A SCALABLE HIGH-PERFORMANCE VIRUS DETECTION PROCESSOR FOR EMBEDDED NETWORK SECURITY

By

KRISHNADEVI.S

Reg. No. 1020106011

of

KUMARAGURU COLLEGE OF TECHNOLOGY

(An Autonomous Institution affiliated to Anna University of Technology, Coimbatore)

COIMBATORE - 641049

A PROJECT REPORT

Submitted to the

FACULTY OF ELECTRONICS AND COMMUNICATION ENGINEERING

In partial fulfilment of the requirements

for the award of the degree

of

MASTER OF ENGINEERING

IN

APPLIED ELECTRONICS

APRIL 2012

BONAFIDE CERTIFICATE

Certified that this project report titled **"A SCALABLE HIGH-PERFORMANCE VIRUS DETECTION PROCESSOR FOR EMBEDDED NETWORK SECURITY"** is the bonafide work of **Ms. KRISHNADEVI.S [Reg.No:1020106011]** who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this for any other candidate.

Project Guide	**Head of the Department**
Ms.K.Thilagavathi	Dr. Rajeswari Mariappan

The candidate with University Register No. 1020106011 was examined by us in Project Viva-Voce examination held on _ _ _ _ _ _ _ _ _

Internal Examiner **External Examiner**

ACKNOWLEDGEMENT

First I would like to express my praise and gratitude to the Lord, who has showered his grace and blessing enabling me to complete this project in an excellent manner.

I express my sincere thanks to our beloved Director **Dr.J.Shanmugam, Ph.D.,** and beloved Principal **Dr.S.Ramachandran**, **Ph.D.,** Kumaraguru College of Technology, who encouraged me in each and every steps of the project work.

I would like to express my deep sense of gratitude to our HOD, **Dr.Rajeswari Mariappan**, **Ph.D.,** Department of Electronics and Communication Engineering, for her valuable suggestions and encouragement which paved way for the successful completion of the project work.

In particular, I wish to thank with everlasting gratitude to the project coordinator **Ms.R.Hemalatha, M.E., Assistant Professor(SRG),** Department of Electronics and Communication Engineering for the expert counseling and guidance to make this project to a great deal of success.

I am greatly privileged to extend my heartfelt thanks to my project guide **Ms.K.Thilagavathi, M.E., Assistant Professor(SRG),** Department of ECE, Kumaraguru College of Technology throughout the course of this project work and I wish to convey my deep sense of gratitude to all the teaching and non-teaching of ECE Department for their help and cooperation.

Finally, I thank my parents and my family members for giving me the moral support and abundant blessings in all of my activities and my dear friends who helped me to endure my difficult times with their unfailing support and warm wishes.

ABSTRACT

The main objective of this project is to provide high performance in most cases while still performing reasonably well in the worst case. With an eye towards high performance, updatability, unlimited pattern sets and low memory requirements, a two-phase architecture is introduced so that it uses off-chip memory to support a large pattern set. The goal of this project is to provide a systematic virus detection hardware solution for network security for embedded systems. Instead of placing entire matching patterns on a chip, a new solution is to provide 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 93.1% 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 impact of the memory gap, three enhancement algorithms are proposed to improve performance: 1) a skipping algorithm; 2) a cache method; and 3) a pre fetching mechanism.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

FPGA	-----	Field-Programmable Gate Arrays
ASIC	-----	Application Specific Integrated Circuits
SOC	-----	Silicon On Chip
AC	-----	Aho and Corasick’s algorithm
FSM	-----	Finite State Machine
EME	-----	Exact Match Engine
FE	-----	Filter Engine

CHAPTER 1

INTRODUCTION

Network security has always been an important issue. End users are vulnerable to virus attacks, spam, 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.

1.1 MOTIVATION

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. However, even under numerous security constrictions, firewall routers are still required to provide high-speed transmission. Therefore, a pattern matching processor is developed to accelerate the detection speed.

1.2 PROJECT GOAL

With an eye toward high performance, updatability, unlimited pattern sets and low memory requirements, a two-phase architecture is developed which uses off-chip memory to support a large pattern set.1) a shift-signature table and 2) a trie-skip mechanism is proposed to improve the performance and cushion the blow of the impact on memory gap for this two-phase architecture.

1.3 OVERVIEW

A two-phase pattern-matching architecture comprises of 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. Only a few unsaved data need to be checked precisely by the exact-matching engine in the second stage.

1.4 SOFTWARES USED

- Modelsim XE 111 6.2g
- Xilinx ISE 7.1i

1.5 ORGANIZATION OF THE REPORT

- **Chapter 2** discusses about the Firewall Router
- **Chapter 3** deals about the construction of Virus Detection Processor
- **Chapter 4** explains the Algorithms involved in Filtering Engine
- **Chapter 5** gives the step involved in Conventional AC Algorithm
- **Chapter 6** deals about the Modified AC Using Merged FSM
- **Chapter 7** discusses the simulation results
- **Chapter 8** shows the Conclusion and Future scope of the project.

CHAPTER 2

FIREWALL ROUTER

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.

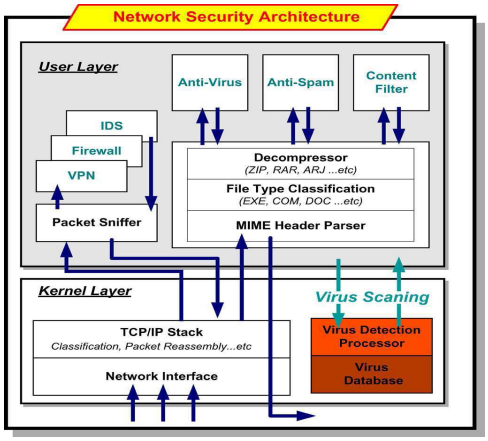


Figure 2 Architecture of firewall router

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 anti-virus programs. In summary, firewall routers require several time-consuming steps to provide a secure connection.

CHAPTER 3

VIRUS DETECTION PROCESSOR

Virus detection processor 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 a back-end module 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.

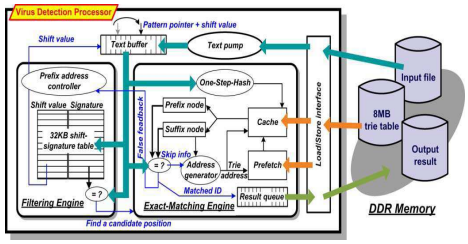


Figure 3.1 Virus detection processor architecture

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. Conversely, the exact-matching engine not only stores the entire pattern in external memory but also provides information to speed up the matching process. The 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.

The 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.

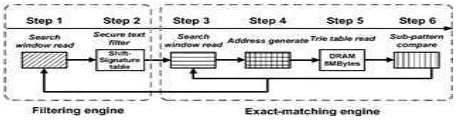


Figure 3.2 Two-Phase Execution Flow

Virus detection processor has six steps shown in Fig. 3 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 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 text and shifts the pattern pointer right multiple characters to continue to check the next position.

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.

After an alarm caused by the filtering engine, the exact-matching engine precisely verifies this alarm 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 text 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 processor then backs out to the filtering engine to search for the next candidate.

CHAPTER 4

FILTERING ENGINE (FE)

Filtering engine provides a high filter rate with limited space and also introduce two classical filtering algorithms for pattern matching in the following sections.

4.1 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.

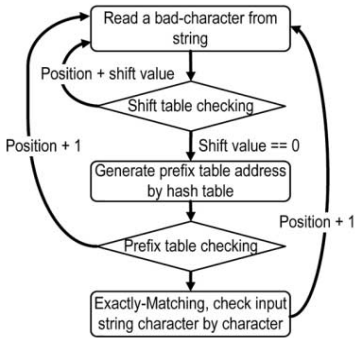


Figure 4.1 Flowchart for Wu-Manber Matching Process

The matching flow is shown in Fig. 4.1. 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 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.

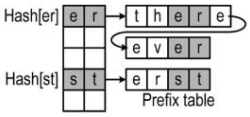


Figure 4.2 Shift Table for Wu-Manber Matching Process

Therefore, its best performance is for the given text with length and the pattern set, which has a minimum length of O(BN/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. 4.4, 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. 4.2 and Fig. 4.3. The Wu-Manber algorithm scans patterns from the head of a text, but it compares the tails of the shortest patterns.

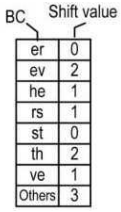


Figure 4.3 Hash table + Prefix table for Wu-Manber Matching Process

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

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 .Consequently, it checks the prefix table and hash table to perform an exact-matching and then outputs the “ever” in step 3.

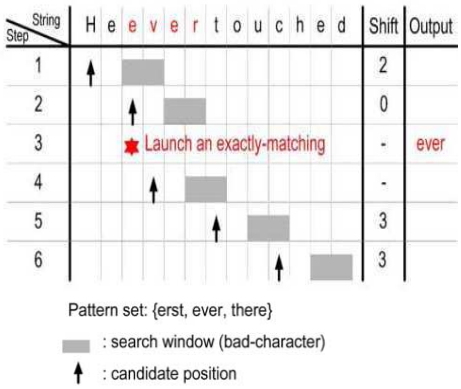


Figure 4.4 Wu-Manber Matching Process

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.

4.2 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 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 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; and 3) the Bloom filter is independent of its pattern length.

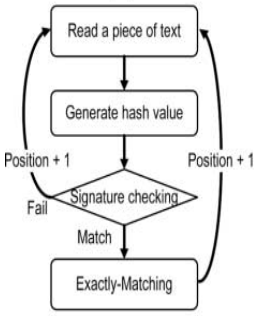


Figure 4.5 Flowchart for Bloom Filter Algorithm

Fig. 4.5 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.

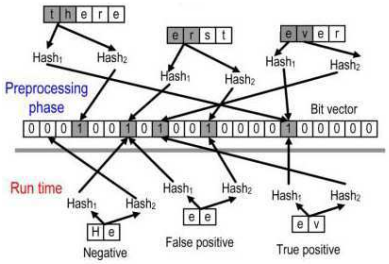


Figure 4.6 Bit-Vector Building for Bloom Filter Algorithm

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. 4.6 shows how a Bloom filter builds its bit vector for a pattern set {erst, ever, there} for two given hash functions.

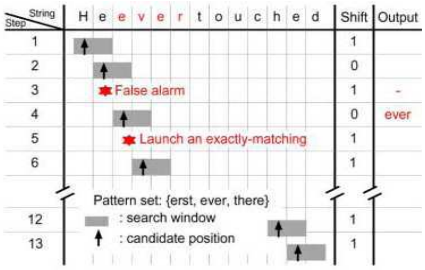


Figure 4.7 Bloom Filter Matching Process

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. 4.7 shows an example of the matching process. The arrows indicate the candidate positions.

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 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.

4.3 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.

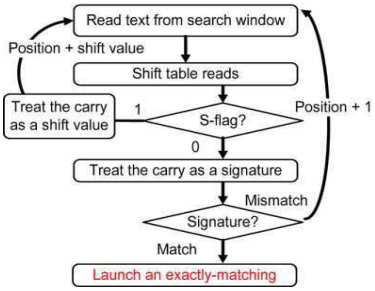


Figure 4.8 Flowchart for Shift Signature Matching Process

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 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. Generated signatures are mapped onto the signature table and indexed by bad-characters, which have shift values of zero in the shift table.

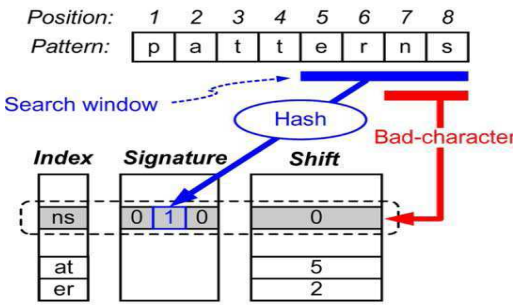


Figure 4.9 Table Generation for Shift-Signature Algorithm

In other words, a pattern is assigned a zero shift value in the shift table by its last characters, and it uses the same index to locate its signature in the signature table. 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. The S-flag is a 1-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.

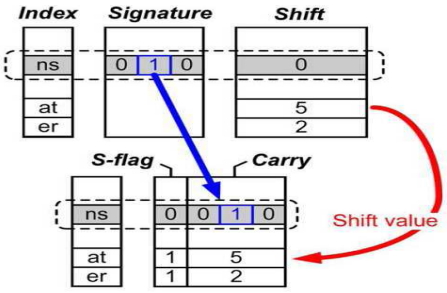


Figure 4.10 Table Re-encoding for shift-signature algorithm

Fig.4.9 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 instance 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 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, one hash function is to be used because of the space limitation of the signature table. The method of merging the shift table and signature table is shown in Fig. 4.10. The shift[ns] is replaced by its signature (“010” in binary) because its shift value is zero. In contrast, shift[at]=5 and shift[er]=2 the and keep their shift values in the shift-signature table.

For the pattern set {patterns}, Fig. 4.11 and Fig. 4.12 illustrate how the filtering engine filters out the given text. The filtering engine fetches the text from the search window as shown in Fig. 4.12. One part of the fetched text shown in Fig. 4.11 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.

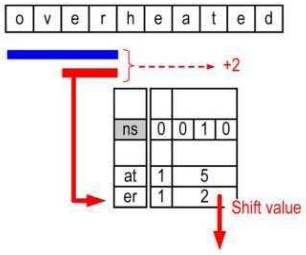


Figure 4.11 Shift Filtering

As a result, the filtering engine shifts the candidate position to the right by two characters for the text “overhead”, as shown in Fig. 4.11.

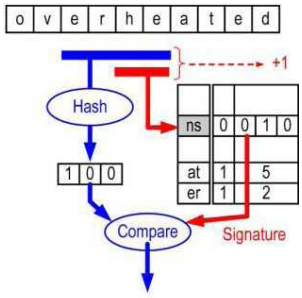


Figure 4.12 Signature filtering

The filtering engine hashes the fetched text and matches it with the signature read from the shift-signature table. Fig. 4.12 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.

CHAPTER 5

CONVENTIONAL AC ALGORITHM

5.1 EXACT MATCH ENGINE

The EME must verify the false positives when the filtering engine alerts. It also precisely identifies patterns for upper-layer applications. Most exact-match algorithms use the two kinds of trie structures namely loose and compact tries, to establish their pattern databases. Both trie structures have their merits. 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. Unlike loose tries, compact tries construct pattern databases with two pointers, sibling and child, to reduce their memory requirements.

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, compact tries are used in exact-matching engine’s algorithm, and several solutions were proposed to mitigate the effect of algorithmic attacks.

5.2 TRIE TABLE GENERATION AND ONE-STEP HASH

A traditional compact trie usually has only one entrance, as shown in Fig. 5.1. However, for multiple patterns, this method requires a significant amount of time to search the prefix node of a pattern in the entrance’s sibling list. To reduce search time, a huge trie is divided into several lightweight tries to generate multiple entrances by hashing the root node of each lightweight trie. The generated hash values are root addresses for each lightweight trie tree.

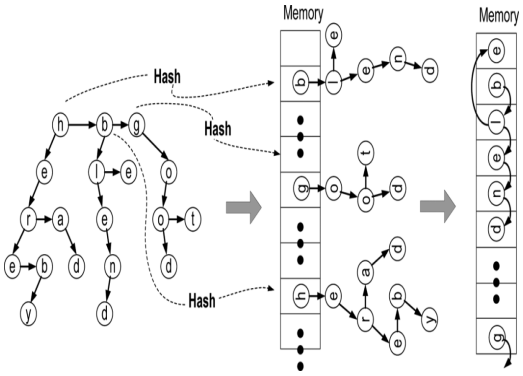


Figure 5.1 One-Step Hash For A Single-Root Problem

In this way, prefetching helps the exact-matching engine overlap computation with memory access time. Therefore, simply hashing the trie to generate multiple entrances and carefully arranging the data in the memory efficiently solves the memory gap problem at the algorithmic level.

5.3 EXACT-MATCHING FLOW

Fig. 5.2 illustrates the flow that exact-matching engine verifies an alarm triggered by the filtering engine.

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.

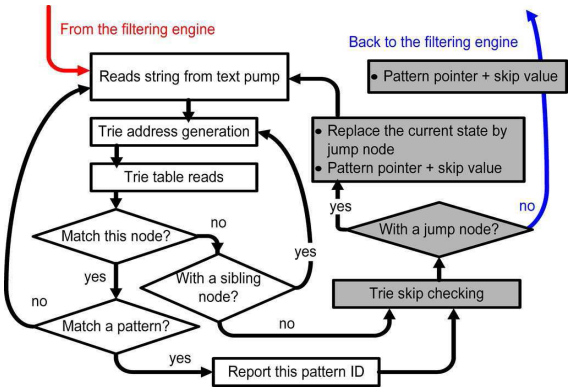


Figure 5.2 Exact-Matching Flow

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.

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.

5.4 EXACT MATCHING WITHOUT THE TRIE SKIP MECHANISM

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.

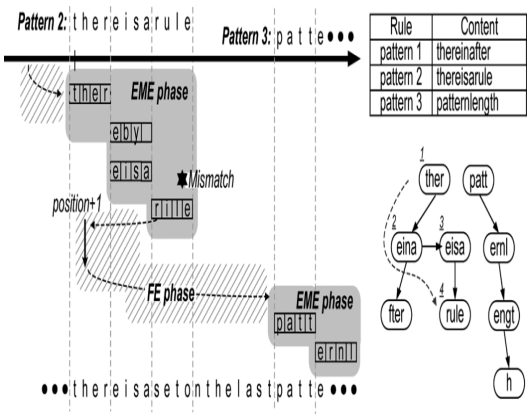


Figure 5.3 Exact-matching with a multiple-character compact trie

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. 5.2, 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. The concept of a failure state, an obvious solution to this problem, could not be implemented directly on the compact trie. Just like the node "rule" in Fig. 5.3, 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.

5.5 TRIE-SKIP MECHANISM FOR ALGORITHMIC ATTACK AVOIDANCE

There are two cases that trie-skip mechanism can remove redundant comparison. The first case, shown in Fig. 5.4 has two wanted patterns in its database. Pattern 2 does not exist in any part of pattern 1. When the exact-matching engine mismatches pattern 1, the filtering engine should restart to find the next candidate from the point where the exact matching engine mismatched because the string (nodes 1 and 2) that has already been compared cannot contain another pattern.

To implement this concept, our trie node contains a pre calculated skip value that lets the filtering engine find the next candidate position by skipping after a mismatch occurs. In

this case, the exact-matching engine can skip eight characters to search for the next candidate after a mismatch. The second case contains two wanted patterns such that one is a prefix of the other.

In this case, the exact-matching engine does not go back to the filtering engine after a mismatch occurs. Instead, the engine launches another exact match because the mismatched point might contain another pattern.

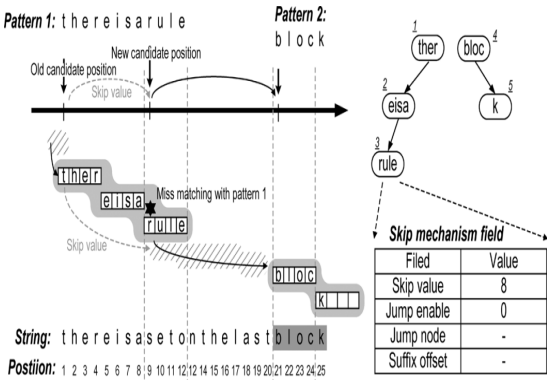


Figure 5.4 Skip Value of Skip Mechanism

The trie-skip mechanism is implemented by four major fields shown in Fig. 5.5 for each trie node. The skip value is eight, meaning that the closest candidate pattern is behind the current candidate by eight characters. However, because the exact matching engine does not always start from the beginning of a pattern, the jump node field indicates the first node that the exact-matching engine should compare for the new candidate pattern after a mismatch occurs.

The exact-matching engine can continue to compare from the node “ernl” of pattern 2 because the node “patt” of pattern 2 has already been checked. The suffix offset fixes the search window and notifies the exact-matching engine, which fetches characters behind the new pattern pointer with four characters. The jump-enable bit and jump node are used to implement this jumping idea.

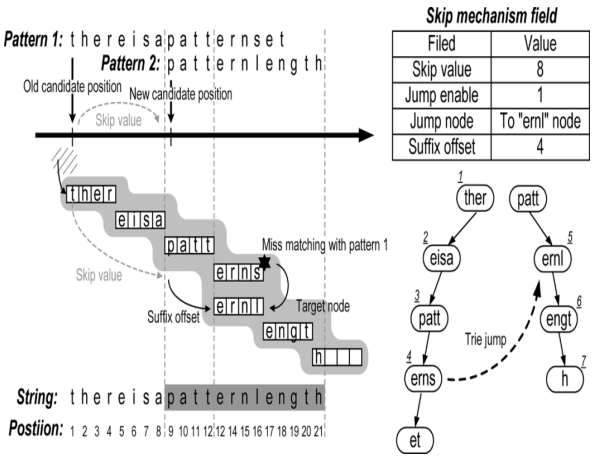


Figure 5.5 Jump node of skip mechanism

CHAPTER 6

MODIFIED AC USING MERGED FSM

Network intrusion detection system is used to inspect packet contents against thousands of predefined malicious or suspicious patterns. Among hardware approaches, memory-based architecture has attracted a lot of attention because of its easy re configurability and scalability. In order to accommodate the increasing number of attack patterns and meet the throughput requirement of networks, a successful network intrusion detection system must have a memory-efficient pattern-matching algorithm and hardware design. A memory-efficient pattern-matching algorithm is constructed using modified AC using merged FSM.

6.1 FSM ARCHITECTURE

.

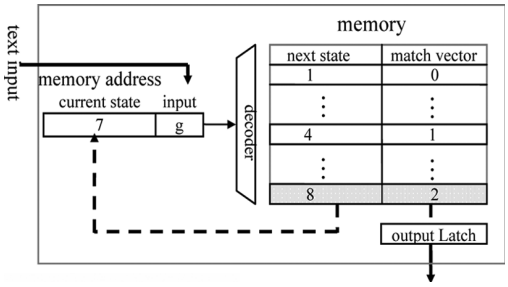


Figure 6.1 Basic FSM Memory Architecture

States 4 and 8 are the final states indicating the matching of string patterns “bcdf” and “pcdg”, respectively. Fig. 6.1 presents a simple memory architecture to implement the FSM. In the architecture, the memory address register consists of the current state and input character; the decoder Relay out. The basic memory architecture works as follows

First, the (attack) string patterns are compiled to a finite-state machine (FSM) whose output is asserted when any substring of input strings matches the string patterns. Then, the corresponding state transition table of the FSM is stored in memory.

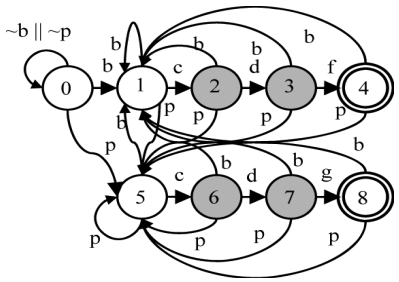


Figure 6.2 DFA for Matching “bcdf” and “pcdg”

For instance, Fig.6.2 shows the state transition graph of the FSM to match two string patterns “bcdf” and converts the memory address to the corresponding memory location, which stores the next state and the match vector information. A “0” in the match vector indicates that no “suspicious” pattern is matched; otherwise the value in the matched vector indicates which pattern is matched. Suppose the current state is 7 and the input character is g. The decoder will point to the memory location which stores the next state 8 and the match vector 2. Here, the match vector 2 indicates the pattern “pcdg” is matched.

6.2 MODIFIED AC ALGORITHM

Among all memory architectures, the AC algorithm has been widely adopted for string matching because the algorithm can effectively reduce the number of state transitions and therefore the memory size. Using the same example as in Figs 6.2 and 6.3 shows the state transition diagram derived from the AC algorithm where the solid lines represent the valid transitions while the dotted lines represent a new type of state transition called the failure transitions. The failure transition is explained as follows. Given a current state and an input character, the AC machine first checks whether there is a valid transition for the input character; otherwise, the machine jumps to the next state where the failure transition points.

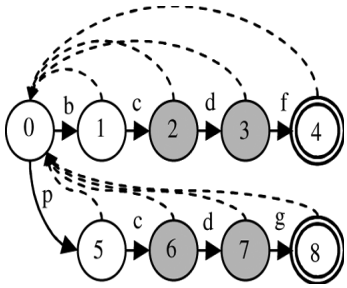


Figure 6.3 State diagram of an AC machine

Then, the machine recursively considers the same input character until the character causes a valid transition. Consider an example when an AC machine is in state 1and the input character is p. When there is no valid transition, the AC machine takes a failure transition back to state 0. Then in the next cycle, the AC machine reconsiders the same input character in state 0 and finds a valid transition to state 5. This example shows that an AC machine may take more than one cycle to process an input character.

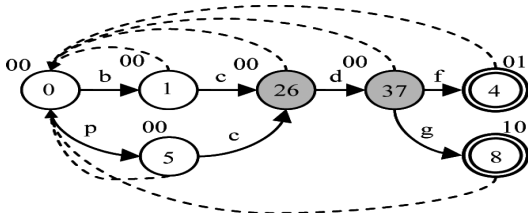


Figure 6.4 Merging Similar States

In Fig.6.3, the double-circled nodes indicate the final states of patterns. In Fig. 6.3, state 4, the final state of the first string pattern "bcdf", stores the match vector {P2P1}={01}and state 8, the final state of the second string pattern "pcdg", stores the match vector of {P2P1}={10}. Except the final states, the other states store the match vector {P2P1}={00} to simply express those states are not final states.

6.3 STATE TRAVERSAL MACHINE

Due to the common substrings of string patterns, the compiled AC machine may have states with similar transitions. Despite the similarity, those similar states are not equivalent states and cannot be merged directly. In this section, the functional errors can be created if those similar states are merged directly. Then, a mechanism is proposed to rectify those functional errors after merging those similar states. In Fig.6.3, states 2 and 6 are similar because they have identical input transitions, identical failure transitions to state 0.

Also, states 3 and 7 are similar. Note that merging similar states results in an erroneous state machine. As shown in Fig. 6.4, the state machine merges the similar states 2 and 6 to become state 26, and merges the similar states 3 and 7 to become state 37. Given an input string "pcdf", the original AC state machine shown in Fig. 6.2 moves from state 0, through state 5, state 6, state 7, and then takes a failure transition to state 0. On the other

hand, the merg_FSM moves from state 0, through state 5, state 26, state37, and finally reaches state 4 which indicates the final state of the pattern "bcdf". As a result of merging similar states, the input string "pcdf" is mistaken as a match of the pattern "bcdf".

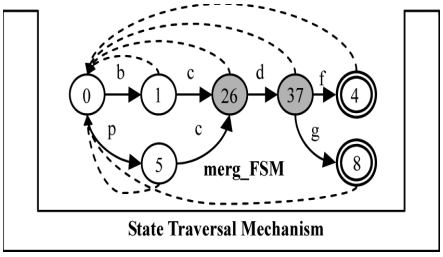


Figure 6.5 Architecture of the State Traversal Machine

This example shows the merg_FSM may causes false positive results. The merg_FSM is a different machine from the original state machine but with a smaller number of states and transitions. A direct implementation of merg_FSM has a smaller memory than the original state machine in the memory architecture. Our objective is to modify the AC algorithm so that we can store only the state transition table of merg_FSM in memory while the overall system still functions correctly as the original AC state machine does. The overall architecture of our state traversal machine is shown in Fig. 6.5. The new state traversal mechanism guides the state machine to traverse on the merg_FSM and provides correct results as the original AC state machine.

6.4 STATE TRAVERSAL MECHANISM ON A MERG_FSM

State 26 represents two different states (state 2 and state 6) and state 37 represents two different states (state 3 and state 7).We have shown that directly merging similar states leads to an erroneous state machine. To have a correct result, when state 26 is reached, a

mechanism is needed to understand in the original AC state machine whether it is state 2 or state 6. Similarly, when state 37 is reached, we need to know in the original AC state machine whether it is state 3 or state 7.

State 2 or state 6 will be differentiated by memorize the precedent state of state 26. If the precedent state of state 26 is state 1, we know that in the original AC state machine, it is state 2. On the other hand, if the precedent state of state 26 is state 5, the original is state 6. This example shows that how to memorize the precedent state entering the merged states by differentiating all the merged states. In the following section, how the precedent path vector can be retained during the state traversal in the merg_FSM is going to be discussed.

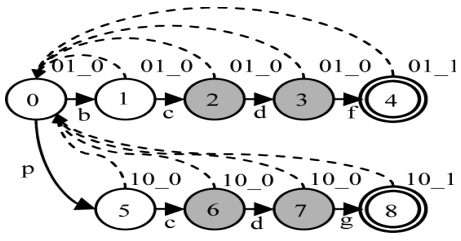


Figure 6.6 New data structure, PathVec and IfFinal

First of all, in a traditional AC state machine, a final state stores the corresponding match vector which is one-hot encoded is to be mentioned. In Fig. 6. 3, state 4, the final state of the first string pattern "bcdf", stores the match vector{P2P1}={01} and state 8, the final state of the second string pattern "pcdg", stores the match vector of {P2P1 }={10}.

Except for the final states, the other states store {P2P1} = {00} simply to express those states are not final states. One-hot encoding for a match vector is necessary because a final state may represent more than one matched string pattern. Therefore, the width of the match vector is equal to the number of string patterns. The majority of memories in the column "match vector" store the zero vectors {00} which are not efficient. In our design,

those memory spaces storing zero vectors {00} are used to store useful path information called pathVec. First, each bit of the pathVec corresponds to a string pattern.

Then, if there exists a path from the initial state to a final state, which matches a string pattern, the corresponding bit of the pathVec of the states on the path will be set to 1. Otherwise, they are set to 0. Consider the string pattern "bcdf" whose final state is state 4 in Fig. 6.6. The path from state 0, via states 1, 2, 3 to the final state 4 matches the first string pattern "bcdf". Therefore, the first bit of the pathVec of the states on the path, {state0, state 1, state 2, state 3, and state 4}, is set to 1. Similarly, the path from state 0, via states 5, 6, 7 to the final state 8 matches the second string pattern "pcdg".

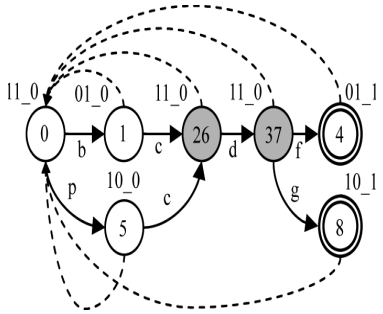


Figure 6.7 New State Diagram of Merg_FSM

Therefore, the second bit of the pathVec of the states on the path, {state 0, state 5, state 6, state 7, and state 8}, is set to 1. In addition, we add an additional bit, called ifFinal, to indicate whether the state is a final state. For example, because states 4 and 8 are final states, the ifFinal bits of states 4 and 8 are set to 1, the others are set to 0. As shown in Fig. 6.6, each state stores the pathVec and ifFinal as the form, "pathVec_ifFinal". Compared with the original AC state machine in Fig. 6.3, we only add an additional bit to each state.

States 2 and 6, states 3 and 7 are similar because they have similar transitions. However, they are not equivalent. Note that two states are equivalent if and only if their next states are equivalent. In Fig. 6.6, states 3 and 7 are similar but not equivalent because for the same input F, state 3 takes a transition to state 4 while state 7 takes a failure transition to state 0. Similarly, state 2 and state 6 are not equivalent states because their next states, state 3 and state 7, are not equivalent states.

6.5 PSEUDOEQUIVALENT STATES

Two states are defined as pseudo-equivalent states if they have identical input transitions, identical failure transitions, and identical ifFinal bit, but different next states. States 2 and 6 are pseudo-equivalent states because they have identical input transitions ,identical failure transitions to state 0 and identical ifFinal bit 0. Also, state3 and state 7 are pseudo-equivalent states.

In modified AC algorithm, the pseudo-equivalent states 2 and 6 are merged to be state 26 and states 3 and 7 are merged to be state 37, as shown in Fig. 6.7. The pathVec_ifFinal are updated by taking the union on the pathVec_ifFinal of the merged states. Therefore, the pathVec_ifFinal of states 26 and 37 are modified to be {11_0}.In addition, register is needed, called preReg, to trace the precedent pathVec in each state. The width of preReg is equal to the width of pathVec.

State	0	5	26	37	0
Input Char	P	C	D	F	F
Path Vec	11	10	11	11	01
Pre Reg	11	10	10	10	00
IF final	0	0	0	0	1

Table 6.1 State Transitions of the Input String "pcdf"

The preReg is updated in each state by performing a bitwise AND operation on the pathVec of the next state and its current value. By tracing the precedent path entering into the merged state, we can differentiate all merged states. When the final state is reached, the value

of the preReg indicates the match vector of the matched pattern. During the state traversal, if all the bits of the preReg become 0, the machine will go to the failure mode and choose the failure transition as in the AC algorithm. After any failure transition, all the bits of the preReg are reset to 1.

Consider an example in Table1 where the string "pcdf" is applied. Initially, in state 0, the preReg is initialized to{P2P1}={11}. After taking the input character , the merg_FSM goes to state 5 and updates the preReg by performing a bitwise AND operation on the pathVec {10} of state 5 and the current preReg {11}.

The resulting new value of the preReg will be {P2P1}={10 AND 11}={10}. Then, after taking the input character , the merg_FSM goes to state 26 and updates the preReg by performing a bitwise AND operation on the pathVec {11} of state 26 and the current preReg {10}. The preReg remains {P2P1}={11AND 10}={10}. Further, after taking the input character, the merg_FSM goes to state 37 and updates the preReg by performing a bitwise AND operation on the pathVec {11} of state 37 and the current preReg {10}.

The preReg remains {P2P1}={11AND 10}={10}. Finally, after taking the input character , the merg_FSM goes to state 4. After performing a bitwise AND operation on the pathVec {01} of state 4 and the current preReg {10}, the preReg becomes{P2P1}={11AND10}={10} . According to AC algorithm, during the state traversal, if all the bits of the preReg become 0, the machine will go to the failure mode and choose the failure transition as in the AC algorithm.

State	0	5	26	37	0
Input Char	P	C	D	G	G
Path Vec	11	10	11	11	01
Pre Reg	11	10	10	10	00
IF final	0	0	0	0	1

Table 6.2 State Transitions of the Input String "pcdg"

Similarly, consider another example in table 6.2 where the string "pcdg" is applied. The process of state traversal is similar to the previous example until the state machine reaches state 37 and the input character is g . After taking the input character , the merg_FSM goes to state 8 and the preReg becomes by performing a bitwise AND operation on the pathVec {10} of state 8 and the current preReg {10}. Because the value of ifFinal is 1, the value of preReg, indicates the pattern is matched.

6.6 CONSTRUCTION OF STATE TRAVERSAL MACHINE

The construction of a state traversal machine consists of: 1) the construction of valid transition, failure transition, pathVec, and ifFinal functions and 2) merging pseudo-equivalent states. For a set of string patterns, a graph is created for the valid transition function. The creation of the graph starts at an initial state 0. Then, each string pattern is inserted into the graph by adding a directed path from initial state 0 to a final state where the path terminates. Therefore, there is a path, from initial state 0 to a final state, which matches the corresponding string pattern. For example, consider the three patterns, "abcdef", "apcdeg" and "awcdeh".

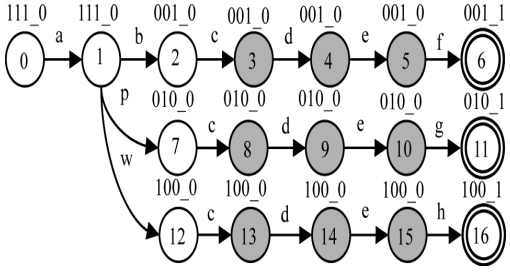


Figure 6.8 Construction of pathVec and ifFinal

When the pattern “apcdeg” is added to the graph, because there is already an edge labeled from state 0 to state 0 and 1 is set to {P3P2P1}={011} and the pathVec of other states, {state 7, state 8, state 9, state 10, state 11} on the path is set to {P3P2P1}={010}.Furthermore, the ifFinal of state 11 is set to 1 to indicate the final state for the second pattern. Similarly, when the third pattern “awcdeh” is added to the graph, the edge labeled from state 0 to state 1 is also reused. Therefore, the pathVec of states 0 and 1 is set to {P3P2P1}={111}. The pathVec of other states {state 12, state 13, state 14, state 15, and state 16} on the path is set to{P3P2P1}={000} . The ifFinal of state 16 is set to 1 to indicate the final state of the third pattern. Finally, Fig. 6.8 shows the directed graph consisting only of valid transitions.

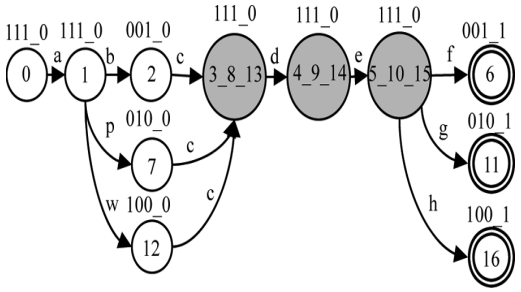


Figure 6.9 State Diagram of the State Traversal Machine

Consider the same example as shown in Fig. 6.8. States 3, 8, and 13 are pseudo-equivalent states because they have identical input transitions, identical failure transitions to state 0 and identical ifFinal 0. Similarly, states 4, 9, and 14 are pseudo-equivalent states and states 5, 10, and 15 are pseudo-equivalent states. As shown in Fig. 15, those pseudo-equivalent states are merged to states 3_8_13, 4_9_14 and 5_10_15, respectively. The pathVec of state 3_8_13 is modified to be by performing the union on the pathVec of state 3, state 8, and state 13. Similarly, the pathVec of states 4_9_14 and 5_10_15 is also modified to be {111}. Fig. 6.9 shows the final state diagram of our state traversal machine. Compared with the original AC state machine in Fig. 6.8, six states are eliminated.

6.7 LOOP BACK IN MERGED STATES

When certain cases of multiple sections of pseudo-equivalent states are merged, it may create loop back problem in a state machine. The reason for the loop back problem comes from merging common sub-patterns with different sequences. For example, the two patterns, “abcdef” and “wdebcg,” have common sub-patterns, “bc” and “de,” which appear in different sequences. Fig. 6.8 shows the corresponding state machine. Because of the common sub-patterns, “bc”, states 2 and 10, states 3 and 11 are pseudo-equivalent states. And, because of the common sub-patterns, “de”, states 4 and 8, states 5 and 9 are also pseudo-equivalent states. Merging the pseudo-equivalent states will create a loop back transition from state 5 to state 2, as shown in Fig. 6.9. The loop transition may cause false positive matching results. In other words, as long as the common substrings appear in sequence, merging the corresponding pseudo-equivalent states will not result in loop back transitions

CHAPTER 7

7.1 SIMULATION RESULTS

The simulation of this project has been done using ModelSim XE III 6.2g and Xilinx ISE 9.1i. ModelSim is a simulation tool for programming {VLSI} {ASIC}s, {FPGA}s, {CPLD}s, and {SoC}s. Modelsim provides a comprehensive simulation and debug environment for complex ASIC and FPGA designs. Support is provided for multiple languages including Verilog, SystemVerilog, VHDL and SystemC. The following diagram shows the basic steps for simulating a design in ModelSim.

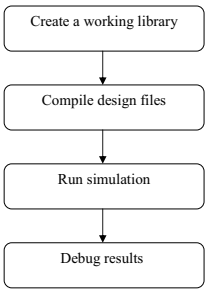


Figure 7.1 Basic Steps of Simulation in Model Sim

In ModelSim, all designs, be they VHDL, Verilog, or some combination thereof, are compiled into a library. We typically start a new simulation in ModelSim by creating a working library called "work". "Work" is the library name used by the compiler as the default destination for compiled design units. After creating the working library, we’ve to compile your design units into it. The ModelSim library format is compatible across all supported platforms. We can simulate our design on any platform without having to recompile design. With the design compiled, invoke the simulator on a top-level module (Verilog) or a configuration or entity/architecture pair (VHDL). Assuming the design loads

successfully, the simulation time is set to zero, and enter a run command to begin simulation.

Xilinx was founded in 1984 by two semiconductor engineers, Ross Freeman and Bernard Vonderschmitt, who were both working for integrated circuit and solid-state device manufacturer Zilog Corp. The Virtex-II Pro, Virtex-4, Virtex-5, and Virtex-6 FPGA families are particularly focused on system-on-chip (SOC) designers because they include up to two embedded IBM PowerPC cores The ISE Design Suite is the central electronic design automation (EDA) product family sold by Xilinx. The ISE Design Suite features include design entry and synthesis supporting Verilog or VHDL, place-and-route (PAR), completed verification and debug using Chip Scope Pro tools, and creation of the bit files that are used to configure the chip .Xilinx is a synthesis tool which converts schematic/HDL design entry into functionally equivalent logic gates on Xilinx FPGA, with optimized speed & area. So, after specifying behavioral description for HDL, the designer merely has to select the library and specify optimization criteria; and Xilinx synthesis tool determines the net list to meet the specification; which is then converted into bit-file to be loaded onto FPGA PROM . Also, Xilinx tool generates post-process simulation model after every implementation step, which is used to functionally verify generated net list after processes, like map, place & route.

7.1.1 WU-MANBER ALGORITHM

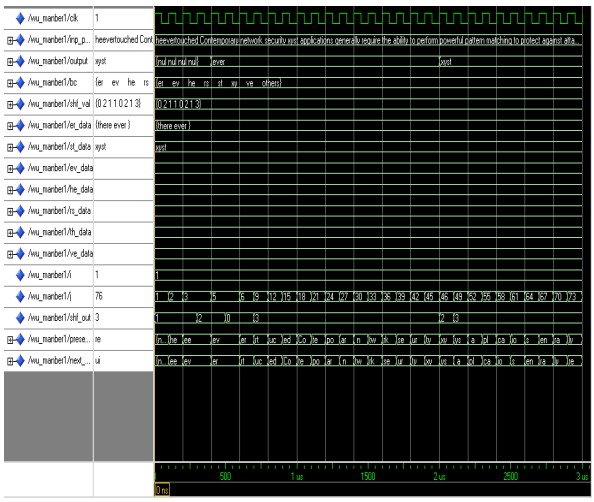


Figure 7.1.1 Simulation Result of Wu-Manber Algorithm

7.1.2 BLOOM FILTER ALGORITHM

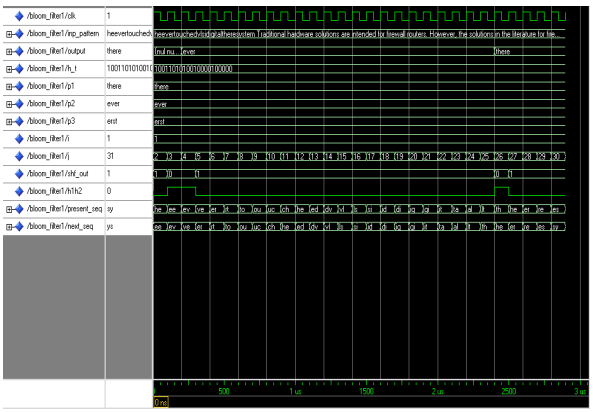


Figure 7.1.2 Simulation Result of Bloom Filter Algorithm

7.1.3 SHIFT SIGNATURE ALGORITHM

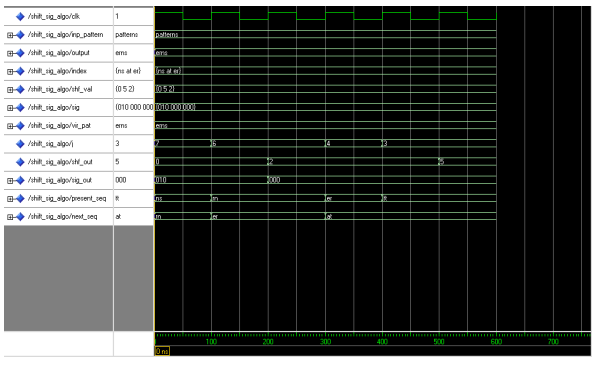


Figure 7.1.3 Simulation Result of Shift Signature Algorithm

7.1.4 CONVENTIONAL AC ALGORITHM

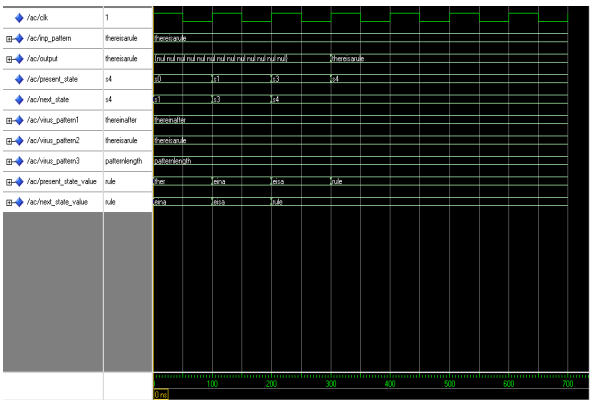


Figure 7.1.4 Simulation Result of Conventional AC Algorithm

7.1.5 MERGED FSM

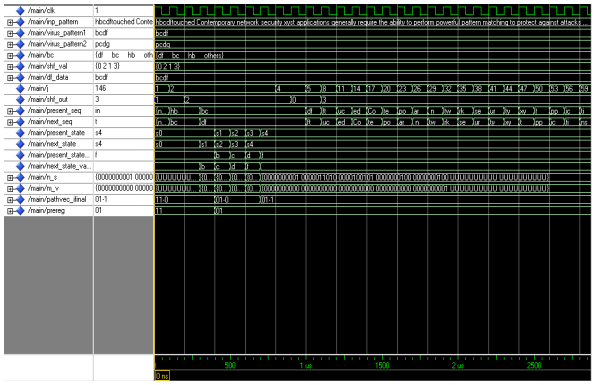


Figure 7.1.5 Simulation Result of Merged FSM

7.1.6 MODIFIED AC USING MERGED FSM

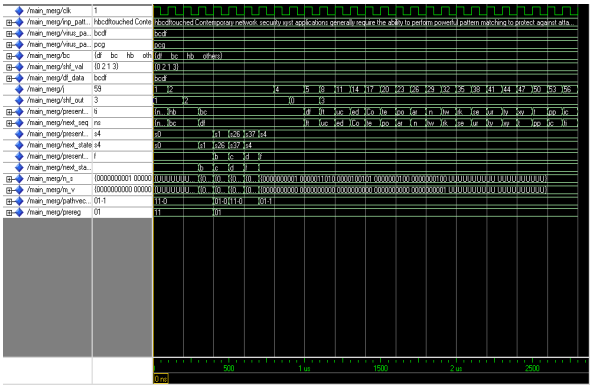


Figure 7.1.6 Simulation Result of Modified AC using Merged FSM

7.2 SYNTHESIS REPORT

XC2S200E-FG456:

7.2.1 Conventional AC Algorithm

Timing Summary:

Speed Grade: -7

Minimum period: 2.987ns (Maximum Frequency: 334.784MHz)

Minimum input arrival time before clock: 4.926ns

Maximum output required time after clock: 6.347ns

Maximum combinational path delay: No path found

Design Summary:

Number of errors: 0

Number of warnings: 2

Logic Utilization:

Number of Slice Flip Flops: 16 out of 4,704 1%

Number of 4 input LUTs: 52 out of 4,704 1%

Logic Distribution:

Number of occupied Slices: 28 out of 2,352 1%

Number of Slices containing only related logic: 28 out of 28 100%

Number of Slices containing unrelated logic: 0 out of 28 0%

Total Number of 4 input LUTs: 52 out of 4,704 1%

Number of bonded IOBs: 64 out of 285 22%

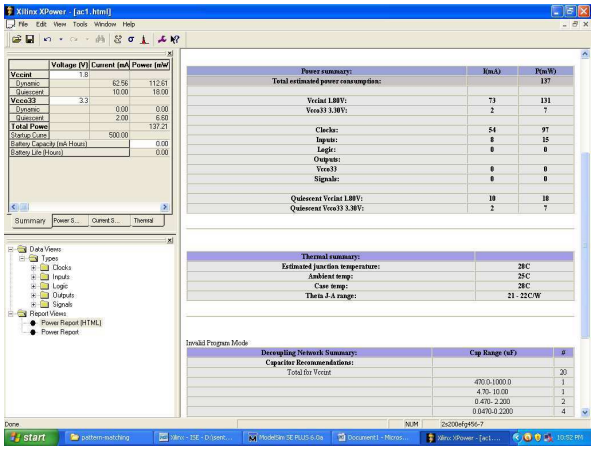
Number of GCLKs: 1 out of 4 25%

Number of GCLKIOBs: 1 out of 4 25%

Total equivalent gate count for design: 440

Additional JTAG gate count for IOBs: 3,120

7.2 .2 Power Report of Conventional AC Algorithm



7.2.3 Modified AC using Merged FSM

Timing Summary:

Speed Grade: -7

Minimum period: 2.729ns (Maximum Frequency: 366.434MHz)

Minimum input arrival time before clock: 4.926ns

Maximum output required time after clock: 6.347ns

Maximum combinational path delay: No path found

Design Summary:

Number of errors: 0

Number of warnings: 2

Logic Utilization:

Number of Slice Flip Flops: 8 out of 4,704 1%

Number of 4 input LUTs: 26 out of 4,704 1%

Logic Distribution:

Number of occupied Slices: 14 out of 2,352 1%

Number of Slices containing only related logic: 14 out of 14 100%

Number of Slices containing unrelated logic: 0 out of 14 0%

Total Number of 4 input LUTs: 26 out of 4,704 1%

Number of bonded IOBs: 32 out of 285 11%

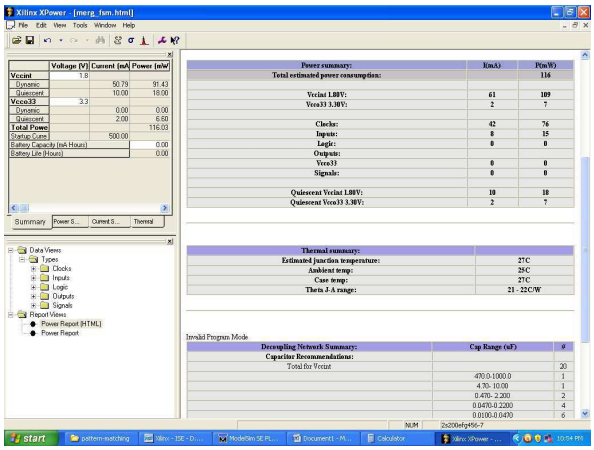
Number of GCLKs: 1 out of 4 25%

Number of GCLKIOBs: 1 out of 4 25%

Total equivalent gate count for design: 220

Additional JTAG gate count for IOBs: 1,584

7.2 .2 Power Report of Modified AC using Merged FSM



7.3 Comparison of the Conventional AC Algorithm and Modified AC using Merged FSM

Parameters	Conventional AC Algorithm	Modified AC using Merged FSM
Minimum period	2.978ns	2.729ns
Number of 4 input LUTs:	52 out of 4,704	26 out of 4,704
Total equivalent gate count for design	440	220
Total Estimated Power Consumption	136mw	116mw

Table 7.3.1 Conventional AC Algorithm Vs Modified AC using Merged FSM

CHAPTER 8

CONCLUSION AND FUTURE SCOPE

CONCLUSION

In this project, a two-phase architecture comprises of the filtering engine and the exact matching engine is developed using signature and skipping algorithms which provides the matching mechanism scalable to large pattern sets. The filtering engine in the front-end module is responsible for filtering out secure data efficiently and indicates to candidate positions that patterns possibly exist at the first stage. The exact-matching engine in the back-end module which uses AC Algorithm is responsible for verifying the alarms caused by the filtering engine. Conventional AC Algorithm used in the exact-matching engine removes the unnecessary memory accesses by Trie Skip Mechanism and it has moderate detection speed. The proposed AC Algorithm namely Modified AC Using Merged FSM merges two pseudo-equivalent states to a single state hence the detection speed is accelerated faster than that of the Conventional AC Algorithm. Thus Modified AC Using Merged FSM provides high performance, good storage efficiency, low power consumption and low memory requirement.

FUTURE SCOPE

The versatile architectures like dual-port match-line scheme can be designed and implemented in FPGA which reduces power consumption and increase storage efficiency due to shared storage spaces. In addition, it provides high flexibility for regularly updating the virus-database. Further number of techniques to approximate memory congestion state shuffling, state replication, alphabet shuffling and state caching can be analyzed to achieve the optimal performance.

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” in IEEE VLSI transactions on vol.20, pp. 841-854,2011.

[2]Cheng-Hung Lin and Shih-Chieh “Efficient Pattern Matching Algorithm for Memory Architecture” in IEEE VLSI transactions on vol. 19,pp.33-41,2011.

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

[4]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.

[5]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.

[6] M. Alicherry, M. Muthuprasanna, and V. Kumar, “High speed pattern matching for network IDS/IPS,” in Proc. IEEE Int. Conf. Netw. Protocols(ICNP), 2006, pp. 187–196.

[7] B. Brodie, R. Cytron, and D. Taylor, “A scalable architecture for high-throughput regular-expression pattern matching,” in Proc. 33rd Int. Symp. Comput. Arch. (ISCA), 2006, pp. 191–122.

[8] M. Aldwairi, T. Conte, and P. Franzon, “Configurable string matching hardware for speeding up intrusion detection,” Proc. ACM SIGARCH Comput. Arch. News, vol. 33, no. 1, pp. 99–107, 2005.

[9]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.

[10]. 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.

[11].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.

[12].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.

[13].R. S. Boyer and J. S. Moore, “A Fast String Searching Algorithm,”Commun. ACM, vol. 20, pp. 762–772, 1977.

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

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