Optimization of Pattern Matching Algorithm for Memory Based Architecture

Cheng-Hung Lin
National Tsing Hua University,
Taiwan, R.O.C
d918326@oz.nthu.edu.tw

Yu-Tang Tai
National Tsing Hua University,
Taiwan, R.O.C
g944377@oz.nthu.edu.tw

Shih-Chieh Chang
National Tsing Hua University,
Taiwan, R.O.C
scchang@cs.nthu.edu.tw

ABSTRACT

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 total Snort string patterns, the new algorithm achieves 29% of memory reduction compared with the traditional Aho-Corasick algorithm [5]. Moreover, since our approach is orthogonal to other memory reduction approaches, we can obtain substantial gain even after applying the existing state-of-the-art algorithms. For example, after applying the bit-split algorithm [9], we can still gain an additional 22% of memory reduction.

Categories and Subject Descriptors

C.2.0 **[Computer Communication Networks]**: General-Security and protection (e.g., firewalls)

General Terms

Algorithms, Design, Security

Keywords

Pattern matching, intrusion detection, DFA

1. INTRODUCTION

The purpose of a 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.

ANCS'07, December 3–4, 2007, Orlando, Florida, USA.

The primary function of an intrusion detection system is to perform matching of attack string patterns. However, string matching using the software-only approach can no longer meet the high throughput of today's networking. To speed up string matching, many researchers have proposed hardware improvements which can be classified into two main approaches, the logic [1][2][3][4] and the memory architectures [6][7][8][9][10].

In terms of re-configurability and scalability, the memory architecture has attracted a lot of attention because it allows on-the-fly pattern update on memory without re-synthesis and re-layout. The basic memory architecture works as follows. First, the (attack) string patterns are compiled to a *finite state machine* (FSM) whose output is asserted when any substring of input strings matches the string patterns. Then, the corresponding state table of the FSM is stored in memory. For instance, Figure 1 shows the state transition graph of the FSM to match two string patterns "bcdf" and "pcdg", where all transitions to state 0 are omitted. States 4 and 8 are the final states indicating the matching of string patterns "bcdf" and "pcdg", respectively. Figure 2 presents a simple memory architecture to implement the FSM. In the architecture, the memory address register consists of the current state and input character; the decoder converts the memory address to the corresponding memory location, which stores the next state and the *match vector* information. If the match vector is "0", it is not a final state; otherwise, the match vector indicates the matched pattern. For example, suppose the current state is in state 7 and the input character is *g*. 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.

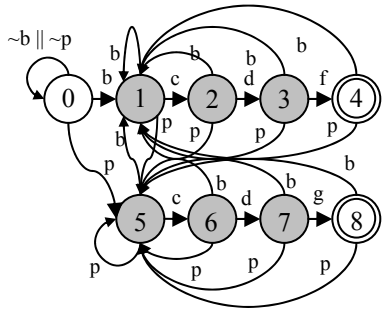


Figure 1: DFA for matching "bcdf" and "pcdg"

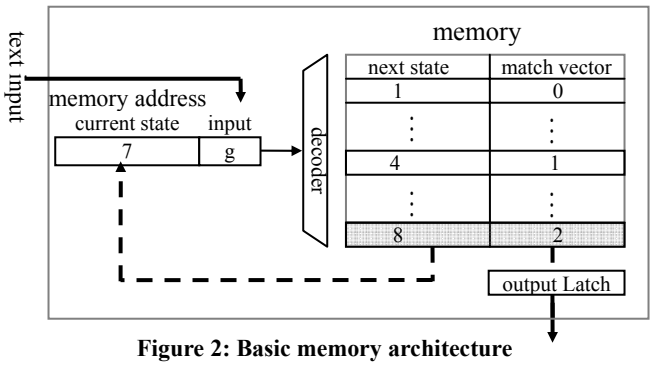


Figure 2: Basic memory architecture

Due to the increasing number of attacks, the memory required for implementing the corresponding FSM increases tremendously. Because the performance, cost, and power consumption of the memory architecture is directly related to the memory size, reducing the memory size has become imperative.

We observe that many string patterns are similar because of common sub-strings. However, when string patterns are compiled into an FSM, the similarity does not lead to a small FSM. Consider the same example in Figure 1 where two string patterns have the same sub-string "cd". Because of the common sub-string, state 2 (state 3) has "similar" state transitions to those of state 6 (state 7). Still, state 2 (state 3) and state 6 (state 7) are not equivalent states and cannot be merged directly. We call a state machine merging those non-equivalent "similar" states, *merg_FSM*.

In this paper, we propose a state-traversal mechanism on a merge_FSM while achieving the same purposes of pattern matching. Since the number of states in merg_FSM can be significantly smaller than the original FSM, it results in a much smaller memory size. We also show that hardware needed to support the state-traversal mechanism is limited. Experimental results show that our algorithm achieves 29% of memory reduction compared with the traditional AC algorithm for total Snort string patterns. In addition, since our approach is orthogonal to other memory reduction approaches, we can obtain substantial gain even after applying the existing state-of-the-art algorithms. For example, after applying the bit-split algorithm [9], we can still gain an additional 22% of memory reduction.

2. REVIEW OF THE AHO-CORASICK ALGORITHM

In this section, we review the Aho-Corasick (AC) algorithm [5]. Among all memory architectures, the AC algorithm has been widely adopted for string matching in [6][7][8][9][10] because the algorithm can effectively reduce the number of state transitions and therefore the memory size. Using the same example as in Figure 1, Figure 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 from [5].

The failure transition is explained as follows. Given a current state and an input character, the AC machine checks to see whether the input character causes a valid transition; 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*. As shown in Figure 4, the AC state table shows that there is no valid transition from state 1 with the input character *p*. Therefore, the AC machine takes a failure transition to state 0. Then in the next cycle, the AC machine re-considers the input character *p* in state 0 and finds a valid transition to state 5.

Besides, the double-circled nodes indicate the final states of patterns. In Figure 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 $\{P_2P_1\} = \{00\}$.

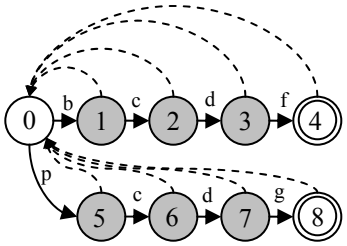


Figure 3: State diagram of an Aho-Corasick machine

	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

Figure 4: Aho-Corasick state table

3. BASIC IDEA

Due to the common sub-strings of string patterns, the compiled AC machine has states with similar state transitions. Despite the similarity, those similar states are not equivalent 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.

Note that two states are *equivalent* if and only if their next states are equivalent. In Figure 3, state 3 and state 7 are similar but not equivalent states because for the same input *f*, 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. We have the following definitions.

Definition: Two states are defined as pseudo-equivalent states if they have identical inputs, failure transitions, and outputs.

In Figure 3, state 2 and state 6 are pseudo-equivalent states because they have identical input *c*, identical failure transition to state 0 and identical output 00. Also, state 3 and state 7 are

pseudo-equivalent states. Note that merging pseudo-equivalent states results in a functional error FSM. For the same example, Figure 5 shows an FSM that merges the pseudo-equivalent states 2 and 6 to become state 26, and merges the pseudo-equivalent states 3 and 7 to become state 37. Again, we refer to the FSM that merges the pseudo-equivalent states as the *merg_FSM*. Given an input string "pcdf", the merg_FSM reaches the erroneous state 4 which indicates the pattern "bcdf" is matched while the original AC state machine (in Figure 3) goes back to state 0. This shows the merg_FSM may causes false positive results.

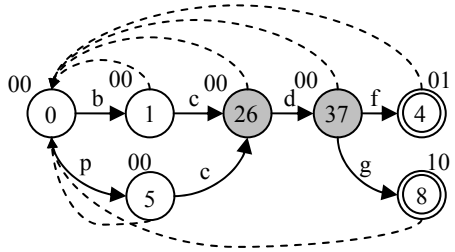


Figure 5: Merging non-equivalent states

The merg_FSM is a different machine from the original FSM but with a smaller number of states and state transitions. A direct implementation of merg_FSM has a smaller memory than the original FSM in the memory architecture. Our objective is to modify the algorithm so that we store only the merg_FSM table in memory while the overall system still functions in the same way as the original FSM did. The overall architecture of our state traversal machine is shown in Figure 6 where the state traversal mechanism guides the state machine to traverse on the merg_FSM and provides correct results as the traditional AC state machine. In section 4, we first discuss the state traversal mechanism. Then, in section 5, we discuss how the state traversal machine is created in our algorithm.

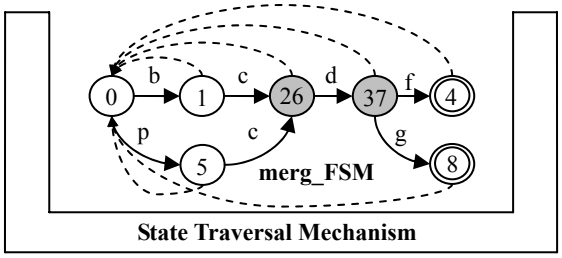


Figure 6: The architecture of the state traversal machine

4. 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). To have a correct result, when state 26 (state 37) is reached, we need a mechanism to understand in the original FSM whether it is state 2 or state 6 (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 is state 1 when reaching state 26, we know that in the original FSM, it is state 2. On the other hand, if the precedent state is state 5, the original is state 6. This example shows that if we can memorize the precedent path entering into the merged states, we can differentiate all merged states. In the following, we discuss how the precedent path vector can be retained during the state traversal in the merg_FSM.

In a traditional AC state machine, a final state stores the corresponding match vector which is one-hot encoded. For example in Figure 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 $\{P_2P_1\} = \{00\}$. One-hot encoding for a match vector is necessary because a final state may represent more than one matched string pattern [5]. Therefore, the width of the match vector is equal to the number of string patterns. As shown in Figure 4, the majority of memories in the column "match vector" store the zero vectors {00} simply to express that those states are not final states.

In our design, we re-use those memory spaces storing zero vectors {00} and match vectors 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 Figure 7. The path 0->1->26->37->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 26, state 37, and state 4}, is set to 1. Similarly, the path 0->5->26->37->4 matches the second string pattern "pcdg". Therefore, the second bit of the pathVec of the states on the path, {state 0, state 5, state 26, state 37, and state 8}, is set to 1. Finally, the pathVec of all states are shown in Figure 7. In addition, an additional bit, called *ifFinal*, is added to each state to indicate whether the state is a final state. As shown in Figure 7, each state stores the pathVec and ifFinal as the form of "*pathVec_ ifFinal"*.

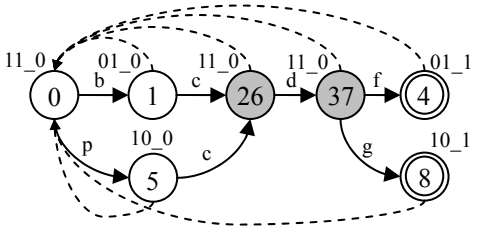


Figure 7: New State diagram of merg_FSM

In addition, we need a register, called *preReg*, to trace the precedent pathVec in each state. The width of preReg is equal to the width of pathVec. Each bit of the preReg also corresponds to a string pattern. The preReg is updated in each state by performing a bitwise AND operation on the pathVec of the next state and its current value. By tracing the precedent path entering into the merged state, we can differentiate all merged states. When the final state is reached, the value of the preReg indicates the match vector of the matched pattern. During the state traversal, if all the bits of the preReg become 0, the machine will go to the failure mode and choose the failure transition as in the AC algorithm. After any failure transition, all the bits of the preReg are reset to 1.

Consider an example in Figure 8 where the string "pcdf" is applied. Initially, in state 0, the preReg is initiated to $\{P_2P_1\} = \{11\}$. After taking the input character *p*, the merg_FSM goes to state 5 and updates the preReg by performing a bitwise AND operation on the pathVec {10} of state 5 and the current preReg {11}. The resulting new value of the preReg will be $\{P_2P_1\} = \{10 \text{ AND } 11\} = \{10\}$. Then, after taking the input character *c*, the merg_FSM goes to state 26 and updates the preReg by performing a bitwise AND operation on the pathVec {11} of state 26 and the current preReg {10}. The preReg remains $\{P_2P_1\} = \{11 \text{ AND } 10\} = \{10\}$. Further, after taking the input character *d*, the merg_FSM goes to state 37 and updates the preReg by performing a bitwise AND operation on the pathVec {11} of state 37 and the current preReg {10}. Still, the preReg remains $\{P_2P_1\} = \{11 \text{ AND } 10\} = \{10\}$. Finally, after taking the input character *f*, the merg_FSM goes to state 4. After performing a bitwise AND operation on the pathVec {01} of state 4 and the current preReg {10}, the preReg becomes $\{P_2P_1\} = \{01 \text{ AND } 10\} = \{00\}$. According to our algorithm, during the state traversal, if all the bits of the preReg become 0, the machine will go to the failure mode and choose the failure transition as in the AC algorithm. Therefore, the machine takes the failure transition to state 0 instead of state 4. We would like to point out that the same string applied to the merg_FSM, using the traditional state traversal algorithm in Figure 5, leads to an erroneous result.

state	0	5	26	37	**0**
input char		p	c	d	f	f
pathVec	11	10	11	11	01
preReg	11	10	10	10	**00**
ifFinal	0	0	0	0	1

Figure 8: State transitions of the input string "pcdf"

The algorithm of our state traversal pattern-matching machine is shown in Figure 9.

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:

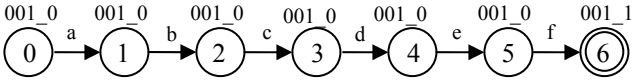
```
begin
    state ← 0
    preReg ← 1….1        //all bits are initiated to 1.
    for i ← until n do
      begin
        preReg = preReg & pathVec(state)
        while g(state, a_i) == fail || preReg == 0 do
                begin
                    state ← f (state)
                    preReg ← 1….1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

Figure 9: State traversal pattern matching algorithm

5. CONSTRUCTION OF THE STATE TRAVERSAL MACHINE

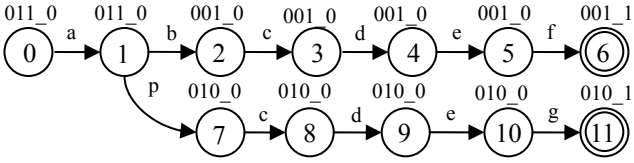
The construction of a state traversal machine consists of (1) the construction of valid transition, failure transition, pathVec, and ifFinal functions and (2) merging pseudo-equivalent states. In the first step, the states and valid transitions are created first. And then, the failure transitions are created. The construction of pathVec and ifFinal begins in the first step and completes in the second step.

For a set of string patterns, a graph is created for the valid transition function. The creation of the graph starts at an initial state 0. Then, each string pattern is inserted into the graph by adding a directed path from initial state 0 to a final state where the path terminates. Therefore, there is a path, from initial state 0 to a final state, which matches the corresponding string pattern. For example, consider the three patterns, "abcdef", "apcdeg", and "awcdeh". Adding the first pattern "abcdef" to the graph, we obtain:



The path from state 0 to state 6 matches the first pattern "abcdef". Therefore, the pathVec of all states on the path is set to $\{P_3P_2P_1\} = \{001\}$, and the ifFinal of state 6 is set to 1 to notify the final state where the path terminates.

Adding the second pattern "apcdeg" into the graph, we obtain:



Note that when the pattern "apcdeg" is added to the graph, because there is already an edge labeled *a* from state 0 to state 1, the edge is reused. Therefore, the pathVec of states 0 and 1 is set to $\{P_3P_2P_1\} = \{011\}$ and the pathVec of other states, {state 7, state 8, state 9, state 10, state 11}on the path is set to $\{P_3P_2P_1\} = \{010\}$. Besides, the ifFinal of state 11 is set to 1 to indicate the final state for the second pattern. Similarly, when the third pattern "awcdeh" is added to the graph, the edge labeled *a* from state 0 to state 1 is also reused. Therefore, the pathVec of states 0 and 1 is set to $\{P_3P_2P_1\} = \{111\}$. The pathVec of other states {state 12, state 13, state 14, state 15, and state 16} on the path is set to $\{P_3P_2P_1\} = \{100\}$. The ifFinal of state 16 is set to 1 to indicate the final state of the third pattern. Finally, Figure 10 shows the directed graph consisting only of valid transitions.

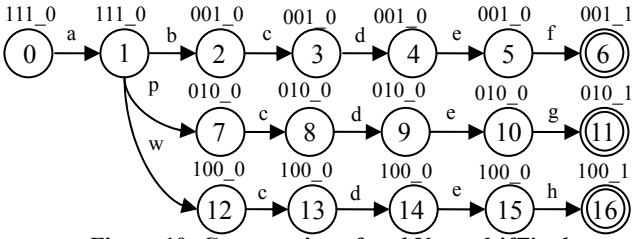


Figure 10: Construction of *pathVec* and *ifFinal*

In the second step, our algorithm extracts and merges the *pseudo-equivalent states*. Note that merging pseudo-equivalent states includes merging the failure transitions and performing the union on the pathVec of the merged states. Consider the same example as in Figure 10. We can find that state 3, state 8, and state 13 are pseudo-equivalent states because they have identical input *c*, identical failure transitions to state 0 and identical ifFinal 0. Similarly, state 4, state 9, and state 14 are pseudo-equivalent states and state 5, state 10, and state 15 are pseudo-equivalent states. As shown in Figure 11, these pseudo-equivalent states are merged into states 3, 4 and 5. The pathVec of state 3 is modified to be $\{P_3P_2P_1\}$ = {001} || {010} || {100} = {111} by performing the union on the pathVec of state 3, state 8, and state 13. Similarly, the pathVec of state 4 and state 5 is also modified to be {111}. Figure 11 shows the final state diagram of our state traversal machine. Compared with the original AC state machine in Figure 10, six states are eliminated.

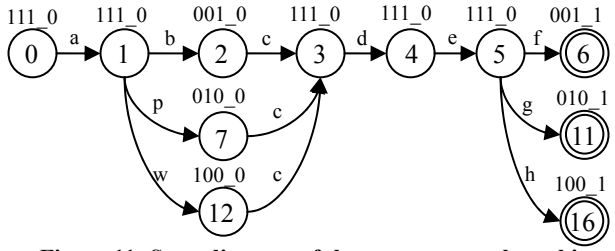


Figure 11: State diagram of the state traversal machine

6. CYCLE PROBLEMS WHEN MERGING MULTIPLE SECTIONS OF PSEUDO-EQUIVALENT STATES

When certain cases of multiple sections of pseudo-equivalent states are merged, it may create cycle problems in a state machine. The cycle problem may cause false positive matching results. Consider the two patterns, "abcdef" and "wdebcg." whose corresponding AC state machine is shown as Figure 12. We can find that states 2 and 10, states 3 and 11 are pseudo-equivalent states while states 4 and 8, states 5 and 9 are also pseudo-equivalent states. Figure 13 shows the state machine merging the two sections of pseudo-equivalent states. The state machine after merging the two disorder sections of pseudo-equivalent states creates a loop transition from state 5 to state 2. The loop transition will cause false positive matching results. For example, the input string "abcdebcdef" will be mistaken as a match of the pattern "abcdef."

To prevent the cycle problem, we only merge pseudo equivalent states when no cycle problem occurs. If there is a cycle, we will skip the merging.

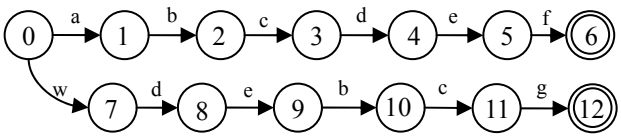


Figure 12: AC state machine for the two patterns, "abcdef" and "wdebcg"

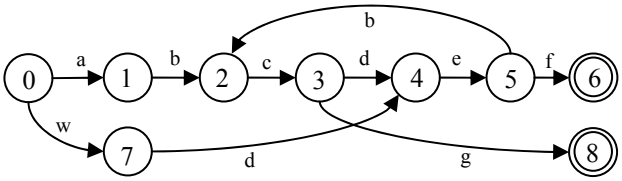


Figure 13: Merging two disorder section of pseudo-equivalent states

7. EXPERIMENTAL RESULTS

We performed experiments on the seven largest rule sets and the total string patterns from the Snort rule sets to compare with the methods from [5] [9].

Table 1 shows the results of our approach compared with [5]. Columns one, two and three show the name of the rule set, the number of patterns, and the number of characters of the rule set. Columns four, five, and six show the number of state transitions, the number of states, and the memory size of [5]. Columns seven, eight, and nine show the results of our approach. Column ten shows the memory reduction compared to [5] and [9]. For example in the first row of the Table 1, the Oracle rule set has 138 patterns with 4,674 characters. Applying the traditional AC algorithm, the total number of states is 2,185 and the memory size is 880,009 bytes. Applying our algorithm, the number of states is reduced to 1,221 and the memory size is reduced to 452,533 bytes, 49% of memory reduction from [5].Consider the total 1,595 string patterns of Snort rule set. As shown in the ninth row of Table 1, our algorithm achieves a 29% memory reduction compared with [5].

We also compared with the bit-split algorithm [9]. The results are shown in Table 2. Consider the same Oracle rule set in the first row of Table 2. Applying the bit-split algorithm which splits the traditional AC state machine into 4 state machines, the total number of states is 6,665 and the size of memory is 633,175 bytes. Applying our algorithm after the bit-split algorithm, the number of states is reduced to 3,603 and the size of memory is reduced to 358,499 bytes. The memory reduction achieves 43%. For the total 1,595 string patterns of the Snort rule set, applying our algorithm after the bit-split algorithm can further achieve additional an 22% of memory reduction.

8. CONCLUSIONS

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 orthogonal 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.

9. REFERENCE

1. R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *Proc. of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001,pp. 227-238.
2. R. Franklin, D. Carver, and B.L. Hutchings. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proceedings of IEEE FCCM 2002*, pp. 111-120, Apr. 2002.
3. J. Moscola, J. Lockwood, R. P. Loui and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proc. of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2003.
4. C. R. Clark and D. E. Schimmel. "Scalable Parallel Pattern Matching on High Speed Networks," in *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM)*, 2004.
5. A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In *Communications of the ACM*, 18(6):333–340, 1975.
6. Young H. Cho and William H. Mangione-Smith, A Pattern Matching Co-processor for Network Security. In *42nd IEEE/ACM Design Automation Conference,* Anaheim, CA, June 13-17, 2005.
7. M. Aldwairi*, T. Conte, and P. Franzon. Configurable String Matching Hardware for Speeding up Intrusion Detection. In *ACM SIGARCH Computer Architecture News*, 33(1):99–107, 2005.
8. S. Dharmapurikar and J. Lockwood. Fast and Scalable Pattern Matching for Content Filtering. In *Proceedings of Symposium on Architectures for Networking and. Communications Systems (ANCS)*, Oct 2005.
9. L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA'05: 32nd Annual International Symposium on Computer Architecture*, pp. 112-122, 2005.
10. H. J. Jung, Z. K. Baker, and V. K. Prasanna. Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems. In *IPDPS 2006: 20th International Parallel and Distributed Processing Symposium*, 2006.
11. M. Roesch. Snort- lightweight Intrusion Detection for networks. In *Proceedings of LISA99, the 15th Systems Administration Conference*, 1999

Table 1: Experimental results of the AC algorithm and our algorithm

Rule Sets	# of patterns	# of char.	Tradition AC [5]			Our algorithm			
			# of transitions	# of states	Memory (bytes)	# of transitions	# of states	Memory (bytes)	Memory reduction
Oracle	138	4,674	2,180	2,185	880,009	1,389	1,221	452,533	49%
Sql	44	1,089	421	422	129,290	321	284	87,011	33%
Backdoor	57	599	563	565	191,253	523	497	152,268	20%
Web-iis	113	2,047	1,533	1,537	569,651	1,273	1,155	428,072	25%
Web-php	115	2,455	1,670	1,675	620,797	1,295	1,142	423,254	32%
Web-misc	310	4,711	3,576	3,587	1,444,664	3,031	2,734	1,101,119	24%
Web-cgi	347	5,339	3,407	3,419	1,377,002	2,672	2,358	949,685	31%
Total rules	**1,595**	**20,921**	**17,472**	**17,522**	**8,745,668**	**14,704**	**13,381**	**6,248,927**	**29%**
Ratio			1	1	1	84%	76%	71%	

Table 2: Experimental results of the bit-split algorithm and our algorithm

Rule Sets	# of patterns	# of char.	Bit-split [9]			Bit-split + Our algorithm			
			# of transitions	# of states	Memory (bytes)	# of transitions	# of states	Memory (bytes)	Memory reduction
Oracle	138	4,674	6,645	6,665	633,175	4,146	3,603	358,499	43%
Sql	44	1,089	1,211	1,215	110,565	866	769	72,671	34%
Backdoor	57	599	1,697	1,705	155,155	1,441	1,305	126,585	18%
Web-iis	113	2,047	4,869	4,885	464,075	3,844	3,374	335,713	28%
Web-php	115	2,455	4,991	5,011	476,045	3,871	3,345	332,828	30%
Web-misc	310	4,711	10,959	11,003	1,067,291	8,861	7,816	797,232	25%
Web-cgi	347	5,339	9,901	9,949	965,053	7,875	6,957	709,614	26%
Total rules	**1,595**	**20,921**	**53,930**	**54,130**	**5,467,130**	**43,550**	**38,701**	**4,237,760**	**22%**
Ratio			1	1	1	81%	71%	78%	