International Journal of Modern Engineering Research (IJMER)
www.ijmer.com Vol.2, Issue.4, July-Aug. 2012 pp-2742-2747 ISSN: 2249-6645

# Novel Pattern Matching using FSM Algorithm for Memory Architecture

B. Prasad kumar, B. Chinna rao, P. M. Francis
[1]*M. tech (PG student) GITAS*
[2]*Prof. & Head, Dept. of ECE, GITAS*
[3]*Asst. Prof. in Dept. of ECE, GITAS*

***Abstract:*** *Network intrusion detection system is used to inspect packet contents against thousands of predefined malicious or suspicious patterns. Because traditional software alone pattern matching approaches can no longer meet the high throughput of today's networking, many hardware approaches are proposed to accelerate pattern matching. Among hardware approaches, memory-based architecture has attracted a lot of attention because of its easy reconfigurability 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. In this paper, we propose a memory-efficient pattern-matching algorithm which can significantly reduce the memory requirement. For Snort rule sets, the new algorithm achieves 21% of memory reduction compared with the traditional Aho–Corasick algorithm. In addition, we can gain 24% of memory reduction by integrating our approach to the bit-split algorithm which is the state-of-the-art memory-based approach.*

***Index Terms:*** *Aho–Corasick (AC) algorithm, finite automata, pattern matching.*

## I. Introduction

**The** purpose of a signature-based network intru-sion detection system is to prevent malicious network attacks by identifying known attack patterns. Due to the in-creasing complexity of network traffic and the growing number of attacks, an intrusion detection system must be efficient, flexible and scalable.The primary function of an intrusion detection system is to perform matching of attack string patterns. Because string matching is the most computative task in network intrusion detection (NIDS) systems, many hardware approaches are pro-posed to accelerate string matching. The hardware approaches may be classified into two main categories, the logic [5], [8], [13],[16], [21], [26] and the memory architectures [4], [6], [7], [11], [14], [15], [22]–[24], [27]–[29] In terms of reconfigurability and scalability, the memory architecture has attracted a lot of attention because it allows on-the-fly pattern update on memory without resynthesis and relay out

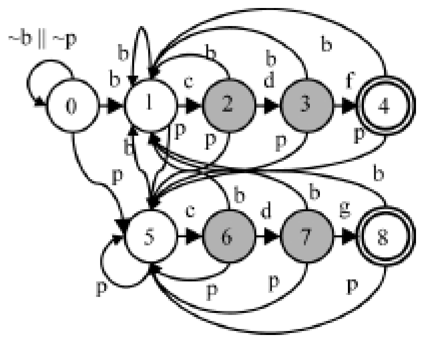


Fig 1 DFA for matching "bcdf" and "pcdg"

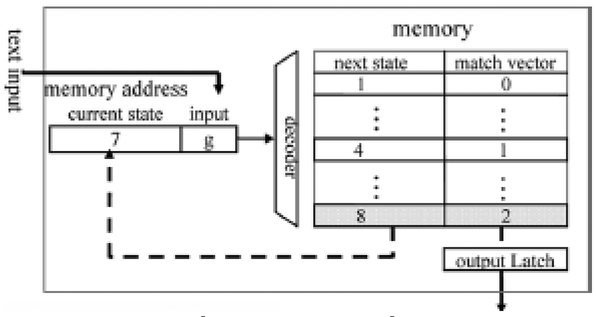


Fig 2 basic memory architecture

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 corre-sponding state transition table of the FSM is stored in memory. For instance, Fig. 1 shows the state transition graph of the FSM to match two string patterns "bcdf" and "pcdg", where all tran-sitions to state 0 are omitted. States 4 and 8 are the final states indicating the matching of string patterns "bcdf" and "pcdg", respectively. Fig. 2 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 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. For example in Fig. 2, suppose the current state is 7 and the input character is . 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. Due to the increasing number of attacks, the memory re-quired for implementing the corresponding FSM increases to the memory size, reducing the memory size has become imperative. Certain complicated virus string patterns can be represented by regular expressions. For example, the pattern for detecting the internet radio protocol is represented as "membername*session*player" For memory architecture, only few pre-vious works [15], [29] proposed $session.*player$ to reduce the complexity of reg-ular expressions. Still, majority of the patterns are exact string patterns. For example in Snort V2.4, there are 85% of exact string patterns. In this paper, we focus on reducing the memory size of the exact string patterns. We observe that many string patterns are similar because of common sub-strings. However, when string patterns are compiled into an FSM, the similarity does not lead to a small FSM. Consider the same example in Fig. 1 where two string patterns have a common sub-string "cd". Because of the common sub-string, state 2 has "similar" state transitions to those of state 6. Similarly, states 3 and 7 have "similar" transitions. However, states 2 and 6, states 3 and 7 are not equivalent states and cannot be merged directly. We call a state machine merging those non-equivalent "similar" states, *merg_FSM*.

In this paper, we propose a state-traversal mechanism on a merge_FSM while achieving the same purposes of pattern matching. Since the number of states in merg_FSM can be drastically smaller than the original FSM, it results in a much smaller memory size. We also show that hardware needed to support the state-traversal mechanism is limited. Experimental results show that our algorithm achieves 21% of memory reduction compared with the traditional AC algorithm for total string patterns of Snort [24]. In addition, since our approach is complementary to other memory reduction approaches, we can obtain substantial gain even after applying to the existing state-of-the-art algorithms. For example, after integrating with the bit-split algorithm [27], we can gain 24% of memory reduction.

## II. RELATED RESEARCHES

In this section, we review several related researches in this area. In the past few years, many algorithms and hardware designs are proposed to accelerate pattern matching. The hard-ware approaches can be classified into two main categories, logic and memory architectures. The logic architectures mostly use on-chip logic resources of field-programmable gate array (FPGA) to convert regular expression pattern into parallel state machines or combinatorial circuits because FPGA allows for updating new attack patterns. Sidhu *et al.* [26] proposed algorithm to compile regular expression patterns into combi-natorial circuits based on nondeterministic finite automaton (NFA). Hutchings *et al.* [13] developed a module generator that shared common prefixes to reduce the circuit area on FPGA. Moscola *et al.* [21] presented a content-scanning module on FPGA for an internet firewall. Clark *et al.* [8] improved area and throughput by adding predecoded wide parallel inputs to traditional NFA implementations. Baker *et al.* [5] presented a pre-decoded multiple-pipeline shift-and-compare matcher which reduced routing complexity and comparator size by con-verting incoming characters into many bit lines. Lin *et al.* [16] proposed a sharing architecture which significantly reduces circuit areas by sharing common infix and suffix sub-patterns.From the perspectives of reconfigurability and scalability, memory architectures are attractive because memory is flexible and scalable. The Aho–Corasick (AC) algorithm [1] is the most popular algorithm which allows for matching multiple string patterns. Aldwairi *et al.* [2] proposed a configurable string matching accelerator based on a memory implementation of the AC FSM. Tan *et al.* [27] proposed the bit-split algorithm partitioning a large AC state machine into small state machines to significantly reduce the memory requirements. Jung *et al.* [14] presented an FPGA implementation of the bit-split string matching architecture. Piyachon *et al.* [22] proposed to reduce the memory size by relabeling states of AC state machine. Ad-ditionally, Piyachon *et al.* [23] proposed to use Label Transition Table and CAM-based Lookup Table to significantly reduce the memory size. Cho *et al.* [6], [7] proposed a hash-based pattern matching co-processor where memory is used to store the list of substrings and the state transitions. Dharmapurikar *et al.* [11] proposed a pattern matching algorithm which modifies the AC algorithm to consider multiple characters at a time. Furthermore, the content addressable memories (CAM) is also widely used for string matching because it can match the entire pattern at once when the pattern is shifted past the CAM. Gokhale *et al.* [12] used CAM to perform parallel search at a high speed. Sourdis *et al.* [25] applied the pre-decoded tech-nique for the CAM-based pattern matching to reduce the area. Additionally, Yu *et al.* [30] presented a ternary content address-able memory (TCAM)-based multiple-pattern matching which can handle complex patterns, correlated patterns, and patterns with negation.

The hash-based approach was proposed to utilize Bloom filter for deep packet inspection. Dharmapurikar *et al.* [10] proposed a hashing-table lookup mechanism utilizing parallel bloom filters to enable large number of fixed-length strings to be scanned in hardware. Lockwood *et al.* [19] proposed an intelligent gateway based on Bloom filter that provides Internet worm and virus protection in both local and wide area networks.

## III. REVIEW OF AC ALGORITHM

In this section, we review the AC algorithm. Among all memory architectures, the AC algorithm has been widely adopted for string matching in [2], [14], [15], [22], [23], [27] because the algorithm can effectively reduce the number of state transitions and therefore the memory size. Using the same example as in Figs. 1 and 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 cur-rent state and an input character, the AC machine first checks whether there is a valid transition for the input character; oth-erwise, 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 transi-tion. Consider an example when an AC machine is in state 1 and the input character is . According $n$ to the AC state table in Fig. 4, there is no valid transition from state 1 given the input

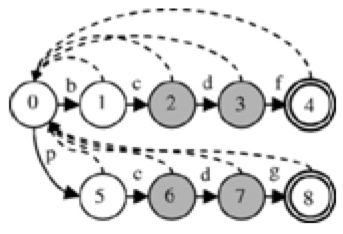


Fig. 3. State diagram of an AC machine.

| | input | next state | failure | match vector |
|---|---|---|---|---|
| State 0: | b | 1 | 0 | 00 |
| State 0: | p | 5 | 0 | 00 |
| **State 1:** | **c** | **2** | **0** | **00** |
| State 2: | d | 3 | 0 | 00 |
| State 3: | f | 4 | 0 | 01 |
| State 5: | c | 6 | 0 | 00 |
| State 6: | d | 7 | 0 | 00 |
| State 7: | g | 8 | 0 | 10 |

Fig4 . AC state table

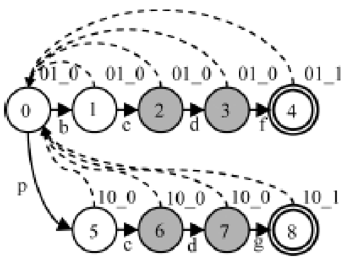


Fig 5 . Merging similar states

character . 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 In Fig. 3, the double-circled nodes indicate the final states of patterns. In Fig. 3, state 4, the final state of the first string pattern "bcdf", stores the match vector {P2,P1} = {01} and state 8, the final state of the second string pattern "pcdg", stores the match vector of {P2,P1} = {10} Except the final states, the other states store the match vector {P2,P1} = {00} to simply express those states are not final states.

## IV. STATE TRAVERSAL MECHANISM ONAMERG FSM

In our design, we reuse those memory spaces storing zero vectors {00} 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. 7. The path from state 0, via states 1, 2, 3 to the final state 4 matches the first string pattern "bcdf". There-fore, the first bit of the pathVec of the states on the path, {state 0, 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". 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. 7, each state stores the pathVec and ifFinal as the form, "*pathVec_ ifFinal*". Compared with the original AC state machine in Fig. 3, we only add an additional bit to each state. We have mentioned that in this example, states 2 and 6, states 3 and 7 are similar because they have similar transitions. How-ever, they are not equivalent. Note that two states are equivalent if and only if their next states are equivalent. In Fig. 7, states 3 and 7 are similar but not equivalent because for the same input , 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 equiv-alent states because their next states, state 3 and state 7, are not equivalent states. In our algorithm, we define such similar states as pseudo-equivalent states. The definition is as follows.

*Definition:* 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 Fig. 7, states 2 and 6 are pseudo-equivalent states be-cause they have identical input transitions , identical failure transitions to state 0 and identical ifFinal bit 0. Also, state 3 and state 7 are pseudo-equivalent states. In our 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. 8. 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, we need a register, called *preReg*, to trace the precedent pathVec in each state. The width of preReg is equal to the width of pathVec. Each bit of the preReg also corresponds to a string pattern. The preReg is updated in each state by per-forming 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. the string "pcdf" is ap-plied. Initially, in state 0, the preReg is initialized to{P2,P1} ={11} After taking the input character ,P the merg_FSM goes to state 5 and updates the preReg by performing a bitwise AND operation on the pathVec {10} f state 5 and the current preReg {11} The resulting new value of the preReg will be {P2,P1} ={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{P2,p1}={11AND10} ={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} = {11AND10}={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}={01AND10}={00} According to our 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. Therefore, the machine takes the failure transition to state 0 instead of state 4.

## V. 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. 16 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

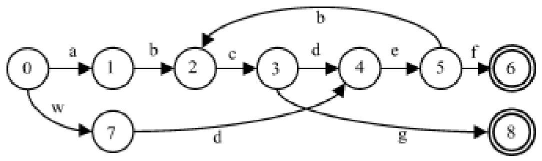


Fig. 6 Merging pseudo-equivalent states with different sequences.

The loop transition may cause false positive matching results. For example, the input string “abcdebcdef” will be mistaken as a match of the pattern “abcdef.” 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. Therefore, in our program, we record and identify the orders of common sub-patterns. If the common sub-patterns appear in sequence, the corresponding pseudo-equivalent states can be merged without loop back problems. Fig. 18 shows the pseudo code of our algorithm to find common substrings without the loop back problem. First, all common sub-strings are extracted by the longest common substring algorithm [9]. The algorithm can report all of the common substrings. Then, the common substrings are labeled as new sequences. Next, we use the longest common subsequence (LCS) algorithm [20] to find all of the longest subsequence common to all strings. The results from the LCS algorithm guarantee that there will be no loop back transition. For example, consider the two patterns, “abcdefghijklm” and “abcwsghidefxyklm.” Using the longest common substring algorithm, we can extract all of the common substrings of these two patterns such as “abc”, “def”, “ghi” and “klm”. Then, we label the substrings “abc”, “def”, “ghi”, and “klm” as ,αβγδ , and , respectively. Therefore, the sequence of substrings in “abcdefghijklm” is labeled as “ ” while the sequence of substrings in “abcwsghidefxyklm” is labeled as “ ”. We subsequently use LCS algorithm to find all of the longest common subsequences among the two new sequences, “αβγδ ” and “ ” and the results are “ αβδ” or “ αγδ”. Therefore,we can merge the subsequences of (“abc”), (“ghi”) and (“klm”) or the subsequences of (“abc”), (“def”) and (“klm”) without the loop back problem. Notice that the result of LCS may not be unique.

## VI. HARDWARE ARCHITECTURE

Fig.7 shows our hardware module which can be configured for matching 16 or 32 patterns with a state machine containing 1024 valid transitions at most. In Fig.7,8, the register, called *address_register*, is used to store the current state and the input character. The *valid_memory* is used to store the information of *valid_state*, *pathVec*, and *ifFinal* corresponding to each valid transition while the *failure_memory* is used to store the *failure_state* corresponding to each failure transition. In this prototype, we use a hardwired circuit, called *A2P*, to translate the content of the address_register to a contiguous scope, called *pos*, to utilize the valid_memory. The circuit A2P can be implemented using hardwired circuit or CAM [17]. In addition, the signal *n_valid* is high if there is no valid transition corresponding to the address_register. Furthermore, the register

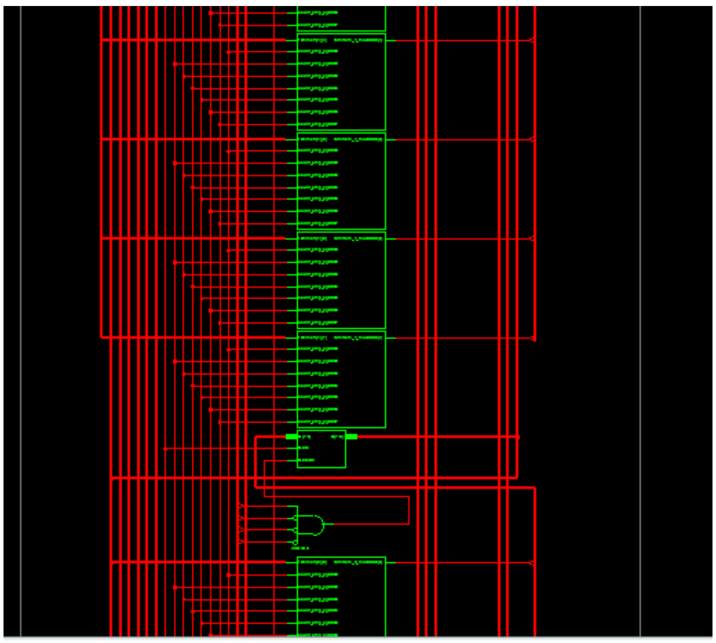
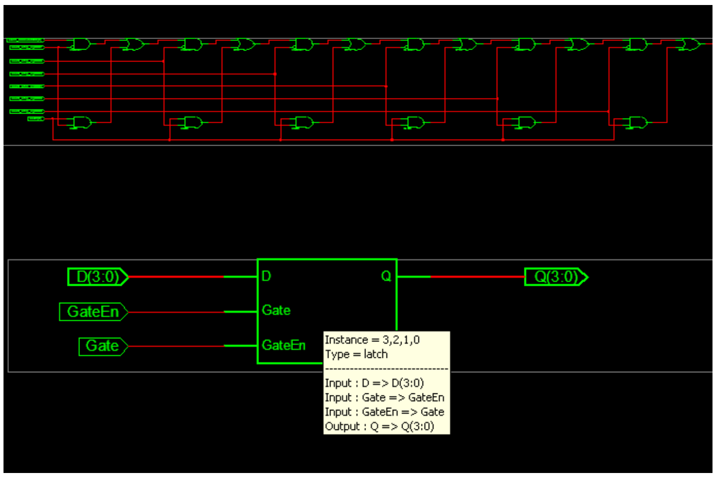


Fig. 7. Hardware module for the new algorithm.



Instance = 3,2,1,0
Type = latch
Input : D => D(3:0)
Input : Gate => GateEn
Input : GateEn => Gate
Output : Q => Q(3:0)

Fig 8 Internal architecture of proposed system

called *preReg*, is used to trace the precedent pathVec in each state. The preReg is initiated to be 1 for all bits and is updated by performing a bitwise AND operation on its current value and the pathVec from the valid_memory. The *ns_ctrl* unit is used to determine the next state by the value of preReg and n_valid. If the preReg is 0 for all bits or the n_valid is 1, the ns_sel will output low to let the failure_state update the current_state register. On the other hand, if the preReg is not zero and the n_valid is not 1, the ns_sel will output high to let the valid_state update
the current_state register.

## VII. EXPERIMENTAL RESULTS

Using the version 2.4 of Snort rule set, we extract 2217 exact string patterns containing 36 539 characters from the rule database. The results are compared with the methods of the AC algorithm and the bit-split algorithm.

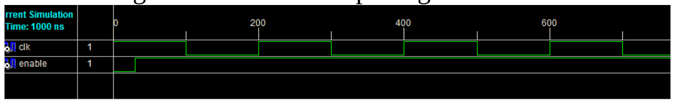


Fig 9 string patterns from Snort rule database

The flow of our experiment is shown in Fig.9. In the first stage, we obtain string patterns from Snort rule database. In the second stage, we group 32 string patterns as a module based on the similarity of string patterns. Further, in the third stage, we use LCS to extract substrings without loop back problem. Because the solution of LCS may not be unique, we select the common substrings which have the largest *sharing gain*. The sharing gain of common substrings is defined as the length of common substrings multiplied by the number of patterns sharing the common substrings.

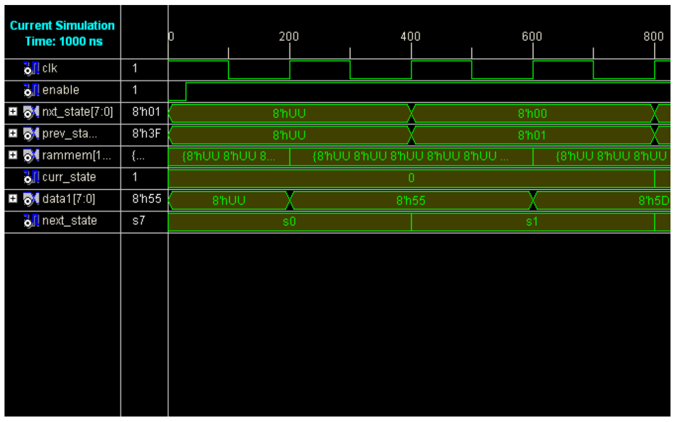


Fig 10 the selection line with clk to the pesodo random order adjustment

For example, three patterns, "1common1", "2common2", and "3common3" have the common substrings common". The sharing gain of the common substrings is because the substring "common" has six characters which are shared by three patterns. In the final stage, we merge the extracted common substrings and generate the transition table. Table I shows the results before and after integrating our algorithm to the AC algorithm. Columns one, two and three show the name of the rule set, the number of patterns, and the number of characters of the rule set. Columns four, five, and six show the number of state transitions, the number of states, and the memory size of the AC algorithm. Columns seven, eight, and nine show the results of our approach. Column ten shows the memory reduction compared to the AC algorithm. As shown in Fig. 10, the memory requirement includes the size of the valid memory and the failure memory. Because the memory requirement is proportional to the number of states, our algorithm has reduced memory size on the traditional AC algorithm.

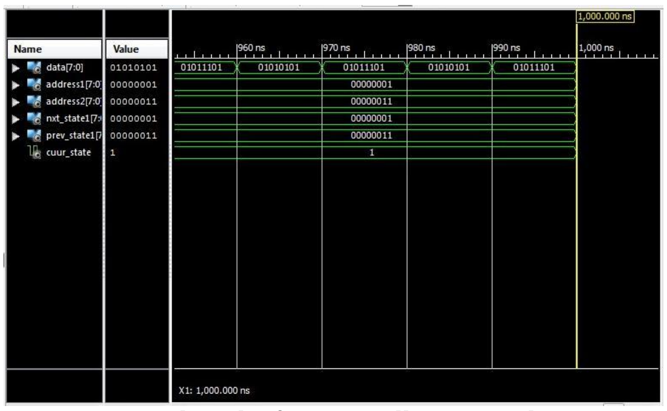


Fig 11 Final result of memory allocation and ram adjustment

Using the traditional AC algorithm, the number of transitions and states are 6793 and 6804, respectively. The memory size is 49 267 bytes. Integrating our algorithm to the AC algorithm, the number of transitions and states are reduced to 4432 and 3846, respectively. The memory size is reduced to 30 699 bytes, 38% of memory reduction from the AC algorithm. For total 2217 string patterns of Snort rule sets, our algorithm achieves a 21% memory reduction compared with the AC algorithm. Because the state-of-the-art bit-split algorithm is based on the AC algorithm, our algorithm can also be integrated to the bitsplit algorithm to further reduce memory requirements. Applying the bit-split algorithm which splits the traditional AC state machine into 4 state machines, the number of transitions and states are 21 949 and 21 993, respectively. The size of memory is 159 202 bytes. Integrating our algorithm to the bit-split algorithm, the number of transitions and states are reduced to 14 437 and 12 664, respectively. The size of memory is reduced to 98 400 bytes. The memory reduction achieves 38%. For total 2,217 string patterns of Snort rule sets, integrating our algorithm to the bit-split algorithm can achieve 24% of memory reduction. Furthermore, we have synthesized the hardware module in Fig. 19 using the ASIC flow of the UMC 0.18 m technology. The results are compared with [2], [6], [27], [28], [30] as shown in Fig 11, columns 2, 3, and 4 shows the number of characters, the memory size, and the throughput. Column 5 shows the memory utilization per character while column 6 shows the memory efficiency which is defined as the following equation:

$$\boldsymbol{Memory\ efficenvy} = \frac{\boldsymbol{(throughputXChar.Num)}}{\boldsymbol{Mem}}$$

## VIII. CONCLUSION

We have presented 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. In addition, the new algorithm is complementary to other memory reduction approaches and provides further reductions in memory needs. The experiments demonstrate a significant reduction in memory footprint for data sets commonly used to evaluate IDS systems.

## REFERENCES

[1] 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.

[2] 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.

[3] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network IDS/IPS," in *Proc. IEEE Int. Conf. Netw. Protocols*

[4] 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.

[5] 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.

[6] 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.

[7] Y. H. Cho and W. H. Mangione-Smith, "Fast reconfiguring deep packet filter for" in *Proc. 13th Ann. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2005, pp. 215–224.

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

[9] G. Dan, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge, U.K.: Cambridge University Press, 1997.

[10] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood,"Deep packet inspection using parallel bloom filters," in *Proc. 11th Symp. High Perform. Interconnects*, Aug. 2003, pp. 44–53. [26] R. Sidhu and V. K. Prasanna, "Fast regular expression matchingusing FPGAS," in *Proc. 9th Ann. IEEE Symp. Field-Program. CustomComput. Mach. (FCCM)*, 2001, pp. 227–238.

[11] 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.

[12] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. H. Granidt, "Towards gigabit rate network intrusion detection," in *Proc. the Eleventh Annual ACM/SIGDA International Conference on Field- Programmable Logic and Applications (FPL '03)*, 2002, pp. 404–413.

[13] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. 10 th Annu. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2002, pp. 111–120.

[14] H. J. Jung, Z. K. Baker, and V. K. Prasanna, "Performance of FPGA implementation of bit-split architecture for intrusion detection systems," presented at the 20th Int. Parallel Distrib. Process. Symp. (IPDPS),Rhodes Island, Greece, 2006.

[15] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*,2006, pp. 339–350.

[16] C. H. Lin, C. T. Huang, C. P. Jiang, and S. C. Chang, "Optimization of pattern matching circuits for regular expression on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 12, pp.1303–1310, Dec. 2007.

[17] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, "A memory-efficient parallel string matching architecture for high-speed intrusion detection," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 10, pp. 1793–1804, Oct. 2006.

[18] J. V. Lunteren, "High-performance pattern-matching for intrusion detection," in *Proc. IEEE INFOCOM*, 2006, pp. 1–13.

[19] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks,"Internet worm and virus protection in dynamically reconfigurable hardware," presented at the Military Aerosp. Program. Logic Device(MAPLD), Washington, DC, Sep. 2003, E10.

[20] D. Maier, "The complexity of some problems on subsequences and supersequences," *J. ACM*, vol. 25, no. 2, pp. 322–336, 1978.

[21] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," in *Proc. 11th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2003, pp. 31–38.

[22] P. Piyachon and Y. Luo, "Compact state machines for high performance pattern matching," in *Proc. 41nd IEEE/ACM Des. Autom. Conf.*,2007, pp. 493–496.

[23] P. Piyachon and Y. Luo, "Design of high performance pattern matching engine through compact deterministic finite automata," in *Proc. 42nd IEEE/ACM Des. Autom. Conf.*, 2008, pp. 852–857.

[24] M. Roesch, "Snort- lightweight intrusion detection for networks," in *Proc. 15th Syst. Administration Conf. (LISA)*, 1999, pp. 229–238.

[25] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in *Proc. 12th Annu. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 258–267.

[27] 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.

[28] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficien string matching algorithms for intrusion detection,"in *Proc. 23nd Conf. IEEE Commun. Soc. (INFOCOMM)*, Mar. 2004, pp. 2628–2639.

[29] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ACM/IEEE Symp. Arch. Netw. Commun. Syst. (ANCS)*, 2006, pp. 93–102.

[30] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet patternmatching using TCAM," in *Proc. 12th IEEE Int. Conf. Netw. Protocols (ICNP)*, 2004, pp. 174–183.