An Efficient TCAM-Based Implementation of Multipattern Matching Using Covered State Encoding

SangKyun Yun, *Member*, *IEEE Computer Society*

Abstract—This paper proposes a state encoding scheme called a covered state encoding for the efficient TCAM-based implementation of the Aho-Corasick multipattern matching algorithm, which is widely used in network intrusion detection systems. Since the information of failure transitions of the Aho-Corasick Nondeterministic Finite Automata (NFA) is implicitly captured in the covered state encoding and the failure transition entries can be completely eliminated, the Aho-Corasick NFA can be implemented on a TCAM with smaller number of entries than other schemes. We also propose constructing the modified Aho-Corasick NFA for multicharacter processing, which can be implemented on a TCAM using the covered state encoding. The implementation of modified Aho-Corasick NFA using the covered state encoding is also superior to other schemes in both TCAM memory requirement and lookup speed.

Index Terms—String matching, multipattern matching, TCAM, intrusion detection system, Aho-Corasick algorithm.

1 Introduction

With increased growth in malicious network activity, Network Intrusion Detection Systems (NIDS) are being devised and deployed to detect the presence of any malicious or suspicious content in packet data. Signature-based NIDS rely on a multipattern matching algorithm. 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 multipattern matching. Since the rule sets are continuously updated, memory/ternary content addressable memory (TCAM)-based architecture [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] and FPGA-based architecture [12], [13], [14], [15], which are a programmable or reconfigurable, are commonly adopted for a hardware-based pattern matching.

Most multipattern matching solutions today are based on the Aho-Corasick (AC) algorithm [16]. It performs multipattern matching in linear time based on constructing a finite state machine to do so. A finite state machine can be efficiently implemented using the TCAM-based architecture, where the number of TCAM entries is equal to the number of transitions in the state machine. A TCAM has the "don't care" matching feature, which can be used in reducing the number of TCAM entries. Gould et al. [3] proposed a method to reduce the number of TCAM entries by using a logic minimization method. Alicherry et al. [1], [2] proposed a state encoding scheme exploiting the "don't care" feature of TCAMs, where some transitions are eliminated in the TCAM entries. Their methods, however, have limitations in reducing the number of entries since they are based on a deterministic finite automaton (DFA) of the AC algorithm which has much more transitions than the corresponding nondeterministic finite automaton (NFA). The NFA can be implemented with smaller TCAM entries than the DFA although it makes more transitions than the DFA during the pattern matching due to failure transitions.

In this paper, we present 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. Since the information of failure transitions in the NFA of the AC algorithm is implicitly captured in the covered state encoding, the failure transition entries can be completely eliminated in the proposed scheme and one can further substantially reduce the memory requirement.

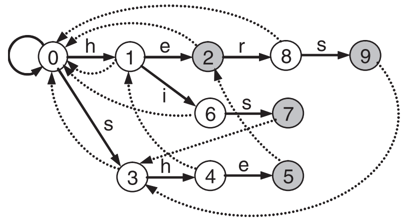
This paper is organized as follows: Section 2 presents related work. Section 3 proposes a covered state encoding scheme and the corresponding algorithms. Section 4 evaluates the covered state encoding scheme. Section 5 presents a multicharacter processing scheme using a covered state encoding. Section 6 includes concluding remarks.

2 Related Work and Problem Analysis

The multipattern matching problem is defined as finding occurrences in a text string T, of any pattern in a set of patterns $P = \{P_1, P_2, \dots, P_q\}$. This problem was first solved by Aho and Corasick [16]. The AC algorithm first constructs a finite state machine from the set of patterns and then uses the finite state machine to process the text string in a single pass. The algorithm builds a nondeterministic finite automaton by constructing the *goto* and *failure* transitions and then converts it into a deterministic finite automaton. A DFA has more transitions than the corresponding NFA.

• *The author is with the Department of Computer and Telecom. Engineering, Yonsei University, Wonju, Gangwon 220-710, Korea. E-mail: skyun@yonsei.ac.kr.*

Manuscript received 20 May 2010; revised 26 Oct. 2010; accepted 18 Nov. 2010; published online 9 Dec. 2010.
Recommended for acceptance by R. Figueiredo.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-05-0307.
Digital Object Identifier no. 10.1109/TC.2010.273.



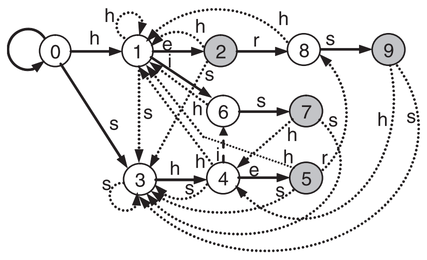


Fig. 1. Aho-Corasick finite automata for {he, she, his, hers}. (a) Goto and failure transitions (NFA). (b) DFA.

Fig. 1 shows the finite state machines for the set of patterns {he, she, his, hers} built by the AC algorithm. In the NFA, the solid lines and dotted lines represent goto and failure transitions, respectively. In the DFA, the dotted lines represent transitions, called by *cross transitions*, which are newly added by eliminating failure transitions. Shaded states represent the pattern matching states called *output states*. The *trivial* transitions going to the initial state are omitted in the figure.

The pattern matching can be performed using either the NFA or the DFA. To match a string, one starts from the initial state (usually 0). If a goto transition or a cross transition is matched with an input in the current state, the current state is moved along the matched transition. Otherwise, for a DFA-based matching, the current state goes back to the initial state and the matching process repeats for the next input, and for an NFA-based matching, the current state is moved along its failure transition and the matching process repeats for the current input. The DFA examines each input only once while the NFA may examine each input several times along the failure transition path. In matching a text string of length n, the DFA makes n state transitions and the NFA makes fewer than $2n$ state transitions [16]. Thus, the AC algorithm has a deterministic execution time.

The AC DFA requires a large amount of memory in a straightforward RAM-based implementation that keeps a table of pointers to next states for every input since the table contains also trivial transitions which go back to the initial state. Tan et al. [9] and Lunteren [10] proposed methods reducing the memory requirement in the RAM-based implementation.

The AC DFA can be implemented more efficiently using a TCAM since it needs only nontrivial transitions. Yu et al. [5] proposed a TCAM-based multibyte multiple string matching algorithm with limited support for wildcards. Weinsberg et al. [6] presented a pattern matching algorithm called Rotating TCAM (RTCAM) improving Yu et al.'s scheme. However, two TCAM-based schemes are not the implementation of Aho-Corasick finite state machines. Fig. 2 shows the TCAM-based hardware architecture for a finite state machine. The architecture consists of a TCAM, SRAM, and a logic. Each TCAM entry represents a lookup key, which consists of *current state* and *input*, and has corresponding data, which is the *next state*, in the SRAM whose address is given by the TCAM output. Two registers *current state* and *input* are initialized to the state 0 and the start data of the input buffer, respectively. If there is a matching entry for the state and input value in the TCAM, the TCAM outputs the index of the matching entry and then the SRAM outputs the *next state* data located in the corresponding location. Because a TCAM has "don't care" bits, multiple entries can be simultaneously matched and when this is the case, the index of the first matched entry is outputted. If there is no such match in the TCAM, the next state is the initial state. At every state transition, an input is advanced to the next input and the next state value is stored into the current state register.

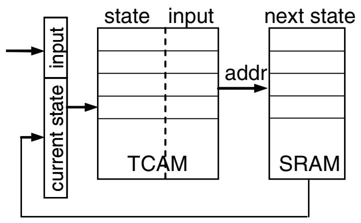
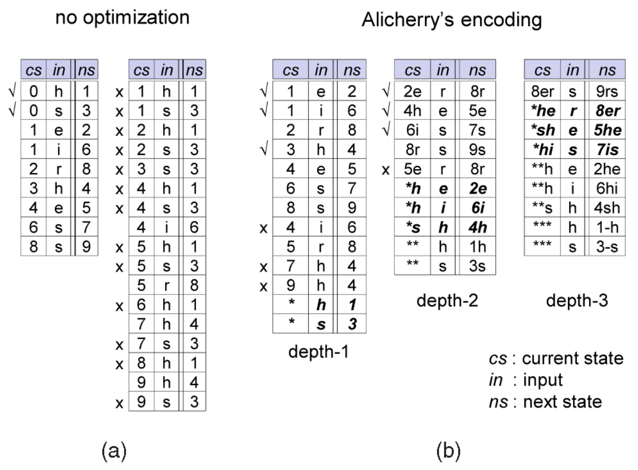


Fig. 2. TCAM-based hardware architecture.

Each TCAM entry represents a transition in the state machine. The number of TCAM entries is equal to the number of transitions and independent of the number of states. For a transition $g(s,i)=t$, where s is a current state, i is an input, and t is the next state, we will simply represent a pair of TCAM and SRAM entry by the combined form $(s,i|t)$.

Although the AC DFA has deterministic execution time, the number of transitions increases rapidly as the number of patterns increases. This makes the TCAM-based implementation impossible when there are a large number of patterns. In order to solve this problem, Alicherry et al. [1], [2] proposed a state encoding exploiting a "don't care" feature of TCAM, which can implement a DFA with a much smaller number of TCAM entries than the number of transitions in the DFA. We call the distance from the initial state to a given state as its *depth*, and a cross transition which goes to a depth i state as *depth i transition*. In their encoding, the current state 0 in the TCAM is replaced with "$*\,*\,**$," which is matched with any state, and all depth 1 transitions are eliminated in the TCAM since they are covered by the entries with current state "$*\,*\,**$." Since most of cross transitions in the AC DFA are at depth 1, this encoding reduces significantly the number of TCAM entries. In their work, they can also propose state encoding schemes eliminating the higher depth transitions by suffixing the state encoding by the prefix string from the root node to that state such as "$****h$" and "$****he$." In order to eliminate all the depth $\leq m$ transitions, the current state field requires $(m-1)$ bytes additionally. Thus, these encoding schemes are inefficient since they require encoding the states with a larger width.

Fig. 3 shows the TCAM entries in several encodings: no optimization and Alicherry's codings with several depths. In


no optimization

	cs	in	ns
√	0	h	1
√	0	s	3
	1	e	2
	1	i	6
	2	r	8
	3	h	4
	4	e	5
	6	s	7
	8	s	9

	cs	in	ns
x	1	h	1
x	1	s	3
x	2	h	1
x	2	s	3
x	3	s	3
x	4	h	1
x	4	s	3
	4	i	6
x	5	h	1
x	5	s	3
	5	r	8
x	6	h	1
	7	h	4
x	7	s	3
x	8	h	1
	9	h	4
x	9	s	3

(a)

Alicherry's encoding

depth-1

	cs	in	ns
√	1	e	2
√	1	i	6
	2	r	8
√	3	h	4
	4	e	5
	6	s	7
	8	s	9
x	4	i	6
	5	r	8
x	7	h	4
x	9	h	4
	*	h	1
	*	s	3

depth-2

	cs	in	ns
√	2e	r	8r
√	4h	e	5e
√	6i	s	7s
	8r	s	9s
x	5e	r	8r
	*h	e	2e
	*h	i	6i
	*s	h	4h
	**	h	1h
	**	s	3s

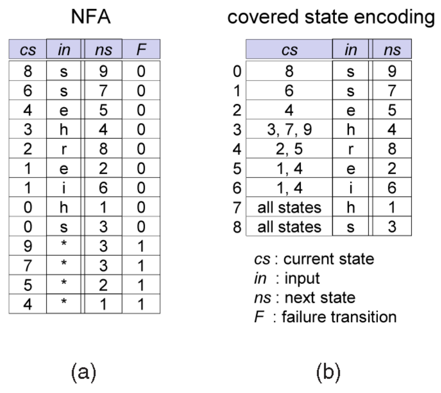
depth-3

cs	in	ns
8er	s	9rs
*he	r	8er
*sh	e	5he
*hi	s	7is
**h	e	2he
**h	i	6hi
**s	h	4sh
***	h	1-h
***	s	3-s

cs : current state
in : input
ns : next state

(b)

Fig. 3. TCAM/SRAM entries in several schemes.


NFA

cs	in	ns	F
8	s	9	0
6	s	7	0
4	e	5	0
3	h	4	0
2	r	8	0
1	e	2	0
1	i	6	0
0	h	1	0
0	s	3	0
9	*	3	1
7	*	3	1
5	*	2	1
4	*	1	1

(a)

covered state encoding

	cs	in	ns
0	8	s	9
1	6	s	7
2	4	e	5
3	3, 7, 9	h	4
4	2, 5	r	8
5	1, 4	e	2
6	1, 4	i	6
7	all states	h	1
8	all states	s	3

cs : current state
in : input
ns : next state
F : failure transition

(b)

Fig. 4. TCAM/SRAM entries in NFA-based architecture.

the entries with no optimization, the left entries are goto transitions in the NFA and the right entries are cross transitions. The entries marked as "×" are cross transitions eliminated when the depth of the state encoding is increased. The entries marked as "√" are goto transitions whose current state field is replaced with the state including "don't care" when the depth of the state encoding is increased. Note that the TCAM entries containing "don't care" must be located in the last positions since they have the lowest priority.

Recently, Bremler-Barr et al. [11] proposed a novel method called *CompactDFA* to compress the DFA entries in the TCAM-based implementation. Although CompactDFA originated from essentially the same idea as Alicherry's encoding which eliminates cross transitions by suffixing the state encoding by the prefix string from the root node to that state, it provides more efficient encoding scheme than Alicherry's encoding by constructing the common suffix tree and encoding the states, and it eliminates all the cross transitions. However, CompactDFA state encoding is still nonoptimal and its building algorithms are more or less complex since the CompactDFA is built from AC DFA.

In this paper, we propose a new state encoding scheme in TCAM-based implementation of AC NFA. We can reduce the number of TCAM entries to the number of goto transitions in the NFA by fully utilizing the "don't care" feature of TCAM using a novel state encoding.

3 Covered State Encoding Scheme

3.1 Basic Idea

Since the AC NFA has a smaller number of transitions than the AC DFA, the NFA can be implemented with smaller TCAM entries than the DFA. The number of TCAM entries in the NFA-based implementation is the sum of the number of goto transitions and the number of nontrivial failure transitions in the NFA.

In the NFA-based architecture, 1-bit field F indicating a failure transition is added in each SRAM entry. If an entry is associated with a failure transition, its F is 1 and its input field is "$*$" which can match with any input value. Fig. 4a shows TCAM/SRAM entries for the AC NFA in Fig. 1a. If the matched transition is a failure transition, or $F = 1$, an input is not advanced and current input is used again at the next matching. One character may be repeatedly processed along the states in the failure transition path until a nonfailure transition is matched ($F = 0$) or a state goes back to the initial state, which is a major disadvantage of the NFA-based architecture.

If we exploit the "don't care" feature of a TCAM, we can encode states so that a code covers some other codes. For example, code $****$ covers all of 4-bit codes and code $11**$ covers four codes 1100, 1101, 1110, and 1111. We call a code each bit of which consists of 0 or 1 by a *unique* code, and a code each bit of which consists of 0, 1, or "$*$" by a *cover* code. Since the next state field in the SRAM cannot store "$*$," it should be represented by a unique code. However, the current state field in the TCAM can store a cover code.

The failure transition graph is a graph consisting of only failure transitions in the AC NFA. In a failure transition graph, the set of all the predecessors of a state s is denoted by $PRED(s)$ and the set of all the successors of a state s is denoted by $SUCC(s)$. Fig. 5 shows $SUCC(s)$ and $PRED(s)$ for a state s in a failure transition graph. If current state fields of TCAM entries associated with each state t are replaced with a cover code to cover not only state t but also all its predecessors $PRED(t)$, we can simultaneously examine all the entries of s and $SUCC(s)$ for a current state s since s is a predecessor of $SUCC(s)$ and the failure transition entries are not needed any longer.

Fig. 4b shows TCAM/SRAM entries in a new state encoding. In this figure, each current state field also includes predecessor states which it should cover. Note that the entries of a state s should be located more front than the entries of $SUCC(s)$ since it has the priority. In this example, assume that the current state is 4. The current state 4 can match three entries 2, 5, and 6. If input is "e," two entries 2 and 5 are exactly matched and the index of the front entry 2

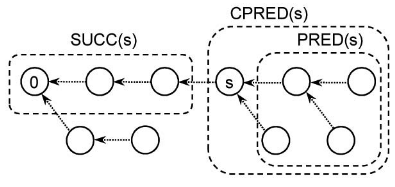


Fig. 5. $SUCC(s)$ and $PRED(s)$ in a failure transition graph.

is outputted. If input is "i," one entry 6 is matched. If input is "h" or "s," two entries 7 and 8 are matched. Otherwise, there is no matched entry and the next state becomes the initial state 0. Since this operation is performed at one step in the new scheme, the NFA can also process a text string with the same number of transitions as the DFA.

We call a new encoding scheme to make the basic idea possible by *covered state encoding*. In the covered state encoding, each state s has two state encoding: a unique code $u\_code(s)$ used in the next state field and a cover code $c\_code(s)$ used in the current state field. Let $CPRED(s) = PRED(s) \cup \{s\}$. In covered state encoding, a code of each state should be assigned so that the following conditions are satisfied:

1. $c\_code(s)$ should cover $u\_code(t)$ for any state t in $CPRED(s)$.
2. $c\_code(s)$ should not cover $u\_code(t)$ for any state t which is not in $CPRED(s)$.

3.2 Covered State Encoding Algorithm

Now, we propose an algorithm for performing the covered state encoding for the AC NFA. The proposed algorithm consists of four stages as follows:

- Step 1. Construct a failure tree.
- Step 2. Determine the size (or dimension) of a cover code of each state.
- Step 3. Assign a unique code and a cover code to each state.
- Step 4. Build the TCAM entries.

3.2.1 Construction of a Failure Tree

The failure transition graph is a graph consisting of only failure transitions in the AC NFA. We can avoid making unnecessary failure transitions by using a generalization of the next function from [17]. At first, we construct a failure tree by reversing failure transitions in the failure transition graph so that each state s can easily find $PRED(s)$. The initial state becomes a root of the failure tree and the set of all descendants of a state s in the failure tree is $PRED(s)$.

3.2.2 Determining the Size of a Cover Code

The number of "$*$" bits in a cover code is called its *dimension* and the number of unique codes covered in a cover code is called its *size*. The dimension and size of the cover code of a state s are represented by $dim(s)$ and $size(s)$, respectively. $size(s)$ is $2^{dim(s)}$. If a state s has no child in a failure tree, $dim(s)$ is 0 since $c\_code(s)$ need not cover any other code, and its cover code is the same as its unique code. If a state s has any children in a failure tree, $c\_code(s)$ should cover not only $u\_code(s)$ but also $c\_code(c)$ for its all children c. The $size(s)$ is greater than or equal to $1 + \sum_c size(c)$, where c is a child. Therefore, the dimension of a state s is determined by the following equation:

$$dim(s) = \begin{cases} \lceil \log_2(1 + \sum_c 2^{dim(c)}) \rceil, & \text{if } s \text{ has children } c, \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

The dimensions of all states are recursively determined during the calculation of $dim(0)$, where 0 indicates the initial state.

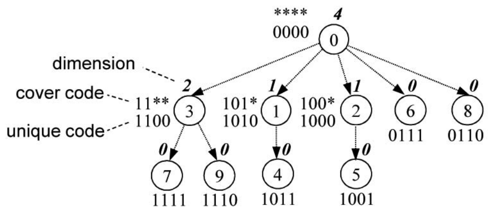


Fig. 6. The failure tree and covered state encoding.

3.2.3 Assigning State Codes

For a state s, the unique code $u\_code(s)$ can use a code covered by $c\_code(s)$ and we use a code obtained by replacing $*$ with 0 in $c\_code(s)$ as $u\_code(s)$. The codes are recursively assigned. At first, the code of the root 0 is assigned as follows: $c\_code(0) = ** \cdots *$ and $u\_code(0) = 00\ldots0$. For a state s, the codes of its children are assigned in decreasing dimension order in order to assign codes efficiently. The code of each child is assigned from upper codes in the range covered by the $c\_code(s)$.

Fig. 6 shows the failure tree for AC NFA in Fig. 1a. In the example, the dimension of a root is 4 and it has five children whose dimensions are 2, 1, 1, 0, and 0, respectively. The cover codes of the children are assigned $11**$, $101*$, $100*$, 0111, and 0110. Their unique codes are 1100, 1010, 1000, 0111, and 0110 which are obtained by replacing $*$ with 0.

The codes of children share the fixed bit values (0 or 1) of the cover code of their parent and assign new values in "don't care" bit locations. The procedure *AssignCode* performs Step 3 as described above.

```
procedure AssignCode(node $s$, int $base$)
    // assign codes to this node $s$
    $u\_code[s] = base$ // unique code
    $c\_code[s] = \text{covercode}(base, dim(s))$ // cover code
    if $s$ has no child then return
    sort the children of node $s$ in decreasing dimension order
    $cbase = base + 2^{dim(s)}$
    // assign codes to children recursively
    for each child node $c$ of node $s$ do
        $cbase = cbase - 2^{dim(c)}$
        AssignCode($c$, $cbase$)
    endfor
end
```

In the AssignCode procedure, covercode($base, dim$) returns the value where the lowest dim bits of the $base$ are replaced with a $*$. For example, $\text{covercode}(1100,\ 2) = 11**$. Calling AssignCode(0,0) assigns the codes of all the states recursively.

3.2.4 Building TCAM Entries

Any child entry in a failure tree should be located more front than its parent entry in TCAM. The procedure *BuildTCAM* builds TCAM/SRAM entries so that this requirement is satisfied.

```
procedure BuildTCAM(node $s$)
    // insert entries of children recursively
    for each child node $c$ of node $s$ do
```

current st	in	next st
11** (3)	h	1011 (4)
1011 (4)	e	1001 (5)
101* (1)	e	1000 (2)
101* (1)	i	0111 (6)
100* (2)	r	0110 (8)
0111 (6)	s	1111 (7)
0110 (8)	s	1110 (9)
**** (0)	h	1010 (1)
**** (0)	s	1100 (3)

Fig. 7. TCAM/SRAM entries using cover state encoding.

```
        BuildTCAM($c$)
    // insert the entry of this node $s$
    for each goto transition $g(s, i)$ of node $s$ do
        $next = g(s, i)$
      insert ($c\_code[s]$, $i$) into TCAM
      insert ($u\_code[next]$) into SRAM
    endfor
end
```

The procedure BuildTCAM guarantees the correct order of entries since it inserts children in the failure tree first into TCAM. Calling BuildTCAM(0) constructs all the TCAM/SRAM entries recursively.

Example. Fig. 7 shows the TCAM entries using covered state encoding, constructed by the proposed algorithm for AC NFA in Fig. 1a. Fig. 8 shows the state transitions for the input sequence "*shershiss*". The initial state is 0. After processing three input characters "*she*", the state becomes 5. Although there is no entry of state 5, the TCAM entry of state 2 ($c\_code = 100*$) is matched with input r in state 5 ($u\_code = 1001$) and the state becomes 8. After processing the following input sequence "*shiss*" from state 8, the state becomes 3. Boxes in Fig. 8 represent the cases that an input is matched with the TCAM entry of a failure transition state of the current state.

4 Evaluation of Covered State Encoding

4.1 Memory Requirement

We used the Snort rule set version 2.8 [18] and ClamAV version 0.96 antivirus signature [19] to evaluate the TCAM memory requirement of the covered state encoding. The Snort rule set has 5,169 patterns whose average length is 16.7 and ClamAV signatures have 30,385 patterns whose average length is 67.4.

We compare the implementations of the AC DFAs using Alicherry's encoding of depth m (denoted by DFA-m) and the CompactDFA with the implementation of the AC NFA using the covered state encoding (denoted by AC NFA-c).

Let T_g and T_f be the number of goto transitions and the number of failure transitions in the AC NFA, respectively. In the AC NFA-c, the TCAM entries consist only goto

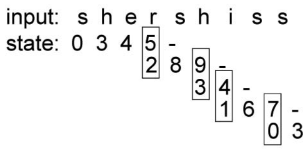


Fig. 8. State transitions for a given input sequence.

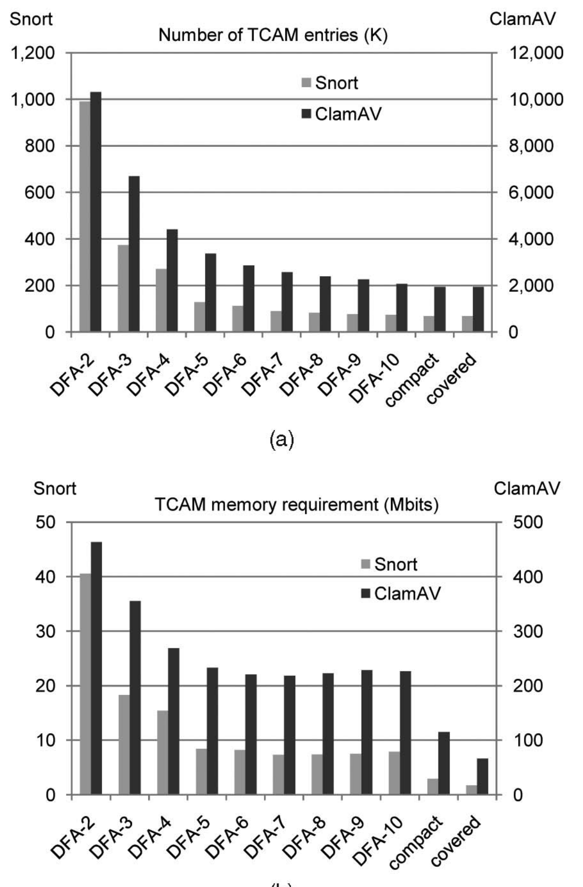


Fig. 9. Comparison of some state encoding schemes. (a) The number of TCAM entries. (b) TCAM memory requirement.

transitions and the number of TCAM entries is $T = T_g$. The number of TCAM entries in the CompactDFA is the same as that in the AC NFA-c since the CompactDFA eliminates all the cross transitions. Let $T_{c,i}$ be the number of depth i transitions in the AC DFA. In the DFA-m, the number of TCAM entries is $T = T_g + \sum_{i=m+1}^{d_{max}} T_{c,i}$, where d_{max} is the maximum depth.

Fig. 9a shows the numbers of TCAM entries in various DFA-m ($2 \leq m \leq 10$), CompactDFA, and AC NFA-c for the Snort rule set and ClamAV signatures. As m increases, the number of TCAM entries in DFA-m approaches that of AC NFA-c.

In the DFA-m, the number of bits required for the state encoding is $E = \log_2 S + 8(m-1)$ bits where S is the number of states [1]. In the AC NFA-c, the state code width is $E = \log_2 S + \delta$, where δ is the number of extra bits required in the covered state encoding and depends on the patterns. In the covered state encoding, the number of state bits E is equal to the number of states S at the worst case. A set of n strings $\{a_1b_1, a_2a_1b_2, \ldots, a_na_{n-1}\ldots a_1b_n\}$, where $b_i \neq b_j$ if $i \neq j$, is an example of the worst case. This case, however, is almost impossible for practical rule sets. The state code width in the CompactDFA is also represented by $E = \log_2 S + \delta$, where δ is the number of extra bits required in the CompactDFA.

Table 1 presents the state code widths and extra bit widths for Snort and ClamAV. In the AC NFA-c, the number of extra bits δ (0, 5) is much smaller than $\log_2 S$. In the CompactDFA,

TABLE 1
The Code Widths for Snort and ClamAV

pattern set	states S ($\log_2 S$)	code width E (extra δ) covered	compact
Snort	68,155 (17)	17 (0)	35 (18)
ClamAV	1,945,802 (21)	26 (5)	51 (30)

however, the extra bit width δ (18, 30) is comparable to $\log_2 S$. Thus, the covered state encoding is more efficient than the CompactDFA and Alicherry's encoding.

The width of a TCAM entry is $W = E + 8$ since a TCAM entry consists of a current state and an 8-bit input data. The TCAM memory requirement is $M = T \cdot W = T \cdot (E + 8)$ bits. Thus, the TCAM memory requirements of AC NFA-c, CompactDFA, and DFA-m are given by following equations:

$$M_{covered} = M_{compact} = T_g \cdot (\log_2 S + \delta + 8),$$
$$M_{DFA-m} = \left(T_g + \sum_{i=m+1}^{d_{max}} T_{c,i}\right)(\log_2 S + 8(m-1) + 8).$$

The SRAM memory requirement is $T \cdot E$ bits since an SRAM entry consists of the next state field.

Fig. 9b shows the TCAM memory requirements for DFA-m ($2 \leq m \leq 10$), CompactDFA, and AC NFA-c. The DFA-1 is omitted in this figure due to its large value. This figure shows that the TCAM memory requirement in the covered state encoding is much less than those in the other schemes. For the Snort rule set, the TCAM memory requirement of DFA-m has the minimum value ($= 7.3$ Mbit) when $m = 7$, which is 4.3 times of $M_{covered}(= 1.7 \text{ Mbit})$ and the ClamAV signatures have the minimum $M_{DFA-m}(= 218 \text{ Mbit})$ when $m = 7$, which is 3.3 times of $M_{covered}(= 66 \text{ Mbit})$. $M_{covered}$ is about 58 percent of $M_{compact}$ for both Snort and ClamAV. Thus, the TCAM memory requirement of AC NFA-c is the smallest. In the AC NFA-c, the memory utilization per character is 2.47 B/char for the Snort rule set and 4.04 B/char for ClamAV signatures.

The commercially available TCAMs have the fixed entry widths which are multiples of a specific size D (either 36 or 40 bits). When the required TCAM entry width is W, the actual TCAM entry width is $\lceil \frac{W}{D} \rceil \cdot D$. In the AC NFA-c, the actual TCAM entry widths are D for both Snort and ClamAV since the required entry widths are $W = 17 + 8 = 25$ for Snort and $W = 26 + 8 = 34$ for ClamAV. In the DFA-m ($m \geq 3$) and CompactDFA, the actual TCAM entry widths are $2D$ or more since the TCAM entry widths are larger than 40. Thus, the actual TCAM memory requirement in the AC NFA-c is also the smallest since the number of TCAM entries and the TCAM entry width in the AC NFA-c are smaller than in the other schemes.

4.2 Time Complexity

The lookup operation in the TCAM-based implementation using the covered state encoding requires N transitions for an input string with data length N since there is no failure transition. This lookup speed is the same as the DFA-based architecture.

The algorithm building all TCAM entries using the covered state encoding takes at most $O(n \log n)$ time, where n is the number of states. This result is obtained as follows: Step 1 requires obviously n steps. The time complexity of Steps 2 and 4 is $O(n)$, which is that of tree traversal algorithm. Step 3 includes sorting its children in decreasing dimension order at each node. Let m_i be the number of children of node i. The sorting at each node takes $O(m_i \log m_i)$. The sum of all m_i's is $n - 1$, in other words, $\sum_i m_i = n - 1$. The time complexity of the procedure AssignCode is as follows:

$$O\left(\sum_i m_i \log m_i\right) \leq O\left(\sum_i m_i \log n\right) = O(n \log n).$$

The algorithm consists of four steps performing sequentially and the time complexity of Step 3 is larger and dominant.

5 Multicharacter Processing Using Covered State Encoding

We have proposed the covered state encoding scheme for the efficient TCAM-based implementation of Aho-Corasick algorithm. However, Aho-Corasick algorithm processes only one character at a time and multicharacter processing is required to achieve high-speed matching.

Alicherry et al. constructed the compressed DFA (cDFA) that has transitions on multiple input characters, by combining k consecutive states of Aho-Corasick DFA in addition to proposing a state encoding scheme using properties of the TCAM [1]. In the compressed DFA, however, each transition corresponds to variable length of input characters (at most k) and in rare cases, the input pointer may be moved backward. Moreover, the compressed DFA cannot use the covered state encoding in the TCAM-based implementation because it has no failure transition.

In this section, we propose constructing a finite state machine called *k-AC NFA* which has state transitions on k input characters by combining k consecutive goto transitions of the Aho-Corasick NFA. Since k-AC NFA consists of goto transitions and failure transitions like the AC NFA, the covered state encoding scheme can be used in the TCAM-based implementation of the k-AC NFA. The major advantage of the k-AC NFA is that the state transition consumes exactly k input characters while in the compressed DFA, the state transition is done on variable length (between 1 and k) of characters.

5.1 Construction of k-AC NFA

The basic method constructing the k-AC NFA is similar to the hardware-based method proposed in [12]. When k input characters are processed at a time, the patterns can be started at one of k possible offsets of the input characters. In order to detect the patterns starting at any of k offsets, we construct k finite state machines each of which corresponds to one of k offsets.

For example, we consider the set of patterns {*abc*, *xyapq*, *pqrxyz*}. Fig. 10a shows the AC NFA for these patterns. We construct the 4-AC NFA by creating four state machines each of which can detect the patterns starting at one of four offsets, as shown in Fig. 10b. In Fig. 10, the states with the same label are the same state and the gray colored states are output states.

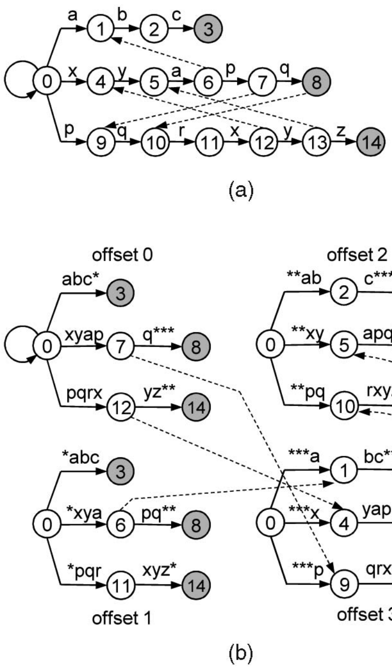


Fig. 10. Construction of k-AC NFA ($k = 4$). (a) AC NFA. (b) k-AC NFA.

In Fig. 10b, the dotted lines represent the failure transitions. The failure function of the k-AC NFA is the same as that of the AC NFA and it is proved as follows:

Theorem 1. *The failure function $f_k(s)$ of the k-AC NFA is the same as the failure function $f(s)$ of the corresponding AC NFA.*

Proof. Let $string(s)$ be the sequence of input characters of consecutive goto transitions from the initial state to state s in the AC NFA. The failure function $f(s) = t$ if and only if $string(t)$ is the longest suffix of $string(s)$ [16]. All the states in the AC NFA also exist in the k-AC NFA. The goto function $g_k(s, a_1a_2 \dots a_k)$ of the k-AC NFA corresponds to k input characters. The $string_k(s)$ of k-AC NFA is the same as $string(s)$ of the AC NFA if $*$s are not included in $string_k(s)$ among the k input characters of goto transitions. Since $string_k(s)$ and $string(s)$ are the same, the failure function $f_k(s)$ of k-AC NFA is the same as the failure function $f(s)$ of the corresponding AC NFA. □

The optimization of the failure function is performed in the same way as the AC NFA except using $g_k()$ instead of $g()$ as the goto function.

5.2 Implementation of k-AC NFA

The k-AC NFA can be implemented in the TCAM-based architecture using the covered state encoding. In the k-AC NFA, start transitions from the initial state may be done on inputs with leading $*$s (e.g., $**pq$) and output transitions may be done on inputs with trailing $*$s (e.g., $bc**$). Both transitions may be simultaneously matched with k input characters (e.g., $bcpq$) and we cannot determine the priority between two transitions since they have no inclusion relationship.

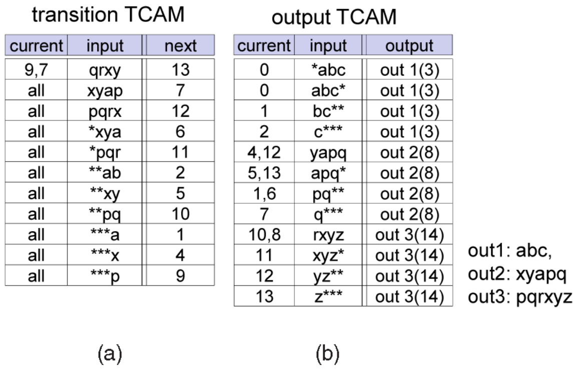
In order to solve this problem, we use two TCAMs called a *transition TCAM* and an *output TCAM*, which can simultaneously generate the matched entries. The output transitions are stored in the output TCAM. Since the end output states do not have goto transitions, the end output transitions do not proceed the state transition further. Therefore, the transition TCAM stores all the transitions except end output transitions. The nonend output transitions are stored in both TCAMs. The output TCAM detects the final pattern matching and the transition TCAM controls the state transition in the k-AC NFA.

Since the start transitions on inputs with leading $*$s and the end output transition on inputs with trailing $*$ are stored in the transition TCAM and the output TCAM, respectively, two transitions can generate the matching results simultaneously. If two TCAMs generate the matching results simultaneously, the state is moved according to the result of the transition TCAM and the output is generated by the output TCAM.

Example. Fig. 11 shows entries of two TCAMs in the implementation using the covered state encoding of the k-AC NFA in Fig. 10. Fig. 12 shows the state transitions for a given input data stream in this implementation. In Fig. 12, the states inside parenthesis represent the states which covers the current state and has the input field matched with the input characters. For example, at state 13, the input $apqr$ is not matched with the entry of state 13 (the last entry of output TCAM), but matched with an output TCAM entry $apq*$ of state 5, which generates the output of state 8 ($xyapq$). It is also simultaneously matched with a transition TCAM entry $*pqr$ of the initial state, which makes state transition to 11.

5.3 Evaluation of the Implementation of k-AC NFA

We evaluate the TCAM-based implementation of k-AC NFA using the covered state encoding. The k-AC NFA of N patterns has N output states. Since each output state has k output transitions for all offsets in the k-AC NFA, the number of output TCAM entries is $T_o = kN$. Let N_e be the number of *end* output states which have no goto transition.



transition TCAM

current	input	next
9,7	qrxy	13
all	xyap	7
all	pqrx	12
all	*xya	6
all	*pqr	11
all	**ab	2
all	**xy	5
all	**pq	10
all	***a	1
all	***x	4
all	***p	9

(a)

output TCAM

current	input	output
0	*abc	out 1(3)
0	abc*	out 1(3)
1	bc**	out 1(3)
2	c***	out 1(3)
4,12	yapq	out 2(8)
5,13	apq*	out 2(8)
1,6	pq**	out 2(8)
7	q***	out 2(8)
10,8	rxyz	out 3(14)
11	xyz*	out 3(14)
12	yz**	out 3(14)
13	z***	out 3(14)

out1: abc,
out2: xyapq
out3: pqrxyz

(b)

Fig. 11. TCAM entries in the k-AC NFA.

```
input:  abxp  qrxy  apqr  xyza  bccc  xyap  qrxy  zddd ...
state:  0     9     13    11    1     0     7     13    0
                    (5,0) (0)   (0)         (9)
output:                   8     14    3           8     14
```

Fig. 12. An Example of state transitions in k-AC NFA.

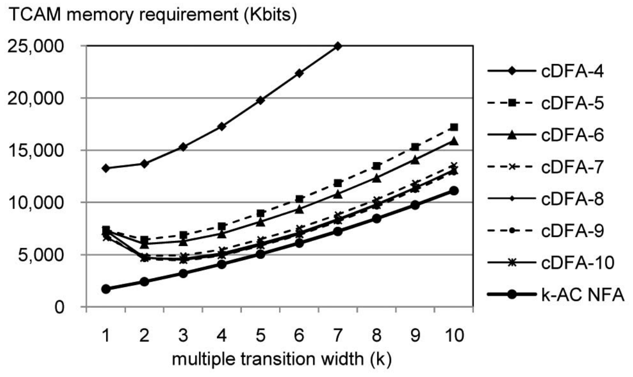


Fig. 13. TCAM memory requirement of multicharacter processing.

Since goto transitions to end output states are not included in the transition TCAM, the number of transition TCAM entries is $T_t = T_g - N_e$, where T_g is the number of goto transitions in the AC NFA. The total number of TCAM entries is $T = T_t + T_o = T_g - N_e + kN$. If $N_e = N$, then $T = T_g + (k-1)N$.

Since the failure function of the k-AC NFA is the same as that of the AC NFA, the bit size of encoded states is the same as that of the AC NFA, $E = \log_2 S + \delta$, where S is the number of states and δ is the extra bit size of the covered state encoding. The TCAM entry width of the k-AC NFA using the covered state encoding is $W = E + 8k = \log_2 S + \delta + 8k$, where $8k$ is the input data width. The total TCAM memory requirements of k-AC NFA are given as follows:

$$M_{covered,k} = (T_g - N_e + kN)(\log_2 S + \delta + 8k).$$

In the compressed DFA proposed by Alicherry, transitions consist of compressed goto transitions and depth-i transitions for all $i > m$. The number of TCAM entries in cDFA-m is $T = T_{g,k} + \sum_{i=m+1}^{d_{max}} T_{c,i}$, where $T_{g,k}$ is the number of the compressed transitions in cDFA and depends on multiple transition width k. The TCAM entry width is $W = \log_2 S + 8(m-1) + \log_2 k + 8k$, where $\log_2 k$ is the bit size for shift value. The total TCAM memory requirement is given by the following equation:

$$M_{cDFA-m,k} = \left(T_{g,k} + \sum_{i=m+1}^{d_{max}} T_{c,i}\right)(\log_2 S + 8(m-1) + \log_2 k + 8k).$$

We evaluate the TCAM memory requirements of multi-character processing schemes k-AC NFA and the compressed DFA with depth m (cDFA-m) for the Snort rule set v2.8. Fig. 13 shows the TCAM memory requirements of various schemes for each k. Due to the efficiency of the covered state encoding, the memory requirement of k-AC NFA is less than that of any compressed DFA, where the compressed DFA has the minimum memory requirement when $m = 9$ except $k = 1$.

We investigate the pattern matching speed. In the k-AC NFA, after k input characters are compared, the input pointer is always advanced to exactly k characters. while the compressed DFA has variable length transition width and the average transition width is about $0.8k$ or less. In the k-AC NFA using a covered state encoding, pattern matching operation for an input string with length N can be done in $\lceil N/k \rceil$ transitions Thus, the k-AC NFA using a covered state encoding is superior to the compressed DFA-m in both the TCAM memory requirement and the pattern matching speed.

6 Conclusion

In this paper, we proposed a covered state encoding scheme for the TCAM-based implementation of Aho-Corasick algorithm, which is a multiple pattern matching algorithm widely used in network intrusion detection systems. The covered state encoding takes advantage of "don't care" feature of TCAMs and information of failure transitions is implicitly captured in the covered state encoding. If we use the covered state encoding, the failure transitions do not need to be implemented as TCAM entries since all the states in the failure transition path can be simultaneously examined. The covered state encoding requires the smaller number of TCAM entries than other schemes since it is used in NFA-based implementation and the failure transitions are not needed. The time complexity of the algorithm building TCAM entries using the covered state encoding is $O(n \log n)$, where n is the number of states. Thus, the covered state encoding enables an efficient TCAM-based implementation of a multipattern matching algorithm.

We also proposed the scheme constructing the finite state machine called k-AC NFA for multicharacter processing, which can use a covered state encoding. The k-AC NFA using the covered state encoding has the smaller TCAM memory requirement and can process exact k characters at a time. Thus, a covered state encoding can be efficiently used in multicharacter processing.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No.2009-0077793).

References

[1] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High Speed Pattern Matching for Network IDS/IPS," *Proc. 14th IEEE Int'l Conf. Network Protocols (ICNP)*, vol. 11, pp. 187-196, 2006.

[2] M. Alicherry and M. Muthuprasanna, "Method and System for Multi-Character Multi-Pattern Pattern Matching," US Patent Application No. 20080046423, Feb. 2008.

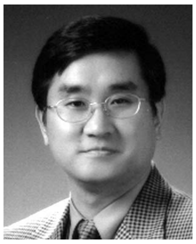
[3] M. Gould, R. Barrie, D. Williams, and N. de Jong, "Apparatus and Method for Memory Efficient, Programmable, Pattern Matching Finite State Machine Hardware," US Patent No. 7082044 B2, July 2006.

[4] M. Gao, K. Zhang, and J. Lu, "Efficient Packet Matching for Gigabit Network Intrusion Detection Using TCAMs," *Proc. 20th Int'l Conf. Advanced Information Networking and Applications (AINA)*, 2006.

[5] F. Yu, R. Katz, and T. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," *Proc. 12th IEEE Int'l Conf. Network Protocols (ICNP '04)*, pp. 174-183, 2004.

[6] Y. Weinsberg, S. Tzur-David, D. Dolev, and T. Anker, "High Performance String Matching Algorithm for a Network Intrusion Prevention System (NIPS)," *Proc. IEEE High Performance Switching and Routing (HPSR)*, pp. 147-154, 2006.

[7] S. Dharmapurikar, M. Attig, and J. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," *IEEE Micro,* vol. 24, no. 1, pp. 52-61, Jan./Feb. 2004.
[8] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," *Proc. IEEE INFOCOM,* vol. 4, pp. 2628-2639, 2004.
[9] L. Tan, B. Brotherton, and T. Sherwood, "Bit-Split String-Matching Engines for Intrusion Detection and Prevention," *ACM Trans. Architecture and Code Optimization,* vol. 3, no. 1, pp. 3-34, 2006.
[10] J. van Lunteren, "High-Performance Pattern-Matching for Intrusion Detection," *Proc. IEEE INFOCOM,* vol. 4, 2006.
[11] A. Bremler-Barr, D. Hay, and Y. Koral, "CompactDFA: Generic State Machine Compression for Scalable Pattern Matching," *Proc. IEEE INFOCOM,* 2010.
[12] C. Clark and D. Schimmel, "Scalable Pattern Matching for High Speed Networks," *Proc. 12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* 2004.
[13] B. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *Proc. 10th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* pp. 111-120, 2002.
[14] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded Cams for Efficient and High-Speed NIDS Pattern Matching," *Proc. 12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* pp. 258-267, 2004.
[15] Y.H. Cho, S. Navab, and W.H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," *Proc. 12th Int'l Conf. Field-Programmable Logic and Applications (FPL),* pp. 337-357, 2002.
[16] A. Aho and M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Comm. ACM,* vol. 18, no. 6, pp. 333-340, 1975.
[17] D.E. Knuth, J. James, H. Morris, and V.R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Computing,* vol. 6, no. 2, pp. 323-350, 1977.
[18] SNORT Official Web Site, http://www.snort.org, 2011.
[19] ClamAV Official Web Site, http://www.clamav.net, 2011.



SangKyun Yun received the BS degree in electronics engineering from Seoul National University, Korea, in 1984, and the MS and PhD degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 1986 and 1995, respectively. He is a professor in the Department of Computer and Telecom. Engineering, Yonsei University, Wonju, Korea. He worked at Hyundai Electronics (now Hynix), Korea, from 1986 to 1990. He was on the faculty of Seowon University, Cheongju, Korea, from 1992 to 2001. He was a visiting scholar in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, and the Department of Electrical Engineering at the University of Texas at Austin. His interests include embedded systems, reconfigurable systems, network security, and computer architecture. He is a member of the IEEE Computer Society.