A Memory Efficient Multiple Pattern Matching Architecture for Network Security

Tian Song, Wei Zhang
Dept. of Computer Science and Technology
Tsinghua University
Beijing, P.R.China
{songt02, zhwei02}@mails.tsinghua.edu.cn

Dongsheng Wang, Yibo Xue
Microprocessor and SoC Tech. R&D Center
Tsinghua University
Beijing, P.R.China
{wds, yiboxue}@tsinghua.edu.cn

***Abstract*—Pattern matching is one of the most important components for the content inspection based applications of network security, and it requires well designed algorithms and architectures to keep up with the increasing network speed. For most of the solutions, AC and its derivative algorithms are widely used. They are based on the DFA model but utilize large amount of memory because of so many transition rules. An algorithm, called ACC, is presented in this paper for multiple pattern matching. It uses a novel model, namely cached deterministic finite automate (CDFA). In ACC, by using CDFA, only 4.1% transition rules for ClamAV (20.8% for Snort) are needed to represent the same function using DFA built by AC. This paper also proposes a new scheme named next-state addressing (NSA) to store and access transition rules of DFA in memory. Using this method, transition rules can be efficiently stored and directly accessed. Finally the architecture for multiple pattern matching is optimized by several approaches. Experiments show our architecture can achieve matching speed faster than 10Gbps with very efficient memory utilization, i.e., 81KB memory for 1.8K Snort rules with total 29K characters, and 9.5MB memory for 50K ClamAV rules with total 4.44M characters. A single architecture is memory efficient for large pattern set, and it is possible to support more than 10M patterns with at most half amount of the memory utilization compared to the state-of-the-art architectures.**

Keywords-pattern matching; string matching; virus scanning; intrusion detection

I. INTRODUCTION

Computer network has become an essential part of our daily life. To ensure the safety of network, various network security measures are taken. Being the most widely deployed one, firewall ensures information transfer from trusted sources to destinations by inspecting the packet headers. However, numerous malicious contents, such as intrusions, viruses, spam, spyware, can still outplay firewalls by hiding themselves in the payload of packets. Consequently content inspection based applications emerged, including intrusion detection and prevention systems (IDS/IPS), virus scanners, spam filters, content security management appliance, and et al. Furthermore, unified threat management system (UTM) is introduced to incorporate all the functionalities. For these applications, one of the most challenging tasks is to improve inspecting speed and capacity to catch up with the rapid growth of network speed.

As one of the basic operations, multiple pattern matching is the performance bottleneck of many content inspection or deep packet inspection based applications. Pattern matching architectures for network intrusion detection systems have emerged in the past few years, which can handle thousands of patterns. A clear trend is that pattern sets of applications become larger and larger. For example, tens of thousands of signatures are already common in anti-virus scanners.

In our work, a pattern matching architecture for tens of thousands of signatures is proposed. The first idea is to use an algorithm based on a novel model, namely *cached DFA* (CDFA), to express the pattern set more efficiently. The algorithm is called ACC.

The second idea, *next state addressing* (NSA), is to store transition rules of finite automata using less memory. It is achieved by taking states as addresses and employing feature of the state acting as the next state in DFA or CDFA.

These two ideas both increase the memory efficiency. Moreover, the architecture for multiple pattern matching is given with some optimizations for reducing critical path and the memory utilization.

In conclusion, the contributions of this paper can be summarized as follows:

- We present an algorithm based on CDFA, which extends DFA by associating some memory as cache. Compared to AC, ACC can reduce the number of transition rules in Snort and ClamAV to 20.8% and 4.1% respectively.
- We present a novel scheme named next-state addressing (NSA) to store and access transition rules in memory. Some analyses show that NSA is memory efficient in usual cases.
- We give several optimizations to shorten critical path and increase the memory utilization efficiency. The methods include fine grain multithreading-like pipelining, entry combination, set-associative policy etc.
- Our approach inherits all the merits of DFA, such as deterministic matching performance, dynamic update. It is suitable for implementation on FPGAs and ASICs.

The rest of the paper is organized as follows. In section 2, we go over the related works in pattern matching. In section 3, we analyze DFA model and explain its limitations. Then in section 4, we introduce CDFA, our algorithm ACC, NSA scheme and our architecture in details. Further optimizations

This work is supported by National Natural Science Foundation of China (No.60673145).

and results about critical path and memory usage are given in section 5. Finally conclusions are drawn in section 6.

II. RELATED WORKS

Many pattern or string matching architectures have been proposed in recent years for network security. Most of the researches focus on pattern matching issue for network intrusion detection and prevention system (NIDS/NIPS), in which pattern set consists of about three thousand patterns.

The early researches of pattern matching architecture are based on programmable logics in FPGAs. The key issue is how to efficiently map patterns to the circuits of logic on FPGAs [2,3,4,6,11,21]. Among the related works, I. Sourdis [11] and C.R.Clark [4] present their architectures to give several methods to efficiently map patterns on FPGAs.

At the same time, some RAM based architectures for ASICs were also proposed using Aho-Corasick [19] like algorithms [1,7,9,10,12,14]. For those architectures, pattern sets are stored in RAM other than logics. The main issue is to use less memory to effectively support bigger pattern set with deterministic high frequency. Lin Tan [12] developed an approach to split the state machines into tiny ones for less memory. Jan v. Lunteren [1] gave an architecture based on B-FSM with string set partitioning to achieve the goal. Some other optimization methods were also proposed by using bloom filter [7] and TCAM [5].

Besides NIDS/NIPS, some other network applications also require pattern matching, such as anti-virus scanners and spam filters. These applications may have much bigger pattern set than the one in NIDS/NIPS. In this paper we aim at exploring an efficient pattern matching architecture on ASICs for most content or deep packet inspection based security applications. The pattern set ranges from tens of thousands to hundreds of thousands.

III. PROBLEM ANALYSES

Pattern matching has different meanings in different context. In our work, patterns can be found anywhere in the input data. Typically Aho-Corasick(AC) and its derivative algorithms are used. These algorithms can build a DFA from pattern set and run it in step for searching. In this section, we give analyses about the inefficiency of DFA of AC and present the motivation of our approaches.

Figure 1 shows the DFA of AC for accepting pattern set of {SEC, SSH}. There are 6 states and 16 transitions in it.

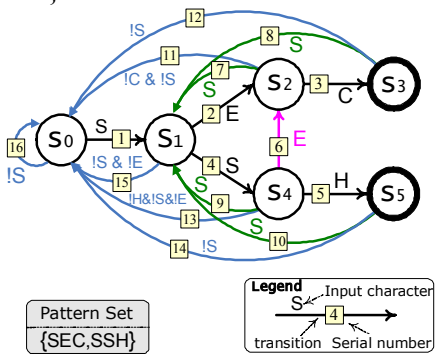


Figure 1. DFA of AC for accepting pattern set {SEC, SSH}

Considered the different functions, all transition rules can be classified into four categories: basic transitions, cross transitions, failure transitions and restartable transitions.

- Basic transitions are the ones that successfully accept pattern set from the beginning state(S_0), such as the one numbered 1-5 in figure 1. They act as the backbone of the DFA. All states can be decided after the basic transitions are generated.
- Cross transitions are the ones that transfer from one pattern to another or one part to another part within one pattern. They are required because the sub-pattern that one state accept and represent may be the same as the prefix of other patterns or this pattern itself, such as transition No.6.
- Restartable transitions transfer current state to the next states of S_0. They restart the procedure of matching of DFA, such as transition No.7-10.
- Failure transitions transfer current state to S_0, which stand for failure in stepping forward of the matching. In figure 1, 11-16 belong to this category.

Because more transitions will consume more memory in AC like algorithms, optimizations are proposed to reduce the number of transitions.

Although the four categories are not explicitly proposed, Jan v. Lunteren[1] successfully reduced all the failure and restartable transitions to at most 256 transitions using the priority approach. He set failure and restartable transitions a priority of 0 and 1 respectively. Then the natural feature of lower priority can exploit “don’t care” technique, which does not care the accepting characters using lower priority, so that all failure transitions are combined to one transition (lowest priority with “don’t care”). Similarly, restartable transitions are combined to 256 transitions (lower priority with “don’t care”). Without distinguishing cross transitions and basic transitions, he set both of them priority of 2.

In our work, the motivation is to reduce the cross transitions, because they outnumber all the other ones. Cross transitions represent common sub-patterns, and we cannot efficiently avoid them by selecting pattern set. A natural method is to partition the pattern set to many unrelated smaller ones, handled by individual DFAs, so as to avoid common sub-patterns. Here we give some statistics based on Snort and ClamAV rules to evaluate this natural method.

Snort[16] is an open source network intrusion detection system consisting of thousands of patterns. We use one pattern set of 2005, which has about 3000 patterns. After eliminating duplicated ones, there are 1785 different patterns left. It stands for a moderate set.

ClamAV [17] is an open source anti-virus system with a daily updated signature set. The set of Oct. 8, 2006 is used with about 50000 signatures. After eliminating duplicated ones, there are 49644 patterns left. It stands for a large set.

In our statistics, a pattern set is divided into smaller ones by the method of even partition on the number of patterns. For example, a pattern set with 256 patterns can be divided into two smaller sets with 128 patterns each. During the process of partitioning, no optimization is used. Because

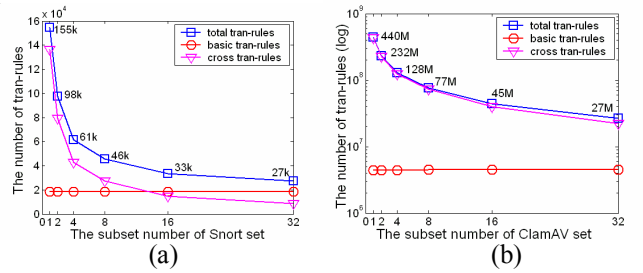


Figure 2. Statistics of basic and cross transitions

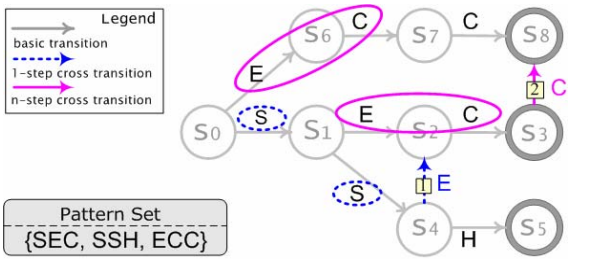


Figure 3. 1-step and n-step cross transitions

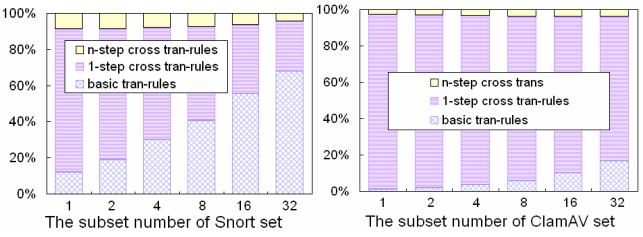


Figure 4. Statistics of 1-step and n-step cross transitions

failure and restartable transitions can be optimized to at most 256 [1], they are not counted in statistics.

In figure 2, (a) and (b) show the trend of the number of transitions("tran-rules" for short) in different categories in Snort and ClamAV set respectively. The number of subsets is named "subset number". For subsets, the results are the sum of all related ones in the subsets. As we can see, the total number of transitions is changing with the number of cross ones. The more subsets a pattern set has, the less total transitions it is. However, for larger set, such as ClamAV, the method of partition still results in huge cross transitions.

To further the understanding of cross transitions, they are classified into two types: 1-step cross transition and n-step (n>1) cross transition, shown in figure 3 for example.

Figure 3 shows the DFA of AC for pattern set {SEC, SSH, ECC}. Transition 1 is a 1-step cross transition because one-character prefix of "SEC" matches substring of "SSH". Transition 2 is an n-step cross transition because two-character prefix of "ECC" matches substring of "SEC".

Taking real patterns as example, figure 4 (a) and (b) show respectively the number of 1-step and n-step cross transitions. Obviously, the number of 1-step is the major part of all cross transitions, even of all transitions. The proportion of 1-step becomes larger when the pattern set becomes larger.

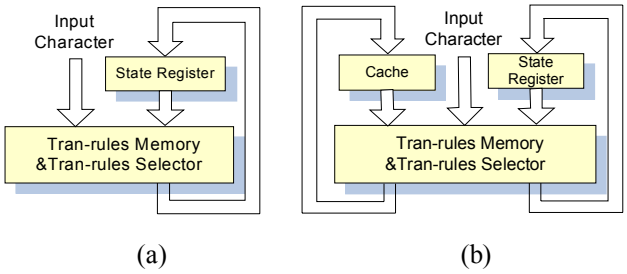


Figure 5. DFA and CDFA

From our research, we believe that the reason why cross transition rules in AC like algorithms cannot be efficiently optimized or eliminated is because of their basic model, i.e., DFA. To improve the efficiency, a more suitable basic model for pattern matching algorithms is needed. In our work, a novel model, namely cached DFA, is proposed, which is the basis of our solution.

IV. OUR APPROACH

A. *Cached Deterministic Finite Automata*

As illustrated in figure 5(a), the traditional DFA is a simple and concise model. The transitions are stored in transition rules ("tran-rules") memory and accessed by tran-rules selector. The next state is only determined by input character and current state (stored in state register).

We extend the traditional DFA model by using certain number of registers as cache (only one register is used in this work), as figure 5 (b) shows. Then some information can be temporarily stored and employed.

The new model is named as cached DFA (CDFA), because the cache acts as an internal structure and is handled by CDFA automatically, similar to the cache in memory systems. With cached states, CDFA extends DFA with the capability of temporary memory. The next state in CDFA is determined not only by input character and current state but also the cached states, which is the main difference between DFA and CDFA.

A CDFA is a 7-tuple, $\{K, \Sigma, s_0, F, N, \delta, \theta\}$, concluding

- A finite set of states, K
- A finite input character set, called "alphabet", Σ
- A start state, s_0
- A set of accepting states, $F \subseteq K$
- The number of cache size, N, in this paper $N=1$
- A *transition function* that , $\delta : K \times K^N \times \Sigma \to K$
- A *caching function*, $\theta : K \times \Sigma \to K$

For CDFA, the next state is determined by current state, current input and N cache states, which is described by transition function. The N cache states are handled by the policy of caching function.

Another important difference in CDFA from DFA is that caching function and transition function are both user-defined. It is understandable if we take various policies for caching in memory system as an example.

B. *Pattern Matching Algorithm based on CDFA*

AC and its derivative algorithms are all based on DFA. Here we propose a multiple pattern matching algorithm based on CDFA, called AC-CDFA (ACC). The process of ACC is similar to AC except the operation of CDFA. It has all the merits of AC and produces much less transitions.

The basic idea of ACC is to dynamically generate and accept all 1-step cross transitions by using CDFA, so that 1-step ones are no longer stored in memory. That is, all 1-step cross transitions in DFA are eliminated in CDFA. As the major part is eliminated, ACC can save much memory space.

At the preprocessing stage, while CDFA is built from a pattern set, only basic transitions and n-step cross transitions called *c-trans* in original DFA are generated

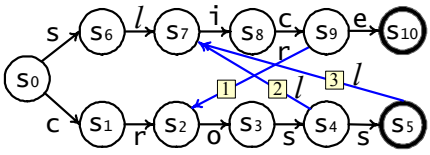
At the matching stage, CDFA accepts input characters using *transition function* and *caching function*.

Transition function δ, which is used to generate next state, is defined as follows. The next state is first determined by input character and current state. If there is no corresponding c-trans, the next state is then determined by input character and cached state (in our algorithm, only one cached state is used, i.e. N=1).

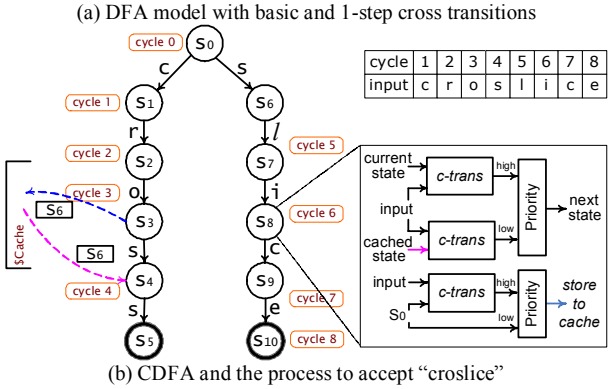
Caching function θ, which is used to cache state in CDFA, is defined as follows. If the beginning state s_0 accepts current character and step to the state s_j by c-trans, s_j is stored as cache state. Otherwise, s_0 is stored as cache state.

Actually, in ACC, not only 1-step cross transitions but also failure and restartable transitions can be dynamically generated and never required to store. At the matching stage, the "store-to-cache" operation is always performed at every cycle. An example of this process is given in figure 6 to show our algorithm.

In figure 6 (a), an original DFA model for the pattern set {cross, slice} is given, in which all failure and restartable transitions are omitted for concision. Transitions numbered 1-3 are 1-step cross transitions. CDFA model is built in figure 6 (b) for ACC which only consists of basic transitions (and n-step cross transitions if there are) in original DFA. At the same time, a cache is added and the function of state transfer is changed. The process to accept the input "croslice" is taken as an example. When a character comes, the next state is firstly determined by current c-trans if there is a proper one, such as S_1, S_2. If no c-trans are found, cached state is fetched and used as "current state" to find proper c-trans, and then next state can be determined. In cycle 5, there are no c-trans for "S_4" to accept character "*l*". Then cached state "S_6" is fetched, and there is one transition from "S_6" to "S_7" by accepting "*l*", so next state is "S_7".



(a) DFA model with basic and 1-step cross transitions



(b) CDFA and the process to accept "croslice"

Figure 6. DFA and CDFA model for pattern set {cross, slice}

In ACC, ONE state cache is used to eliminate 1-step cross transitions. Theoretically, N state caches can be used in CDFA to eliminate all cross transitions from 1-step to N-step. Because 1-step cross transitions are the major part, our work processes them first. In appendix, a proof is given to show that our algorithm and AC algorithm are equivalent.

With ACC, all 1-step cross transitions in figure 4 can be eliminated. As a result, 95.9% total transition rules are released for ClamAV set, and 79.2% for Snort set. For cross transitions, 96.9% are released for ClamAV set, and 89.9% for Snort set. In above statistics, failure and restartable transitions are excluded, because they are easy to control within 256 transition rules by the priority approach.

C. *Next State Addressing Scheme*

Our algorithm can be implmented into both software and hardware solutions for multiple pattern matching. In our work, we apply it to hardware solution.

When AC like algorithms are used in hardware solutions, another important issue is how to efficiently store and access transitions in memory. Here, we present another idea, namely *next state addressing (NSA)*, to handle this issue.

There are two methods to store and access transitions of DFA in other papers, using CAM and hashing.

Content-addressable memory (CAM) is a special type of computer memory, which can compare the input internally and output the address of matched content in parallel. When the transitions are stored in CAM, the address can be output in one cycle and the next state can be found together with a SRAM. [23] CAMs are good but they consume lots of chip area, power and cost, because of their internal architectures.

Hashing uses common memory. The address of memory for transition rules is computed from current state and input character. Then some candidate items are back for checking. BART[1] and other hashing methods are relatively chip-area efficient, but inevitably result in conflicts in hashing item.

The main purpose of next state addressing (NSA) is to precisely address the proper transition rule for the current state with common memory. The essence of states in finite automata is uniquely indexed, which help specify transition rules. The only useful information is the character state accepts. NSA store characters only. Compared with other methods, it stores transition rules more efficiently.

The basic idea of NSA is to exploit the current state in order to precisely calculate one possible next state, and take the next state as the address to access the memory of transition rules. After finding the transition, it is verified by comparing the incoming character and the accessed one.

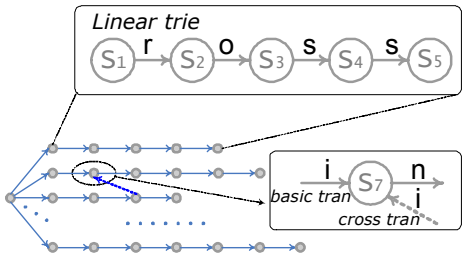


Figure 7. Linear trie in DFA or CDFA

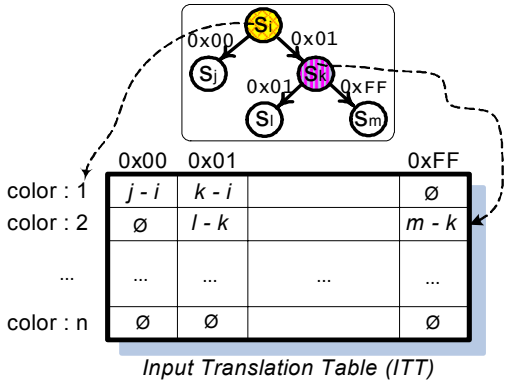


Figure 8. Details of ITT

The reason why next (future) state can be used as address is based on two observations. First, after 1-step cross transitions (96.9% of all cross transitions for ClamAV, and 89.9% for Snort) are eliminated, there are a lot of linear trie in CDFA, as shown in figure 7. Each state in the linear trie has only one next state. By orderly labeling the states in linear trie, next state can be calculated from the current state.

The second observation is that, for each state, the input characters of all "incoming" transitions are the same no matter how many and what kind of they are. That is, each state accepts only one character when it is regarded as "next state" of one transition. For example, in figure 7, S_7 only accepts character "i" no matter what types of transition.

NSA is proposed based on the two above observations. To implement the method, states of CDFA (or DFA) should be specially numbered. During compilation (CDFA or DFA is built from a pattern set), states are labeled by using the following algorithm.

- The start state of CDFA (or DFA) is numbered as S_0.
- If a state has only one next state, the next state is numbered one bigger than the current one.
- If a state has several next states, depth-first algorithm is used to number the next states.

To use next states as addresses, the address (next state) should be calculated by current state and input. For the states in the linear trie, the next state can be calculated by adding one. However, there are some states that have several next states. (They are fewer in CDFA because of fewer transitions.) To handle this situation, those states are individually colored. The color can be considered as the second number of the state. We use the word "color" just for avoiding the confusion of state number.

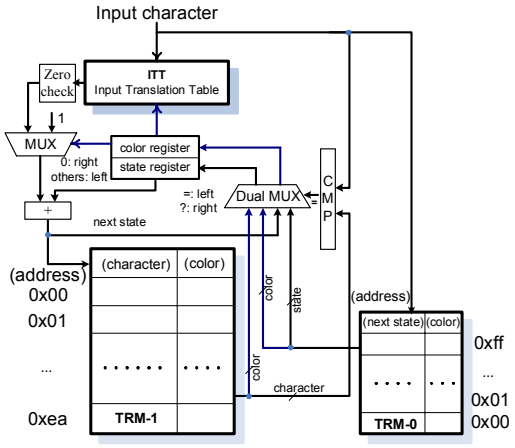


Figure 9. Overall architecture of NSA scheme

To calculate next state for colored states, a structure named *input translation table* (ITT) is used as figure 8 shows. Each color corresponds to one entry of ITT and each entry has 256 items which correspond to 256 input characters. Each item stores the result of subtracting next state number from current state. By state color and input character, next state of colored state can be calculated by looking up ITT table. In ITT table, Ø means null value, which is actually invalid. The next state is calculated differently for colored and non-colored state using the following equation.

$$next\_state = \begin{cases} current\_state + 1, & if\ current\_color = 0 \\ current\_state + itt(input, current\_color), & if\ current\_color \neq 0 \end{cases}$$

Besides the basic idea of NSA, *NSA scheme* for storing and accessing transitions consists of an optimization. That is, a standalone small memory is used to store all transitions from S_0, which is named as *transition rules' memory zero* (TRM-0). TRM0 is a lookup table structure that has 256 entries indexed by all possible input characters. Each entry stores the state after S_0 accepting the addressed character or S_0. Details can be found in [1]. The other transitions are stored in *transition rules' memory 1* (TRM-1).

The overall architecture of NSA scheme is shown in figure 9. Some trivial details are omitted.

In the NSA scheme, the next state is firstly calculated. Then the TRM-1 is secondly accessed via next state as address. Then the result is compared with the input character. If characters are the same, CDFA steps forward and the next state and the color are turned into current ones. If they are different, the output state and color of TRM-0 are set to the current ones. It means that failure or restartable transition is performed.

D. *Our Pattern Matching Architecture*

Our pattern matching architecture incorporates our ACC algorithm based on CDFA (for fewer transitions) and NSA scheme (for effectively storing transitions), as shown in figure 10 (Some trivial details are omitted for concision).

In our architecture, ITT and TRM-1 are designed to use dual port memory, so that state register and state cache can access them in parallel. (Single port memory can also be

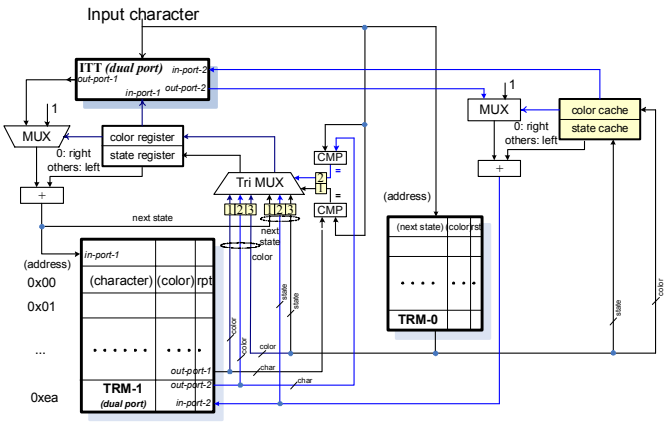


Figure 10. Our pattern matching architecture

used regardless of parallelism accessing.) One Tri-MUX is used as the function for priority switching. It accepts three sets of inputs (three colors and three states) and outputs one color and state based on two control signals. The control signals are results of two CMP (compare) units, numbered as "1" and "2". The function of Tri-MUX is

$$output(state,color)=\begin{cases}(state,color,"1"), & if\ "1"=equal, \quad high\_priority\\ (state,color,"2"), & if\ "2"=equal, \quad medium\_priority\\ (state,color,"3"), & others, \quad low\_priority\end{cases}$$

When an input character arrives, register (state and color) and cache (state and color) both start to access TRM-1 in parallel. At the same time, TRM-0 is accessed for failure and restartable transitions. Three possible next states and their corresponding colors are generated and sent to Tri-MUX component. It outputs only one state and its color to update state and color registers respectively. The update drives CDFA stepping forward. The caches (state and color) are directly overwritten by the outputs of TRM-0.

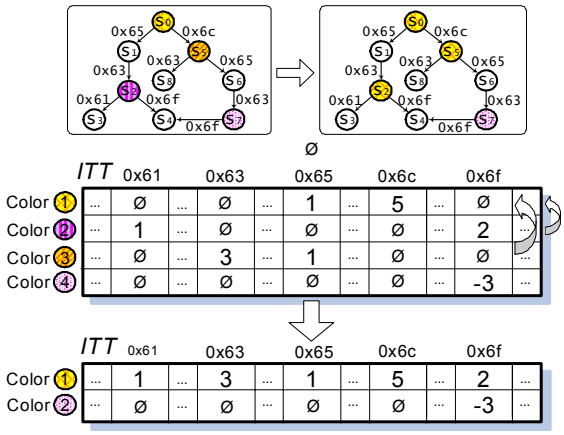
V. Optimizations and Results

A. ITT Optimizations

In the original design, one colored state is assigned to an entry of ITT, with 256 items. Most colored states have only a few next states, and most items of their corresponding entry of ITT are set to Ø. To effectively utilize ITT's space, two optimizations (entry combination and set-associative strategy) are proposed in this section.

1) Entry Combination

The basic idea of entry combination is to combine some entries into one. Two entries can be combined if and only if all items of the same column have no conflicts. The conflicts may occur in two cases. (1) As two corresponding items have different non-Ø values, conflict (resource conflict) exists. (2) As the non-Ø item overwrites the Ø one, the next state of written entry is the state calculated by adding the non-zero item. If that state accepts the same character, it leads to conflict, called overwriting conflict.



(a) An example

```
for (k=0;k<256;k++)
  if ((Ci[k]==0)&&(Cj[k]!=0))
    if (Acp(S(Ci)+Cj[k])==k)
      return NO
  if ((Ci[k]!=0)&&(Cj[k]==0))
    if (Acp(S(Cj)+Ci[k])==k)
      return NO
  if ((Ci[k]!=0)&&(Cj[k]!=0)&&(Ci[k]!=Cj[k]))
    return NO
return YES
```

(b) The algorithm for entry combination

Figure 11. Entry combination for ITT optimization

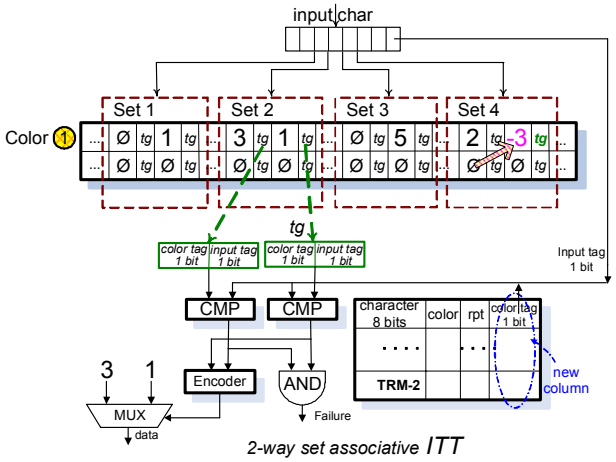
Given two colors Ci and Cj, the kth items of their entries are designated as $Ci[k]$ and $Cj[k]$, and the corresponding states are $S(Ci)$ and $S(Cj)$. $Acp(S)$ represents the character which triggers the transition to S. The algorithm for judging whether they can be combined is listed in figure 11(b). For example, in figure 11(a), the initial ITT has 4 entries corresponding to the 4 colored states of CDFA. After combination, only 2 colors are left.

For each entry of ITT, the algorithm tries to combine it with the previous ones. In figure 11(a), color "2" is evaluated whether it can be combined with color "1" first. It turns out that, the two can be combined and color "1" is updated. Then color "3" is evaluated with color "1".

2) Set-associative Strategy

In figure 11(a), although color "4" just uses one item, it cannot be combined with color "2", because color "2" has different value of item "0x6f". To address this issue, a method of set associativity is presented, which is similar to the one in cache system of micro-architecture.

For N-way set-associative, there are $256/N$ sets. The qth item of pth set corresponds to $column(p,q)$, and the content of color Ci is $Ci[column(p,q)]$. For two colors Ci and Cj, the algorithm judging whether they can be combined with set-associative strategy is shown in figure 12(b).



(a) An example for 2-way set associative ITT

```
for (p=0;p<256/N;p++)
{
   count=0
   for (q=0;q<N;q++)
   {
      if ( Ci[column(p,q)]!=Ø) count++;

      if ( Cj[column(p,q)]!=Ø) count++;
   }
   if (count>N) return NO
}
return YES
```

(b) The algorithm for set associativity

Figure 12. 2-way set associative strategy

Two entries can be combined by set-associative strategy if and only if all items of the same set have no conflicts. The conflict may occur only when all items of any set have more than N non-Ø values. An example for 2-way set associative strategy is shown in figure 12(a).

Similar to traditional set-associativity in memory system, tag is attached to each item. The tag has two fields: color tag and input tag, which represent the color and input index before set-associative strategy respectively.

B. Analyses and Results of Memory Requirement

In CDFA, all the 1-step cross transitions, failure and restartable transitions are eliminated, which are the major part of total ones. Thus ACC based on CDFA is memory efficient. In this section, we focus on analyzing memory requirements of NSA scheme.

Suppose that the given pattern set has n patterns with m characters. The CDFA model has s states, including c colored states. After ITT optimization, there are c' different colors, with α-way set associative strategy. Because CDFA has eliminated all 1-step cross transitions (more than 90% of all cross ones), all cross ones can be regarded as eliminated. Thus the number of total transition rules is approximately equal to the number of basic transitions. As each state except S_0 corresponds to one basic transition, the total number of transition rule is s. The memory requirement of TRM is (in bytes)

$$M_{TRM} \approx M_{TRM-1} \approx s \times (1 + \lceil \log_2 c' \rceil / 8 + \lceil \log_2 a \rceil / 8)$$

The memory requirement of ITT (in bytes) is, (one more sign bit for negative value)

$$M_{ITT} = 32 \times c' \times (\lceil \log_2 s \rceil + 1 + 2 \times \lceil \log_2 a \rceil)$$

Therefore, the memory requirement for each transition rule is approximately

$$M_{tran} = 1 + 0.125 \times (\lceil \log_2 a \rceil + \lceil \log_2 c' \rceil) + 32 \times c' \times (\lceil \log_2 s \rceil + 1 + 2 \times \lceil \log_2 a \rceil) / s$$

The traditional ideal case is that each transition rule consists of input and one state, requiring

$$M_{ideal} = 1 + \lceil \log_2 s \rceil$$

Thus compared to the ideal cases, approximately if $s > 32 \times c'$, NSA scheme will be more efficient. This is very common after two steps of optimizations for ITT.

TABLE I. COMPARISON OF MEMORY USAGE FOR SNORT SET

Architectures		2 subsets	4	8	16	mem/char*
		1785 patterns with 29.0K characters				
ACC with NSA	2-way	256KB	181	168	151	6.2B
	4-way	163KB	123	115	106	4.2B
	8-way	129KB	97	87	81	3.3B
B-FSM [1]		1.5K patterns with 25.2K characters				
		N/A	188	120	92	7.4B
Bitmap compression [15]		2.8MB				154B
Path compression [15]		1.1MB				60B

*: Mem/char column is calculated for 4 subsets (shaded) only.

The result of memory utilization for Snort pattern set is listed in table 1 compared with other methods.

In table 1, our architecture uses nearly constant memory from 4 subsets to 32 subsets. This implies that with CDFA, pattern set partitioning is no longer the key to reduce memory usage, compared to other architectures. For the same subset, ACC algorithm requires less memory than other architectures, such as B-FSM scheme with optimized partitioning. Table 2 gives the memory usage for ClamAV pattern set. The similar features are concluded, and at least 9.5MB memory is required for about 50K patterns totaling 4.44M characters.

When our architecture is compared with others, two facts should be taken into account. The first one is that SDRAM or SRAM are used in our method other than CAM or TCAM. The other one is that our results are given in the fewer number of subsets. This means our method is the most straightforward way to handle larger pattern set.

To explain the overhead of more subsets, our architecture is implemented in verilog. The verified HDL architecture is synthesized with a 0.18μm standard cell library using Synopsys tools. Synthesized results about chip area are shown in figure 13. It is obvious that more subsets can result in more chip area, and the conclusion is fit for other architectures.

TABLE II. MEMORY USAGE FOR CLAMAV PATTERN SET

CDFA	32subsets	64	128	256	512	Mem/char*
2-way	26.8MB	19.9	16.3	13.9	12.0	6.0B
4-way	21.6MB	16.6	13.8	11.7	10.3	4.9B
8-way	18.7MB	14.8	12.4	10.8	9.5	4.2B

*: Mem/char column is calculated for 32 subsets (shaded) only.

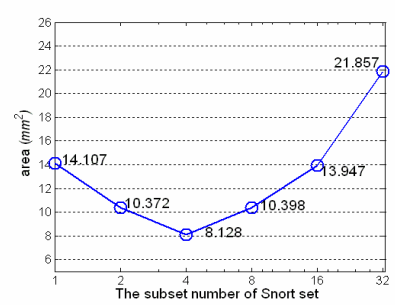


Figure 13. Chip area of Snort with 8-way set associativity

C. Critical Path Optimization

In our architecture, two larger memories (ITT and TRM) are accessed in one cycle respectively. To achieve high frequency, the critical path can be pipelined to more stages.

Since next state is highly dependent on current state, pipelining becomes very challenging for single data steam, therefore we adopt the method of fine-grain multi-threading [20]. The basic idea is to concurrently fetch from different data streams on a cycle-by-cycle basis, while only one data stream is involved at each pipelining stage. Suppose that we have two pipeline stages in figure 14, two data streams are involved. On even cycles, one character from data stream A arrives and occupies the first pipeline stage. On odd cycles, one character from data stream B arrives at the first stage while the character from stream A occupies the second stage.

Stream parallelism is one feature for network applications. Therefore our method of pipelining is natural and feasible. Using this method, our architecture can achieve very high throughput. Table 3 shows the comparison of our design to the others on critical path delay and throughput.

The item of "more pipelines" is the longest delay in the design. The critical path is the ADD component that computes next state. It can be further optimized by using customized component.

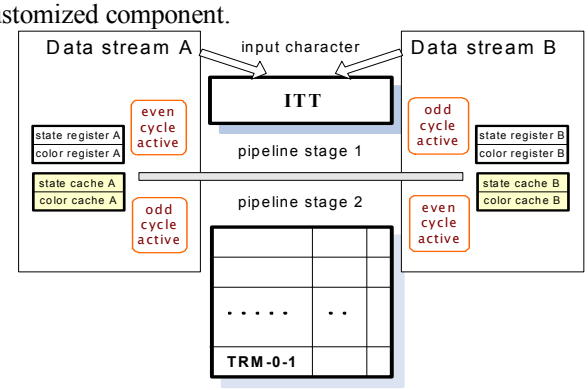


Figure 14. Method of fine-grain multithreading-like pipelining

TABLE III. COMPARISON OF CRITICAL PATH DELAY

Architectures		Critical path delay (ns)	Matching speed (Gbps)	Notes (1 byte/cycle input)
ACC with NSA	No pipeline	1.65	6.1	0.18µm tech
	2 pipelines	1.18	8.5	
	more pipelines	0.85	11.7	
Bit Split FSM [12]		N/A	8.4~10.0	simulator
B-FSM [1]		N/A	0.8~1.0	FPGA
Cho-MSmith [10]		1.12	7.14	0.18µm tech
Predec CAMs [11]		N/A	2.68	FPGA
Bloom Filter [7]		N/A	0.5	FPGA
Decoder NFA [4]		N/A	2.0	FPGA
USC Unary [21]		N/A	2.1	FPGA
Compressed DFA[22]		N/A	1~10	calculated

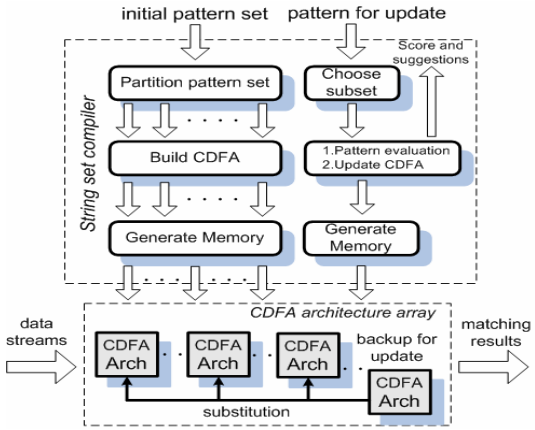


Figure 15. Pattern set compiler

D. Pattern Set Compilier and Dyanmic Update

Our pattern set compiler has two modes, as shown in figure 15. The first mode is building CDFA from initial pattern set and allocating memory for our architecture. The second mode is pattern update. Based on the algorithms of building CDFA and coloring states, score and suggestions are given to the pattern for update. This will help the experts of network security to define more efficient patterns (signatures) for our architecture.

The pattern set compiler is implemented in C language. Only seconds are used to build CDFA for Snort and ClamAV set on a normal server.

To achieve dynamic update without interrupting ongoing data matching operations, a backup CDFA architecture is used as the method of Tan and Sherwood [12].

E. Regular Expression Matching

In this work, we mainly focus on normal patterns such as strings and simple variations of strings. The CDFA model can also be used for multiple regular expression matching and still performed with very efficient memory utilization.

VI. CONCLUSIONS

This paper proposes a memory efficient pattern matching architecture for all kinds of network security applications, with the size of pattern set ranging from 1K to 10M or even more. Our pattern matching algorithm, ACC, is based on a novel model, namely CDFA, which can eliminate more than 90% transitions for the applications of network security. An NSA scheme for efficiently storing and accessing transitions is proposed. Moreover, our pattern matching architecture is optimized with several approaches to achieve better performance and less memory utilization. Experiments show that only 81KB memory (SRAM or SDRAM) is needed for about 1.8K Snort rules (total 29K characters) and 9.5MB for 50K ClamAV rules (total 4.44M characters).

REFERENCES

[1] Jan van Lunteren. High-Performance Pattern-Matching for Intrusion Detection. In 25th Conference of IEEE INFOCOM, Apr. 2006

[2] Z.K. Baker and V.K.Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In 12th Annual IEEE FCCM, April 2004

[3] Z.K. Baker and V.K.Prasanna. A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. In 12th Annual IEEE FCCM, April 2004

[4] C.R.Clark and D.E. Schimmel. Scalable Pattern Matching for high Speed Networks. In 12th Annual IEEE FCCM, April 2004

[5] Fang Yu, R. H. Katz and T.V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In 12th Conference of IEEE ICNP, Oct. 2004

[6] Long Bu, John A. Chandy. FPGA Based Network Intrusion Detection using Content Addressable Memories. In Conference of IEEE FPT 2004, Dec. 2004

[7] Michael Attig, Sarang D., John Lockwook. Implementation Results of Bloom Filters for String Matching. In Conference of IEEE FPT 2004, Dec. 2004

[8] Y. H. Cho and W. H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. In 12th Annual IEEE FCCM, April 2004.

[9] Y. H. Cho and W. H. Mangione-Smith. Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network. In 13th Annual IEEE FCCM, April 2005

[10] Y. H. Cho and W. H. Mangione-Smith. A Pattern Matching Co-processor for Network Security. In 42nd Conference of DAC, June, 2005

[11] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for Efficient and High-speed NIDS Pattern Matching. In 12th Annual IEEE FCCM, April 2004

[12] Lin Tan, T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In 32nd Annual International Symposium on Computer Architecture, ISCA, 2005

[13] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer and N. Weaver. Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In USENIX Hot Security, 2006

[14] Hongbin Lu, K. Zheng, B. Liu, X. Zhang and Y. Liu. A Memory-Efficient Parallel String Matching Architecture for High Speed Intrusion Detection. In IEEE Journal on Selected Areas in Communications, Vol. 24, No.10. Oct. 2006

[15] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection; In 23rd Conference of IEEE INFOCOM, Mar. 2004

[16] M. Roesh. Snort – lightweight intrusion detection for Networks. In Proceedings of 13th Systems Administration Conference, Nov. 1999

[17] ClamAV: http://www.clamav.net/ (accessed on Oct. 2006)

[18] B. C. Brodie, R. K. Cytron and D. E. Taylor. A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching. In 33rd International Symposium on Computer Architecture, ISCA, 2006

[19] A.V. Aho and M.J. Corasick. Efficient String Matching: An aid to Bibliographic Search. Communications of the ACM, vol.18, 1975

[20] M. Loikkanen and N. Bagherzadeh. A fine-grain multithreading superscalar architecture. In proceedings of the Conference Parallel Architectures and Compilation Techniques, Oct. 1996.

[21] Z.K. Baker, V. K. Prasanna. High-throughput linked-pattern matching for intrusion detection systems. In symposium on Architecture for Networking and Communications Systems (ANCS), Oct. 2005

[22] Alicherry, M. Muthuprasanna, M. Kumar, V., "High Speed Pattern Matching for Network IDS/IPS," Proceedings of the 2006 14th IEEE International Conference on Network Protocols (ICNP'2006), Nov. 2006, pp:187-196.

[23] Sensory Networks, "Apparatus and Method for Memory Efficient, Programmable, Pattern Matching Finite State Machine Hardware", US Patent No. 7082044 B2, July 25, 2006

VII. APPENDIX

Theorem 1: The Aho-Corasick algorithm and ACC algorithm in our work are equivalent.

Proof: Because Aho-Corasick algorithm is based on DFA, our algorithm and AC are equivalent if and only if the DFA and CDFA from a pattern set are equivalent. Here we define DFA as a 5-tuple $\{K, \Sigma, s_0, F, \delta\}$. The transition function δ can be classified to five sub-functions: δ_{basic} for basic transitions, $\delta_{failure}$ for failure ones, $\delta_{restart}$ for restartable ones, $\delta_{1-cross}$ for 1-step cross ones and $\delta_{n-cross}$ for n-step cross ones. CDFA is defined as a 7-tuple $\{K, \Sigma, s_0, F, N, \delta', \theta\}$. Now we can prove that using δ' and θ in our algorithm, the transition function δ in DFA can be represented.

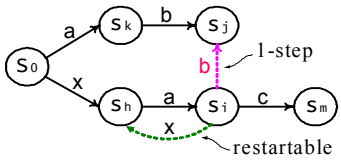


Figure: For proof

For basic and n-step transitions, CDFA remains the same.

For 1-step cross transition, $\delta_{1-cross}(s_i, b) = s_j$, it is equal to $\delta_{basic}(s_k, b) = s_j$. Because the transition is 1-step cross one, in state "s_h", when character "a" comes, the state "s_k" is cached in CDFA, that is $\theta(s_h, a) = s_k$. So that in state "s_i", when $\delta'(s_i, b)$ returns null, the low priority policy $\delta'(\theta(s_h, a), b) = s_j$ is performed. That is, all 1-step cross transitions in DFA can be represented using CDFA.

For restartable transition in DFA, $\delta_{restart}(s_i, x) = s_h$, it can be replaced by $\theta(s_i, x) = \delta(s_0, x) = s_h$.

For failure transition in DFA, suppose $\delta_{failure}(s_i, k) = s_0$. It can be replaced by $\theta(s_i, k) = s_0$.

Now we prove that all the transitions in DFA can be represented by transition and caching functions in CDFA. In other words, our algorithm and AC are equivalent.