to the start of the given word at the filtering stage. Suppose the pattern matching processor examines the word from left to right. The filtering engine fetches a piece of word from the word buffer according to the pattern pointer and checks it by a shift-signature table. If the position indicated by the pattern pointer is not a candidate position, then the filtering engine skips this piece of word and shifts the pattern pointer rights multiple characters to continue to check the next position. The shift signature table created by the data structure used the Bloom filter algorithm, and it provides layer filtering. If layer is missing their filter, the processor enters the exact-matching phase. The next section has details about the shift-signature table.

After filtering engine filters the word, the exact matching engine precisely verifies this word by retrieving a trie structure. This structure divides a pattern into multiple sub-patterns and systematically verifies it. The exact-matching engine generally has four steps for each check. First, the exact-matching engine gets a slice of the word and hashes it to generate the trie address. Then, the exact-matching engine fetches the trie node from memory. This step causes a long latency due to the access time of the off-chip memory.

Finally, the exact-matching engine compares the trie node with this slice. When this node is matched, the exact-matching engine repeatedly executes the above steps until it matches or misses a pattern. The pattern matching then backs out to the filtering engine to search for the next candidate. The details of table generation and matching flow are explained in the following sections.

A. Filtering Engine Techniques

In this work, the overall performance strongly depends on the filtering engine. Providing a high filter rate with limited

space is the most important issue. Filter Engines have individual memories for storing significant information. For cost reasons, only a small amount of significant information regarding the patterns can be stored in the filtering engine's on-chip memory. The filtering engine techniques contain two algorithms, which are Wu-Manber algorithm and Bloom filter algorithm.

A.1 Wu-Manber Algorithm

The Wu-Manber algorithm is a high-performance, multi-pattern matching algorithm based on the Boyer-Moore algorithm. The Wu-Manber algorithm is an exact-matching algorithm, but its shift table is an efficient filtering structure. The shift table is an extension of the bad-character concept in the Boyer-Moore algorithm [8]. It builds three tables in the pre-processing stage, which includes shift table, hash table and prefix table.

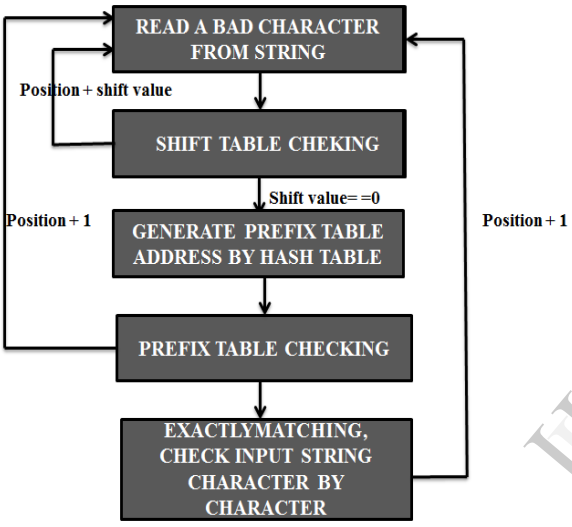


Fig.3 Wu-Manber Matching flow

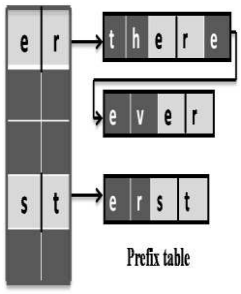
The matching flow is shown in Figure 3. The performance of the Wu-Manber algorithm [2] is not proportional to the size of the pattern set directly, but it is strongly dependent on the minimum length of the pattern in the pattern set. The minimum length of the pattern dominates the maximum shift distance (m-B+1) in its shift table, where m is the minimum length of the pattern in the pattern set and B is the block of characters from the text.

For the pattern set {erst, ever, there}, the maximum shift value is three characters for B=2 and m=4. The related shift table, hash table and prefix are shown in Table 1 and Table 2. The algorithm starts by pre-computing two tables, a bad character shift table and a hash table. When the bad character shift fails, the first two characters of the string are indexed into a hash table to find a list of pointers to possible matching patterns. The Wu-Manber algorithm scans patterns from the head of a text, but it compares the tails of the shortest patterns.

TABLE I
Shift Table

Bad Character	Shift value
er	0
ev	2
he	1
rs	1
st	0
th	2
ve	1
others	3

TABLE II
Hash table + Prefix Table



The Figure 4 explains the process steps. In step 1, the arrow indicates to a candidate position that a wanted pattern probably exists, but the search window is actually the character it fetches for comparison. According to shift [ev] = 2, the arrow and search window are shifted right by two characters. Then, the Wu-Manber algorithm finds a candidate position in step 2 due to shift[er] = 0. Consequently, it checks the prefix table and hash table to perform an exact-matching and then outputs the "ever" in step 3. After completing the exact match, the Wu-Manber algorithm returns to the shifting phase, and it shifts the search window to the right by one character to find the next candidate position in step 4. The algorithm shifts the search window until it touches the end of the string in step 6.

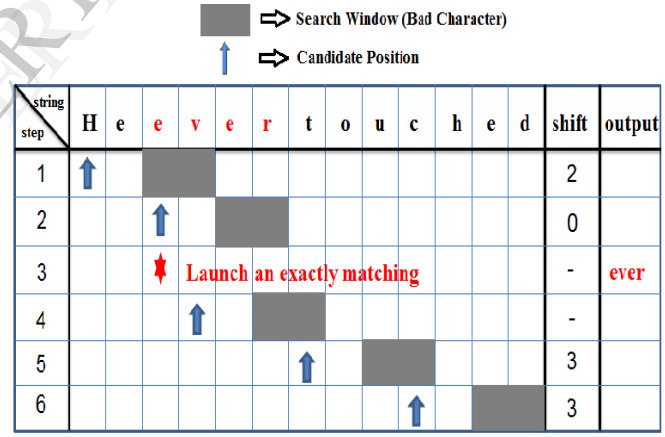


Fig.4 Wu-Manber matching process

A.2 Bloom Filter Algorithm

A Bloom filter is a space-efficient data structure used to test whether an element exists in a given set. This algorithm is composed of different hash functions and a long vector of bits. Initially, all bits are set to 0 at the pre-processing stage. To add an element, the Bloom filter hashes the element by these hash functions and gets k positions of its vector. The Bloom filter [3] then sets the bits at these positions to 1.This algorithm fetches the prefix of a pattern from the text and hashes it to generate a signature. Then, this algorithm checks

whether the signature exists in the bit vector. If the answer is yes, it shifts the search window to the right by one character for each comparison and repeats the above step to filter out safe data until it finds a candidate position and launches exact-matching.

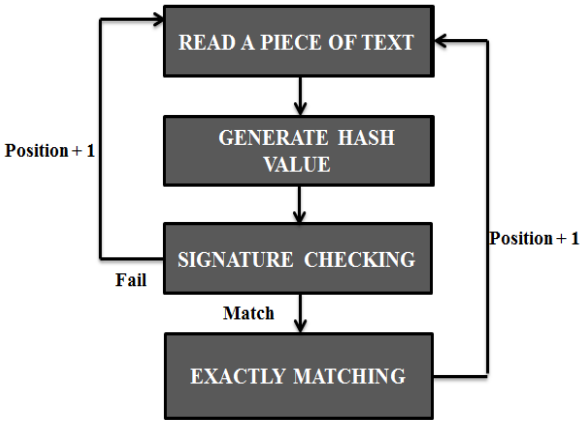


Fig.5 Bloom filter- Matching flow

The value of a vector that only contains an element is called the signature of an element. To check the membership of a particular element, the Bloom filter hashes this element by the same hash functions at run time, and it also generates 'k' positions of the vector. If all of these 'k' bits are set to 1, this query is claimed to be positive, otherwise it is claimed to be negative. The output of the Bloom filter can be a false positive but never a false negative. Therefore, some pattern matching algorithms based on the Bloom filter must operate with an extra exact-matching algorithm. However, the Bloom filter still features the following advantages: 1) It is a space-efficient data structure; 2) The computing time of the Bloom filter is scaled linearly with the number of patterns.

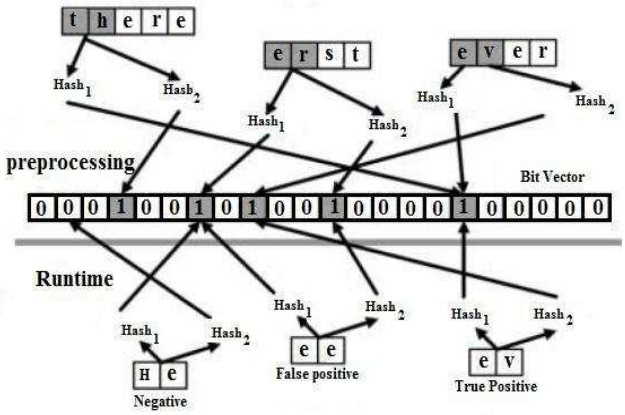


Fig.6 Bit vector building

The Bloom filter is independent of its pattern length. Figure 6 shows how a Bloom filter builds its bit vector for a pattern set {erst, ever, there} for two given hash function0s. The filter only hashes all of the pattern prefixes at the pre-processing stage. Multiple patterns setting the same position of the bit vector are allowed. Figure 7 shows an example of the matching process. The arrows indicate the candidate positions. The gray bars represent the search window that the Bloom filter actually fetches for comparison. Both the candidate position and search window are aligned together. Thus, the Bloom filter scans and compares patterns from the head rather than the tail, like the Wu-Manber algorithm.

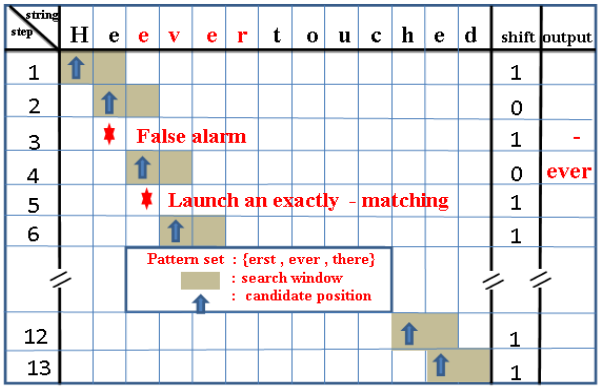


Fig.7 Bloom filter - matching process

In step 1, the filter hashes "He" and mismatches the signature with the bit vector. The filter then shifts right 1 character and finds the next candidate position. For the search window "ee", the Bloom filter matches the signature and then causes a false alarm to perform an exact-matching in steps 2 and 3. The filter then returns to the filtering stage and shifts one character to the right in step 4, which launches a true alarm for the pattern "ever". Finally, the Bloom filter filters the rest of text and finds nothing.

A.3 Shift Signature Algorithm

Shift Signature algorithm is a combination of Wu Manber and Bloom Filter algorithm. The proposed algorithm re-encodes the shift table to merge the signature table into a new table named the shift-signature table. There are two fields, S-flag and carry, in the shift signature table. The carry field has two types of data, a shift value and a signature. These two data types are used by two different algorithms as in figure 8.

The filtering engine can then filter the text using a different algorithm while providing a higher filter rate. The method used to merge these two tables is described as follows. First, the algorithm generates two tables, a shift table and signature table, at the preprocessing stage. The generation of the shift table is the same as in the Wu Manber algorithm. The shift table is used as the primary filter. The signature table could be considered a set of the bit vector of the Bloom filter, and it is used for the second level filtering. The signature table's generation is similar to the Bloom filter but is not identical. It hashes the tail characters of patterns to generate their signatures instead of the prefix. Signature is indexed by bad-characters. After the shift table and signature table are generated. The algorithm re-encodes the shift value into two fields an S-flag and a carry in the shift-signature table.

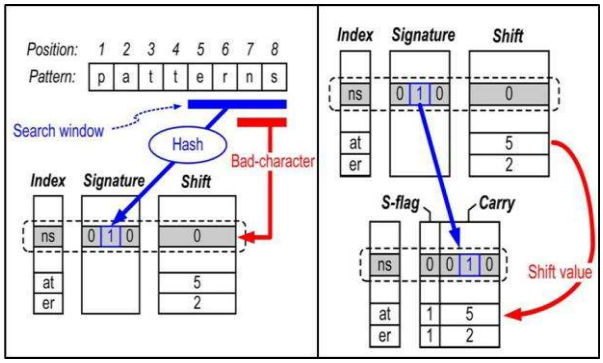


Figure 8.Table generation and re-encoding of shift-signature algorithm.

To merge these two tables, the algorithm maps each entry in the shift table and signature table onto the shift-signature table. For the non-zero shift values, the S-flags are set, and their original shift values are cut out at 1-bit to fit their carries. Conversely, for the zero shift values, their S flags are clear, and their carries are used to store their signatures.

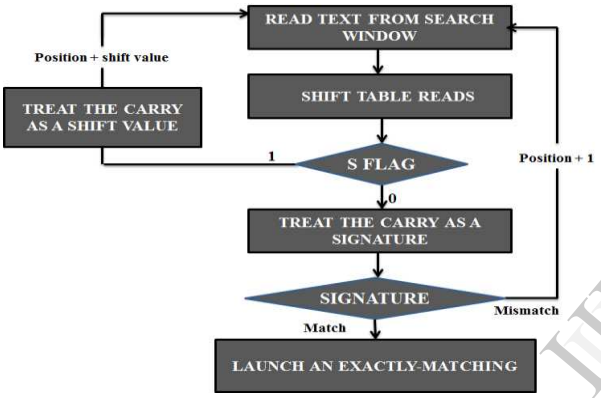


Figure 9.Shift Signature - matching process

The filtering flow is shown in Figure 9. For the pattern set {patterns}, Figure 10 illustrates how the filtering engine filters out the given text. The filtering engine fetches the text from the search window (blue bar) and one part of the fetched text (red bar), which is used as a bad character to index the shift-signature table, as shown in Figure 10. If the S-flag is set, the carry is treated as a shift value. As a result, the filtering engine shifts the candidate position to the right by two characters for the text “overheated”, as shown in Figure 10. Conversely, if the S-flag is clear, the carry is treated as a signature. The filtering engine hashes the fetched text and matches it with the signature read from the shift-signature table. Figure 10 indicates that the fetched text “he” has the same index as the bad-character “ns”, but it fails to match the signature. Thus, the filtering engine shifts the candidate position to the right by one character to provide second-level filtering.

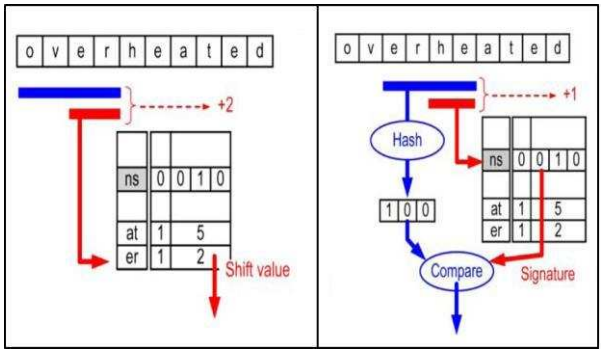


Figure 10.Shift Filtering and Signature Filtering

IV. SIMULATION RESULTS

The design entry is modelled using VHDL in Xilinx ISE Design Suite 13.2 and the simulation of the design is performed using Modelsim from Xilinx ISE to validate the functionality of the design. The Filtering Engine designed differently, using three different algorithms is considered in this work. When a set of input data are given, it will perform the pattern matching process and produces an alarm when a matching is found. The Performance of the three algorithms is also compared in this work.

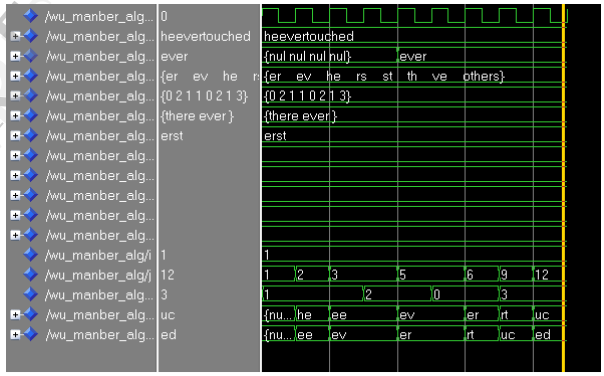


Figure 11.Simulation result of Wu-manber algorithm

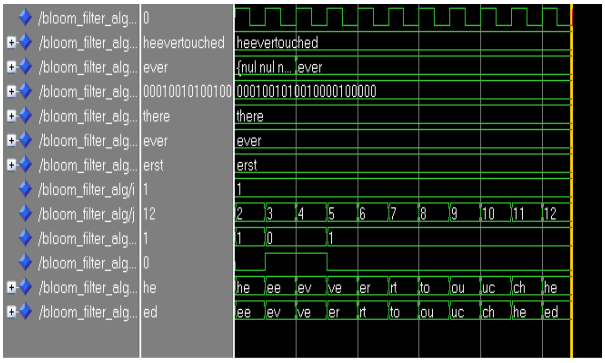


Figure 12.Simulation result of Bloom filter algorithm

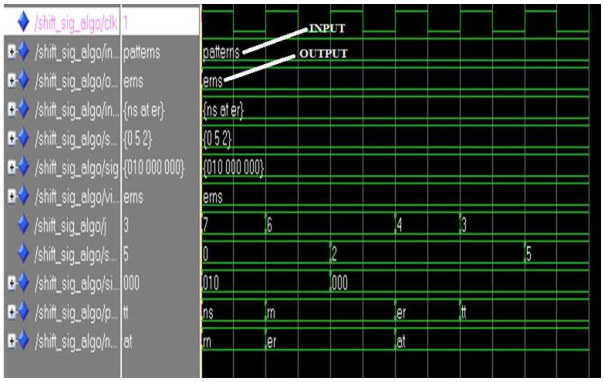


Figure 13.Simulation result of Bloom filter algorithm

Figure 11 shows the simulation result of the Wu-Manber algorithm, Figure 12 shows the simulation result of the Bloom Filter algorithm and Figure 13 shows the simulation result of the Shift signature algorithm, in the filtering engine section of a virus detection processor.

V. COMPARISON

From the synthesis reports, it can be concluded that Shift signature algorithm is more space efficient than bloom Filter algorithm, which is more space efficient algorithm than Wu-Manber algorithm. Comparison of the three algorithms is shown in figure 14. In case of memory usage, Shift signature algorithm is more advantageous than the other two algorithms. Also the time required to perform the virus detection process in the filtering engine part is least for Bloom filter, then comes the Shift signature and Wu Manber algorithms. Bloom filters are so useful that it is possible for significant reduction in the time required to perform a Bloom filter operation. The Shift Signature algorithm has high filtering rate when compared to the other two algorithms, as it has two level filtering of both the other algorithms.

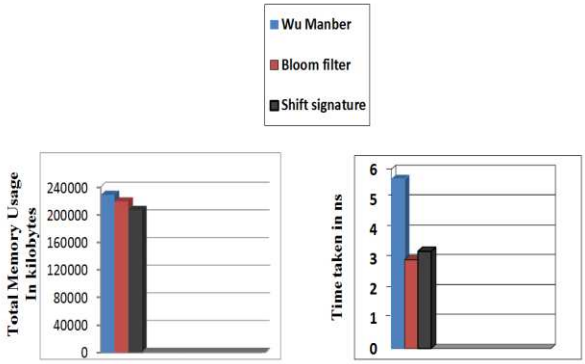


Figure14. Comparison of the algorithms

VI. CONCLUSIONS

A Network Intrusion detection system for the purpose of detecting viruses through pattern matching is developed, as an alternative to firewall. This processor is having a filtering engine part and an exact matching engine part. The filtering engine section can be done with different algorithms. Wu-Manber and Bloom filter algorithm are discussed in this work. Bloom filter algorithm is more advantageous than Wu-Manber algorithm as it has a space efficient data structure. Two-phase heuristic algorithms are a solution with a tradeoff between performance and cost due to an efficient filter table existing in internal memory; however, their performance is easily threatened by malicious attacks. The two algorithms are designed in VHDL and simulated using ModelSim 6.2b design suit from Mentor Graphics.

REFERENCES

[1] Chieh-Jen Cheng, Chao-Ching Wang, Wei-Chun Ku,Tien-Fu Chen and Jinn- Shyan Wang, "A Scalable High-Performance Virus Detection Processor Against a Large Pattern Set for Embedded Network Security", *IEEE Transactions on Very Large Scale Intergrations(VLSI) Systems*, Vol. 20, NO. 10, May 2012.

[2] Huangshan, China P. R., "High Concurrence Wu-Manber Multiple Patterns Matching Algorithm", *Proceedings of the 2009 International Symposium on Information Processing (ISIP'09),* Aug 21-23, 2009, pp. 404-409.

[3] Sarang Dharmapurikar and John Lockwood, "Fast and Scalable Pattern Matching for Network Intrusion Detection Systems", *IEEE Journal On Selected Areas In Communications*, Vol. 24, NO.5, Oct 2006.

[4] Cho Y. H. and Mangione-Smith W. H., "A pattern matching coprocessor for network security," presented at the 42nd Annu. Des. Autom. Conf. Anaheim, CA, 2005.

[5] Tan L. and Sherwood T. , "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. 32nd Annu. Int. Symp. Comput. Arch.*, 2005, pp. 112–122.

[6] Liu R.T. , Huang N.F., Kao C.N., and Chen C.H. ,"A fast string matching algorithm for network processor-based intrusion detection system," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 614–633, 2004.

[7] Clark C. R. and Schimmel D. E., "Scalable pattern matching for high speed networks," in *Proc. 12th Annu. IEEE Symp. Field-Program.Custom Comput. Mach.*, 2004, pp. 249–257.

[8] Wu S. and Manber U., "A fast algorithm for multi-pattern searching," Univ. Arizona, Tucson, Report TR-94-17, 1994.

[9] Boyer R. S.and Moore J. S., "A fast string searching algorithm, *"Commun. ACM*, vol.20, pp. 762-772, 1977.

[10] Aho A. V. and Corasick M. J., "Efficient string matching: An aid to bibliographic search," *Commun.* ACM, vol.18, pp.333-340 , 1975.