行政院國家科學委員會專題研究計畫 成果報告

正規表示法樣式比對之演算法與硬體架構設計
研究成果報告(精簡版)

計 畫 類 別 ：個別型
計 畫 編 號 ：NSC 97-2218-E-003-002-
執 行 期 間 ：97 年 08 月 01 日至 98 年 08 月 31 日
執 行 單 位 ：國立臺灣師範大學工業科技教育學系（所）

計畫主持人：林政宏

計畫參與人員：大專生-兼任助理人員：黃俊程
大專生-兼任助理人員：楊雲州
大專生-兼任助理人員：呂信德

報 告 附 件 ：出席國際會議研究心得報告及發表論文

處 理 方 式 ：本計畫可公開查詢

中　華　民　國　98 年 10 月 02 日

行政院國家科學委員會補助專題研究計畫成果報告

正規表示法樣式比對之演算法與硬體架構設計

計畫類別：■個別型計畫 □整合型計畫

計畫編號：97-2218-E-003-002-

執行期間：97年8月1日至 98年8月31日

計畫主持人：林政宏

共同主持人：

計畫參與人員：黃俊程、楊雲洲、呂信德

成果報告類型(依經費核定清單規定繳交)：■精簡報告　□完整報告

本成果報告包括以下應繳交之附件：
□赴國外出差或研習心得報告一份
□赴大陸地區出差或研習心得報告一份
■出席國際學術會議心得報告及發表之論文各一份
□國際合作研究計畫國外研究報告書一份

處理方式：除產學合作研究計畫、提升產業技術及人才培育研究計畫、列管計畫及下列情形者外，得立即公開查詢
□涉及專利或其他智慧財產權，□一年□二年後可公開查詢

執行單位：國立台灣師範大學工業科技教育學系

中　華　民　國　九十　八　年　八　月　卅一　日

一、 中文摘要

正規表示法（regular expression）比對被廣泛運用在網路入侵偵測系統（network intrusion detection system, NIDS）因為其強大的表示能力與彈性。為了同時比對多個樣式，將多個樣式編譯成一個有限狀態機（deterministic finite automata, DFA）是一種直覺且有效的方法。然而，傳統記憶體架構對於特定複雜的正規表示法樣式面臨記憶體爆量的問題。這類的複雜樣式可分為兩類。其中第一類為包含wildcard 與constraint repetitions的正規表示法樣式，第二類為包含多重star-closure (“.*”)的樣式。

在這個計畫中，我們提出三個技術來降低正規表示法樣式的記憶體需求。首先我們提出一個有效率的樣式比對演算法（efficient pattern matching algorithm）[1]可以藉由分享共同樣式以降低記憶體需求。第二，針對包含多重star-closure (“.*”)的複雜樣式，我們提出一個階層式的狀態機架構（hierarchical state machine architecture）[2]，可以有效降低記憶體需求。最後，我們提出一個複合的記憶體架構（hybrid memory architecture）[3]，它藉由插入些微的邏輯元件可以改良傳統記憶體架構降低包含wildcard 與constraint repetitions的正規表示法樣式的記憶體需求。研究成果分別被IEEE Transactions on Very Large Scale Integration (VLSI) Systems接受，並已分別發表於今年(2009) ACM Great Lakes Symposium on VLSI (GLSVLSI 2009) 與IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2009)。.

關鍵字：正規表示法，樣式比對、有限狀態機

二、 英文摘要

Regular expression matching has been widely used in Network Intrusion Detection Systems due to its strong expressive power and flexibility. To match multiple regular expression patterns simultaneously, compiling multiple regular expressions into a composite DFA is an intuitive and efficient way. However, traditional memory architecture suffers from the problem of memory explosion for certain complex regular expressions which can be classified into two main types. One is composed of wildcard and constraint repetitions. Another one is composed of multiple instances of star-closure (".*").

In this project, we have proposed three techniques to reduce memory requirement of regular expression patterns. First, we propose an efficient pattern matching algorithm which can reduce memory requirement by sharing common sub-pattern. Second, we propose a hierarchical state machine architecture for matching the complex regular patterns composed of multiple star-closures. Finally, we propose a hybrid memory architecture which integrate logic and memory architecture for complex regular expressions composed of wildcard and constraint repetitions. The experiments demonstrate a significant reduction in memory for the complex regular expression patterns commonly used in network intrusion detection systems. The research results have been accepted by IEEE Transactions on Very Large Scale Integration (VLSI) Systems and have been published in the ACM Great Lakes Symposium on VLSI (GLSVLSI 2009) and IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2009), respectively.

Keywords: Regular Expression, Pattern Matching, Finite State Machine

三、 計畫緣由與目的

The main purpose of a network intrusion detection system (NIDS) is to inspect the packet header and payload against thousands of predefined malicious or suspicious patterns. These patterns describe behaviors such as denial of service attacks, port scans, or malware. To efficiently represent suspicious patterns, regular expressions are commonly adopted such as Snort[4], Bro[5], and ClamAV[6] because they have better expressive power and flexibility than explicit string patterns.

The key function of an NIDS is to perform the matching of regular expressions representing the suspicious patterns. Because software approaches of matching regular expressions can no longer meet the high throughput of today's networking, many hardware approaches are being proposed. Basically, these hardware approaches can be classified into two categories, logic [7][8][9][10] and memory [11][12][13][14][15] architectures. The logic architectures are mostly based on Nondeterministic Finite Automaton (NFA) and implemented on FPGA. On the other hand, the memory architectures are based on Deterministic Finite Automaton (DFA). The memory architectures have attracted a lot of attention because it is well suited for synthesis in an ASIC and has the advantages of easy re-configurability and scalability [17][20][21].

The memory architecture for matching regular expression pattern works as follows. First, regular expression patterns are compiled into a Deterministic Finite Automaton (DFA). Then, the corresponding state transition table is stored in memory to recognize regular expression patterns. Take a regular expression ".*AB.*CD" for example where the metacharacters ".*" denotes zero or more instances of any character. Figure 1 shows the DFA to recognize the regular expression pattern ".*AB.*CD", where State 0 is the initial state, and state 4 is the final state indicating the matching of the regular expression pattern. Figure 2 shows the memory architecture of matching regular expressions. First, a physical memory stores the information of the state transition graph, and the address registers store the current state. With the current state and input character, the next state and the *match vector* (*MV*) are fetched from the memory. The match vector is used to specify the matched pattern.

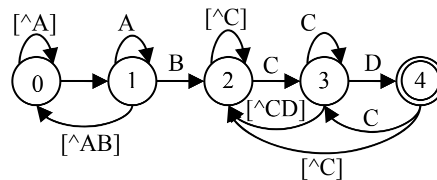


Figure 1: Deterministic finite automaton of ".*AB.*CD"

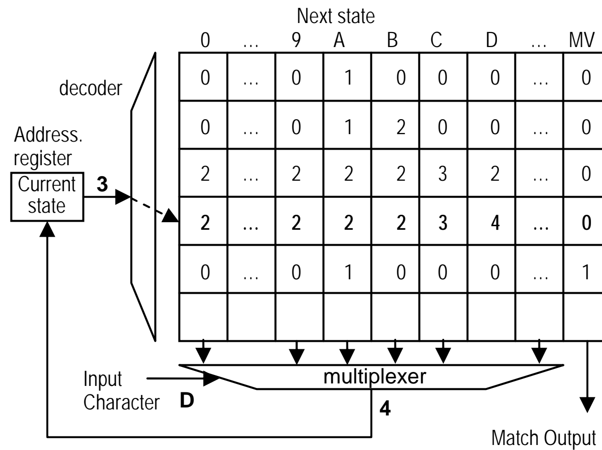


Figure 2: Simple memory architecture for pattern matching

It's well known that when *m* patterns are compiled into a single DFA, the matching time complexity for matching *m* patterns against the input texts of length n is reduced from O(*mn*) to O(*n*). Therefore, compiling multiple regular expression patterns into a single composite DFA is an efficient way to match multiple regular expression patterns. However, there are specific regular expression patterns which cause the composite DFA to grow exponentially. These complex regular expressions can be classified into two main types. One is composed of meta-characters and constraint repetitions. For example, the regular expression pattern for identifying the POP3 CAPA overflow attempt, ".*CAPA\s[^\n]{10}", and the pattern for identifying the POP3 TOP overflow attempt, ".*TOP\s[^\n]{10}" contain 112 and 137 states, respectively. Compiling these two patterns into a composite DFA will cause the number of state exponentially increasing to 649. Another complex regular expression is composed of multiple instances of star-closure (".*"). For example, using Flex [18] for compilation, the two patterns in Linux L7-filter, ".* member-name.*session.*player" and ".*rdpdr.*cliprdr.* rdpsnd" contain 47 and 43 states, respectively. When compiling these two patterns into a composite DFA, the number of states is up to 232.

四、 研究方法

This project addresses the memory issue for regular expression pattern matching. Our major tasks include the followings. (1) Efficient Pattern Matching Algorithm for Memory Architecture, (2) Hierarchical State Machine Architecture for Regular Expression Pattern Matching, and (3) Hybrid Memory Architecture for Regular Expression Matching. We describe individually the method of each task as following.

1. Efficient Pattern Matching Algorithm for Memory Architecture

In the first part of this project, we reuse those memory spaces of traditional AC state machine storing zero vectors {00} to store useful path information called *pathVec*. First, each bit of the pathVec corresponds to a string pattern. Then, if there exists a path from the initial state to a final state, which matches a string pattern, the corresponding bit of the pathVec of the states on the path

will be set to 1. Otherwise, they are set to 0. Consider the string pattern "bcdf" whose final state is state 4 in Fig. 3. The path from state 0, via states 1, 2, 3 to the final state 4 matches the first string pattern "bcdf". Therefore, the first bit of the pathVec of the states on the path, {state 0, state 1, state 2, state 3, and state 4}, is set to 1. Similarly, the path from state 0, via states 5, 6, 7 to the final state 8 matches the second string pattern "pcdg". Therefore, the second bit of the pathVec of the states on the path, {state 0, state 5, state 6, state 7, and state 8}, is set to 1. In addition, we add an additional bit, called *ifFinal*, to indicate whether the state is a final state. For example, because states 4 and 8 are final states, the ifFinal bits of states 4 and 8 are set to 1, the others are set to 0. As shown in Fig. 3, each state stores the pathVec and ifFinal as the form, "*pathVec_ ifFinal*". Compared with the original AC state machine in Fig. 3, we only add an additional bit to each state.

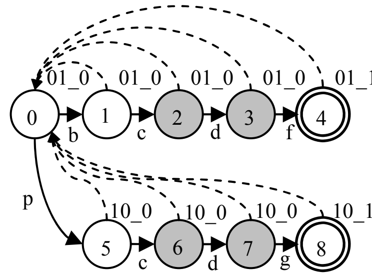


Fig. 3. New Data Structure, pathVec and ifFinal

We have mentioned that in this example, states 2 and 6, states 3 and 7 are similar because they have similar transitions. However, they are not equivalent. Note that two states are equivalent if and only if their next states are equivalent. In Fig. 3, state 3 and state 7 are similar but not equivalent 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.

In our algorithm, we define such similar states as pseudo-equivalent states. The definition is as follows.

Definition: Two states are defined as *pseudo-equivalent states* if they have identical input transitions, identical failure transitions, and identical ifFinal bit, but different next states.

In Fig. 3, states 2 and 6 are pseudo-equivalent states because they have identical input transitions *c*, identical failure transitions to state 0 and identical ifFinal bit 0. Also, state 3 and state 7 are pseudo-equivalent states. In our algorithm, the pseudo-equivalent states 2 and 6 are merged to be state 26 and states 3 and 7 are merged to be state 37, as shown in Fig. 4. And, the pathVec_ifFinal are updated by taking the union on the pathVec_ifFinal of the merged states. Therefore, the pathVec_ifFinal of states 26 and 37 are modified to be {11_0}.

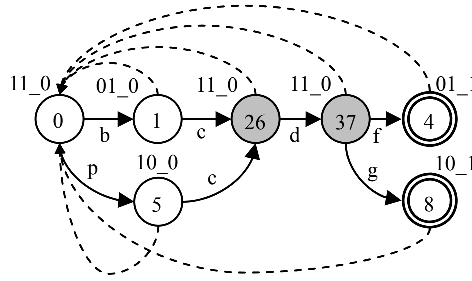


Fig. 4. New State diagram of merg_FSM

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 Fig. 5 where the string "pcdf" is applied. Initially, in state 0, the preReg is initialized 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 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 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}. The preReg remains $\{P_2P_1\}$ = {11 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 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.

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

Fig. 5. State transitions of the input string "pcdf"

Similarly, consider another example in Fig. 6 where the string "pcdg" is applied. The process of state traversal is similar to the previous example until the state machine reaches state 37 and the input character is *g*. After taking the input character *g*, the merg_FSM goes to state 8 and the preReg becomes $\{P_2P_1\}$ = {10} by performing a bitwise AND operation on the pathVec {10} of

state 8 and the current preReg {10}. Because the value of ifFinal is 1, the value of preReg, $\{P_2P_1\}$ = {10}, indicates the pattern P_2 is matched.

state	0	5	26	37	**8**
input char		p	c	d	g
pathVec	11	10	11	11	10
preReg	11	10	10	10	**10**
ifFinal	0	0	0	0	1

Fig. 6. State transitions of the input string "pcdg"

The algorithm of our state traversal algorithm is shown in Fig. 7.

```
Algorithm: State traversal pattern matching algorithm
Input: A text string x=a1a2…an where each ai 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, ai) == fail || preReg == 0 do
                    begin
                       state ← f (state)
                       preReg ← 1….1
                    end
             state ← g(state, ai)
             if ifFinal(state) = 1 then
               begin
                  print i
                  print preReg
             end
         end
end
```

Fig. 7. State traversal pattern matching algorithm

For Snort rule sets, the new algorithm achieves 21% of memory reduction compared with the traditional Aho-Corasick algorithm. In addition, we can gain 24% of memory reduction by integrating our approach to the bit-split algorithm which is the state-of-the-art DFA-based approach.

2. Hierarchical State Machine Architecture for Regular Expression Pattern Matching

In the second part of this project, we propose a hierarchical state machine architecture composed of two state machines, called master and slave, to match the second type of complex regular expression patterns. The master state machine is used to match against the segmented sub-patterns while the slave state machine is used to identify the correlation of the segmented sub-patterns. Our basic idea is that if we can use two concurrent state machines, where one state machine is for matching the segmented sub-patterns, while the other state machine is for

identifying the correlation of the segmented sub-patterns, then we can efficiently reduce the additional states. The reason is that the master state machine is in charge of matching segmented sub-patterns, compiling *m* patterns of length n_i (*i*=0…*m*) needs O($(\sum_{i=0}^{m} n_i)$) states. On the other hand, because the slave is in charge of identifying the correlation of the segmented sub-patterns, compiling *m* patterns with one ".*" in each pattern needs O(2^m) states. Therefore, for compiling *m* patterns of length n_i (*i*=0…*m*) with one ".*" in each pattern, the total space is O($(\sum_{i=0}^{m} n_i) + 2^m$) states. Compared to traditional DFA, the space complexity is reduced from O($(\sum_{i=0}^{m} n_i) \cdot 2^m$) to O($(\sum_{i=0}^{m} n_i) + 2^m$) states. Furthermore, for compiling *m* patterns of length n_i (*i*=0…*m*) with *x* instances of ".*" in each pattern, the space is reduced from O($(\sum_{i=0}^{m} n_i) \cdot (m+x)^m$) to O($(\sum_{i=0}^{m} n_i) + (m+x)^m$) states.

For *m* regular expressions, ".*R_1.*$R_{1'}$", ".*R_2.*$R_{2'}$",…, and ".*R_m.*$R_{m'}$", Figure 8 shows the hierarchical state machine architecture which are composed of two parallel state machines, *master* and *slave* state machines. The master state machine is used to match input streams to the segmented sub-patterns, such as R_1, $R_{1'}$, R_2, $R_{2'}$,…R_m and $R_{m'}$. Then, the matching results, α, β, γ, δ,…, ε, and θ of the master state machine, called *inter-result* is fed to the slave state machine which is used to identify the correlation of the segmented sub-patterns and to output final results, called *match vector*.

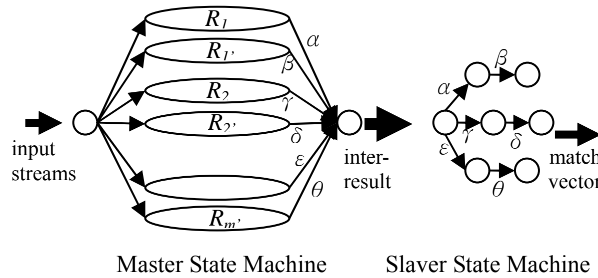
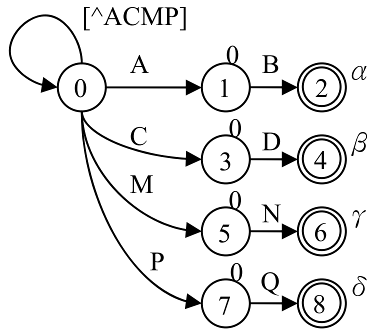
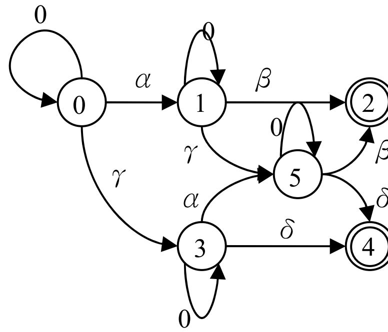


Figure 8: Hierarchical State Machine Architecture

We now illustrate the hierarchical state machine architecture using an example. Considering the following two patterns, ".*AB.*CD" and ".*MN.*PQ", we first construct a master state machine to match input characters to the segmented sub-patterns, "AB", "CD", "MN", and "PQ", as shown in Figure 9(a) where partial edges are omitted for easy description. In Figure 9(a), the outputs, called *inter-results*, of final states 2, 4, 6, and 8 are represented as α, β, γ, and δ, respectively. Except for final states, the outputs are 0. Then, we construct a slave state machine to match the inter-results to the new regular expression ".* α .* β" and ".* γ .* δ" as shown in Figure 9(b).



(a) Master State Machine



(b) Slave State Machine

Figure 9: Master and Slave State Machine

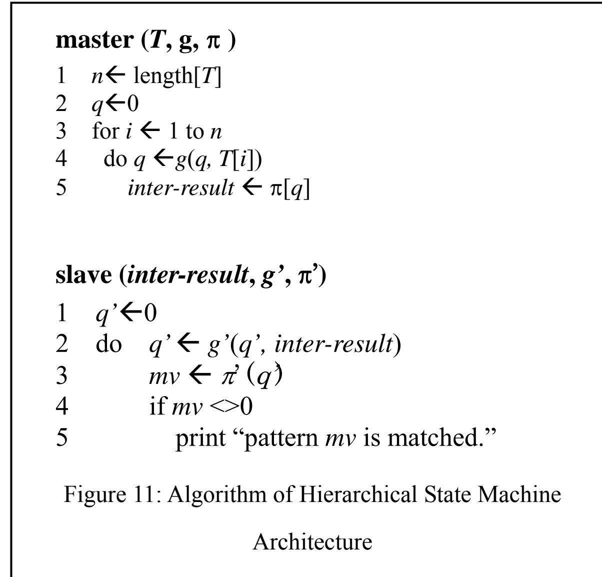
Consider the example in Figure 10 which matches the input string “ABMNCD” to the regular expressions “.*AB.*CD” and “.*MN.*PQ”. First, as the master state machine reads “AB”, we can see that the master state machine moves from state 0 to state 2 and outputs “α”. At the same time, the slave state machine reads the inter-results “00α” and moves from state 0 to state 1. Then, the master state machine reads “MN”, and moves from state 2 to state 6 and outputs “γ”. Concurrently, the slave state machine reads “0γ” and moves from state 1 to state 5. Finally, the master state machine reads “CD”, moves from state 6 to state 4, and outputs “β”. On the other hand, the slave state machine reads “0β”, and moves from state 5 to state 2 which represents the final state of the pattern “.*AB.*CD”.

Input character:		A	B	M	N	C	D	
State of master:	0 →	1 →	2 →	5 →	6 →	3 →	4	
Inter-results:	0	0	α	0	γ	0	β	
State of slave:	0→	0→	0 →	1 →	1 →	5 →	5 →	2

Figure 10: State transitions of matching “ABMNCD”

To clarify the operation of the hierarchical state machine architecture, the algorithm is given in pseudocode below as the procedures master and slave in Figure 11. The overall architecture is used to find the occurrences of multiple complicated regular expression patterns in an input text $T[1...n]$. The g and g' denote the transition functions, the π and π' denote the output functions and the *inter-result* and *mv* denote the matching outputs of master and slave state machines,

respectively.



```
master (T, g, π )
1   n← length[T]
2   q←0
3   for i ← 1 to n
4     do q ←g(q, T[i])
5         inter-result ← π[q]

slave (inter-result, g', π')
1   q'←0
2   do    q' ← g'(q', inter-result)
3         mv ← π'(q')
4         if mv <>0
5             print "pattern mv is matched."
```

Figure 11: Algorithm of Hierarchical State Machine Architecture

The algorithm shows that the computational complexity of the hierarchical state machine architecture to match multiple complicated regular expression patterns on a text string of length n is $O(n)$. In addition, because the hierarchical state machine architecture is based on general memory architecture, it can also be applied to process simple regular expression patterns, such as exact string patterns. Compared to the traditional DFA, the hierarchical state machine architecture can significantly reduce the number of states and therefore the memory size. Experimental results show that the hierarchical state machine architecture achieves an average of 95% of state reduction and 97% of memory reduction for the complicated regular expression patterns in Snort, Bro, and L7-filter systems.

3. Hybrid Memory Architecture for Regular Expression Matching

In the third part of this project, we propose a hybrid memory architecture which inserts marginal logic elements to improve the ability of traditional memory architecture to deal with the first type of complex regular expressions. The hybrid memory architecture allows for multiple complex regular expression patterns to be merged and matched simultaneously without the problem of memory explosion. The overall architecture is shown in Figure 12. The new architecture contains two major elements: the memory element (as in the memory architecture) and the logic element. Inserting logic elements into the memory architecture allows for efficiently dealing with both simple and complex regular expression patterns. Similar to that in the memory architecture, the memory element is used to match simple regular expression patterns, such as explicit string patterns. We can observe that complex regular expressions also contain explicit string patterns. For example, the regular expression ".*CAPA\s[^\n]{10}" contains the explicit string pattern "CAPA\s" in its prefixes, which can be dealt with by the memory element. All the explicit string patterns are converted to a DFA whose corresponding transition table is stored in the memory element. When the memory element matches one of the prefix substrings, its outputs, called prefix-result activates its subsequent logic elements to match suffix constraint repetitions.

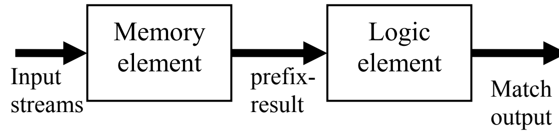


Figure 12: Hybrid Memory Architecture

The logic element is based on the Nondeterministic Finite Automaton (NFA) and employed to deal with suffix constraint repetitions. Each character of a pattern is mapped to a circuit module. Consider the regular expression pattern, ".*CAPA\s[^\n]{10}". We use the logic elements as shown in Figure 13 to match the suffix constraint repetitions, "[^\n]{10}". The text input is broadcasted to each circuit module which is used to match non-line-feed characters (^\n). Ten circuit modules are concatenated to mach the suffix pattern, "[^\n]{10}". As shown in Figure 3, the size of logic elements is proportional to the length of suffix constraint repetitions.

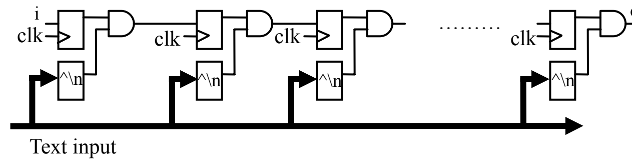


Figure 13: Logic elements for matching "[^\n]{10}"

As shown in figure 14, the hybrid memory architecture composes of memory elements and logic elements. When implementing logic elements, we can share common constraint repetitions among regular expressions to reduce area overhead. For example, the FTP rule set of Snort has 29 patterns with the same constraint repetition, "[^\n]{100}", whose corresponding NFA blocks can be merged. Note that a logic element can be activated several times, but multiple activations can cause some confusion about which regular expression is matched. To determine which regular expression is matched, we add an FIFO to each NFA block to keep track of the matching results. When a regular expression is matched, the FIFO outputs the match vector. Take the two regular expression patterns, ".*PASS[^\n]{100}" and ".*USER[^\n]{100}" as an example. The prefix string patterns "PASS" and "USER" are converted to a DFA as shown in Figure 5. State 4 is the final state of pattern "PASS" whose match vector is one-hot encoded as {01}; and state 8 is the final state of pattern "USER" whose match vector is {10}. The same constraint repetitions, "[^\n]{100}," are merged and translated to a single NFA block. Let us consider an input string containing a substring "PASS" followed by a substring "USER." When the first substring "PASS" is matched by the memory element, its match vector {01} is pushed into the FIFO, and the subsequent NFA block which attempts to match "[^\n]{100}" is activated. When the logic element finishes matching, the output of the FIFO indicates which regular expression is matched.

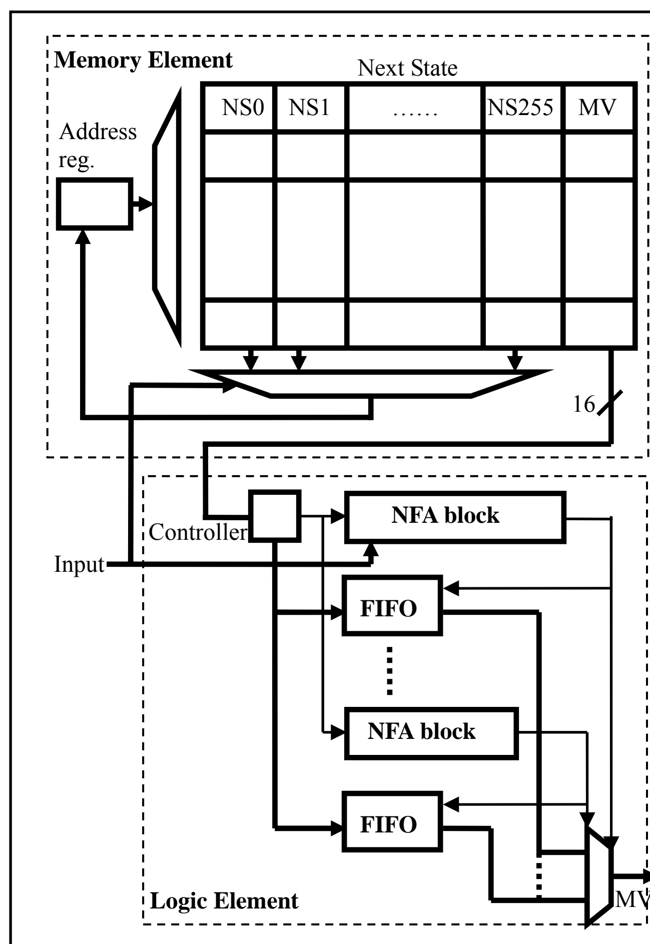


Figure 14: Block Diagram of Hybrid Architecture

The experimental results show the hybrid architecture achieves significant memory reduction for the Snort regular expression patterns. In addition, we also integrate the bit-split algorithm [11] to obtain additional 88% memory reduction.

五、 重要執行成果和價值

In this project, we have proposed three techniques to reduce memory requirement of regular expression patterns. First, we propose an efficient pattern matching algorithm which can reduce memory requirement by sharing common sub-pattern. Second, we propose a hierarchical state machine architecture for matching the complex regular patterns composed of multiple star-closures. Finally, we propose a hybrid memory architecture which integrate logic and memory architecture for complex regular expressions composed of wildcard and constraint repetitions. All techniques have been accepted and published in international conferences and journals. In the following, we list the related papers of above techniques.

[1] Cheng-Hung Lin and Shih-Chieh Chang, "Efficient Pattern Matching Algorithm for Memory Architecture" accepted by IEEE Transactions on Very Large Scale Integration (VLSI) Systems.(SCI)

[2] Cheng-Hung Lin, and Hsien-Sheng Hsiao, "Hierarchical State Machine Architecture for Regular Expression Pattern Matching", Proc. ACM Great Lakes Symposium on VLSI (GLSVLSI), Boston, Massachusetts,USA, May 10-12, 2009, pp.133-136.

[3] Cheng-Hung Lin, "Hybrid Memory Architecture for Regular Expression Matching" Proc. IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2009), Cancun, Mexico, August 2-5, 2009, pp.1159-1162

六、 結論

In this project, we have proposed three techniques to reduce memory requirement of regular expression patterns. First, we propose an efficient pattern matching algorithm which can reduce memory requirement by sharing common sub-pattern. For Snort rule sets, the new algorithm achieves 21% of memory reduction compared with the traditional Aho-Corasick algorithm. In addition, we can gain 24% of memory reduction by integrating our approach to the bit-split algorithm which is the state-of-the-art DFA-based approach.

Second, we propose a hierarchical state machine architecture for matching the complex regular patterns with multiple star-closures. Experimental results show that the hierarchical state machine architecture achieves an average of 95% of state reduction and 97% of memory reduction for the complicated regular expression patterns in Snort, Bro, and L7-filter systems.

Finally, we propose a hybrid memory architecture which integrate logic and memory architecture for complex regular expressions composed of wildcard and constraint repetitions. The experimental results show the hybrid architecture achieves significant memory reduction for the Snort regular expression patterns. In addition, we also integrate the bit-split algorithm to obtain additional 88% memory reduction.

七、 參考文獻

[1] Cheng-Hung Lin and Shih-Chieh Chang, "Efficient Pattern Matching Algorithm for Memory Architecture" *accepted by IEEE Transactions on Very Large Scale Integration (VLSI) Systems.*

[2] Cheng-Hung Lin, and Hsien-Sheng Hsiao, "Hierarchical State Machine Architecture for Regular Expression Pattern Matching", Proc. *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, Boston, Massachusetts,USA, May 10-12, 2009, pp.133-136.

[3] Cheng-Hung Lin, "Hybrid Memory Architecture for Regular Expression Matching" Proc. *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2009)*, Cancun, Mexico, August 2-5, 2009, pp.1159-1162.

[4] Snort official website, “http://www.snort.org/”

[5] Bro official website, “http://www.bro-ids.org/”

[6] ClamAV official website, “http://www.clamav.net/”

[7] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using FPGAs,” in *Proc. 9th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2001, pp. 227-238.

[8] B.L. Hutchings, R. Franklin, and D. Carver, “Assisting Network Intrusion Detection with Reconfigurable Hardware,” in *Proc.10th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM),* 2002, pp. 111-120.

[9] C. R. Clark and D. E. Schimmel, “Scalable Parallel Pattern Matching for High Speed Networks,” in *Proc. 12th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 249-257

[10] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, “Specialized Hardware for Deep Network Packet Filtering,” in *Proc. 10th Ann. ACM/SIGDA Int. Conf. Field-Program. Logic Appl. (FPL),* 2002, pp. 452-461

[11] Y. H. Cho and W. H. Mangione-Smith, “A Pattern Matching Co-processor for Network Security,” in *Proc. 42nd Des. Autom. Conf. (DAC),* 2005, pp. 234-239

[12] M. Aldwairi*, T. Conte, and P. Franzon, “Configurable String Matching Hardware for Speeding up Intrusion Detection,” in *ACM SIGARCH Computer Architecture News*, 2005, pp. 99–107

[13] S. Dharmapurikar and J. Lockwood, “Fast and Scalable Pattern Matching for Content Filtering,” in *Proc. of Symp. Architectures Netw. Commun. Syst. (ANCS)*, 2005, pp. 183-192

[14] L. Tan and T. Sherwood, “A high throughput string matching architecture for intrusion detection and prevention,” in *32nd Ann. Int. Symp. on Comp. Architecture, (ISCA)*, 2005, pp. 112-122

[15] H. J. Jung, Z. K. Baker, and V. K. Prasanna, “Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems,” in *20th Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2006.

[16] F. Yu, Z. Chen, Y.Diao, T.V. Lakshman, and R.H. Katz, “Fast and Memory-Efficient Regular Expression Matching for Deep packet Inspection,” in *Proc. ACM/IEEE Symp. Architectures Netw. Commun. Syst. (ANCS),* 2006, pp. 93-102

[17] Z. K. Baker, H. J. Jung, and V. K. Prasanna, “Regular Expression Software Deceleration for Intrusion Detection systems,” in *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, 2006, pp. 1-8.

[18] 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.

[19] I. Sourdis, J.Bispo, J. M.P. Cardoso, and S. Vassiliadis, “Regular expression matching for reconfigurable packet inspection”, in *Proc. IEEE International conference on Field Programmable Technology (FPT)*, 2006, pp.119-126.

[20] B. Brodie, R. Cytron, D.Taylor, “A Scalable Architecture for High-throughput Regular Expression Matching,” in *Proc. 33rd Int’l Symposium on Computer Architecture (ISCA)*, 2006, pp191-202.

[21] S. Kumar, S.Dharmapurikar, F.Yu, P. Crowley, and J. Turner, “Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection,” in *ACM SIGCOMM Computer Communication Review*, ACM Press, vol.36, Issue. 4, Oct. 2006, pp. 339-350.

[22] J. Moscola, Y. H. Cho, J. W. Lockwood, “A Scalable hybrid regular expression pattern matcher,” in *Proc. 14th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2006, pp.337-338

出席國際學術會議心得報告

計畫名稱：正規表示法樣式比對之演算法與硬體架構設計

計畫編號：97-2218-E-003-002-

報告人：林政宏副教授

會議名稱：IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2009)

會議地點：Cancun, Mexico

會議時間：August 2-5, 2009

出國目的：口頭發表論文

論文名稱：Hybrid Memory Architecture for Regular Expression Matching

與會心得：

It's a really good experience to attend the the 52nd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2009) held on August 2-5, 2009, in Cancún, México. The Midwest Symposium on Circuits and Systems (MWSCAS) is the oldest Circuits and Systems symposia sponsored by the IEEE. My paper titled "Hybrid Memory Architecture for Regular Expression Matching" was accepted as oral presentation. Because many attendances work on relative researches, I obtained a lot of positive feedback. After my presentation, many professors and students discussed with me about my researches. In addition, I also listened several interesting research topics including health care system, analog IC design tread, and so on.

Finally, I am very thankful for NSC supporting me to attend this conference.

Hybrid Memory Architecture for Regular Expression Matching

Cheng-Hung Lin
National Taiwan Normal University, Taipei, Taiwan
brucelin@ntnu.edu.tw

Abstract

Regular expression matching has been widely used in Network Intrusion Detection Systems due to its strong expressive power and flexibility. To match multiple regular expression patterns simultaneously, compiling multiple regular expressions into a composite DFA is an intuitive and efficient way. However, traditional memory architecture suffers from the problem of memory explosion for certain complex regular expressions. In this paper, we propose a hybrid memory architecture which inserts marginal logic elements to improve the ability of traditional memory architecture to deal with complex regular expressions. The experimental results demonstrate a significant memory reduction for complex regular expressions.

1. Introduction

The main purpose of a network intrusion detection system (NIDS) is to inspect the packet header and payload against thousands of predefined malicious or suspicious patterns. These patterns describe behaviors such as denial of service attacks, port scans, or malware. To efficiently represent suspicious patterns, regular expressions are commonly adopted such as Snort[1], Bro[2], and ClamAV[3] because they have better expressive power and flexibility than explicit string patterns.

The key function of an NIDS is to perform the matching of regular expressions representing the suspicious patterns. Because software approaches of matching regular expressions can no longer meet the high throughput of today's networking, many hardware approaches are being proposed. Basically, these hardware approaches can be classified into two categories, logic [4][5][6][7] and memory [8][9][10][11][12] architectures. The logic architectures are mostly based on *Nondeterministic Finite Automaton* (NFA) and implemented on FPGA. On the other hand, the memory architectures are based on *Deterministic Finite Automaton* (DFA). The memory architectures have attracted a lot of attention because it is well suited for synthesis in an ASIC and has the advantages of easy re-configurability and scalability [14][17][18].

The basic memory architecture has two steps. First, regular expression patterns are compiled into a DFA. Then, the corresponding state transition table of the DFA is stored in memory for pattern matching. It's well known that when m patterns are compiled into a composite DFA, the matching time complexity for matching m patterns against the input texts of length n is reduced from $O(mn)$ to $O(n)$. Therefore, compiling multiple regular expression patterns into a single composite DFA is an intuitive and efficient way to match multiple regular expression patterns simultaneously. However, there exists specific regular expression patterns which cause the composite DFA to grow exponentially. For example, the regular expression pattern for identifying the POP3 CAPA overflow attempt, ".*CAPA\s[^\n]{10}", and the pattern for identifying the POP3 TOP overflow attempt, ".*TOP\s[^\n]{10}" contain 112 and 137 states, respectively. Compiling these two patterns into a composite DFA will cause the number of state exponentially increasing to 649. Such complex regular expressions can be divided into two segments, prefixes and suffixes. For example, the sub-pattern "CAPA\s" and "TOP\s" are referred to prefixes while the sub-pattern "[^\n]{10}" are referred to suffixes. The state explosion comes from the *overlapping* of prefixes and suffixes in such complex regular expressions. The overlapping denotes the characteristic that any substring matches the prefixes, "CAPA\s" and "TOP\s" also matches the suffixes "[^\n]{10}". With the increasing number of such complex patterns, the corresponding state table, and therefore the memory size will blows up.

In this paper, we propose a hybrid memory architecture which inserts marginal logic elements to improve the ability of traditional memory architecture to deal with the complex regular expression patterns. The hybrid memory architecture allows for multiple complex regular expression patterns to be merged and matched simultaneously without the problem of memory explosion. The experimental results show the hybrid architecture achieves significant memory reduction for the Snort regular expression patterns. In addition, we also integrate the bit-split algorithm [11] to obtain additional 88% memory reduction. In section 2, we give an analysis of the complex regular expressions commonly used in NIDS. In section 3, we present the hybrid memory architecture. In sections 4 and 5, we present the experimental results and our conclusions, respectively.

2. Complex Regular Expression Patterns

In this paper, the complex regular expression pattern denotes the regular expression patterns with the characteristic of overlapping, such as the regular expression pattern for the identifying POP3 CAPA overflow attempt, ".*CAPA\s[^\n]{10}", and the pattern for identifying POP3 TOP overflow attempt, ".*TOP\s[^\n]{10}". As described in the introduction, when compiling multiple complex regular expression patterns into a composite DFA, the overlapping characteristic causes the state machine to grow exponentially. In this section, we discuss the reason why the overlapping characteristic causes the composite DFA to grow exponentially.

First, let us consider the following two regular expression patterns, ".*CAPA\s[^\n]{10}" and ".*TOP\s[^\n]{10}" . The pattern ".*CAPA\s[^\n]{10}" means that whenever an input substring contains "CAPA\s", followed by ten of non-line-feed characters (^\n), the regular expression pattern ".*CAPA\s[^\n]{10}" is matched. The pattern ".*TOP\s[^\n]{10}" has the same behavior. The state explosion comes from the fact that the prefixes "CAPA\s" and "TOP\s" overlap with the suffixes "[^\n]{10}". Since any substring containing "CAPA\s" or "TOP\s" simultaneously matches prefixes and suffixes of these two patterns, the corresponding DFA needs much more states to keep track of all possible occurrences of the substrings "CAPA\s" or "TOP\s". For example, consider the composite NFA of these two regular expression patterns in Figure 1. If the input contains a substring "CAPA\sC", the NFA will activate states 1 and 6 simultaneously. Similarly, if the input string contains a sub-string "CAPA\sTO", the NFA will activate states 7 and 17 simultaneously. We can observe that the overlapping occurs not only in each pattern, but also occurs in each other. Therefore, when converting the NFA into a DFA, the state number of the DFA will grow exponentially.

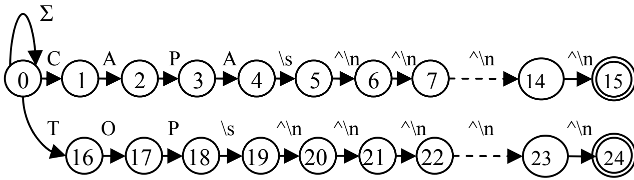


Figure 1: NFA of ".*CAPA\s[^\n]{10}" and ".*TOP\s[^\n]{10}"

3. Hybrid Memory Architecture

In this section, we propose a hybrid memory architecture to improve the ability of the traditional memory architecture to deal with these memory-intensive regular expressions. Because the overlapping characteristic of complex regular expression patterns causes the corresponding DFA having exponential number of states, our basic idea is to split the complex regular expression into two segments, the prefix exact string and the suffix constraint repetitions, and then, insert marginal logic elements into the memory architecture to handle the suffix constraint repetitions, so as to mitigate the memory explosion caused by the overlapping characteristic.

The overall architecture is shown in Figure 2. The new architecture contains two major elements: the memory element (as in the memory architecture) and the logic element. Inserting logic elements into the memory architecture allows for efficiently dealing with both simple and complex regular expression patterns. Similar to that in the memory architecture, the memory element is used to match simple regular expression patterns, such as explicit string patterns. We can observe that complex regular expressions also contain explicit string patterns. For example, the regular expression ".*CAPA\s[^\n]{10}" contains the explicit string pattern "CAPA\s" in its prefixes, which can be dealt with by the memory element. All the explicit string patterns are converted to a DFA whose corresponding transition table is stored in the memory element. When the memory element matches one of the prefix substrings, its outputs, called *prefix-result* activates its subsequent logic elements to match suffix constraint repetitions.

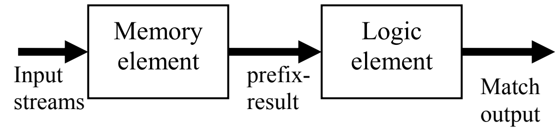


Figure 2: Hybrid Memory Architecture

The logic element is based on the Nondeterministic Finite Automaton (NFA) and employed to deal with suffix constraint repetitions. Each character of a pattern is mapped to a circuit module. Consider the regular expression pattern, ".*CAPA\s[^\n]{10}". We use the logic elements as shown in Figure 3 to match the suffix constraint repetitions, "[^\n]{10}". The text input is broadcasted to each circuit module which is used to match non-line-feed characters (^\n). Ten circuit modules are concatenated to mach the suffix pattern, "[^\n]{10}". As shown in Figure 3, the size of logic elements is proportional to the length of suffix constraint repetitions.

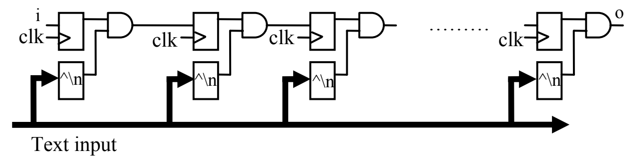


Figure 3: Logic elements for matching "[^\n]{10}"

As shown in figure 4, the hybrid memory architecture composes of memory elements and logic elements. When implementing logic elements, we can share common constraint repetitions among regular expressions to reduce area overhead. For example, the *FTP* rule set of Snort has

29 patterns with the same constraint repetition, “[^\n]{100}”, whose corresponding NFA blocks can be merged. Note that a logic element can be activated several times, but multiple activations can cause some confusion about which regular expression is matched. To determine which regular expression is matched, we add an FIFO to each NFA block to keep track of the matching results. When a regular expression is matched, the FIFO outputs the match vector. Take the two regular expression patterns, “.*PASS[^\n]{100}” and “.*USER[^\n]{100}” as an example. The prefix string patterns “PASS” and “USER” are converted to a DFA as shown in Figure 5. State 4 is the final state of pattern “PASS” whose match vector is one-hot encoded as {01}; and state 8 is the final state of pattern “USER” whose match vector is {10}. The same constraint repetitions, “[^\n]{100},” are merged and translated to a single NFA block. Let us consider an input string containing a substring “PASS” followed by a substring “USER.” When the first substring “PASS” is matched by the memory element, its match vector {01} is pushed into the FIFO, and the subsequent NFA block which attempts to match “[^\n]{100}” is activated. When the logic element finishes matching, the output of the FIFO indicates which regular expression is matched.

4. Experimental results

We have implemented the hybrid memory architecture and tested it on the seven largest regular expression rule sets in the Snort (V. 2.4) database. We compared our results with the original memory architecture and the more recent bit-split algorithm [11]. First, we obtain regular expression patterns from the Snort rule database. Then, we group patterns based on their similarity, including the type of wildcard in the beginning of the pattern and constraint repetitions. Furthermore, we partition the grouped regular expressions into two segments, prefix sub-pattern and suffix constraint repetitions. Finally, the DFA for prefix sub-patterns and the logic elements for suffix constraint repetitions are constructed. In addition, because the state-of-art bit-split algorithm [11] can further reduce the memory size of a DFA, we also apply the technique to our memory element to obtain additional memory improvement.

Table 1 shows the results of our approach compared with the original memory architecture. Columns one and two show the name of rule set and the number of patterns. Columns three and four show the number of states and the memory requirements of the original memory architecture. Columns five and six show the number of states and the memory requirements of the hybrid memory architecture. Columns seven and eight show the number of states and the memory requirements of the hybrid memory architecture integrating with the bit-split algorithm. Columns nine and ten show the size of the NFA block and FIFO. For example in the first row of Table 1, the FTP rule set has 49 regular expression patterns. The original memory architecture has 2,185 states and requires more than 1M bytes of memory. In contrast, the hybrid memory architecture has 199 states and requires 52K bytes of memory. The results show that the hybrid architecture significantly reduces the size of memory. Besides, by splitting the original DFA into four smaller DFAs, the total number of states is 685 states and the memory requirement is reduced to 8K bytes. The experimental results show that the hybrid architecture with the bit-split algorithm significantly reduces the size of memory caused by the complex regular expressions with small logic overheads.

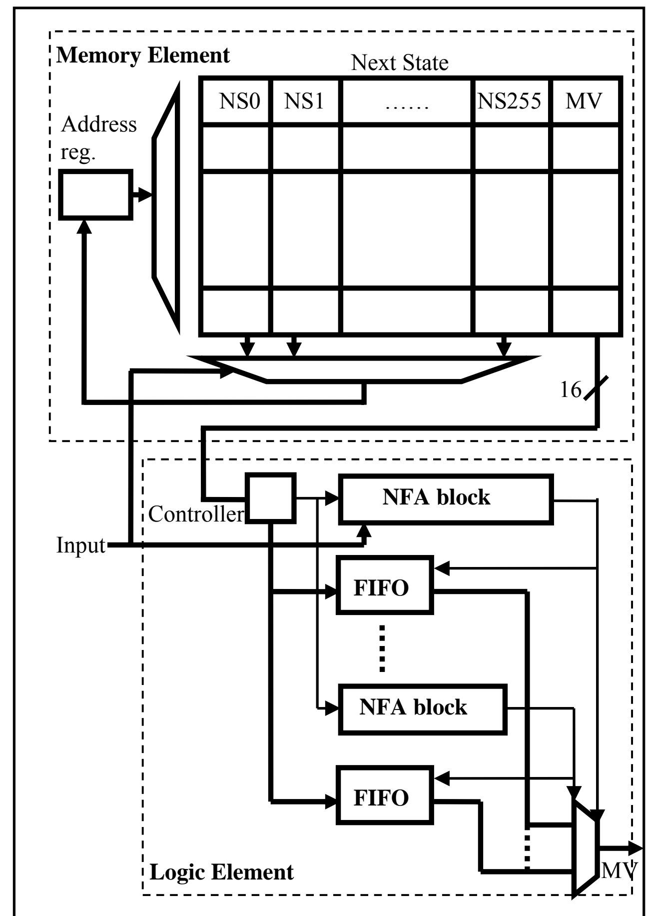


Figure 4: Block Diagram of Hybrid Architecture

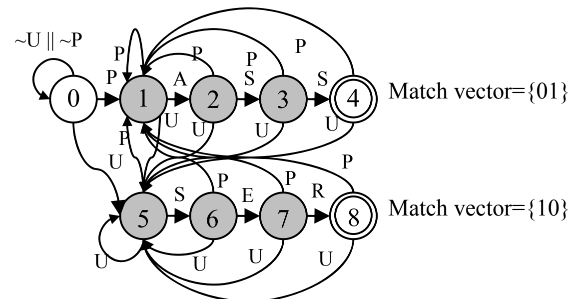


Figure 5: DFA for matching “PASS” and “USER”

5. Conclusions

We have presented a novel memory architecture which can efficiently deal with complex regular expressions without consuming a large amount of memory. In addition, we also integrate the bit-split algorithm to our memory architecture and obtain additional reductions in memory requirements. The experiments demonstrate a significant reduction in memory requirements for complex regular expressions.

Table 1: Experimental results on Snort rule sets

Rule set	# of pattern	Original memory architecture		Hybrid memory architecture		Hybrid memory architecture with bit-split algorithm		Logic element	
		# of states	memory (bytes)	# of states	memory (bytes)	# of states	memory (bytes)	NFA block (# of FF)	FIFO (bytes)
FTP	49	3,099	1M	199	52K	685	8K	100	363
EXPLOIT	16	254,293	147M	142	36K	486	3.5K	1,350	138
IMAP	34	3,195,536	2,263M	44	8K	144	1.3K	1,380	552
NNTP	13	2,000,527	1,347M	108	24K	364	2.2K	1,309	277
POP3	16	848	273K	76	17K	258	1.7K	316	100
WEB-CLIENT	13	816	262K	16	2K	58	232	400	100
WEB-MISC	17	196,446	113M	80	18K	275	1.8K	1,572	128
AVG.					1		12%		

Reference

[1] Snort official website, "http://www.snort.org/"

[2] Bro official website, "http://www.bro-ids.org/"

[3] ClamAV official website, "http://www.clamav.net/"

[4] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. 9th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2001, pp. 227-238.

[5] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proc.10th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM),* 2002, pp. 111-120.

[6] C. R. Clark and D. E. Schimmel, "Scalable Parallel Pattern Matching for High Speed Networks," in *Proc. 12th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 249-257

[7] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," in *Proc. 10th Ann. ACM/SIGDA Int. Conf. Field-Program. Logic Appl. (FPL),* 2002, pp. 452-461

[8] Y. H. Cho and W. H. Mangione-Smith, "A Pattern Matching Co-processor for Network Security," in *Proc. 42nd Des. Autom. Conf. (DAC),* 2005, pp. 234-239

[9] M. Aldwairi*, T. Conte, and P. Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," in *ACM SIGARCH Computer Architecture News*, 2005, pp. 99–107

[10] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Content Filtering," in *Proc. of Symp. Architectures Netw. Commun. Syst. (ANCS)*, 2005, pp. 183-192

[11] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *32nd Ann. Int. Symp. on Comp. Architecture, (ISCA)*, 2005, pp. 112-122

[12] H. J. Jung, Z. K. Baker, and V. K. Prasanna, "Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems," in *20th Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2006.

[13] F. Yu, Z. Chen, Y.Diao, T.V. Lakshman, and R.H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep packet Inspection," in *Proc. ACM/IEEE Symp. Architectures Netw. Commun. Syst. (ANCS),* 2006, pp. 93-102

[14] Z. K. Baker, H. J. Jung, and V. K. Prasanna, "Regular Expression Software Deceleration for Intrusion Detection systems," in *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, 2006, pp. 1-8.

[15] 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.

[16] I. Sourdis, J.Bispo, J. M.P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection", in *Proc. IEEE International conference on Field Programmable Technology (FPT)*, 2006, pp.119-126.

[17] B. Brodie, R. Cytron, D.Taylor, "A Scalable Architecture for High-throughput Regular Expression Matching," in *Proc. 33rd Int'l Symposium on Computer Architecture (ISCA)*, 2006, pp191-202.

[18] S. Kumar, S.Dharmapurikar, F.Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in *ACM SIGCOMM Computer Communication Review*, ACM Press, vol.36, Issue. 4, Oct. 2006, pp. 339-350.

[19] J. Moscola, Y. H. Cho, J. W. Lockwood, "A Scalable hybrid regular expression pattern matcher," in *Proc. 14th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2006, pp.337-338