실시간 침입 탐지 및 대응을 위한 String Matching 알고리즘 개발*

김주엽° 김준기* 한나래* 강성훈* 이상후* 예흥진**
°아주대학교 정보통신전문대학원, *경기과학고등학교, **아주대학교 정보통신전문대학원
drimtime@korea.com, {daybreaker12, durga21}@hanmail.net, seitero@tokigun.net, cybers@empal.com, , hjyeh@ajou.ac.kr

String Matching Algorithms for Real-time Intrusion Detection and Response

J.Y. Kim° J.G. Kim* N.R. Han* S.H Kang* S.H. Lee* H.J. Yeh**
°Ajou Univ. Graduate School of Information and Communication, *Gyeonggi Science High School,
**Ajou Univ. Graduate School of Information and Communication

요 약

최근 들어 웜 바이러스의 출현과 더불어, 인터넷 대란과 같은 서비스 거부 공격의 피해 사례가 급증하고 있다. 이에 따라 네트워크 보안이 많은 관심을 받고 있는데, 보안의 여러 분야 가운데에서도 특히 침입 탐지와 대응에 관한 연구가 활발히 이루어지고 있다. 또한 이러한 작업들을 자동화하기 위한 도구들이 개발되고 있지만 그 정확성이 아직 신뢰할 만한 수준에 이르지 못하고 있는 것이 지금의 현실이다. 본 논문에서는 이벤트 로그를 분석하여 침입 패턴을 예측하고, 이를 기반으로 자동화된 침입 탐지 및 대응을 구현할 수 있는 String Matching 알고리즘을 제안하고자 한다.

1. 서 론

웜 바이러스의 출현과 더불어, 인터넷 대란과 같은 서비스 거부 공격의 피해 사례가 급증하고 있다. 더욱이 최근 나타난 웜 바이러스의 출현으로 인해, 인터넷 대란과 같은 서비스 거부 공격의 피해 사례가 급증하고 있다. 이에 따라 네트워크 보안이 많은 관심을 받고 있는데, 보안의 여러 분야 가운데에서도 특히 침입 탐지와 대응에 관한 연구가 활발히 이루어지고 있다.

특히 침입의 탐지에 있어서, 관리자가 직접 수집된 이벤트 로그를 분석하여 침입을 탐지하는 것은 매우 비효율적이고 침입과 탐지사이에 시간차가 발생하여 많은 피해가 발생하게 된다. 이러한 이유로 이러한 작업들을 자동화하기 위한 도구들이 개발되고 있지만 그 정확성이 아직 신뢰할 만한 수준에 이르지 못하는 것이 지금의 현실이다.

본 논문에서는 침입 탐지를 자동화하여 수집된 이벤트 로그(Audit Data)를 분석, 침입 패턴을 예측하고 자동화된 대응이 이루어 질수 있도록 String Matching을 이용한 알고리즘을 제안하고자 한다.

본 논문이 제안하는 알고리즘은 일정 기준으로 로그 데이터가 쌓일 때마다 검색을 수행하게 되는데, 검색 범위에 있어서 전체적인 범위는 일정하게 유지한 채 이전에 검색했던 문자열에서 새롭게 추가된 데이터의 범위 내의 쉬프트 되는 window-size 개념을 도입하였다.

또한, 본 알고리즘은 오용 탐지 기법인 Model Based Detection을 기본 탐지 기법으로 삼았는데 그 구성 요소 중에서 지금까지 수집된 이벤트들이 어느 정도 주어진 공격 패턴과 일치하면 침입으로 간주하고 침입 가능 후보로 등록하여 이 결과를 출력하는 *Anticipator*의 기능을 구현하였다. [1]

2장에서는 본 논문이 제안하고자 하는 알고리즘의 정의를 기술하고, 3장에서는 기존의 수집된 이벤트 로그로부터 침입을 탐지해내는 기법들을 알아본다, 4장에서는 본 논문이 제안하는 알고리즘에 대해 알아볼 것이고 5장에서는 파라미터를 통한 성능을 분석하였으며, 마지막으로 6장에서 결론을 맺음으로 본 논문을 마무리하고자 한다.

2. 모델의 설정

* 본 연구는 한국 과학 재단 2003년도 과학고 영재교육 내실화 지원 사업의 창의적 사사연구 지원으로 수행되었음

네트워크 상에서 여러 행동들을 관찰하는 센서가 보내온 사건 정보를 이벤트 라 하고 이벤트 하나를 알파벳 소문자로 보고, 중간에 누락되지 않는다고 할 때 이것들을 시간 순으로 나열하면 하나의 문자열이 된다.

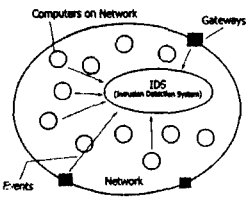


그림 1. IDS Model

2.1 정 의

- *Text* : 어떤 임의의 시간 동안 주어지는 이벤트
- *Window-size* : Text 속에서 우리가 실제 검색을 수행하는 부분의 제한 크기로, 계속적으로 수집되는 이벤트들을 따라 윈도우가 이동하면서 검색을 수행하게 된다. 이 윈도우 안에 분포한 패턴들만이 검색에 있어 유효하다.
- *Interval* : 다음 윈도우로 넘어가기까지 새로 수집되어야하는 데이터의 양. 본 논문에서는 이것을 데이터의 크기로 정의하였다.

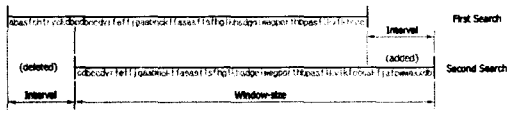


그림 2. Sliding Window 개념

- *단어, 사전* : 단어는 어떤 침입 행동 패턴의 이벤트들을 알파벳 소문자로 표현하여 나열한 순열이고 사전은 단어의 집합이다. 단, 단어들 중에는 어느 한 단어가 다른 단어에 완전히 포함되는 (being a substring) 경우가 없다고 가정한다.
- *패턴* : (단어와 일치하는) Text의 부분문자열(subsequence).

► *Max. Gap* : 어떤 패턴 내의 두 문자 사이에 들어갈 수 있는 최대의 문자 수로 두 이벤트 사이의 최대 시간 간격을 의미한다. 이것은 너무 많은 경우의 수를 방지하고, 두 이벤트가 이만큼 이상의 간격을 가지면 서로 연관성이 없다고 본다.
► *징조* : 단어의 앞쪽 일부분으로, 이것이 Text에서 부분문자열로 나타나면 그 단어의 패턴이 나타날 것이라는 뜻으로 '후보'로 간주한다. 우리는 이것은 단어의 끝 2문자를 뺀 나머지 앞 문자들로 정의하였다. 이것은 후보 단어의 목록을 말하기도 한다.
► *위험수준* : 징조 중에서 남은 2문자 중 앞 1문자가 발견되면 거의 침입 행동이 완성되었다고 보고 한 단계 더 경보 수준을 높이는 것 또는 그렇게 등록된 단어들의 목록을 말한다.
► *예방실패* : 어떤 단어와 완전히 일치하는 패턴이 발견되어 침입 행동이 완성된 상태를 말한다.

2.2 변수의 설정

window-size가 interval의 정수 t배라 하고, d를 maximum gap으로, w(=st)를 window-size, s를 interval, 사전에 있는 최대의 패턴 길이를 l, 그 패턴이 실제 문자열에서 나타나는(중간에 다른 문자가 끼어있는) 길이를 l'이라고 하자. 그럼 다음과 같은 사실들을 알 수 있다.

• 일단, 우리가 찾는 것은 *징조*(최대 가능한 징조길이 $= l-2+(l-3)d$)이지만 *예방 실패* 경우까지 본다면 전체 패턴이 검색될 수 있어야 한다. 또 제한 조건에서 최대 길이의 패턴이 한 윈도우 양 끝에 그 패턴의 양 끝 문자를 두고 있다면 그 윈도우내에서는 예방 실패로 감지되어야 한다. (이 경우 다음 윈도우로 이동시 앞이 잘려 제거된다)
• 따라서 $l \leq l' \leq l+(l-1)d$가 성립하고, 여기서 l'의 최대값을 m으로 놓으면 가능한 모든 경우의 패턴에 대해 matching이 이루어지려면 최소한 $w(=st) \geq m$이어야 한다. 따라서 w에 의해 d는 다음과 같은 범위의 자연수로 결정된다. :

$$d \leq \frac{w-l}{l-1}$$

• 데이터의 모든 문자들을 한 번씩은 검사하기 위해 $s \leq w = st$, $\therefore t \geq 1$이어야 한다.
• $d > s$일수록 sub-window가 서로 중복되는 트리를 가지는 경우가 많아져 비효율적이 된다. 따라서 $d < s$ 여야 한다.

위의 조건들을 모두 만족하면서 [그림 4]처럼 d, s, t가 설정된다면 패턴이 예방실패로 검색되지 않을 수도 있다. 왜냐하면 예방실패로서의 패턴 길이는 제한 조건 안에 들 수 있고 위쪽 윈도우에서 앞쪽 일부를 검색해서 징조나 위험수준으로 탐지가 되었더라도, 아래 윈도우로 넘어갈 때 window 범위를 벗어나 목록에서 제거되기 때문이다. 이 경우는 휴리스틱 알고리즘을 적용하여, 관련 변수들을(s, l, d, t) 실험을 통해 최적화해야 한다. 보통 $s \to 0$, $t \to \infty$일수록 정확해지지만 부하가 많이 걸리므로 적정 수준으로 맞춰야 한다.
여기서 주의해야 할 것은, window 내에서 가장 뒤쪽에 있는 패턴을 찾아야 한다는 것이다. 앞과 뒤에 모두 패턴(징조)이 존재할 경우 앞쪽에 대해 적용하면 d의 제한 조건으로 인해 원래 있던 패턴(징조)을 못 찾는 일이 일어날 수 있기 때문이다. 왜냐하면 기본적으로 발견하는 대로 예방을 위한 대응 조치를 한다고 보기 때문에 한 window 끝에서부터 보았을 때 이미 d를 넘어가 앞에 있는 것들은 예방 처리가 끝났다는 의미가 된다. 따라서 그 앞쪽에 존재하는 패턴의 조각들은 소용이 없게 된다.

3. 기존 알고리즘

3.1 Tree Brute-force Forward

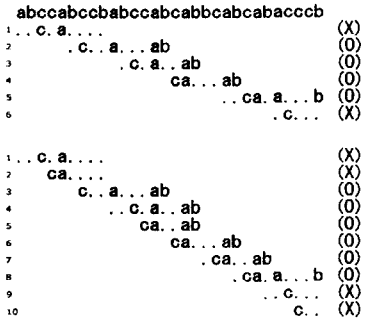


그림 3. Interval에 따른 정확도의 비교.
(위:Interval=5일때, 아래:Interval=3일때)

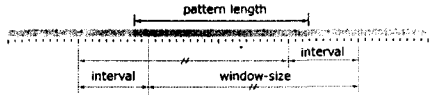


그림 4. 제한조건을 만족하면서 matching 안되는 경우

윈도우가 이동할 때마다 그 윈도우의 앞에서부터 뒤까지 검색을 수행하는 방식이다. 이 방식은 다음 장의 Tree Interval-by-interval에서 설명할 트리 테이블(Tree table)을 하나만 가지고 작동한다. 즉 하나의 윈도우에 대한 트리 테이블을 만드는 것이다. 각 윈도우마다 1번씩 검색이 이루어지되 이전 윈도우의 검색 결과를 활용하지 않는다.
이 방법에서는 *Max. Gap* 제한에 의해 앞에서부터 나타나 후보가 됐던 부분 단어들이 윈도우 끝의 마지막 *Interval* 부분에 해당하는 곳을 검색할 때쯤이면 대부분 삭제된다. 즉, 윈도우 끝의 *Max. Gap* 내에 의미 있는 문자 (여기선 패턴 끝에서 첫 번째, 두 번째, 세 번째 문자)가 나타나지 않으면 그 단어를 후보에서 삭제하는 것이다. 그 앞까지 한 작업들은 의미가 없어지게 된다.

3.2 Tree Brute-force Backward

이것은 윈도우가 이동할 때마다 그 윈도우의 끝(가장 새로운 데이터)부터 맨 앞까지 검색을 수행한다. 마찬가지로 1개의 트리 테이블을 이용하게 되지만, 예방실패, 위험수준, 후보를 순서대로 각각 검색하여 먼저 한 것(예방실패 등의 목록)에 들어 있는 것은 현재 목록에 넣지 않는 방식으로 3번을 검색하게 된다. 예를 들어 단어 abcde가 있다고 했을 때 윈도우 안에서 후보가 될 것을 판단하기 위해서는 먼저 c-b-a 순으로, 위험수준인지 판단하기 위해서는 d-c-b-a 순으로, 예방실패인지 판단하기 위해서는 e-d-c-b-a 순으로 3번의 검색이 이루어 져야 하는데 예방실패로 검색을 해서 e를 발견하지 못했더라도 (e를 못 찾으면 다음 문자인 d를 찾지 않는다) 다시 위험수준 끝 문자인 d가 있는지 검색해 봐야 한다. 후보에서도 같은 경우가 되므로 결국 3번의 검색이 이루어져야만 하는 것이다.

다만 이 경우에 해당 윈도우 맨 끝에서부터 *Max. Gap* 이내에 의미 있는 문자가 나타나지 않으면 검색을 할 필요가 없어 결과적으로는 Backward가 Forward보다는 훨씬 적은 양을 검색하게 된다. 즉 Forward는 이전 정보를 그대로 활용하면서 나아갈 수 있지만 맨 끝 *Max. Gap* 내에서 끝나는 것이 없으면 앞에서 한 작업들이 아무 소용없지는 반면 Backward는 애초에 뒤부터 검색하므로 *Max. Gap*까지만 찾아보고 없으면 다음 *Interval*을 받으면 되는 것이다.

4. Tree Interval-by-interval 알고리즘

본 논문이 제시하는 알고리즘은 brute-force 알고리즘들을 개선하여 윈도우를 *Interval*로 구분한 sub-window의 개념을 사용한다. 즉 트리 테이블이 sub-window 개수만큼 있게 되는데, 먼저 사전의 데이터들을 [그림 5]과 같이 트리 구조로 만든다.[3]

각 트리 항목에는 $t(=w \div s)$개의 정수 2개의 구조체 배열로 구성되어 있는데, 첫 번째 변수에는 그 항목의 깊이(depth)를 표시하고, 두 번째 변수에는 이전 글자에서부터의 거리(distance) (*Max. Gap*를 검사하기 위해)가 저장된다. 이것이 t개로 만들어진 것은 한 윈도우 내의 sub-window 개수가 t이기 때문이고, 다음 윈도우로 넘어가서 새로운 *Interval*이 전달되면 배열의 첫 번째 부분을 삭제하고 거기부터 다시 배열이 채워지게 된다. 이에 대해서는 [그림 6]을 참조하면 된다.

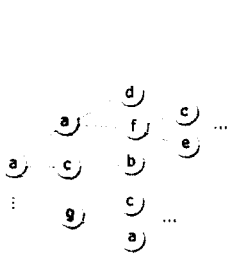


그림 5. 사전의 Rooted Tree

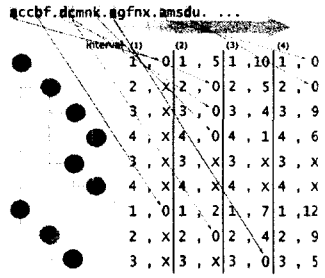


그림 6. Interval-by-interval Matching

[그림 6]을 보면, 단어 adck, adfn, bce로 이루어진 사전이 트리로 표현되어 있다. text를 검색하는데 여기서는 interval이 5, d가 5로 가정되어 있다. 그림에서 4개의 열로 나누어진 것 각각은 interval 크기로 이루어진 sub-window들이다.

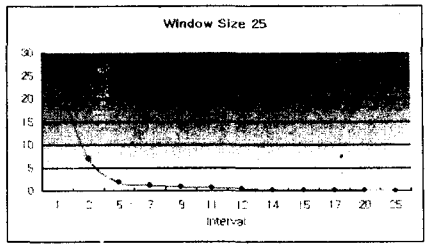
첫 번째 sub-window의 내용을 보면, 각 열에서 첫 번째 수들은 트리에서의 Indent를 나타내는 것으로 트리를 검색할 때 사용된다. 두 번째 수들이 중요한데, 이것들은 d를 검사하기 위해 사용된다. 아직 찾지 못한 상태는 X로 표현되어 있고 찾으면 그 문자에서는 0, 다음 문자로 하나하나 진행할 때마다 1씩 증가되며 다시 그 문자와 같은 문자가 등장하면 0으로 되돌려진다. (되돌려지는 것은 마지막 sub-window의 파란색 화살표에 해당한다) 그림 7에서 0으로 검색되는 부분을 제외한 나머지는 그 sub-window 시작점을 기준으로 썼다.

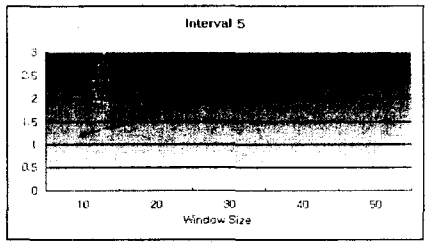
하나의 트리에서 어떤 문자를 찾아야만 그 하위 트리항목의 문자를 찾게 되고, 이때 d를 검사하여 두 번째 숫자가 d 이상이면 (여기서는 d가 7인데 이미 세 번째 sub-window 시작점에서 f 상위 항목의 두 번째 수가 9이기 때문에 f, n은 한번도 검색되지 못하도록 disable 시키고 최초 문자가 다시 나오면 enable 시킨다) 다음 값을 찾더라도 찾았다고 표시하지 않는다. 그림 7의 경우 Fail(예방실패) 패턴으로 adck, bce가 검색되었다.

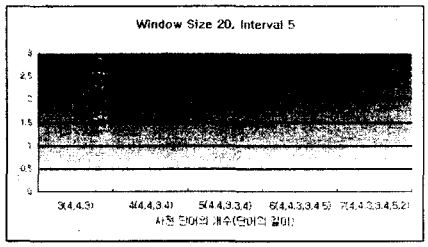
위와 같은 방법으로 첫 윈도우에서 검색이 이루어졌다면, 다음 윈도우로 넘어갈 때(interval만큼 새 데이터를 받았을 때) 첫 번째 sub-window를 지우고 새 데이터에 해당하는 sub-window의 내용이 그 자리를 대신 채우게 된다. 또 다음번 새 데이터가 들어오면 그것은 그림 7의 두 번째 sub-window 자리를 채우는 것이다. 이런 식으로 계속 순환하며 패턴을 찾게 되며, 이것이 바로 이전 윈도우의 검색결과를 활용하여 interval 만큼의 새 데이터만 가지고 계속해서 수행하게 되는 원리이다.

5. 성능 비교 결과

본 장에서는 시스템 파라미터 값의 변화에 따른 Tree Interval-by-interval 알고리즘의 수행 속도 변화를 비교해보도록 하겠다. (여기서 Text는 100개의 단어 나열로 가정하였으며, y축은 수행속도를 의미한다)







개선하고자 만들어낸 Tree Interval-by-interval 알고리즘의 수행 속도를 System Parameter들을 변화시켜 가면서 비교하였다. 전혀 다른 원리로 만들어진 알고리즘들의 성능을 속도로 비교하는 것에는 프로그래밍 기법 등에 따른 차이도 발생할 수 있어 본 논문에서 제안한 하나의 알고리즘 자체의 데이터 량, interval, window size (시험결과, depth는 수행 속도와 관계가 없어 제외)등에 따른 시간과의 상관관계를 조사하였다. (단, 이때 최소한의 제한 조건들은 만족하는 범위에서 수행하였다.)

6. 결 론

성능 비교 결과를 토대로, Tree Interval-by-interval 알고리즘에서는 d가 커짐에 따라 패턴의 일치 되는 정확성이 커지고, w가 일정하고 s가 커짐(t가 작아짐)에 따라 로그함수의 형태로 수행 속도가 빨라지며, w가 커짐에 따라(s는 일정하고 t가 커짐) 탐색속도는 비교적 큰 비율로 늘어난다, 또한 사전의 단어수가 많아짐에 따라 속도는 점진적으로 늘어나고, 입력되는 data의 량이 많아짐에 따라 수행속도의 변화는 거의 같은 비율로 늘어나고 있음을 알 수 있었다.

7. 참고 문헌

[1] C. Jason Coit, Stuart Staniford and Joseph McAlerney, Towards Faster String Matching for Intrusion Detection http://www.silicondefense.com/software/acbm/index.htm

[2] Dan Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, pp16-28, 1997

[3] Thomas H. Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein, Introduction To Algorithms (Second Edition), MIT Press, pp 200-214, 350-356, 906-932, 2001