IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

High Speed Low Area Pattern Matching Algorithm for Memory Architecture

Kiran Kumar.M[*1], Murali Kante[2], Narayana Reddy Vanteru[3]
[*1,2,3] Narayana Engineering College, India
kiranyadav418@gmail.com

Abstract

Pattern matching is one of the most important components for the content inspection based applications of network security, and it requires well designed algorithms and architectures to keep up with the increasing network speed. Due to the advantages of easy re-configurability and scalability, the memory-based string matching architecture is widely adopted by network intrusion detection systems (NIDS). In order to accommodate the increasing number of attack patterns and meet the throughput requirement of networks, a successful NIDS 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.

Keywords: Aho–Corasick (AC) algorithm, finite automata, pattern matching.

Introduction

The main purpose of a signature-based network intrusion detection system is to prevent malicious network attacks by identifying known attack patterns. Due to the increasing 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 [1]-[6] and the memory architectures [7]-[11]. 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 relayout. 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

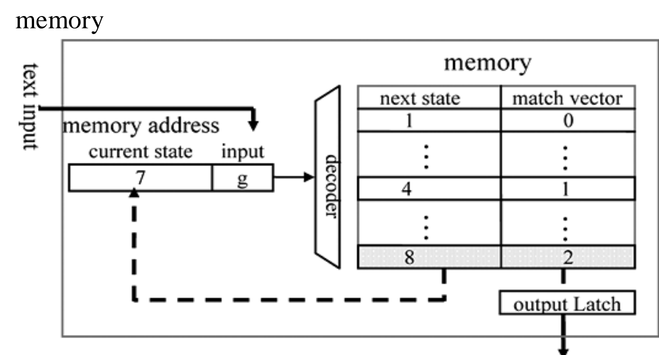

Fig.1 Basic memory architecture

Fig. 1 Represents 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. 1, 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.

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. Our algorithm achieves 21% of memory reduction compared with the traditional AC algorithm.

Review of AC Algorithm

The Aho–Corasick (AC) algorithm [12] is the most popular algorithm which allows for matching multiple string patterns. In this section, we review the AC algorithm. Among all memory architectures, the AC algorithm has been widely adopted for string matching in [2], [10], [11] because the algorithm can effectively reduce the number of state transitions and therefore the memory size.

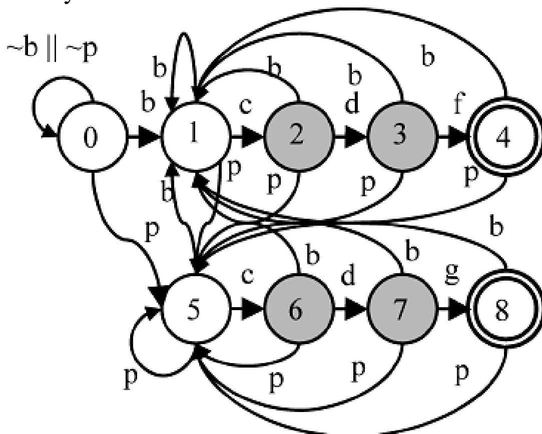

Fig. 2 DFA for matching "bcdf" and "pcdg"

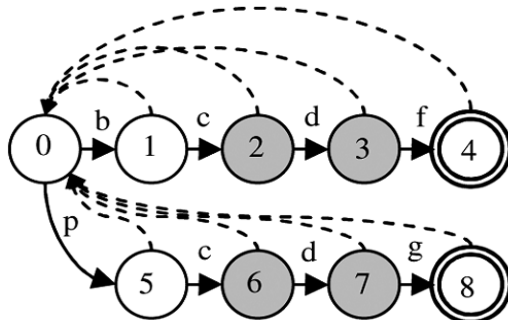

Fig. 3 State diagram of an AC machine

	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

Fig. 4 AC state table

In figures 2 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; 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. Consider an example when an AC machine is in state 1 and the input character is p. According to the AC state table in Fig. 4, there is no valid transition from state 1 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 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 $\{P_2P_1\}=\{01\}$ and state 8, the final state of the second string pattern "pcdg", stores the match vector of $\{P_2P_1\}=\{10\}$. Except the final states, the other states store the match vector $\{P_2P_1\}=\{00\}$ to simply express those states are not final states.

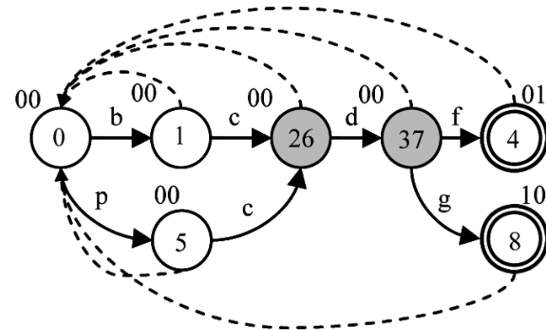

Fig. 5 Merging similar states

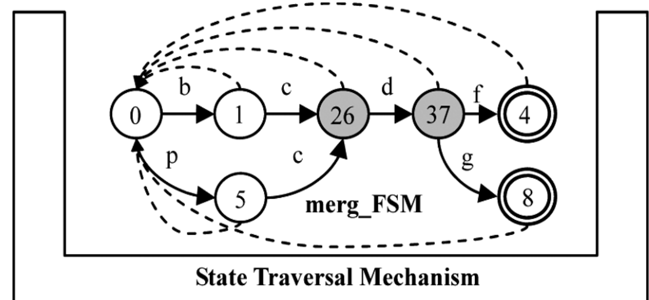

Fig. 6 Architecture of the state traversal machine

Basic Idea

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, we first show that functional errors can be created if those similar states are merged directly. Then, we propose a mechanism that can rectify those functional errors after merging those similar states.

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. 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. In Section IV, we first discuss the state traversal mechanism. Then in Section V, we discuss how the state traversal machine is created in our algorithm.

State Traversal Mechanism on a MERG_FSM

In the previous example, 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, we need a mechanism 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. In this example, we can differentiate state 2 or state 6 if we can 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 if we can memorize the precedent state entering the merged states, we can differentiate all merged states. In the following section, we discuss how the precedent path vector can be retained during the state traversal in the merg_FSM.

First of all, we would like to mention that in a traditional AC state machine, a final state stores the corresponding match vector which is one-hot encoded. For example in Fig. 3, state 4, the final state of the first string pattern “bcdf”, stores the match vector and state 8, the final state of the second string pattern “pcdg”, stores the match vector of . Except for the final states, the other states store 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 [4].

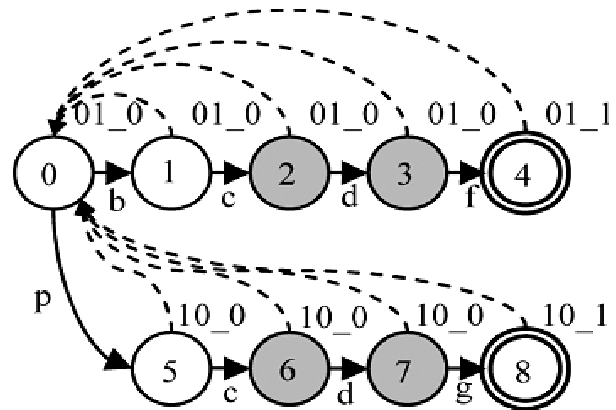

Fig. 7 New Data Structure, Path Vector, and if Final

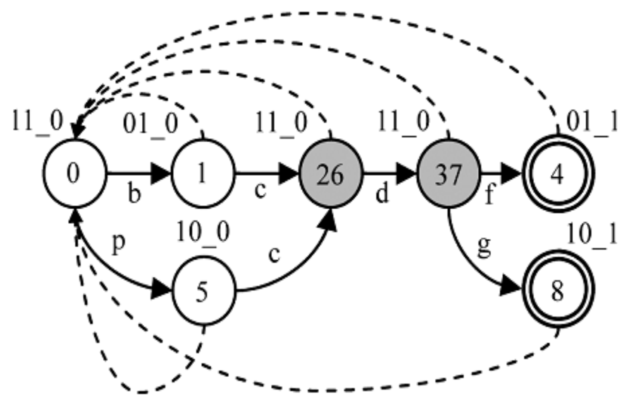

Fig. 8 New state diagram of Merg_FSM

Therefore, the width of the match vector is equal to the number of string patterns. As shown in Fig. 4, the majority of memories in the column “match vector” store the zero vectors {00} which are not efficient.

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”. Therefore, 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. However, 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 equivalent states because their next states, state 3 and state 7, are not equivalent states.

State Traversal Pattern Matching Algorithm

Algorithm: State traversal pattern matching algorithm
Input: A text string $x=a_1a_2\ldots a_n$ where each a_i is an input symbol and a state traversal machine *M* with valid transition function *g*, failure transition function *f*, path function *pathVec* and final function *ifFinal*.
Output: Locations at which keywords occur in x.
Method:

begin
  state ← 0
  preReg ← 1….1   //all bits are initiated to 1.
  for *i* ← until *n* do
   begin
    preReg = *preReg* & *pathVec*(*state*)
    while *g*(*state*, a_i) == *fail* || *preReg* == 0 **do**
      begin
       state ← *f* (*state*)
       preReg ← 1….1
      end
    state ← g(*state*, a_i)
    if *ifFinal*(*state*) = 1 **then**
     begin
      print *i*
      print *preReg*
     end
   end
end

Hardware Architecture

This hardware module which can be configured for matching 16 or 32 patterns with a state machine containing 1024 valid transitions at most. In Fig. 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 in-formation of valid_state, pathVec, and if Final 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, calledA2P , 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. In addition, the signal n_valid is high if there is no valid transition corresponding to the address_register. Furthermore, the register, 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.

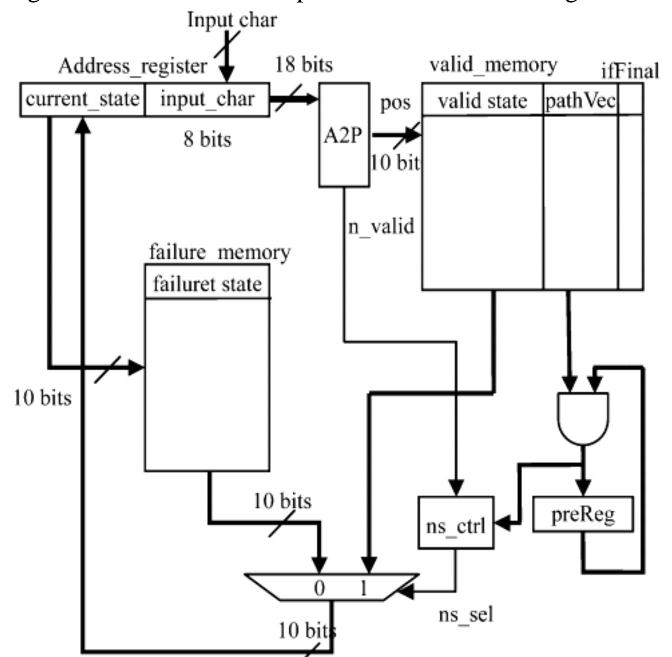

Fig. 8 Hardware Module for New Algorithm

Results and Discussion

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

 ISSN: 2277-9655

patterns of Snort rule sets, our algorithm achieves a 21% memory reduction compared with the AC algorithm.

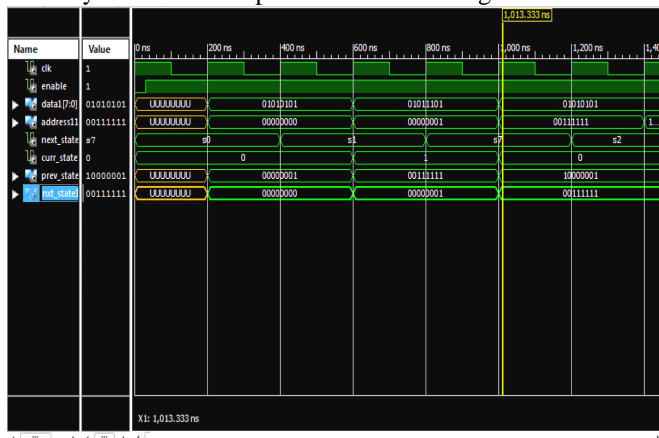

Fig. 9 Indicates the current state output

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] 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 matchinghardware for speeding up intrusion detection," *Proc. ACM SIGARCH Comput. Arch. News*, vol. 33, no. 1, pp. 99–107, 2005.

[3] 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.

[4] 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.

[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 in *Proc. 13th Ann. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2005, pp. 215–224.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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