Tối ưu không gian trạng thái của thuật toán Aho-Corasick sử dụng kỹ thuật nén dòng và bảng chỉ số

Optimizing State Space of Aho-Corasick Algorithm using Compressed State Storage and Index Table Techniques

Lê Đắc Nhường, Lê Đăng Nguyên và Lê Trọng Vĩnh

Abstract: *The pattern matching algorithms are important roles in most applications of information technology. For example, Network Intrusion Detection System looks for evidence of malicious behavior based on matching packet contents with known patterns. Therefore, the study of the pattern-matching algorithm is a hot topic many researchers are interested. In this paper, we propose a new method to optimize the state storage of the pattern matching algorithms, Aho-Corasick by using compressed row and index table techniques. The experimental results that compare the efficacy performed between the original Aho-Corasick algorithm and the improved algorithm installed in Snort showed that our method achieved better results.*

Keywords: *Pattern Matching; Aho-Corasick; Compressed Row Storage; Index table.*

I. ĐẶT VẤN ĐỀ

Bài toán so khớp mẫu được mô tả như sau: Cho một bảng chữ cái *S*, một mẫu *P* (*P[1..m]*) độ dài *m* và một chuỗi ký tự *T* (*T[1..n]*) độ dài *n* (trong đó *m*<<*n*). Bài toán đặt ra là cần tìm các vị trí xuất hiện của *P* trong *T* hoặc *P* có khớp với một chuỗi con của *T* hay không? Trong [2], Christian Charras và Therry Lecroq đã giới thiệu hơn 35 thuật toán so khớp khác nhau. Các thuật toán so khớp đều sử dụng cơ chế *cửa sổ trượt* để so sánh các ký tự của mẫu trong cửa sổ với các ký tự trong văn bản. Ứng dụng của thuật toán so khớp mẫu trong an ninh mạng được các tác giả trong [3] phân tích và đánh giá theo hai tiêu chí: số lượng mẫu (so khớp đơn mẫu, so khớp đa mẫu), cơ sở thiết kế thuật toán (so khớp dựa trên tiền tố, so khớp dựa trên hậu tố và so khớp thừa số) dựa vào thời gian và bộ nhớ.

Ngày nay, việc so khớp mẫu trở nên khó khăn hơn nhiều do sự gia tăng về số lượng các mẫu. Chẳng hạn, trong an ninh mạng số đợt tấn công, hình thức tấn công, các loại virus, spam, trojan… không chỉ tăng nhanh về số lượng mà cả sự đa dạng của nó. Vì vậy, việc thu thập đầy đủ các mẫu làm cho kích thước tập mẫu ngày càng tăng nhanh. Do vậy việc tối ưu bộ nhớ thực thi của các thuật toán so khớp mẫu đang rất được quan tâm bởi lẽ nó quyết định tốc độ và hiệu năng của hệ thống.

Đã có rất nhiều phương pháp được đề xuất trong việc quản lý bộ nhớ và truy cập bộ nhớ thông qua các bảng cú pháp như của Johnson [4], nén nội dung bảng dùng tập đa vectơ của Yao [5], Yacc và Lex đã cải tiến cách tìm kiếm dựa trên các biểu thức chính qui. Các phương pháp sử dụng bảng băm, cây, ánh xạ địa chỉ được trình bày bởi các tác giả trong [6-7]. Mars A.Nortoon và các cộng sự đã đề xuất phương pháp tìm kiếm đa mẫu trong [8-9]. Branimir Z. Lambov [10] đã trình bày phương pháp điều khiển không gian lưu trữ trạng thái sử dụng định dạng dòng thay thế (*Row-Displaced Format*). Tất cả các phương pháp trên được đưa ra đều nhằm mục đích tối thiểu hóa cách thức biểu diễn bộ nhớ trong các bảng tra trạng thái so với cách thức biểu diễn đầy đủ dựa trên ma trận.

Trong bài báo này, chúng tôi sẽ tiếp cận theo hướng mới để tối ưu không gian bộ nhớ biểu diễn trạng thái của thuật toán Aho-Corasick (AC) bằng kỹ thuật nén dòng và bảng chỉ số. Các kết quả thực nghiệm so sánh hiệu quả thực thi giữa thuật toán AC

gốc và thuật toán cải tiến cài đặt trên hệ thống phát hiện xâm nhập mạng Snort [9] đã chỉ ra phương pháp của chúng đạt hiệu quả cao hơn. Phần còn lại của bài báo sẽ được tổ chức như sau: Mục II trình bày cách biểu diễn không gian trạng thái của thuật toán AC và các cải tiến bằng kỹ thuật nén dòng và bảng chỉ số. Các kết quả thực nghiệm, phân tích và đánh giá sẽ được trình bày trong mục III. Cuối cùng là kết luận và hướng phát triển.

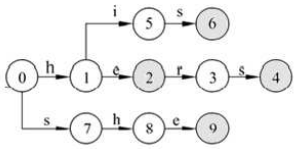
II. TỐI ƯU KHÔNG GIAN TRẠNG THÁI CỦA AC BẰNG KỸ THUẬT NÉN DÒNG VÀ BẢNG CHỈ SỐ

Thuật toán AC [11] là cải tiến của KMP [2] dựa trên tiếp cận tiền tố từ tập đơn mẫu cho tập đa mẫu sử dụng mô hình ôtômat với sự kết hợp của 3 hàm *Goto*, *Failure* và *Output*. AC có thể sử dụng DFA (*Deterministic Finite Automata*) hay NFA (*Non-DFA*). Khai báo ôtômat trong Snort được mô tả bằng tập (*S, Q, I, T, F*) với *S* là bảng chữ cái, *Q* là tập hữu hạn trạng thái, *I* là trạng thái bắt đầu, *T* là tập các trạng thái kết thúc, *F* là hàm dịch chuyển. Việc dịch chuyển trạng thái từ trạng thái hiện tại sang trạng thái mới phụ thuộc vào ký tự đầu vào, chúng ta hoàn toàn có thể xây dựng DFA từ NFA.

II.1 Biểu diễn không gian lưu trữ và tối ưu hóa bằng kỹ thuật nén dòng

Mô hình NFA của AC biểu diễn tập mẫu *P*={*hers, she, his, he*} được cho trong Hình 1.

Rõ ràng, khi tập mẫu lớn với nhiều mẫu thì số trạng thái sẽ tăng lên rất nhanh. Thêm nữa, chúng ta thấy trong ma trận trạng thái của NFA có rất nhiều thành phần có giá trị 0. Do vậy, ý tưởng chính của chúng tôi là sử dụng các kỹ thuật nén dòng (*CSR: Compressed Row Storage*) [10] để giảm bớt không gian lưu trữ. Với tập mẫu *P* ở trên, không gian trạng thái của AC sau khi sử dụng kỹ thuật CSR được trình bày trong Hình 2.



(a) Hàm Goto

i	1	2	3	4	5	6	7	8	9
f(i)	0	0	0	7	0	7	0	1	2

(b) Hàm Failure

i	2	9	6	4
Output(i)	he	she, he	his	hers

(c) Hàm Output

Trạng thái	**Đầu vào**				
	e	**h**	**i**	**r**	**s**
0	0	1	0	0	7
1	2	0	5	0	0
2	0	0	0	3	0
3	0	0	0	0	4
5	0	0	0	0	6
7	0	8	0	0	0
8	9	0	0	0	0

(d) Ma trận chuyển cho hàm Goto

Hình 1. Không gian trạng thái của AC với tập mẫu *P*

Giá trị	1	7	2	5	3	4	6	8	9
Cột	2	5	1	3	4	5	5	2	1
Dòng	1	1	2	2	3	4	5	6	7

Hình 2. Nén ma trận chuyển hàm Goto với CSR

Không gian lưu trữ hàm failure bằng vectơ một chiều trong Hình 1b cũng có số thành phần bằng 0 rất lớn. Để tối ưu không gian lưu trữ, chúng tôi sử dụng kỹ thuật bảng chỉ số lưu lại các dòng thưa bằng con trỏ vị trí có giá trị khác 0 như Hình 3 với phần tử đầu tiên là số lượng các mục, tiếp theo lần lượt là chỉ số và giá trị các mục tương ứng.

Giá trị	7	7	1	2
Chỉ số	4	6	8	9

(a) Nén hàm Failure dùng bảng chỉ số 2 chiều

8	4	7	6	7	8	1	9	2

(b) Nén hàm Failure dùng bảng chỉ số 1 chiều

Số lượng mục	8							
Chỉ số bắt đầu	4							
Giá trị	7	0	7	0	1	2		
Dải lưu trữ	8	4	7	0	7	0	1	2

(c) Nén hàm Failure dùng bảng chỉ số lưu trữ

Hình 3. Nén hàm failure của AC dùng bảng chỉ số

Chúng ta hoàn toàn có thể áp dụng cách lưu trữ này cho mỗi dòng trong ma trận để tối ưu không gian trạng thái, chúng ta biểu diễn bảng dịch chuyển bằng danh sách các con trỏ vectơ trạng thái.

II.2 Cải tiến giai đoạn tiền xử lý của AC

Do DFA yêu cầu xác định duy nhất trạng thái tiếp theo nào ứng với mỗi ký tự vào nên cho hiệu quả thực thi tốt hơn nên trong Snort chúng tôi khai báo thuật toán AC sử dụng DFA. Mô tả cấu trúc dữ liệu danh sách liên kết của DFA như sau:

```
struct DFA {
        struct State *NextState[256];
        struct State *Failurelist;
        struct Matched *Matchlist; }
```

Với mỗi trạng thái chúng ta có tập các con trỏ trỏ đến trạng thái tiếp theo (`*NextState`) với số lượng con trỏ tối đa là 256 ứng với tập ký tự vào 256 ký tự trong bảng mã. Tập các trạng thái lỗi được mô tả bằng danh sách con trỏ `*Failure` và tập các trạng thái kết thúc được mô tả bằng danh sách con trỏ các mẫu được khớp `*Matchlist`. Như vậy với mỗi trạng thái của DFA có thể chứa 256 trạng thái tiếp theo ứng với ký tự vào.

Thuật toán 1. *Xây dựng ma trận chuyển trạng thái dựa trên DFA ứng với tập mẫu P sử dụng cấu trúc bảng chỉ số ký tự.*

INPUT: Tập mẫu P có n mẫu.
OUTPUT: Bảng chuyển trạng thái của DFA chấp nhận tất cả các mẫu trong P.

```
DFA dfaBuild(Patterns P[], int n){
DFA dfa = new DFA();
String w;
State *state, *NextState;
//Cấp phát trạng thái mới
state = dfa.newState();
//Thiết lập trạng thái ban đầu
dfa.setStartState(state);
for (int i = 0; i < n; i++){
     w = P[i];       //Xử lý từng mẫu P[i]
     state = dfa.getStartState();
     for (int j = 0; j < w.length(); i++){
     *NextState   =   dfa.getTransition(*state,
w(j))
     if (!NextState.isValid()){

       // Loại bỏ các trạng thái 0
       *NextState = dfa.newState();
       //Cấp phát trạng thái mới
       // Thêm liên kết chuyển trạng thái các
nút với ký tự vào w(j)
       dfa.addTransition(*state,w(j),*NextState)
;}
       *state = *NextState; }
       dfa.addMatchlist(*state);}
return dfa;}
```

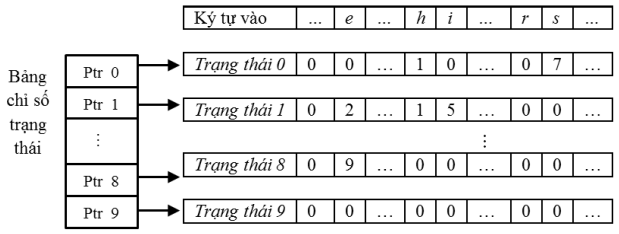
Trong đó, các hàm `addMatchlist`, `addTransition` có nhiệm vụ thêm các trạng thái mới vào danh sách `NextState` và `Matchlist` ứng với ký tự đang xét $w(j)$ của mẫu $P[i]$.

Thuật toán 2. *Xây dựng bảng chỉ số con trỏ failure ứng với DFA*

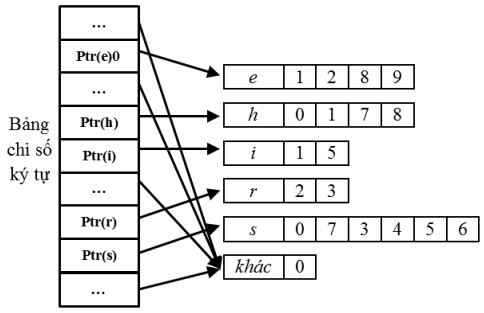
INPUT: DFA ứng với tập mẫu P được xây dựng trong thuật toán 1.
OUTPUT: Danh sách các trạng thái lỗi lưu trữ trong *failure.

```
Void FailureBuild(DFA dfa){
*dfa.Failurelist = new Failure();
Queue q = new Queue();
State state, nextState, s; char ch;
q.add(dfa.getStartState());
*dfa.Failurelist.setFailure(dfa.getStartState(), null);
while (! q.isEmpty()) {
    state = q.remove();
    for (int i = 0; i < 256; i++) {
        ch = *dfa.Nextstate(i);
        nextState=dfa.getTransition(state, ch);
        if (nextState.isValid()){
        s = *dfa.Failurelist.getFailure(state);
        while((s!=null)&&!dfa.getTransition(s,ch).i
              sValid()){
              s = *dfa.Failurelist.getFailure(s);}
              if (s ! = null)
    {*dfa.Failurelist.setFailure(nextState,
                dfa.getTransition(s,ch));}
            else
    {*dfa.Failurelist.setFailure(nextState,
                dfa.getStartState());}
            if
    (f.getFailure(nextState).isMatchlist()){
               dfa.addMatchlist(nextState);}
            q.add(nextState);}}}}
```

Không gian lưu trữ của thuật toán AC gốc cho tập mẫu *P* được minh họa trong hình 4a với số lượng các phần tử được lưu trữ trong mỗi trạng thái bằng với độ dài các ký tự trong gói tin cần kiểm soát và phụ thuộc vào số trạng thái. Chính điều này dẫn đến kích thước không gian lưu trữ trong thuật toán AC gốc rất lớn khi kích thước tập mẫu và kích thước gói tin tăng.



a) Không gian lữu trữ của thuật toán AC gốc



b) Không gian lữu trữ của thuật toán AC khi tối ưu

Hình 4- Không gian trạng thái của AC trước và sau tối ưu

Với giải pháp tối ưu cách biểu diễn không gian trạng thái trong mục II.1, chúng ta đã loại bỏ được các trạng thái rỗng trong không gian lưu trữ khi thực hiện các thuật toán 1 và 2. Thay vì phụ thuộc vào độ dài gói tin, bây giờ không gian lưu trữ thực thi chỉ phụ thuộc vào số lượng các ký tự khác nhau xuất hiện trong tập mẫu do đó độc lập hoàn toàn với kích thước gói tin được kiểm soát khi thực thi (xem trong hình 4b)

II.3 Cải tiến giai đoạn tìm kiếm của thuật toán AC

Trong thuật toán AC gốc, quá trình tìm kiếm mẫu được thực hiện bằng cách sử dụng bảng chỉ số trạng thái. Việc xác định các trạng thái tiếp theo phụ thuộc vào trạng thái hiện thời và ký tự vào trong bảng trạng thái. Sau đây là thuật toán tìm kiếm cải tiến dựa trên bảng chỉ số ký tự, trạng thái tiếp theo được xác định bằng bảng chỉ số các ký tự.

Thuật toán 3. *Tìm kiếm mẫu trên bảng chỉ số ký tự*

```
INPUT: Tập mẫu P có n mẫu. Gói tin cần kiểm soát M.
OUTPUT: Danh sách các mẫu trong P xuất hiện trong nội dung gói tin M
Results Search(Patterns P, Message M){
DFA dfa = dfaBuild(P, n);
FailureBuild(DFA dfa)
State state, nextState; char ch;
Results r = new Results();
state = dfa.getStartState();
while (! M.eof()){
    ch = M.getChar();
    nextState = dfa.getTransition(state,ch);
    if (!NextState.isValid()){
        nextState =
*dfa.Failurelist.getFailure(state);
        while((nextState!=null)&&
            !
dfa.getTransition(nextState,ch).isValid())

{nextState=*dfa.Failurelist.getFailure(nextState);}
        if (nextState ! = null) {
            nextState = dfa.getTransition(s,ch);}
        else {
            nextState = dfa.getStartState();}
    }
    if (nextState.isMatchlist()) {
     r.add(M.getPosition(),*dfa.Matchlist);}
    state = *dfa.NextState[state.getChar];}
 return r;}
```

II.4 Đánh giá độ phức tạp của thuật toán

Đánh giá thuật toán 1 khi xây dựng ma trận chuyển trạng thái dựa trên DFA ứng với tập mẫu *P* sử dụng cấu trúc bảng chỉ số ký tự ta có độ phức tạp tính toán là O(*n*)=*n***k*. Trong đó: *n* là số lượng mẫu còn *k* là độ dài lớn nhất của các mẫu trong tập mẫu *P*. Nhưng do độ dài các mẫu này rất nhỏ so với số lượng mẫu trong tập mẫu *P*, tức là *k*<<*n* nên ta có thể coi *k* như một hằng số. Vậy độ phức tạp của thuật toán 1 là O(*n*)=*n* và độc lập với kích thước chuỗi vào nên cho tốc độ thực thi tốt với tập nhiều mẫu.

Độ phức tạp tính toán của thuật toán 2 khi xây dựng bảng chỉ số con trỏ failure ứng với DFA của

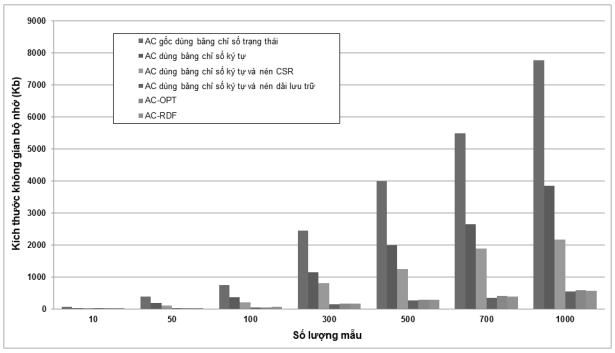
thuật toán 1 sinh ra là O(*n*) = 256*(*số trạng thái của DFA*) thay vì là O(*n*)=*m**(*số trạng thái của DFA*) của thuật toán AC gốc. Ở đây *m* là độ dài của chuỗi dữ liệu vào thường *m* là rất lớn. Độ phức tạp tính toán của thuật toán 3 khi tìm kiếm mẫu trên bảng chỉ số ký tự O(*n*) = 256*(*số trạng thái của DFA)* thay vì là *m**(*số trạng thái của DFA*) của thuật toán AC gốc.

Như vậy, qua đánh giá trên có thể nhận thấy thuật toán mà chúng tôi đề xuất cho số lượng không gian trạng thái của DFA nhỏ hơn so với thuật toán AC gốc.

III. THỰC NGHIỆM VÀ ĐÁNH GIÁ

Để đánh giá hiệu quả về không gian bộ nhớ thực thi, chúng tôi đã khai báo và cài đặt thử nghiệm trên hệ thống Snort 2.4.2 với thuật toán AC gốc, thuật toán AC-OPT [8], thuật toán AC sử dụng dụng định dạng dòng thay thế (AC-RDF) [10], và thuật toán AC sau khi đã nén không gian lưu trữ các trạng thái. Số lượng mẫu thực nghiệm là lần lượt từ 10 đến 1000 mẫu, chiều dài các mẫu từ 8 đến 30 ký tự trên bảng chữ cái |*S*|=256.

Kết quả đánh giá không gian bộ nhớ cuả thuật toán AC chuẩn so với các thuật toán AC cải tiến đã có và thuật toán mà chúng tôi đề xuất sử dụng kỹ thuật nén không gian lưu trữ từ bảng chỉ số trạng thái sang bảng chỉ số ký tự được cho trong Hình 5.



Hình 5- So sánh không gian bộ nhớ của thuật toán AC với các cách tiếp cận lưu trữ trạng thái khác nhau.

Kết quả thống kê thực nghiệm so sánh thuật toán cải tiến của chúng tôi với thuật toán AC gốc trên các tập luật chuẩn của Snort 2.4.2 được cho trong Bảng 1 với 2 tiêu chí là số lần chuyển trạng thái và tổng số trạng thái.

Qua thực nghiệm ta dễ nhận thấy số lượng các trạng thái được giảm đi phụ thuộc rất nhiều vào số lượng mẫu và số lượng các ký tự. Khi số lượng mẫu càng lớn và tổng số ký tự càng nhiều thì hiệu quả tối ưu càng lớn.

IV. KẾT LUẬN

Trong bài báo này, chúng tôi đã phân tích kỹ thuật nén để tối ưu không gian trạng thái các thuật toán so khớp chuỗi AC dùng bảng chỉ số thay cho bảng chỉ số trạng thái và thử nghiệm trên hệ thống phát hiện xâm nhập mạng Snort 2.4.2. Các kết quả thực nghiệm cho thấy thuật toán mà chúng tôi đề xuất cho kết quả bằng và tốt hơn thuật toán AC gốc và một số thuật toán AC đã được cải tiến AC-OPT [8] và AC-RDF [10]. Việc hiểu rõ cấu trúc biểu diễn không gian trạng thái của thuật toán sẽ giúp chúng ta xây dựng các hệ thống an ninh mạng đạt hiệu quả cao trong thực tế đáp ứng được sự phát triển nhanh và đa dạng của các mẫu virus.

Các hướng tiếp cận mới hiện nay được trình bày trong [12-20]. Sau khi tối ưu các thuật toán người ta còn chuyển các thuật toán thành dạng kiến trúc bộ nhớ được thiết kế sẵn cho phép thực thi quá trình so khớp một cách nhanh nhất sử dụng thêm cấu trúc bảng băm đa lựa chọn. Khi kết hợp sử dụng nhiều hàm băm sẽ làm tăng khả năng tìm kiếm và tốc độ so sánh giữa tập mẫu với nội dung gói tin cần kiểm soát. Việc xây dựng mô hình ánh xạ địa chỉ bộ nhớ trong bảng băm dựa trên các thông tin tóm tắt được tích hợp trên thiết bị phần cứng hay tiếp cận song song hóa các thuật toán cũng là những hướng đầy hứa hẹn và đây là hướng tiếp cận nghiên cứu của chúng tôi trong tương lai.

LỜI CẢM ƠN

Bài báo này được thực hiện do sự hỗ trợ một phần kinh phí từ đề tài QG.12.21, Đại học Quốc gia Hà Nội.

Bảng 1. Thống kê không gian trạng thái thực nghiệm trên Snort với các tập luật chuẩn

Tập luật	Số lượng mẫu	Số lượng ký tự	Thuật toán AC gốc		Thuật toán AC cải tiến		Tỷ lệ % tổng số trạng thái rút gọn được
			Số lần chuyển trạng thái	*Tổng số trạng thái*	*Số lần chuyển trạng thái*	*Tổng số trạng thái*	
Ftp	96	466	402	406	391	375	7.63 %
Smtp	104	989	715	719	613	602	16.27 %
Web-misc	160	2.052	1.420	1.425	1.318	1.259	11.65 %
Oracle	337	11.128	6.793	6.804	4.512	3.957	41.84 %

TÀI LIỆU THAM KHẢO

[1] H. DEBAR, M. DACIER, and A. WESPI, *Towards a taxonomy of intrusion-detection systems*, Computer Networks, vol. 31, pp. 805-822, 1999.

[2] CHRISTIAN CHARRAS, THERRY LECROQ, T, *Handbook of Exact String Matching Algorithms*, King's College Publications, 2004.

[3] LÊ ĐẮC NHƯỜNG, LÊ ĐĂNG NGUYÊN, TRỊNH THỊ THÙY GIANG, LÊ TRỌNG VĨNH. *Phân tích, đánh giá hiệu quả của các thuật toán so khớp chuỗi dùng trong an ninh mạng*, Hội thảo các vấn đề chọn lọc về CNTT & TT lần thứ 14, Tr.451-463, Cần Thơ 7-8/10/2011. NXB Khoa học kỹ thuật Hà Nội 2012.

[4] JOHNSON, S.C, Yacc. *Yet another compiler*, Computing Science Technology Report, AT&T Bell Laboratories, Murray Hill, N.J, 1975.

[5] TARJAN, R.E and Yao, A.C-C. *Storing a sparse table*. ACM 21,1979.

[6] STEPHEN GOSSEN, NEIL JONES, NEIL MCCURDY, RAYAN PERSAUD. *Pattern Matching in Snort*, 2002.

[7] T.V LAKSHMAN AND D.STIDIALIS. *High Speed Policy-Based packet Forwarding using Efficient Multi-Dementional Range Matching*, ACM Sigcomm, 1998

[8] MARS A.NORTOON et.al, *Methods and Systems for Multipattern Searching*, Patent US7996424, 2009

[9] MARC NORTON AND DANIEL ROELKER, *Snort Multi-rule Inspection Engine*. www.idsreserch.org/papers.html

[10] BRANIMIR Z. LAMBOV, *Efficient Storage for Finite State Machines*, Patent 7949679, 2011.

[11] ALFRED V. AHO and MARGARET J. Corasick. 1975. *Efficient string matching: an aid to bibliographic search*. Commun. ACM 18, pages 333-340.

[12] F.YU, R. H. KATZ, and T. V. LAKSHMAN, *Gigabit rate packet pattern matching using TCAM*, Proc 12th IEEE Int Conf Networks Protocols (ICNP),pp.174-183, 2004.

[13] Z. W. ZHOU,Y. B. XUE, J. D. LIU, W. ZHANG, and J. LI, *MDH: A High Speed Multi-Phase Dynamic Hash String Matching Algorithm for Large-Scale Pattern Set*, Proc. of the 9th International Conference on Information and Communication Security (ICICS), 2007.

[14] TIAN SONG, WEI ZHANG, DONGSHENG WANG, YIBO XUE, *A Memory Efficient String Matching Architecture for Network Security*, In 27th Conference of INFOCOM 2008.

[15] X. ZHOU, B. XU, Y. X. QI, et al. MRSI: a Fast *Pattern Matching Algorithm for Anti-virus Applications*. Proceedings of the 7th International Conference on Networking (ICN), 2008

[16] CHENG-HUNG LIN, YU-TANG TAI, SHIH-CHIEH CHANG, *Optimization of Pattern Matching Algorithm for Memory Based Achitecture*. ACM/IEEE, 2009

[17] KEDAR NAMJOSHI V GIRIJA NARLIKAR, *Robust and Fast Pattern Matching For Intrusion Detection*, Bell Labs, INFOCOM 2010.

[18] CHENG-HUNG LIN, AND SHIH-CHIEH CHANG, *Efficient Pattern Matching Algorithm for Memory Architecture*, IEEE Transactions on Very Large Scale Integration (VLSI), 2011.

Nhận bài ngày: 14/08/2012

SƠ LƯỢC VỀ CÁC TÁC GIẢ

LÊ ĐẮC NHƯỜNG

Sinh năm 1983 tại Hải Phòng.

Tốt nghiệp Thạc sĩ CNTT tại Đại học Công nghệ, ĐHQG Hà Nội năm 2009.

Hiện là giảng viên Khoa Công nghệ thông tin, Đại học Hải Phòng.

Lĩnh vực nghiên cứu chính là: Lý thuyết thuật toán, Mạng máy tính và An ninh mạng.

Email: Nhuongld@hus.edu.vn

LÊ ĐĂNG NGUYÊN

Sinh năm 1976 tại Hải Phòng.

Tốt nghiệp Thạc sĩ CNTT tại Đại học Công nghệ, ĐHQG Hà Nội năm 2005.

Hiện công tác tại Đại học Hải Phòng.

Lĩnh vực quan tâm: Lý thuyết thuật toán, Mạng máy tính

Email: Nguyenld@hus.edu.vn

LÊ TRỌNG VĨNH

Sinh năm 1973 tại Thanh Hóa.

Nhận học vị Tiến sĩ năm 2006, tại Viện khoa học và công nghệ tiên tiến Nhật bản, chức danh PGS năm 2012.

Hiện là giảng viên Khoa Toán-Cơ-Tin học, Trường Đại học Khoa học Tự nhiên, ĐHQG Hà nội.

Lĩnh vực nghiên cứu chính là: Lý thuyết thuật toán và mạng máy tính.

Email: Vinhlt@vnu.edu.vn