Advanced Pattern Based Virus Detection Algorithm for Network Security

Binroy T.B.
M.E. Communication Systems
Department of Electronics and Communication Engineering
RVS College of Engineering & Technology, Coimbatore, India

Lakshmanan B.
Assistant Professor
Department of Electronics and Communication Engineering
RVS College of Engineering & Technology, Coimbatore, India

Abstract: - The ordinary network security applications generally require the ability to perform powerful pattern matching to protect against attacks such as viruses. In normal case hardware solutions are intended for firewall routers. The solutions in the literature for firewalls are not scalable and they do not address the difficulty of a antivirus with ever large pattern set. The goal of this work is to provide a systematic virus detection hardware solution for embedded network security systems. It 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. In the first stage, the filtering engine can filter out more than 93% of data as safe, using a merged shift table. Only 7% or less of potentially unsafe data must be precisely checked in the second stage by the exact-matching engine from off-chip memory.

Keywords - Algorithmic attacks, embedded system, memory gap, network security, virus detection

I. INTRODUCTION

Network security is an important issue. The end users are vulnerable to virus attacks. They may visit malicious websites or hackers may gain to their computers and use them as zombie computers to attack others. Firewalls were first introduced to ensure a secure network environment 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 multilayered protection. As a result, these routers stop threats on the network edge and keep them out of corporate networks.

1.1. 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 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 antivirus programs.
In summary, firewall routers require several time-consuming steps to provide a secure connection. Therefore here it is introducing an advanced pattern based virus detection algorithm that can be used for developing a virus detection processor to accelerate the detection speed.

II PROPOSED 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 lineartime algorithm for multipattern. 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. It's 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. Software-based Bloom filters were first described in 1970. These filters can determine whether an element is a non-member of a given set in a constant amount of time using several hash functions and a bit vector. The Bloom filter method is exceptionally space-efficient. In a typical case, the filter rate for 30 000 patterns reaches 90% and requires only 34.76 kB of memory.

2.1 Virus Detection Processor

Virus detection processor shown in Fig1 is a two phase pattern matching architecture mostly comprising the filtering engine and the exact-matching engine.

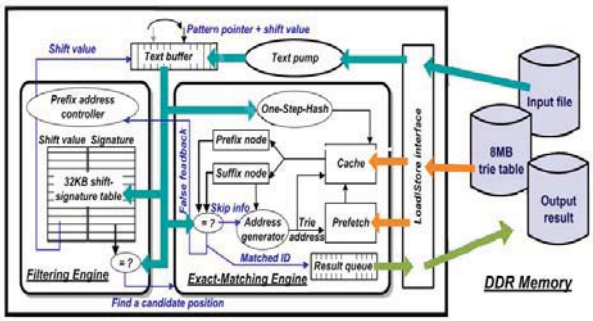


Fig. 1. Virus Detection Processor architecture

The filtering engine is a front end module responsible for filtering out secure data efficiently and indicating

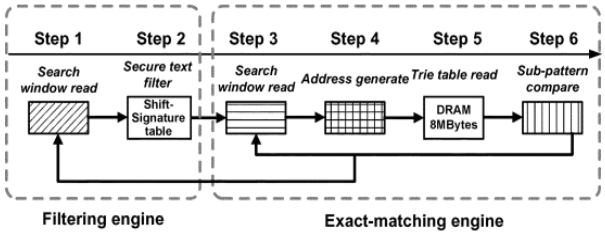


Fig. 2. Two phase execution flow

to candidate positions that patterns possibly exist at the first stage. The exact-matching engine is responsible for verifying the alarms caused by the filter engine. Only a few unsaved data need to be checked precisely by the exact-matching engine in the second stage.

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
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

2.2. Filtering Engine (FE) -

The overall performance strongly depends on the filtering engine. The most important issue is to provide a high filter rate with limited space. Two classical filtering algorithms were introduced for pattern matching in the following sections.

2.3 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). The matching flow matches patterns from the tail of the minimum pattern in the pattern set, and it takes a block of *B* 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 theWu-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.

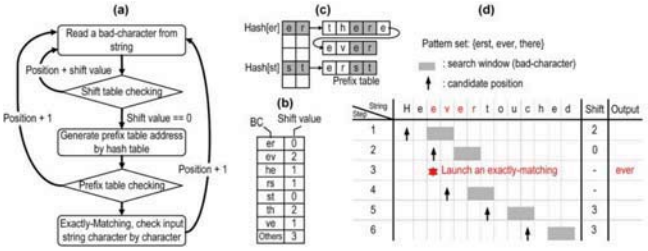


Fig. 3. Wu-Manber matching process. (a) Matching flow; (b) shift table; (c) hash table + prefix table; (d) matching process.

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.

2.4 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 '*k*' different hash functions and a long vector of '*v*' 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 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. 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.

Fig. 4(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

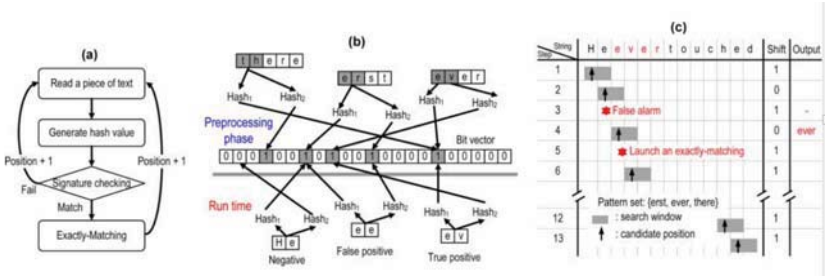


Fig.4 Bloom filters matching process. (a) Matching flow; (b) bit-vector building; (c) matching process.

position and launches exact-matching. Fig. 4(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. 4(c) shows an example of the matching process. The arrows indicate the candidate positions. The grey 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”. Finally the bloom filter filters the rest of the text and finds nothing.

2.5 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 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

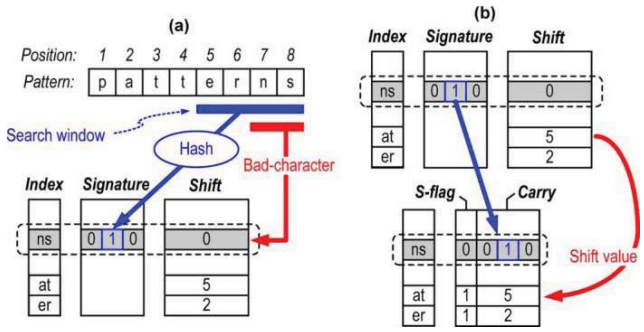


Fig. 5. Table generation and re-encoding of shift- signature algorithm. (a) Table generation; (b) table re- encoding.

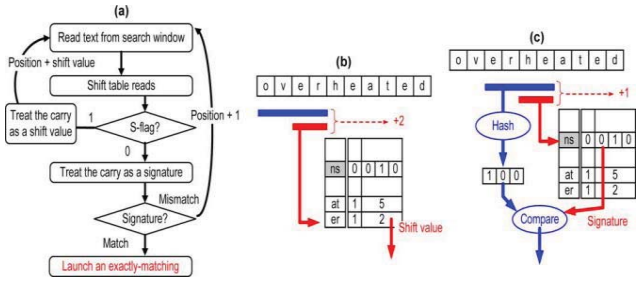


Fig. 6. Matching flow and filtering example. (a) Filtering flow; (b) shift filtering; (c) signature filtering.

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". 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). The *shift[ns]* 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.

The filtering flow is shown in Fig. 6(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. 6(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). 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.

2.6. Exact Match Engine (EME)

The EME must verify the false positives when the filtering engine alerts. It precisely identifies patterns for upper-layer applications. The AC algorithm uses loose tries, which check each input character in a constant amount of time because of their fan-out states for all possible input characters. Thus, the input data do not affect the AC-based algorithm's performance, but their memory requirements increase exponentially with pattern size. However, this method has potential performance problems because it may redundantly search link lists formed by sibling pointers. Despite this limitation, compact tries are still highly practical because, in practice, attack texts are not easy to generate. Attacks can be avoided by removing patterns that cause attacks before constructing the pattern database. For this reason, we use compact tries as our exact-matching engine's algorithm, and we propose several solutions to mitigate the effect of algorithmic attacks.

2.7 Exact-matching flow

Fig 8 shows the flow chart of the Exact-matching flow used in the exact matching engine. The processing steps are as given below.

Step 1) This engine fetches a piece of text from the text pump according to the address given by the filter engine.

Step 2) If this is the first reading of the trie table for this alarm, then this engine hashes this text to generate the root address of its trie tree. Otherwise, it chooses the sibling pointer of the trie node that the engine last read as the new address.

Step 3) This engine fetches the trie node from memory according to the address provided by the above step.

Step 4) The engine compares this piece of text with the trie node. If the content of the trie node is the same as the piece of text, it jumps to Step 6. Otherwise, the engine continues and checks whether this node has a sibling pointer.

Step 5) If a sibling exists, the engine jumps to Step 3 and fetches its sibling node, according to the pointer. Otherwise, it jumps to Step 7 to execute the trie-skip mechanism.

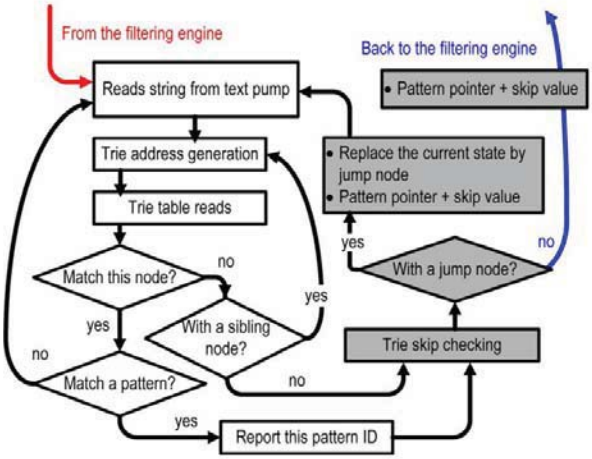


Fig. 7 Exact-Matching Flow.

Step 6) If a pattern exists at this node, the engine reports the pattern ID and goes to Step 7. Otherwise, it shifts the pattern pointer right and back to Step 1 to repeatedly examine the next piece of text.

Step 7) The pattern pointer shifts right several characters by the skip value. If the node has a jump node, the engine updates its state using this jump node and fixes its search window by the suffix offset. The engine then returns to Step 1. Otherwise, the engine finishes the verification and hands control back to the filtering engine.

Fig. 8. shows an example of exact matching. After the filtering engine launches an alert, the exact-matching engine gets a slice of the input text “ther” and hashes this text to generate the root address. The engine then reads the root node from memory compares it with the text and successfully matches the string at step 1. The exact-matching engine continues to compare the child node “eina”, indicated by the child pointer of the root node at step 2, but it mismatches its child node. However, the child node “eina” has a sibling node; thus, it keeps comparing its sibling node at step 3. The engine then mismatches the node “rule” with the text “seto” at step 4. The exact-matching engine then returns control to the filtering engine to find the next candidate position. Finally, the pattern matching matches pattern 3 at the tail of the text. Observing the matching flow of the exact-matching engine in Fig. 8, we notice that the filtering engine can only shift one character right to the next candidate position after the exact matching engine mismatches. This method may be vulnerable to algorithmic attacks. Just like the node “rule” in Fig. 8 the exact-matching engine cannot be sure where to jump when the input string is not “rule.” The engine cannot enter its failure state immediately when a mismatch occurs because the compact trie does not contain failure states for all possible input strings.

However, the engine can still jump to its failure state based on its previously matched nodes: “ther” and “eisa”. The following section describes cases for the failure state and how we implement it in the compact trie.

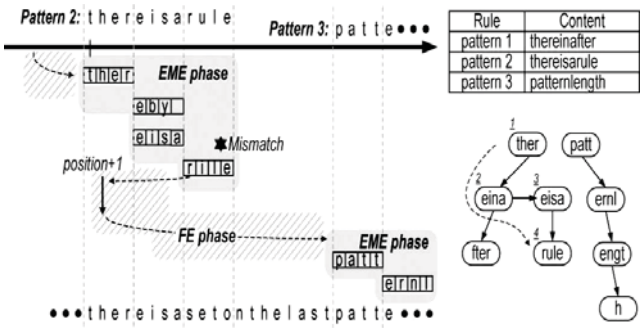
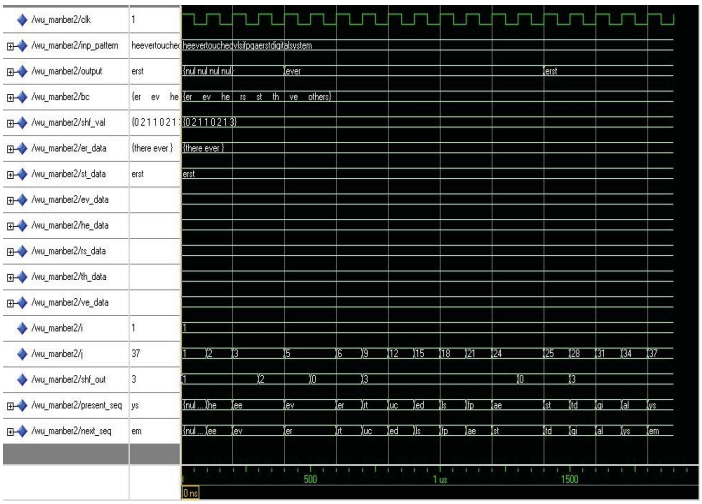


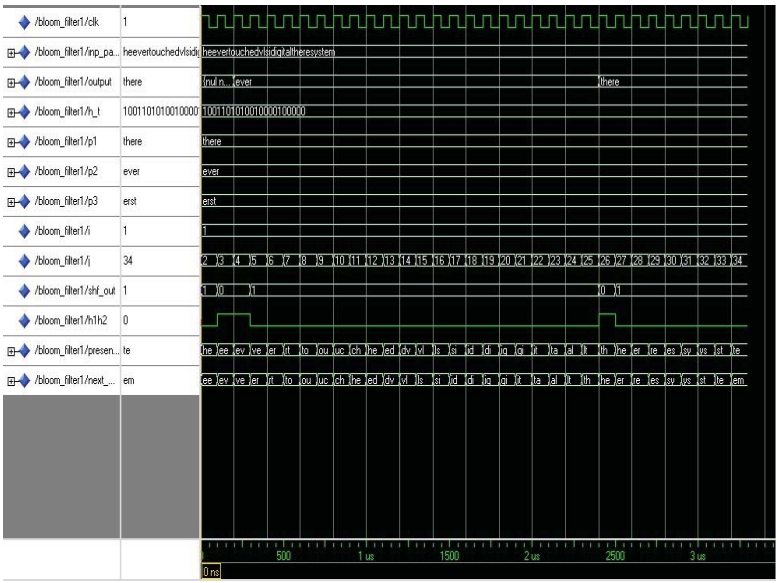
Fig. 8. Exact-Matching With A Multiple-Character Compact Trie

III. SIMULATION RESULTS

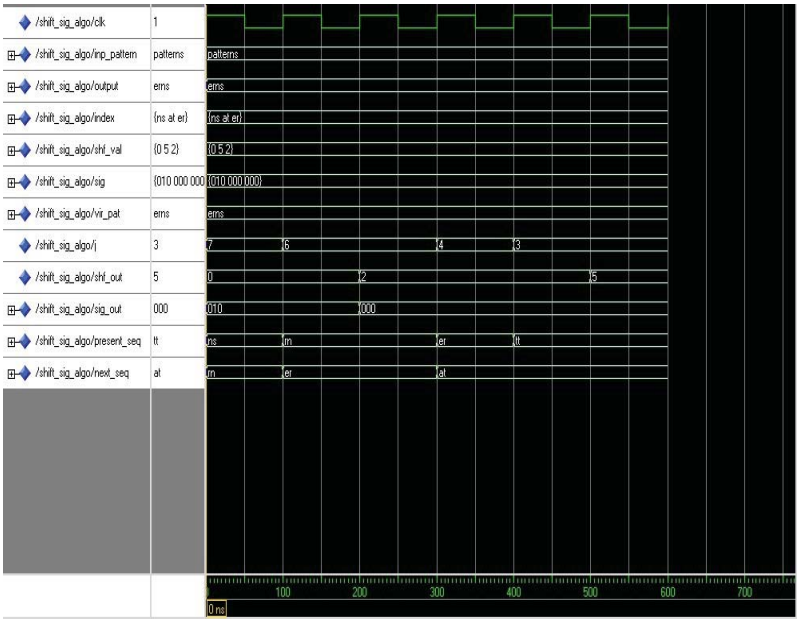
3.1. Wu-Manber algorithm



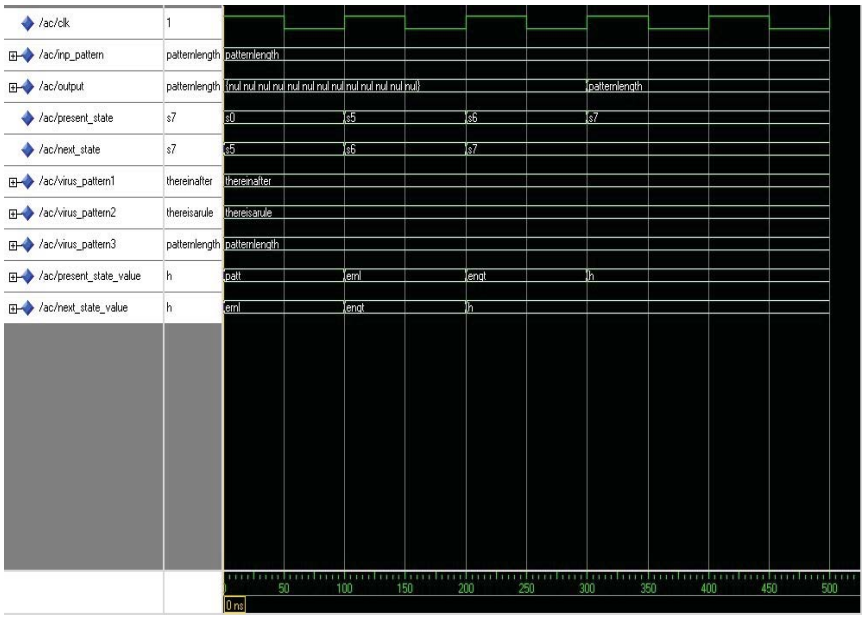
3.2. Bloom Filter algorithm



3.3. Shift-Signature Algorithm



3.4. AC algorithm



IV 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*" IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 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. Proce2008, 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 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.