Advanced Pattern Based Virus Detection Algorithm for Network Security

T.B. Binroy and B. Lakshmanan

Abstract——The ordinary network security applications require the ability to perform powerful pattern matching to protect against attacks such as viruses. In normal cases, 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 an antivirus with ever-large pattern sets. This work provides 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 entry 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 header (MAC address/IP address/port number). This method significantly reduces the probability of being attacked.

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 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 firewall 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.

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. 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.

T.B. Binroy, P.G. Scholak, ME Communication Systems, R.V.S. College of Engg & Technology, Coimbatore, India. E-mail:binroytb@gmail.com
B. Lakshmanan, Asst. Professor, Dept. of E.C.E, R.V.S. College of Engg & Technology, Coimbatore, India. E-mail:lakshmanan.rvs@gmail.com

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 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 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.

IV. FILTERING ENGINE (F.E)

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.

A. Wu-Manber Algorithm

This algorithm is a multi-pattern matching algorithm. It is based on the Boyer-Moore algorithm. In the pre processing stage it builds three tables: a shift table, a hash table and a prefix table. Here the shift table is an efficient filtering structure. The shift table is an extension of the bad character concept in the Boyer-Moore algorithm.

The matching flow of Wu-Manber algorithm 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. The shift table gives a shift value that is used to skip the several characters without comparing after a occurring 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. Its best performance is \( O(\text{BN}/m) \) 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 strongly dependent on the minimum length of the pattern in the pattern set. The maximum shift distance is calculated by the formula \( (m - B + 1) \).

For the pattern set \{erst, ever,there\} shown in Fig 3(d), the maximum shift value is 3 characters for \( B = 2 \) and \( m = 4 \). Fig. 3(b) and Fig. 3(c) show the related shift table, hash table and prefix. 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, but the search window is actually the character it fetches for comparison. According to \( \text{shift} [eV] = 2 \), 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 \( \text{shift}[e=0] \).

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 in step 4. The algorithm keeps shifting the search window until touching the end of the string in step 6.

B. Bloom Filter Algorithm

Bloom filter is a space-efficient data structure. It is used to check whether an element exists in a given set. This algorithm used in this Bloom filter is composed of \( m \) different hash functions and a long vector of \( n \) bits. At the pre processing stage all bits are set to 0. To add an element, the filter
hashes the element by these hash functions and gets positions of its vector. The 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. It also generates positions of the vector. If all of these bits are set to 1, this question is claimed to be positive, otherwise it is claimed to be negative. The output of the Bloom filter can no be a false negative.

Fig. 4(a) gives the process 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. 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.

C. Shift-Signature Algorithm

Both the candidate position and search window are aligned together. In the Shift-Signature Algorithm the shift table is merged with the signature table to form a new table named the shift-signature table. The new table has the same width and length as the shift table. There are two fields in the shift-signature table, S-flag and carry. The carry field contains a shift value and a signature. These two data types are used by two different algorithms. S-flag is used to indicate the data type of a carry. The FE can provide a higher filter rate with this new algorithm. The merging procedure for these two tables is described below. First generates a shift table and signature table at the pre-processing stage. S-flag is a 1-bit field and is used to indicate the data type of the carry.

Fig. 5(a) shows an example of generating the shift and signature tables. The length of the shortest pattern “patterns” in the pattern set is 8. The size of the bad-character is 2, 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”. 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[er] = 5 and shift[at] = 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.

V. EXACT MATCHING ENGINE (E.M.E)

The exact matching engine is used to verify the false positives. It identifies the virus pattern in higher layer applications.
Fig. 6: Matching flow and filtering example. (a) Filtering flow; (b) shift filtering; (c) signature filtering.

A. Exact-matching flow

Fig. 8 shows the flow chart of the matching flow used in the EME. The processing steps are as given below.

Step 1) According to the address given by the FE, EME takes the piece of text from the text pump.

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

Step 3) The engine takes the trie node from memory according to the address provided by the previous step.

Step 4) Then the engine compares this piece of text with the trie node.

Step 5) If the content of the trie node is the same as the piece of text, it jumps to Step 6. Otherwise, the EME continues and checks whether this node has a sibling pointer.

Step 6) If a sibling exists, the EME jumps to Step 3 and fetches its sibling node, according to the pointer. Otherwise, the EME jumps to Step 7 to execute the trie-skip mechanism.

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

Step 8) By using the skip value the pattern pointer shifts right several characters. If the node has a jump node, the EME 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 FE.

Fig. 8 given shows an example for the exact matching process. After the FE gives an alert, the EME gets a slice of the input text “there” and hashes this text to generate the root address. Engine then reads the root node from memory and compares it with the text and successfully matches the string at step 1. The EME continues to compare the child node “eina”, indicated by the child pointer of the root node at
B. Bloom Filter Algorithm

C. Shift-Signature Algorithm

D. AC Algorithm

VII. CONCLUSION

Many of existing designs can provide high performance. But with the increase in the size of virus data base decreases the performance because of the memory gap created by external memory. Furthermore, practically these algorithms have some restrictions for embedded network security due to the limitation in the resources. The two phase algorithm presented here, with its efficient filtering, is a tradeoff between the performance and cost. In this work first with the filtering section it provides a bad character heuristic feature and second with the skip mechanism provides a good performance with algorithmic attack.

REFERENCES


