Increased Speed for Network Security through Multi-Character Processing

Lavanya, Saranya, Uma Maheshwari

Abstract- A clear trend that can be observed in the Internet is the increasing amount of packet data that is being inspected before a packet is delivered to its destination. More recently, Network Intrusion Detection Systems (NIDS), virus scanners, spam filters and other content-aware applications go one step further by also performing scans on the packet payload. Pattern matching algorithm is used in Network Intrusion Detection System (NIDS). The system is used to detect network attacks by identifying attack patterns. This paper proposes a memory-efficient pattern matching algorithm which can significantly reduce the number of states and transitions by merging pseudo-equivalent states while maintaining correctness of string matching. Pattern matching is achieved through Aho-Corasick (AC) algorithm. By comparison result we say our matching algorithm is memory efficient than previously proposed method. As an extension of our work, through Multi-character processing, sufficient speed has been increased. The reduction in clock cycle indirectly increases speed in operation.The architecture is coded in VHDL and simulated using Modelsim and Xilinx.

Keywords- Aho-Corasick (AC) algorithm, Finite State Machine (FSM), Non-Deterministic Automation (NFA).

I. INTRODUCTION

Network intrusion detection systems (NIDS) are an important part of any network security architecture. They provide a layer of defense which monitors network traffic for predefined suspicious activity or patterns, and alert system administrators when potential hostile traffic is detected. Commercial NIDS have many differences, but Information Systems departments must face the commonalities that they share such as significant system footprint, complex deployment and high monetary cost.

The growth of internet in the last decade and society increasing dependence on it has bought along a flood of security attacks on networking and computing infrastructure. Intrusion detection /prevention systems provide defences against these attacks by monitoring headers and payload of the packets flowing through the network. Multiple strings matching that compare hundreds of string pattern simultaneously, which is a critical component of these systems. Most of the string matching solutions today are based on the classic Aho-Corosick (AC) algorithm.

The string matching algorithms have been traditionally used in many applications like word processing, search and replace operations, bibliographic search etc.

Manuscript published on 30 February 2013.
***** Correspondence Author (s)

Lavanya, ECE Department, Deemed University, Avinashilingam Institute for Home Science and Higher Education for Women University, Coimbatore, India.
Saranya, SE Department, Sir Ramakrishna Engineering College, Coimbatore, India.
Uma Maheshwari, CSE Department, Sir Ramakrishna Engineering College, Coimbatore, India.

They are also increasingly used for IP lookup in router, virus/worm detection using signature matching, network monitoring for packet filtering etc.

Researchers have proposed various software and hardware based solutions in tackling the string matching problem. The classic Aho-Corosick algorithm was the first technique proposed and constructs a Finite State Machine (FSM) to do multiple string matching. Main objectives in this paper,

- ✓ Modification of classic AC algorithm.
- ✓ Through the concept of merging pseudo equivalent states, we can able to reduce number of states and transitions.
- ✓ Increasing speed through multi-character processing.

II. RELATED WORK

The task of pattern matching in network application consists of comparing large number of static patterns against a high speed stream of data. String matching is the most competitive task in Network Intrusion Detection System (NIDS). Many hardware approaches are proposed. They are mainly classified into logic and memory architecture.

An idea of regular expression matching using Field Programmable Gate Array (FPGA) was given in [1]. The algorithm proposed here based on NFA (Non-deterministic Finite Automation). NFA technique is used to provide improved area and throughput by adding pre-coded wide parallel inputs [2]. A pre-decoded multiple pipeline shift was presented and matches are compared with reduced routing complexity and comparator size by converting incoming character into many bit lines [3].

Due to complexity of programming, the conventional algorithms for constructing finite automata from regular expression, we are going for using finite state machines in pattern matching applications. From these techniques state minimization can be possibly done. AC algorithm is the most popular algorithm which allows for matching multiple string patterns. The pattern matching scheme described [4], is suited for application having occurrences of large number of keywords in text strings.

Architecture for pattern matching co-processor for network intrusion system was described in [5]. Architecture for programmable parallel pattern matching co-processor has been presented in [6]. The AC algorithm is modified to consider multiple characters at a time [7]. In [8] an architecture that solves the more general problem of regular expression pattern matching with high throughput was described.

Checking every byte of every packet to see if it matches one of a set of ten thousand strings becomes a computationally intensive task as network speeds grow into the tens, and eventually hundreds, of gigabits/second.

In [9] L. Tan and T. Sherwood, have developed an approach that relies on a special purpose architecture that executes novel string matching algorithms specially optimized for implementation in their design. A new algorithm called bit-split algorithm was proposed here. In [10], author has presented a state encoding scheme called a covered state encoding. This is used for efficient Ternary Content Addressable Memory (TCAM)-based implementation of the Aho-Corasick multi-pattern matching algorithm, which is widely used in network intrusion detection systems.

III. AC ALGORITHM

The AC algorithm [4] was proposed in 1975 and remains, to this day, one of the most effective pattern matching algorithms when matching patterns sets. The AC algorithm is a very flexible and efficient matching algorithm that can scan the existence of a query string among multiple test strings looking at each character exactly once, making it one of the main options for software-based intrusion detection systems such as SNORT.

The algorithm consists of two parts. In the first part a finite state pattern matching machine is constructed from the set of keywords: in the second part text string is applied as input to the pattern matching machine. The machine signals whenever it has found a match for a keyword.

First, the string (attack) patterns are complied with FSM. The output is asserted when any substring of input string matches the string pattern. The corresponding state transition table of FSM is stored in memory.

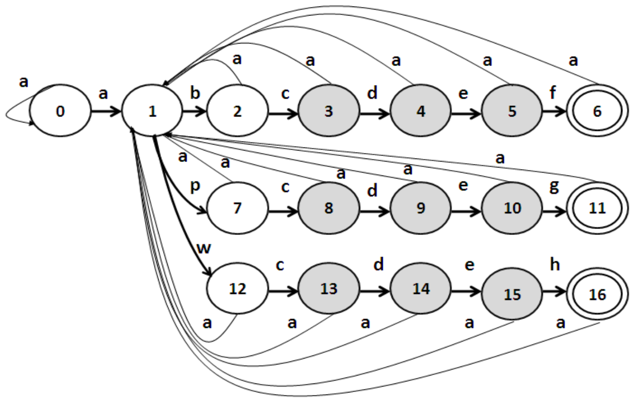

Fig.1 State Transition Graph

Fig.1 shows the state transition graph of the FSM to match three strings “abcdef”, “apcdeg” and “awcdeh”. The states given in double circle indicates final states. States 6, 11 and 16 are the final states indicating the matching of string patterns “abcdef”, “apcdeg” and “awcdeh”. Fig.2 represents memory architecture to implement the FSM. The memory address register consists of current state and input character. The decoder converts memory architecture to corresponding memory location. This stores next state and match vector. A ‘0’ in match vector indicates no pattern is matched.

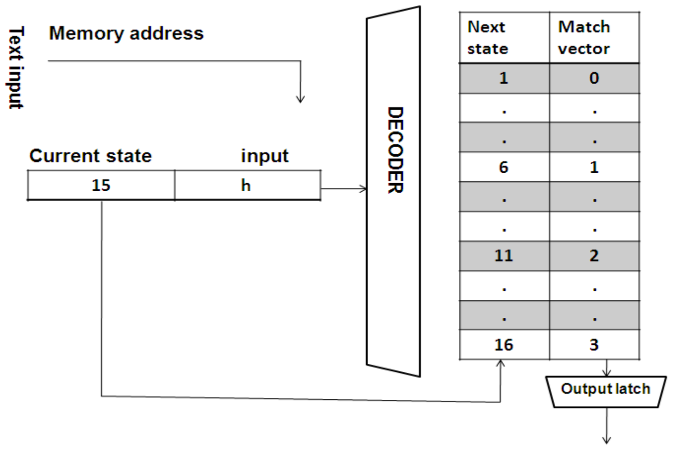

Fig.2 Memory Architecture

A value in match vector indicates which pattern is matched. For example in Fig.2, if the current state is 10 and the input character is g. The decoder will point to the memory location, which stores the next state 11 and the match vector 2. Here, the match vector 2 indicates the pattern “apcdeg” is matched.

The major issue in memory architecture is due to increasing number of attacks, the required memory increases. The performance cost and power consumption directly related to memory size so need comes here to reduce memory size.

Consider the same example in Fig.1 where three string patterns have a common sub-string “cde”. Because of the common sub-string, state 3 has “similar” state transitions to those of state 8 and 13. Similarly, states 4,9,14 and 5, 10, 15 have “similar” transitions. However 3,8,13, 4,9,14 and 5, 10, 15 states are not equivalent states and cannot be merged directly. We call a state machine merging those non-equivalent “similar” states as MERG. We propose a state-traversal mechanism on a MERG while achieving the same purposes of pattern matching. Since the number of states in MERG can be drastically smaller than the original FSM, it results in a much smaller memory size. So that the hardware needed to support the state-traversal mechanism is limited.

IV. MERGE FSM

\Among all the memory architectures, AC is widely adopted for string matching algorithm. It can reduce number of state transition and therefore the memory size can be reduced

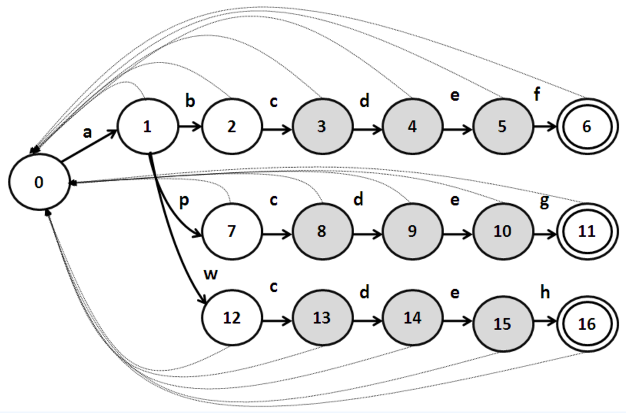

Fig.3 State transition diagram of AC algorithm

In the above diagram, solid line represents valid transition. Dotted line represents 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. Then, the machine recursively considers the same input character until the character causes a valid transition.

In the Fig.3, let us consider example when an AC machine is in state 2 and the input character is p. According to the AC state table in Table.1, there is no valid transition from state 2 given the input character 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 7. In Fig.3, the double-circled nodes indicate the final states of patterns. In Fig. 3, state 6, the final state of the first string pattern "abcdef", stores the match vector $\{P^3P^2P^1\}=\{001\}$, "apcdeg" stores the match vector $\{P^3P^2P^1\}=\{010\}$ and "awcdeh" stores the match vector $\{P^3P^2P^1\}=\{100\}$. Match vector is one hot encoded. The width of match vector equals to number of string pattern.

Table.1 AC State table

	Input	Next state	Failure	Match vector
State 0:	a	1	0	000
State 1:	b	2	0	000
State 1:	p	7	0	000
State 1:	w	12	0	000
State 2:	c	3	0	000
State 3:	d	4	0	000
State 4:	e	5	0	000
State 5:	f	6	0	001
State 7:	c	8	0	000
State 8:	d	9	0	000
State 9:	e	10	0	000
State 10:	g	11	0	010
State 12:	c	13	0	000
State 13:	d	14	0	000
State 14:	e	15	0	000
State 15:	h	16	0	100

Many string patterns are similar because of their common substrings. Due to the common substrings of string patterns, the compiled AC machine may have states with similar transitions. Those similar states are not equivalent states and cannot be merged directly. Functional errors can be created if those similar states are merged directly. In Fig.3, states 3,8,13 are similar because they have identical input transitions, identical failure transitions to state 0. Also, states 4,9,14 and 5, 10, 15 are similar.

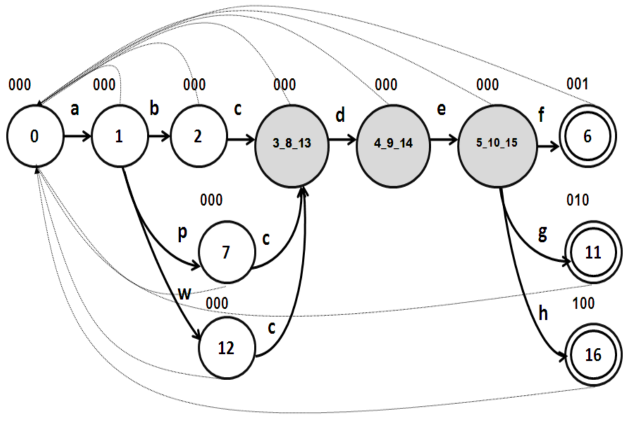

Fig.4 State machine merges similar states

But merging similar states directly will results in an erroneous state machine. As shown in Fig.4, the state machine merges the similar states 3,8,13 to become state 3_8_13 and 4,9,14 to become state 4_9_14 and merges the similar states 5, 10, 15 to become state 5_10_15. We refer to the state machine that merges the similar states as the MERG. Given an input string "apcdef", the original AC state machine shown in Fig.3 moves from state 0, through state 7, state 8 , state 9 and then takes a failure transition to state 0. On the other hand, the MERG moves from state 0, through state 7, state 3_8_13, state 4_9_14, state 5_10_15 and finally reaches state 6 which indicates the final state of the first pattern "abcdef". As a result of merging similar states, the input string "apcdef" is mistaken as a match of the pattern "abcdef". This example shows the MERG may causes false positive results if those similar states are merged directly. We propose a mechanism that can rectify those functional errors after merging those similar states. The MERG is a different machine from the original state machine but with a smaller number of states and transitions. A direct implementation of MERG has a smaller memory than the original state machine in the memory architecture. AC algorithm is modified so that we can store only the state transition table of MERG in memory. The new state traversal mechanism shown in Fig.5 guides the state machine to traverse on the MERG and provides correct results as the original AC state machine.

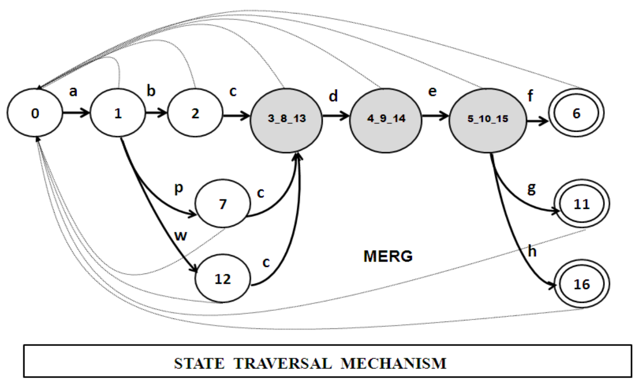

Fig.5 State Traversal Machine

V. STATE TRAVERSAL MECHANISM

In the merged states, state 3_8_13 represents three different states (state 3, state 8 and state 13), 4_9_14 represents three different states and 5_10_15 represents three different states. From the previous example we can understand that directly merging similar states leads to an erroneous state machine. To have a correct result, when state 3_8_13 is reached, we need a mechanism to understand in the original AC state machine whether it is state 3, state 8 or state 13. When state 4_9_14 is reached, we need to know in the original AC state machine whether it is state 4, state 9 or state 14. Similarly, for the state 5_10_15.

In this example, we can differentiate state 3, state 8 or state 13 if we can memorize the precedent state of state 3_8_13. If the precedent state of state 3_8_13 is state 2, we know that in the original state machine, it is state 3. On the other hand, if the precedent state of state 3_8_13 is state 7, the original is state 8 and if the precedent state of state 3_8_13 is state 12, the original is state 13.

This example shows that if we can memorize the precedent state entering the merged states, we can differentiate all merged states. Through this way we can eliminate the functional errors.
In a traditional AC state machine, a final state stores the corresponding match vector which is one-hot encoded. In Fig.3, state 6, the final state of the first string pattern "abcdef", stores the match vector $\{P^3P^2P^1\}=\{001\}$ and state 11, the final state of the second string pattern "apcdeg", stores the match vector of $\{P^3P^2P^1\}=\{010\}$ and state 16, the final state of the third string pattern "awcdeh" stores the match vector of $\{P^3P^2P^1\}=\{100\}$. The other states (not the final states) stores $\{P^3P^2P^1 = \{000\}$ as shown in Table.1. By expressing the match vector in one hot encoding format we can avoid the problem in representing the final states. The width of the match vector is equal to the number of string patterns. From the Table.1, we note that majority of memories in the column "match vector" store the zero vectors {000}. In our method, we reuse those memory spaces storing zero vectors {000} to store useful path information called "PATH VECTOR" (paVEC).
Each bit of the paVEC corresponds to a string pattern. Then, if there is a path from the initial state to a final state, which matches a string pattern, the corresponding bit of the paVEC of the states on the path will be set to 1. Otherwise, they are set to 0.

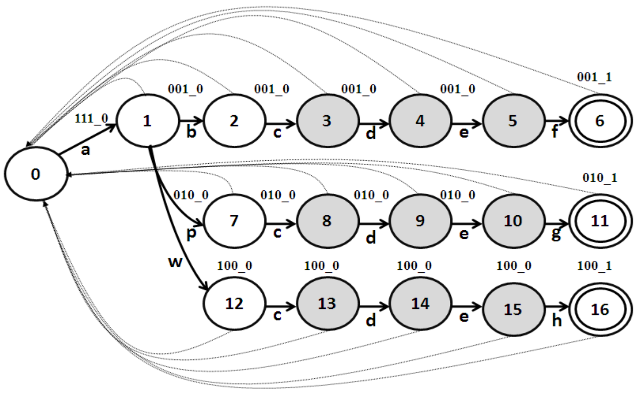

Fig.6 New data structure

Consider the string pattern "abcdef", whose final state is state 6 in Fig.6. The path from state 0 via states 1, 2, 3, 4, 5 to the final state 6 matches the first string pattern "abcdef". Therefore, the first bit of the paVEC of the states on the path, {state 0, state 1, state 2, state 3,state 4, state 5 and state 6}, is set to 1.The path from state 0, via states 1, 7, 8, 9, 10 to the final state 11 matches the second string pattern "apcdeg". Therefore, the second bit of the paVEC of the states on the path, {state 0, state 1 state 7, state 8, state 9, state 10 and state 11}, is set to 1. Similarly, the path from state 0 via states 1, 12,13,14,15 to the final state 16 matches the third string pattern "awcdeh". Therefore, the third bit of the paVEC of the states on the path, {state 0, state 1, state 12, state 13, state 14, state 15 and state 16}, is set to 1.
Also to indicate whether the state is a final state, we add an additional bit, called ifFINAL. In our example states 6, state 11 and 16 are final states, so the ifFINAL bits of states 6, 11 and 16 are set to 1; the others are set to 0. As shown in Fig.6, each state stores the paVEC and ifFINAL as the form, "paVEC_ ifFINAL". Compared with the original AC state machine in Fig.3, we only add an additional bit to each state.
In this example, states 3, 8 and 13, states 4, 9 and 14 and states 5, 10 and 15 are similar because they have similar transitions. But they are not equivalent ones. Two states are equivalent if and only if their next states are equivalent. In Fig. 6, states 5, 10 and 15 are similar but not equivalent. Because for the same input f, state 5 takes a transition to state 6, state 10 takes a failure transition to state 0 while state 15 takes a failure transition to state 0. In our algorithm, we define such similar states as pseudo-equivalent states.
PSEUDO-EQUIVALENT 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.*
In our example pattern, state 3, state 8 and 13 are pseudo-equivalent states because they have identical input transitions c, identical failure transitions to state 0 and identical ifFINAL bit as 0. Also, states 4,9,14 and states 5, 10, 15 are pseudo- equivalent states. In our algorithm, the pseudo-equivalent states 3, 8 and 13 are merged to be state 3_8_13 and states 4,9 and 14 are merged to be state 4_9_14 and states 5,10 and 15 and merged to be state 5_10_15 as shown in Fig.7.
The equivalent states are merged, so the states need to be updated. The paVEC_ifFINAL are updated by taking the union on the paVEC_ifFINAL of the merged states. Therefore, the paVEC_ifFINAL of states 3_8_13, 4_9_14 and 5_10_15 are modified as {111_0}.
To avoid functional errors, we need a mechanism to memorize the precedent state of the merged state. For that we need a register, called "PREREGISTER" (prREG), to trace the precedent paVEC in each state. The width of prREG is equal to the width of paVEC. Each bit of the prREG also corresponds to a string pattern. The prREG is updated in each state by performing a bitwise AND operation on the paVEC of the next state and its current value.

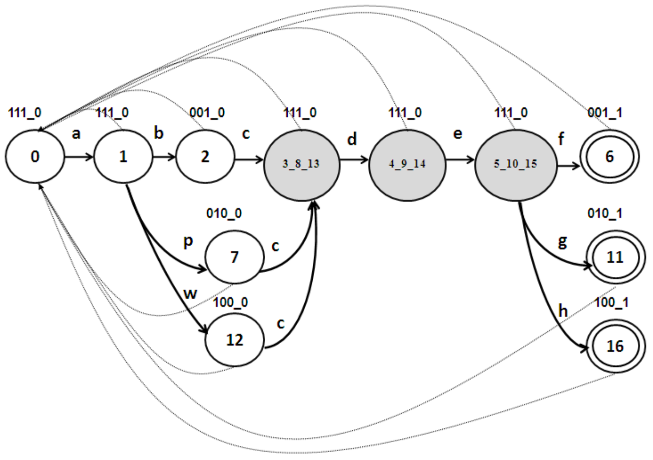

Fig.7 New state diagram

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 prREG indicates the match vector of the matched pattern. During the state traversal, if all the bits of the prREG 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 prREG are reset to 1.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)
© Copyright: All rights reserved.

Table .2 State transitions of input string "awcdeg"

State	0	1	12	3_8_13	4_9_14	5_10_15	11
Input character	a	w	c	d	e	g	g
paVEC	111	111	100	111	111	111	010
prREG	111	111	100	100	100	100	000
ifFINAL	0	0	0	0	0	0	1

For an example, string "awcdeg" when applied the transitions are given in Table 2. Initially, in state 0, the prREG is initialized to $\{P^3P^2P^1\} = \{111\}$. After taking the input character a, the MERG goes to state 1 and updates the prREG by performing a bitwise AND operation on the paVEC {111} of state 1 and the current prREG {111}. The resulting new value of the prREG will be $\{P^3P^2P^1\}$ = {111 AND 111} = {111}.

For the next input character w, the MERG goes to state 12 and updates the prREG by performing a bitwise AND operation on the paVEC {100} of state 12 and the current prREG {111}. The resulting new value of the prREG will be $\{P^3P^2P^1\}$ = {100 AND 111} = {100}. Then after taking next input character c, the MERG goes to state 3_8_13 and updates the prREG by performing a bitwise AND operation on the paVEC {111} of state 3_8_13 and the current prREG {100}. The resulting new value of the prREG will be $\{P^3P^2P^1\}$ = {111 AND 100} = {100}.

Further, after taking the input character d , the MERG goes to state 4_9_14 and updates the prREG by performing a bitwise AND operation on the paVEC {111} of state 4_9_14 and the current prREG {100}. The prREG remains $\{P^3P^2P^1\}$ = {111 AND 100} = {100}. For the next input character e, the MERG goes to state 5_10_15 and updates the prREG by performing a bitwise AND operation on the paVEC {111} of state 5_10_15 and the current prREG {100}. The resulting new value of the prREG will be $\{P^3P^2P^1\}$ = {111 AND 100} = {100}.

Finally, after taking the input character g, the MERG goes to state 11. After performing a bitwise AND operation on the paVEC {010} of state 11 and the current prREG {100}, the prREG becomes $\{P^3P^2P^1\}$ = {010 AND 100} = {000}.

According to our algorithm, during the state traversal, if all the bits of the prREG become 0, the machine will go to the failure mode and choose the failure transition as in the AC algorithm. Therefore, the machine takes the failure transition to state 0 instead of state 11. Also the above discussed string is not the example pattern constructed in FSM. So for this incorrect string, machine goes to failure state 0.

Table .3 State transitions of input string "awcdeh"

State	0	1	12	3_8_13	4_9_14	5_10_15	16
Input character	a	w	c	d	e	h	h
paVEC	111	111	100	111	111	111	100
prREG	111	100	100	100	100	100	100
ifFINAL	0	0	0	0	0	0	1

Table.3 shows the state traversal of input string "awcdeh". The process of state traversal is similar to the previous example until the state machine reaches state 5_10_15 and the input character is h. After taking the input character h, the MERG goes to state 16 and the prREG becomes $\{P^3P^2P^1\}$ = {100 AND 100} = {100}, by performing a bitwise AND operation on the paVEC {100} of state 5_10_15 and the current prREG {100}. Because the value of ifFINAL is 1, the value of prREG, $\{P^3P^2P^1\}$ = {100}, indicates the pattern P^3 is matched.

VI. MULTICHARACTER PROCESSING

Traditional software-based NIDS architecture fails to keep up with the throughput of high-speed networks because of the large number of patterns and complete payload inspection of packets. This has led to hardware-based schemes for multi-pattern matching.

Most multi-pattern matching solutions today are based on the AC algorithm. It performs multi-pattern matching in linear time based on constructing a finite state machine to do so. In [10], they have presented a state encoding scheme called a covered state encoding, which takes advantage of the "don't care" feature of TCAMs in the TCAM-based implementation of the AC algorithm.

For the patterns, "abcdef", "apcdeg" and "awcdeh", single character is taken to construct FSM. Now here through the concept of MULTI-CHARACTER PROCESSING, we are grouping the alphabets (characters) of pattern. Two characters are grouped for all the three patterns.

Fig.8 explains grouping of characters. '*' symbol denotes offset (don't care) terms. The patterns have common terms. So grouping of characters is possible. Through MERG concept, we are merging common characters.

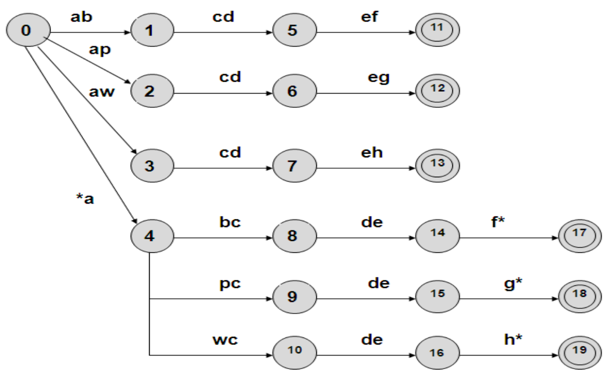

Fig.8 Grouping of characters

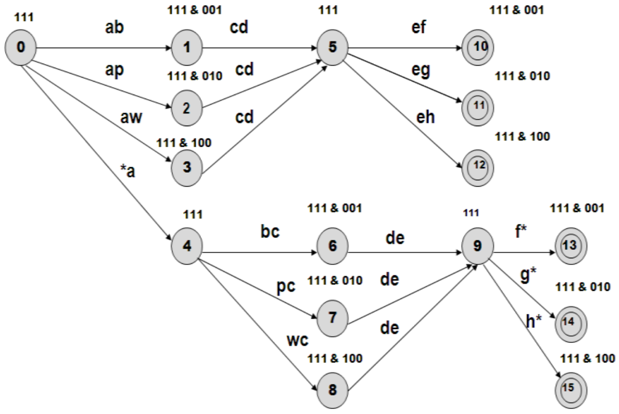

Fig.9 Multi-character process

In Fig.8, “cd” and “de” are common in all the patterns. So they are merged as shown in Fig.9. The paVEC and prREG are updated correspondingly.

VII. COMPARISON

All the 3 patterns are coded in VHDL and simulated in Modelsim. The comparison results are shown below.

Table.4 Comparison 1

	AC Algorithm	AC+ State Traversal Mechanism
NUMBER OF GATE COUNTS	373	346
MEMORY USAGE	148704 kilobytes	147680 kilobytes

Table.5 Comparison 2

	STATE TRAVERSAL MECHANISM (clock cycles)	MULTI CHARACTER PROCESSING (clock cycles)
3 patterns	8	4

From Table.1 it is concluded that sufficient memory has been reduced from previous methods. By state traversal mechanism, number of states for creating FSM also reduced.

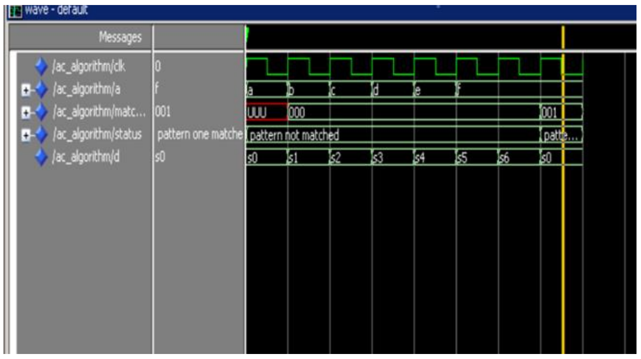

Fig.10 State Traversal Mechanism for I pattern

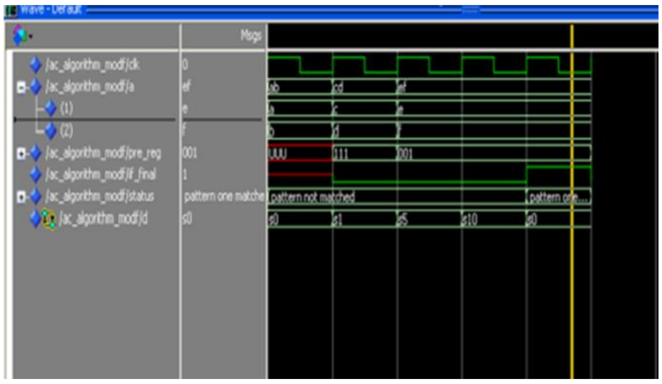

Fig.11 Multi-character processing for I pattern

The simulation result shows that speed has been increased when comparing to state traversal mechanism. The comparison Table.5 gives the clear details. For example, to process I pattern the previous mechanism takes 8 clock cycles. But through multi character processing, it has been reduced to 4.

VIII. CONCLUSION

Memory efficient pattern matching algorithm has been clearly proposed. Through pseudo-equivalent states, number of states reduced. MERG algorithm efficiently decreases memory, which is the basic requirement in NIDS. In our paper speed is taken as important parameter. Multi character processing greatly increases speed by reducing clock cycles. The future work can extend by implementing our proposed method in bit split pattern matching by which we can expect more memory reduction which is needed for today networking.

REFERENCES

[1] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using FPGAS,” in Proc. 9th Ann. IEEE Symp. Field-Program.CustomComput. Mach. (FCCM), 2001, pp. 227–238.

[2] C. R. Clark and D. E. Schimmel, “Scalable pattern matching on high speed networks,” in Proc. 12th Ann. IEEE Symp. Field Program.CustomComput. Mach. (FCCM), 2004, pp. 249–257.

[3] Z. K. Baker and V. K. Prasanna, “High-throughput linked-pattern matching for intrusion detection systems,” in Proc. Symp. Arch. For Netw.Commun. Syst. (ANCS), Oct. 2005, pp. 193–202.

[4] A. V. Aho and M. J. Corasick, “Efficient string matching: An AID to bibliographic search,” Commun. ACM, vol. 18, no. 6, pp. 333–340, 1975.

[5] Y. H. Cho and W. H. Mangione-Smith, “A pattern matching co-processor for network security,” in Proc. 42nd IEEE/ACM Des. Autom. Conf., Anaheim, CA, Jun. 13–17, 2005, pp. 234–239.

[6] Y. H. Cho and W. H. Mangione-Smith, “Fast reconfiguring deep packet filter for 1 + Gigabit Network,” i n Proc. 13th Ann. IEEE Symp. Field Program.CustomComput. Mach. (FCCM), 2005, pp. 215–224.

[7] S.Dharmapurikar and J. Lockwood, “Fast and scalable pattern matching for content filtering,” in Proc. Symp. Arch. for Netw.Commun. Syst. (ANCS), Oct. 2005, pp. 183–192.

[8] 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.

[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. (ISCA), 2005, pp. 112–122. SangKyunYun, Member, IEEESComputer Society

[10] “An Efficient TCAM-Based Implementation of Multipattern Matching Using Covered State Encoding”, IEEE TRANSACTIONS ON COMPUTERS, VOL. 61, NO. 2, FEBRUARY 2012

Lavanya received the BE degree in Electronics and Communication from Francis Xavier Engineering College, Tirunelveli affiliated to Anna University, Chennai. She presented a paper in National level Conference. Presented and participated ISTE workshop on “Introduction to Research Methodologies” conducted by IIT, Bombay through ICT (MHRD). She is currently pursuing her ME VLSI Design in Avinashilingam Institute for Home Science & Higher Education for Women University, Coimbatore- India.

Saranya received the BE degree in Computer Science and Engineering from Mahendra Engineering College, Affiliated to Anna University, Coimbatore. She presented a paper in International level Conference. Presented and participated DRDO Sponsored National Seminar on "Recent Research Application of Support Vector Machines"(RRASVM). She is currently pursuing her ME Software Engineering in Sir Ramakrishna Engineering College, Coimbatore- India

Uma Maheshwari received the BE degree in Computer Science and Engineering from P.T.R College Of Engineering And Technology, Affiliated to Anna University, Tirunelveli. Participated in a Seminar on "Simulation of Networks with NS2 a Practical Approach". She is currently pursuing her ME Computer Science and Engineering in Sir Ramakrishna Engineering College, Coimbatore- India