A Deterministic Cost-effective String Matching Algorithm for Network Intrusion Detection System

Nen-Fu Huang
Department of Computer Science and
Inst. of Communication Engineering
National Tsing-Hua University
Hsin-Chu, Taiwan

Yen-Ming Chu
Institute of Communication Engineering
National Tsing-Hua University
Hsin-Chu, Taiwan

Chen-Ying Hsieh, Chi-Hung Tsai and Yih-Jou Tzang
Dep. of Computer Science
National Tsing-Hua University
Hsin-Chu, Taiwan

Abstract—Network Intrusion Detection Systems (NIDS) are more and more important in today’s network security for identifying and preventing malicious attacks over the network. This paper proposes a novel and effective string matching algorithm (named ACMS) with advantages of both compact memory and high performance. By employing the characteristics of magic states observed from the deterministic finite state automata, the proposed ACMS significantly reduces the memory requirement without sacrificing high speed no matter it is implemented in software or hardware. The ACMS algorithm also provides high flexibility that it can be tuned to fit specific performance requirement and resource constraints. The experimental results show that the performance of ACMS is over 3.5 times in hardware implementation and 21 times in software implementation better than that of the state-of-the-art studies.

Keywords-Automaton; Network Security; NIDS; String Matching

I. INTRODUCTION

NIDS is designed to detect and identify threats, such as worms, virus, spyware, and malicious codes, by performing deep packet inspection on packet payloads. The performance of signature-based NIDS is dominated by the speed of string matching algorithm used to compare the packet header and payload with signatures. For instance, Snort, an open source NIDS, takes over 2,500 patterns as signatures and takes more than 80% of CPU time for string matching [1]. A NIDS needs a fast string matching algorithm or other mechanisms to improve its performance. Otherwise, an under-performing system not only becomes the network bottleneck but also misses some critical attacks.

String matching algorithms have been widely studied for the past years. For example, the Boyer-Moore algorithm solved single-pattern string matching problem [2], and the Aho-Corasick (AC) [3] and Wu-Manber [4] algorithms solved multi-pattern string matching problems. Recently, owing to the requirements of deep packet inspection for network security reasons, various new concepts and algorithms have been introduced and implemented, such as Bitmap AC [5], Bit-Split AC [6], Delay Input DFA (D\^{}F\^{}A) [7], BFSM-based pattern matching (BFPM) [8], parallel bloom-filter [9], reconfigurable silicon hardware [10] and TCAM-based mechanism [11].

Nevertheless, most of the presented studies usually can not properly balance the following trade-offs. (1) Average-case vs. worst-case. Among classical string matching algorithms, the Wu-Manber’s algorithm is accredited as the best performance one. However, it is fact that Wu-Manber’s worst-case performance is much worse [4, 5]. Thus, the algorithm itself may become the victim of denial of service (DoS) attacks, in case the attack traffic are special designed to trigger the worst case conditions. For this reason, many AC-based algorithms, which had the best worst-case computational time-complexity, had been proposed and widely adopted for further researches [5-8]. (2) Speed vs. space. The main issue of string matching algorithms is the trade-off between speed and space. A faster algorithm usually needs larger memory space to store the data structures. It has been pointed out that it is not easy to get good balance between them [5, 12]. Some researches tried to decrease the memory usage by putting their data structures into the on-chip memory, to speed up the memory access and improve performance. (3) Software vs. hardware. Some string matching algorithms were designed for software implementation, but some were for hardware implementation. The design concept for data structures and algorithms are usually different [6-8, 10-11].

Therefore, this paper proposes a cost-effective high-speed string matching accelerator to solve the above mentioned issues. The rest of the paper is organized as follows. In Section II, past researches on automaton-based pattern matching algorithms and related studies are introduced. Section III presents the proposed pattern matching algorithm. The evaluation and experiments of the proposed algorithm are depicted in Sections IV and V, respectively. Finally, the conclusions are given in Section VI.

II. RELATED WORK

The AC algorithm guarantees linear time-complexity in worst-case situation [3]. The algorithm works with two kinds of data structures: deterministic finite state automaton (DFA) and non-deterministic finite state automaton (NFA). In DFA, only single memory reference is needed for each input character, and the time-complexity is guaranteed to be $O(n)$, where $n$ is the input string length. But the memory requirement is large in this way. On the contrary, in NFA, the time-complexity becomes $O(n + k)$, where $k$ is the times of state transitions with failure path.

Due to the extremely large memory requirement when DFA is adopted in AC algorithm, N. Tuck et al proposed a method called AC-BITMAP [5] in which the “bitmap” data structure is applied to reduce the memory usage for NFA. For every state,
a bitmap is used to record whether the next state exists or not for every symbol. In this way, the memory usage is decreased dramatically. Nevertheless, due to the time cost for \texttt{popcount} function and link list searching is expensive, the performance of AC-BITMAP implemented in software is bad.

To accelerate AC algorithm, Toshio Nishimura proposed a method to rearrange states [13]. Since the states near to initial state are used more frequently, they will be gathered into the cache of CPU and therefore the CPU cache is utilized in a more efficient way.

III. THE PATTERN-MATCHING WITH MAGIC STATE

A. Automaton Mode

Generally, the model of automaton-based model has two stages. Stage-1 is responsible for searching the next state in the state table by an index composed of current state and input symbol. Stage-2 takes the task that it identifies if the output next state from Stage-1 is an acceptance state and to fetch the corresponding matched pattern IDs.

To fetch next state in the state table, it needs current state and input symbol. In this paper, all the symbols are ASCII codes, and there are 256 different symbols, so the bit size \( u \) of a symbol is 8. The bit size of state is determined by the number of states in this constructed automaton, and it is generally 16 or 32. Take Snort2.4 pattern for example, there are 21584 states in the automaton, hence the bit size of \( v \) is 16.

Therefore, we can find that in an automata-based algorithm, the key point is the lookup of next state (Stage-1, the State Search). This paper focuses on this stage and tends to design cost effective tiny data structure. The state table in Stage-1 can be represented as the state transition matrix shown in Fig. 1. The variables \( u \) and \( v \) represent the bit sizes of the state and the symbol, respectively, and the rows and columns of the matrix represent the states in the machine and the symbols that are used by the strings and patterns, respectively. The element \( e_{(x,y)} \) represents the appropriate next state when the current state \( y \) receives the input symbol \( x \), and all the state transitions in the FSM can be stored in the matrix. In the remaining sections of this paper, the FSM will be represented in the form of matrix \( M \).

B. Data Structure Construction

The data structures for our proposed algorithm (AC with Magic State Algorithm, denoted as ACMS) and the way to construct them are described as follows. The original AC algorithm constructs an NFA of given patterns, and then transforms the NFA into a DFA Transition Matrix \( M \). A threshold state value is set to divide \( M \) into two parts that are respectively maintained by two data structures. Different from [13] which rearranges the whole NA, if threshold state value is \( t \), ACMS just rearranges state 0 to state \( t \) of a DFA. We call this process as \texttt{renumbering}, and the transformation process is presented as the Procedure \( R(t) \) shown in Fig. 2. Note that every time it renumbers a new state, it must trace the whole matrix \( M \), and exchange new state and old state number.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Input symbol & 0x47 & 0x48 & 0x49 & 0x4A & 0x4B & 0x4C & 0x4D \\
\hline
0 & 269 & 139 & 161 & 802 & 1059 & 527 & 18 \\
1 & 269 & 16888 & 151 & 802 & 1059 & 527 & 18 \\
2 & 56 & 269 & 139 & 15278 & 802 & 1059 & 527 & 3996 \\
3 & 57 & 14193 & 554 & 606 & 802 & 1059 & 527 & 20998 \\
\hline
\end{tabular}
\caption{Input symbol}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{Automaton Matrix \( M \)}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Input symbol & 0x47 & 0x48 & 0x49 & 0x4A & 0x4B & 0x4C & 0x4D \\
\hline
0 & 55 & 56 & 57 & 58 & 59 & 60 & 61 \\
1 & 128 & 56 & 57 & 129 & 59 & 60 & 130 \\
2 & 56 & 55 & 56 & 57 & 58 & 59 & 60 & 61 \\
3 & 57 & 55 & 475 & 57 & 58 & 59 & 60 & 476 \\
\hline
\end{tabular}
\caption{Input symbol}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{The differences of Partial DFA table of Snort 2.4 before and after renumbering (\( t = 61 \))}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Phase & 0x47 & 0x48 & 0x49 & 0x4A & 0x4B & 0x4C & 0x4D \\
\hline
\hline
Input symbol & 0x47 & 0x48 & 0x49 & 0x4A & 0x4B & 0x4C & 0x4D \\
\hline
0 & 475 & 55 & 56 & 57 & 58 & 59 & 60 & 61 \\
1 & 476 & 1317 & 56 & 499 & 58 & 500 & 60 & 61 \\
2 & 499 & 55 & 475 & 57 & 58 & 59 & 60 & 1598 \\
3 & 500 & 55 & 56 & 484 & 58 & 59 & 60 & 61 \\
\hline
\end{tabular}
\caption{Input symbol}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{The differences of Partial DFA table of Snort 2.4 before and after renumbering (\( t = 61 \))}
\end{figure}
Additionally, if \( t \) equals 0, that means there is no need to perform any renumbering; if \( t \) equals 2\(^{t-1}\), that means it renumbers whole matrix \( M \). Next, the structure of state 0 to state \( t \) after renumbering is maintained in DFA state matrix.

For illustration, consider the example shown in Fig. 3, where Fig. 3(a) is the table format for a state transition matrix of an automaton. When \( t = 61 \), the automaton will be rearranged, and the structure of state 0 to state 61 will be maintained by a table as shown in Fig. 3(b). Let \( M' \) denote the renumbered automaton transition matrix. Then the renumbering process can be represented as \( M \xrightarrow{\text{R0}} M' \).

And we define \( M' \) as:

\[
M' = \begin{bmatrix}
C[0], C[1], \ldots, C[2^{t} - 1]
\end{bmatrix}
\]

and \( C[x] = \begin{bmatrix}
e^{(x,0)}, e^{(x,1)}, \ldots, e^{(x,2^{t}-2)}, e^{(x,2^{t}-1)}
\end{bmatrix}\).

Let \( M' \) be processed by a Procedure \( Q(t) \) of two parts, one is to find the magic states and another is to partition \( M' \). For the first part, it is interesting to observe that in a DFA state transition matrix \( M \), while receiving the same input symbol, most of the states have the same next state. We call this next state “magic state”. For each symbol \( x \), the “magic state” of \( x \) is defined as:

\[
ms(x) = \max \{\#e^{i,j} \mid e^{i,j} = e^{i',j}, \forall i \neq j\}
\]

In other words, \( ms(x) \) is the MS that appears most times in \( C[x] \). Then a Magic State Matrix MS can be constructed as \( MS = \{ms(0), ms(1), \ldots, ms(2^{t} - 1)\} \). Moreover, in the DFA matrix \( M \) constructed by AC algorithm after renumbering, all the next states of initial state will be magic states. For the second part, the \( M' \) is partitioned into two sub-matrices \( M'_{\text{DFA}} \) and \( M'_{\text{IDFA}} \) by threshold state \( t \). \( M'_{\text{DFA}} \) is the part that state values are smaller than \( t \), and \( M'_{\text{IDFA}} \) is another part that state values are equal to or greater than \( t \). Matrix \( M' \) can be denoted as:

\[
M' = \begin{bmatrix}
e^{(0,0)}, \ldots, e^{(0,2^{t}-1)} \\
\vdots & \ddots & \vdots \\
e^{(2^{t}-1,0)}, \ldots, e^{(2^{t}-1,2^{t}-1)}
\end{bmatrix}
\]

\( M' \) is compacted via the following method. Let \( M' \) be processed by a Procedure \( T(MS) \) (shown in Fig. 4) to generate Bitmap Matrix \( B \) and State List Matrix \( S \). The size of \( B \) is the same as \( M' \). As shown in Fig. 5, \( B \equiv \{b(x, y), 0 \leq x \leq 2^{t}-1, 0 \leq y \leq 2^{t}-1\} \). Define \( S = [s[t], s[t+1], \ldots, s[2^{t}-1]] \), where \( s[i] \) is a state list and every state in \( M'_{\text{IDFA}} \) has one \( s \) individually. As Fig. 4 describes, it identifies all elements in \( M' \). If \( e^{(i,j)} \) is not equal to \( ms(x) \), set the bit corresponding to \( e^{(i,j)} \) as 1 in the bitmap and insert the next state into state \( y \)’s State List \( s[y] \) in order. If it equals to \( ms(x) \), set the bit as 0.

**Procedure T(MS)**

```plaintext
1: for each state in \( M' \) do
2:    for each symbol \( x \) do
3:        if \( e^{(i,j)} = ms(x) \) then
4:            set bit(bitmap, state, symbol, 1);
5:        end if
6:    end for
7: end for
```

**Figure 4.** Pseudo code of Procedure \( T(MS) \)

C. Magic State-based Search Algorithm

Let \( T = \{t_{1}, t_{2}, \ldots, t_{n}\} \) be the input string where \( t_{i} \) denotes the \( i \)-th input symbol of \( T \). The magic state based search algorithm first determines if the current state is smaller than the threshold \( t \). If it is, then get next state directly from matrix \( M'_{\text{DFA}} \) through the index composed of current state and input symbol \( t_{i} \). Otherwise, refer to the bit \( B(x, \text{current state}) \) which is corresponding to symbol \( t_{i} \), in Bitmap \( B(x, \text{current state}) \), \( 0 \leq x \leq 2^{t}-1 \), and then identify if it is a magic state. If \( B(x, \text{current state}) = 0 \), fetch next state directly from matrix \( MS \), which is a magic state \( ms(t) \); if \( B(x, \text{current state}) = 1 \), perform popcount in \( B(x, \text{current state}) \) to calculate how many “1” appeared before \( B(x, \text{current state}) \). Let offset denote this value and offset is the position of the element in State List \( s[\text{current state}] \), whose value is the next state value.

D. Example

For illustration, we use examples to demonstrate how the proposed algorithm works. Consider the case shown in Fig. 3(b) again. The magic states for input symbols from 0x47 to 0x4D are 55, 56, 57, 58, 59, 60 and 61 respectively. Assume input \( T = \{\text{HIMKU}\} \) and one of the pre-defined patterns is \{IMK\}. Assume the initial state is 0 and the threshold state \( t = 61 \). First of all, get \( t_{1} = \text{“H”} \) and because state 0 is smaller than threshold 61, query the DFA state matrix \( (M'_{\text{DFA}}) \) directly to get next state 56. Next, \( t_{2} = \text{“I”} \), and since state 56 \( \leq 61 \), query the matrix \( M'_{\text{DFA}} \) again to get next state 57. Then \( t_{3} = \text{“M”} \), do the same process to get the next state 475. Now, \( t_{4} = \text{“K”} \), because state 475 \( > 61 \), query the matrix \( B \). Since \( b(475, 475) = 1 \), it is a non-magic state, do popcount to compute offset that is how many “1” appeared before 475th bit. We can get offset = \( \text{sum of} B(x, 475) \). Then get the node at the offset position in State List \( s[475] \) to know that the next state is 500. Through Stage-2, state 500 is identified as an acceptance state, and the matched pattern is \{IMK\}. Finally, \( t_{5} = \text{“J”} \), due to state 500 \( > 61 \), check the matrix \( B \) and find \( b(74, 500) = 0 \) so it is a magic state. Query the magic state table and get next state 58.

IV. Evaluation

To evaluate the performance of the proposed ACMS under different conditions, the following parameters are defined first.

\( L \): the total latency time required for memory accessing for an input string \( T \) of \( k \) symbols.

\( n \): number of accesses to matrix \( M'_{\text{DFA}} \).

\[
\begin{bmatrix}
b_{0,0} & b_{0,1} & \ldots & b_{0,2^{t}-2} & b_{0,2^{t}-1} \\
b_{1,0} & b_{1,1} & \ldots & b_{1,2^{t}-2} & b_{1,2^{t}-1} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
b_{2^{t}-1,0} & b_{2^{t}-1,1} & \ldots & b_{2^{t}-1,2^{t}-2} & b_{2^{t}-1,2^{t}-1}
\end{bmatrix}
\]

**Figure 5.** Bitmap matrix \( B \) corresponding to the matrix \( M' \)
\[ m \text{ : number of accesses to matrix } MS. \]

\[ m' \text{ : number of accesses to matrix } S. \]

\[ l_D \text{ : latency of each access of matrix } M'_{DFA}. \]

\[ l_{MS} \text{ : latency of each access of } MS \]

\[ l_B \text{ : latency of each access of } B \]

\[ l_S \text{ : latency of each access of } S. \]

\section{Memory Access Analysis}

Generally, the number of memory accesses is the key factor to evaluate the performance of string matching algorithms. We analyze this factor by considering software implementation and hardware implementation.

\subsection{Software implementation}

For software based implementation, algorithms are usually executed sequentially by the processor. Therefore, for the proposed ACMF algorithm, the total latency time required for memory accessing for an input string \( T \) of \( k \) symbols can be calculated as follows:

\[ L = l_D \cdot n + l_B (k-n) + l_S \cdot m + l_{MS} \cdot m, \text{ where} \]

\[ l_D \cdot n : \text{ latency of querying the original DFA State table} \]

\[ l_B (k-n) : \text{ latency of querying Bitmap Matrix } B \]

\[ l_S \cdot m : \text{ latency of searching in matrix } S \]

\[ l_{MS} \cdot m : \text{ latency of querying matrix } MS \]

This calculation only considers the memory access time without considering the instruction execution time of the accessed data. However, some of the instruction execution time is not negligible, especially for \textit{popcount} function. Thus, if most of the current state values during state transitions are within the threshold state, we can have a significantly improved performance. Furthermore, in case most of the state transitions are beyond \( t \), we can still have good performance if most of them are magic states. Otherwise, we need to perform the \textit{popcount} function which takes longer time and therefore the performance is impacted.

\subsection{Hardware implementation}

Considering the trade-off between performance and cost, here two possible hardware implementations are analyzed: off-chip memory and on-chip memory.

\textbf{Off-Chip memory:} with hardware implementation, the accesses of Magic State Matrix \( MS \) and Bitmap Matrix \( B \) can be processed in the same time. Hence, as soon as next state is found as a magic state, these two matrices can be fetched simultaneously and the latency can be represented as:

\[ \text{Max}\{l_{MS} \cdot m, l_B \cdot m\} \text{--------------------- (1)} \]

If it is a non-magic state, the two matrices should be accessed one by one and the latency is \((l_B + l_S) \cdot m\).

Consequently, the latency for off-chip memory implementation can be represented as:

\[ L = l_D \cdot n + \text{Max}\{l_{MS} \cdot m, l_B \cdot m\} + (l_B + l_S) \cdot m. \]

\textbf{On-Chip memory:} assume the on-chip memory is large enough to store all the data structures used by the ACMF algorithm. Then \( l_B \) will be equal to \( l_S \), and \( l_{MS} \cdot m = l_B \cdot m \). Refer to Equation (1), \text{Max}\{l_{MS} \cdot m, l_B \cdot m\} can be briefly represented as \( l_B \cdot m \).

Therefore, the latency for on-chip memory implementation can be represented as:

\[ L = l_D \cdot n + l_B \cdot m + (l_B + l_S) \cdot m \text{--------------------- (2)} \]

Although it takes a longer time to perform the \textit{popcount} operation in software implementation, it can be performed rapidly by silicon circuit of hardware implementation. Thus, the performance of hardware implementation is dominated by the latency of accessing data structures. The hardware implementation also has the best performance when most of the current state values while state transitions are within the threshold state. In case the on-chip memory is too small to store all the four matrices in ACMF, we can then arrange some of the matrices to be stored in off-chip memory to achieve as best performance as possible.

\section{Memory versus Throughput}

Usually to have better throughput, we need larger memory. In order to balance the time and the space (memory, especially the on-chip memory) of the proposed ACMF, we need to figure out the relation between the required memory space and its performance. A PC-based Windows XP system is used to evaluate this. The Snort 2.4 signatures with total 2389 patterns are employed and the Defcon9 traces [14] are adopted as the input strings. The evaluation results are demonstrated in Fig. 6: where we can see that the ACMF furnishes good scalability. The strategy of memory configuration can be adaptive for target speed demands. As expected, larger memory stands for higher throughput. In addition, a larger threshold state value \( t \) produces a larger DFA State Matrix and smaller matrices \( B \) and \( S \). For \( t > 2048 \), the DFA State Matrix is the major memory contribution and therefore the total memory increases for larger value of \( t \). For throughput, a larger threshold state value \( t \) offers a better throughput due to more state data can be stored in matrix \( M'_{DFA} \), and \( l_B \) is the least latency compared with other parts. Also the renumbering process not only diminishes nodes stored in matrix \( S \) and memory usage, but also increases the frequency that state transitions to magic state. This improves the performance too.

\section{Implementation and Simulation}

\subsection{Implementation Issues}

The ACMF algorithm is implemented by rewriting the AC implementation of Snort. The testing computer runs Windows XP and is a Pentium-4 3.0GHz with 1MB L2 Cache and 768MB DDR SDRAM. Again, the Snort 2.4 signatures with
total 2389 patterns are employed and the Defcon9 traces [14] are adopted as the input strings. Three different implementations for the popcount function, called pop-1, pop-2, and pop-3, are designed as follows:

- **pop-1**: shift all bits one by one to identify if they are 1.
- **pop-2**: a popcount algorithm proposed in [17] which is suitable for software implementation.
- **pop-3**: A popcount algorithm developed by us which can count the number of “1s” in 8-bit each time by looking up a pre-defined table of size 256. The value of the 8-bit is used as an index to access the table and the output is the number of “1s” in the 8-bit.

For comparison purpose, five more string matching algorithms are also implemented: AC-DFA [3], AC-NFA [3], AC-Sparse [12], AC-Rearranging [13], AC-Bitmap [5]. Following are descriptions of these algorithms:

- **AC-DFA**: DFA structure and a full matrix [3, 12].
- **AC-NFA**: NFA structure [3, 12].
- **AC-Sparse**: DFA structure with “sparse” data structure [12].
- **AC-Rearranging**: From AC-DFA, renumbering whole full matrix [13].
- **AC-Bitmap**: Compressed NFA structure with pop-2 and pop-3 popcount function [5].

**B. Simulation Analysis**

Fig. 7 shows the memory required by the six different software implementations. For the ACMS implementations, the threshold state value \( t \) is set to 128.

We can see that the memory size used by each of AC-DFA, AC-NFA and AC-Rearranging is about 18MB (If the memory consumption analysis in [5, 12] is applied, the result is 50–60MB). The AC-BITMAP needs least memory of 0.9MB. The proposed ACMS implementations also require a tiny memory of 1.728MB. Both these two tiny data structures can be stored into the current available commercial on-chip memory (today’s FPGA provides up to 2MB on-chip memory). However, the proposed ACMS compacts a DFA but the AC-BITMAP only compresses an NFA.

The measured throughput for different software implementations are depicted in Fig. 8. The AC-Rearranging and AC-DFA produce best throughput. But due to the required large memory space, they are not suitable in most resource-limited network appliances. For AC-SPARSE, although it also compresses memory, but the compressed data structure is still larger than the baseline of 2MB, which is the reasonable limit of embedded memory in common VLSI techniques nowadays.

**Figure 7. Comparison of memory requirement for different algorithms**

**Figure 8. Comparison of throughput for different algorithms (software implementation)**

As a result, the AC-SPARSE encounters the condition of low performance. The AC-BITMAP provides the worst throughput of 5.737Mbps due to the expensive popcount operations even least memory is required. The ACMS performs around 21 times (125.088/5.737) faster than AC-BITMAP. It also performs faster than AC-NFA whose data structure is not compressed, and also close to that of AC-DFA and AC-Rearranging. Note that the ACMS performance is affected by different popcount implementations. This is because the performance is impacted not only by the number of memory accesses but also the computing time of related instructions. This also indicates that we can have better performance if the popcount function can be implemented by more efficient instructions, such as MMX and SSE instructions.

Based on the measured data shown in Fig. 7 and Fig. 8, we realized that there is a trade-off between the required memory size and the obtained throughput. To have more practical and realistic comparison, we define the efficiency \( E = E_T / E_M \) of a particular algorithm; where \( E_T \) represents the throughput (Mbps), and \( E_M \) represents the memory usage (MB). The higher \( E \) value represents the higher balance between performance and memory usage. Table I shows the Efficiency of different algorithms for software implementation. For example, for AC-SPARSE, \( E_T = 112.982 \), and \( E_M = 11.67 \), so its efficiency is 9.68. The efficiency \( E \) of ACMS(pop-3) is 70.20 (125/1.782) which is seven times larger than that of AC-SPARSE. The efficiency of ACMS algorithms with pop-1 and pop-2 popcount function are 8.55 and 34.21, respectively. Although AC-BITMAP(pop-3) diminishes 95% memory usage, it faces a dilemma of low throughput. Its efficiency \( E \) is only 6.33 in software implementation. It is also interesting to see that by using the same pop-3 algorithm, the efficiency of ACMS is 11 times (70.20/6.33) larger than that of AC-BITMAP.

**Table I. Efficiency of different algorithms**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Efficiency (E)</th>
<th>Algorithm</th>
<th>Efficiency (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC-DFA</td>
<td>8.07</td>
<td>ACMS(pop-1)</td>
<td>8.35</td>
</tr>
<tr>
<td>AC-NFA</td>
<td>2.48</td>
<td>ACMS(pop-2)</td>
<td>34.21</td>
</tr>
<tr>
<td>AC-SPARSE</td>
<td>9.68</td>
<td>ACMS(pop-3)</td>
<td>70.20</td>
</tr>
<tr>
<td>AC-BITMAP(pop-3)</td>
<td>6.33</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
hardware model with 2MB embedded SRAM and 200MHz clock frequency. The width of embedded memory is 128-Byte (1024-bit) that is the same with that used in [5]. As the memory usage of AC-DFA and AC-NFA both exceeded 2MB in SRAM, an additional memory module of DDR2-400 SDRAM with 200MHz clock frequency [15] is added. Fig. 9 depicts the hardware throughputs, measured by the number of memory accesses, from the viewpoint of electronic system level (ESL[16]), for different implementations. First of all, the performance of AC-DFA and AC-NFA does not increase a lot compared to their software implementations. This is because their data structures are too large to be stored into SRAM, and most of them are kept in slower SDRAM. Due to the data structure in AC-NFA contains extra fail state transitions compared that of AC-DFA, its throughput is clearly lower than that of AC-DFA. However, for AC-BITMAP, since its NFA data structures are compressed significantly and could be stored into the on-chip SRAM, it provides a much higher throughput (around 432Mbps) than that of AC-DFA. Based on previous analysis, the popcount function can be completed in very short time by hardware circuit. Therefore the performance of an algorithm is determined and dominated by the number of memory accesses and access latency. The proposed ACMS needs less number of memory accesses and most of these accesses can be completed within the fast on-chip memory. This also explains why the ACMS offers best throughput (around 432Mbps) which is around 10% (1456/1456) to 10 times faster than other algorithms. In summary, the ACMS is an efficient string matching algorithm that successfully balances memory and speed.

VI. CONCLUSION

This article introduces a novel string matching algorithm, named AC with Magic State Algorithm (ACMS), that using only a tiny memory and provides high throughput for either hardware or software implementations. ACMS is designed based on the observation of magic states in the deterministic finite state automata. Two features, renumbering and bitmap are employed to make a balance between the required memory and targeted speed.

ACMS is a practical string matching algorithm and especially suitable for identifying network applications. Evaluation and experimental results show that the overall efficiency gain of ACMS is over at least 11 times better than that of the other researches. Through compacting mechanism, we show that the entire data structure constructed from the Snort2.4 (2389 patterns with 35K characters) is less than 2MB, which is less than 10% of the original memory size. Thus, the entire automaton not only can be stored into the on-chip memory for fast accessing, but also suitable to embedded systems with restricted resources. Compared to past NFA-based string matching studies with similar memory requirement, the proposed ACMS algorithm achieves more than 3.5 times throughput in hardware performance. As for software implementation, the performance of the ACMS is over at least 21 times better than that of state-of-the-art researches.

REFERENCES