# 네트워크 프로세서 기반 고성능 네트워크 침입 탐지 엔진에 관한 연구

## (An Implementation of Network Intrusion Detection Engines on Network Processors)

조 혜 영 † 김 대 영 ††
(Hyeyoung Cho) (Daeyoung Kim)

**요 약** 초고속 인터넷 망이 빠른 속도로 구축이 되고, 네트워크에 대한 해커나 침입자들의 수가 급증함에 따라, 실시간 고속 패킷 처리가 가능한 네트워크 침입 탐지 시스템이 요구되고 있다. 본 논문에서는 일반적으로 소프트웨어 방식으로 구현된 침입 탐지 시스템을 고속의 패킷 처리에 뛰어난 성능을 가지고 있는 네트워크 프로세서를 이용하여 재설계 및 구현하였다. 제한된 자원과 기능을 가지는 다중 처리 프로세서(Multi-processing Processor)로 구성된 네트워크 프로세서에서 고성능 침입 탐지 시스템을 실현하기 위하여, 최적화된 자료구조와 알고리즘을 설계하였다. 그리고 더욱 효율적으로 침입 탐지 엔진을 스케줄링(scheduling)하기 위한 침입 탐지 엔진 할당 기법을 제안하였으며, 구현과 성능 분석을 통하여 제안된 기법의 적절성을 검증하였다.

**키워드** : 침입 탐지 시스템, 네트워크 프로세서, Snort, IXP1200

***Abstract*** Recently with the explosive growth of Internet applications, the attacks of hackers on network are increasing rapidly and becoming more seriously. Thus information security is emerging as a critical factor in designing a network system and much attention is paid to Network Intrusion Detection System (NIDS), which detects hackers' attacks on network and handles them properly. However, the performance of current intrusion detection system cannot catch the increasing rate of the Internet speed because most of the NIDSs are implemented by software. In this paper, we propose a new high performance network intrusion using Network Processor. To achieve fast packet processing and dynamic adaptation of intrusion patterns that are continuously added, a new high performance network intrusion detection system using Intel's network processor, IXP1200, is proposed. Unlike traditional intrusion detection engines, which have been implemented by either software or hardware so far, we design an optimized architecture and algorithms, exploiting the features of network processor. In addition, for more efficient detection engine scheduling, we proposed task allocation methods on multi-processing processors. Through implementation and performance evaluation, we show the proprieties of the proposed approach.

**Key words** : Intrusion Detection System, Network Processor, Snort, IXP1200

## 1. 서 론

최근 인터넷 사용이 활발해 지면서, 네트워크를 통한 해커들의 공격이 급속히 증가하고 또한 그 심각성이 날로 심해지고 있다. 이에 정보보호의 중요성이 더욱 부각되고 있으며, 이러한 네트워크를 통한 해커들의 공격을 탐지하고 그에 대응하기 위한 네트워크 침입 탐지 시스템(Network Intrusion Detection System)에 대한 연구가 매우 활발히 진행되고 있다. 그러나 초고속 인터넷 구축을 위한 네트워크 관련 기술들이 테라급으로 급속히 발전하고 있는데 비해, 상대적으로 침입 탐지 기술들은 네트워크의 발전 속도를 따라가지 못하고 있는 실정이다. 그 이유는 대부분의 네트워크 침입 탐지 시스템들이 소프트웨어로 구현되어 있다는 데 있다.

침입 탐지 엔진(Intrusion Detection Engine)은 침입 탐지 시스템의 성능을 결정하는 가장 중요한 요소이다

† 정 회 원 : 한국과학기술정보연구원 슈퍼컴퓨팅센터 연구원
chohy@kisti.re.kr
†† 정 회 원 : 한국정보통신대학교 공학부 교수
kimd@icu.ac.kr
논문접수 : 2005년 5월 24일
심사완료 : 2005년 11월 28일

침입 탐지 엔진은 구현 방식에 따라 소프트웨어 기반 침입 탐지 엔진과 하드웨어 기반 침입 탐지 엔진으로 분류할 수 있다. Snort나 Hogwash와 같은 소프트웨어 기반 침입 탐지 엔진은 침입 탐지 시스템의 새로운 룰 시그너처를 쉽게 추가하고 변경할 수 있는 장점을 가진 반면에, 성능에 한계를 가지고 있다. 따라서 소프트웨어 기반 침입 탐지 엔진에서의 성능 향상을 위한 연구로 패킷과 이미 알려진 침입을 표현하는 한 셋트의 규칙(시그너처: Signature)을 비교하는 패턴/스트링 매칭 알고리즘의 개선, 다중 컴퓨터를 사용한 Load Balancing, 트래픽 센서 앞단에 Splitter을 사용하는 방법들이 연구되고 있다[1,2]. 반면 하드웨어 기반 침입 탐지 엔진은 침입 탐지 엔진 알고리즘을 하드웨어로 구현하여 처리 속도를 향상시킨 방식이다. 예를 들면, 침입 탐지 엔진 알고리즘을 ASIC 형태로 제공하거나 빠른 패턴 매칭을 위하여 NFA(Non-Deterministic Finite Automata)를 사용하는 등의 연구들이 진행되었다[3,4]. 그러나 하드웨어 기반 침입 탐지 엔진 방식은 패킷을 고속으로 처리할 수 있는 장점을 가진 반면에 계속 업그레이드가 필요한 룰 시그너처(Rule Signature) 데이타베이스를 효율적으로 관리하기가 힘든 단점이 있다.

본 논문에서는 고속 패킷 프로세싱 기능을 강화한 특화된 구조를 가지고 있으며, 다양한 프로토콜을 수용할 수 있도록 소프트웨어 프로그래밍이 가능한 네트워크 프로세서(Network Processor)를 사용하여 침입 탐지 시스템(Intrusion Detection System)을 구현함으로써, 소프트웨어와 하드웨어 기반 침입 탐지 엔진의 장점인 유연성과 고성능을 모두 추구할 수 있다. 구현된 네트워크 프로세서 기반 침입 탐지 시스템(NP-IDS)은 현재 가장 널리 사용되고 있는 공개 소스 네트워크 침입 탐지 시스템(Network Intrusion Detection System)인 Snort 구조를 모델로 하였으며, Snort에서 지원하는 룰 시그너처를 그대로 포팅없이 지원한다. 기존의 소프트웨어나 하드웨어적으로 구현되던 침입 탐지 엔진 알고리즘을 제한된 기능을 가지는 다중 처리 프로세서(Multi-processing Processor)로 구성된 네트워크 프로세서에서 실현하기 위하여, 최적화된 자료구조와 알고리즘 설계하였다. 그리고 더욱 효율적으로 탐지 엔진을 스케줄링(scheduling)하기 위한 탐지 엔진 할당 기법을 제안하였으며, 구현과 성능 분석을 통하여 제안된 기법의 적절성을 검증하였다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구를 통해 기존 침입 탐지 시스템과 네트워크 프로세서에 대해 분석하고, 3장에서는 네트워크 프로세서를 기반으로 하는 고속 네트워크 침입 탐지 시스템의 전체 구조를 설계한다. 4장에서는 침입 탐지 시스템에서 핵심 기술인 침입 탐지 엔진을 상세 설계하고 5장에서는 침입 탐지 엔진 할당 기법을 제안한다. 6장에서는 구현된 침입 탐지 시스템의 성능을 분석하고 4장에서 제안한 침입 탐지 엔진 할당 기법의 성능을 분석, 그 적절성을 검증한다. 마지막으로 7장에서는 결론과 향후 연구 계획으로 끝을 맺는다.

## 2. 관련연구

침입 탐지 시스템은 네트워크나 시스템으로부터 비정상적인 사용, 오용, 남용 등 미심쩍은 점을 조사 및 감시하여 침입 및 침입시도의 징후를 찾아내고, 필요한 조치를 취하는 시스템이다[5]. 침입 탐지 시스템은 원시 데이타의 근원지에 따라 호스트 기반 방식과 네트워크 기반 방식 시스템으로 구분된다. 그리고 탐지 분석 방법에 따라 오용 기반 침입 탐지 방식과 비정상 기반 침입 탐지 시스템으로 나누어 질 수 있다. 오용 기반 침입 탐지 방식은 미리 정의된 침입 정보에 따라 비정상 패킷을 탐지하는 시그너처 분석 방법을 많이 쓰고 있고, 비정상 기반 침입 탐지는 침입 정보가 없는 미상의 침입 공격을 탐지 하는 기법으로 통계적 방법, 데이타 마이닝 기법, 또는 전문가 시스템 등을 많이 사용한다. 본 논문에서는 네트워크 기반 방식의 시그너처 분석 방식을 기본으로 채택한다[6,7].

본 논문의 침입 탐지 시스템은 현재 널리 사용되고 있는 네트워크 침입 탐지 시스템 중에 하나인 Snort를 참고 모델로 하였다. Snort는 공개 소프트웨어이며, 다양한 플랫폼에서 사용될 수 있고, 성능이 우수하며, 유연성이 뛰어난 장점을 가지고 있다. Snort는 이미 알려진 침입 탐지 규칙(Rule Signature)과 패킷을 비교하여 침입을 탐지하고, 규칙에 기술 된 경고 조건에 따라, 일반 로그, tcpdump 포맷 로그, 실시간 경과, 윈도우즈 팝업 등 다양한 포맷으로 침입을 알려준다[8].

alert tcp 1.1.1.1 any -> 2.2.2.2 any
(flages:F; msg:"FIN Scan";)

위에 보인 룰의 예는 패킷이 TCP 패킷이고, 소스 주소가 1.1.1.1이고 목적지 주소가 2.2.2.2이며, 소스의 포트와 목적지의 포트 수에는 관계없이 패킷의 F flages 비트가 셋팅되어 있는 패킷을 탐지한다. 이런 패킷이 발견되면 Snort는 “FIN Scan”이라는 메시지 포함한 경고 시그널을 보낸다. Snort의 패턴 매칭 알고리즘은 처음에는 부분적인 Boyer-Moore 패턴 매칭 알고리즘을 사용하여 구현되었다[9]. 그 후 되풀이 되는 노드를 사용하는(recursive node walking) 2차원 링크 리스트를 이용하는 방법이 도입되었고, 이 방법을 통해 snort 성능을 200~500% 높였다. 현재는 함수 포인터의 링크 리스트, 즉 3차원 링크 리스트를 사용하고 있다. 그리고 snort는

패킷을 캡쳐하기 위해서 libpcap을 사용한다[1].

네트워크 프로세서는 네트워크 프로토콜 처리 성능 향상을 위해 패킷 처리 기능을 강화한 특화된 구조를 가지며, 다양한 프로토콜을 수용할 수 있도록 설계된 프로그래밍 가능한 네트워크 전용 프로세서이다. 일반적으로 네트워크 프로세서는 현 네트워크상에 존재하는 다양한 패킷을 보다 효율적이고 신속하게 처리할 수 있는 다중 패킷 처리 구조를 제공한다. Intel, Agere(루슨트), Vittesse, IBM 등에서 상용 네트워크 프로세서를 공급하고 있으며[10-14], Network Processor Forum을 통해 표준화 작업들을 수행하고 있다[15].

최근 고속의 패킷 처리에 특화된 네트워크 프로세서를 이용한 연구들이 활발히 진행되고 있다. 네트워크 프로세서를 이용하여 스위치, 라우터 및 방화벽 등의 네트워크 디바이스에서 고속으로 패킷을 분류하게 하거나[16-18], IPv4/IPv6의 주소 변환에 네트워크 프로세서를 이용하는 등의 연구가 진행되고 있다[19]. 또한 네트워크 프로세서를 이용하여 TCP/IP Offload Engine을 구현함으로써 호스트 프로세서의 오버헤드를 감소시키는 연구[20-22], 암호화 알고리즘을 네트워크 프로세서에 구현하는 연구 이루어지고 있다[23].

본 논문에서는 인텔의 네트워크 프로세서인 IXP1200을 이용하여 침입 탐지 시스템을 구현하였다. 그림 1에 도시된 IXP1200은 StrongArm과 6개의 마이크로엔진 그리고 SDRAM, SRAM, PCI 버스 인터페이스로 구성된다. 6개의 마이크로엔진은 각기 4개의 하드웨어 쓰레드를 제공하며, 이는 고속 다중 패킷 처리에 효율적인 구조이다. 마이크로엔진의 수행능력은 ASIC의 속도보다 조금 떨어지지만 초당 300만개이상의 이더넷 패킷을 처리할 수 있을 정도의 처리 능력을 가지고 있으며, 고속의 패킷이나 프로토콜 처리에 효율적이다[24].

## 3. 네트워크 프로세서 기반 네트워크 침입 탐지 시스템

본 논문에서 제안한 네트워크 프로세서를 이용한 침입 탐지 시스템은 기존에 호스트 프로세서에서 소프트웨어 방식으로 구현되던 기능들을 고속의 패킷 처리 능력을 가지고 있는 네트워크 프로세서를 활용하여 효율적으로 처리한다.

인텔 IXP1200 네트워크 프로세서를 활용한 침입 탐지 시스템의 구조는 그림 2와 같이 호스트 프로세서, StrongArm(IXP1200), 마이크로 엔진(IXP1200)의 3단계 계층 구조를 가진다[25]. 호스트 프로세서는 시스템과 전체 룰 시그너처를 관리하고 상위 네트워크 관리시스템과의 인터페이스를 제공한다. StrongArm에서는 호스트 프로세서와 마이크로엔진 사이에서 중재 기능을 담당하고, 호스트 프로세서로부터 다운받은 룰 시그너처를 마이크로엔진 구조에 알맞게 분류 관리한다. 또한 마이크로엔진을 관리하고 보조하는 역할을 한다. 마이크로엔진에서는 이더넷 프레임의 수신, 송신, 에러 체크 등 이더넷 MAC, 디바이스 드라이버의 역할과 IP 패킷의 수신 및 송신, 에러 체크 등을 담당한다. 또한 침입 탐

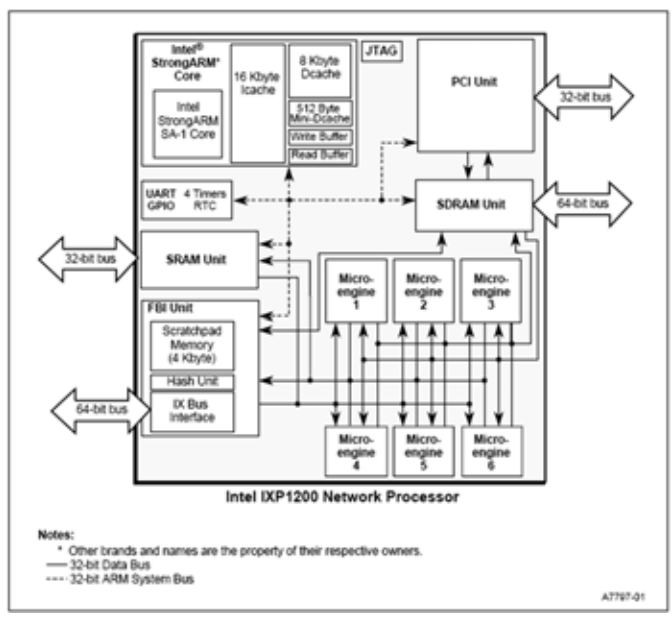


그림 1 네트워크 프로세서 IXP1200 블록 다이어그램

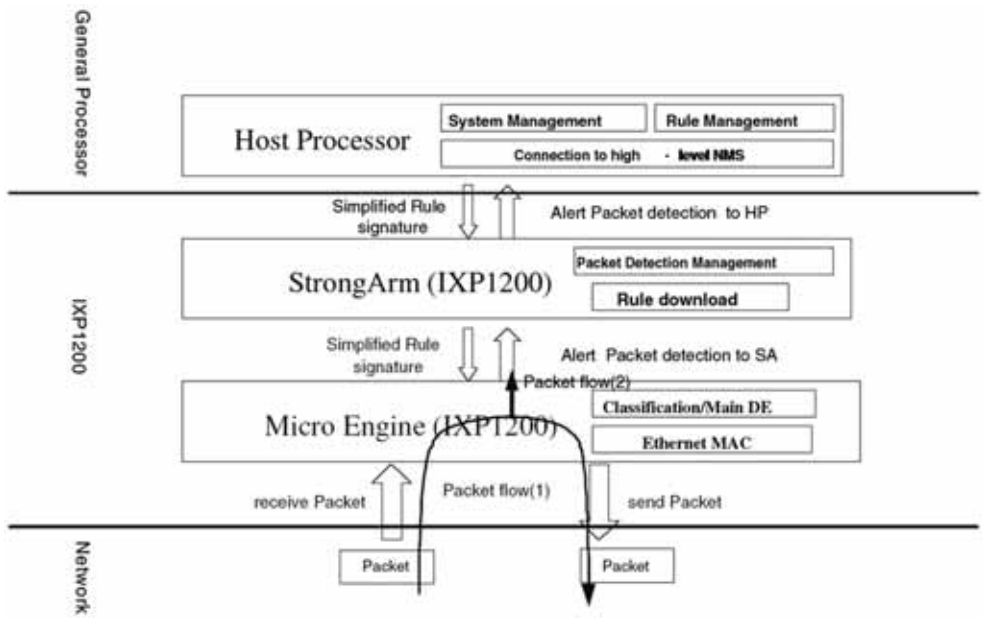


그림 2 IXP1200 네트워크 프로세서 기반 침입 탐지 시스템의 구조

지를 위한 시그너처를 호스트 프로세서와 StrongArm을 통해서 받고, 이 시그너처 데이타베이스와 네트워크 상에 수신되어 분류된 패킷을 비교하여, 침입을 탐지한다.

IXP12000 네트워크 프로세서를 기반으로 하는 침입 탐지 시스템(Network Processor-based Intrusion Detection System, NP-IDS)은 침입을 탐지하고자 하는 대상 네트워크 상에 존재하면서 수상한 패킷이 있는지 탐지한다. 네트워크 프로세서의 MAC 디바이스로부터 수신된 패킷을 저장된 룰 시그너처 데이타베이스에서 검색하여, 침입 여부를 판정한다. 이상이 없을시(그림 2의 Packet flow(1)), 내부 망으로 패킷을 수정 없이 전달하고, 침입 탐지시(그림 2의 Packet flow(2)), StrongArm이 호스트에 경고 메시지를 보낸다.

## 4. 고속 네트워크 침입 탐지 엔진

침입 탐지 엔진은 수신된 패킷을 저장된 시그너처 데이타베이스를 검색하여 침입을 탐지하는 시간 소모적인 알고리즘을 수행하며, 침입 탐지 시스템의 성능을 결정하는 주요한 요인이다. 네트워크 프로세서 기반 침입 탐지 시스템(NP-IDS)에서 침입 탐지 엔진은 IXP1200의 마이크로엔진에서 하나의 하드웨어 쓰레드로 구현된다. IXP1200에는 6개의 마이크로엔진이 존재하며, 각 마이크로 엔진마다 4개의 하드웨어 쓰레드를 가지므로 총 24개의 쓰레드를 사용할 수 있다. IXP1200의 다음 버전인 IXP2400의 경우 8개의 마이크로 엔진에 각 8개의 하드웨어 쓰레드를 지원하므로 총 64개를 사용할 수 있으며, IXP2800의 경우 16개의 마이크로 엔진을 가지고 있으므로 총 128개의 하드웨어 쓰레드가 가용하다.

그림 3은 침입 탐지 엔진의 기능을 간단히 나타낸 것이다. 침입 탐지 엔진은 네트워크로부터 패킷을 수신하여 프로토콜에 따라 분류하고, 해당 프로토콜에 대하여 미리 정의된 침입 정보를 가지고 있는 시그너처 데이타베이스와 비교하여 수상한 패킷인지 결정한다. 해당 패킷이 수상하다고 결정되면 StrongArm에게 알리고, 그렇지 않으면 네트워크로 포워드한다.

네트워크 상에는 TCP, IP, UDP, ICMP 등 다양한 프로토콜의 패킷이 존재하며, 네트워크를 침입하는 방법도 각 프로토콜마다 다양한 패턴이 존재한다. 따라서 각각의 프로토콜에 대한 침입을 탐지하기 위하여 다양한 시그너처들을 정의하고 있다. 이러한 시그너처를 관리하고 시그너처를 이용하여 침입을 탐지하는 엔진의 설계에 있어서, 하나의 침입 탐지 엔진에서 네트워크 상의 모든 프로토콜에 대해서 침입을 탐지하고 관리하느냐, 아니면 탐지 엔진을 프로토콜별로 역할을 나누어 침입을 탐지하고 관리하느냐에 따라, 통합형 침입 탐지 엔진(Combined Intrusion Detection Engine, CIDE)과 분산형 침입 탐지 엔진(Distributed Intrusion Detection Engine, DIDE)을 제안하였다. 그리고 각 탐지 엔진은 룰 매칭을 순차적으로 하느냐 병렬적으로 하느냐에 따라 각각 순차(Serial)과 병렬(Parallel) 침입 탐지 엔진으로 나뉜다. 따라서, 침입 탐지 엔진은 통합형 순차 침입 탐지 엔진(CIDE-S), 통합형 병렬 침입 탐지 엔진(CIDE-P),

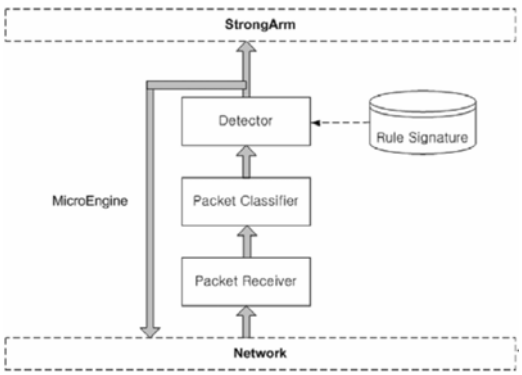


그림 3 침입 탐지 엔진 블록 다이어그램

분산형 순차 침입 탐지 엔진(DIDE-S), 분산형 병력 침입 탐지 엔진(DIDE-P)의 4가지로 구현될 수 있다.

### 4.1 통합형 침입 탐지 엔진

통합형 침입 탐지 엔진 방식에서는 하나의 탐지 엔진이 TCP, IP, UDP, ICMP 등 네트워크의 모든 프로토콜에 대해서 침입을 탐지하고 관리한다. 그러므로 통합형 침입 탐지 엔진 방식에서는 IXP1200의 모든 마이크로 엔진에서는 같은 침입 탐지 프로그램이 수행된다. 그림 4는 통합형 침입 탐지 엔진 방식에서의 패킷 처리 절차를 나타낸 것으로 절차는 다음과 같다.

(1) IXP1200의 침입 탐지 엔진은 MAC 디바이스로부터 패킷을 수신 받는다.

(2) 수신 받은 패킷을 TCP, IP, UDP, ICMP 프로토콜 종류에 따라 패킷을 분류한다. 패킷 분류는 IP 헤더의 프로토콜 필드 값에 따라 분류하며, TCP(0x06), UDP(0x11), ICMP(0x01) 가 아니면, Snort에서와 같이 IP로 분류된다.

(3) 침입 탐지 엔진은 수신된 패킷의 종류에 해당하는 시그너처를 읽어서 패킷의 정보와 비교한다. 예를 들어 수신된 패킷이 TCP 패킷이면 TCP 시그너처 데이타베이스와 비교한다.

(4) 시그너처 데이타베이스에서 수신된 패킷의 정보와 일치하는 룰을 찾지 못하면 패킷을 포워딩한다.

(5) 시그너처의 정보와 일치하는 패킷을 발견하면 StrongArm에게 알린다. 이 때 룰 번호와 패킷의 내용을 StrongArm에게 보낸다.

(6) StrongArm은 룰 번호와 패킷 정보를 호스트 프로세서에게 전달한다.

(7) 호스트 프로세서는 받은 룰 번호와 패킷 정보를 이용하여 관리 파일에 기록하거나, 메시지를 출력하든지, 상위 NMS(Network Management System)에 알리는 등 적절한 액션을 취한다.

통합형 침입 탐지 엔진 방식은 탐색 엔진이 모든 프로토콜에 대해서 침입을 탐지하고 관리하기 때문에 work conserving하다는 장점이 있다. 즉, 유휴(idle)한 마이크로엔진이 있는 한 수신 패킷은 바로 서비스를 받을 수 있다. 반면, TCP, IP, UDP, ICMP 등 여러 종류

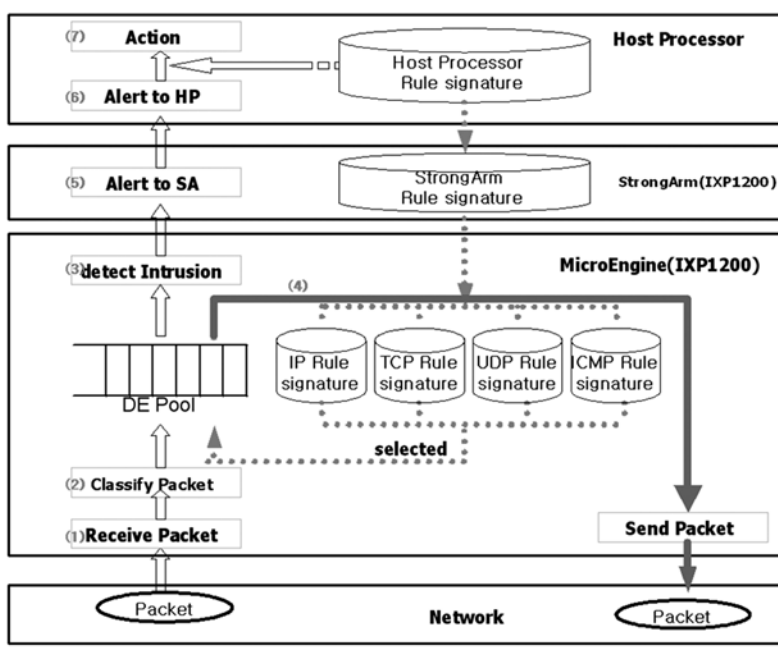


그림 4 통합형 침입 탐지 엔진 패킷 처리 절차도

의 프로토콜에 대해서 침입 탐지 룰을 수용할 수 있어야 하기 때문에, 시그너처 데이타베이스의 구조가 복잡하고, 수신 패킷당 처리 시간이 길다는 단점이 있다. IXP1200의 경우 한 마이크로 엔진 당 4KB(1KB*32 bit) Control Store, 128 General Purpose 레지스터(32bit), SDRAM과 SRAM과의 통신을 위한 128 Transfer 레지스터(32bit)를 가지고 있다. 이 자원을 4개의 하드웨어 쓰레드가 공유하므로 프로그래밍에 제한이 있다. 그러나 IXP2400의 경우 한 마이크로엔진 당 4K 개의 인스트럭션이 가능하며, 256 General Purpose 레지스터, 512 Transfer 레지스터를 가지므로, 통합형 침입 탐지 시스템의 적용이 가능하다.

그림 5는 CIDE-S 침입 탐지 엔진의 패킷 처리 과정을 나타낸 것이다. 엔진 E1과 E2는 모든 프로토콜을 처리할 수 있는 침입 탐지 엔진이다. 패킷은 P1(TCP), P2(IP), P3(TCP), P4(ICMP)의 순서로 패킷이 수신된다. CIDE는 DE(Detection Engine)이 모든 프로토콜을 처리할 수 있으므로 하나의 Pool로 관리된다. 먼저 수신된 P1은 DE Pool의 E1이 처리를 맡는다. E1은 P1과 TCP 시그너처를 시그너처 개수 n만큼 순차적으로 비교한다. IP 패킷 P2는 E2에 보내어 지고, E2는 P2와 IP 시그너처를 시그너처 개수 m만큼 순차적으로 비교한다. E1과 E2 중 한 탐지 엔진이 작업이 끝나면, 다음 패킷을 처리한다. 즉 유휴(idle)한 탐지 엔진이 있는 한 수신된 패킷은 바로 서비스를 받을 수 있다.

그림 6은 CIDE-P 침입 탐지 엔진의 패킷 처리 과정을 나타낸 것이다. 엔진 E1, E2는 모든 프로토콜을 처리하는 CIDE 엔진이다. 프로토콜 종류에 따라 비교해야 될 시그너처 개수가 다르기 때문에, TCP와 같이 많은 룰을 비교해야 하는 경우, 보다 많은 탐지 엔진을 할당하여, 룰 검색이 병렬로 수행될 수 있게 한다. 그림 6에서와 같이 TCP 패킷에 대하여 두 개의 탐지 엔진이 할당될 경우, 각 엔진은 시그너처를 반으로 나누어 동시에 검사한다.

### 4.2 분산형 침입 탐지 엔진

분산형 침입 탐지 엔진 방식은 프로토콜별로 전용 마이크로엔진 쓰레드를 할당하여 탐지 엔진을 운영하는 방식이다. 즉 하나의 탐지 엔진은 네 개의 프로토콜 중 하나만 처리하도록 할당되고, 각 마이크로엔진에는 서로 다른 탐지 엔진 프로그램이 탑재되어 실행된다. Snort의 경우 TCP, IP, UDP, ICMP 네 가지 종류의 탐지 엔진이 독립적으로 구현되어 있다[26]. 그림 7은 분산형 침입 탐지 엔진 방식에서의 패킷 처리 절차를 나타낸 것이며 그 절차는 다음과 같다.

(1) IXP1200의 침입 탐지 엔진은 MAC 디바이스로부터 패킷을 수신 받는다.

(2) 수신 받은 패킷을 TCP, IP, UDP, ICMP 프로토콜 종류에 따라 패킷을 분류한다. 패킷 분류는 IP 헤더의 프로토콜 필드 값에 따라 분류하며, TCP(0x06), UDP(0x11), ICMP(0x01) 가 아니면, Snort에서와 같이 IP로 분류된다.

(3) 분류된 패킷은 해당 프로토콜을 처리하는 전용 탐지 엔진에 의해 분석된다. 침입 탐지 엔진은 수신된 패킷의 프로토콜 종류에 해당하는 시그너처를 읽어서 패킷의 정보와 비교한다. 예를 들어 수신된 패킷이

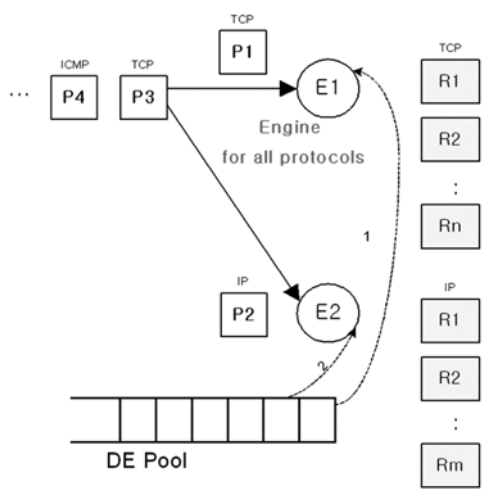


그림 5 CIDE-S 패킷 처리

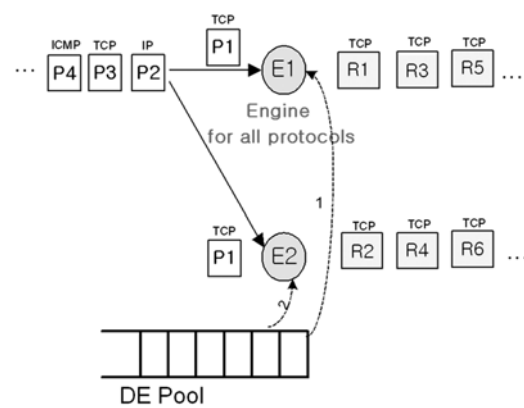


그림 6 CIDE-P 패킷 처리

(P: Packet, E: Engine, R: Rule)

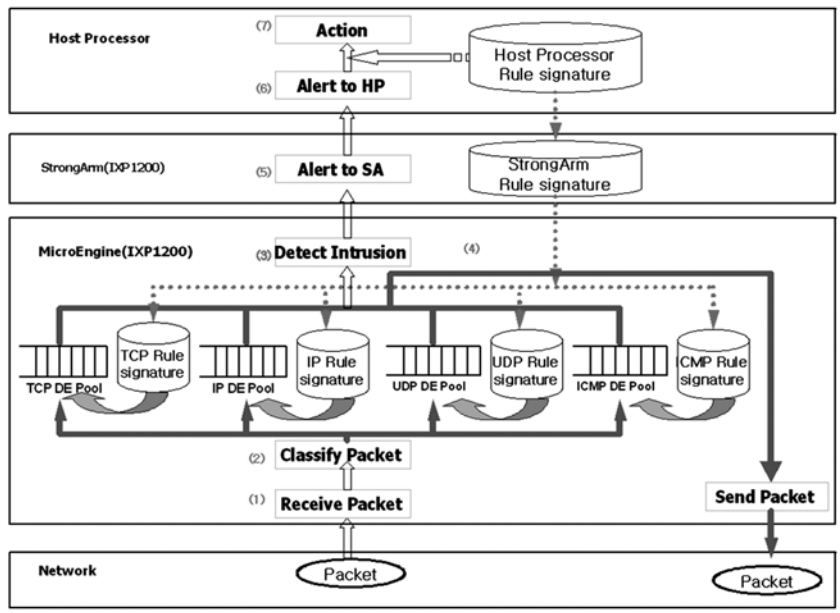


그림 7 분산형 침입 탐지 엔진 패킷 처리 절차도

TCP 패킷이면 TCP 시그너처 데이타베이스의 룰과 비교한다.

(4) 시그너처 데이타베이스에서 수신된 패킷의 정보와 일치하는 시그너처를 찾지 못하면 패킷을 포워딩한다.

(5) 시그너처의 정보와 일치하는 패킷을 발견하면 StrongArm에게 침입 탐지를 알린다. 이 때 룰 번호와 패킷의 내용을 StrongArm에 함께 보낸다.

(6) StrongArm은 룰 번호와 패킷 정보를 호스트 프로세서에게 전달한다.

(7) 호스트 프로세서는 받은 룰 번호와 패킷 정보를 이용하여 기록하거나, 메시지를 출력하든지, 상위 NMS (Network Management System)에 알리는 등 적절한 액션을 취한다.

분산형 침입 탐지 엔진 방식은 프로토콜별로 다른 룰 시그너처를 사용하기 때문에 룰 시그너처 데이타베이스 구조가 간단하고, 각 프로토콜에 따라 최적화될 수 있다는 장점이 있다. 또한 프로토콜별 룰 시그너처에 따라 탐지 엔진이 최적화 되어서 프로그램 코드 길이가 짧고, 수신 패킷을 처리하는 속도가 빠르다. 반면, 프로토콜별로 전용 마이크로엔진 쓰레드를 할당하여 탐지 엔진을 운영하기 때문에, work conserving하지 못하다는 단점이 있다. 즉, 해당 프로토콜로 할당된 탐지 엔진이 모두 동원되고 유휴(idle)한 탐지 엔진이 없을 시, 다른 프로토콜을 위한 탐지 엔진이 유휴(idle)하더라도 대기해야 한다.

통합형의 경우 한 엔진이 모든 프로토콜의 처리가 가능한 homogeneous 탐지 엔진이므로, 침입 탐지 엔진 pool이 하나로 관리되는 반면, 분산형은 프로토콜에 따라 침입 탐지 엔진 pool이 따로 관리된다. 따라서 분산형 방식은 프로토콜별 침입 탐지 엔진을 하드웨어 쓰레드에 할당하는 방식에 따라 침입 탐지 시스템의 성능이 결정된다. DIDE-P의 할당 방법에 5장에서 설명하였다.

분산형도 통합형과 마찬가지로 시그너처와 패킷을 비교할 때, 룰을 순차적(serial)으로 비교하는 방법과 여러 개의 룰을 동시에(parallel) 비교하는 방법에 따라 나뉜다.

그림 8은 DIDE-S 침입 탐지 엔진의 패킷 처리 과정을 나타낸 것이다. 분산형 탐지 엔진 방식에서는 프로토콜별로 DE(Detection Engine) Pool이 따로 관리된다. 패킷이 P1(TCP), P2(IP), P3(IP), P4(ICMP)의 순서로 수신된다. P1은 TCP 타입이므로 TCP DE Pool의 탐지엔진 E1에게 할당되며, P2는 IP 타입이므로 IP DE Pool의 탐지 엔진 E2에게 할당된다. E1은 TCP 룰을 순차적으로 n개만큼 비교한다. P2(IP 패킷)는 IP DE Pool의 E2에게 할당되며, E2는 P2와 IP 룰을 m개만큼 비교한다.

그림 9는 DIDE-P 침입 탐지 엔진의 패킷 처리 과정을 나타낸 것이다. 엔진 E1, E2는 TCP 프로토콜을 처리하기 위한 전용 탐지 엔진이다. 패킷이 P1(TCP), P2(IP), P3(TCP), P4(ICMP)의 순서로 수신된다. 여기서, P1이 TCP 타입이므로 TCP DE Pool의 적절한 수의 탐지엔진이 시그너처를 분산하여 동시에 검사한다.

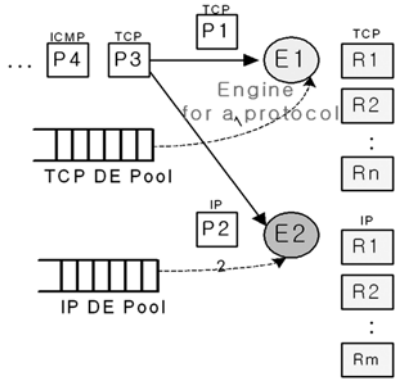


그림 8 DIDE-S 패킷 처리

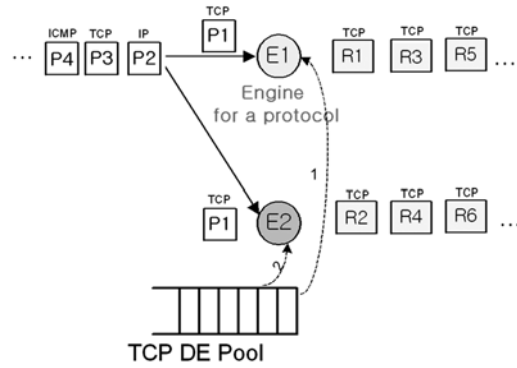


그림 9 DIDE-P 패킷처리

(P: Packet, E: Engine, R: Rule)

그림 9에서는 엔진이 2개이므로 E1은 룰 R1, R3, R5 ... 를 처리하고, E2는 룰, R2, R4, R6 ...을 처리한다. 다음 P2(IP)가 수신되면, IP DE Pool에서 적절한 수의 탐지 엔진이 시그너처를 나누어 동시에 검사한다.

### 4.3 분산형 침입 탐지 엔진 설계

본 장에서는 네트워크 프로세서 IXP1200을 이용한 분산형 침입 탐지 엔진(NP-NIDE)을 구현하기 위하여 설계된 프로그램의 상세 구조에 대해서 알아본다. ACE (Active Computing Element)란 Intel에서 제안하는 프로그래밍 모델로 인텔 네트워크 프로세서의 프로그래밍의 기반이 된다. 그림 10과 표 1에 고성능 침입 탐지 시스템을 위해 네트워크 프로세서 IXP1200을 이용하여 설계된 NP-NIDE ACE 프레임워크 구조와 각 프로그램 블록의 기능을 설명하였다.

그림 11은 분산형 침입 탐지 엔진에 대한 블록도를 나타낸다. Input Interface 마이크로 블록 네트워크의 패킷을 수신하고, 이 패킷을 Classification 마이크로 블록이 IP 헤더의 프로토콜 필드에 따라 패킷을 분류한다. 분류된 패킷은 각 프로토콜 담당 탐지 엔진에 의해 룰과 비교하여 수상한 패킷이 있는지 체크된다. 수상한 패킷이 발견되면 Detection Interface ACE를 통해 Strong-Arm과 호스트에게 경고 메시지를 보낸다. 정상적인 패킷일 경우 Output Interface 마이크로 블록이 패킷을 네트워크로 포워드 한다.

## 5. 침입 탐지 엔진 할당 기법

분산형 탐지 엔진 방식은 work conserving이 아니므로, 4종의 네트워크 프로토콜에 대한 침입 탐지 엔진을

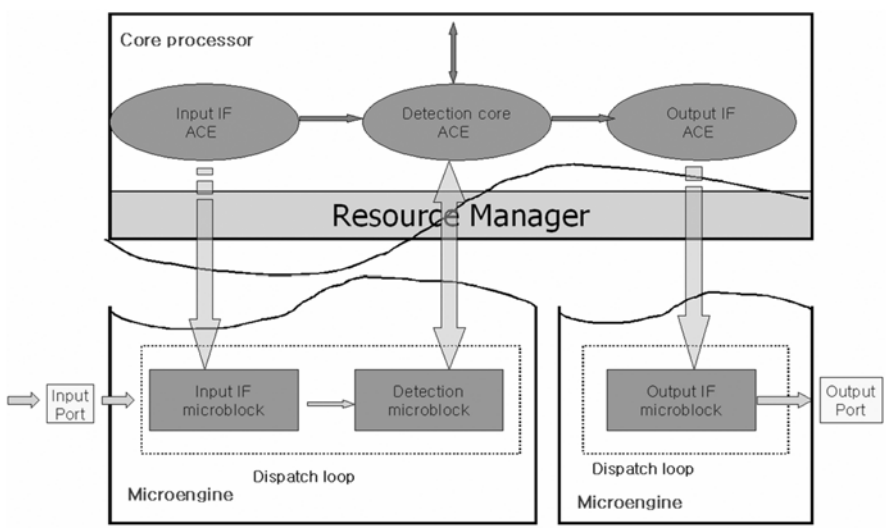


그림 10 NP-NIDE ACE 프레임워크 설계 구조

표 1 NP-NIDE ACE 프레임워크의 블록 설명

| 이 름 | 정 의 |
|---|---|
| Input IF microblock | MAC 입력 포트로부터 수신된 패킷을 mpacket(64바이트) 단위로 SDRAM의 패킷 버퍼에 저장하는 역할을 담당한다. |
| Input IF ACE | 각 입력 포트에 대한 물리적, 논리적 설정을 관리하며 네트워크 트래픽에 관한 통계를 담당한다. |
| Detection microblock | Input IF microblock에 의해서 버퍼에 저장된 패킷을 마이크로엔진의 레지스터로 복사하여 SRAM의 시그너처와 패킷을 비교하여 침입을 탐지한다. |
| Detection core ACE | Detection microblock 패킷 처리 시, 침입을 탐지한 상황을 관장하는데, Detection microblock에서 수상한 패킷이 탐지되었을 경우에는 Detection core ACE는 호스트 프로세서에게 경고 메시지를 보내는 역할을 담당한다. |
| Output IF microblock | 네트워크 프로토콜 처리를 끝낸 패킷을 SDRAM의 패킷 버퍼에서 MAC 송신 포트의 송신 버퍼로 옮기는 기능을 수행한다. |
| Output IF ACE | 패킷을 TX Proc microblock으로부터 output I/F microblock으로 연결시켜주며, 물리 포트의 특성, 송신상태 정보를 관리한다. |

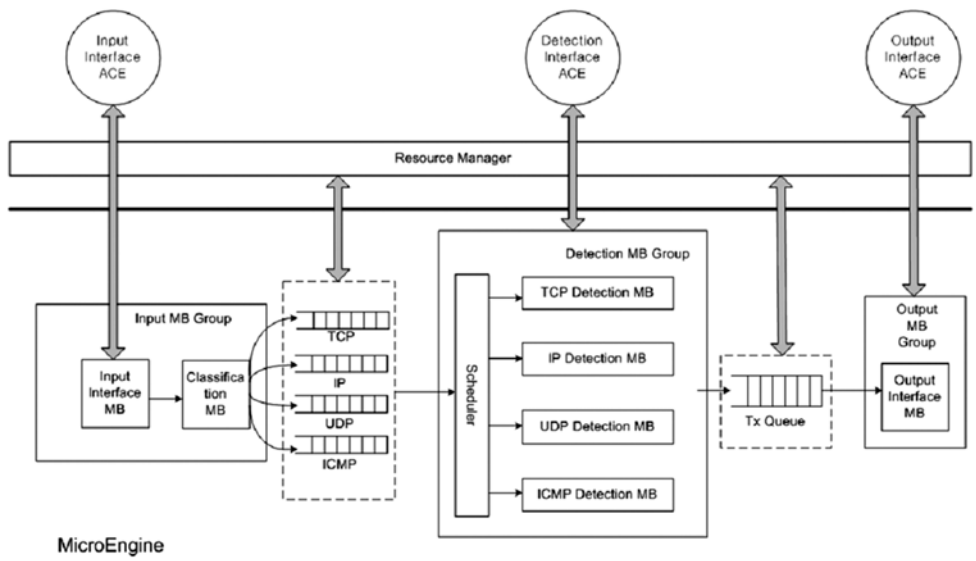


그림 11 분산형 침입 탐지 엔진 블록도

구현하기 위하여 각 프로토콜에 하드웨어 쓰레드를 최적으로 할당하는 것이 이용도(utilization)의 측면에서 매우 중요하다. IXP1200에서는 총 24개의 하드웨어 쓰레드, IXP2400에서는 64개, IXP2800에서는 128개를 각각 제공한다. 각 하드웨어 쓰레드에 프로토콜 탐지 엔진을 할당할 때, 전체 네트워크 트래픽에서 각 프로토콜이 차지하는 비율(네트워크 트래픽 비율)이나 전체 시그너처 개수에서 각 프로토콜에 대한 시그너처 개수가 차지하는 비율(프로토콜 시그너처 비율)에 따라 각 탐지 엔진을 분배하는 방법, 그리고 네트워크 트래픽 비율과 프로토콜 시그너처 비율에 가중치(weight)를 주어 할당하는 방법이 가능하다. 본 장에서는 분산형 침입 탐지 엔진 시스템에서 네트워크 프로세서에 주어진 하드웨어 쓰레드를 최적으로 사용하기 위한 방안들을 살펴본다.

### 5.1 네트워크 프로토콜 트래픽 비율에 따른 할당 방법

네트워크 프로토콜 트래픽 비율에 따른 할당 방식은 전체 네트워크 트래픽 중 각 프로토콜의 트래픽이 차지하는 비율에 따라 탐지 엔진을 할당하는 방식이다. 전체 네트워크 트래픽에서 TCP, IP, UDP, ICMP 프로토콜의 트래픽이 차지하는 정도에는 차이가 있다. 표 2에 실제 데이타 트래픽을 분석한 결과를 보였다. MIT의 Lincoln Lab에서 침입 탐지 시스템을 시험하기 위해 실제 트래픽을 tcpdump한 데이타를 제공하는데, 표 2는 데이타 집합 2개의 트래픽을 분석한 결과이다[27].

표 2에서 네트워크 상의 패킷은 프로토콜별로 트래픽량에 많은 차이가 있음을 알 수 있다. 가장 많은 비율을 차지하는 프로토콜(TCP)과 가장 적은 비율을 차지하는 프로토콜(ICMP)과의 차이가 1000배 이상 존재한다. 따라서 마이크로 엔진의 각 하드웨어 쓰레드에 프로토콜별 전용 탐지 엔진을 할당할 때, 전체 네트워크의 프로토콜별 트래픽의 비율을 고려하여 할당함으로써, 네트워크 프로세서의 이용도(utilization)를 높일 수 있다. 프로

표 2 네트워크 프로토콜 트래픽 비율(IP는 TCP, UDP, ICMP가 아닌 나머지 트래픽)

| | TCP | IP | UDP | ICMP | SUM |
|---|---|---|---|---|---|
| The number of Packets | 1,369,134 | 22,789 | 107,257 | 1,071 | 1,500,251 |
| Ratio(%) | 91.26 | 1.52 | 7.15 | 0.07 | 100 |
| The number of Packets | 1,247,366 | 62,702 | 59,679 | 1,036 | 1,370,783 |
| Ratio(%) | 91.00 | 4.57 | 4.35 | 0.08 | 0 |

토콜 비율에 기반한 TCP 전용 침입 탐지 엔진 개수는 다음과 같이 구할 수 있다.

$RP_{TCP}$ : TCP Protocol Rate, $RP_{IP}$ : IP Protocol Rate
$RP_{UDP}$ : UDP Protocol Rate, $RP_{ICMP}$ : ICMP Protocol Rate
$RP_{IP} = 1 - (RP_{TCP} + RP_{UDP} + RP_{ICMP})$
$T_{TCP}$ : The number of TCP Detecion Engine Threads
$N$ : The number of MicroEngine threads (24 in IXP1200)
$T_{TCP} = N \bullet RP_{TCP}$

### 5.2 프로토콜 시그너처 비율에 따른 할당 방법

프로토콜 시그너처 비율에 기반한 할당 방법은 전체 시그너처 중 각 프로토콜이 차지하는 비율을 고려하여 탐지 엔진 쓰레드를 분배하는 방법이다. 표 3, 4는 Snort 버전1.8.6과 Snort 버전 2.0.0에서 기본 시그너처 중 각 프로토콜이 차지하는 비율을 나타낸 것이다. 버전 1.8.6에는 UDP 시그너처와 ICMP 시그너처는 전체 룰의 10%정도로 비슷한 비율을 차지하고, TCP 시그너처는 전체 룰의 76.72%, IP 시그너처는 2.45%로, 가장 높은 비율을 차지하는 TCP 시그너처는 가장 낮은 비율을 차지하는 IP 시그너처의 31배가 넘는다. 또한 버전 2.0.0에서는 TCP 시그너처가 IP 시그너처의 40배가 넘는다. 마이크로엔진의 각 쓰레드에 프로토콜 탐지 엔진을 할당할 때, 이와 같은 프로토콜 시그너처 비율에 따라, 프로토콜별 전용 침입 탐지 엔진 쓰레드의 수를 조절하고 할당함으로써, 더욱 효율적으로 네트워크 프로세서를 활용할 수 있다. TCP 패킷을 위한 탐지 엔진의 개수는 아래와 같이 나타낼 수 있다.

$RR_{TCP}$ : TCP Rule Signature Rate, $RR_{IP}$ : IP Rule Signature Rate
$RR_{UDP}$ : UDP Rule Signature Rate, $RR_{ICMP}$ : ICMP Rule Signature Rate
$RR_{IP} = 1 - (RR_{TCP} + RR_{UDP} + RR_{ICMP})$
$N$ : The number of MicroEngine threads (24 in IXP1200, 64 in IXP2400)
$T_{TCP}$ : The number of TCP Detecion Engine Threads
$T_{TCP} = N \bullet RR_{TCP}$

### 5.3 네트워크 프로토콜 트래픽과 시그너처 가중 비율에 의한 할당 방법

가중 비율에 따른 할당 방식은 상기 두 가지 방법에 가중치를 부여하여 하드웨어 쓰레드를 분배하는 방식이다. 네트워크 프로토콜 트래픽 비율과 프로토콜 시그너처 비율의 가중치를 조절하여 보다 최적화된 침입 탐지 엔진 시스템의 구현이 가능하다. TCP 패킷을 위한 탐지엔진의 개수는 아래와 같이 구할 수 있다. 본 논문에서는 직접 침입 탐지 시스템을 구현한 후, 성능을 측정하여, 최적의 $W_P$와 $W_R$을 구한다.

$W_P$ : Weight for Protocol Traffic Ratio
$W_R$ : Weight for Rule Signature Ratio
$T_{TCP} : N \bullet [(W_P \bullet RP_{TCP}) + (W_R \bullet RR_{TCP})]$

## 6. 성능 평가

앞에서 설계한 네트워크 프로세서 기반 침입 탐지 시스템을 구현하기 위해서 그림 12와 같은 Linux를 기반으로 한 환경을 갖추고 있으며, RadiSys에서 ENP-2506P 보드를 위해 ENP-SDK 2.01을 사용하였다[28]. 테스트 데이타는 MIT의 Lincoln Lab에서 제공하는 침입 탐지 시스템을 시험하기 위해 실제 트래픽을 tcpdump한 데이타를 사용하였다. 구현된 탐지 엔진을 테스트하기 위해서, 스크립트 기반 네트워크 패킷 엔진으로 물리적 계층에서 어플리케이션 계층까지 다양한 계층의 패킷을 발생 시킬 수 있는 Packet Excalibur라는 프로그램을 사용하였다[29].

### 6.1 NP-NIDE의 성능 측정

NP-NIDE의 코드 수행 시간을 측정하기 위하여 IXP1200의 FBI CSR에 있는 Cycle Count 레지스터를 이용하였다. 이 Cycle Count 레지스터는 64비트 카운터

표 3 프로토콜 시그너처 비율(Snort version 1.8.6)

| | TCP | IP | UDP | ICMP | SUM |
|---|---|---|---|---|---|
| The number of Rules | 972 | 31 | 136 | 128 | 1267 |
| Ratio(%) | 76.72% | 2.45% | 10.73% | 10.10% | 100% |

표 4 프로토콜 시그너처 비율(Snort version 2.0.0)

| | TCP | IP | UDP | ICMP | SUM |
|---|---|---|---|---|---|
| The number of Rules | 1514 | 37 | 192 | 133 | 1876 |
| Ratio(%) | 80.70% | 1.97% | 10.23% | 7.09% | 100% |

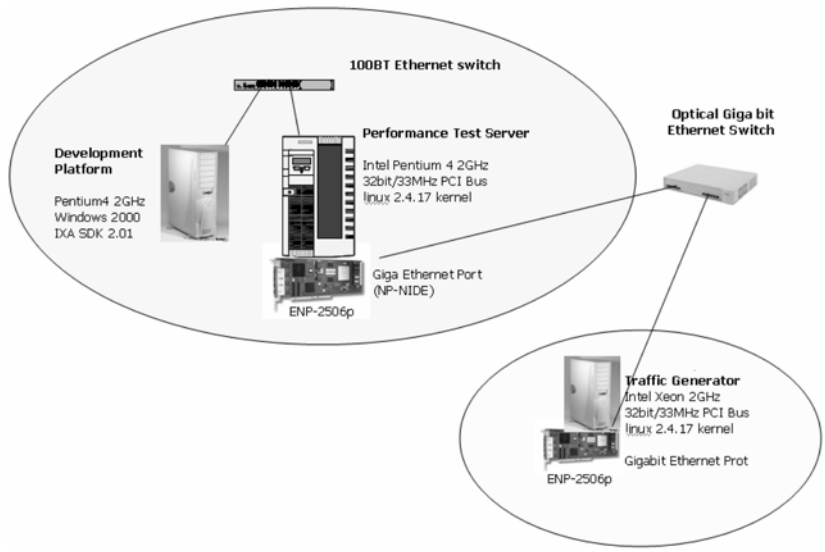


그림 12 성능 측정 환경

이고, Core Clock의 Frequency(232MHz, 4.3ns/clock (cycle))로 동작한다. 측정하고자 하는 마이크로엔진 코드의 전과 후에서 FBI CSR의 Cycle Count 레지스터의 값을 읽고, 두 값의 차를 구하는 방법으로 시간을 측정하였다.

6.1.1 숫자와 스트링 매칭 시간 측정

시그너처는 크게 스트링 비교를 포함하고 있는 Content 룰 과 String 비교를 포함하고 있지 않고, 숫자 비교만을 포함하고 있는 non-content 룰로 나눌 수 있다. Content 룰은 시그너처에 'content'라는 옵션 키워드가 있어서 패킷의 페이로드(Payload)에서 특별한 스트링 패턴을 찾아야 한다. Non-content 룰은 이러한 스트링 패턴 매칭 없이 숫자 필드를 비교만을 한다. 예를 들어, 패킷의 헤더의 필드(IP's TTL Field, Flag Field) 값을 비교하는 것은 모두 숫자 비교이다.

IXP1200의 마이크로엔진에 구현한 탐지 엔진의 성능을 측정하기 위해서, 먼저 숫자 필드를 체크하는 시간과 스트링 패턴 매칭에 필요한 시간을 측정하였다. 숫자를 체크하는 코드와 스트링 매칭을 수행하는 마이크로엔진 코드의 전과 후에서 FBI CSR의 Cycle Count 레지스터의 값을 읽고, 두 값의 차를 측정하였다.

그림 13은 숫자 필드를 비교하는 마이크로엔진 코드

| Microcode | Description |
|---|---|
| 1. .local startTime endTime runTime | |
| 2. xbuf_alloc($cycle_xfer, 2) | allocate SRAM Transfer Register |
| 3. csr[read, $cycle_xfer[0], CYCLE_CNT], ctx_swap | read from Cycle Count Register to $cycle_xfer |
| 4. alu[startTime, --, B, $cycle_xfer[0]] xbuf_free[$cycle_xfer] | copy $cycle_xfer to startTime |
| 5. | |
| 6. .if( checkField & CHECK_ICMP_TYPE) | check checkField |
| 7. sdram[read, $$packet_data[0], data_ptr, 4, 2], sig_done | read packet from SDRAM |
| 8. ctx_arb[sdram] | |
| 9. xbuf_extract(packet_byte, $$packet_data, 0, 2, 1) | extract icmpType(1byte) field from packet_data |
| 10. | |
| 11. sram[read, $rule_data[0], rules_addr, 4, 2], sig_done | read rule from SRAM |
| 12. ctx_arb[sram] | |
| 13. xbuf_extract(rule_byte, $rule_data, 0, 4, 1) | extract icmpType field from rule_data |
| 14. .if( packet_byte != rule_byte) | compare icmpType field on packet with |
| 15. br[get_next_rule#] | icmpType value on rules |
| 16. .endif | |
| 17. | |
| 18. xbuf_alloc($cycle_xfer1, 2) | read from CSR to $cycle_xfer1 |
| 19. csr[read, $cycle_xfer1[0], CYCLE_CNT], ctx_swap | |
| 20. alu[endTime, --, B, $cycle_xfer1[0]] | copy $cycle_xfer1 to endTime |
| 21. alu[runTime, endTime, -, startTime] | subtract startTime from endTime |
| 22. xbuf_free[$cycle_xfer].endlocal | (runTime=endTime-startTime) |

그림 13 숫자 필드 비교 측정 예제 코드

| Briefly described IXP1200 Microcode (1) | Briefly described IXP1200 Microcode (2) |
|---|---|
| **1. /*Start Time Read*/**<br>2. allocate SRAM Transfer Register<br>3. read from CSR(Cycle Count Register) to $cycle_xfer<br>4. copy $cycle_xfer to startTime<br>5.<br>**6. /*String Check Code */**<br>7. .if( checkField & CHECK_ICMP_TYPE)<br>8.<br>**9. get_one_byte#:**<br>10. extract each 1byte(rule_byte,packet_byte) from packet_data and rule_data<br>11. .if( rule_byte == packet_byte)<br>12. br[byte_match#]<br>13. .else<br>14. br[byte_not_match#]<br>15. .endif<br>**16. byte_match#:**<br>17. .if(curr_rule_len >= max_rule_len)<br>18. set match_result to 1<br>19. br[cmp_end#]<br>20. .endif<br>21. .if(packet_len >= max_packet_len)<br>22. set match_result to 0<br>23. br[cmp_end#]<br>24. .endif<br>25. set rule_read_addr and packet_read_addr<br>26. br[get_one_byte#] | **27. byte_not_match#:**<br>. increase packet_NB_offset and curr_packet_base_len<br>29. .if( curr_packet_len >= max_packet_len)<br>30. br[get_next_rule#]<br>31. .endif<br>32. br[NB_read_sdram#]<br>**33. cmp_end#:**<br>34. .if(match_result==0)<br>35. br[mepass#]<br>36. .else<br>37. br[exception#]<br>38. .endif<br>**39. exception#:**<br>40. alert to StrongArm<br>41. br[end#]<br>**42. mepass#:**<br>43. pass the packet to Egress Block<br>44. end#:<br>45. go next to rule check<br>46. .endif<br>47.<br>**48. /* End Time Read*/**<br>49. read from CSR to $cycle_xfer1<br>50. copy $cycle_xfer1 to endTime<br>51. subtract startTime from endTime (runTime=endTime-startTime) |

그림 14 스트링 필드 비교 측정 예제 코드

의 사이클 수를 측정하기 위한 예제 코드를 나타낸 것이다. 좌측은 시그너처 중 숫자 필드를 비교하는 마이크로엔진 코드이고, 오른쪽은 코드에 대한 설명이다. 실측에 의하면 시그너처 중 숫자 필드 하나를 비교하는데 평균 108 clock이 소요되었다. 스트링 비교 시간도 동일한 방법으로 측정하였다. 스트링 매칭에 걸리는 시간을 측정하는 마이크로엔진 코드는 길이가 길어서, 그림 14에 마이크로엔진 코드를 간단히 pseudo 코드로 기술하였다. 스트링 비교는 단순한 기본 스트링 비교 알고리즘을 사용하였다. 패킷의 전체 길이가 74 바이트 (페이로드 길이 32 바이트)인 패킷에서 14 바이트 길이의 패턴을 찾는 경우를 실측한 결과, 평균 5,416 clock이 소요되었다.

6.1.2 ICMP 분산형 침입 탐지 엔진 성능

TCP, IP, UCP, ICMP의 4가지 침입 탐지 엔진 중 하나인 ICMP 엔진을 구현하여, 침입 탐지에 걸리는 시간과 예상 처리 속도를 구하였다. 구현된 ICMP 탐지 엔진은 DIDE방식으로 구현했으며, Snort 2.0에서 default로 제공하는 ICMP 룰 133개를 지원한다. 133개의 ICMP에 대한 시그너처는 non-content 룰 95개와 content를 포함한 룰 38개로 이루어져 있다. 그림 15, 16, 17은 ICMP 탐지 엔진의 수행시간을 나타낸다. 앞 절에서 사용한 Cycle Count 레지스터를 읽어서 수행 clock을 측정하는 방법으로, 74바이트의 ICMP 패킷이 처리될 때의 시간을 측정하였다. 일반적으로 정상적인 Network에 침입이 거의 발생하지 않는다고 보고, 정상 패킷에 대해서 성능 측정을 하였다. 정상 패킷을 보내어

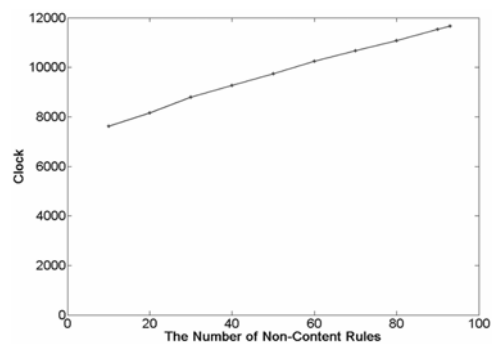


그림 15 Non-Content 룰 탐지 시간

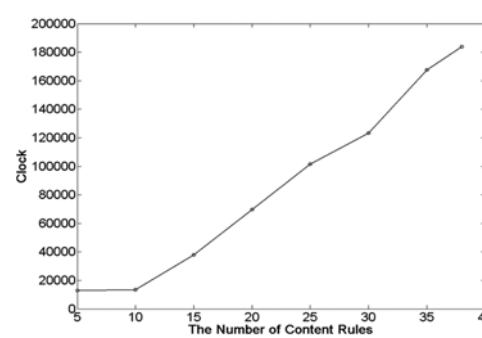


그림 16 Content 룰 탐지 시간

측정하였기 때문에 탐지되는 시그너처가 없으므로, 모든 시그너처에 대해 모든 패킷이 일대일 비교가 일어난다.

따라서 성능 측정값은 Worst Case의 성능이다.

그림 15는 최소 10개에서 최대 93개의 Non-Content 룰에 대해서 10개 단위로 룰 수를 변화시켜가면서 측정한 결과이다. 초기에 시간이 많이 걸리는 이유는 SDRAM에 저장된 수신 패킷을 읽어오고, SRAM에서는 비교할 시그너처를 읽어와서 SDRAM Transfer 레지스터와 SRAM Transfer 레지스터에 저장하는데 시간이 걸리기 때문이다. 룰 10개가 증가될 때, 평균 500 clock 정도의 증가를 보였으며, 모든 Non-Content 룰 93개를 검색하는데 평균 11,665 clock이 소요되었으며, 한 개의 Non-Content 룰을 측정하는데는 평균 125 clock이 소요된다. 74 바이트의 ICMP 패킷을 처리하는데, 11,665 clock이 소요되므로, 엔진 하나당 최대 12Mbps의 속도로 탐지를 할 수 있다.

그림 16은 최소 5개 최대 38개의 Content 룰에 대해서 5개 단위로 룰 수를 변화시켜가면서 측정한 결과이다. Content 룰 38개 모두를 검색하는데는 183,754 clock이 소요되었으며, 하나의 Content 룰을 측정하는데 평균 4,835 clock이 걸렸다. 이 값이 5.1에서 측정한 스트링 매칭하는데 걸리는 5,416 clock보다 적은 이유는 앞의 테스트에서는 패킷에서 14 바이트 길이의 패턴을 찾는 것을 실험하였는데, 실제 ICMP 룰에는 "00", "1234"와 같이 14바이트 보다 짧은 패턴을 찾는 룰이 많고, 또한 실제 비교에서는 한 룰에서 숫자 필드를 먼저 비교하여 모두 만족할 경우만 스트링 매칭이 실행되기 때문이다.

그림 17은 Content 룰과 Non-Content 룰을 모두 합한 ICMP Rules 133개에 대해서 수행시간을 측정한 결과이다. 한 ICMP룰을 측정하는데 평균 1436 clock이 걸렸다. 그림 17에서 룰이 0-30개 사이와 100-130개 사이에 비해 30-100 사이의 증가도가 낮은 것은 30-100 사이의 룰이 대부분 측정시간이 짧은 Non-Content 룰이기 때문이다. Content와 Non-Content 룰이 섞여있는 전체 시그너처 중에서 상대적으로 시간이 많이 걸리는 Content 룰의 위치에 따라 탐지 엔진의 성능이 차이가 날 수 있다. 향후 필요시 최적의 시그너처 배치도 성능 향상을 위해서 고려해 볼 수 있다.

표 5는 그림 15, 16, 17의 측정 결과를 토대로 IXP1200에 하드웨어 쓰레드 24개를 사용했을 때의 예상 성능을 보여 준다. 24개의 DE(Detection Engine)을 사용했을 경우, 133개의 룰이 있을 때, 패킷을 17.52Mbps 정도의 속도로 처리할 수 있다.

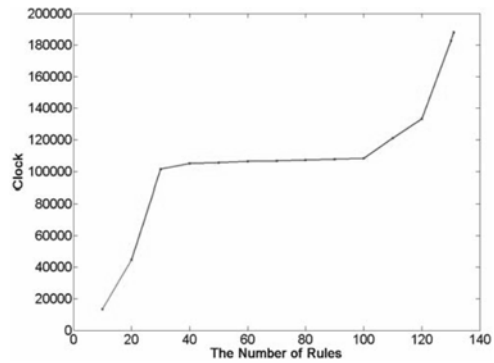


그림 17 ICMP 룰 탐지 시간

6.1.3 TCP/IP/UDP 분산형 침입 탐지 엔진 성능

본 논문에서는 앞에서 IXP1200의 FBI CSR에 있는 Cycle Count 레지스터를 사용해 측정한 결과를 이용해서, Snort 2.0의 TCP, UDP, IP 룰에 대한 탐지 시간을 시뮬레이션하였다. Snort 2.0에서 TCP, UDP, IP에 대하여 각각 기본적으로 제공하는 1,514개, 192개, 37개의 시그너처에 기반하여 content의 옵션에 대하여, 그리고 패킷의 전체 길이가 74 바이트(패이로드 길이 32바이트)인 패킷에 대하여 소모되는 탐지 시간을 측정하였다. 총 1,514개의 TCP 룰에 대하여는 1,471개가 content 옵션을 포함하고 있었고, 평균 하나의 content 옵션을 체크하는데 2,116 클럭이 소요되었다. UDP 룰 192개 중 178개가 content 옵션을 포함하고 있었고, 4,477 클럭이 평균 하나의 content 옵션을 체크하는데 소모되었다. 또한 IP 룰 37개 중 36개가 content 옵션을 포함하고 있었고, 하나의 content 옵션 체크에 평균 5,984 클럭이 소모되었다. 이러한 차이는 각 TCP, UDP, IP 룰에서 content 옵션 체크에서 패턴의 길이에 의한 것으로 TCP 룰에는 특히 짧은 스트링을 찾는 룰이 많았다. 참고로, Snort 2.0에서는 하나의 룰에 여러 개의 옵션을 포함할 수 있으므로 content 옵션의 수가 content 옵션을 포함하는 룰의 수를 의미하지는 않는다.

숫자 필드를 탐지하는 작업은 패킷 헤더에서의 위치만 다를 뿐 모두 ADD연산자를 이용한 bit연산이므로, 4장에서 실험한 것과 같이 평균 108 clock정도 소요된다고 가정할 수 있다. Snort 2.0의 TCP, UPD, IP 각 룰을 분석하여 숫자 필드와 스트링 필드를 개수를 조사하고, TCP, UDP, IP 룰의 content 필드 탐지 시간과

표 5 IXP1200에서의 ICMP 탐지 엔진 성능

| | Clocks for a rule (232MHz) | Clocks for all rules (232MHz) | Mbps for all rules (at a DE) | Mbps for all rules (at 24 DE) |
|---|---|---|---|---|
| Total Rules | 1,436 | 188,141 | 0.73 | 17.52 |

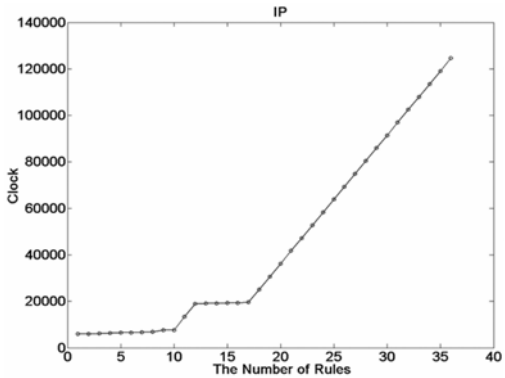


그림 18 IP 룰 탐지 시간

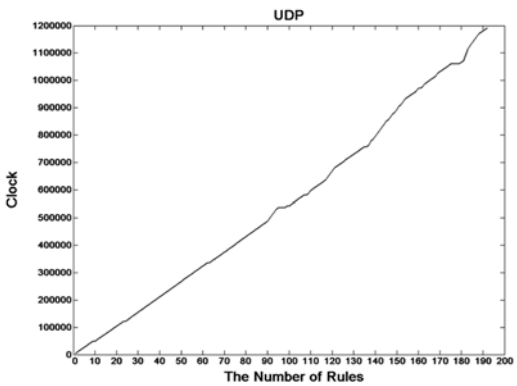


그림 19 UDP 룰 탐지 시간

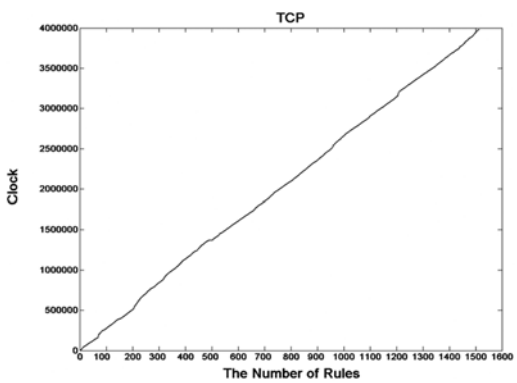


그림 20 TCP 룰 탐지 시간

4, 5장에서 실험한 데이타를 바탕으로 TCP, UDP, IP 룰을 탐지하는데 소요되는 시간을 시뮬레이션하였다.

이들 결과 중 그림 18에 Snort 2.0에서 기본적으로 제공하는 37개의 IP 룰에 대하여 탐지하는데 걸리는 시간을 나타내었다. IP 룰 36개 모두를 검색하는 데는 124,580 clock이 소요되었으며, 한 개의 IP 룰을 측정하는 데는 평균 3,460 clock이 걸렸다. 그림 19에는 Snort 2.0에서 기본적으로 제공하는 192개의 UDP 룰을 탐지하는데 걸리는 시간을 나타내었다. UDP 룰 192개 모두를 검색하는데는 1,192,013 clock이 소요되었으며, 한 UDP 룰을 측정하는데 평균 6,208 clock이 걸렸다. 그림 20에는 Snort 2.0의 1,514개의 TCP 룰을 탐지 하는데 걸리는 시간을 나타내었다. TCP 룰 1,514개 모두를 검색하는 데는 3,987,672 clock이 소요되었으며, 하나의 TCP 룰을 측정하는데 평균 2,634 clock이 걸렸다.

### 6.2 침입 탐지 엔진 할당 방법에 따른 성능 비교

아래에는 분산형 침입 탐지 엔진에서 하드웨어 쓰레드에 프로토콜 탐지 엔진을 할당하는 방법에 대한 성능 분석을 기술한다. 본 논문은 침입 탐지 엔진 할당 방법으로 '네트워크 트래픽 비율'이나 '프로토콜 시그너처 비율'에 따라 각 탐지 엔진을 분배하는 방법, 그리고 네트워크 트래픽 비율과 프로토콜 시그너처 비율에 가중치(weight)를 주어 할당하는 방법을 제시하였다. IP, TCP, UDP, ICMP를 담당하는 침입 탐지 엔진을 마이크로 엔진의 하드웨어 쓰레드에 균등하게 배분하였을 때와 침입 네트워크 트래픽 비율과 프로토콜 시그너처 비율을 고려하여 침입 탐지 엔진을 할당하였을 때의 성능을 분석하였다.

#### 6.2.1 균등 마이크로 엔진 할당

네트워크 프로세서 IXP1200는 24개의 하드에어 쓰레드를 가지고 있다. IXP1200에서 균등 마이크로 엔진 할당 기법을 쓸 경우, 마이크로 엔진 쓰레드 할당 방법을 표 6에 나타내었다. IXP1200의 마이크로 엔진의 속도를 고려했을 때, 포트 0과 포트 1의 수신, 송신을 위해 각각 2개씩의 하드웨어 쓰레드에 할당하고, 나머지 20개를 4개의 프로토콜을 위해 5개씩 균등 할당하였다.

표 7에 IXP1200에서의 마이크로 엔진에 균등 할당 기법을 사용하여 탐지 엔진을 할당할 경우의 성능을 나타내었다. 탐지 엔진의 속도는 TCP, UDP, IP, ICMP 중 가장 늦게 패킷 처리를 하는 TCP에 의해 속도가 결정된다고 볼 때, TCP 탐지 엔진이 1,514개의 모든 룰을 검사하는데, 3,987,672 clock이 소요되므로, 한 탐지 엔진은 0.03Mbps의 속도로 패킷을 처리할 수 있고, 이러한 엔진이 5개이므로 IXP1200에서 각 프로토콜에 균등 마이크로 엔진 쓰레드를 할당했을 때는 대략 190Kbps의 속도로 패킷을 처리할 수 있음을 알 수 있다. 이 시뮬레이션 결과는 모든 패킷과 전체 시그너처 데이타베이스가 전부 일대일 비교할 경우의 결과이므로, Worst 경우의 결과이다.

본 논문의 개발 플랫폼인 IXP1200은 네트워크 프로세서의 초기 버전으로, 명령어 처리속도(232MHz), 하드웨어 쓰레드 개수(24개), control store(엔진당 1K in-

표 6 균등 마이크로엔진 쓰레드 할당 (2×1G포트) (MB : Microblock)

| Thread | Microengine0 | Microengine1 | Microengine2 | Microengine3 | Microengine4 | Microengine5 |
|---|---|---|---|---|---|---|
| 0 | 145 | Input IF | UDP Detection MB | IP Detection MB | IP Detection MB | ICMP Detection MB |
| 1 | TCP Detection MB | TCP Detection MB | UDP Detection MB | IP Detection MB | ICMP Detection | ICMP Detection |
| 2 | TCP Detection MB | TCP Detection MB | UDP Detection MB | IP Detection MB | ICMP Detection | ICMP Detection |
| 3 | TCP Detection MB | UDP Detection MB | UDP Detection MB | IP Detection MB | Output IF | Output IF |

표 7 IXP1200에서의 균등 마이크로엔진 쓰레드 할당 기법에 따른 성능

| | Clocks for all Rules (232MHz) | Mbps for all Rules (at a DE) | Mbps for all Rules (at 5 DE) | Bandwidth (Mbps) |
|---|---|---|---|---|
| TCP | 3,987,672 | 0.03 | 0.17 | 0.19 |
| UDP | 1,192,013 | 0.12 | 0.58 | 8.06 |
| IP | 124,580 | 1.10 | 5.51 | 362.65 |
| ICMP | 188,141 | 0.73 | 3.65 | 5,214.33 |

struction)와 레지스터 개수(128 General Purpose 레지스터, 128 Transfer 레지스터)면에서 한계가 많았다. 그러나 IXP2400의 경우, 8개의 마이크로 엔진에 총 64개의 하드웨어 쓰레드를 지원하고 명령어 처리속도 600MHz을 제공한다. IXP2800는 16개의 마이크로 엔진에 총 128개의 하드웨어 쓰레드를 지원하며, 명령어 처리 속도 700MHz를 제공한다. 또한 control store면에서 IXP2400는 엔진당 4K instruction, IXP2800은 엔진당 8K instruction을 제공한다. 이러한 차이는 하드웨어의 양적 증가 외에도 IXP1200에서 control store의 제약 때문에 프로그램에 효율적인 알고리즘을 사용하는데 문제되었던 부분을 해결할 수 있다. 또한 레지스터의 개수면에서 IXP2400와 IXP2800의 경우 256개의 General Purpose 레지스터와 512개의 Transfer 레지스터를 제공하므로, IXP1200에서 SDRAM에 저장된 수신 패킷과 SRAM에서 비교할 룰 시그너처를 SDRAM Transfer 레지스터와 SRAM Transfer 레지스터로 읽어올 때, 한 번에 읽어올 수 있는 데이타량에 제한을 받았던 점을 극복할 수 있다. 따라서 IXP2400과 IXP2800에서는 향상된 명령어 처리속도, 하드웨어 쓰레드 수 증가, control store 향상, 레지스터 개수의 향상, 한 번에 읽어올 수 있는 데이타량의 향상 등으로 IXP1200에 비해 높은 성능이 기대할 수 있다.

6.2.2 네트워크 프로토콜 트래픽과 시그너처 가중 비율에 의한 할당

보다 효율적인 탐지 엔진을 설계하기 위하여, MIT의 Lincoln Lab에서 제공하는 침입 탐지 시스템을 시험하기 위한 실제 네트워크 트래픽을 tcpdump한 결과를 분석해서 얻어진 표 2의 결과와 Snort 2.0.0의 룰 비율을 분석하여 얻은 표 4의 결과를, 5장에서 제시했던 네트워크 프로토콜 트래픽과 시그너처 가중 비율에 의한 할당 방법을 나타내는 아래의 식에 대입하고, $W_P$와 $W_R$의 값에 변화를 주어, 탐지 엔진의 개수를 계산하여 표 8에 나타내었다.

$$W_P \quad : Weight\ for\ Protocol\ Traffic\ Ratio$$
$$W_R \quad : Weight\ for\ Rule\ Signature\ Ratio$$
$$T_{TCP} : N \bullet [(W_P \bullet RP_{TCP}) + (W_R \bullet RR_{TCP})]$$

위의 표에서 알 수 있듯이 IXP1200의 마이크로 엔진 쓰레드를 네트워크 트래픽과 시그너처 비율을 고려하고, $W_P$와 $W_R$의 비율을 0.50 : 0.50, 0.25 : 0.75, 0.75 : 0.25로 달리하여 계산해 본 결과, 각 비율별 탐지 엔진 수는 별로 차이가 나지 않는다. 그 첫 번째 이유는 IXP1200의 하드웨어 쓰레드 수가 적기 때문이다. IXP1200의 24개의 하드웨어 쓰레드 중, 송신, 수신을 제외한 기능을 담당하는 하드웨어 쓰레드를 제외하면 20개 밖에 남지 않는데, 20개의 탐지 엔진을 $W_P$와 $W_R$의 비율을 변경하여 계산해도 엔진 수에 별로 영향을 주지 못했다. IXP2400과 IXP2800와 같이 지원하는 하드웨어 쓰레드 수가 증가한다면, $W_P$와 $W_R$에 따른 엔진 개수의 변화가 좀 더 커질 것으로 보인다. $W_P$와 $W_R$의 비율에 따른 각 프로토콜별 마이크로 엔진의 수가 별로 변화가 없는 두 번째 이유는 네트워크 트래픽에서의 각 프로토콜에 해당하는 비율과 시그너처에서의 각 프로토콜이 차지하는 비율이 거의 비슷하기 때문이다. 예를 들면,

표 8 네트워크 트래픽과 시그너처 가중 비율에 따른 마이크로엔진 쓰레드 할당수 (IXP1200)

| $W_P$ : $W_R$ | TCP DE | IP DE | UDP DE | ICMP DE | Total DE |
|---|---|---|---|---|---|
| 0.50 : 0.50 | 17.196 | 0.349 | 1.738 | 0.716 | 20 |
| 0.25 : 0.75 | 17.724 | 0.3265 | 1.584 | 0.365 | 20 |
| 0.75 : 0.25 | 16.668 | 0.3715 | 1.892 | 1.067 | 20 |

표 2의 네트워크 프로토콜 트래픽 비율에서 TCP(80.70%), UDP(10.23%), IP(1.97%), ICMP (7.09%)의 비율로 TCP가 가장 많았고, 표 3의 시그너처 비율에서도 TCP(76.72%), UDP(10.73%), ICMP (10.10%), IP(2.45%)의 비율로 TCP가 가장 많았다. 즉, 우리가 실험한 네트워크 tcpdump 트래픽 데이타에서는 시그너처에서의 프로토콜별 시그너처 비율과 네트워크 프로토콜에서의 프로토콜별 비율이 비슷한 양상을 보이기 때문에 $W_P$와 $W_R$의 비율에 변화를 주어도 실제 할당할 프로토콜별 탐지 엔진 개수에는 별로 변화가 없었다. 그러나 네트워크 트래픽의 프로토콜 비율은 사용하는 네트워크 환경에 따라 다를 수 있다. 예를 들어 영화 스트리밍 서비스를 제공하는 회의의 네트워크 환경의 경우, 대부분의 네트워크 트래픽이 UDP 패킷일 것이다. 따라서 기본적으로 송·수신 라인 속도를 보장하도록 룰 시그너처 비율을 고려하여 탐지 엔진을 할당하고, 여기에 추가 성분으로 네트워크에 트래픽 비율을 고려하는 방법을 생각할 수 있다.

표 8에서 $W_P$와 $W_R$을 각각 0.5로 셋팅한 결과에 따라 IXP1200에 침입 탐지 엔진을 할당했을 경우, 성능을 측정하여 표 9에 나타내었다. 24개의 하드웨어 쓰레드를 제공하는IXP1200은 패킷의 송·수신을 위해 4개를 제외하고, TCP 탐지 엔진으로 16개, IP 탐지 엔진으로 1개, UDP 탐지 엔진으로 2개, ICMP 탐지 엔진으로 1개의 하드웨어 쓰레드가 할당되었다. 침입 탐지 엔진의 속도가 가장 느린 성능에 좌우된다고 볼 때, IXP1200을 이용한 침입 탐지 시스템은 TCP 엔진의 결과에 따라 0.6Mpbs정도를 처리할 수 있다. 이 결과는 앞에서 측정한 균등 마이크로 엔진 할당 방법의 성능과 비교할 때 IXP1200의 경우 3.2배의 높은 성능이다. 명령어 처리속도 향상, 쓰레드 수의 향상, 한 번에 읽어올 수 있는 데이타량의 향상 등으로 보다 뛰어난 성능을 가진 IXP2400과 IXP2800의 경우, 균등 마이크로 엔진 할당 방법의 성능과 네트워크 트래픽과 시그너처 가중 비율에 따른 할당 방법과의 성능차이는 더욱 클 것으로 예상된다.

## 7. 결론 및 향후 계획

본 논문에서는 기가/테라급 인터넷 라우터와 방화벽 시스템에 최근 많이 활용되고 있는 네트워크 프로세서를 기반으로 하는 실시간/고속 네트워크 침입 탐지 시스템을 설계하고, 구현하였다. 고속 패킷 처리에 뛰어난 성능을 가진 마이크로 엔진을 최대한 활용하여, 침입 탐지 시스템의 핵심 기술인 침입 탐지 엔진을 설계하고, 제한적인 자원을 가진 네트워크 프로세서에서 보다 효율적인 침입 탐지 엔진을 구현하기 위해서, 하드웨어 쓰레드 수, 시그너처 수, 네트워크 트래픽 등을 고려하여 효율적인 침입 탐지 엔진 할당 기법을 연구하였다. 또한 실제 인텔 네트워크 프로세서인 IXP1200을 사용하여 프로토타입 침입 탐진 엔진을 구현하였고, 침입탐지엔진 할당 기법에 따른 성능을 비교 분석하였다

본 논문의 네트워크 프로세서 기반 침입 탐지 시스템은 공개 소스 네트워크 침입 탐지 시스템인 Snort 구조를 모델로 하여, Snort에서 지원하는 룰 시그너처를 그대로 포팅없이 지원한다. 네트워크 프로세서 IXP1200을 이용한 침입 탐지 시스템은 프로그래밍 가능하고 네트워크 패킷 처리에 특화된 RISC 기반의 네트워크 프로세서를 사용함으로써, 소프트웨어 기반 침입 탐지 엔진과, 하드웨어 기반 침입 탐지 엔진의 장점인 유연성과 고성능을 모두 가질 수 있는 구조로 설계되었다. 그러나 성능 분석 결과에서 보듯이 IXP1200 처리 속도와 Control Store면에서 제약이 많았으므로, 성능은 별로 뛰어나지 못했다. 그러나 IXP2400과 IXP2800의 경우, 명령어 처리속도, 하드웨어 쓰레드 수 증가, control store 향상, 레지스터 개수의 향상, 한 번에 읽어올 수 있는 데이타량의 향상 등으로 IXP1200에 비해 높은 성능이 기대할 수 있다.

본 논문에서 제안한 네트워크 프로토콜 트래픽과 시그너처 가중 비율에 의한 할당기법과 균등 마이크로엔진 할당 기법의 성능을 비교한 결과, 네트워크 프로토콜 트래픽과 시그너처 가중 비율에 의한 할당기법의 성능이 균등 마이크로엔진 할당 기법의 성능보다 IXP1200

표 9 IXP1200에서의 네트워크 트래픽과 시그너처 가중 비율에 따른 할당 방법의 탐지 엔진 성능

| | Clocks for all Rules (232MHz) | Mbps for all Rules (at a DE) | Mbps for all Rules | Bandwidth (Mbps) |
|---|---|---|---|---|
| TCP | 3,987,672 | 0.03 | 0.55 (at 16 DE) | 0.60 |
| UDP | 1,192,013 | 0.12 | 0.12 (at 1 DE) | 1.61 |
| IP | 124,580 | 1.10 | 2.20 (at 2 DE) | 145.06 |
| ICMP | 188,141 | 0.73 | 0.73 (at 1 DE) | 1,042.87 |

의 경우 3.2배 높았다. 이러한 성능의 차이는 IXP2400, IXP2800의 경우 더욱 클 것으로 여겨진다.

앞으로 IXP2400, IXP2800에서 침입탐지시스템을 구현할 계획이며, 보다 고성능의 네트워크 프로세서 기반 침입 탐지 엔진 구현을 위해서 룰 최적화 기법을 이용하여, 룰 매칭 검사수를 최대한 줄일 수 있도록 룰 자료구조를 개선하고, 더불어 마이크로엔진 구조에서 효율적인 String Matching 방법을 연구할 것이다. 또한 실시간으로 룰을 바꾸고, 룰의 우선순위를 바꿀 수 있는 동적할당방법에 대해서 연구할 것이다.

## 참 고 문 헌


[1] N. Desai, "Increasing Performance in High Speed NIDS," A look at Snort's Internals, 2002.

[2] I. Charitakis, K. Anagnostakis, and E. Markatos "An Active Traffic Splitter Architecture for Intrusion Detection," Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp.238-241, Orlando Florida, October 2003.

[3] H. Debar, M. Dacier, and A. Wespi, "Towards a taxonomy of intrusion detection system," Computer Networks, Vol.31, No.8, pp.805-822, 1990.

[4] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," IEEE Symposium on Field-Programmable Custom Computing Machines(FCCM01), 2001.

[5] B. Mukherjee, L. T. Heberlein, and K. N. Levitt, "Network Intrusion Detection," IEEE Network, Volume 8, Issue 3, pp.26-41, 1994.

[6] Korea Information Security Agency, available at http://www.kisa.or.kr

[7] COAST(Computer Operations, Audit, and Security Technology), available at http://www.cerias.purdue.edu /coast/coast.html

[8] Snort 홈페이지, available at http://www.snort.org

[9] R. S. Boyer, and J. S. Moore, "A Fast String Searching Algorithm," Comm. ACM 20, 10, pp. 761-772, 1977.

[10] Intel corporation homepage, available at http://www.intel.com

[11] Agere systems, "PyaloadPlus Routing Switch Processor," 2002.

[12] Agere systems, "NP-Complete Fore agere System PayloadPlus Family of Network Processor," 2002.

[13] Motorola corporation, "C-port Documentation," 2002.

[14] IBM homepage, available at http://www.ibm.com

[15] Network Processing Forum homepage, available at http://www.npforum.org

[16] Y. Tang, L. Qian, B. Bou-Diab, A. Krishnamurthy, G. Damm, and Y. Wang, "High-Performance Implementation for Graph-Based Packet Classification Algorithm on Network Processor," *IEEE International Conference on Communications (ICC 2004)*, vol.2, pp.1268-1272, 2004.

[17] Y. Chen and S. Lee, "An Efficient Packet Classification Algorithm for Network Processors," IEEE International Conference on Communications (ICC2003), vol.3, pp.1596-1600, 2003.

[18] C. Sheng, Z. Xu, C. Yingxin and D. Wei, "Implementation of 10Gigabit Packet Switching Using IXP Network Processors," IEEE *International Conference on Communications Technology (ICCT2003)*, vol.1, pp.532-535, 2003.

[19] E. Grosse and L. Y. N., "Network Processors Applied to IPv4/IPv6 Transition," IEEE Network, vol.17, 2003.

[20] E. Yeh, H. Chao, V. Mannem, J. Gervais, and B. Booth, "Introduction to TCP/IP Offload Engine," April 2002.

[21] Teja Technologies, Inc, available at http://www.teja.com.

[22] X. Nie, U. Nordqvkt, L. Gazsi and D. Liu, "Network Processors for Access Network(NP4AN): Trends and Challenges," IEEE International Symposium on System-on-chip(SOC2004), 2004.

[23] Z. Tan, C. Lin, H. Yin and B. Li, "Optimization and benchmark of cryptographic algorithms on network processors," IEEE Micro vol.24, pp.55-69, 2004.

[24] Intel corporation, Intel Network Processors product information.

[25] H. Cho, D. Kim, J. Kim, Y. Doh and J. Jang, "Network Processor based High-speed Network Intrusion Detection System," LNCS 3090, pp. 973-982, 2004.

[26] M. Roesch, snort source, version 1.8.6, available at www.snort.org, 2002.

[27] MIT Lincoln Lab homepage, DARPA Intrusion Detection Evaluation, available at http://www.ll.mit.edu.

[28] RadiSys corporation, "ENP-2506 Hardware Reference," 2002.

[29] Jitsu, Packet Excalibur, version 1.0. GPL, 2002.

조 혜 영

2000년 2월 부산대학교 전자계산학과 학사. 2004년 2월 한국정보통신대학교 공학부 석사. 2004년 3월~현재 한국과학기술정보연구원 슈퍼컴퓨팅센터 연구원. 관심분야는 실시간 임베디드 시스템, 네트워크 프로세서, 클러스터 컴퓨팅, 병렬 파일 시스템

김 대 영

1990년 2월 부산대학교 전산통계학과 학사. 1992년 2월 부산대학교 전산통계학과 석사. 2001년 8월 University of Florida 컴퓨터공학 박사. 1992년 1월~2001년 7월 한국전자통신연구원 연구원 1999년 5월~1999년 8월 AlliedSignal Aerospace 연구소 방문연구원. 2001년 9월~2002년 1월 Arizona State University 컴퓨터공학과 연구 조교수. 2002년 2월~현재 한국정보통신대학교 조교수. 관심분야는 센서 네트워크, RFID, 실시간 임베디드 시스템