International Journal of Computer and Communication Technology

Volume 6 | Issue 3 Article 8

July 2015

AN EFFICIENT AND SYSTEMATIC VIRUS DETECTION PROCESSOR FOR EMBEDDED NETWORK SECURITY

P.MUTHU KUMARAN
Sasurie Academy of Engineering, muthukumaran23@gmail.com

R.V.ASHOK PRATHAP
Sasurie Academy of Engineering, ashokeee619@gmail.com

D. MATHAVAN
Sasurie Academy of Engineering, madymathavan24@gmail.com

AN EFFICIENT AND SYSTEMATIC VIRUS DETECTION PROCESSOR FOR EMBEDDED NETWORK SECURITY

P.MUTHU KUMARAN[1], R.V.ASHOK PRATHAP[2] & D.MATHAVAN[3]

[1,2&3]Sasurie Academy of Engineering
Email:muthukumaran23@gmail.com[#1,] ashokeee619@gmail.com[#2,] madymathavan24@gmail.com[#3]

Abstract— Network security has always been an important issue and its application is ready to perform powerful pattern matching to protect against virus attacks, spam and Trojan horses. 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. However, the solutions in the literature for firewalls are not scalable, and they do not address the difficulty of an antivirus. The goal is to provide a systematic virus detection hardware solution for network security for embedded systems. Instead of placing entire matching patterns on a chip, our solution is based on an antivirus processor that works as much of the filtering information as possible onto a chip. The infrequently accessing off-chip data to make the matching mechanism scalable to large pattern sets. In the first stage, the filtering engine can filter out more than 93.1% of data as safe, using a merged shift table. Only 6.9% or less of potentially unsafe data must be precisely checked in the second stage by the exact-matching engine from off-chip memory. To reduce the memory gap and to improve the performance, we also propose three algorithms are used: 1) a skipping algorithm; 2) a cache method; and 3) a prefetching mechanism.

Index Terms— *Algorithmic Attacks, Embedded System, Memory Gap, Network Security, Virus Detection.*

I. INTRODUCTION

NETWORK security has always been an important issue. 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.

A. Firewall Routers

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 in-coming 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 com-pressed 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**.**

II. EXISTING SYSTEM

There are many algorithms and accompanying hardware accelerators for fast pattern matching. One of the typical algorithms is the automation approach. This approach is based on Aho and Corasick's algorithm (AC), which introduces a linear-time algorithm for multi-pattern search with a large finite-state ma-chine. Its performance is not affected by the size of a given pattern set (the sum of all pattern lengths). In contrast, heuristic approaches are based on the Boyer- Moore algorithm, which was introduced in 1977. Its key feature is the shift value, which shifts the algorithm's search window for multiple characters when it encounters a mismatch.

The search window is a range of text exactly fetched by pattern matching algorithms for each examination. This algorithm performs better because it makes fewer comparisons than the naïve pattern-matching algorithm. At runtime, the Boyer-Moore algorithm uses a pattern pointer to locate a candidate position by assuming that a desired pattern exists at this position. The algorithm then shifts its search window to the right of this pattern. By default, desired patterns can exist in any position of a text; therefore, all positions in a text are candidate positions and must be examined. If the string of search windows does not appear in the pattern, the algorithm can shift the pattern pointer to the right and skip multiple characters from the candidate position to the end of the pattern without making comparisons. Based on this concept, Wu and Manber (WM) modified the Boyer-Moore algorithm to search for multiple patterns. However, the performance of both of these algorithms is bounded by the pattern length.

B. Related Work

Focus on algorithms and have even developed for specialized circuits to increase the scanning speed. However, these works have not considered the interactions between algorithms and memory hierarchy. Because the number of attacks is increasing, pattern-matching processors require external memory to support an unlimited pattern set. This method makes the memory systemthe bottleneck. However,when the pattern set is already intractably large, a perfect solution is unattainable.

III. VIRUS DETECTION PROCESSOR

Virus Detection Processor shown in Fig.1 is a two-phase pattern matching architecture mostly comprising the filtering engine and the exact-matching engine. The filtering engine is a frontend module responsible for filtering out secure data efficiently and indicating to candidate positions that patterns pos sibly exist at the first stage. The exact-matching engine is responsible for verifying the alarms caused by the filtering engine. Only a few unsaved data need to be checked precisely by the exact-matching engine in the second stage.

Both 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. In this case, we use a 32-kB on-chip memory for the ClamAV virus database, which contained more than 30 000 virus codes and localized most of the computing inside the chip.

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. In this case, 8 MB of off-chip memory was required for the ClamAV virus database (2 MB).

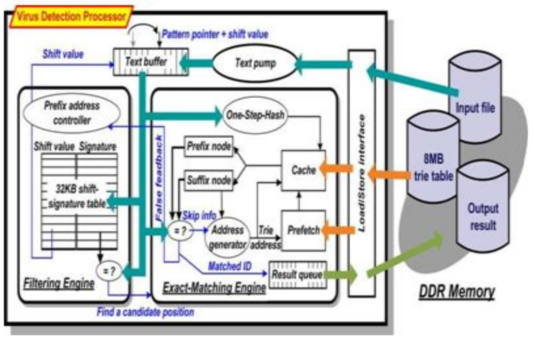


Figure 1: Virus Detection Processor Architecture.

The proposed exact-matching engine also supports data pre fetching 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 pre-fetches text in streaming method to overlap the matching progress and text reading. A load/store interface was used to support bandwidth sharing.

This proposed architecture has six steps shown in Fig.2 for finding patterns. Initially, a pattern pointer is assigned to point to the start of the given text at the filtering stage. Suppose the pattern matching processor examines the text from left to right. The filtering engine fetches a piece of text from the text buffer. If the position indicated by the pattern pointer is not a candidate position, then the filtering engine skips this piece of text and shifts the pattern pointer right multiple characters to continue to check the next position.

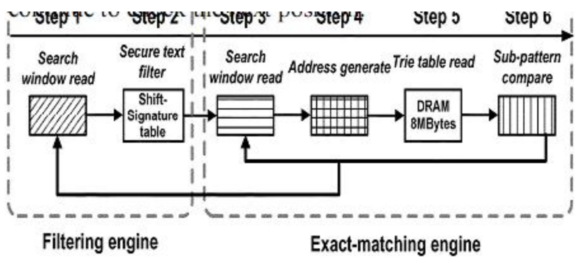


Figure 2: Two-phase Execution Flow

The shift-signature table combines two data structures used by two different filtering algorithms, the Wu Manber algorithm and the Bloom filter algorithm, and it provides two-layer filtering. If both layers are missing their filter, the processor enters the exact-matching phase. The next section has details about the shift-signature table.

IV. FILTERING ENGINE (FE)

Designs that feature filters indicate that the action behind these filters is costly and necessary. 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. Two classical filtering algorithms were introduced for pattern matching in the following sections. We then show how to merge their string structures in the space to to improve the filter rate.

A. Wu-Manber Algorithm

The Wu-Manber algorithm is a high-performance, multi-pattern matching algorithm based on the Boyer-Moore algorithm. It builds three tables in the pre processing stage: a shift table, a hash table and a prefix table. 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, but they are not identical. The matching flow is shown in Fig. 3(a).

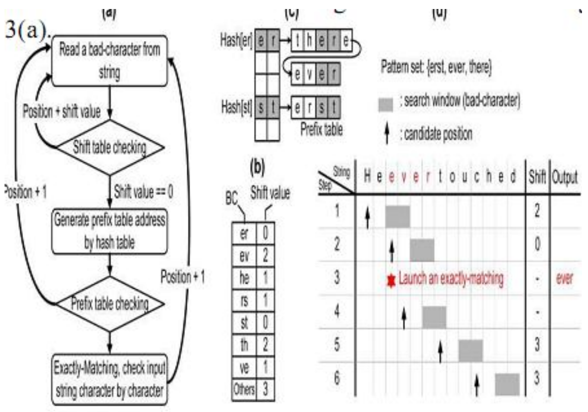

3(a).
Read a bad-character from string
Position + shift value
Shift table checking
Position + 1
Shift value == 0
Generate prefix table address by hash table
Position + 1
Prefix table checking
Exactly-Matching, check input string character by character
Hash[er]
Hash[st]
Prefix table
Pattern set: {erst, ever, there}
: search window (bad-character)
: candidate position
(b)
BC
Shift value
er 0
ev 2
he 1
rs 1
st 0
th 2
ve 1
Others 3
String
Step
H e e v e r t o u c h e d
Shift
Output
Launch an exactly-matching
ever

Figure 3: Wu-Manber Matching Process. (a) Matching Flow; (b) Shift Table; (c)Hash Table + Prefix Table; (d) Matching Process.

The matching flow matches patterns from the tail of the minimum pattern in the pattern set, and it takes a block *B* of characters from the text instead of taking them one-by-one. The shift table gives a shift value that skips several characters without comparing after a mismatch. After the shift table finds a candidate position, the Wu-Manber algorithm enters the exact-matching phase and is accelerated by the hash table and the prefix table. Therefore, its best performance is *O(BN/m)* for the given text with length *N* and the pattern set, which has a minimum length of *m* . The performance of the Wu-Manber algorithm 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. However, the Wu-Manber algorithm is still one of the algorithms with the best performance in the average case.

For the pattern set {erst, ever, there} shown in Fig. 3(d), the maximum shift value is three characters for *B=2* and *m*=4. The related shift table, hash table and prefix are shown in Fig. 3(b) and Fig. 3(c). The Wu-Manber algorithm scans patterns from the head of a text, but it compares the tails of the shortest patterns. 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 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 instep 4. The algorithm keeps shifting the search window until touching the end of the string in step 6.

B. 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 positions of its vector. The Bloom filter then sets the bits at these positions to 1. 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 positions of the vector.

If all of these 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; and 3) the Bloom filter is independent of pattern length.

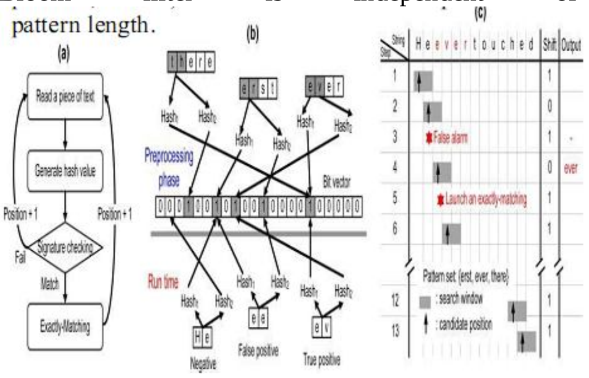


Figure 4: Bloom Filter Matching Process. (a) Matching Flow; (b) Bit-Vector Building; (c) Matching Process

Fig. 5(a) describes a typical flow of pattern matching by Bloom filters. 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(b) shows how a Bloom filter builds its bit vector for a pattern set {erst, ever, there} for two given hash functions. 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. Fig. 5(c) shows an example of the matching process. The arrows indicate the candidate positions. The gray bars represent the search 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. In step1, the filter hashes "He" and mismatches the signature with the bit vector. The filter then shifts right 1character 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".

C. Shift-Signature Algorithm

The proposed algorithm re-encodes the shift table to merge the signature table into a new table named the shift-signature table. The shift-signature table has the same size as the original shift table, as its width and length are the same as the original shift 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. Thus, the S-flag is used to indicate the data type of a carry. 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 pre processing stage. The generation of the shift table is the same as in the Wu-Manber algorithm.

The S-flag is a1-bit field used to indicate the data type of the carry. Two data types, shift value or signature, are defined for a carry. The size and width of the shift signature table are the same as those of the original shift table. 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. In this method, all of the entries in the shift-signature table contribute to the filtering rate at run time. Because of the address collision of bad-characters, most entries contain less than half of the maximum shift distance for a large pattern set. Therefore, although this method sacrifices the maximum shift distance, the filter rate is not reduced but rather improved.

Fig. 5(a) shows an example of generating the shift and signature tables. Suppose the length of the shortest pattern "patterns" in the pattern set is 8 characters. The size of the bad-character is 2 characters, thus the maximum shift distance is 8-2+1=7 characters. Seven possible bad-characters ("pa", "at", "tt", "te", "er", "rn", "ns") are defined according to the Wu-Manber algorithm, and their shift values are 6, 5, 4, 3, 2, 1, and 0. Before replacement, the algorithm first builds the signature table. For each pattern, the algorithm hashes the tail characters of a pattern(blue bar) to generate its signature. The signature is then assigned to the signature table indexed by the bad-character "ns".

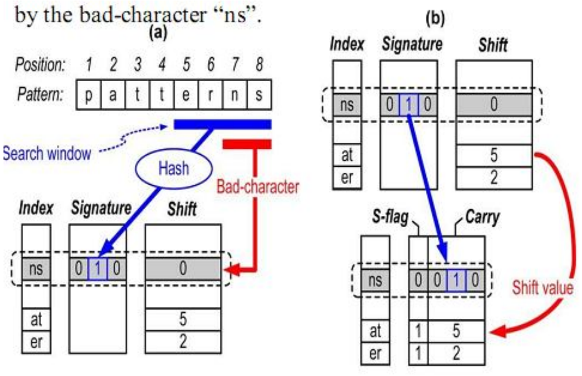


Figure 5: Table Generation and Re-Encoding of Shift-Signature Algorithm. (a) Table Generation; (b) Table Re-Encoding

For multiple signatures mapped to the same entry, the entry stores the results of the OR operation of these signatures. In this work, we only use one hash function because of the space limitation of the signature table. The method of merging the shift table and signature table is shown in Fig. 5(b). Then is replaced by its signature ("010" in binary) because its shift value is zero. In contrast, the shift[at]=5 and shift[er]=2 keep their shift values in the shift-signature table.

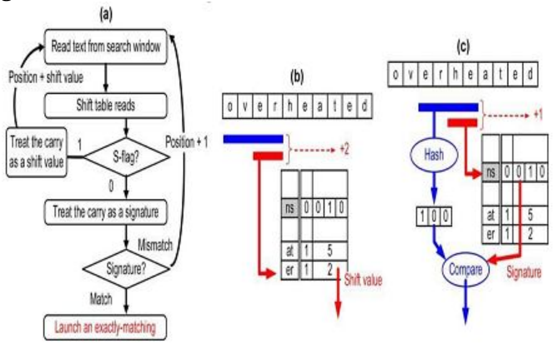


Figure 6: Matching Flow and Filtering Example. (a) Filtering Flow; (b) Shift Filtering; (c) Signature Filtering.

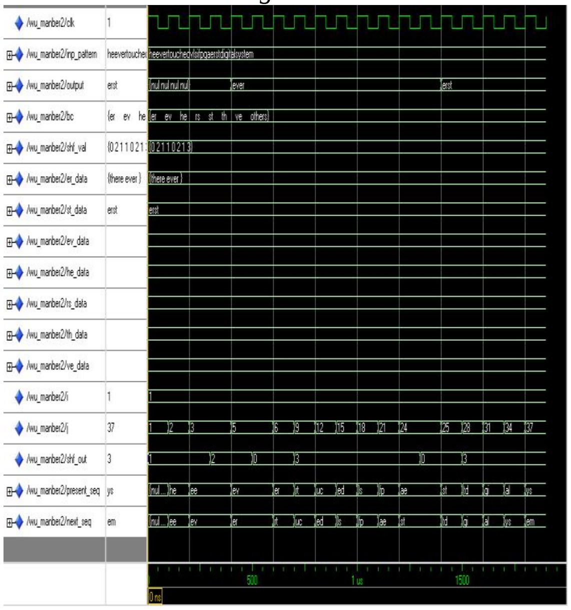
The filtering flow is shown in Fig. 8(a). For the pattern set {patterns}, Fig. 6(b) and Fig. 6(c) illustrate how the filtering engine filters out the given text. The filtering engine fetches the text from the search window (blue bar), as shown in Fig. 8(c).One part of the fetched text (red bar), shown in Fig. 6(b), is used as a bad character to index the shift-signature table. 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 “overhead”, as shown in Fig. 6(b). 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. Fig. 6(c) 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.

V. FUTURE WORK

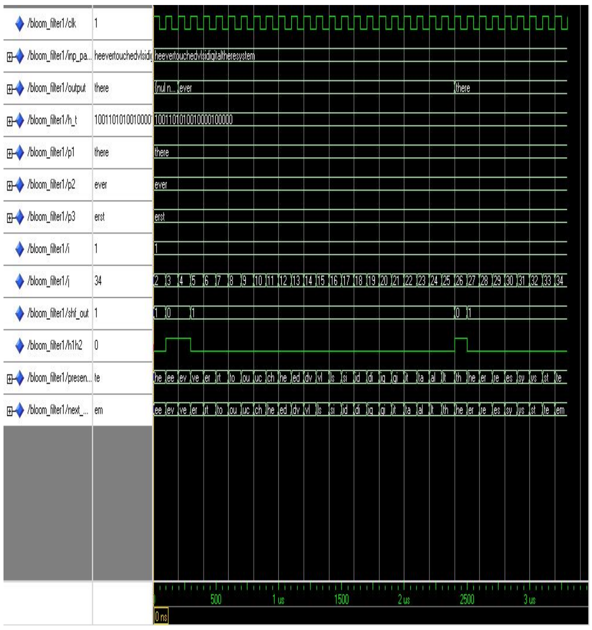
In my future work 1) a shift-signature table and 2) a trie-skip mechanism will be combined to improve the performance and improve the blow of the impact on memory gap for this two-phase architecture. First, we re-encode the shift table and Bloom filter to merge them into the same space, the shift-signature table. The new table not only maintains the shift value of properties but also avoids reducing the filter rate for a large scale pattern set. Second, the trie-skip mechanism avoids performance reduction during malicious attacks. Trie-skip mechanism overcomes these two attacks by skip values and jump nodes. With these two fields, use the new trie structure suitable for prefetched and cached to reduce the off-chip memory access just by rearranging the trie structure.

VI. SIMULATION RESULTS

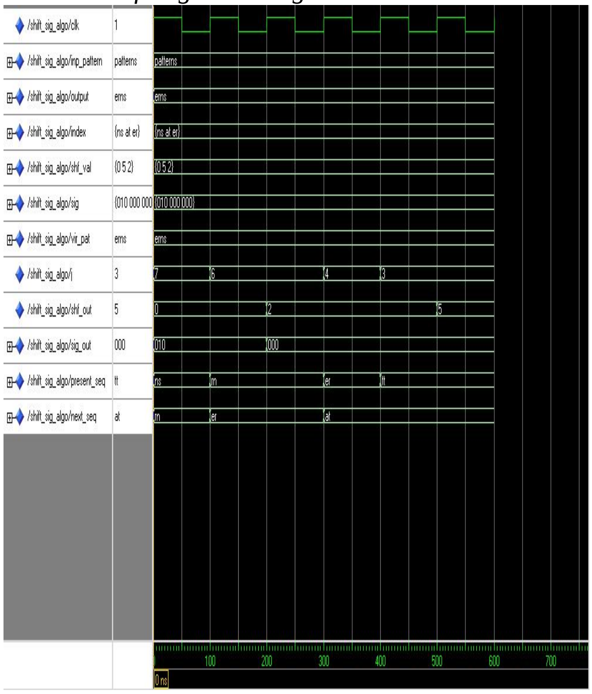
A. Wu-Manber Algorithm



B. Bloom Filter Algorithm



C. Shift-Signature Algorithm



VII. CONCLUSION

Many previous designs have claimed to provide high performance, but the memory gap created by using external memory decreases performance because of the increasing size of virus databases. Furthermore, limited resources restrict the practicality of these algorithms for embedded network security systems. 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. This work analyzes two scenarios of malicious attacks and provides two methods for keeping performance within a reasonable range. First, were-encoded the shift table to make it provide a bad-character heuristic feature and high filter rates for large pattern sets at the same time. Second, the proposed skip mechanism increases the blow to performance under algorithmic attacks.

REFERENCES

[1] Chieh-Jen Cheng, Chao-Ching Wang, Wei-Chun Ku, Tien-Fu Chen , and Jinn-Shyan Wang, "Scalable High-Performance Virus Detection Processor Against a Large Pattern Set for Embedded Network Security" Commun. VOL. 20, NO. 5, MAY 2012

[2] O. Villa, D. P. Scarpazza, and F. Petrini, "Accelerating real-time string searching with multicore processors," Computer, vol. 41, pp. 42–50,2008.

[3] D. P. Scarpazza, O. Villa, and F. Petrini, "High-speed string searching against large dictionaries on the Cell/B.E. processor," in Proc. IEEE Int. Symp. Parallel Distrib. Process., 2008, pp. 1–8.

[4] D. P. Scarpazza, O. Villa, and F. Petrini, "Peak-performance DFA based string matching on the Cell processor," in Proc. IEEE Int. Symp. Parallel Distrib. Process., 2007, pp. 1–8.

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

[6] S. Dharmapurikar, P. Krishnamurthy, and T. S. Sproull, "Deep packet inspection using parallel bloom filters," IEEE Micro, vol. 24, no. 1, pp.52–61, Jan. 2004.

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

[8] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern matching using TCAM," in Proc. 12th IEEE Int. Conf. Netw. Protocols, 2004, pp. 174–178.intrusion detection system," ACMTrans. Embed. Comput. Syst., vol. 3, pp. 614–633, 2004.

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

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

[11] H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Commun. ACM, vol. 13, pp. 422–426, 1970.

◈◈◈