리눅스 넷필터 기반의 인터넷 웜 탐지에서 버퍼를 이용하지 않는 빠른 스트링 매칭 방법

곽 후 근[†] · 정 규 식[††]

요 약

전 세계적으로 큰 피해를 주는 웜을 탐지하고 필터링 하는 것은 인터넷 보안에서 큰 이슈중의 하나이다. 웜을 탐지하는 하나의 방법으로서 리눅스 넷필터 커널 모듈이 사용된다. 웜을 탐지하는 기본 동작으로서 스트링 매칭은 네트웍 상으로 들어오는 패킷을 미리 정의된 웜 시그니쳐(Signature, 패턴)와 비교하는 것이다. 웜은 하나의 패킷 혹은 2개(혹은 그 이상의) 연속된 패킷에 나타난다. 이때, 웜의 일부분은 첫 번째 패킷에 있고 나머지 부분은 연속된 패킷 안에 있다. 웜 패턴의 최대 길이가 1024 바이트를 넘지 않는다고 가정하면, 2048 바이트의 길이를 가지는 2개의 연속된 패킷에 대해서 스트링 매칭을 수행해야만 한다. 이렇게 하기 위해, 리눅스 넷필터는 버퍼에 이전 패킷을 저장하고 버퍼링된 패킷과 현재의 패킷을 조합한 2048 바이트 크기의 스트링에 대해 매칭을 수행한다. 웜 탐지 시스템에서 다루어야 하는 동시 연결 개수의 수가 늘어날수록 버퍼(메모리)의 총 크기가 증가하고 스트링 매칭 속도가 감소하게 된다.

이에 본 논문에서는 메모리 버퍼 크기를 줄이고 스트링 매칭의 속도를 증가시키는 버퍼를 이용하지 않는 스트링 매칭 방식을 제안한다. 제안된 방식은 이전 패킷과 시그니쳐(Signature)의 부분 매칭 결과만을 저장하고 이전 패킷을 버퍼링하지 않는다. 부분 매칭 정보는 연속된 패킷에서 웜을 탐지하는데 사용된다. 제안된 방식은 리눅스 넷필터 모듈을 수정하여 구현하였고, 기존 리눅스 넷필터 모듈과 비교하였다. 실험 결과는 기존 방식에 비해 25%의 적은 메모리 사용량 및 54%의 속도 향상을 가짐을 확인하였다.

키워드 : 인터넷 웜, 리눅스 넷필터, 스트링 매칭, 버퍼(메모리), 속도

A Fast String Matching Scheme without using Buffer for Linux Netfilter based Internet Worm Detection

Hukeun Kwak[†] · Kyusik Chung[††]

ABSTRACT

As internet worms are spread out worldwide, the detection and filtering of worms becomes one of hot issues in the internet security. As one of implementation methods to detect worms, the Linux Netfilter kernel module can be used. Its basic operation for worm detection is a string matching where coming packet(s) on the network is/are compared with predefined worm signatures(patterns). A worm can appear in a packet or in two (or more) succeeding packets where some part of worm is in the first packet and its remaining part is in its succeeding packet(s). Assuming that the maximum length of a worm pattern is less than 1024 bytes, we need to perform a string matching up to two succeeding packets of 2048 bytes. To do so, Linux Netfilter keeps the previous packet in buffer and performs matching with a combined 2048 byte string of the buffered packet and current packet. As the number of concurrent connections to be handled in the worm detection system increases, the total size of buffer (memory) increases and string matching speed becomes low.

In this paper, to reduce the memory buffer size and get higher speed of string matching, we propose a string matching scheme without using buffer. The proposed scheme keeps the partial matching result of the previous packet with signatures and has no buffering for previous packet. The partial matching information is used to detect a worm in the two succeeding packets. We implemented the proposed scheme by modifying the Linux Netfilter. Then we compared the modified Linux Netfilter module with the original Linux Netfilter module. Experimental results show that the proposed scheme has 25% lower memory usage and 54% higher speed compared to the original scheme.

Key Words : Internet Worm, Linux Netfilter, String Matching, Buffer(Memory), Speed

1. 서 론

인터넷 웜이란 네트웍 상에서 자신을 스스로 복제(Replication)하고 전파(Propagation)할 수 있는 독립된 프로그램으로서 호스트 내의 시스템 자원 혹은 네트웍 대역폭을 소비함으로써 전 세계적으로 큰 피해를 주고 있다[1-4]. 이러한 웜을 막을 수 있는 방법에는 침입 탐지 방법(IDS: Intrusion Detection System)[5-7]과 침입 차단 방법(IPS: Intrusion Prevention System)[8, 9]이 존재한다[10].

※ 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌음.
† 준 회 원 : 숭실대학교 전자공학과 대학원
†† 정 회 원 : 숭실대학교 정보통신전자공학부 교수
논문접수 : 2006년 6월 2일, 심사완료 : 2006년 10월 11일

IDS는 Intrusion Detection System(침입 탐지 시스템)의 약자로, 단순한 접근 제어 기능을 넘어서 침입의 패턴 데이터베이스와 전문가 시스템(Expert System)을 사용해 네트워크나 시스템의 사용을 실시간 모니터링하고 침입을 탐지하는 보안 시스템이다. IDS는 허가되지 않은 사용자로부터 접속, 정보의 조작, 오용, 남용 등 컴퓨터 시스템 또는 네트워크 상에서 시도됐거나 진행 중인 불법적인 시도에 대한 예방에 실패한 경우 취할 수 있는 방법으로 의심스러운 행위를 감시하여 가능한 침입자를 조기에 발견하고 실시간 처리를 목적으로 하는 시스템이다[11].

IDS는 모니터링의 대상에 따라 네트워크 기반 IDS와 호스트 기반 IDS로 나눌 수 있다. 호스트 기반 IDS[12]는 시스템 내부에 설치되어 하나의 시스템 내부 사용자들의 활동을 감시하고 해킹 시도를 탐지해내는 시스템이다. 네트워크 기반 IDS[13, 14]는 네트워크의 패킷 캡처링에 기반하여 네트워크를 지나다니는 패킷을 분석해서 침입을 탐지해낸다. 시스템 기반 IDS는 모니터링하려는 시스템마다 하나씩 설치가 되어야 하지만, 네트워크 기반 IDS는 네트워크 단위에 하나만 설치하면 된다[15]. 표 1은 호스트 기반 IDS와 네트워크 기반 IDS를 탐지 대상, 설치 대상 및 기반 기술을 토대로 비교한 것이다.

IPS는 Intrusion Prevention System(침입 차단 시스템)의 약자로, 잠재적 위협을 인지한 후 이에 즉각적인 대응을 하기 위한 네트웍 보안 기술 중 예방적 차원의 접근방식에 해당한다. IPS 역시, 침입 탐지 시스템인 IDS와 마찬가지로 네트웍 트래픽을 감시한다. 공격자가 일단 액세스 권한을 획득하고 나면 시스템의 악의적인 이용이 매우 빠르게 진행될 수 있기 때문에, IPS 역시 네트웍 관리자가 설정해 놓은 일련의 패턴에 기반을 두고 즉각적인 행동을 취할 수 있는 능력을 가지고 있어야 한다. 이를 위하여, IPS는 어떤 한 패킷을 검사하여 그것이 부당한 패킷이라고 판단되면, 해당 IP 주소 또는 포트에서 들어오는 모든 트래픽을 봉쇄하는 한편, 합법적인 트래픽에 대해서는 아무런 방해나 서비스 지연 없이 수신측에 전달한다.

IDS는 침입이 있었다는 것(과거형)을 찾아내어 관리자에게 통보하는 등의 조치를 취하는 반면에 IPS는 현재 통과하는 패킷이 침입이라는 판단(현재형)을 하게 되면 바로 차단하는 것이 IDS와 IPS와의 차이점이다.

효과적인 침입 탐지 시스템이 되려면 개별 패킷은 물론 트래픽 패턴을 감시하고 대응하는 등 보다 복잡한 감시와 분석을 수행할 수 있어야 한다. 탐지 기법으로는 주소 대조, HTTP 스트링과 서브스트링 대조, 일반 패턴 대조, TCP 접속 분석, 변칙적인 패킷 탐지, 비정상적인 트래픽 탐지 및 TCP/UDP 포트 대조 등이 있다. 광범위하게 말하자면, 방화벽이나 앤티바이러스 소프트웨어, 그리고 네트웍 등의 접근 권한을 획득하려는 공격자들을 막기 위해 사용되는 것이라면 어떠한 제품이나 수단이라도 IPS라 지칭할 수 있다.

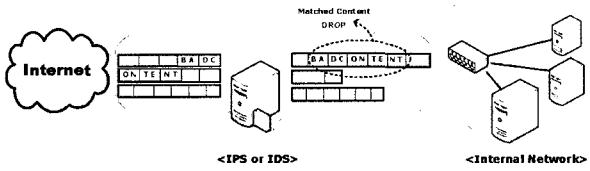
〈표 1〉 호스트 기반 IDS와 네트워크 기반 IDS의 비교

구분	네트워크 기반 IDS	호스트 기반 IDS
탐지 대상	네트워크를 통과하는 패킷	시스템 내부 사용자들의 활동
설치 대상	네트워크 세그먼트	호스트
기반 기술	패킷 캡처링	프로세스 모니터링
	프로토콜별 패킷 분석	실시간 로그분석
	패킷 조각 모음	TTY 모니터링

오픈 소스 중 네트웍을 기반으로 침입을 탐지 혹은 차단하는 프로그램으로는 대표적으로 Netfilter[16]와 Snort[17]가 존재한다. Netfilter는 패킷을 운영 체제의 커널 레벨에서 처리하고, Snort는 유저 레벨에서 처리하는 것이 다르다. 본 논문에서는 Netfilter를 이용하여 IPS를 구성하는 것에 초점을 맞춘다.

현재 웜을 차단하는 가장 기본적인 방식은 스트링 매칭 방식[18, 19]이다. 스트링 매칭 방식은 말그대로 캡쳐한 패킷과 시그니쳐(Signature)를 비교하는 데 주로 사용되는 방식이다. 즉, 캡쳐된 패킷 안의 컨텐츠(스트링)와 시그니쳐(Signature)의 컨텐츠(스트링)를 비교하여 스트링이 일치하면 웜으로 간주하고 패킷을 폐기(Drop)하는 방식으로 동작한다. (그림 1)은 스트링 매칭 방식을 통해 네트웍상의 웜을 차단하는 것을 나타낸다.

본 논문에서는 기존의 리눅스 넷필터 기반의 인터넷 웜 탐지에서 버퍼를 이용하는 스트링 매칭 방식이 가지는 문제점을 분석하고 이를 해결할 새로운 버퍼를 이용하지 않는 스트링 매칭 방식을 제안한다. 본 논문의 구성은 다음과 같다. 제 2장에서는 기존의 버퍼를 이용하는 스트링 매칭 방식과 그 문제점을 소개한다. 3장에서는 기존 방식의 문제점을 해결하는 새로운 버퍼를 이용하지 않는 스트링 매칭 방식을 설명하고, 4장에서는 실험 및 토론을, 5장에서는 결론 및 향후 연구 방향을 제시한다.



(그림 1) 네트웍에 기반한 웜 차단 방식 : 스트링 매칭

2. 연구 배경

2.1 침입 차단 시스템 (IPS : Intrusion Prevention System)

IPS에서 갖추어야 하는 기본적인 보안 기능들 및 각각에 대한 설명은 다음과 같다.

- Access : 액세스는 현재 보안 패턴을 구성하는 장비 자체에 대한 보안을 의미한다. 일반적으로, 모든 연결을 막고 필요한 연결만 허용하도록 구성한다.
- Filter : 필터는 장비를 통과해서 포워딩되는 패킷에 대한 검사를 수행한다. 주소부터 플래그까지 다양한 조건

을 통해 검사가 가능하다.

- Advanced : ACL과 미리 정의된 필터(DoS, Worm)를 정의할 때 사용된다.
- Policy : 위에서 정의된 패턴들에 대해 인터페이스에 실제 적용할 때 사용된다.
- Update : 미리 정의된 필터(DoS, Worm)의 업데이트를 위해 사용되며, 업데이트 서버에 접속하여 업데이트 파일을 가지고 옴으로써 동작한다.

또한 IPS를 실제로 사용하려고 할 때의 고려 사항들 및 이들에 대한 설명은 다음과 같다.

- 성능과 안정성 : IPS의 성능 평가 요소로 볼 수 있는 것은 최대 패킷 처리량(Throughput), 최대 동시 처리 세션 수, 최대 초당 처리 세션 수(Session rate) 등이 있다.
- 관리적인 편의성 : 관리적인 편의성의 요소로 볼 수 있는 것은 설치 및 업데이트의 용이성, 지원하는 관리 인터페이스, 보안 정책 관리의 편이성, 다양한 관리자 역할 지원, 로그 관리 기능, 리포트 기능, 침입 차단 시스템 상태 모니터링 등이 있다.
- 이중화를 통한 고가용성/로드 분산을 통한 고성능 : IPS는 관문의 장비이기 때문에 장애 발생 시 모든 네트워크가 마비되는 특정 지점의 오류(Single point of failure)가 될 수도 있다. 따라서 끊김 없는 네트워크 환경을 위해 이중화 방안을 마련해두는 것이 필수적이다.
- 인증 기능 : 재택 근무자가 유동 IP로 작업이 필요할 시에 정책을 따로 입력하는 대신, 인증을 요구하는 창에 ID와 패스워드를 입력한다. 이렇게 인증 받은 사용자 IP에 대해 인증 정보가 캐쉬되어 있는 동안 정책은 허용돼 있는 것과 같은 효과를 얻고 작업이 완료되면 인증 정보가 사라지면서 다시 해당 정책은 차단된다.
- 상위 레벨 프로토콜 분석(Protocol Inspection) 기능 : IPS에는 FTP와 같은 동적으로 개방되는 데이터 세션을 처리하기 위해 상위 레벨 프로토콜에 대한 지원이 필요하다.
- VPN 기능 : VPN은 저렴한 비용으로 사설망의 효과를 얻을 수 있기 때문에도 사용되지만, 재택 근무자, 파견자 등과 같이 원격 액세스 사용자로 하여금 안전한 접속 환경을 제공하기도 한다.

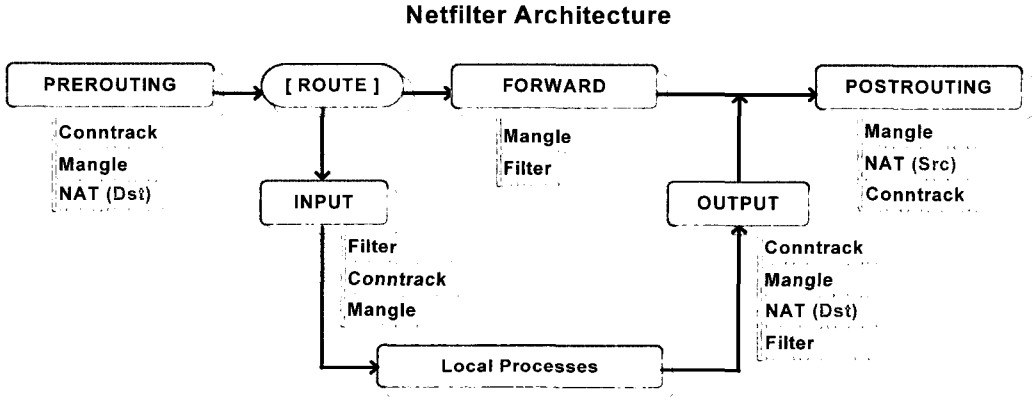
본 논문에서는 IPS를 실제로 사용할 때 고려되어야 하는 성능에 초점을 맞춘다. 기존 논문에서는 성능을 높이기 위해 하드웨어적인 접근이 주류를 이루었지만[20-22], 소프트웨어적인 접근(주요 알고리즘 개선)은 미비한 실정이다. 이에 본 논문에서는 IPS의 성능을 개선하기 위해 IPS의 웜 패턴 비교 알고리즘인 스트링 매칭 방법을 개선한다.

2.2 넷필터 (Netfilter)

넷필터[16]는 리눅스에서 사용되는 커널 모듈로 조건부 패킷 필터링 기능을 담당한다. 넷필터는 크게 세 부분으로 구성되어 있는데, 먼저 각각의 프로토콜은 훅(Hook)이라는 것을 정의하며, 이는 패킷 프로토콜 스택에 있는 잘 정의된 포인터를 의미한다. 이러한 포인터에서, 각각의 프로토콜은 패킷과 훅 넘버(Hook Number)를 이용하여 넷필터 프레임웍을 호출하게 된다. 두 번째로, 커널의 일부분은 각 프로토콜에 대하여 다른 훅을 감시하도록 등록할 수 있다. 따라서 패킷이 넷필터 프레임웍을 통과할 때, 누가 그 프로토콜과 훅을 등록했는지 확인하게 된다. 마지막 부분은 대기된(Queuing) 패킷을 사용자 공간으로 보내기 위해 제어하는 것으로 이러한 패킷은 비 동기 방식으로 처리된다.

넷필터는 프로토콜 스택의 다양한 포인트에 존재하는 훅(Hook)의 연속이다. 이상적인 IPv4의 진행경로 다이어그램은 (그림 2)와 같고 이들의 동작 과정은 다음과 같다.

- 패킷은 그림의 좌측으로부터 들어와서 단순한 데이터 체크(즉, 데이터가 잘렸는지, 혹은 IP 체크 섬의 이상유무, 뒤죽박죽되지는 않았는지 등)를 거쳐, 넷필터 프레임웍의 NF_IP_PRE_ROUTING [PRE] 훅으로 전달된다.
- 패킷은 라우팅 코드로 들어가며, 여기서 패킷이 다른 인터페이스로 향하는지 또는 로컬 프로세스로 향하는지 결정된다. 패킷이 라우팅될 수 없는 경우, 라우팅 코드는 패킷을 버리기도 한다.



(그림 2) Netfilter

- 패킷의 목적지가 자신이라면, 넷필터 프레임웍은 패킷을 프로세스로 전달하기 전에 NF_IP_LOCAL_IN [IN] 훅을 다시 한번 호출하게 된다.
- 다른 인터페이스로 전달하고자 한다면, 넷필터 프레임웍은 NF_IP_FORWARD [FWD] 훅을 호출한다. 그리고 나서 패킷이 넷트웍 라인으로 보내지기 전에 마지막 넷필터 훅인 NF_IP_POST_ROUTING [POST] 훅으로 전달된다.
- 로컬에서 생성된 패킷에 대해서는 NF_IP_LOCAL_OUT [OUT] 훅이 호출된다. 이 때, 이 훅이 호출된 후 라우팅이 발생하는 것을 알 수 있다.

Netfilter는 위에서 언급된 INPUT, FORWARD, OUTPUT 패킷에 대해 다양한 패턴을 적용하고 패킷을 통과(Accept)시키거나 폐기(Drop)할 수 있다. 넷필터의 패턴은 기본 패턴들에 외에 다양한 패턴들이 모듈로 구성되어 있다. 이러한 패턴들은 소스나 목적지 어드레스에 대한 패턴에서부터 어플리케이션 계층에 대한 매칭까지 다양하게 존재한다.

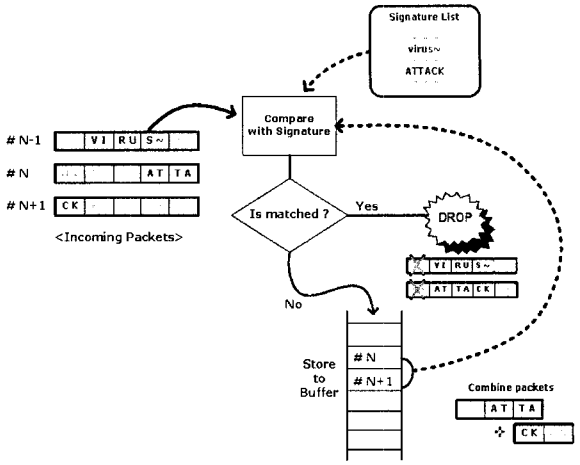
2.3 버퍼를 이용하는 스트링 매칭

(그림 3)은 리눅스 넷필터 기반의 버퍼를 이용한 스트링 매칭의 동작과정을 나타낸다. 이전 패킷을 버퍼에 저장하여 이후 패킷과 합쳐 시그니쳐(Signature)와 매칭하는 이유는 다음과 같다. 시그니쳐(Signature)에서 매칭하려는 스트링이 하나의 패킷에 모두 들어있을 수도 있지만 두개의 패킷에 나뉘어서 들어오는 경우도 있기 때문이다. 그러므로 이전 데이터는 항상 버퍼에 저장하고 이후 패킷과 합쳐져서 시그니쳐(Signature)와 매칭해야 한다[16].

버퍼를 이용한 스트링 매칭 방법의 알고리즘은 다음과 같다.

- 단계 1 : 버퍼링된 이전 패킷과 현재 들어온 패킷을 조합한 2048 바이트 크기의 스트링이 시그니쳐(Signature)와 매칭되었다면 패킷을 폐기(Drop) 한다.
- 단계 2 : 버퍼링된 이전 패킷과 현재 들어온 패킷을 조합한 2048 바이트 크기의 스트링이 시그니쳐(Signature)와 매칭되지 않았다면 이 패킷을 통과(Accept)시키고 버퍼에 저장한다.
- 단계 3 : 현재 들어온 모든 패킷에 대해 1-2단계를 반복한다.

(그림 3)에서 보면 VIRUS~라는 스트링을 가진 패킷은 시그니쳐(Signature) 리스트에 있는 VIRUS~와 매칭이 되어 바로 폐기됨을 알 수 있다. 그러나 ATTACK이라는 스트링을 가진 패킷은 2개의 패킷으로 나뉘어져 들어오게 되어 시그니쳐(Signature) 리스트에 있는 ATTACK이라는 스트링과 매칭이 되지 않음을 알 수 있다. 이때, 버퍼를 사용하게 되면 ATTA라는 스트링을 가진 이전 패킷과 CK를 가지는 현재의 패킷을 합쳐서 시그니쳐(Signature) 리스트에 있는



(그림 3) 버퍼를 이용한 스트링 매칭의 동작 과정

ATTACK이라는 스트링과 매칭할 수 있다.

2.4 제안 방식

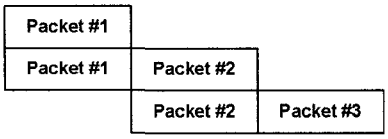
2.4.1 버퍼를 이용하는 스트링 매칭의 문제점

버퍼를 이용하는 스트링 매칭의 문제점은 메모리 사용량에 있다. 즉, 버퍼를 할당하기 위해 메모리를 사용하고, 동시 접속 수가 급격히 증가하게 되면 이에 따라 메모리(버퍼) 사용량도 급격히 증가하게 된다. 예를 들어, 하나의 연결(Connection)을 검사하기 위해 2048 바이트의 버퍼가 필요하다면[16], 이러한 연결이 동시에 10만개 정도 들어오면 메모리 사용량은 204.8 Mbytes가 필요하게 된다. 이것은 필요한 메모리 중 버퍼만을 계산한 것으로 실제로 사용시에는 더 많은 메모리가 필요하게 된다.

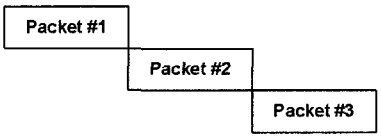
또한, 기존 방식은 예전 데이터를 버퍼링하고 이를 현재의 매칭에 다시 이용함으로써 매칭 속도 측면에서도 느리다는 단점을 가진다.

2.4.2 본 연구의 접근 방식

본 연구에서는 버퍼를 이용한 스트링 매칭의 문제(메모리 소모 및 매칭 속도가 느림)를 해결하는 버퍼를 이용하지 않는 스트링 매칭(메모리를 소모하지 않고 매칭 속도가 빠름) 방식을 제안한다. (그림 4)와 <표 2>는 기존 방법과 제안된 방법의 비교를 나타낸다. (그림 4(a))는 기존의 버퍼를 이용하는 스트링 매칭 방식을 나타내는데, 이전 패킷이 #1이고 현재의 패킷이 #2라면 실제 스트링 매칭은 이전 패킷과 현재의 패킷이 합쳐진 스트링과 시그니쳐(Signature)가 매칭을 하게 된다. 반면 (그림 4(b))는 제안된 버퍼를 이용하지 않는 스트링 매칭 방식으로서, 이전 패킷 #1을 버퍼에 저장하지 않고 부분 매칭된 정보만을 기억하고 있다가 현재 패킷 #2가 들어오면 나머지 매칭을 수행하는 방법이다. 이러한 버퍼를 이용하지 않는 방법은 기존 방법에 비해 메모리 사용 및 매칭 속도 지연을 최소화 한다.



(a) 기존 방법



(b) 제안된 방법

(그림 4) 기존 방법과 제안된 방법의 비교

〈표 2〉 기존 방법과 제안된 방법의 비교

	메모리 사용량	속도	비 고
기존 방법	많다	느리다	이전 패킷과 현재 패킷 모두를 시그니쳐(Signature)와 비교
제안된 방법	적다	빠르다	현재 패킷만 시그니쳐(Signature)와 비교

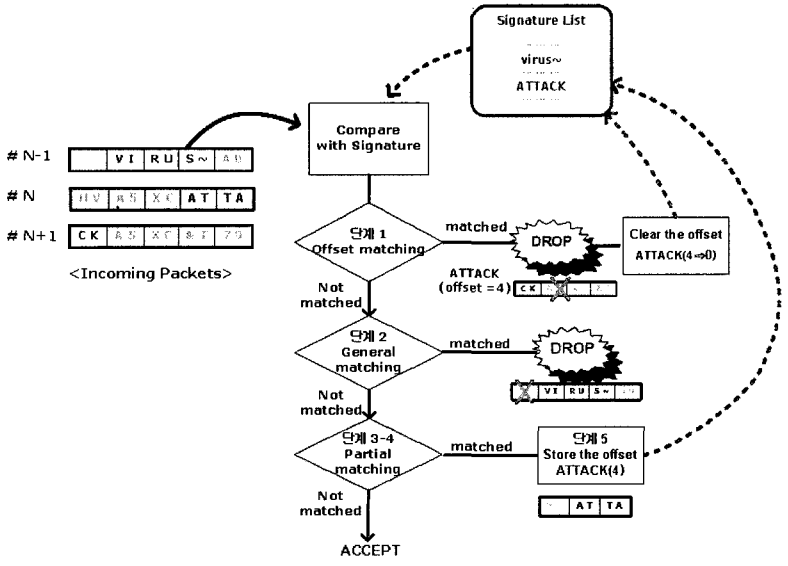
3. 버퍼를 이용하지 않는 스트링 매칭

(그림 5)는 버퍼를 이용하지 않는 스트링 매칭 방식의 동작 과정을 나타낸다. 제안된 방법은 이전 데이터를 저장하지 않고, 이전 데이터와의 매칭 정보(Offset) 만을 저장하는 것을 제외하고 버퍼를 이용하는 스트링 매칭 방식과 동작 과정이 유사하다. 기존 방식은 동시 연결 수에 따라 메모리 소모 및 매칭 속도 지연이 큰 반면 제안된 방식은 동시 연결 수가 늘어나도 적은 메모리 및 속도 지연으로 스트링 매칭을 할 수 있다는 장점을 가진다.

버퍼를 이용하지 않는 스트링 매칭의 알고리즘을 정리하면 다음과 같다.

- 단계 1 : 현재 들어온 패킷은 이전 패킷에서의 매칭 정보를 가지는 변수(Offset_High부터 Offset_Low까지)와 조합하여 시그니쳐(Signature)와 매칭한다. 만일, 매칭되었다면 현재 패킷을 폐기(Drop)한다.
- 단계 2 : 현재 들어온 패킷이 시그니쳐(Signature)와 매칭되었다면 패킷을 폐기(Drop)한다.
- 단계 3 : 현재 들어온 패킷의 마지막 부분을 시그니쳐(Signature)의 처음 부분과 매칭을 한다. 부분 매칭이 되었다면 시그니쳐(Signature)에서 매칭된 부분까지 변수(Offset_Low)에 저장한다.
- 단계 4 : 비슷한 시그니쳐(Signature)가 존재할 수 있음으로 Offset_Low 이후부터 또 다른 부분 매칭이 존재하는지 확인한다. 더 이상 부분 매칭이 안될 때 까지 매칭을 한 후, 매칭이 된 마지막 부분까지 변수(Offset_High)에 저장한다.
- 단계 5 : 부분 매칭 정보(Offset_High, Offset_Low)를 저장하고 현재의 패킷을 통과(Accept) 시킨다.
- 단계 6 : 모든 패킷에 대해 1-5단계를 반복한다.

<표 3>은 기존 스트링 매칭 방법과 제안된 스트링 매칭 방법을 사용하였을 때의 메모리 사용량을 정량적으로 비교한 것이다. 버퍼를 이용하는 스트링 매칭의 경우 이전 데이터와 현재의 데이터를 합쳐서 지정된 패턴과 매칭하기 위해 2048 바이트의 버퍼를 사용하고, 이는 동시 연결(접속) 개수에 따라(2048 x 동시 연결 개수) 바이트 만큼의 메모리가 사용됨을 의미한다. 이에 비해 제안된 방법은 버퍼를 사용



(그림 5) 버퍼를 이용하지 않는 스트링 매칭

하지 않음으로써 버퍼에 따른 메모리 사용이 없고, 대신 이전 데이터에서 매칭된 부분을 저장하는 변수(Offset_High, Offset_Low)에 각각 4 바이트(C 언어의 변수 중 Integer를 사용)가 할당된다. 즉, 동시 연결 개수에 따른 메모리 사용량은 (8 x 동시 연결 개수)바이트 만큼 소모하게 된다.

<표 4>는 기존 스트링 매칭 방법과 제안된 스트링 매칭 방법을 사용하였을 때의 속도를 정량적으로 비교한 것이다. 스트링 매칭의 경우 한 개의 패턴에 대한 매칭 길이는 59 바이트로 계산하였다. 이는 최신 웜들을 차단하기 위해 필요한 10개의 패턴의 길이를 평균한 것으로 <표 5>는 이를 나타낸다. 버퍼를 이용한 스트링 매칭의 경우 2048 바이트와 59 바이트의 패턴을 비교하기 위해서는 총 1989(=2048-59)번의 비교가 필요하고, 이는 패턴의 개수에 따라 (1989 x 패턴의 개수) 만큼의 비교가 필요함을 의미한다(그림 6). 이에 비해 제안된 방법은 1517번의 비교가 필요하고, 패턴의 개수에 따라 (1517 x 패턴의 개수) 만큼의 비교만을 수행하면 된다. 1517번의 비교는 Offset_High와 Offset_Low를 비교하는 부분(58번 비교-그림 7(a)), 사용자 패킷(1460)과 패턴(59)과 비교하는 부분(1401번 비교-그림 7(b)) 및 부분 매칭이 되었는지 비교하는 부분(58번 비교-그림 7(c))으로 구성된다.

〈표 3〉 기존 방법과 제안된 방법의 메모리 사용량 비교 (Bytes)

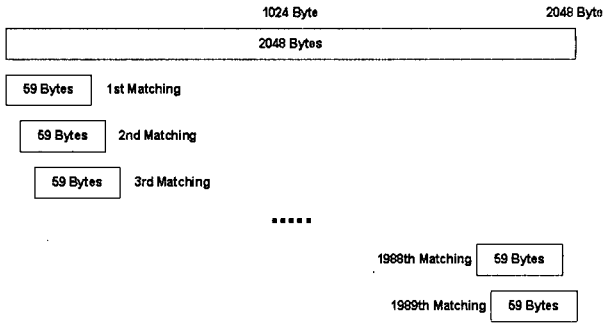
동시 연결 개수	1	10	100	1000
기존 방법	2048	2048 x 10	2048 x 100	2048 x 1000
제안된 방법	8	8 x 10	8 x 100	8 x 100

〈표 4〉 기존 방법과 제안된 방법의 속도 비교 (스트링 매칭 횟수)

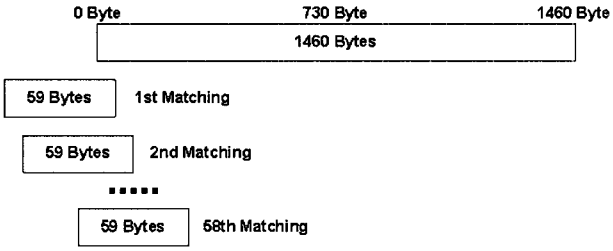
패턴의 개수	1	10	100	1000
기존 방법	1989	1989 x 10	1989 x 100	1989 x 1000
제안된 방법	1517	1517 x 10	1517 x 100	1517 x 1000

〈표 5〉 최신 웜 차단 패턴의 스트링 길이[23]

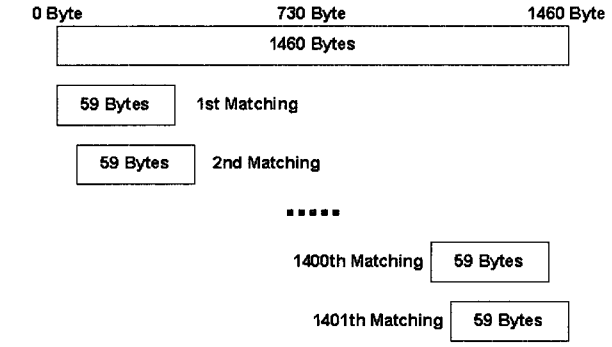
패턴의 이름	패턴에 필요한 컨텐츠 길이
Win32/Yaha.worm	60
Win32/Spig.trojan	14
Win32/Gibe.worm	69
Win32/Mytob.worm	18
Win32/Zafi.worm	71
Win32/Netsky.worm	72
Win32/LovGate.worm	68
Win32/MyDoom.worm	67
Win32/Bagz.worm	60
Win32/Bugbear.worm	60
평균	59



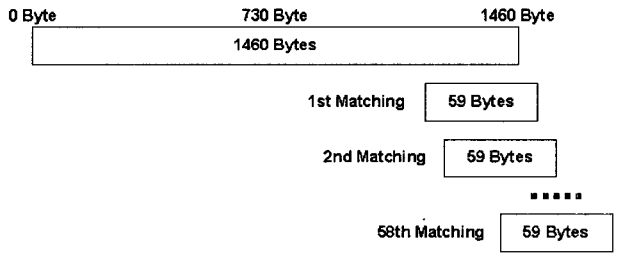
(그림 6) 기존 방법의 스트링 매칭 횟수
(1989번 매칭 = 2048 바이트의 버퍼링된 패킷에 대해 59 바이트의 패턴이 1 바이트씩 오른쪽으로 이동하면서 매칭을 수행함)



(a) 이전 패킷에서 부분 매칭된 Offset 값을 이용하여 부분 매칭하는 부분 (58번 매칭)



(b) 부분 매칭(a)이 없을 때 일반 스트링 매칭 (1401번 매칭)



(c) 일반 매칭(b)이 없을 때 부분 매칭이 되었는지 비교하는 부분 (58번 매칭)

(그림 7) 제안 방법의 스트링 매칭 횟수
(1517번 매칭 = (a)58 + (b)1401 + (c)58)

4. 실험 및 결과 분석

4.1 실험 환경

(그림 8)은 실험 환경을 나타내고, <표 6>은 실험 환경에서 사용된 하드웨어와 소프트웨어를 나타낸다. 실험에 사용된 하드웨어를 보면 클라이언트의 성능이 IPS의 성능보다 좋은 것을 알 수 있다. 이는 클라이언트에서 병목이 발생하지 않는 상황에서 IPS가 처리할 수 있는 최대 패킷을 전송하기 위해서 이다.

제안된 방법은 Netfilter 모듈[16]에서 어플리케이션(Layer7) 매칭에 사용되는 ipt_layer7 모듈[24]을 수정(ipt_layer7.c 내의 match() 함수의 일부분)하여 실험을 수행하였다. 버퍼를 이용한 스트링 매칭 방법은 기존 ipt_layer7 모듈을 그대로 사용하여 실험을 수행하였고, 이때 이전 패킷과 현재 패킷을 버퍼링 하기 위해 2048 바이트의 버퍼(메모리)가 사용됨을 확인하였다.

제안된 버퍼를 이용하지 않는 스트링 매칭 방법은 ipt_layer7 모듈을 수정하여 실험을 수행하였고, 이때 이전 패킷과 현재 패킷을 버퍼링 하지 않고 이전 패킷과의 매칭 정보를 가진 변수만을 두어 매칭을 할 수 있음을 확인하였다. 이때, 사용된 메모리는 이전 패킷과의 매칭 정보를 가진 변수를 위해 8 바이트(= 2개의 오프셋 변수 x 4 바이트)를 사용하였다.

실험은 메모리 사용량과 속도 측면에서 수행하였다. 메모리 사용량은 동시 연결 개수가 변할 때(1개, 10개, 50개, 100개) 각 알고리즘에서의 메모리 사용량을 비교하였다. 속도는 700 Mbytes의 데이터를 전송할 때, 패턴의 개수를 변화시키면서(1개, 10개, 50개, 100개) 각 알고리즘에서의 전송 속도를 비교하였다.

〈표 6〉 실험에 사용된 하드웨어와 소프트웨어

	하드웨어		소프트웨어
	CPU (Hz)	RAM (MB)	
Client	P-4 3.00G	1024	FTP-Client
IPS	P-4 2.26G	256	Netfilter-ipt_layer7

4.2 실험 결과 (정량적 비교)

4.2.1 메모리 사용량

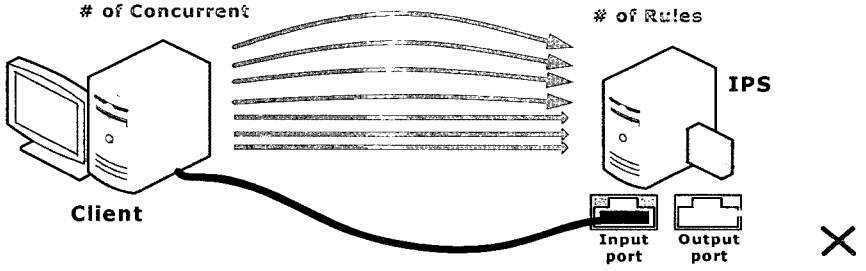
<표 7>과 (그림 9)는 버퍼를 이용하는 스트링 매칭과 이용하지 않는 스트링 매칭의 메모리 사용량을 나타낸다. 버퍼를 이용하는 스트링 매칭은 Netfilter 모듈 중의 어플리케이션 데이터(Layer 7)를 매칭할 수 있는 ipt_layer7 모듈을 사용하였다. ipt_layer7 모듈은 이전 데이터와 현재 데이터를 합쳐서 지정된 패턴과 매칭하기 위해 2048 바이트의 버퍼를 사용하고, 이는 동시 접속 수에 따라 (2048 x 동시 접속 수) 바이트 만큼 메모리 사용량이 늘어나게 된다.

제안된 버퍼를 이용하지 않는 스트링 매칭은 ipt_layer7 모듈을 수정하여 구현되었으며, 버퍼를 사용하지 않음으로 버퍼에 따른 메모리 사용이 없고, 대신 이전 데이터에서 매칭된 부분을 저장하는 변수에 4 바이트가 할당된다. 즉, 동시 접속 수에 따른 메모리 사용량은 (동시 접속 수 x 4 Bytes)만큼 소모하게 된다. 버퍼를 이용하지 않는 스트링 매칭 방법은 버퍼를 사용하지 않기 때문에 전체적으로 기존 방법에 비해 25%의 메모리가 절약된다.

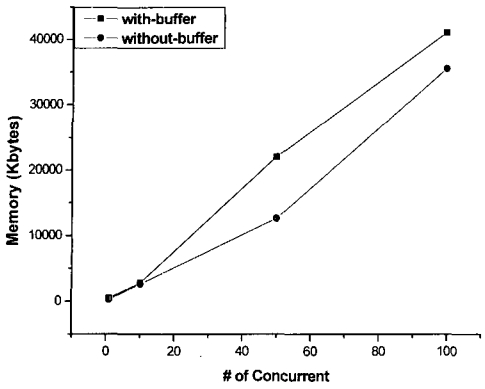
<표 7>과 <표 3>을 비교해보면 메모리 사용량 예측 치와 실험 결과치가 다르다는 것을 알 수 있다. 그 원인으로 지적할 수 있는 것 중 한 가지는 메모리 사용량이 정확하게 (2048 x 패턴의 개수)만큼 사용되지 않았다는 것이고, 나머지 한 가지는 동시 연결이 50개일 때가 100개일 때보다 더 많은 메모리를 사용한다는 사실이다. 메모리 사용량이 정확하게 일치하지 않는 것은 버퍼 할당 이외에 다른 메모리들(연결 추적-Connection Tracking 등)과 같이 사용되는 관계로 이들이 관계함에 따라 메모리 사용량 계산에 차이가 발생하기 때문이다. 또한 50개의 동시 연결에서 더 많은 메모리 사용량을 보이는 것은 동시 연결이 100개가 되면 IPS로 사용되는 장비가 병목이 발생하여 패턴 100개에 대한 정확한 메모리 할당이 이루어지지 않음을 의미한다.

〈표 7〉 메모리 사용량 (KBytes)

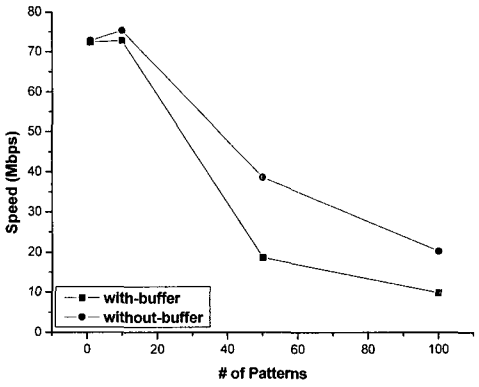
동시 연결 개수	1	10	50	100
With Buffer	492	2768	22036	41000
Without Buffer	300	2600	12632	35468
차이 (With - Without)	192	168	9404	5532



(그림 8) 실험 환경



(그림 9) 메모리 사용량 (KBytes)



(그림 10) 속도 (Mbps)

4.2.2 속도

<표 8>과 (그림 10)은 버퍼를 이용하는 스트링 매칭과 이용하지 않는 스트링 매칭의 속도를 비교한 것이다. 제안된 방식이 버퍼를 시용한 방식에 비해 속도가 더 빠른 것을 알 수 있는데, 이는 버퍼를 사용하는 방식이 기존 데이터를 버퍼링하고 이를 다시 매칭에 사용함으로써 제안된 방식에 비해 느리기 때문이다.

버퍼를 사용하지 않는 제안된 방식은 부분 매칭을 검사하는데 약간의 시간이 소비되나 기존 데이터를 버퍼링 하지 않음으로(기존 데이터를 매칭하는데 시간을 소비하지 않음) 기존 방식에 비해 속도가 빠름을 알 수 있다.

<표 8>을 보면 제안된 알고리즘이 가지는 기존 알고리즘에 비해 가지는 속도 증가 비율은 패턴의 개수에 따라 평균 54%가 됨을 알 수 있다. 패턴의 개수가 50일 때보다 100에서 속도 증가 비율이 낮은 이유는 실험에 사용된 IPS의 하드웨어 사양에 기인한다. IPS의 하드웨어 사양이 클라이언트의 하드웨어 사양보다 낮아 패턴의 개수가 100개일 때 IPS에서 병목(Bottleneck)이 발생했기 때문이다.

〈표 8〉 속도 (Mbps)

패턴의 개수	1	10	50	100
With Buffer	72.32	72.8	18.64	9.84
Without Buffer	72.8	75.28	38.64	20.24
차이 (Without - With)	0.48	2.48	20	10.4

4.3 실험 결과 (정성적 비교)

제안된 버퍼를 이용하지 않는 스트링 매칭 방식이 기존 방식에 비해 갖는 장점은 이전 패킷을 위해 버퍼(메모리)를 할당하지 않아도 된다는 점에 있다. 버퍼를 이용하는 기존 방식이 동시 접속자수가 늘어남에 따라 기하급수적으로 메모리 사용량 및 매칭 속도 지연이 증가하는 반면 제안된 방식은 최소의 메모리 및 매칭 속도 지연을 사용하여 이전 데이터와 스트링 매칭을 할 수 있도록 하였다.

제안된 방식의 단점은 버퍼를 사용하지 않기 위해 기존 패킷과의 매칭 정보를 갖는 변수를 두어 모든 동시 연결마다 이 변수를 관리해야 하는 부담을 가진다.

제안된 버퍼를 이용하지 않는 스트링 매칭 방식을 실제 구현할 때의 고려 사항은 다음과 같다.

- 규칙의 개수 : 규칙의 개수가 증가하면 각 규칙마다 저장해야 하는 부분 매칭 정보(변수)의 수도 증가하게 된다. 현재 IPS에서 사용되고 있는 규칙의 수가 수천 개를 넘지 않지만, 규칙의 개수가 수십만 개로 증가하게 되면 이에 대한 고려가 필요하다.
- 매칭 반복 회수 : 버퍼를 이용한 스트링 매칭에서 패킷의 크기에 따라 이전 패킷을 반복해서 매칭하는 회수가 달라지게 된다. 크기가 작은 패킷들이 연속적으로 들어오게 되면 이전 패킷과의 조합이 2048 바이트를 넘지 않게 됨으로 2번 이상 반복 매칭을 하게 된다. 이에 비해, 제안된 방법은 패킷의 크기에 상관없이 1번만 매칭을 수행한다.
- 패턴의 길이 : 본 논문에서는 패턴의 길이가 1024 바이트를 넘지 않는다고 가정하였다. 이에 따라 필요한 버퍼의 크기도 2048 바이트를 넘지 않는다. 만약, 패턴의 길이가 1024 바이트보다 크다면 버퍼의 크기도 2048 바이트보다 커져야 한다.

5. 결 론

이전 패킷과의 매칭을 위해 버퍼를 이용하는 기존 스트링 매칭의 단점은 동시 연결 수가 늘어남에 따라 버퍼(메모리) 사용량 및 매칭 속도 지연이 급격히 증가하는데 있다. 이에 본 논문에서는 이전 패킷과의 매칭시에도 버퍼(메모리)를 사용하지 않는 방식을 제안하였다. 제안된 방식은 이전 패킷과의 매칭 정보를 담은 하나의 변수만을 두고 버퍼를 사용하지 않는 방식으로 실험을 통해 제안된 방식이 기존 방식에 비해 메모리 사용량 및 속도 지연을 급격히 감소시켰음을 확인하였다. 그러나 버퍼를 사용하지 않기 위해 기존 패킷과의 매칭 정보를 갖는 변수를 두어 모든 동시 연결마다 이 변수를 관리해야 하는 단점을 가진다.

향후 연구 방향은 요약하면 다음과 같다.

- 매칭 정보를 갖는 변수의 관리 : 변수를 따로 두어 관리하기보다 기존에 리눅스 커널에서 사용되고 있는 연결 추적(Connection Tracking)을 위한 구조체 내의 사용하지 않는 변수를 이용하여 관리하는 하는 것을 고려해 볼 수 있다.
- 해싱을 이용한 스트링 매칭 : 규칙의 개수가 늘어남에 따라 매 패킷마다 모든 규칙에 대한 매칭을 수행하면 매칭 속도 지연이 크다는 단점을 가진다. 패킷의 스트링(컨텐츠) 부분에 해싱을 적용하고 이 결과를 매칭에 이용하면 규칙(시그니처-Signature) 개수가 늘어나도 매칭 속도 지연을 작게 유지할 수 있다.

참 고 문 헌

[1] Z. Chen, L. Gao, and K. Kwiat, "Modeling the spread of active worms," 22th Annual Joint Conference of the IEEE Computer and Communications Societies, pp.1890-1900, 2003.

[2] F. Stajano and H. Isozaki, "Security issues for internet appliances," Symposium on Applications and the Internet Workshops, pp.18-24, 2002.

[3] D. Moore, C. Shannon, and J. Brown, "Code-red: a case study on the spread and victims of an Internet worm," In Proceedings of the 2002 Internet Measurement Workshop, 2002.

[4] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A Taxonomy of Computer Worms," In The First ACM Workshop on Rapid Malcode, 2003.

[5] J. McHugh, A. Christie, and J. Allen, "Defending yourself: the role of intrusion detection systems," IEEE Software, Vol.17, No.5, pp.42-51, 2000.

[6] S. Axelsson, "Intrusion Detection Systems: A Survey and Taxonomy," Technical report 99-15, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 2000.

[7] Paul E. Proctor, "Intrusion Detection Handbook," Prentice-Hall, 2001.

[8] IPS, http://www/nss.co.uk/WhitePapers/intrusion_prevention_systems.htm.

[9] X. Zhang, C. Li, and W. Zheng, "Intrusion prevention system design," The Fourth International Conference on Computer and Information Technology, pp.386-390, 2004.

[10] IDS and IPS, http://www.kisa.or.kr.

[11] IDS (Intrusion Detection System), http://www.helloec.net/network/IDS.htm

[12] M. Yasin and A. Awan, "A study of host-based IDS using system calls," International Conference on Networking and Communication, pp.36-41, 2004.

[13] C. Herringshaw, "Detecting attacks on networks," IEEE Computer, Vol.30, No.12, pp.16-17, 1997.

[14] B. Mukherjee, L. Heberlein, and K. Levitt, "Network intrusion detection," IEEE Network, Vol.8, No.3, pp.26-41, 1994.

[15] Network-vs. Host-based Intrusion Detection System, http://www.documents.iss.net/whitepapers/nvh_ids.pdf

[16] Netfilter, http://www.netfilter.org

[17] Snort, http://www.snort.org

[18] N. Desai, "Increasing Performance in High Speed NIDS - A look at Snort's Internals," In www.linuxsecurity.com, 2002.

[19] C. Coit, S. Staniford, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," DARPA Information Survivability Conference and Exposition, 2001.

[20] 김형주, 박대철, "고성능 침입탐지 및 대응 시스템의 구현 및 성능 평가", 정보처리학회논문지C, Vol.11-C, No.2, pp.157-162, 2004.

[21] 강구홍, 김익균, 장종수, "고속 망에 적합한 네트워크 프로세서 기반 인-라인 모드 침입탐지 시스템", 정보과학회논문지 : 정보통신, Vol.31, No.4, pp.363-374, 2004.

[22] 이장행, 황성호, 박능수, "FPGA를 사용한 네트워크 침입탐지 시스템의 문자열 비교", 제32회 추계학술발표회 논문집, 한국정보과학회, Vol.32, No.2, pp.886-888, 2005.

[23] 안철수 연구소, http://info.ahnlab.com/securityinfo/virus.jsp.

[24] Application Layer Packet Classifier for Linux, http://l7-filter.sourceforge.net.

곽 후 근

E-mail : gobarian@q.ssu.ac.kr

1996년 호서대학교 전자공학과(학사)

1998년 숭실대학교 전자공학과 대학원 (석사)

1998년~2006년 숭실대학교 전자공학과 대학원(박사)

1998년 8월~2000년 7월 (주)3R 부설 연구소 주임연구원

2006년 3월~현재 숭실대학교 전자공학과 대학원(postdoc)

관심분야 : 네트워크 컴퓨팅 및 보안

정 규 식

E-mail : kchung@q.ssu.ac.kr

1979년 서울대학교 전자공학과(공학사)

1981년 한국과학기술원 전산학과 (이학석사)

1986년 미국 University of Southern California(컴퓨터공학석사)

1990년 미국 University of Southern California (컴퓨터공학박사)

1998년 2월~1999년 2월 미국 IBM Almaden 연구소 방문 연구원

1990년 9월~현재 숭실대학교 정보통신전자공학부 교수

관심분야 : 네트워크 컴퓨팅 및 보안