Journal of The Korea Institute of Information Security & Cryptology(JKIISC) ISSN 1598-3986(Print)
VOL.23, NO.3, June 2013 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2013.23.3.395

동일한 입력 문자를 가지는 상태의 병합을 통한 메모리 효율적인 결정적 유한 오토마타 구현

최 윤 호†‡
경기대학교

Design of Memory-Efficient Deterministic Finite Automata by Merging States With The Same Input Character

Yoon-Ho Choi†‡
Kyonggi University

요 약

패턴 정합 알고리듬은 침입 탐지 및 방지 시스템의 성능을 좌우하는 중요한 기능 요소로서 일반적으로 정규 표현식(Regualr Expressions)을 사용해 패턴을 표현한다. 공격 패턴이 복잡해지고 다양해짐에 따라, 정규 표현식 또한 복잡해지고 그 수가 증가하고 있으며 이로 인해, 패턴 매칭 알고리듬에서 정규 표현식을 인식하기 위해 사용된 결정적 유한 오토마타(Deterministic Finite Automata)를 구성하는 상태가 폭발적으로 증가(states blowup)하고 있다. 이러한 상태의 폭발적 증가 문제를 해결하고 메모리 효율적인 자료 구조를 구현하기 위해 많은 연구가 이루어졌다. 대부분의 연구 결과들에서는 하나의 정규 표현식을 변환한 결정적 유한 오토마톤(Automaton) 내 상태의 수를 감소시키기 위한 효과적인 방안들을 제안하였다. 하지만, 이들 연구 결과는 단일 패턴 내 상태의 수만을 감소시킬 뿐 패턴의 수에 따라 증가하는 상태의 수를 감소시키지 못하는 한계점을 가지고 있다. 본 논문에서는 이를 해결하기 위해 정규 표현식으로 구성된 유한 오토마타(Automata) 상의 상태 병합을 통한 상태 감소 방안을 제안한다. 이는 동일한 입력 문자를 가지는 상태를 병합함으로써 유한 오토마타 상의 상태의 수를 감소시켜, 기존 결정적 유한 오토마타에 비해 평균 40.0%의 메모리 감소 효과를 나타낸다.

ABSTRACT

A pattern matching algorithm plays an important role in traffic identification and classification based on predefined patterns for intrusion detection and prevention. As attacks become prevalent and complex, current patterns are written using regular expressions, called regexes, which are expressed into the deterministic finite automata(DFA) due to the guaranteed worst-case performance in pattern matching process. Currently, because of the increased complexity of regex patterns and their large number, memory-efficient DFA from states reduction have become the mainstay of pattern matching process. However, most of the previous works have focused on reducing only the number of states on a single automaton, and thus there still exists a state blowup problem under the large number of patterns. To solve the above problem, we propose a new state compression algorithm that merges states on multiple automata. We show that by merging states with the same input character on multiple automata, the proposed algorithm can lead to a significant reduction of the number of states in the original DFA by as much as 40.0% on average.

Keywords: pattern matching, regular expression, DFA, state blowup, state merging

접수일(2012년 12월 6일),수정일(1차: 2013년 3월 28일, 2차: 2013년 5월 8일),게재확정일(2013년 5월 22일)

† 주저자, ychoi@kyonggi.ac.kr
‡ 교신저자, ychoi@kyonggi.ac.kr(Corresponding author)

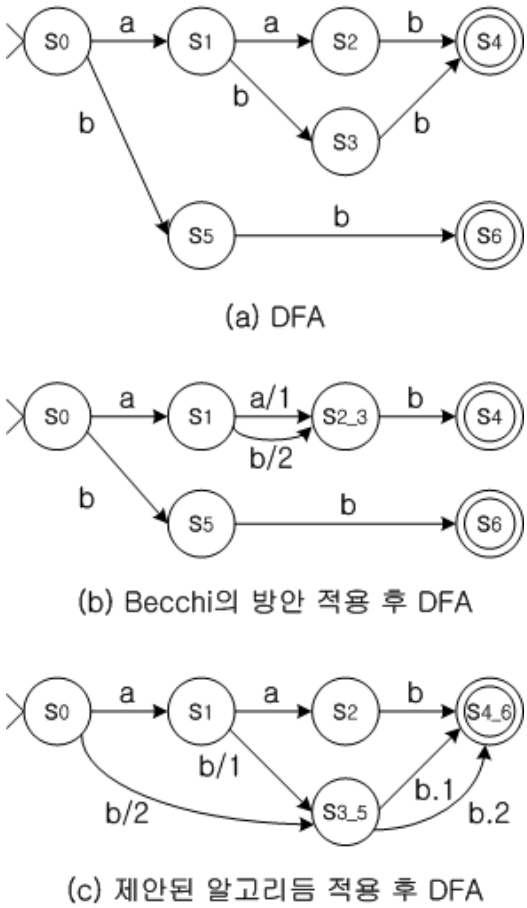
I. 서 론

차세대 네트워크 보안 관리 장비들은 다수의 접속 망들로부터 유입 혹은 유출되는 초당 수~수십 Terabit 이상의 트래픽을 탐색하고 처리함과 동시에 지능화되고 다양화되는 공격 기법에 대응하기 위해 정상적인 트래픽과 악의적인 트래픽을 구분할 수 있어야 한다. 따라서 OSI 7계층 영역에서 페이로드(Payload)에 존재하는 데이터의 유효성에 대해 검사하고 허용된 포트를 이용한 공격이 발생된 경우 이를 효율적으로 탐지하고 차단하기 위한 기술에 대한 필요성이 증대하고 있다.

Deep Packet Inspection(DPI)은 실시간으로 네트워크 트래픽을 식별하고 분류 및 추출하기 위한 진보된 형태의 패킷 분석 기술로서, OSI 2계층부터 7계층까지의 트래픽 헤더 및 페이로드를 분석해 패킷의 헤더뿐만이 아니라 페이로드에 존재하는 특정 문자열 패턴을 탐지할 수 있다. 이는 기존의 트래픽 헤더 분석을 통한 트래픽 분석에 사용된 stateful inspection과 비교해 페이로드 데이터 분석을 통한 애플리케이션 계층의 콘텐츠 및 프로토콜 정보에 대한 깊이 있는 분석을 가능하게 한다. 이러한 정밀 분석을 위해 DPI는 트래픽에 대한 헤더 분석, 정형화(예: Unicode to ASCII code) 및 페이로드 데이터에 존재하는 공격 패턴을 탐지하기 위한 패턴 정합 알고리듬으로 구성되어 있으며 이 중, 패턴 정합 알고리듬은 DPI를 이용한 트래픽 분석 및 처리 성능의 40%~70%를 차지하는 것으로 알려져 있다[16].

패턴 정합 알고리듬[3]-[12]은 악의적인 트래픽을 식별하기 위해, 페이로드 혹은 URI content 문자열의 처음부터 끝까지 탐색하며 미리 정의된 악성 코드의 문자열인 패턴의 존재 유무를 식별하게 된다. 이 과정에서, 악의적 트래픽 탐색을 위한 다양한 패턴의 존재 유무는 패턴 정합 알고리듬의 정확도를 결정하는 중요한 요소인 동시에 알고리듬의 동작 속도를 결정하는 중요한 요소로 작용한다. 또한, 패턴 다양성의 증가는 공격 패턴에 대한 탐지 시간이 길어지는 결과를 초래하며 패턴의 수적인 증가와 더불어, 패턴을 저장하기 위한 메모리 사용량을 증가시키는 요소로 작용하고 있다.

오늘날 다양화되고 증가하는 공격 유형에 대응하기 위해 기존의 문자열로 표현된 패턴은 특정한 규칙을 가진 문자열의 집합을 표현하는 형식 언어인 정규 표현식(Regular Expressions, regex) 패턴으로



〔그림 1〕 P_1="bb"과 P_2="a(a|b)b"의 DFAs 비교: (a)original DFA[14]; (b)Becchi 기법[10] 적용 후 생성된 DFA; (c)제안된 알고리듬 적용 후 생성된 DFA

표현되고 있으며, 현재 수천~수만 개에 이르고 있는 패턴의 수는 계속 증가할 것으로 예상된다. 이러한 정규 표현식을 컴퓨터가 인식하는 자료 구조인 결정적 유한 오토마타(Deterministic Finite Automata, DFA)로 변환하는 과정에서, 문자열 탐색 시 문자의 입출력 정보를 저장하기 위한 상태(state)의 폭발적 증가 문제(states blowup)가 발생하고 있으며 이로 인한 메모리 사용량의 증가는 패턴 정합 알고리듬의 구현을 시스템 메모리 사양에 의존적으로 만드는 문제점을 초래하였다.

이러한 상태의 폭발적 증가 문제를 해결하기 위한 대표적 연구 결과인 Becchi 방법론의 경우, 그림 1-(a)와 같이 하나의 정규 표현식 P_2="a(a|b)b"에 대한 DFA에서 동일한 출력 상태로 천이가 발생하는

상태들 S2와 S3를 결합하여 그림 1-(b)에서와 같은 하나의 상태 S2_3로 표현하는 과정을 통해, DFA내 상태의 수를 줄임으로써 DFA 구현에 필요한 메모리 량을 감소시킬 수 있었다. 하지만, Becchi 방법론을 활용한 상태 감소 효과는 그림 1-(b)에서 같이 하나의 정규 표현식 패턴 P_2로부터 생성된 DFA상에 존재하는 상태의 결합에 한정된다.

본 논문에서는 이러한 한계를 극복하기 위해 서로 다른 패턴에 존재하는 상태의 결합을 가능하게 하는 새로운 알고리듬인 SIC(Same Input character-based state Compression) 알고리듬을 제안한다. 제안된 SIC 알고리듬은 그림 1-(c)에서와 같이, 두 패턴 P_1과 P_2로 부터 생성된 DFA에서 동일한 입력 문자 b를 가지는 상태들을 결합하여 새로운 상태로 표현하는 과정을 통해, DFA내 상태의 수를 줄임으로써 메모리 효율적인 DFA를 구현한다. 즉, P_2로부터 생성된 상태 S4와 P_1으로부터 생성된 상태 S6, P_2로부터 생성된 상태 S3와 P_1으로부터 생성된 상태 S5를 결합함으로써 DFA상에 존재하는 상태의 수를 감소시킨다.

본 논문은 다음과 같이 구성되어 있다. 먼저 단락 II에서는 관련 연구 결과를 소개하고, 단락 III에서는 제안된 알고리듬을 이해하기 위해 일반적인 DFA를 활용한 정규 표현식 패턴 매칭 알고리듬에 대해 설명한다. 단락 IV에서 제안된 알고리듬의 상세 동작을 설명하고, 단락 V에서 그 성능 검증 결과를 기술한다. 마지막으로 단락 VI에서 검증 결과를 바탕으로 본 논문을 요약한다.

II. 관련 연구

정규 표현식이 복잡 다양해지고 그 수가 증가함에 따라 발생 가능한 폭발적 상태의 증가 문제를 해결하기 위한 많은 방안[1]-[10]이 제안되었다.

먼저 [1]과 [2]에서 Hopcroft는 임의의 주어진 패턴에 대해 최소 상태를 가지는 DFA가 정의될 수 있음을 보여주었다. 하지만, 각 정규 표현식 패턴이 서로 다른 패턴을 나타내므로 정규 표현식 패턴의 수가 증가함에 따른 상태의 증가는 피할 수 없다는 한계를 극복할 수 없었다.

패턴 수의 증가에 따른 상태의 폭발적 증가에 효과적으로 대응하기 위해 최근에 잘 알려진 방법인 XFA[4]의 경우, 패턴의 수(N)와 동일한 비트를 가지는 scratch memory를 사용하여 상태 변환 정보를 표현함으로써 패턴의 평균 길이가 l인 경우에 기존 DFA의 공간 복잡도 $O(Nl2^N)$을 $O(Nl)$로 감소시켰다. 또한 VS-DFA[5]에서는 메모리 감소를 위해 상태 변환 테이블을 direct lookup 테이블이 아닌 hash 테이블로 구현하는 방안이 소개되었다.

[표 1] 유사 알고리듬 별 상태 병합 조건 및 특징

	CD2FA [8]	Becchi 알고리듬 [10]	제안된 알고리듬
특징	엄격한 병합 조건	완화된 병합 조건	완화된 병합 조건
병합 조건	동일한 입력 문자 및 공통의 출력 상태를 가지는 경우	공통의 출력 상태를 가지는 경우	동일한 입력 문자를 가지는 경우

다른 대표적인 상태 감소 방안으로는 Tuck 등에 의해 제안된 bitmap compression 방안이 있다[6]. 이는 문자열을 기본적으로 256bit bitmap을 사용해 압축하는 방식으로 현재 상태에서 천이 가능한 다음 상태를 256bit bitmap을 사용해 표현한다. 즉, bitmap은 주어진 입력에 대해 현재 상태로부터 천이가 가능한 다음 상태의 position 정보를 담고 있어 상태를 표현하기 위해 사용되는 메모리 사용량을 줄일 수 있다.

Kumar에 의해 제안된 D2FA(Delayed Input DFAs)[7]에서는 불필요한 메모리 접근을 피하고 서로 다른 상태를 구분하기 위해 state number 대신 content label을 사용하였다. 또한, Kumar는 D2FA의 확장인 CD2FA(Content Addressed Delayed Input DFA)[8]를 소개하였는데 이는, 동일한 입력 문자를 가지고 공통의 상태로 상태 천이가 일어나는 일부 상태들을 병합함으로써 상태의 수를 감소시켰다. 하지만 이 방법의 경우 결합 대상이 되는 상태들이 동일한 입력 문자를 가지고 공통의 출력 상태로 천이가 일어나야 한다는 엄격한 결합 조건을 만족시켜야 한다는 단점이 존재하였으며 이로 인해 실제 잘 알려진 패턴 집합을 통한 실험에서 그 효과가 크지 않았다.

이를 극복하기 위해 Becchi는 새로운 방법론을 제안하였다[10]. 이는 기존 C2FA에 비해 단지 동일한 출력 상태로 천이가 발생하는 경우 결합이 가능한 약한 결합 조건을 통해 상태의 감소 효과를 얻는 방법으로 결합된 상태에서의 출력 상태를 구분하기 위해 bitmap을 사용하였다. 본 논문에서 제안한

방안은 Becchi에 의해 제안된 방안과 유사하게 약한 결합 조건을 통해 상태들을 병합하고 이를 통해, 결정적 유한 오토마타 내 존재하는 상태의 수를 감소시켜 메모리 사용량을 줄이는 방법이다.

Becchi의 방법은 하나의 정규 표현식을 표현한 결정적 유한 오토마톤(automaton) 내에 존재하는 동일한 출력을 가지는 상태들의 결합을 통해 하나의 정규 표현식 오토마톤 내 상태의 수를 감소시킬 수 있다. 이와 달리 제안되어진 알고리듬은 다중 정규 표현식을 표현한 결정적 유한 오토마타 내에 존재하는 동일한 입력 문자를 가지는 상태들을 결합함으로써, 정규 표현식이 복잡 다양해지고 그 수가 증가함에 따라 발생 가능한 DFA내 폭발적 상태 증가 문제를 해결하고 시스템 메모리 사양에 의존적인 패턴 매칭 알고리듬의 구현 문제를 해결한다. 제안된 알고리듬과 기존 유사 알고리듬의 상태 병합 조건을 표 1에서 요약 비교하였다.

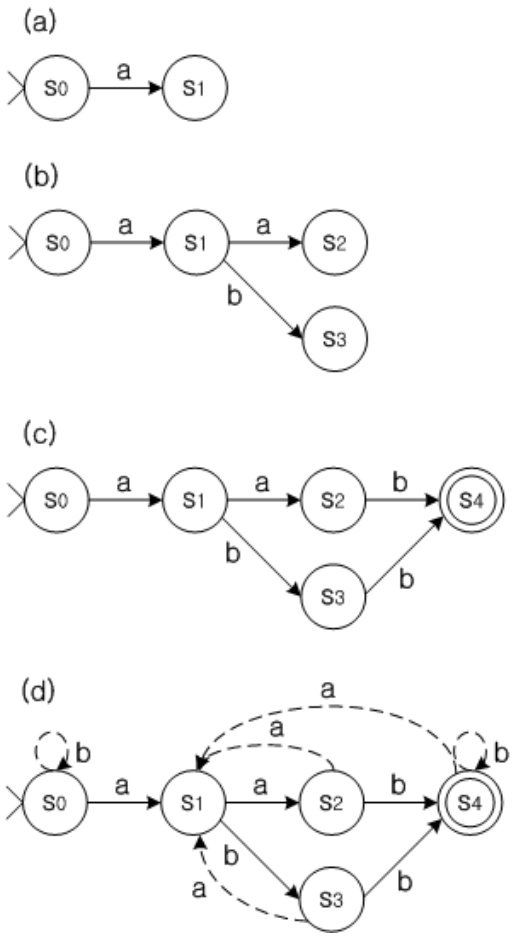
III. 기술적 배경 지식

본 단락에서는 정규 표현식과 DFA에 대해 살펴보고 DFA를 활용한 정규 표현식 패턴 매칭 과정에 대해 소개한다.

3.1 정규 표현식과 결정적 유한 오토마타

정규 표현식은 특정한 규칙을 가진 문자열의 집합을 표현하는 데 사용하는 형식 언어로서, 일반적인 문자와 특정한 논리적 관계를 나타내는 기호로 구성되어 있다. 정규 표현식은 기계가 받아들이는 유한 오토마타(Finite Automata, FA)로 변환가능하다. 유한 오토마타의 모든 상태는 주어진 입력이 기계에 의해 받아들여지는지(accept state 혹은 final state) 또는 받아들여지지 않는지(failure state)에 대한 값을 가진다.

주어진 입력에 대해 한 가지 또는 여러 가지 변환된 상태를 가지는 비 결정적 오토마타(Non-Deterministic Finite Automata, NFA)에 비해 결정적 유한 오토마타(Deterministic FA,DFA)의 경우 정확히 하나의 변환된 상태를 가지기 때문에 고속의 처리 속도를 필요로 하는 패턴 매칭 알고리듬에서 보편적으로 사용되고 있다. 잘 알려진 바와 같이 DFA는 다음의 5-튜플(S, S0, Σ, δ, F)을 가지는 수학적 모델로 정의된다[14].



[그림 2] 정규 표현식 "a(a|b)b"의 DFA 변환. '〉'기호는 시작 상태, 원은 입력 값(문자)에 대한 현재 상태, 이중 원은 final state를 표시하며 실선과 점선은 각각 accept state와 failure state로의 상태 전이를 의미.

- S: 상태의 유한집합, $S \neq \varnothing$
- S0: 초기 상태, $S0 \in S$
- Σ: 입력 값의 유한집합, $\Sigma \neq \varnothing$
- δ: 상태 천이 함수, $S \times \Sigma \rightarrow S$
- F: 발견된 특정한 의미 단위를 나타내는 최종 상태의 집합, $F \subset S$

예를 들어, 입력 문자의 집합 Σ={a,b}로부터 생성된 정규 표현식 "a(a|b)b"을 DFA로 변환하는 경우, 그림 2와 같이 각 입력 문제는 상태 천이 초기 상태 S0를 현재 상태로 상태 천이를 발생시킨다. 즉, 첫 번째 입력 문자 'a'는 그림 2-(1)과 같이 현재 상태를 S0에서 S1로 상태 천이(실선 화살표) 시키고,

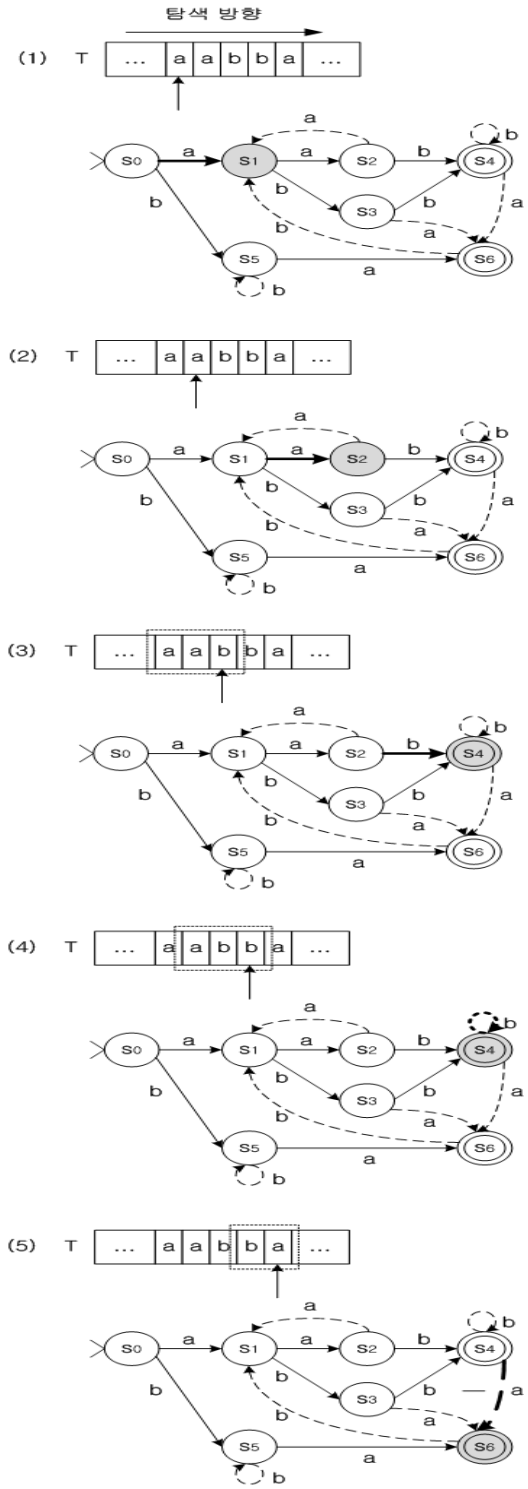
묶기 기호와 선택 기호로 구성된 두 번째 입력 문자 '(a|b)'는 그림 2-(2)와 같이 입력 문자 'a'에 대해 현재 상태를 S2로, 입력 문자 'b'에 대해 현재 상태를 S3로 상태 천이시킨다. 또한 세 번째 입력 문자 'b'는 그림 2-(3)과 같이 현재 상태를 S2와 S3에 대해 최종 상태인 S4(S4∈F⊂S)로 천이시킨다. 마지막으로 입력 문자가 현재 상태에서 받아들여지지 않는 경우에 대해 그림 2-(4)와 같은 상태 천이(점선 화살표)를 가진다.

3.2 정규 표현식 패턴 정합 알고리듬

패턴 정합 알고리듬은 일반적으로 미리 정의된 정규 표현식 패턴을 NFA로 다시 DFA로 변환하는 전처리 단계와 DFA를 이용하여 망으로 유입/유출되는 트래픽 내 해당 패턴의 존재 유무를 식별하는 탐색 단계로 구성된다. 정규 표현식 패턴 매칭 알고리듬의 동작을 설명하기 위해 문자열의 집합 Σ={a,b}, P_1="a(a|b)b"과 P_2="ba"로 구성된 정규 표현식 패턴 집합 P={P_1, P_2}, 입력 트래픽 내 문자열 T="aabba"를 가정하자.

패턴 정합 알고리듬은 그림 2와 같은 '전처리 단계'에서 P_1과 P_2를 DFA로 변환한 결과를 활용하여 그림 3과 같은 '탐색 단계'에서 정규 표현식 패턴의 존재 유무를 알아내기 위해 입력 문자열 T를 문자열의 처음부터 오른쪽(그림 3의 T내 화살표 이동 방향)으로 탐색한다. 먼저 첫 번째 입력 문자 'a'는 그림 3-(1)과 같이 DFA내 현재 상태(회색 원)를 S0에서 S1으로 천이(굵은 화살표)시킨다. 또한, 두 번째 입력 문자 'a'는 그림 3-(2)와 같이 DFA내 현재 상태를 S1에서 S2로 천이 시킨다. 다음 입력 문자 'b'는 그림 3-(3)과 같이 현재 상태가 최종 상태인 S4로 천이하게 되며 T에서 P_1("aab")의 존재를 확인할 수 있다. 또한, 다음 문자들 'b'와 'a'에 대해서도 그림 3-(4)와 3-(5)에서와 같이 현재 상태가 각각 최종 상태인 S4와 S6로 천이하므로 T에 있는 P_1("abb")과 P_2("ba")를 확인할 수 있다.

이러한 DFA를 활용한 대표적인 패턴 정합 알고리듬으로는 AC(Aho-Corasick) 알고리듬[3]과 AC의 변형인 KMP(Knuth-Morris-Pratt) 알고리듬[15]와 같은 많은 알고리듬이 존재한다. 하지만 정규 표현식 패턴의 수가 수 천~수 만 개에 달하고, 유사 문자를 가지는 문자열을 조합하기 위해 패턴 내 논리 기호의 사용이 증가함에 따라 이러한



[그림 3] 정규 표현식 패턴 정합 알고리듬의 동작 설명을 위한 예제. T="aabba", P_1="a(a|b)b"와 P_2="ba".

알고리듬 1 SIC(P)

```
1: P←P의 정규 표현식 패턴들을 길이에 따라 역방향
     정렬;
2: DFA←NFA←P;
3: Pt←ϕ, SS←ϕ;
4: while i=0:(P-Pt)!=NULL do
5:    Pt←Pi;
6:    for j=i+1:(P-Pt)!=NULL do
7:      if 출력 문자가 b.i인 Sx와 출력 문자가 c.j인
Sy의 입력문자가 a로 동일 then
8:          SS←(Sx,a,b,i)와 (Sy,a,c,j);
9:      end if
10:     if SS!=NULL then
11:         DFA←DFA-(Sx,a,b,i)
                -(Sy,a,c,j)
        +(Sx_y,a/i,a/j,b.i,c.j);
12:     end if
13:   end for
14:   i++;
15: end while
```

알고리듬의 DFA내 상태의 수가 폭발적으로 증가하고 있다. 이로 인해 기존 알고리듬의 경우 실제 수천 개 이상 되는 패턴을 처리하고 그 결과물인 상태 정보를 저장하는 과정에서 수백 MB~수 GB에 이르는 많은 메모리를 필요로 하며 이로 인해, 알고리듬의 구현이 시스템의 메모리 사양에 의해 제한되는 한계가 존재한다.

예를 들어, 그림 3의 DFA에서와 같이 P_1내 두 문자열 "aab"와 "abb"를 표현하는 과정에서 사용된 기호 '|'는 두 문자열의 가운데 위치한 'a'와 'b' 두 문자를 구분하는데 효과적이지만 독립된 상태인 S2와 S3로 표현되어야 하고, P_1과 다른 문자 조합으로 구성된 P_2를 DFA로 변환하는 과정에서 S5 및 S6로 인한 상태의 증가를 피할 수 없다.

IV. 제안된 알고리듬

제안된 알고리듬의 구성 및 동작 방식에 대해 설명한다. 먼저 알고리듬을 기술하기 위해 사용된 용어를 정리하면 다음과 같다.

4.1 용어 정의

- **P**: 유한한 정규 표현식 패턴의 집합
- **Pt**: P내 가장 길이가 긴 패턴의 집합
- **P_i**: i번째 패턴
- **DFA**: 결정적 유한 오토마타
- **LLP**: 가장 길이가 긴 패턴의 길이(length of the longest pattern)
- **SS**: 병합 가능한 상태의 집합
- **Sx, Sy**: DFA를 구성하는 임의의 상태 (Sx≠Sy∈S)
- **(Sx,a,b.i)**:임의의 정규 표현식 i를 변환한 DFA에서 입력 문자 a, 출력 문자 b를 가지는 상태Sx(a, b∈Σ)
- **(Sy,a,c.j)**: 임의의 정규 표현식 j에 대한 DFA에서 입력 문자 a, 출력 문자 c를 가지는 상태 Sy(a, c∈Σ)
- **(Sx_y,a/i,a/j,b.i,c.j)**: 정규 표현식 i를 변환한 DFA에서 입력 문자 a(a/i)와 출력 문자 b(b.i)를, 정규 표현식 j를 변환한 DFA에서 입력 문자 a(a/j)와 출력 문자 c(c.j)를 가지는 서로 다른 상태 Sx와 Sy에 대해 SIC 알고리듬을 적용한 후 생성된 통합 상태 Sx_y(Sx_y∈S)

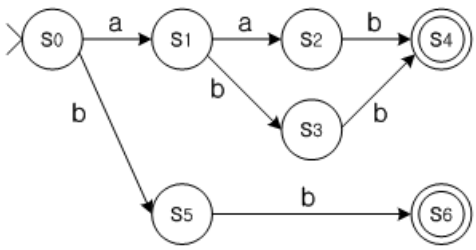
4.2 알고리듬 구성

제안된 SIC 알고리듬은 동일한 입력 문자를 가지는 상태를 병합하기 위한 초기화, 병합 가능한 상태 찾기 및 상태 병합의 3단계로 구성된다.

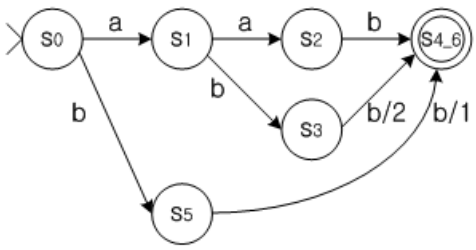
- **초기화 단계**: P에 존재하는 정규 표현식 패턴을 길이에 따라 역방향 정렬하고 정렬된 패턴으로 부터 DFA를 생성한다.
- **병합 가능한 상태 찾기 단계**: 가장 길이가 긴 패턴으로부터 생성된 결정적 유한 오토마톤과 나머지 패턴으로부터 생성된 결정적 유한 오토마톤 내 상태 간 비교 과정을 통해 동일한 입력 문자를 가지는 상태를 찾는다. 즉, (Sx,a,b.i)와 (Sy,a,c.j) (Sx≠Sy∈S).
- **상태 병합 단계**:(Sx,a,b.i)와 (Sy,a,c.j)를 병합한 새로운 상태 (Sx_y,a/i,a/j,b.i,c.j)를 표현한다.

4.3 알고리듬 동작

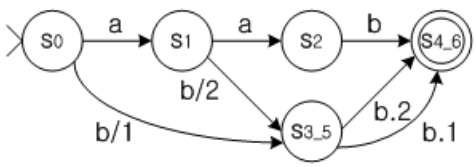
각 구성 단계 별 상세 동작을 알고리듬 1에서 기술하였다. 먼저, 첫 번째 줄에서 세 번째 줄은 초기화 단계로서, 주어진 패턴으로부터 병합 가능한 상태를 찾기 위해 패턴을 역방향 정렬하고 5-튜플(S, S0, Σ, δ, F)로 구성된 DFA를 생성한다. 이 때, 일반적인 DFA 변환 과정인 P에서 NFA를 생성하고



(a) 초기 DFA



(b) 상태 S4와 S6의 결합에 의해 형성된 DFA



(c) 상태 S3와 S5의 결합에 의해 형성된 DFA

〔그림 4〕 SIC 동작 설명을 위한 예제. P_2="a(a|b)b"와 P_1="bb".

다시 DFA를 생성하는 과정을 따른다고 가정한다. 제안된 알고리듬의 성능이 각 상태의 입력 문자에 의존적이므로, 임의의 정규 표현식 패턴의 길이는 패턴으로부터 생성되는 오토마톤 내 상태의 수로 정의한다. 한편, 패턴을 역방향 정렬함은 제안된 알고리듬이 오토마톤 간 상태의 비교 과정에서 동일한 입력 문자를 가지는 상태가 초기 상태로부터 멀리 위치할수록 결합 가능한 상태의 수가 증가할 수 있어, 하나의 오토마톤 내 결합 가능한 상태의 수를 최대로 할 수 있기 때문이다.

3번째 줄에서는 P내 가장 길이가 긴 패턴을 저장하기 위한 집합 Pt와 병합 가능한 상태의 집합 SS를 NULL 값으로 초기화한다. 이 때, 가장 긴 길이를 갖는 표현식이 여러 개가 존재하는 경우 패턴 내 문자 혹은 기호의 분포에 따라 알고리듬의 성능이 달라질 수 있다. 본 논문에서는 참고 문헌 〔17〕〔18〕에서 언급된 바와 같이 패턴 매칭 알고리듬 설계 과정, 특히 침입 패턴 탐지 알고리듬, 에서 일반적으로 가정되어지는 패턴 내 각 문자가 uniform하게 분포한다는 가정을 바탕으로 길이가 같은 패턴이 여러 개 존재할 경우, 그 중 하나를 임의 선택한다고 가정한다.

4번째 줄에서 15번째 줄은 병합 가능한 상태를 찾아 그 상태를 병합하는 단계를 기술하고 있다. 정합 가능한 상태 찾기 단계인 7번째에서 9번째 줄에서는, 길이에 따라 역방향 정렬된 패턴 집합 P의 첫 번째 패턴(i=0)과 다른 패턴들(j)로부터 생성된 결정적 유한 오토마톤의 동일한 입력 문자를 가지는 상태 (Sx,a,b.i)와 (Sy,a,c.j)를 찾아서 SS에 저장한다. 상태 병합 단계인 10번째 줄에서 12번째 줄은 SS가 NULL 집합이 아닌 경우, DFA상에서 해당 상태를 병합해 새로운 상태인 (Sx_y,a.i,a.j,b/i,c/j)로 변환한다. 새로운 상태는 서로 다른 정규 표현식 패턴에 존재하는 상태 Sx와 Sy를 구별하기 위해 입력 및 출력 문자와 함께, 각 상태의 정규 표현식 정보인 i와 j를 포함한다.

예를 들어, P_1="bb"과 P_2="a(a|b)b"로 구성된 P={P_1,P_2}를 가정하자. 초기화 단계에서는, P2의 길이 3이 P_1의 길이 2 보다 크므로, 길이에 따라 역방향 정렬된 P로 부터 그림 4-(a)와 같은 DFA를 생성한다. 다음으로 그림 4-(b)에서처럼, 상태 S4와 S6가 동일한 입력 문자 b를 가지므로 두 상태 를 결합하고, 상태 S3와 S5가 동일한 입력 문자 b를 가지므로 그림 4-(c)에서처럼 두 상태를 결합한다. 이 때, P_1의 입력 문자 b와 P2의 입력 문자 b를 구별하고, P_1의 입력 문자 b(b/1)에 대한 출력 문자 b(b.1)와 P2의 입력 문자 b(b/2)에 대한 출력 문자 b(b.2)를 구별하기 위해, 정규 표현식 패턴 번호 1과 2를 입력 및 출력 문자와 함께 저장한다.

4.4 알고리듬 유효성 검증

알고리듬의 동작 유효성을 검증하기 위하여 제안된 SIC 알고리듬으로부터 생성된 DFA내 각 상태가 알고리듬 적용 이전의 DFA에서와 같이 정확히 하나의 변환된 상태를 가지는지 살펴본다.

먼저, 임의의 두 패턴 P_i와 P_{i+1}로부터 생성된 각 오토마톤 내 임의의 상태 Sx와 Sy를 가정하자. 이때 두 상태가 동일한 입력 문자 a를 가지고 Sx의 출력 값이 b이고 Sy의 출력 값이 c인 경우, Sx와 Sy의

〔표 2〕 snort 패턴 ruleset 분석을 통한 제안된 알고리듬(SIC)의 성능 검증 결과

scheme / ruleset	DFA		SIC		감소율
	# of states	Byte	# of states	Byte	
pop2	280	4,480	46	740	83.6%
web-php	709	11,344	126	2,020	82.2%
nntp	309	4,944	57	916	81.6%
imap	1,057	16,912	199	3,192	81.2%
specific-threats	845	13,520	213	3,412	74.8%
chat	340	5,440	87	1,396	74.4%
misc	749	11,984	199	3,188	73.4%
web-cgi	399	6,384	113	1,812	71.7%
web-iis	593	9,488	168	2,692	71.7%
exploit	3,317	53,072	1,079	17,272	67.5%
ftp	1,599	25,584	601	9,624	62.4%
policy	817	13,072	313	5,012	61.7%
sql	241	3,856	97	1,556	59.8%
smtp	4,815	77,040	2,106	33,704	56.3%
dos	249	3,984	134	2,148	46.2%
shellcode	728	11,648	481	7,700	33.9%
info	38	608	30	484	21.1%
mysql	136	2,176	108	1,732	20.6%
pop2	30	480	24	388	20.0%
dns	81	1,296	70	1,124	13.6%
telnet	81	1,296	78	1,252	3.7%
attack-responses	34	544	34	544	0.0%
ddos	18	288	18	288	0.0%
finger	11	176	11	176	0.0%
multimedia	116	1,856	116	1,856	0.0%
p2p	49	784	49	784	0.0%
rpc	27	432	27	432	0.0%
tftp	41	656	41	656	0.0%
web-frontpage	30	480	30	480	0.0%
전체	17,739	283,824	6,655	106,580	40.0%

결합 후 상태 Sx_y는 $\delta(Sx\text{-}1, a, b.i)=Sx_y$와 $\delta(Sy\text{-}1, a, c.j)=Sx_y$로 표현 가능하다. 또한, $\delta(Sx_y, b.i)=Sx+1$와 $\delta(Sx_y, c.j)=Sy+1$가 성립하므로, $\delta(\delta(Sx\text{-}1, a, b.i), b.i)=Sx+1$과 $\delta(\delta(Sy\text{-}1, a, c.j), c.j)=Sy+1$이 성립한다. 따라서 결합된 상태 Sx_y는 주어진 입력에 대해 정확히 하나의 변환된 상태를 갖는 것을 알 수 있다.

V. 성능 평가

제안된 SIC 알고리듬의 DFA상의 상태 감소 효과를 검증하기 위해 잘 알려진 오픈 소스 기반 침입 탐지 시스템인 snort에서 실제 사용되는 snort ruleset의 pcre(Perl-Compatible REgex)를 DFA로 변환하고 알고리듬 적용 전과 후의 상태 수의 변화 및 메모리 사용량을 측정하였다. 이 때, 패턴의 수가 증가함에 따른 상태의 증가 효과를 검증하기 위해서, 기존 연구에서 많이 사용되어진 snort 2.6 ruleset〔13〕 중 2개 이상의 pcre 패턴을 가지는 ruleset을 대상으로 성능을 측정하였다.

표 2에서 요약된 바와 같이 제안된 알고리듬은 최대 83%에 가까운 메모리 감소 효과와 함께, 평균 40%의 메모리 감소 효과를 나타내었다. 검증 결과에서 감소율이 0%인 경우는 실제 ruleset 내 다중 패턴으로 생성된 결정적 유한 오토마타 내 상태가 동일한 입력 값을 가지지 않거나 ruleset 내 서로 다른 pcre 패턴의 수가 2개 이상이 되지 않아 상태 병합이 발생하지 않기 때문인 것으로 관찰되었다. 실제 감소율이 발생한 ruleset의 분석 결과에서는 평균 80.2%에 이르는 메모리 감소 효과를 나타내었다. 또한, 이러한 평균 40%가량 상태의 수의 감소는 메모리 사용량을 280MB에서 106MB로 큰 폭으로 감소시키는 효과를 나타내었다.

VI. 결 론

본 논문에서는 패턴이 복잡해지고 그 수가 증가함에 따라 다수의 정규 표현식 패턴을 결정적 유한 오토마타로 표현하는 과정에서 결정적 유한 오토마타의 상태가 폭발적으로 증가하는 문제를 다루었다. 상태의 폭발적 문제를 효율적으로 해결하기 위해, 서로 다른 패턴에 존재하는 동일한 입력 문자를 가지는 상태를 병합하는 알고리듬을 제안하고 그 성능을 검증하였다. 성능 검증 과정에서는 실제 오픈 소스 기반 시스템에서 사용되는 패턴에 대한 알고리듬 적용 전과 적용 후의 상태 감소 효과를 비교 분석하였으며, 제안된 알고리듬에 의해 생성된 유한 오토마타의 경우 알고리듬 적용 전의 유한 오토마타와 비교해 평균 40.0%의 메모리 감소 효과를 나타냄을 확인하였다. 이러한 검증 결과를 바탕으로 제안된 알고리듬은 네트워크 보안 관리 시스템인 침입 탐지 및 방지 시스템을 비롯해, 실제 정규 표현식을 활용하는 텍스트 필터기 등의 다양한 응용 프로그램에서도 효율적인 자료 구조를 구현하기 위한 방안으로 널리 사용될 수 있을 것으로 기대된다.

참고문헌

[1] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," Theory of machines and computations, New York: Academic Press, pages 15, 1971.

[2] J. E. Hopcroft, R. Motwani and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Pearson/Addison Wesley, pages 535, 2007.

[3] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," Communications of the ACM, pp. 333-340, June 1975.

[4] R. Smith, C. Estan and S. Jha, "XFA: Faster signature matching with extended automata," IEEE Symposium on Security and Privacy, pp. 158-172, May 2008.

[5] N. Hua, H. Song and T.V. Lakshman, "Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection," The 28th Conference on Computer Communications(INFOCOM 2009), pp.415-423, Apr. 2009.

[6] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," The 23th IEEE International Conference on Computer Communications(INFOCOM 2004), pp. 2628-2639, Mar. 2004.

[7] S. Kumar, J. Turner and J. Williams, "Advanced Algorithms for Fast and Scalable Deep Packet Inspection," Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems(ANCS 2006), pp. 81-92, Dec. 2006.

[8] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Tuner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," ACM SIGCOMM'06, pp. 339-350, Sep. 2006.

[9] F.Yu, Z.Chen, Y.Diao, T.V.Lakshman, and R.H.Katz, "Fast and Memory-Efficient Regular Expression Matching For Deep Packet Inspection," Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems(ANCS 2006), pp. 93-102, Oct. 2006.

[10] M. Becchi and S. Cadambi, "Memory-Efficient Regular Expression Search Using State Merging," The 26th IEEE International Conference on Computer Communications (INFOCOM 2007), pp. 1064-1072, May 2007.

[11] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," Proceedings of the 12th IEEE International Conference on Network Protocols(ICNP 2004), pp. 174-183, Oct. 2004.

〔12〕 L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," Proceedings of the 32nd annual international symposium on Computer Architecture(ISCA 2005), pp. 112-122, May 2005.
〔13〕 The Snort Project, snort 2.6 ruleset, VRT Rules 2006-06-28, June 2006.
〔14〕 Deterministic Finite Automatamon, http://en.wikipedia.org/wiki/Deterministic_finite_automaton
〔15〕 D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," SIAM Journal on Computing vol. 6, no. 2, pp. 323-350, Aug. 1977.
〔16〕 K. G. Anagnostakis and E. P. Markatos, "Generating realistic workloads for network intrusion detection system," Proceedings of the 4th international workshop on Software and performance(WOSP'04), pp. 207-215, Jan. 2004.
〔17〕 Y.-H. Choi, M.-Y. Jung and S.-W. Seo, "A fast pattern matching algorithm with multi-byte search unit for high-speed network security," Elsevier Computer Communications, vol. 34, no. 14, pp. 1750-1763, Sep. 2011.
〔18〕 Wu, S. and Manber, U, "A fast algorithm for multi-pattern searching," Department of Computer Science, University of Arizona. TR94-17. pages 11, May 1994.

〈저자소개〉

최 윤 호 (Yoon-Ho Choi) 종신회원
2002년 8월: 경북대학교 전자전기공학부 학사
2008년 8월: 서울대학교 전기컴퓨터공학부 박사
2008년 12월: 서울대학교 전기컴퓨터공학부 박사후연구원
2009년 12월: 펜실베니아 주립대학 박사후연구원
2012년 2월: 삼성전자 책임연구원
2012년 3월~현재: 경기대학교 조교수
〈관심분야〉 침입방지시스템, 소프트웨어 보안, 통신 보안 소프트웨어, 네트워크 보안