네트워크 침입방지 시스템을 위한 고속 패턴 매칭 가속 시스템

김 선 일†

요 약

패턴 매칭(Pattern Matching)은 네트워크 침입방지 시스템에서 가장 중요한 부분의 하나며 많은 연산을 필요로 한다. 날로 증가되는 많은 수의 공격 패턴을 다루기 위해, 네트워크 침입방지 시스템에서는 회선 속도로 들어오는 패킷을 처리 할 수 있는 다중 패턴 매칭 방법이 필수적이다. 본 논문에서는 현재 많이 사용되고있는 네트워크 침입방지 및 탐지 시스템인 Snort와 이것의 패턴 매칭 특성을 분석한다. 침입방지 시스템을 위한 패턴 매칭 방법은 다양한 길이를 갖는 많은 수의 패턴과 대소문자 구분 없는 패턴 매칭을 효과적으로 다룰 수 있어야 한다. 또한 여러 개의 입력 문자들을 동시에 처리 할 수 있어야 한다. 본 논문에서 Shift-OR 패턴 매칭 알고리즘에 기반을 둔 다중 패턴 매칭 하드웨어 가속기를 제시하고 여러 가지 가정 하에서 성능 측정을 하였다. 성능 측정에 따르면 제시된 하드웨어 가속기는 현재 Snort에서 사용되는 가장 빠른 소프트웨어 다중 패턴 매칭 보다 80배 이상 빠를 수 있다.

A High-speed Pattern Matching Acceleration System for Network Intrusion Prevention Systems

Sunil Kim†

ABSTRACT

Pattern matching is one of critical parts of Network Intrusion Prevention Systems (NIPS) and computationally intensive. To handle a large number of attack signature patterns increasing everyday, a network intrusion prevention system requires a multi pattern matching method that can meet the line speed of packet transfer. In this paper, we analyze Snort, a widely used open source network intrusion prevention/detection system, and its pattern matching characteristics. A multi pattern matching method for NIPS should efficiently handle a large number of patterns with a wide range of pattern lengths and case insensitive patterns matches. It should also be able to process multiple input characters in parallel. We propose a multi pattern matching hardware accelerator based on Shift-OR pattern matching algorithm. We evaluate the performance of the pattern matching accelerator under various assumptions. The performance evaluation shows that the pattern matching accelerator can be more than 80 times faster than the fastest software multi-pattern matching method used in Snort.

키워드 : 네트워크 침입방지 시스템(network intrusion prevention systems), 패턴 매칭(pattern matching), 컴퓨터 구조(computer architecture)

1. 서 론

최근 전 세계적으로 사용자가 폭발적으로 증가하고 있는 인터넷은 가정에서의 웹 서핑에서부터 소리바다와 같은 P2P 응용, 멀티미디어 정보의 전송, 다양해지고 더욱 활성화 된 전자 상거래 등으로 인해 네트워크 트래픽의 급증을 가져왔다. 네트워크의 이용 확대에 대처하기 위해 현재 기간망 네트워크의 대여폭이 이미 수 기가비트급으로 확장됐고, 이러한 네트워크 속도 및 대역폭의 증가 추세는 기가비트 이더넷(ethernet)과 VDSL과 같은 고속 가정용 인터넷 연결의 보편화로 더욱 가속화 되고 있다. 인터넷의 확장과 네트워크 속도의 증가로 네트워크를 통한 분산 서비스거부 공격(DDoS), 인터넷 웜(internet worm), 이메일 바이러스등의 악의적인 공격도 빨라지고 급속히 증가 되고, 이로 인한 피해도 급격히 증가되는 추세이다. 예를 들면 Code Red Worm [1]이나 SQL Slammer Worm [2]은 수 시간에서 수 분만에 전 세계로 퍼져서 네트워크를 마비시며 막대한 피해를 입혔다.

네트워크 시스템에 대한 악의적 공격에 대한 방어로 기존에는 방화벽(firewall)과 네트워크 침입탐지 시스템(NIDS: Network Intrusion Detection Systems)에 의존하였다. 방화벽은 특정 포트로 들어오는 공격은 막을 수 있으나 공개된 포트를 사용한 공격에는 대응 할 수 없는 취약점을 갖고 있다. NIDS는 이러한 취약점을 보완하여 패킷의 헤더뿐만이

※ 이 논문은 2003학년도 홍익대학교 교내연구비에 의하여 지원되었음.
† 종신회원 : 홍익대학교 정보컴퓨터공학부 교수
논문접수 : 2005년 1월 19일, 심사완료 : 2005년 3월 20일

아니라 데이터(payload)도 조사(deep packet inspection)함으로써 공격이 발생했는지를 탐지 할 수 있다. 하지만 NIDS는 네트워크 주 경로 옆에 설치되어 들어오는 패킷을 직접 막지 않고 통과시키기 때문에 침입이 발견되었을 때에는 공격하는 패킷은 이미 호스트에 도달된 뒤 일 수 있다는 문제점이 있다. 최근 이런 문제점을 보완하고 능동적으로 악의적 공격 방지하는 네트워크 침입방지 시스템(NIPS: Network Intrusion Prevention Systems)이 등장하였다[3,4]. 네트워크 침입방지 시스템은 침입탐지 시스템과 달리 주요 네트워크의 경로 중간에 위치하여 공격 패킷을 직접 조사하고 차단한다. 네트워크 침입방지 시스템은 스위치, 방화벽이 발전된 형태나 기존 NIDS 시스템이 인라인 모드(in-line mode), 즉 네트워크 경로의 중간에 위치하도록 변경된 형태 등으로 존재한다. 실제로 수행하는 일은 패킷을 조사하고 악의적 공격인지를 판단하는 점에서는 NIDS와 유사하나 발견된 공격 패킷을 네트워크로부터 격리 제거한다는 점이 다르다. 네트워크 침입방지 시스템에서 가장 중요한 기능의 하나는 패킷의 데이터를 조사하여 정확히 공격 패킷을 찾아내고 이러한 작업의 수행을 연결된 회선 속도로 하는 것이다.

패킷 데이터 조사에서는 이미 알려진 공격 패턴(attack signature)을 패킷의 데이터에서 패턴 매칭(pattern matching)을 통해 찾는다. 이 패턴 매칭 과정은 침입을 탐지하는 과정 중에서도 가장 많은 계산을 요구한다. 예를 들면 NIDS이며 동시에 인라인 모드로 동작하여 NIPS기능을 수행 할 수 있는 Snort [5]에서 패턴 매칭 수행 부분은 전체 수행에서 70~80%의 부분을 차지한다[6]. 모든 공격 패턴에 대해 정확하게 패킷의 데이터에서 일치하는 패턴을 찾을 뿐 아니라, 수행 속도 또한 회선 속도를 능가 할 수 있는 패턴 매칭 방법은 NIPS에 있어서 절대적이다. 최근 네트워크 속도의 증가로 인해 NIPS가 설치될 간선 네트워크의 회선 속도가 기가비트급으로 증가되고 있다는 것을 고려할 때, 이와 같은 패턴 매칭 방법을 개발 하는 것은 아주 어려운 일이다. 현재까지 많은 패턴 매칭 방법들이 연구되고 제시 되었지만 일부만이 NIPS에서 사용 될 수 있고 그것도 대부분 하드웨어로의 구현 되고 있다. 소프트웨어로 패턴 매칭 알고리즘을 구현하는 방법 [7-11] 들은 회선의 속도가 10Gbps가 넘어가고, 공격 패턴의 수가 증가 할수록 NIPS에서 필요로 하는 속도를 맞추기 힘들다.

하드웨어로 패턴 매칭을 구현 하는 방법 [12-19] 들은 이러한 속도의 한계를 극복 할 수 있다. 현재 제시되고 있는 방법들 중에 하나 또는 여러 개의 공격 패턴에 대하여 정규표현식(regular expression)을 만들고 이를 바탕으로 결정적 또는 비결정적 유한 오토마타(NFA/DFA)를 FPGA하드웨어로 직접 구성하는 방법이 있다 [12-14]. 이 방법에서는 패킷의 데이터가 한 바이트씩 차례대로 모든 유한 오토마타 회로에 동시에 전달되어야 한다. 패턴이 많을 경우 브로드캐스트(broadcast) 되는 회로가 증가되고 따라서 출력 분기수(fan-out)가 늘어난다. 또한 상태 전이를 위한 조합 논리회로 (combinatorial logic circuits)가 증가함에 따라 속도의 저하를 가져온다. 다른 하드웨어 방법으로는 CAM(Content Addressable Memory) 또는 비교기(Comparator)와 논리 회로를 사용하는 것이 있다[15-18]. 이 방법 또한 패턴의 수가 늘어날수록 브로드캐스트에 따른 부하와 논리회로의 증가로 인한 속도의 저하를 가져 올 수 있다. 이들 두 가지 하드웨어 방법은 새로운 공격 패턴이 추가 될 때 마다 FPGA의 재 프로그램이 필요하다. 따라서 고속으로 항시 작동해야 하는 NIPS에는 적합하지 않다.

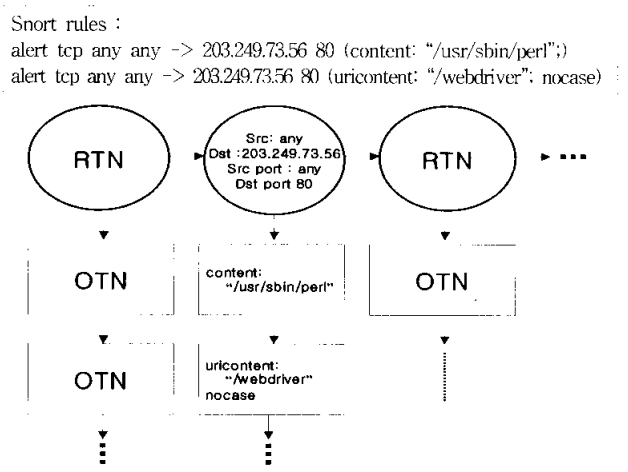
또 다른 방법으로는 Blooming filter를 사용하는 방법[19]이 제시되고 있다. 이 방법에서는 공격 패턴이 바뀌어도 다시 회로를 재구성 할 필요는 없다. Blooming filter는 다수의 해쉬(hash) 함수를 사용하여 같은 길이의 패턴들에 대한 비트 벡터를 만들고, 입력 데이터의 문자열에 해쉬 함수를 결과가 이들 비트 벡터 패턴과 일치 하는지를 확인한다. 만약 하나의 비트 값이라도 다르게 되면 입력 문자열이 패턴들과 일치 하지 않는 것이다. Blooming filter를 이용한 방법에서는 서로 다른 공격 패턴의 길이마다 하나의 Blooming filter가 필요하다. 또한 이들 여러 Blooming filter들이 동시에 패킷의 데이터를 읽어야 하기에 때문에 multi-port 메모리가 필요하다. 실제로 Snort를 분석하여 보면 공격 패턴의 길이가 매우 다양하다. 따라서 적지 않은 수의 Blooming filter와 메모리의 많은 포트가 필요하게 된다. Blooming filter의 다른 단점은 오탐지(false-positive) 결과를 낼 수 있다는 것이다. 즉 실제 데이터는 패턴과 일치하지 않는데 일치하는 것으로 결과가 나오는 것이다. 이것 때문에 패턴 매칭이 발견된 후에도 다시 다른 방법을 통해 정확한 매칭을 확인 하여야 한다.

본 논문에서는 기가비트급의 빠른 속도 요구를 만족하고 기존의 하드웨어 패턴 매칭 방법의 단점을 극복할 수 있는 하드웨어 기반의 패턴 매칭 방법을 제시한다. 이 방법은 잘 알려진 패턴 매칭 방법인 Shift-OR [20]에 기반을 두고있다. 이 알고리즘에서는 주로 SHIFT 와 OR 비트 연산이 사용되고 있으며, 이들 연산은 하드웨어로 효과적으로 구현 될 수 있다. 이 알고리즘은 또한 다중 패턴 매칭을 위해 확장 될 수 있다. 범용 프로세서에서 소프트웨어로 구현 되었을 때는 한 번에 처리 되는 양이 레지스터의 크기에 제한을 받기 때문에 전체 패턴의 크기가 큰 경우에는 매우 느리게 동작하게 된다. 본 논문에서는 다중 패턴을 위한 Shift-OR 알고리즘을 효과적으로 구현하는 패턴 매칭 아키텍쳐를 제안하고, 또한 한 번에 여러 개의 입력 문자를 처리 할 수 있도록 확장 한다. 본 논문은 다음과 같이 구성 된다. 서론에 이어 2장에서는 Snort를 분석하여 NIPS에서 패턴 매칭의 특성을 알아 본다. 3장에서는 간단히 Shift-OR 알고리즘에 대해 설명하고 4장과 5장에서는 각 각 제시된 패턴 매칭 하드웨어의 구조를 설명하고 성능을 평가 한다. 마지막으로 6장에서 논문의 결론을 맺는다.

2. Snort의 구조와 패턴 매칭

Snort는 규칙(rule)을 사용하여 패킷의 시그내쳐와 해당

패킷이 발견 되었을 때 수행 되어야 하는 동작을 정의하고 있다. (그림 1) 에는 Snort에서 사용되는 규칙이 사각형안에 보여지고 있다. Snort의 규칙은 두 부분으로 나누어 진다. 규칙 헤더와 규칙 옵션이 그것이다. 규칙 헤더는 규칙 문장에서 처음부터 왼쪽 괄호가 나오기 전까지의 부분을 말하며, 괄호 안에 들어 있는 것은 규칙 옵션이다. Snort는 이들 규칙들을 파싱하여 2차원의 링크드 리스트를 구성한다. 이 링크드 리스트는 (그림 1) 에서 처럼 Rule Tree Node(RTN)와 Option Tree Node(OTN)로 구성 되었다. 규칙 헤더와 규칙 옵션은 기본적으로 RTN과 OTN에 각각에 매핑 되어 있다. RTN은 여러 규칙이 공통적으로 가질 수 있는 조건들, 예를 들면 패킷의 발신지와 목적지 주소 및 포트, 패킷의 전송 방향, 프로토콜 타입 등을 담고있다. OTN은 각 규칙에 더해질 수 있는 여러 가지 옵션들, 예를 들면 TCP flag, ICMP코드와 타입, 패킷의 데이터에서 찾아야 할 문자열 패턴 등에 대한 정보를 가지고 있다. (그림 1)은 사각형안에 보여지고 있는 두개의 규칙을 포함하여 어떻게 RTN과 OTN이 구성 되는가를 나타내고 있다. 두개의 규칙은 규칙 헤더 부분이 같기 때문에 같은 RTN 밑에 서로 다른 OTN을 형성하고 있다. 패킷이 도착하면 Snort 는 이들 패킷을 RTN과 OTN 링크드 리스트를 따라가며 차례로 조사하게 된다.



(그림 1) RTN and OTN 링크드 리스트 구조와 Snort 규칙

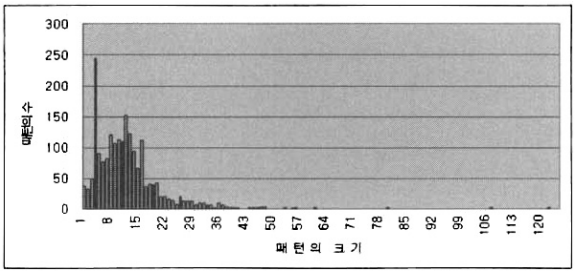
Snort에서 RTN과 OTN을 하나씩 조사하는 것은 많은 시간을 요구하게 된다. 특히 OTN에 있는 'content' 옵션은 패킷에서 찾아야 할 문자열 패턴을 가지고 있는데 이들 문자열 패턴에 대한 조사를 하나씩 하는 것은 많은 시간을 요구한다. Snort 2.0에서부터는 규칙들이 프로토콜과 발신지와 목적지의 포트 번호로 규칙 집단(ruleset)을 이루고 다중 패턴 매칭 기법을 사용하여 규칙과 일치하는 패킷을 찾는 과정의 속도를 향상 시키었다. 패킷이 도착하면, 패킷의 프로토콜과 발신지와 목적지 포트 번호를 가지고 알맞은 규칙 집단을 찾고 그 규칙 집단에 있는 모든 'content' 옵션의 문자열 패턴에 대하여 다중 패턴 매칭 방법을 사용하여 조사를 한다. 만약 매칭되는 패턴이 발견되면 그 패턴을 갖고 있는 모든 규칙에 대하여 RTN과 OTN에 차례로 조사된다. 만약 매칭되는 패턴이 발견 되지 않으면 그 패킷은 'content' 옵션이 없는 모든 규칙에 대하여 조사 된다. 이 방법에서는 패턴이 일치 하지 않은 'content' 옵션을 갖는 다른 모든 OTN을 조사에서 배제 함으로써 Snort의 공격 패킷 발견 속도를 월등히 향상 시켰다 [5]. 다중 패턴 매칭 방법은 Snort의 성능에 큰 영향을 미치는데 주로 사용되는 방법은 Aho-Corasick [10] 알고리즘에 기반을 둔 *AC* 방법과 Wu-Manber [11] 알고리즘에 기반을 둔 *MWM* 방법 있다 [9].

다음은 Snort에서 발견되는 패턴 매칭의 특성을 분석함으로써 침입방지 시스템에서의 패턴 매칭 특성에 대하여 알아본다. Snort의 패턴 매칭 특성 중 가장 중요한 것 중의 하나는 문자열 패턴의 수 이다. Snort 2.1.2 패키지의 규칙에는 무려 1900이상의 문자열 패턴이 있다. 실제 패턴 매칭이 수행 될 때는 각 규칙 집단에 속한 패턴만 조사 되기 때문에 한번에 조사 되어야 할 패턴의 수가 줄어드는 효과가 있다. 하지만 HTTP 규칙 집단 (프로토콜 TCP, 포트 번호 80) 에 속한 문자열 패턴은 무려 800 여 개가 넘는 등, 많은 수의 패턴을 조사하여야 하며, 앞으로도 패턴의 수는 계속 증가하리라 생각된다. 다른 중요한 특성은 패턴의 크기이다. (그림 2) 에서 보듯이 패턴 크기는 1에서 122 까지 변하고 있다. 작은 패턴의 사이즈는 MWM 같은 소프트웨어 방식의 패턴 매칭 방법에 영향을 미친다. MWM에서 사용되는 bad character shift 방식은 입력 데이터에서 패턴에 있지 않는 문자를 발견했을 때 최대 패턴의 길이 만큼 비교하지 않고 처리 할 수 있다. 하지만 많은 규칙 집단에서 길이가 1인 패턴이 포함 되어 있어 MWM 방법에서 입력 데이터를 한 문자 씩 밖에 처리하지 못 한다.

Snort의 규칙에서 패턴 매칭은 'content'와 'uricontent' 옵션에서 사용된다. 이들 옵션은 패킷에서 발견 하여야 할 문자열 패턴에 대한 정보를 가지고 있고 그와 함께 'nocase' 옵션이 사용 될 경우 대소문자 구분 없는 패턴 매칭을 수행하여야 한다. Snort에서 사용되는 다중 패턴 매칭 방법들은 직접 대소문자 구분 없는 경우를 다룰 수 없기 때문에 모든 찾고자 하는 문자열 패턴을 일단 대문자로 변환하여 저장하고, 패킷의 데이터도 대문자로 변환 한 뒤 패턴 매칭을 수행하게 된다. 패턴이 매칭 된 뒤에도 다시 원래 패킷의 데이터와 원 문자열 패턴이 다시 비교된다. 이러한 과정은 패킷 데이터 한 문자 마다 최소한 두 번의 메모리 접근을 필요로 하고, 이에 따라 추가되는 수행 시간은 수 기가비트의 회선 속도를 지원하는 침입방지 시스템에서는 무시 못할 시간 이다.

패킷의 처리 속도를 높이기 위해서는 패턴 매칭 방법은 동시에 여러 개의 데이터 문자들을 처리 할 수 있어야 한다. 만약 입력되는 문자를 하나씩 밖에 처리 할 수 없다면 패킷을 처리 하는 속도는 최악의 경우 각 문자의 처리속도와 패킷의 길이에 의해 정해지고, 결국 빠른 CPU 속도와 메모리 접근 속도에만 의존하게 된다. 현재의 VLSI기술로도 메모리 접근 속도는 느리게 발전 하고있어 네트워크 회선 속도의 빠른 향상을 따라올 수 있을 지 의문시 된다. 따라서 패턴

매칭 방법에서 효율적으로 여러 개의 문자들을 동시에 처리해 주는 것이 필요하다.



(그림 2) 패턴의 길이와 Snort 규칙

3. Shift-OR 패턴 매칭 알고리즘

여기서는 우선 하나의 패턴에 대한 Shift-OR 패턴 매칭 알고리즘[20]에 대하여 간단히 소개한다. Shift-OR 알고리즘은 비트 연산에 기본을 두고 있고, 크기가 m (패턴의 길이)인 패턴 매칭의 상태를 나타내는 비트 벡터인 상태 벡터 R과 패턴에서 특정 문자 c 의 위치를 나타내는 문자위치 벡터 S_c 가 있다. 예를 들면, 패턴 $P = p_0 \cdots p_{m-1}$ 와 입력 문자열 $X = \cdots x_{i+j} \cdots$. 이 있으면, x_{i+j} 를 처리하고 난후 $x_i \cdots x_{i-j}$ 가 $p_0 \cdots p_j$, 와 일치 하면 R의 j 번째 비트, $R[j] = 0$ 이고, 그렇지 않으면 $R[j] = 1$이다. 또 문자위치 벡터의 경우 $p_i = c$ 이면 $S_c[i] = 0$이고 그렇지 않으면 $S_c[i] = 1$로 설정된다. 입력 문자 x_{i+j} 를 처리 후 $R[j]$ 값을 얻으면 다음 입력 문자 x_{i+j+1} 가 패턴의 p_{j+1} 위치에 나타나는가를 가지고 $R[j+1]$를 계산할 수 있다. $R[j+1]$ 값은 다음 수식과 같이 정의 된다.

$$R[j+1] = \begin{cases} 0 & R[j] = 0 \text{ and } S_c[j+1] = 0 \quad \text{where } c = x_{i+j+1} \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

$$R[0] = S_c[0] \quad where \ c = x_{i+j+1} \tag{2}$$

$R[m-1] = 0$은 입력 문자열 $x_i \cdots x_{i+m-1}$ 가 패턴의 $p_0 \cdots p_{m-1}$ 와 일치 했다는 것을 나타낸다. 즉 패턴과 일치하는 문자열을 발견한 것이다. 상태 비트 벡터 R 계산하는 것은 위의 식에서 볼 수 있듯이 간단한 Shift와 OR 비트 연산으로 구성된다. 즉 상태 벡터 R를 한번 Shift 한 뒤, 문자 위치 벡터 S_c 와 OR 연산함으로써 위의 식을 만족하는 새로운 상태 벡터 R을 계산 하게 된다.

Shift-OR 알고리즘은 쉽게 패턴에 문자 집합, 부정 문자(complement symbol), don't care 문자가 있는 경우를 다룰 수 있다. 만약 패턴의 위치 i 에 문자 집합 {x, y, z} 가 매칭 될 수 있다면 $S_x[i] = S_y[i] = S_z[i] = 0$ 로 설정 함으로써 이 경우를 처리 할 수 있다. 부정 문자나 don't care 경우도 같은 방법으로 처리 될 수 있다. 따라서 대소문자 구분 없는 패턴 매칭의 경우 다른 오버헤드 없이 문자위치 벡터의 설정만으로 처리된다. 또한 Shift-OR 알고리즘은 쉽게 다중 패턴을 처리하게 확장 될 수 있다. 각 패턴에 대한 상태 벡터 R을 패턴의 순서대로 합치고 또 문자위치 벡터 S 도 패턴의 순서대로 합친 뒤 이 합쳐진 상태 백터와 문자 위치 벡터에 Shift-OR 비트 연산을 적용하면 된다. 단지 각 패턴의 첫 번째 위치에 해당하는 R의 비트 값 $R[i]$를 계산 할 때 $R[i-1]$ 은 무시하고 계산하는 것이 다르다.

소프트웨어로 Shift-OR알고리즘을 구현 하였을 때 비트 벡터들의 크기가 커질수록 수행이 비효율적이 된다. 특히 패턴이 많아지면 비트 벡터들의 크기가 커지기 때문에 메모리의 접근과 수행 되어야 하는 명령어 수가 크게 늘어 나게 된다. 다음 장에서는 Shift-OR 알고리즘의 소프트웨어 구현이 갖는 문제점을 극복하고 입력되는 문자들을 동시에 여러 개를 처리할 수 있는 패턴 매칭 하드웨어의 구조에 대하여 기술한다.

4. 침입 방지시스템을 위한 하드웨어 가속기의 구조

패턴 매칭 하드웨어 가속기 (PMA)는 다음과 같은 네 부분으로 구성되어 있다: **문자 위치 벡터 장치, 상태 벡터 계산 장치, 매칭 발견 장치, 우선순위 인코더(priority encoder).** 문자 위치 벡터 장치는 N 개의 입력 문자를 입력 버퍼로부터 읽어 N 개의 문자 위치 벡터를 만들어 낸다. 입력 버퍼는 패킷의 데이터(payload)를 저장하고 있다. N 은 PMA의 *shift size* 이다. 즉 N개의 입력 문자가 동시에 처리 된다. (그림 3)은 문자 위치 벡터 장치의 구조를 보여 주고 있다. 문자 위치 벡터 장치는 N개의 문자 위치 벡터 테이블(Character Position Vector Table)을 가지고 있고 각 테이블은 각 문자당 하나씩, 256개 문자 위치 벡터를 가지고 있다. 테이블에 저장된 문자 위치 벡터는 모든 패턴을 위한 문자 위치 벡터가 합쳐진 것이고 Snort규칙의 모든 문자열 패턴으로부터 미리 계산 되어 테이블에 저장된 것이다. 모든 테이블은 같은 문자 위치 벡터들을 저장 하고 있으며, i 번째 테이블은 i 번째 입력 문자를 입력으로 받아 i 번째 문자 위치 벡터 S_i를 (그림 3)에서 처럼 만들어 낸다.

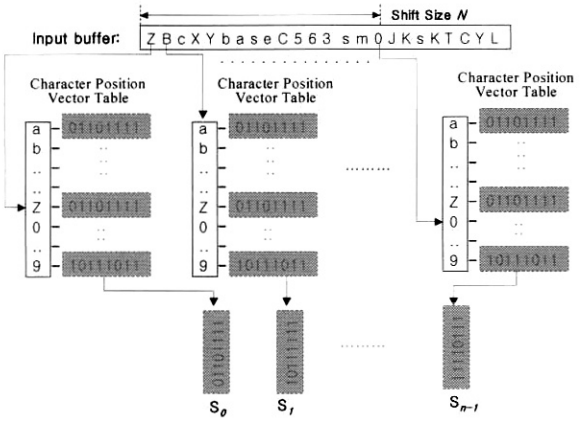
문자 위치 벡터 장치에서 만들어진 N 개의 문자 위치 벡터 $S_0, S_1, ..., S_{N-1}$는 상태 벡터 계산 장치로 전달 된다. 상태 벡터 계산 장치는 문자 위치 벡터들을 받아 그전에 계산 되었던 상태 벡터(State Vector)와 패턴 경계 벡터(Pattern Boundary Vector) B를 가지고 새로운 상태 벡터 R을 계산 한다. 패턴 경계 벡터는 R과 같은 크기의 벡터로, 비트 값 0으로 각 패턴의 경계를 표시하는 벡터이다. (그림 4)는 상태 벡터 계산 장치의 수행 과정을 보여 주고 있다. 처음에는 R과 B 가 AND 되고 SHIFT된 뒤에 S_0 와 OR 하여 중간 결과 벡터인 T_0 를 만든다. 다음 T_0 은 B와 AND 되고 SHIFT된 뒤 S_1 와 OR 하여 중간 결과 벡터인 T_1 를 만든다. 같은 방법의 계산이 각 단계에서 반복되어 T_{N-1}이 만들어 질 때 까지 수행 된다. T_{N-1}은 다시 R 에 저장 되어 다음 계산 주기(cycle)에 사용 되게 된다. B벡터와 AND를 수행하는 이유는 계산의 결과가 패턴의 경계를 넘어 퍼져나

가지 못 하도록 하는데 있다. 이 계산 과정들은 다음 수식으로 표현 된다.

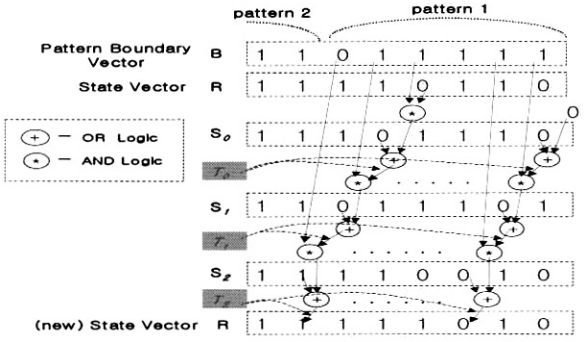
$$T_k(0) = S_k(0) + 0 = S_k(0) \qquad for\ all\ k \quad (3)$$

$$T_0(i) = S_0(i) + (R(i-1) * B(i-1)) \qquad for \quad i > 0 \quad (4)$$

$$T_k(i) = S_k(i) + (T_{k-1}(i-1) * B(i-1)) \qquad for \quad k > 0, i > 0 \quad (5)$$



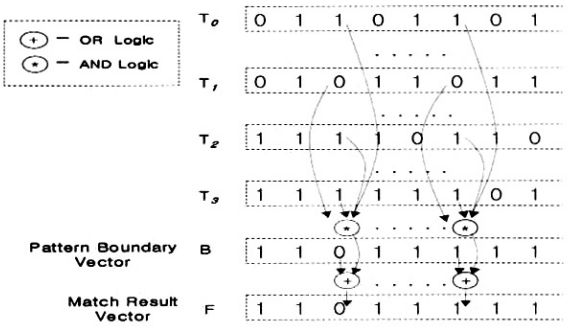
(그림 3) 문자 위치 벡터 장치



(그림 4) 상태 벡터 계산 장치

(그림 4)에서 보듯이 Shift의 수행은 단지 ith 위치의 결과를 i+1th 위치의 OR 게이트의 한 쪽 입력에 연결 함으로써 이루어 진다. 각 단계의 계산은 Shift-OR 알고리즘에서 하나의 Shift-OR 연산과 동일 하다. 상태 벡터 계산 장치는 N 번의 Shift-OR 연산을 한 주기에 처리 할 수 있다. 모든 연산 과정은 조합 회로(Combinatorial circuit)로 구성 되며 중간 계산 결과 $T_0, \cdots T_{N-1}$은 순간적으로 만들어 저장할 필요 없이 사용된다.

N 개의 입력 문자 열이 처리되는 동안, 매칭 되는 문자열이 발견 될 수 있다. PMA의 세 번째 구성 장치인 매칭 발견 장치는 모든 중간 결과 벡터로부터 모든 매칭을 발견 할 수 있다. (그림 5)는 매칭 발견 장치의 구조를 보여 주고 있다. 모든 중간 결과 벡터의 같은 위치에 있는 비트들이 AND 된 뒤에 패턴 경계 벡터 B의 같은 위치의 비트와 OR 된다. 이 결과는 매칭 결과 벡터(Matching Result Vector) F 에 기록 된다. 매칭 결과 벡에 0값을 갖는 비트(**매칭 비트**)가 있으면 이것은 패턴이 매칭 되었다는 것을 의미 한다. 매칭 결과 벡터 F로부터 PMA의 네 번째 구성 요소인 우선 순위 인코더가 첫 번째 매칭 비트의 인덱스 값을 계산해 낸다. 우선 순위 인코더의 결과는 소프트웨어에 의해 읽혀지고 매칭된 패턴을 알아내기 위해 사용된다. 매칭 발견 장치나 우선 순위 인코더는 패턴과 일치되는 문자열을 발견하였을 경우 PMA로부터 어떤 정보를 소프트웨어가 원하는가에 따라 다르게 설계 될 수 있다. 본 논문에서 제시된 구조는 소프트웨어가 단지 매칭이 있는지 그리고 매칭이 있는 경우에 매칭 결과 벡터에서 매칭 비트의 위치만을 알 필요가 있다고 가정하였다.



(그림 5) 매칭 발견 장치성능평가

5. 성능평가

본 논문에서 제시된 패턴 매칭 하드웨어 가속기(PMA)의 성능을 시뮬레이션 모델을 이용하여 평가하고 Snort에서 사용되는 소프트웨어 다중 패턴 매칭 방법들, *AC*와 *MWM*과 비교 한다. 이들 AC와 MWM은 각 각 Aho-Corasick 과 Wu-Manber 알고리즘에 기반을 두고 있다. PMA 가 Snort에서 사용되는 소프트웨어 패턴 매칭 방법 대신에 사용되고 있다고 가정하고 실제 Snort가 수행 중 얻어진 데이터를 바탕으로 성능을 평가한다. Snort 수행에 사용된 PMA 시뮬레이터는 앞의 절에서 설명된 PMA구조를 시뮬레이션 한다. PMA와 관련된 여러 가지 수행 중 문자 위치 테이블을 채우고 패턴 경계 벡터를 설정하는 것과 같은 과정은 시간상 중요하지 않기에 평가에서 제외되고 패턴 매칭에 직접 관계되는 수행 과정만이 평가 한다.

본 논문에서는 PMA 수행에 관계되는 명령어를 먼저 정의하고 이들 명령어들의 수행 시간을 분석하였다. 하나의 **패턴 매칭 연산**은 다음과 같이 수행된다고 정의한다. N개의 문자가 입력 버퍼로부터 읽혀지고, 그 다음 문자 위치 벡터 테이블에서 해당 문자 위치 벡터가 읽혀져 N 개의 중간 결과 벡터가 계산된다. 그리고 마지막으로 패턴 경계 벡터와 N개의 중간 결과 벡터로부터 매칭 결과 벡터가 만들어진다. 한번의 패턴 매칭 연산에 걸리는 시간은 실제 하드웨어 구현에 사용되는 기술에 따라 달라질 수 있다. 성능 평가를

위하여 본 논문에서는 패턴 매칭 연산에 사용되는 시간을 고정하지 않고 여러 가지로 변화 하였다. 수행 시간은 클락 주기 (clock cycle)로 측정되었다. <표 1>은 패턴 매칭에 사용되는 PMA 명령어를 보여주고 있다.

〈표 1〉 PMA 명령어

명령어	수행시간 (cycles)	설 명
PMA_reset	1	PMA의 모든 내부 벡터들을 초기화 한다
PMA_pattern_search	데이터에 따라 변함	패턴 매칭이 발견되기까지 패턴 매칭 연산을 수행한다.
PMA_match_position	n	우선 순위 인코더를 통해 첫 번째 매칭 비트의 위치를 읽고 해당 비트를 초기화 한다

패턴 매칭이 시작 될 때, 상태 벡터와 매칭 결과 벡터를 포함한 모든 벡터들은 PMA_reset 명령어에 따라 초기화 된다. PMA가 초기화 된 뒤 패턴 매칭은 PMA_pattern_search 명령어에 의해 시작되고 이 명령어의 수행은 패턴 매칭이 발견되어 그 결과가 패턴 매칭 결과 벡터에 기록 될 때 까지 계속된다. 패턴 매칭이 발견되어 PMA_pattern_search 명령어가 수행을 멈추게 되면 소프트웨어가 PMA_match_position 명령어를 수행하여 매칭 결과 벡터로부터 첫 번째 매칭 비트의 위치를 읽어낸다. PMA_match_position 명령어는 우선 순위 인코더의 결과를 읽고 매칭 결과 벡터에서 해당 매칭 비트를 초기화 하여 다음 PMA_match_position 명령어 수행 때는 다음 매칭 비트의 위치를 알 수 있게 한다. 매칭 비트의 위치로부터 매칭이 된 패턴을 알아낸 뒤, 소프트웨어는 다른 필요한 연산을 하여 해당 패킷이 규칙의 조건과 일치하는 지를 판단한다. 모든 매칭 비트가 다 처리되면, 다시 PMA_pattern_search 명령어가 수행되어 다음 패턴 매칭을 발견하게 된다.

성능 평가에서 PMA_reset 명령어는 모든 내부 벡터들을 초기화 하는 수행 만 하기 때문에 1 사이클(cycle)동안에 수행 된다고 가정 하였다. PMA_pattern_search 명령어의 수행 시간은 매칭이 발견되기 전까지 수행 되어야 하는 패턴 매칭 연산의 수에 따라 변하게 된다. 패턴 매칭 연산은 크게 **메모리 접근**과 **벡터 계산**의 두 부분으로 나누어 질 수 있다. 메모리 접근 시간(T_m)은 입력 버퍼와 문자 위치 벡터 테이블에 접근하는 데 걸리는 시간이다. 만약 입력 버퍼나 문자 위치 테이블들이 범용 프로세서의 첫째 레벨 캐쉬처럼 구현된다면 접근 시간은 각 입력 버퍼와 문자 위치 벡터 테이블에 대해 1 사이클 씩 걸릴 것이다. 만약 느린 메모리 구현 방법이 사용되었다면은 접근 시간은 100 사이클을 넘길 수 있다. 본 논문에서는 성능 평가를 위하여 이 메모리 접근 시간을 2~64 사이클 까지 변하였다. 벡터 계산은 N개의 Shift-OR 연산과 하나의 매칭 발견 과정으로 구성되어 있다. Shift-OR 연산은 두개의 게이트 지연(1 AND and 1 OR gate delays)이 걸리고 매칭 발견 과정은 N의 값에 따라 1~4 게이트 지연(0~4 AND and 1 OR gate delays)이 걸린다. 각 게이트의 연산은 무척 빠르기 때문에 Shift-OR 연산 시간(T_{so})은 0.25 에서 1 사이클이 걸리고 매칭 발견 과정은 2 사이클이 걸린다고 가정하였다. 벡터 계산 시간이 소수점 밑의 숫자를 가질 경우 소수점 올림을 하여 정수 값을 취한다.

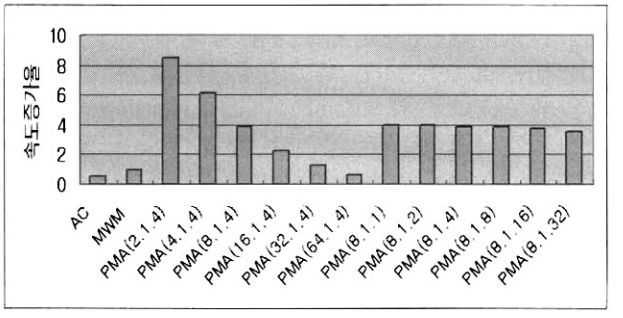
PMA_match_position의 수행 시간은 우선 순위 인코더의 실행 시간(T_p)이다. 우선 순위 인코더는 많은 입력을 갖게 되고 그 크기는 모든 패턴의 크기를 더한 것과 같다. Snort 2.1.2의 규칙의 경우 약 24K 정도 된다. 이와 같은 많은 입력을 갖는 우선 순위 인코더는 parallel priority look-ahead architecture로 구성 할 수 있다[21]. 이 경우 게이트의 지연은 약 $\log_2 M$-3 가 되며, PMA 경우에는 약 12 게이트 지연이 된다. 성능 평가를 위해 우선 순위 인코더의 수행 시간을 2~8 사이클로 변화시켰다.

PMA가 하나의 패킷을 처리하기위해 걸리는 시간은 다음은 식으로 표현된다.

$$\text{PMA execution time} = 1+\left(T_m + \left\lfloor N \times T_{so} + 2 \right\rfloor\right) \times N_{po} + T_p \times N_{pm} \qquad (6)$$

N_{po} 패턴 매칭 연산의 수
N_{pm} 패턴 매칭의 수
N shift size
T_m 메모리 접근 시간
T_{so} Shift-OR 연산 수행 시간
T_p 우선 순위 인코더 수행 시간

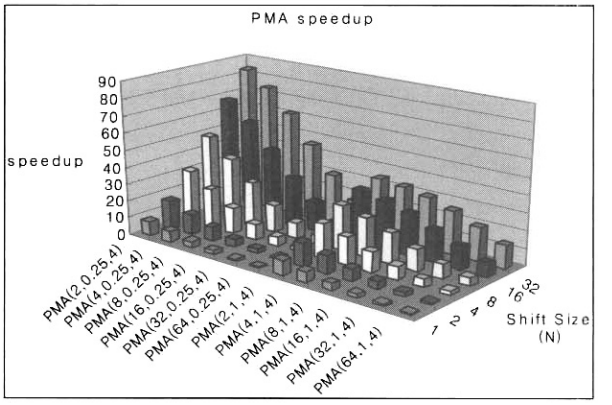
위의 수식에서 '1'은 PMA_reset 수행 시간이고, 수식의 두 번째 항목은 PMA_pattern_search의 수행 시간이고, 마지막으로 세 번째 항목은 PMA_match_position 수행 시간이다. 패턴 매칭 연산의 수(N_{po})와 패턴 매칭의 수(N_{pm})는 Snort가 패킷 트래이스(Packet trace)를 가지고 수행 하는 동안 측정하여 얻고, N, T_m 와 T_p은 위에서 설명된 것처럼 변화하며 수식 (6)에 따라 PMA의 수행 시간을 평가하였다. 사용된 패킷 트래이스는 DefCon 11 [22,23]의 'Capture the flag contest'에서 수집된 것이다. 'Defcon's Capture the Flag(CtF)' 경기는 가장 큰 오픈 컴퓨터 보안 해킹 게임이다.



(그림 6) 메모리 접근시간과 우선 순위 인코더 수행 시간 변화에 따른 PMA 속도증가율 (배수)

Snort에서 사용되는 다중 패턴 매칭 소프트웨어의 수행 시간은 2.4GHz Pentium IV와 512MB 메모리를 갖는 Linux PC 에서 Snort를 수행하며 측정 되었다. 이 측정을 위하여 Pentium 프로세서의 Time-stamp counter명령어[24]를 사용하여 이들 소프트웨어들이 수행에 걸린 사이클을 측정 하였다. (그림 6)은 MWM 방법의 수행 에 걸린 사이클을 기준으로 PMA 와 AC, MWM 의 속도증가율(speedup)을 보여주고 있다. Y 축의 숫자 n은 MWM 보다 n 배 빠르다는 것을 나타낸다. X 축에 표시되어 있는 PMA(X,Y,Z) 는 T_m = X, T_{so} = Y, 그리고 T_p = Z 사이클로 설정되었을 때의 PMA를 나타낸다. (그림 6)에 나타난 결과는 MWM 이 AC 방법에 비해 약 두 배가 빠르다는 것을 보여주고 있다. 메모리 접근 시간과 Shift-OR 연산 수행 시간이 각각 8과 1 사이클로 고정 되고 우선 순위 인코더의 수행 시간이 1 에서 부터 32 사이클로 증가 하였을 때, PMA의 속도 증가율은 4배에서 별 변화가 없었다. 반면 Shift-OR 연산의 수행과 우선 순위 인코더의 수행이 각각 1과 4 사이클로 설정 되고 메모리 접근 시간이 2부터 64 사이클까지 증가 되었을 때, PMA의 속도 증가율은 8 배에서 0.5 배로 떨어 졌다. 이 결과는 PMA의 성능이 메모리 접근 시간에 영향을 많이 받는 다는 것을 보여 주고 있다. 이 결과는 또한 패턴 매칭 연산에서 메모리 접근에 들어가는 수행 시간이 우선 순위 인코더의 수행 시간보다 더 많은 영향을 PMA 성능에 미친다는 것을 보여 주고 있다. 우선 순위 인코더의 수행 시간 증가는 PMA 성능을 크게 변화 시키지 못 한 반면 메모리 접근시간의 증가는 PMA의 속도 증가율을 크게 줄였다. 이러한 결과가 나온 이유는 패턴 매칭 연산의 수 N_{pa} 가 패턴 매칭의 수 N_{pm} 보다 훨씬 크기 때문이다.

다음으로 shift size N, 즉 한번에 처리 될 수 있는 문자들의 수를 1 에서 32까지 변화 시키면서 PMA 의 성능을 측정하였다 (그림 7). Shift-OR 연산에 걸리는 시간은 0.25와 1 사이클로 정하고 우선 순위 인코더의 수행 시간은 4 사이클로 고정하였다. (그림 7)은 shift size를 증가 시킴에 따라 거의 비례하여 PMA의 성능이 증가 됨을 보여 준다. Shift size의 영향은 Shift-OR 연산의 수행 시간이 적을 때 더 크게 나타난다. 따라서 빠른 Shift-OR 연산 수행 시간을 최대한 이용하기 위해서는 될 수 있는 한 큰 shift size을 가질 필요가 있다. 메모리 접근 사간이 2 사이클 걸리고 Shift-OR 연산이 0.25 사이클 걸릴 때 32 개의 문자 위치 벡터 테이블을 사용하면 PMA는 MWM 보다 약 80배 이상 빠르고, 16개의 문자 위치 벡터 테이블을 사용하면 70 배 정도 빠르게 된다. 종합적으로 이 결과는 빠른 메모리 접근 시간과 큰 shift size는 PMA가 좋은 성능을 얻는데 중요하다는 것을 보여 주고 있다. 고속의 메모리 접근 시간은 빠른 메모리 구현 기술을 사용하거나 입력 버퍼와 문자 위치 테이블의 접근 경로를 파이프 라인화 함으로서 성취할 수 있다. Shift size N의 값은 입력 버퍼에서 동시에 몇 개의 문자를 읽어올 수 있는가와 문자 위치 벡터 테이블을 구성할 수 있는 칩(Chip)의 영역에 따라 정해진다. 각 문자 위치 벡터 테이블은 768K (256 x 24K bits) 바이트의 크기를 갖는다. 따라서 32개 테이블은 약 24M bytes 영역을 차지한다. Intel 에서 개발되고 있는 Itanium 프로세서가 26.5Mbytes의 on-chip캐쉬[25]를 갖고있는 것을 볼 때, 현재의VLSI 기술은 32개의 문자 위치 벡터 테이블을 갖는 고성능 PMA를 구현할 수 있다고 생각된다.



(그림 7) Shift size N, 메모리 접근 시간, Shift-OR 연산 시간 변화에 따른 PMA 속도증가율(배수)

6. 결 론

네트워크 침입방지 시스템(NIPS)은 주요 네트워크의 경로 중간에 위치하여 공격 패킷을 직접 조사하고 차단함으로써 능동적으로 악의적 공격으로부터 네트워크 시스템을 보호한다. 최근 폭발적으로 증가하는 네트워크 트래픽과 악의적 공격 및 기간망 네트워크의 수 기가비트 급으로의 속도 증가는 더욱 더 빠른 침입방지 시스템을 요구한다. 패턴 매칭은 NIPS에서 악의적 공격의 패턴을 패킷의 데이터에서 발견하는데 사용되고, Snort의 분석에서 알 수 있듯이 수행 시간의 상당 부분을 차지하며 NIPS의 성능에 결정적인 영향을 미친다. 본 논문에서는 Snort의 구조와 패턴 매칭의 특성을 분석하였다. Snort는 효율적인 패턴 옵션 처리를 위하여 다중 패턴 매칭 방법을 사용하는데 이 방법들은 다양한 길이를 갖는 많은 수의 패턴을 효과적으로 다룰 수 있어야 하며, 대소문자 구별 없는 패턴 매칭과 여러 개의 입력 문자를 동시에 처리 할 수 있는 능력이 있어야 한다.

본 논문에서는 Shift-OR 알고리즘을 효율적으로 구현하는 하드웨어 패턴 매칭 방법을 제시하고 확장 하였다. 제시된 패턴 매칭 하드웨어(PMA)는 위에서 분석된 NIPS의 패턴 매칭 방법 갖아야 하는 조건들을 효과적으로 만족 시킨다. PMA를 구성하는 장치들의 속도를 다양하게 가정하고 성능을 측정하여 보았을 때 PMA는 Snort에서 사용되는 가장 빠른 소프트웨어 패턴 매칭 방법보다 80배 이상 빠를 수 있었다.

참 고 문 헌

[1] Code Red worm exploiting buffer overflow in IIS indexing

service DLL. CERT Advisory CA-2001-19, Jan 2002.

[2] MS-SQL Server Worm. CERT Advisory CA-2003-04, Jan 2003.

[3] 정보흠, 김정녀, 손승원, "침입방지시스템 기술 현황 및 전망," 주간기술동향 통권 1098호, 2003. 6. 3.

[4] X. Zhang, C. Li, and W. Zheng, "Intrusion Prevention System Design", Proceedings of the Fourth International Conference on Computer and Information Technology, September, 2004

[5] Snort. http://www.snort.org/

[6] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems", ACM Workshop on Software and Performance, 2004.

[7] E. P. Markatos, S. Antonatos, M. Polychronakis, and K. G. Anagnostakis, "Exclusion-based Signature Matching for Intrusion Detection", The IASTED International Conference on Communications and Computer Networks, Oct. 2002.

[8] C. J. Coit, S. Staniford, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort", The 2nd DARPA Information Survivability Conference and Exposition (DISCEX II), June 2002.

[9] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection", The 23rd Conference of the IEEE Communications Society (INFOCOM'04), March 2004.

[10] A. V. Aho and M.J. Corasick, "Efficient string matching : An aid to bibliographic search", Communications of the ACM, 18(6):333-340, 1975

[11] Sun Wu and Udi Manber, "AGREP - A Fast Approximate Pattern-Matching Tool", The 1992 Winter USENIX Conference, January, 1992.

[12] R. Sidhu, and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs", The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, May 2001.

[13] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware", The 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, September 2002.

[14] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall", The 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, April 2003.

[15] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology", The 12th International Conference on Field-Programmable Logic and Applications, September 2002.

[16] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering", The International Conference on Field Programmable Logic and Applications, September 2002.

[17] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System", The 13th International Conference on Field Programmable Logic and Applications, September 2003.

[18] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching", The 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines, April 2004.

[19] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters", The International Symposium on High Performance Interconnects (HotI), Aug. 2003.

[20] Ricardo A. Baeza-Yates, and Gaston H. Gonnet, "A New Approach to Text Searching", The Communications of the ACM, October 1992.

[21] C. KUN, S. Quan, and A. Mason, "A Power-Optimized 64-bit Priority Encoder Utilizing Parallel Priority Look-Ahead", IEEE Int. Symposium on Circuits and Systems (ISCAS), May 2004

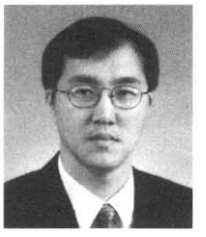
[22] C. Cowan, S. Arnold, S. Beattie, C. Wright, and J. Viega, "Defcon Capture the Flag: Defending Vulnerable Code from Intense Attack", The DARPA DISCEX III Conference, April 2003

[23] Capture the RootFu!, The Shmoo Group, url http://www.shmoo.com/cctf/

[24] IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide, Intel, 2004.

[25] S. Naffziger, T. Grutkowksi, and B. Stackhouse, "The Implementation of a 2-core Multi-Threaded Itanium Family Processor", IEEE International Solid-State Circuits Conference, 2005

김 선 일



e-mail : sikim@cs.hongik.ac.kr

1985년 서울대학교 컴퓨터공학과(학사)

1987년 서울대학교 컴퓨터공학과(석사)

1995년 University of Illinois at Urbana-Champaign (전산학 박사)

1995년~1999년 IBM, USA (연구원)

1999년~현재 홍익대학교 정보컴퓨터공학부 교수