석 사 학 위 논 문

Master's Thesis

어플리케이션 시그네쳐 자동 생성에 관한 연구

Automatic Generation of Application Signatures

김 태 호(金 太 鎬 Kim, Tae Ho)

전자전산학과 전산학 전공

Department of Electrical Engineering and Computer Science
Division of Computer Science

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

2006

어플리케이션 시그네쳐 자동 생성에 관한 연구

Automatic Generation of Application Signatures

Automatic Generation of Application Signatures

Advisor: Professor Moon, Sue Bok

by

Kim, Tae Ho
Department of Electrical Engineering and Computer Science
Division of Computer Science
Korea Advanced Institute of Science and Technology

A thesis submitted to the faculty of the Korea Advanced Institute of Science and Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Electrical Engineering and Computer Science, Division of Computer Science.

Daejeon, Korea
2006. 1. 5.
Approved by

_____
Professor Moon, Sue Bok

Advisor

어플리케이션 시그네쳐
자동 생성에 관한 연구

김태호

위 논문은 한국과학기술원 석사학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2005년 12월 14일

심사위원장 문 수 복 (인)

심사위원 최 기 선 (인)

심사위원 황 규 영 (인)

MCS

20033167

김 태 호. Kim, Tae Ho. Automatic Generation of Application Signatures. 어플리케이션 시그네쳐 자동 생성에 관한 연구. Department of Electrical Engineering and Computer Science, Division of Computer Science. 2006. p67. Advisor Prof. Moon, Sue Bok

Abstract

Network administrators want to know the current state of the network which they manage. For this purpose, they classify the network traffic into the predefined application categories. The port-based classification method of application traffic can be applied to the server service based applications (e.g. HTTP, DNS, SMTP) which communicate through the static port number. As P2Ps, online games and streaming applications which use the dynamic port allocation are more spreading, traffic classification using only port-based method are getting more inaccurate. The signature-based method can be applied to these applications but about 10 applications' among several thousands applications are well-known. It is not efficient manually to generate signatures for many applications and to update signatures daily. This paper investigates the challenges in automatic application signature generation and describes the automatic application signature generator, which can construct the unknown signatures from network traffic trace. With traffic traces from DAGMON monitoring system in KAIST campus, we can automatically construct known signatures (e.g. BitTorrent, edonkey, gnutella, HTTP) and unknown signatures (e.g. MSN messenger, nateon messenger). Evaluating using network traces shows that the signatures generated from our approach have low false positive (below 4%) and false negative (below 0.8%) in network traffic classification.

Contents

List of Figures

List of Tables

Chapter 1. Introduction

네트워크 관리자들은 그들이 관리하는 네트워크의 현황을 파악하고자 하며, 이를 위해 어떠한 유저가 많은 대역폭을 사용하는지, 어떠한 어플리케이션이 주로 사용되는지, 어떠한 사용자가 비정상적인 트래픽을 유발하는지를 지속적으로 모니터링 한다. 이러한 현황 파악의 한 방법으로 네트워크 관리자들은 사전에 정의된 어플리케이션 카테고리로 네트워크 트래픽을 분류하려고 한다. 이러한 분류를 위해 사용하는 네트워크 트래픽 분류법에는 크게 포트 기반 분류방식과 시그네쳐 기반 분류방식이 있다. 포트 기반 방식은 네트워크 트래픽을 Internet Assigned Numbers Authority(IANA)에 잘 알려진 포트(well-known) 리스트[1]를 참조하여 분류하는 방식이다. DNS, SMTP, FTP, TELNET등과 같은 서버 서비스 기반의 전통적인 어플리케이션은 정적 포트 할당(static port allocation)에 의해 통신을 하기 때문에, 포트 기반 분류방식이 이러한 전통 어플리케이션 트래픽 분류에 적용될 수 있다. 하지만 클라이언트, 서버 역활을 둘 다 할 수 있는 있거나 동적 포트 할당을 사용하거나 전통적인 포트를 경유하여 통신하는 P2P, 게임, 스트리밍 어플리케이션의 사용이 늘어나면 늘어날 수록 포트 기반 분류 방식은 한계성을 가지면서 부정확한 분류를 하게 된다[2]. 이러한 어플리케이션에 대해서는 시그네쳐 기반 분류 방식이 정확성을 기할 수 있다.

비록 시그네쳐 기반 분류 방식이 이러한 어플리케이션에 적용 가능하다고 할지라도 시그네쳐를 생성하는데 있어서 몇 가지 넘어야 할 과제들이 있다. 우선, 기존의 알려진 (well-known) 시그네쳐들이 대부분 어플리케이션의 프로토콜 정의 문서에 기반하여 수작업으로 만들어 졌다는 것이다[2]. 모든

어플리케이션들이 자신의 프로토콜 정의 문서를 외부에 공개하는 것은 아니기 때문에 많은 수의 어플리케이션은 프로토콜 문서에 기반해서 시그네쳐를 만들어 낼 수 없다. 모두 공개되어있다고 하더라도 모든 어플리케이션 시그네쳐를 수작업으로 생성한다는 것은 비효율적인 작업임에 틀림없다. 또한, 시그네쳐가 생성되었다고 하더라도, 그 시그네쳐들은 몇 개월이나 몇 달안에 소프트웨어 업그레이드나 패치, 보안상의 이유로 바뀌어 져서 기존의 시그네쳐가 유효하지 않게 될 수도 있다. 업그레이드나 패치에 의해 프로토콜 정의에서 변화가 생겨서 어플리케이션 시그네쳐에 영향을 주기도 하고, 게임과 같이 사용자의 치팅에 민감한 어플리케이션의 경우 주기적으로 업데이트를 수행하여 시그네쳐의 변화를 줄 수도 있다. 이러한 기존의 한계성들로 인해서, 본 논문에서는 어플리케이션 시그네쳐의 자동 생성 방법을 제안하고자 한다.

본 논문의 성과물을 통해서, 알려지지 않은 시그네쳐의 생성은 분류할 수 없었던 트래픽 분류에 기여할 뿐 아니라 어플리케이션 서비스마다 특화된 서비스를 제공하는데도 사용될 수 있다. 일례로 서비스 품질에서는 하나의 어플리케이션은 블록되고, 다른 하나는 더 높은 대역폭으로 전송되고 나머지 어플리케이션들은 그냥 보통의 방법으로 전송되게 해 줄 수 있다. 다른 서비스 품질 제공뿐 아니라 각 어플리케이션 별로 다른 사용 요금 체계를 적용할 수도 있다. 물론 이러한 종류의 시그네쳐를 기반한 서비스들은 고성능 네트워크 트래픽 내에서 플로우의 페이로드안에 시그네쳐를 찾아서 필터링 해주는 Cisco의 NBAR[5]와 같은 하드웨어 장비가 지원되는 경우에 유효하다.

본 논문은 자동 어플리케이션 시그네쳐 생성에 있어서의 도전들에 대해서 조사하고, 네트워크 트래픽 데이터를 기반으로 알려지지 않은 시그네쳐를 생성할 수 있는 자동 어플리케이션 시그네쳐 생성기의 프로토 타입에

대해서 기술하고 있다. 이 논문의 접근방식은 기본적으로 두 가지 가정에 기반을 두고 있다. 첫째로 네트워크 트래픽에서 자주 반복되는 패턴은 어떠한 어플리케이션의 시그네쳐이다. 만약 어떠한 어플리케이션이 프로토콜 정의상 반복되는 패턴을 가지고 있고 그것이 그 어플리케이션의 시그네쳐가 될 수 있다. 이 어플리케이션이 네트워크를 통해서 패킷들을 주고 받으면 그 패킷들 안에는 해당 시그네쳐가 자주 반복되게 된다. 두 번째로 시그네쳐는 TCP 핸드 쉐이킹(handshaking)뒤에 첫 번째 패킷에 있을 확률이 다른 패킷에 있을 경우보다 높다[2]. 기존의 연구 [2]에서는 P2P트래픽을 TCP 핸드쉐이킹 후 첫 번째 패킷만을 검사 하였을 때, 82%~100%, 두 번째 패킷까지 검사하였을 때는 93%~100%, 세 번째 패킷까지 검사하였을 때는 100%를 분류해 낼 수 있었다. 즉 P2P를 구분해 낼 수 있는 시그네쳐가 TCP 핸드쉐이킹 뒤 3번째 패킷 안에 존재한다는 것이다. 본 연구에서는 문제의 단순화를 위해서 우선 첫 번째 패킷만을 기반으로 시그네쳐 생성을 시도해 보았다.

본 논문의 프로토 타입에서는 우리는 오프라인 네트워크 트래픽을 기반으로 자동 어플리케이션 시그네쳐 생성 알고리즘을 개발하였다. 카이스트 캠퍼스 데그몬(DAGMON)[4] 모니터링 시스템으로부터 수집한 오프라인 트래픽 데이터를 기반으로 기존의 알려진 (well-known) 시그네쳐들(예: bittorrent, edonkey, gnutella, HTTP)과 알려지지 않은 시그네쳐들(예: MSN messenger, nateon messenger)을 생성해 내었다. 네트워크 트래픽 데이터를 기초로 접근 방식을 평가해 보았으며, 자동 시그네쳐 생성 알고리즘에 의해 생성된 시그네쳐의 정확도를 측정해 보았다. 그 결과 4%이하의 false positive와 0.8%이하의 false negative의 높은 정확도를 보여 주었다.

본 논문의 나머지 부분은 다음과 같이 진행된다. 2장에서 어플리케이션의

기초적인 내용과 본 논문과 관련이 있는 기존 연구들을 되짚어 보도록 하겠다. 3장에서는 본 연구에서 해결하고자 하는 문제가 무엇인지 기술하고 이에 대한 접근 방식을 기술한다. 4장에서는 본 연구의 다자인 이슈와 구현상에 문제점들을 살펴보도록 하겠다. 5장에서는 본 논문에서 제안한 방식에 의해 생성된 시그네쳐와 기존의 시그네쳐를 비교해 보고 그 정확도와 여러 결과들을 고찰해 보도록 하겠다. 마지막으로 6장에서는 결론을 맺도록 하겠다.

.

Chapter 2. Related Work

포트 기반 분류 방식은 최신의 네트워크 어플리케이션 분류에 사용되기에는 정확도가 떨어지는 방식이다. 많은 인터넷 어플리케이션들이 정적 포트 할당에 의해서 서비스가 제공이 막히거나 (blocking) 제한되는 것을 벗어나기 위해서 동적 포트 할당 방식을 사용해 나가고 있다. 기존의 많은 연구에서 내용 기반(content-based) 분류 방식에 대해서 연구하고 있고, 어플리케이션 시그네쳐를 생성하는 방식에 대해서도 많은 연구가 진행되어 오고 있다.

Moore and Papagiannaki [5]에서는 잘 알려진 포트 번호를 기반으로 포토 기반 분류의 부정확성과 시그네쳐 분류와의 비교를 보여주고 있다. 더 정확한 분류를 위해서 이 논문에서는 어떠한 플로우가 이미 분류된 플로우 결과인지 테스트하고 만약 플로우가 제대로 분류가 안되면 몇 가지 추가적인 분류 기준을 적용한다. 그들은 플로우가 잘 알려진 포트를 사용하거나 잘 알려진 시그네쳐가 플로우 전체나 플로우의 처음 몇 Kbyte 혹은 첫 번째 패킷에 존재하는지를 검사한다. 그 플로우가 어떠한 잘 알려진 시그네쳐나 시그네쳐 프로토콜을 가지고 있지 않다면, 수작업으로 분류한다. 이 논문에서는 어플리케이션 시그네쳐 생성을 labor-intensive 작업으로 해결하였다.

Sen, Spatcheck and Wang [6]에서는 P2P트래픽 분류에 포트 기반 방식보다 내용기반(content-based) 방식을 사용하였다. KAZZA같은 P2P어플리케이션의 경우 포트 기반으로 분류할 때 보다 시그네쳐를 기반으로 분류할 때 분류된 트래픽 양이 3배가 증가됨을 볼 수 있었다. P2P 어플리케이션들은 방화벽을 통과하기 위해 대부분 잘 알려진 포트를 사용하지 않고 있었다.

이 논문에서는 시그네쳐를 자동 생성하는 방안에 대해서는 따로 거론하지 않고 있다. 시그네쳐를 수작업으로 어플리케이션 정의 문서나 어플리케이션의 순수 패킷 수집 후 분석으로 얻어 내었다.

Kim and Karp [8]에서는 웜의 사전 정보 없이 페이로드로부터 웜의 시그네쳐를 자동 생성하는 방식을 제안하였다. 그들은 의심스러운 트래픽 데이터를 수집하여 작은 내용 블록(content block) 나누었다. 가장 자주 발생된 블록이 그 플로우의 시그네쳐가 되게 하였다. 그들은 웜의 시그네쳐 생성을 위해서 고정된 사이즈의 블록을 사용하였다.

Haffner, Sen, Spatcheck and Wang [9] 머신 러닝 알고리즘을 사용하여 어플리케이션시그네쳐 자동 생성을 구현하였다. 그들은 페이로드를 64바이트 나 256바이트 블록으로 나누고 각 블록들의 빈도 정도를 조사하였다. 트레이닝 기간 동안 각 블록들은 확률값을 유지하는데, 새로운 플로우가 들어올 경우 시스템은 지정된 사이즈 블록으로 페이로드를 나누고 어떠한 어플리케이션이 가장 높은 빈도를 가지는지 조사한다. 시스템은 그 플로우를 가장 높은 빈도 정도를 가지는 어플리케이션에 매핑한다. 이 접근 방식은 어플리케이션 시그네쳐를 수동적으로 생성하지 않고 자동으로 생성하는 최초의 연구이다. 하지만 이 접근 방식은 어플리케이션 마다 충분한 트레이닝 데이터가 필요하고 다양하게 어플리케이션 트래픽이 존재하는 데이터로는 각 어플리케이션의 시그네쳐를 생성해 낼 수 없다.

많은 연구들이 어플리케이션 시그네쳐를 어플리케이션의 정의 문서나 패킷 데이터로부터 수동적으로 생성해 내었다. 그 작업은 labor-intensive하고 어플리케이션 시그네쳐가 변화되었을 때 바로 시그네쳐의 변화를 확인하여 시그네쳐를 업데이트 해 줄 수 없었다. 또한 어떠한 연구에서는 정규화 된

시그네쳐에 대한 문제를 고려하지 않고 있었다

Chapter 3. Problem Statement

(1) 가정들(Assumption)

본 논문은 다음의 두 가지 가정에 기초하고 있다.

1) 어플리케이션이 보내고 받는 패킷들의 페이로드에는 공통적으로 존재하는 byte sequence가 있다.

어떠한 어플리케이션이 네트워크를 통해서 통신할 때, 인터넷 레이어 프로토콜, transport 레이어 프로토콜, 어플리케이션 레이어 프로토콜등과 같은 사전의 정의된 프로토콜을 통해서 패킷들을 주고 받는다. 어플리케이션은 이러한 그 자신의 어플리케이션 프로토콜을 따라 통신하는데, 이 때 몇가지 필드들(프로토콜 정의 부분, 어플리케이션 이름, 헤더 길이등)이 반복적으로 나타나기 쉽다. 이러한 이유로 몇 가지 반복적인 패턴들이 교환되는 패킷의 페이로드등에서 나타나게 된다. 이러한 반복되는 패턴들이 그 어플리케이션 트래픽들 다른 어플리케이션 트래픽과 구분시켜 주는 시그네쳐가 될 수 있다.

2) TCP 핸드쉐이킹 뒤 첫 번째 패킷이 다른 패킷들 보다 어플리케이션 시그네쳐를 포함하고 있을 확률이 높다.

그림 3.1)에서 볼 수 있듯이, P2P시그네쳐를 사용하여 트래픽 분류를 시도한 기존의 연구[4]에서는 P2P트래픽을 TCP 핸드쉐이킹 후 첫 번째 패킷만을 검사 하였을 때, 82%~100%, 두 번째 패킷까지 검사하였을 때는 93%~100%, 세 번째 패킷까지 검사하였을 때는 100%를 분류해 낼 수 있었다.

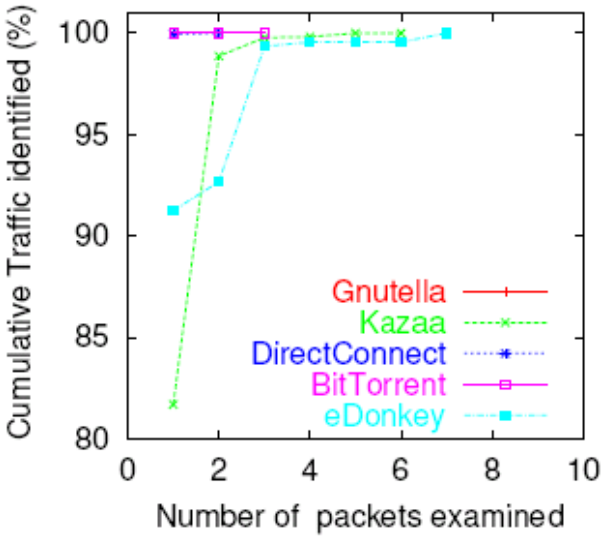

그림 3.1 최대 N번째 패킷을 검사하였을 때, 총 분류된 어플리케이션 트래픽의 누적 분산[4]

많은 P2P 어플리케이션들이 TCP 핸드쉐이킹 이후 첫 번째에서 세 번째 패킷까지 그들의 시그네쳐를 포함하고 있다. 본 연구에서는 첫 번째 패킷들만으로 시그네쳐를 생성하는데 포커스를 하도록 하겠다.

(2) Problem statement

이 장에서는 이 논문에서 해결하고자 한 문제(problem)를 동기(motivation)들과 함께 명확하게 기술하고 이를 해결할 수 있는 접근 방안에 대해서 설명하도록 하겠다.

과거 정적 포트 할당(static port allocation)를 통해서 통신하는 어플리케이션의 트래픽을 분류하는데 에는 서비스가 사용하는 지정된 포트를 기반으로 트래픽을 분류할 수 있었지만, P2P와 같은 동적 포트 할당 방식(dynamic port allocation)을 통해서 통신하는 어플리케이션이 증가함에 따라 기존의 포트기반 분류방식이 한계성이 나타나게 되었다. 어플리케이션이 동적 포트 할당 방식으로 통신할 경우 지정된 포트로 통신하지 않기 때문에 특정 어플리케이션의 트래픽을 파악하기가 힘들어 졌다. 예를 들어 P2P 어플리케이션 edonkey는 디폴트로 포트 46661에서 4665까지의 포트를 사용하게 설정되어 있지만 이는 사용자의 선택에 따라 달라지게 될 수 있어서 포트 4661에서 포트 4665까지의 트래픽을 edonkey로 분류할 경우 edonkey의 트래픽이 포함될 수 있지만 다른 트래픽이 포함 될 수 있고(false positive) 다른 포트를 통해서 통신하는 edonkey트래픽을 파악할 수 없게 된다(false negative)[2]. 이러한 동적 포트 할당 방식을 사용하는 어플리케이션의 트래픽을 분류해 내기 위해 제안된 방식이 시그네쳐 기반 분류방식이다. 시그네쳐 기반 분류방식은 특정 어플리케이션이 통신할 경우 패킷들의 페이로드에 그 어플리케이션임을 나타내는 시그네쳐(signature)가 존재한다는 관찰을 기반으로 어플리케이션 트래픽을 분류하는 방식이다. 예를 들어 Bittorrent라는 P2P어플리케이션이 통신할 경우 TCP 핸드쉐이킹(handshaking)이 끝난 후 첫 번째 패킷의 페이로드 0번째 오프셋(offset)위치에 "(0x13)BitTorrent protocol" 라는 문자열이 존재한다. 동적 포트 할당 방식을 사용하는 어플리케이션의 트래픽 분류의 경우 포트 기반보다 시그네쳐 기반 분류방식이 정확도가 높다고 알려져 있다[4].

Protocol	All Connections	
	Port-based (MB)	Signature-based (%)
Gnutella	487.12	145
Kazaa	548.41	347.38
DirectConnect	2000.75	163.45
BitTorrent	54444.67	90.97
eDonkey	2149.84	102.37

표 3.1 포트 기반 방식과 시그네쳐 기반 분류 방식의 정확도 비교[4]

시그네쳐 기반 어플리케이션 트래픽 분류를 위한 알려진 (well-known) 시그네쳐들은 여러 연구[2], [4], [6], [7]에서 생성하여 알려져 있다. 이 연구들에서의 시그네쳐 생성은 1)특정 어플리케이션의 프로토콜 정의 문서를 참조하거나 2)특정 어플리케이션의 패킷들만이 모여져 있는 순수 트래픽에서 수작업으로 생성하는 것이었다. 1) 생성 방식의 경우, 어플리케이션 프로토콜 정의 문서가 존재하지 않거나 외부에 공개되어 있지 않은 경우가 있다. 특정 어플리케이션이 어떠한 기업에 의해서 만들어졌다면 기업 비밀 보호 차원에서 프로토콜 정의 문서가 외부에 공개되지 않을 수 있다. 개인이 어플리케이션을 만든 경우에는 프로그램 개발 이외의 작업 분량을 줄이기 위해서 프로토콜 정의 문서를 만들지 않았을 수도 있다. 2) 생성 방식의 경우, 하나의 순수한 어플리케이션의 트래픽을 수집하기가 어렵고 수집하여도 각 패킷들 중 어떠한 바이트 시퀀스(byte sequence)가 시그네쳐인지 사람이 직접 눈으로 확인해 가면서 생성해야 하는 휴먼 인텐시브 작업(human intensive job)이다. 이러한 한계성 이외에도 인터넷 상에 존재하는 수천의 어플리케이션 시그네쳐를 생성하는 데 있어서 이러한 접근 방식은 너무나 비효율적이다.

또한 이렇게 생성된 어플리케이션 시그네쳐가 바뀌는 경우에 있어서도 한계성이 있다. 어플리케이션이 새로운 버젼으로 버젼업그레이드 되거나

패치가 되면서 어플리케이션이 가지고 있던 시그네쳐가 변화할 수도 있기 때문이다. 어플리케이션이 업데이트 될 때마다 시그네쳐의 변화가 있었는지 지속적으로 확인하는 접근방식 또한 비효율적이다.

새로운 어플리케이션이 출현하였을 때에도 위의 접근 방식들은 한계성을 가지고 있다. 새로운 어플리케이션이 나타났는지 확인 할 수 없고, 그 어플리케이션의 시그네쳐가 존재하지 않기 때문에 나타난 순간부터 트래픽을 분류해 나갈 수도 없다.

이러한 한계성들을 가지고 있는 기존의 시그네쳐 생성방식을 극복하기 위해서는 다음의 조건들을 만족하는 새로운 시그네쳐 생성방식을 고안하여야 한다.

이상의 조건들을 만족하면서 위의 한계성을 극복하기 위하여 이 논문에서는 다음의 접근 방식을 제안한다.

네트워크의 특정 지역에서 일정 시간 동안 수집한 트래픽 데이타를 기반으로 어플리케이션 시그네쳐를 자동생성하는 방식을 고안한다.

다음 3장에서는 트래픽 데이터의 수집 방식, 어플리케이션 시그네쳐 자동 생성 방식 디자인과 구현에 대해서 설명하도록 하겠다.

Chapter 4. Design & Implementation

본 장에서는 시그네쳐 자동 생성 방식 디자인과 그 구현에 대해서 논의하도록 하겠다.

(1) Design

본 절에서는 어플리케이션 시그네쳐 자동 생성기의 컴포턴트들의 디자인 이슈에 대해서 논의해 보도록 하겠다. 본 절에서의 어플리케이션 시그네쳐 자동 생성 방식은 어떠한 어플리케이션의 순수한 트래픽이 존재할 때, 이 트래픽 데이타를 기반으로 생성하는 방식이다. 특정 어플케이션의 순수한 트래픽 수집에 관한 문제는 구현(implementation)에서 확장시켜 다룰 것이며, 문제를 좀 더 단순화시켜서 점진적으로 풀어나가도록 하겠다. 어떠한 한 어플리케이션의 트래픽만을 수집하는 것은 어려운 일이며, 우리의 접근 방식은 트래픽 내에서 여러 어플리케이션의 트래픽이 섞여 있더라도 많은 양을 차지 하는 주요한 어플리케이션의 시그네쳐 생성에 초점을 맞추고 있다. 이 후 5장에서 우리의 접근 방식을 평가할 때, 이 문제에 대한 우리의 접근 방식의 강건함(Robustness)을 보여주도록 하겠다.

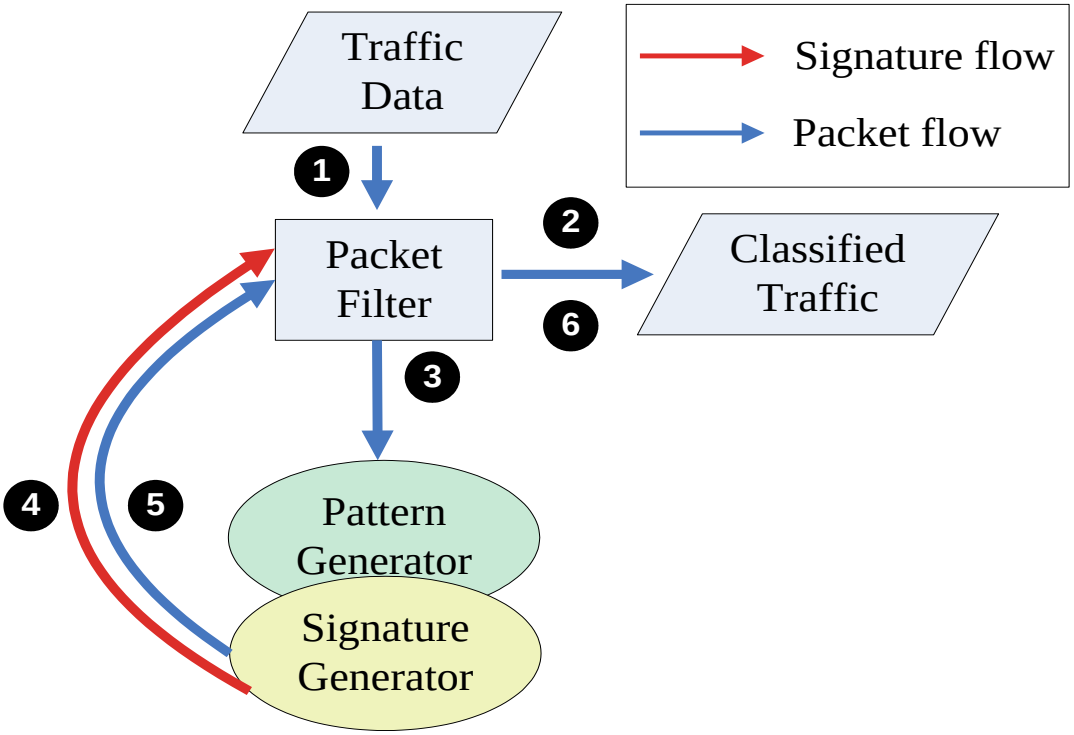

그림 4.1 어플리케이션 시그네쳐 자동 생성기 개괄도

1) 트래픽 데이타 수집

시그네쳐 생성을 위한 트래픽 수집에 있어서는 고려해야 하는 문제는 크게 두 가지로 나눌 수 있다. 첫 번째로 트래픽 데이터 수집에 관한 측면이고 두 번째는 수집된 트래픽 중 시그네쳐 생성을 위한 패킷들의 선별이다. 첫 번째 측면에서는 시그네쳐 생성을 위한 트래픽 수집 방식과 수집 위치를 고려해야만 한다. 시그네쳐 생성을 위한 트래픽 수집 방식으로는 능동적 수집(Active collection)과 수동적 수집(Passive collection) 방식이 존재한다. 능동적 수집은 특정 어플리케이션의 트래픽을 수집하기 위하여 어플리케이션을 실험자가 직접 사용하여 트래픽을 유발시키고 이를 수집하는 방식이다. 의도한 어플리케이션의 순수 트래픽 데이타를 수집하기는 용이하지만 의도한 어플리케이션에만 국한되게 사용될 수 있는

방식이여서 수천의 어플리케이션들의 시그네쳐를 자동 생성하는 방식으로서는 비효율적이다. 또한 실험자가 수행한 작업들(접속, 파일 교환, 접속 종료등)에 대응되는 패킷들만을 수집하여 분석할 수 있기 때문에 실험자의 작업 시나리오를 벗어나는 작업들에 대응되는 패킷 데이타는 수집이 불가능하여, 정확한 어플리케이션 시그네쳐를 생성하지 못하는 경우도 있다. 이에 반해 수동적 수집방식에 의한 트래픽 데이타 수집은 수집되는 어플리케이션의 수가 그 네트워크에서 사용되는 어플리케이션 수만큼 다양하다. 이들 어플리케이션들은 그 네트워크에서 사용되는 어플리케이션인 만큼 네트워크 관리자가 관심을 가지고 지켜보고자 하는 어플리케이션들이여서 시그네쳐 생성 시 네트워크 관리자에게 중요한 정보로 이용될 수 있다. 네트워크 규모가 크면 클수록 사용자의 수가 증가하고 이에 따라 사용자의 작업 패턴이 다양해 질 수 있기 때문에 특정 어플리케이션의 시그네쳐 생성에 있어서 능동적 수집 방식보다 더 높은 정확성을 가질 수 있다. 본 논문에서 제안한 시그네쳐 생성 방식을 기본적으로 수동적 수집을 통한 트래픽을 기반으로 시그네쳐를 생성하지만 능동적 방식으로 수집된 트래픽에 대해서도 시그네쳐를 자동 생성해 낼 수 있다.

두번째로 수집된 트래픽 중 시그네쳐 생성을 위한 패킷 선별의 문제가 있다. 이렇게 수집된 패킷들을 모두 시그네쳐를 생성하는 데 이용하는 것이 아니라 시그네쳐가 존재할 수 있는 패킷들만을 선별하여 그 패킷들을 기반으로 시그네쳐를 생성해 내야 한다. P2P 시그네쳐를 통한 어플리케이션 트래픽 분류의 이전의 연구 [4]에서는 TCP 핸드쉐이킹(handshaking) 후 첫번째 패킷만을 검사하여 트래픽을 분류 시 각 P2P어플리케이션의 82%~100%까지 분류가 가능하며, 두번째 패킷까지 검사할 경우 92%~100%, 세번째 패킷까지 검사할 경우 99.99%~100%까지 트래픽을 분류해 낼 수 있었다. 즉, 세번째 패킷내에 특정 P2P 어플리케이션의 트래픽을 다른 어플리케이션의 트래픽과 구분해 주는 시그네쳐가 존재하였다는 것이다. 본 논문에서는 문제의 단순화를 위해서 수집된 트래픽 데이타 중 TCP

핸드쉐이킹 후 첫번째 패킷만을 우선 분석하였다. 세번째 패킷까지의 분석은 Future Work으로 남겨두도록 하겠다.

2) Pattern generation

어플리케이션 시그네쳐 자동 생성에 관한 알려진 효율적인 접근 방법이 존재하지는 않지만, 인터넷 웜(worm)의 시그네쳐 자동 생성에 관한 접근 방식은 [7]과 [8]에서 제안되었다. 인터넷 웜의 시그네쳐의 자동생성과 어플리케이션의 시그네쳐 자동 생성이 기본적으로 네트워크 트래픽 데이타 내에 존재하는 공통 패턴을 찾아내는 점에서 비슷한 성격이 존재하지만, 웜과 어플리케이션의 트래픽 성격(웜은 짧은 시간에 다수의 호스트에 접근성을 보이지만 어플리케이션의 경우 다수의 호스트에 접근하는데 걸리는 시간이 웜보다는 상대적으로 길다), 시그네쳐의 길이(알려진 어플리케이션의 시그네쳐는 1~20바이트 정도로 짧지만, 알려지 있는 웜의 시그네쳐는 몇 백바이트 수준으로 상대적으로 어플리케이션보다 긴편이다)등에서 차이를 보이고 있다. 본 논문에서는 이전의 연구[7]에서와 같이 패킷들 내에 존재하는 일정한 패턴들을 생성하여 이 패턴들의 조합으로 시그네쳐를 생성해 나가는 접근방식을 선택하였다. 이 접근 방식은 트래픽 내에 공통적인 시그네쳐가 존재할 때, 웜이나 어플리케이션의 상관없이 적용가능한 접근 방식이다.

Pattern generator는 패킷의 페이로드를 랜덤하게 잘라 패턴을 생성하고, 각 생성된 패턴으로 트래픽 데이터의 패킷 중 몇 개의 패킷을 분류해 낼 수 있는 지 조사하는 것이다. 패턴으로 트래픽 데이터를 조사할 때에는 패턴의 offset과 real value값을 사용한다. 패턴의 offset은 패킷에서 페이로드가 시작되는 부분을 0의 위치로 잡았을 때의 위치를 의미하여 real value는 패턴의 byte sequence를 의미한다.

가. 패턴 생성

패킷의 어떠한 byte sequence가 공통적으로 나와 시그네쳐가 될 수 있는지 알 수 없기 때문에 payload를 랜덤하게 잘라 패턴을 생성한 후 빈도에 대해 조사하여야 한다. 각 패킷의 payload에 대해서 1~20 사이의 랜덤한 길이로 잘라 패턴으로 생성해 낸다. 이 때, offset과 real value가 같은 패턴이 이미 존재하면 pattern matching단계를 거치지 않고 다음 패턴을 랜덤하게 만들어 낸다. 생성된 패턴은 4.2 pattern matching단계에서 offset과 real value를 비교하여 트래픽 데이터내의 몇 개의 패킷을 분류해 낼 수 있는지 조사된다.

패턴 생성의 길이 기준은 기존에 알려져 있는 시그네쳐 중에 eDonkey의 시그네쳐 0xe3, 0xc5의 1byte를 패턴의 최소 길이로 정하였고, BitTorrent의 시그네쳐 "!BitTorrent Protocol"를 기준으로 하여 패턴의 최대 길이를 20 bytes로 하였다. 랜덤길이는 평균이 10인 normal distribution을 따른다.

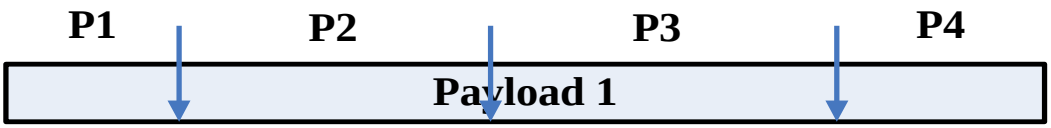

그림 4.2 패턴 랜덤 생성

P_ID	Offset	Length	Byte sequence	Occurrence

표 4.1 패턴 테이블

나. 패턴 비교

앞의 패턴 생성 방식에 의해서 생성된 패턴과 네트워크 트래픽 데이터내의 모든 패킷들의 payload를 비교하여, 이 패턴이 몇 개의 패킷에서 나타나는 지, 즉 이 패턴으로 전체 네트워크 트래픽 데이터 중에서 몇 개의 패킷을 분류해 낼 수 있는지를 조사한다.

생성된 패턴을 트래픽 데이터의 패킷들의 페이로드와 비교하기 위하여, 패턴의 real value 값을 비교하기 전에 hash value를 구하여 비교한 후, hash value 가 일치할 경우 패킷들의 페이로드의 real value와 패턴의 값을 비교함으로써 패턴값을 비교하는 overhead를 줄였다. String과 text를 비교하는 방식으로는 대표적으로 libc, AST regex, Karp-Rabin Fingerprinting 방식 등이 있으며 이 중에선 AST regex방식이 성능 면에서 우수하다고 알려져 있다. [3] 하지만 application signature 자동 생성기가 이 후 라우터와 같은 하드웨어 장비에 모듈로 들어갈 필요가 있으므로 구현의 용이와 하드웨어 구현의 장점을 가지는 Karp-Rabin Fingerprinting방법을 우선적으로 적용해 보았다. (Karp-Robin Fingerprinting은 1bit shifting이나 bit operation이 많아서 하드웨어 구현에 용이하다)

- Karp-Robin Fingerprinting

문자 비교를 하지 않고 문자를 숫자로 치환하여 비교 후 같다면 실제 비교 수행

P: pattern T: text

n: pattern P의 길이 m: text T의 길이

H: strings → numbers

Let s be a string of length n,

$$H(s) = \sum_{i=1}^{n} 2^{n-i} s(i)$$

Definition: let Tr denote the n length substring of T starting at position r.

예>

T = 1 0 1 1 0 1 0 1

P = 0 1 0 1

T = 1 0 1 1 0 1 0 1 H(T2) = 6 ∴ H(T2) ≠ H(P)

P = 0 1 0 1 H(P) = 5

T = 1 0 1 1 0 1 0 1 H(T5) = 5 ∴ H(T5) = H(P)

P = 0 1 0 1 H(P) = 5

Theorem:

There is an occurrence of P starting at position r of T **if and only if** H(P) = H(Tr)

We can compute H(Tr) from H(Tr-1)

$$H(T_r) = 2H(T_{r-1}) - 2^n T(r-1) + T(r+n-1)$$

예>

T = 1 0 1 1 0 1 0 1 T1 = 1 0 1 1

T2 = 0 1 1 0

$$H(T_1) = H(1011) = 11$$
$$H(T_2) = 2 \cdot 11 - 2^4 \cdot 1 + 0 = 22 - 16 = 6 = H(0110)$$

For some integer p The *fingerprint* of P is defined by Hp(P) = H(P) (mod p)

Lemma:

$$H_p(P) = \{[...(\{[P(1)\cdot 2(\text{mod } p) + P(2)]\cdot 2(\text{mod } p) + P(3)\}\cdot 2(\text{mod } p) + P(4))...](\text{mod } p) + P(n)\}(\text{mod } p)$$
$$= H(P)(\text{mod } p)$$

And during this computation no number ever exceeds 2p.

예>

P = 1 0 1 1 1 1 H(P) = 47

p = 7 Hp(P) = 47 (mod 7) = 5

$$1\cdot 2(\text{mod } 7) + 0 = 2$$
$$2\cdot 2(\text{mod } 7) + 1 = 5$$
$$5\cdot 2(\text{mod } 7) + 1 = 4$$
$$4\cdot 2(\text{mod } 7) + 1 = 2$$
$$2\cdot 2(\text{mod } 7) + 1 = 5$$
$$5(\text{mod } 7) = 5 = H_p(P)$$

Karp-Rabin 적용 알고리즘

P(i) : Pattern의 i번째 byte

$T_i(k)$: Text의 i번째 byte부터 k개의 byte

패턴의 길이를 n

텍스트의 길이를 m

Modulor M 라 할 때,

$H(P) = \{ P(1) + P(2) + \cdots + P(n) \} / M$

$H_i(T) = \{ T(i) + T(2) + \cdots + T(i+n-1) \} / M$

for(i = 1 to m-n)

{

if($H(P) == H_i(T)$)

compare P and T_i

else

do nothing

$H_{i+1}(T) = \{H_i(T) - T_{i-1}(1) + T_{i+1}(1) \} / M$

}

여기서 M은 nm^2보다 작은 소수(Prime)를 잡는 것이 좋다.

패턴과 텍스트의 비교 시 match가 된다는 것은 다음의 3가지 조건을 만족하여야 한다.

① **Karp-Rabin fingerprint 값이 같다**

② **두 string 이 실제로 같다**

③ **두 string 의 packet payload 에서의 offset 이 같다.**

3) Pattern clustering

2절에서 패턴들을 생성하고 생성된 패턴들의 트래픽 데이타에서 나타나는 패킷 occurance를 조사하였다. 본 3절에서는 이러한 패턴들을 occurance에 기반해서 그룹을 짓고 각 그룹마다 패턴들 간의 포함관계를 조사하여

패턴들을 머징(merging)하여 준다.

이 작업이 필요한 이유는 2절에서 패킷의 어떠한 바이트 시퀀스(byte sequence)가 시그네쳐인지 예상할 수 없으므로 랜덤하게 페이로드(payload)를 잘라 패턴을 생성하였는데, 생성된 패턴들의 패킷 occurance가 동일한 패턴들이 존재하였다. 즉, 트래픽 데이타 내에서 같은 패킷 occurance만큼 그 패턴들이 나타났다는 것이다. 이렇게 occurance가 같은 패턴들을 조사하여 보니 패턴들간의 상관관계가 존재하였다. 같은 occurance를 가지는 패턴들은 어떠한 바이트 sequence의 subset들이여서, 패턴 생성시에 각 패킷의 페이로드 마다 랜덤하게 잘려서 생성되게 됨으로 인해서 같은 byte sequence가 각 패킷들마다 다르게 잘리게 되어 다른 패턴이 형성되었다. 하지만 전체 트래픽에서 그 byte sequence가 동일하게 존재하였기 때문에 생성된 패턴들은 동일한 occurance를 가지게 된 경우가 많았다. 이렇게 패킷 occurance가 같고, 하나의 동일한 byte sequence의 sub-byte sequence인 패턴들을 머징(merge)하여 초기의 하나의 동일한 byte sequence를 생성해 내는 것이 이 패턴들을 포함하고 있는 패킷들 간에 가장 긴 공통적인 바이트 시퀀스(longest common byte sequence)이다. occurance가 같은 패턴들을 결합(merge)하여 가장 긴 공통적인 바이트 시퀀스를 생성할 때 패턴 생성시 랜덤하게 자름으로 인해 완전하지 않고 특정 byte가 누락될 수도 있지만, 본 논문에서 충분히 큰 트래픽 데이타를 기반으로 패턴을 생성하기 때문에 확률상 중간이 누락되어 생성되는 경우는 발생하지 않았다. 이 가장 긴 공통적인 바이트 시퀀스가 하나의 시그네쳐 후보가 될 수 있으며, 다음 4절에서 이 시그네쳐 후보들을 결합하고 선별하고 트래픽 데이타를 대표할 수 있는 시그네쳐를 생성해 낸다.

예를 들어, (0x)13BitTorrent라는 잘 알려진 (well-known) 시그네쳐를 가지는 Bittorrent라는 어플리케이션의 시그네쳐를 고려하여 보자.

전체 패킷 : N

A : 페이로드 위치 0에서부터 시작하는 (0x)13BitTorrent라는 byte seqence

Occ(x) : x라는 byte sequence가 나타난 패킷 수

Occ(A) = N

If P_1, P_2, P_3, … , P_n is sub-byte sequence of A (disjoint subset),

$Occ(P_1) >= N$, $Occ(P_2) >= N$, $Occ(P_3) >= N$, … , $Occ(P_n) >= N$

(우연적으로 같은 오프셋에 같은 byte sequence가 나타날 수 있으므로 N보다 크거나 같다.)

이때, 패킷들 간에 가장 긴 공통 바이트 시퀀스를 생성해 내고자 한다면, A라는 바이트 시퀀스를 얻어내야 하고 이 A는 P_1, P_2, P_3, … , P_n 등을 결합하면 얻어 낼 수 있다. 이 패턴들이 A의 disjoint subset이므로 결합 시 원래의 A를 얻어 낼 수 있고, 전체 패킷 중 단 하나의 패킷에서 패턴 생성시 A의 disjoint subset을 만들어 내면, 항상 A를 다시 생성해 낼 수 있다.

BitTorrent 어플리케이션 같은 경우 트래픽 데이터로부터 패턴들을 생성하면 오프셋 0에 위치하는 “ !B”, “ !Bi”, ” Bi”, “ !Bit”, “ Bit”, “ it”, “ !BitT”, “ !BitTo”, “ !BitTor”, …, “ !BitTorrent” 와 같은 패턴이 생성된다. (“ !” 는 16진수로 (0x)13임) 이 패턴들의 패킷 occurrence가 동일하였을 때, 이 패턴들이 속한 패킷들이 가질 수 있는 가장 긴 바이트 시퀀스는 이 패턴들을 결합(merge)함으로서 얻어 낼 수 있다.

패킷 occurrence가 동일한 패턴들간에는 다음의 세가지 관계 중 하나의 관계에 항상 놓여 있게 된다.

1) Pi is sub-byte sequence of Pj (or Pj is sub-byte sequence of Pi)

2) Pi and Pj have a common byte sequence

3) Pi and Pj have no relation

이 때 두 패턴들의 페이로드 오프셋 위치도 함께 고려되어야 한다. 즉, 두 패턴의 포함관계는 오프셋을 기준으로 하여 포함관계나 겹치는 부분의 유무를 확인하는 것이다. 1)의 경우는 단순히 두 패턴의 오프셋 위치를 기반으로 해서 byte단위로 비교해 나가면 알 수 있다. Pi가 Pj에 오프셋을 기준으로 sub-byte sequence이면 Pj를 더 긴 공통 바이트 시퀀스로 유지하면 된다.

2)와 같이 오프셋을 기준으로 해서 두 패턴이 공통적인 부분을 가지고 있다면, 공통부분을 기준으로 두 패턴을 결합(merge)하여 두 패턴보다 더 긴 공통 바이트 시퀀스를 만들어 낸다. 이때는 1)에서와는 다르게 두 패턴이 같은 어떠한 byte sequence의 부분이었는지에 대한 확인이 필요하다. 예를 들어 Pi = “abcde”, 오프셋 0라고 하고 Pj = “defg”, 오프셋 3이라고 한다면 두개의 패턴이 공통 byte sequence를 가지고 있지만, 실제로 하나의 패킷에서 두 패턴이 함께 존재했는지 유무에 대한 정보는 존재하지 않는다. 이것을 확인하기 위해서는 Pi가 존재한 패킷들에 Pj가 존재 하였는지 유무를 확인해야만 한다.

3)에서는 두 패턴이 하나의 어떠한 byte sequence의 sub-byte sequence가 일수도 있고 아닐 수도 있다. 두 패턴들을 이어주는 중간에 위치하는 패턴이 패턴 생성 단계에서 만들어 졌다면 두 패턴은 그 중간 패턴을 매개로 하여 1)이나 2)의 방법으로 합쳐지게 된다. 만약 두 패턴을 이어주는 패턴이 존재하지 않거나 처음부터 두 패턴 자체가 어떠한 하나의 byte sequence의 sub-byte sequence가 아니었다면, 두 패턴은 어떠한 작업도 해 줄 필요가 없다.

이상의 관계를 정리하여 Occurrence가 같은 패턴들을 결합(merge)시켜주는 알고리즘을 만들면 다음과 같다.

```
P = {P1, P2, ..., Pn}                          // list of patterns

struct pattern {                               // structure of pattern P1,P2, …, Pn
        int size;
        int occurrence;
        int merged ;
        int hash_value;
        char *real_value;
} P;

P1. occurrence = P2. occurrence = ... = Pn. occurrence      // occurrence of all
                                                                //patterns is same
P1.size <= P2.size <= ... <= Pn.size                        // set P is sorted by
                                                                //size of patterns
P1.merged = P2.merged = … = Pn.merged = 0

/*-----------------------------------------------------
   Pi 와 merge되는 패턴이 있는지 찾는다. Pi가 merge되면 Pi+1과 merge
   되는 패턴을 찾는다.
-----------------------------------------------------*/
merge_same_count()
{
```

```
for(i = 1 to n-1) {.
 if(!Pi.merged){                                    //Pi이 아직 merge 안 된 경우

  /* Pi와 merge되는 패킷이 있는지 Pi+1인 패턴부터 조사한다.
     이 때 Pi+1이 이미 merge되었다면 그 패턴은 조사하지 않는다. */
   for(j = i+1 to n) {
     if(!Pj.merged){                              // Pj가 아직 merge 안 된 경우

        /*   Pi∈Pj 인 경우, Pi.merged = True, Pj.merged = False*/
          if( Pi is substring of Pj){
             Pi.merged = TURE;
             break;
           }
           /*Pi와 Pj가 겹치는 경우, Pi와 Pj를 merge해 얻은 새로운
           패턴을 Pj 값에 넣고, Pi.merged = True, Pj.merged=False */
           else if( there are any characters included both Pi and Pj)
           {
             Ptemp = merge_pattern(Pi, Pj);
             copy Ptemp to Pj;
             Pi.merged = TRUE;
               break;
```

```
                }
            }
        }
        else          // Pj.merged == True 인 경우
            j++;
    }
}
else                  // Pi.merged == True인 경우
    i ++;
Get patterns whose merged field is false from P;
}
```

특히 2)과 같이 패킷 확인이 다시 필요한 경우 실제로 구현시에는 패턴 클러스터링 단계를 시그네쳐 생성 단계에 삽입하여 프로세싱 속도를 향상 시킬 수 있었다.

이 알고리즘에서 패턴들은 occurrence를 sort의 첫번째 키로, byte sequence 길이를 두번째 sort키로 사용하여 정렬 되어진 상태에서 알고리즘을 적용하였다. occurrence별로 우선 정렬한 이유는 앞에서와 같이 같은 패킷 occurrence를 가지는 패턴끼리 결합 시키고자 하기 때문이고 두번째 키로 패턴의 길이를 선택한 이유는 같은 occurrence내에서 패턴의 길이가 긴 패턴이 다른 패턴을 포함할 확률이 짧은 것보다는 높기 때문에 패턴의 길이가 긴 순서대로 정렬하여 결합하는 휴리스틱을 사용하였다.

4) Signature generation

본절에서는 3절에서 클러스터링 된 패턴들을 기반으로 시그네쳐를 생성하는 방식에 대해서 논의하겠다. 앞 절에서는 시그네쳐의 빌딩 블록으로서의 패턴들의 결합에 중점을 두었다면 본 절에서는 결합된 패턴들을 하나의 시그네쳐 후보로 생각하고 이들 후보들 중에서 시그네쳐로서 다른 패턴들보다 자질이 우수한 패턴을 시그네쳐로 선택하는 문제에 중점을 두어 해결해 나가도록 하겠다. 물론 패턴들 간의 선택문제에서 패턴들 간의 결합이 좀 더 좋은 시그네쳐로 존재한다면, 패턴들을 결합하여 좀 더 패킷들에 대한 대표성이 강한 시그네쳐를 생성해 내도록 하겠다.

패턴을 선택하여 시그네쳐로 결정하는 메인 휴리스틱은 다음과 같다.

(ㄱ) 패턴의 패킷 occurance가 높을수록 시그네쳐로서 자질이 우수하다.

패턴이 나타나는 패킷 수가 많으므로 그 패턴을 시그네쳐로 삼을 경우 다른 패턴들 보다 더 많은 패킷들을 분류해 낼 수 있다.

(ㄴ) one byte 패턴은 시그네쳐로 삼지 않는다

one byte 패턴의 경우 패킷 occurrence에서 항상 상위를 차지하고 있다. 이는 반복되는 어떠한 byte sequence내에 존재하는 one byte 패턴으로 occurrence가 조사되기도 하지만 우연적으로 어떠한 byte sequence가 반복되지 않을 때도 나타날 수 있기 때문이다. 예를 들어 “!BitTorrent”의 오프셋 0에서의 “!”라는 패턴의 occurrence는 패킷들 중에서 오프셋 0 위치에 “!BitTorrent”라는 byte sequence가 반복되어서 카운트되기도 하지만 우연적으로 어떤 패킷에서 오프셋 0위치에 “!”라는 byte sequence를 가지고 있게 되어 카운트 될 확률도 있기 때문이다. 비록 edonkey라는 어플리케이션의 잘 알려진(wellk-known) 시그네쳐가 one byte로서 오프셋 0의

위치에 “(0x)e3”이나 (0x)c5”이라 할지라도 이러한 one byte 시그네쳐는 그 어플리케이션 트래픽을 다른 어플리케이션의 트래픽과 구분하게 해주기에는 false positive가 너무 높다. 이를 위해 one byte패턴이 대표하는 패킷들을 대표해 줄 수 있는 다른 패턴들을 선택하여 시그네쳐를 생성해 주었다.

(ㄷ) one byte 패턴을 대신하여 one byte패턴이 대표하는 패킷들을 대표해 낼 수 있는 패턴들을 시그네쳐로 선택한다.

전체 패킷의 집합 : A

One byte 패턴 : P0, One byte 패턴의 오프셋 : P0off

Pkt0 = {x | x ∈ A and x includes P0 on offset P0off}

Find P1, P2, … , Pn whose offset are P1off, P2off, …, Pnoff

P1 = { x | x ∈ Pkt0 and x includes P1 on offset P1off}

P2 = { x | x ∈ Pkt0 and x includes P2 on offset P2off}

.

.

.

Pn = { x | x ∈ Pkt0 and x includes Pn on offset Pnoff}

And P1 ∩ P2 = Ø ,P1 ∩ P3 = Ø ⋯ and P1 ∩ Pn = Ø

P2 ∩ P3 = Ø, P2 ∩ P4 = Ø ⋯ and P2 ∩ Pn = Ø

.

.

.

Pn-1 ∩ Pn = ∅

And n(P1) + n(P2) + …. + n(Pn) <= n(P0) ;

이상의 조건을 만족시키는 P1, P2, …., Pn을 최대한 많이 찾고 이를 시그네쳐로 등록 시킨다.

(ㄹ) 시그네쳐 리스트에 추가된 시그네쳐들이 대표하는 패킷들에 포함된 패턴들은 조사하지 않고 스킵(skip)한다.

이미 시그네쳐 리스트에 추가된 시그네쳐들이 해당 패킷을 대표하는 시그네쳐가 되었기 때문에 새로운 패턴이 그 해당 패킷을 대표할 필요가 없다.

(ㅁ) 한 개의 패킷만을 대표하는 패턴으로는 시그네쳐를 생성하지 않는다.

한 개의 패킷에서만 나타나는 패턴은 어떠한 반복적인 byte sequence로 볼 수 없기 때문에 비록 하나의 패킷을 대표하지만 시그네쳐로 인정하지 않는다.

이상의 휴리스틱으로 디자인한 시그네쳐 생성 알고리즘은 다음과 같다.

```
struct pattern {                          // structure of pattern P1,P2, …, Pn
    int size;
    int occurrence;
```

```
    int merged ;
    int hash_value;
    char *real_value;
} P;



S = {}                                          // 새로 생성될 시그네쳐가 들어갈
배열
P = {P1, P2, ..., Pn}                          // list of patterns

P1. occurrence >= P2. occurrence >= ... >= Pn. occurrence

int total_packets                    // 데이터 트래픽의 총 패킷 수
int coverage_count                   // 시그네쳐 목록 S의 시그네쳐들로 인해
                                     // unknown에서 known으로 분류되는 패킷 수
int  sub_cov_count                   /* 1byte pattern일 때 사용될, 패턴 pi를
                                     대신해 시그네쳐 리스트에 들어가는
                                     패턴들이 분류해 내는 패킷 수 */
signature_generating ()
{
    for(i = 1; i <=n; i ++)          //pattern P1 ~ Pn까지
    {
        /* check_existence()는 패턴 Pi로 분류되는 패킷에 시그네쳐 목록
```

```
        S의 시그네쳐가 동시에 나오는 지를 조사하는 함수이다. 동시에
        나오면 True값이 리턴 되고, 동시에 나오지 않으면 false값이 된다.

        패턴의 occurrence가 전체 패킷 중 시그네쳐 목록 S로 분류된
        패킷 수를 뺀 unknown인 패킷보다 작은 패턴만 조사한다. */

flag = False;

if( (Pi.occurrence <= total_packets - coverage_count) &&

 ((flag = check_existence(Pi)) == False))

 {

        if(Pi.size == 1)              // 패턴 길이가 1 byte인 경 우

        {

             sub_cov_count = 0;


             /* Pi가 1 byte이므로, Pi를 포함하고 시그네쳐가 될 수
             있는 패턴들을 찾기 위해서 Pi+1부터 Pn까지 Pi가 분류
             할 수 있는 패킷을 다 분류할 때까지 반복 된다.      */

             for(j = i +1 1; j <= n; j++)

             {

                  flag = False;


            /*패턴 Pj의 occurrence가 P가 분류할 수 있는패킷 수에서 Pi
            대신에 시그네쳐가 된 시그네쳐들로 분류된
            패킷 수를 뺀 unknown인 패킷보다 작은 패턴만 조
            사한다. compare 함수는 Pi가 Pj에 포함되는지를
            검사한다. Pi가 Pj에 포함될 때 Pj를 S에 추가한다*/
```

```
                    if((Pj.occurrence <= Pi.occurrence - sub_cov_count)
                     && ((flag = check_existence(Pj)) == 0) &&
                    (compare(Pi, Pj) == TRUE))
                      {
                            coverage_count += Pj.occurrence;
                           sub_cov_count += Pj.occurrence;
                            add Pj to S[];

 /* Pi가 분류할 수 있는 패킷 수보다 Pi를 대신 시그네쳐 조건을 만족하는
패턴들에 의해 분류되는 패킷수가 더 많을 때 더 이상 패턴 Pj를 조사하지
않는다. */
                            if(sub_cov_count >= Pi.occurrence)
                                  break;
                             }
                  }          // end of for j

          }                  // if pattern size is one byte
          else               // 패턴 Pi의 size가 1 bytes가 아닐 경우.
                             // Pi를 시그네쳐 목록 S에 추가시킨다.
          {
            coverage_count += Pi.occurrence;
              add Pi to S[];
```

```
                    // 시그네쳐 목록 S의 시그네쳐들로 데이터 트래픽의전체
                    패킷을 분류하였으면 더 이상 패턴을 조사하지 않는다. */
                            if(coverage_count >= total_packets)

                                break;
                        }
                }
            }  //end of for i
}
```

위 알고리즘을 사용하여 패턴으로부터 시그네쳐를 생성하게 되면 생성된 시그네쳐는 트래픽 데이터의 패킷들을 대표하게 된다. 즉, 트래픽 데이터의 모든 패킷들을 생성된 시그네쳐 만으로 분류가 가능하게 된다. (이 때, 반복되는 공통 byte sequence를 가지지 않고 한번만 나타난 패킷들에 대해서는 분류를 할 수 없고, 이러한 패킷들로 인해 100%보다 낮은 비율로 트래픽 데이터를 분류해 내기도 한다.)

(2) Implementation

본 절에서는 본 연구에서 제안한 어플리케이션 시그네쳐 자동 생성 알고리즘을 바탕으로 어플리케이션 시그네쳐 자동 생성기를 구현하면서 고려하였던 구현 이슈들에 관해서 다루고자 한다. 구현상에 수정된 어플리케이션 시그네쳐 자동 생성기는 그림 4.1과 같다.

1) 트래픽 데이터

(ㄱ) 트래픽 데이터 수집

본 연구에 필요한 트래픽 데이타는 카이스트 대그몬(Dagmon) 이라는 패시브(Passive) 모니터링 장비를 사용하여 수집하였다. 대그몬 모니터링 시스템(그림 4.4)은 카이스트 보더(border) 라우터와 ISP업체인 하나로 라우터 간에 연결된 링크를 탭핑하여 패시브하게 모니터링 하는 장비로서 양 방향에 대해 전체 패킷(헤더+payload)를 수집할 수 있는 장비이다. 연구에 필요한 네트워크 트래픽 데이타는 패킷의 헤더와 페이로드(payload)가 필요하였고 이에 전체 트래픽을 양 방향에 대해 각각 수집하였다. 이 탭핑 위치는 카이스트에서 나가고 들어오는 트래픽의 99%이상을 관찰할 수 있는 위치이며 1000Mbit/s의 사용률을 보이고 있는 링크이다. 카이스트의 가장 높은 사용률을 보이는 링크들 중 하나이며 가장 많은 사용자들의 어플리케이션 사용 패턴이 보일 수 있는 링크 중 하나이다. 이 트래픽 데이터에서 어플리케이션 시그네쳐 생성을 위해 TCP 핸드쉐이킹(handshaking)후에 첫 번째 패킷만을 시그네쳐 생성에 사용하였다.

(ㄴ) 트래픽 데이터 패킷 구조

수집한 네트워크 트래픽 data는 그림 4.3와 같다. 대그몬 시스템에서 수집 시 기본적으로 Extensible Record Format(ERF)[7] 포맷으로 패킷을 수집한다. 본 연구에서는 ERF 헤더, 이더넷(Ethernet) 헤더, IP 헤더, TCP 헤더를 제외한 부분에 대해서 페이로드로 생각하고 이 페이로드 내에 존재하는 공통적으로 반복되는 byte sequence, 어플리케이션 시그네쳐를 생성한다.

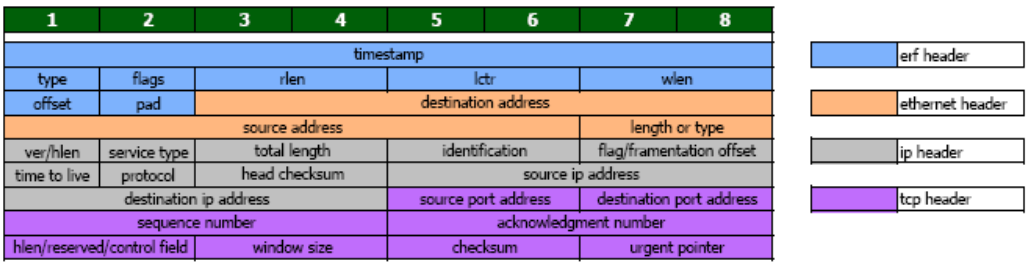

그림 4.3 Raw Traffic data 패킷 구조 (단위 byte)

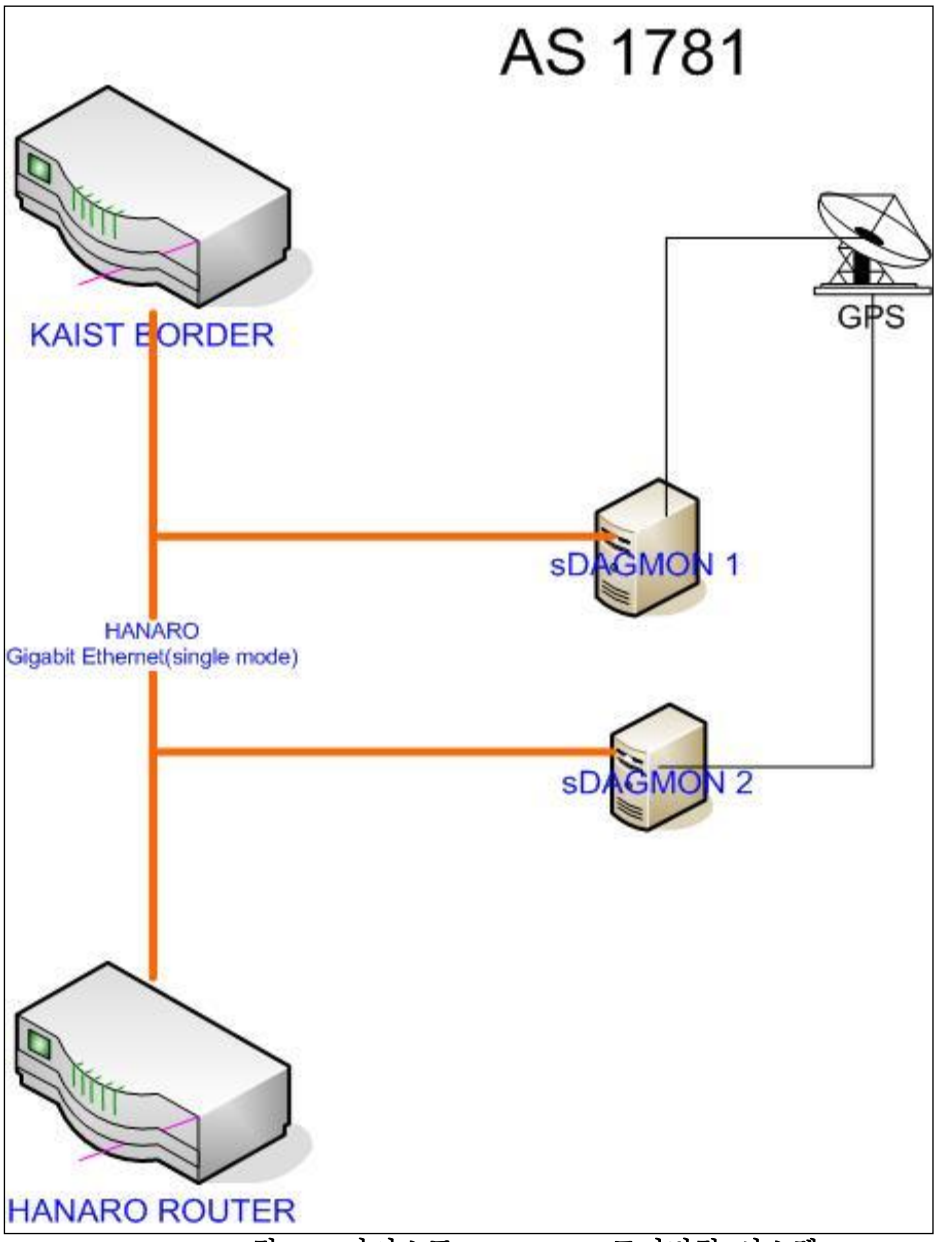

그림 4.4 카이스트 DAGMON 모니터링 시스템

2) Packet filter

(1)절에서 시그네쳐를 자동 생성하는 방식은 기본적으로 특정 어플리케이션에 대한 순수한 트래픽 데이터를 기반하여 생성하는 방식이다. 하지만 수동적(passive) 트래픽 수집으로는 특정 어플리케이션에 대한 순수한 트래픽 데이터를 수집하는 것은 어려운 일이며 현재도 연구되고 있는 분야이다. 본 논문에서는 특정 어플리케이션의 순수한 트래픽을 수집하는 것은 어렵다고 판단하여 순수한 트래픽에 근접한(approximate) 트래픽을 수집하고 이 트래픽을 기반으로 시그네쳐를 생성한다. 이렇게 생성된 시그네쳐는 하나의 어플리케이션의 시그네쳐가 아니라 다수의 어플리케이션 시그네쳐가 될 수 있고, 하나의 어플리케이션에 속하는 시그네쳐들을 묶어주는 문제는 Future Work으로 남겨두었다.

그림 1)에서 보는 바와 같이, 순수한 트래픽에 근접한 트래픽을 생성해 내기 위해서 우선적으로 잘 알려진 (well-known) 시그네쳐 리스트를 만들고 이 리스트를 기반으로 트래픽을 필터링을 하였다. 잘 알려진 시그네쳐 리스트는 표 4.2)과 같다.

시그네쳐 #1	어플리케이션	시그네쳐	오프셋(offset)
1	HTTP	HTTP	0
2		GET	
3		POST	
4	eDonkey	0xe3	0
5		0xc5	
6	BitTorrent	0x13BitTorrent	0

7	Gnutella	GNUTELLA	0

표 4.2 이미 알려진 시그네쳐 리스트

그 이후 알려진 시그네쳐에 의해 걸러지지 않는 트래픽에 한하여 목적 포트(destination port) 별로 트래픽을 분류하였다. 이렇게 포트 별로 분류된 각각의 트래픽은 순수한 어플리케이션의 트래픽은 아니더라도 트래픽 내에 존재하는 어플리케이션의 수가 대폭적으로 줄어든 순수함에 근접한 트래픽 데이터가 된다. 또한 포트별로 모인 트래픽을 그 크기에 따라 정렬하면 상위에 존재하는 포트들은 그 포트를 디폴트 포트로 사용하는 어떠한 어플리케이션이 존재할 확률이 높게 된다. 비록 동적 포트 할당을 특정 어플리케이션이 사용한다고 하더라도 그 어플리케이션이 기본적으로 사용하는 디폴트 포트가 존재하는 경우가 많고, 이러한 경우 포트별로 트래픽을 분류하였을 때 해당 어플리케이션이 모든 포트에 분산되어 있지만 특정 디폴트 포트에 몰려있는 경향을 띠게 된다. 이러한 어플리케이션의 경우 트래픽 양이 많은 포트 별로 시그네쳐를 생성한 후 생성된 시그네쳐를 피드백(feedback)하여 다시 적용하면, 다음 포트 트래픽에서는 이전보다 더욱 순수한 트래픽 하에서 시그네쳐를 생성할 수 있고 피드백이 계속 되면 될수록 포트 각 각에 존재하는 트래픽은 하나의 어플리케이션으로 줄어들게 된다.

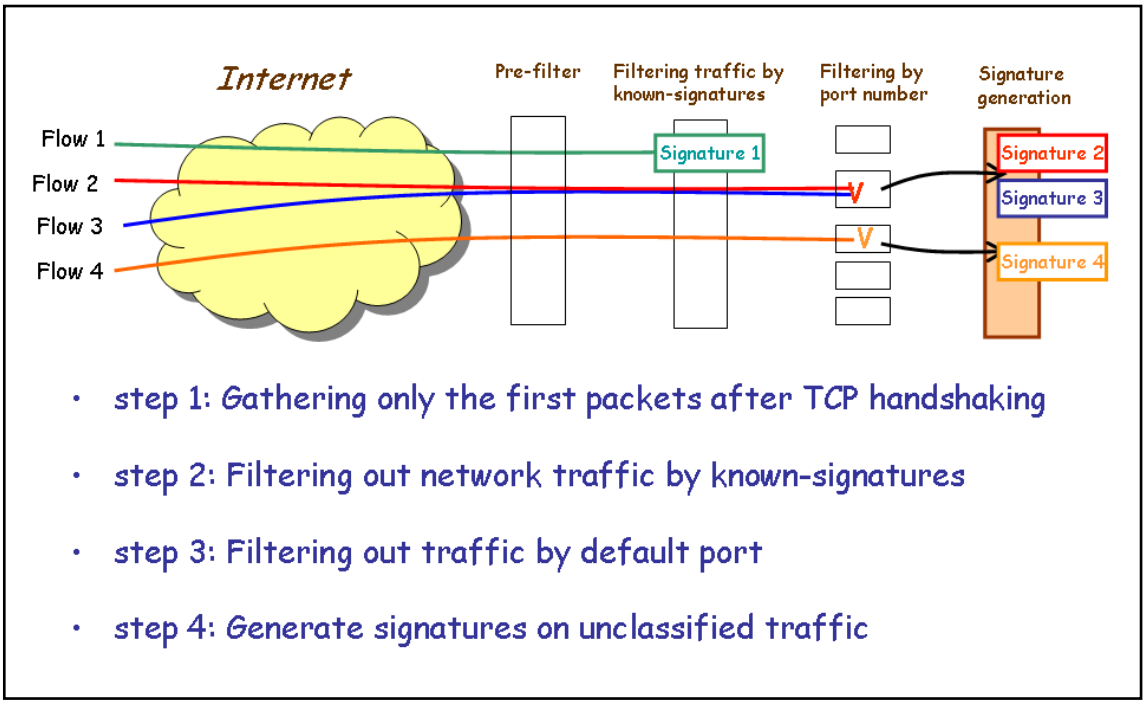

그림 4.5 어플리케이션 분류 방식

또한 패킷 필터는 시그네쳐 자동 생성 방식으로 생성된 시그네쳐를 이용하여 남은 트래픽에서 시그네쳐로 분류될 수 있는 트래픽을 필터링하여, 이 후의 시그네쳐 생성에서의 프로세싱 해야 하는 패킷의 양을 줄이고 하나의 시그네쳐에 속하는 패킷들을 제거함으로서 트래픽의 순수성을 전단계보다 개선시켜준다. 그림 4.6을 보면, packet filter이후에 포트 별로 트래픽 데이터가 분류되어 있을 때, 세 개의 어플리케이션이 존재하는 경우에 대해서 고려하여 보자. 세 개의 어플리케이션들이 동적 포트 할당을 사용하여 통신을 하지만, 디폴트 포트를 각각 포트 A', 포트 B', 포트 C'을 갖는다고 하자.

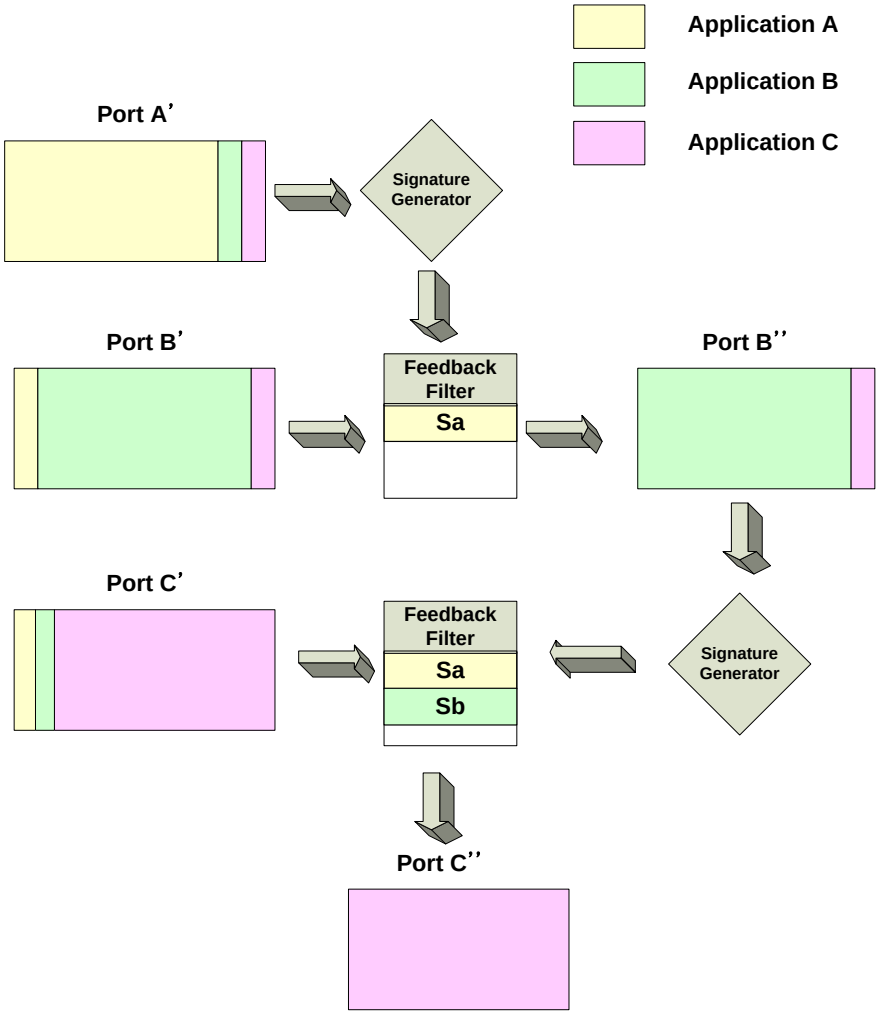

그림 4.6 트래픽 데이터를 packet filter로 필터링 한 후 3개의 어플리케이션 존재하는 경우, packet filter 필터 동작 순서

이러한 경우 포트 A'에는 비교적 어플리케이션 A의 트래픽이 많고 시그네쳐 자동 생성기로 이 트래픽에서 시그네쳐를 생성하면 시그네쳐 Sa가 생성되었다고 하자. 생성된 시그네쳐를 패킷 필터로 피드백 한 후 포트 B'에 대해서 시그네쳐 생성을 하기 전에, 포트 B'의 트래픽을 업데이트 된

피드백 필터로 필터링 한다. 어플리케이션 A에 대한 시그네쳐 Sa가 피드백 필터에 존재하기 때문에, 어플리케이션 A의 트래픽은 포트 B'에서 제거되어 두 개의 어플리케이션에 대한 트래픽만 포트 B''이라는 트래픽 데이터로 남게 된다. 이 트래픽에 대해서 시그네쳐 생성을 다시 하면 Sb라는 시그네쳐를 얻게 되고 이를 다시 피드백 필터에 피드백 해준다. 다시 같은 과정을 반복하면 포트 C'의 트래픽으로부터 어플리케이션 A, B에 대한 트래픽을 제거한 순수 어플리케이션 C에 대한 트래픽만 남게 되어 어플리케이션 C에 대한 스그네쳐를 생성할 수 있게 된다. 피드백 필터로 시그네쳐가 생성 될 때 마다 피드백 하여 트래픽 데이터를 다시 필터링 하여 프로세싱 해야 할 패킷 수도 줄이고 정확도도 높일 수 있다.

3) Signature Generator

(1)절 디자인에서 패턴 클러스터링이 있었지만 패턴 비교 시에 원래 트래픽 데이터를 다시 조사하게 되는데, signature generator에서도 패턴들 비교 시에 트래픽 데이터를 찾아보는 부분이 존재하여 performance측면에서 두 모듈을 합쳐서 구현하였다.

Chapter 5. Evaluation

본 연구에서 사용한 실험 데이터의 수집 기간, 데이터 양은 <표1>와 같다.

Trace	날짜	기간	방향	바이트	패킷	플로우
1	2005/8/26	2.5 hour	Tx	595,746,325,125	593,144,810	10,104,298
2	2005/11/7	2 hour	Rx	440,563,168,752	811,014,070	19,826,314
3	2005/11/7	2 hour	Tx	510,850,712,748	508,619,922	8,664,406

표 5.1 실험 데이터

수집 데이터의 방향성에 따른 어플리케이션 분류 정도 및 시그네쳐 생성을 파악하기 위해서 양방향 데이터를 수집하였다. 페이로드(payload)를 포함하는 연속되는 데이터 수집은 수집 장비 저장장치의 한계성 때문에, 최대 2시간에서 2시간 30분 동안만 수집할 수 있었다. 2장 내용 기반 트래픽 분류 방식 개괄에서 설명하였듯이, 이 트래픽 파일 1, 2, 3 각각에 대하여 TCP 핸드쉐이킹(handshaking)이 후 첫 번째 패킷들만 모아서 시그네쳐 생성을 고려하였다. 트래픽 파일 1, 2, 3에 대해 TCP 핸드쉐이킹 후 첫 번째 패킷들만을 수집할 경우 트래픽들의 양은 <표2>과 같이 바뀌게 된다.

Trace #	바이트	패킷
1	1,786,940,728	3,453,810
2	3,598,344,512	6,202,446
3	1,532,296,392	3,087,510

표 5.2 수집된 데이터 트래픽

각 트래픽 데이터는 TCP핸드쉐이킹 후 첫 번째 패킷들만을 수집함으로써, 처음 모은 트래픽의 평균 1%정도로 크기가 줄어들게 된다. 앞으로의 논의에서 트래픽 1, 2, 3은 RAW 트래픽에서 핸드쉐이킹 후 첫 번째 패킷만을 모은 <표2>의 데이터를 지칭하도록 하겠다. 트래픽 1, 2, 3을 알려진 시그네쳐 <표3>을 기반으로 분류하면 <그림 5.1>와 같은 분포를 이루고 있다. 실험 데이터는 2절에서 정확성을 평가하기 위해 순수 어플리케이션 트래픽을 얻기 위해, 논문에서 시그네쳐 정확성을 검증했거나 이미 알려진 시그네쳐가 있는 어플리케이션으로 4가지를 선택하였다.

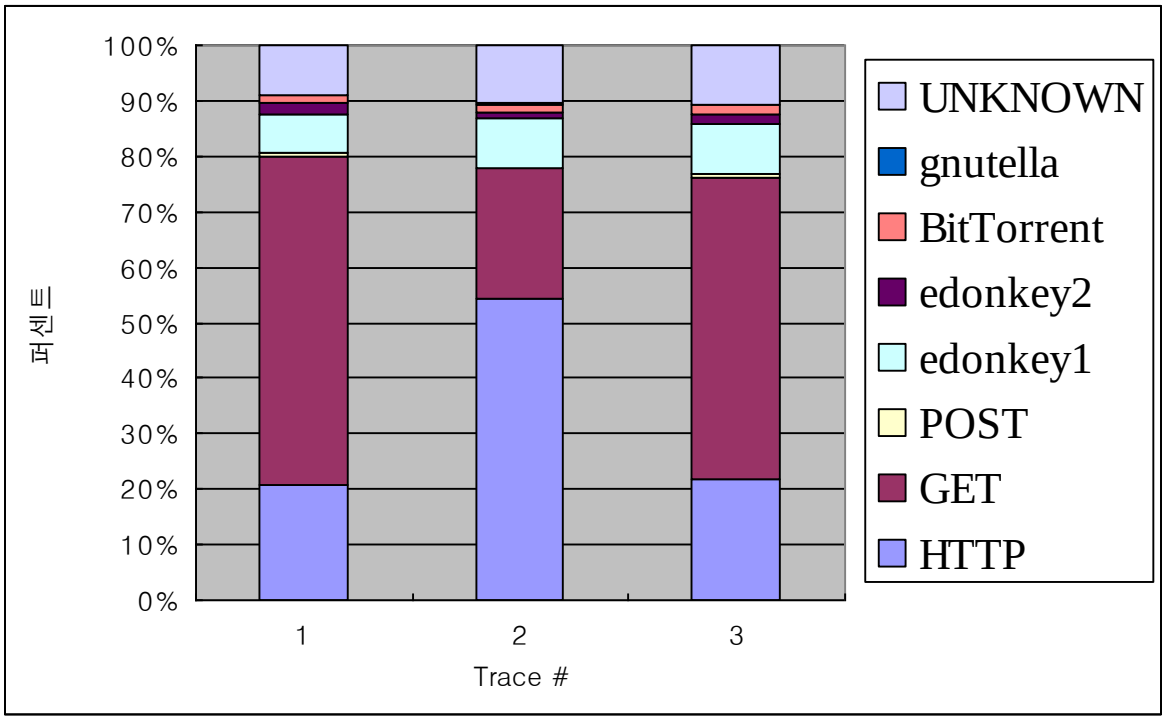

그림 5.1 Trace별 어플리케이션 분류

본 연구에서 제안한 시그네쳐 생성 방식의 정확도와 비교우위를 검증하기 위하여 2, 3, 4절에서는 특정 데이터에 대한 시그네쳐 생성 및 그 정확도를 고찰해 보고자 한다.

2절에서는 순수한 어플리케이션 트래픽에서 어플리케이션 시그네쳐 자동 생성기를 통해 생성된 시그네쳐의 정확성을 분석해 보았다.

3절에서는 생성된 시그네쳐가 시간과 트래픽 방향성에 얼마나 견고한가를 분석하고, 4절에서는 알려진 시그네쳐를 사용하여 트래픽 데이터를 분류한 후 분류되지 않은 트래픽 데이터에서 시그네쳐가 아직 알려지지 않은 어플리케이션 시그네쳐를 생성해 보고 그 결과를 분석해 보았다.

(1) 생성된 시그네쳐의 정확성

1) 정확성 평가 기준

어플리케이션 시그네쳐란 한 어플리케이션에서 반복되는 byte sequence로, 그 시그네쳐가 얼마나 정확하게 어플리케이션을 분류하는지 그리고 시간, 패킷 통신간의 방향성에 얼마나 영향을 받지 않고 견고한 지가 중요하다. 먼저, 어플리케이션 시그네쳐 자동 생성기에 의해 생성된 시그네쳐의 정확성을 평가하기 위해 두 가지 방법을 사용한다. 하나는 어플리케이션 A의 시그네쳐 Sa를 생성한 후, Sa를 사용하여 전체 데이터 트래픽을 분류했을 때, 분류된 트래픽 중 어플리케이션 A가 아닌 다른 어플리케이션 트래픽을 평가하는False Positive(FP)이다. 또 다른 하나는 어플리케이션 A트래픽에서 시그네쳐 Sa에 의해 분류되지 못한 트래픽 양을 평가하는 False Negative(FN)이다.

어플리케이션 A에 대해, n은 총 어플리케이션 A 트래픽 양 이고, m은 생성된 시그네쳐 Sa로 분류된 어플리케이션 A 트래픽 양, t 는 Sa로 분류된 트래픽 중 어플리케이션 A가 아닌 다른 어플리케이션들의 트래픽 양 이라고 하면, FP = t /n 이고, FN = (n-m) / n이다. FP와 FN의 값이 작을수록, 어플리케이션 시그네쳐가 더 정확한 것이다. 시그네쳐 정확성을 평가할 어플리케이션은HTTP, BitTorrent, eDonkey, Gnutella 어플리케이션으로, FP는 시그네쳐 어플리케이션 자동 생성기로 만들어진 각각의 시그네쳐들을 가지고 전체 트래픽에서 각 어플리케이션 별로 분류된 트래픽을 직접 눈으로 확인하였고, FN은 알려진 시그네쳐를 사용하여 표2의 데이터 트래픽에서 순수한 어플리케이션 트래픽만 모은 후, 어플리케이션 시그네쳐 자동 생성기로 생성된 시그네쳐로 분류하지 못한 트래픽 양으로 값을 구한다.

Protocol	Signature		
	Offset	Size	Real value
BitTorrent	0	20	Ox13BitTorrent protocol
HTTP	0	4	HTTP
		4	GET0x20
		4	POST
GNUTELLA	0	8	GNUTELLA
eDonkey	0	5	0xE3 0x6C 0x00 0x00 0x00
		5	0xE3 0x68 0x00 0x00 0x00
		5	0xE3 0x42 0x00 0x00 0x00
		…	…
	0	5	0xC5 0x3F 0x00 0x00 0x00
		5	0xC5 0x22 0x00 0x00 0x00
		5	0xC5 0x47 0x00 0x00 0x00
		…	…

표 5.3 어플리케이션 시그네쳐 자동 생성기로 생성된 시그네쳐

(ㄱ) False Positives

False positive는 어플리케이션 A의 시그네쳐 Sa 로 분류된 트래픽 중 어플리케이션 A가 아닌 다른 어플리케이션 트래픽 양을 말한다. 시그네쳐로 분류된 트래픽을 직접 눈으로 확인하여 조사하였고, 어플리케이션을 분류하기 위해서 사용된 시그네쳐는 어플리케이션 시그네쳐 자동 생성기에 의해 생성된 것이며, 표4에 정리 되어 있다.

BitTorrent 어플리케이션을 " 0x13BitTorrent Protocol" 시그네쳐로

분류한 결과 DataSet1에서 총 47,182개의 패킷이 분류되었고 offset 0위치부터 “ 0x13BitTorrent Protocol”이라는 시그네쳐가 있는 BitTorrent 어플리케이션으로만 이루어져 있어서 False positive는 0이다. HTTP는 HTTP, GET0x20, POST라는 종류의 세 가지 시그네쳐가 생성되었고, 전체 HTTP 어플리케이션은 3,453,810 개의 패킷으로 이루어진 트래픽 중 2,754,351개가 분류되었다. 각각의 경우, 724013개, 2071450개, 28888개의 패킷들로 분류되었으면, 분류된 패킷에는 HTTP가 아닌 다른 어플리케이션은 존재 하지 않았다. ” GNUTELLA”라는 생성된 시그네쳐로 분류한 결과 766개의 패킷만 분류되었으며 여기에도 다른 어플리케이션은 포함되어 있지 않았다. eDonkey 어플리케이션은 1개 이상의 시그네쳐를 가지고 있는 어플리케이션으로, 표 5.3에 S4는 “ 0xe3 0x6C 0x00 0x00 0x00”,

Application		**Dataset 1**		**Dataset 2**		**Dataset 3**	
		Total(pkt)	**FP**(%)	**Total**(pkt)	**FP**(%)	**Total**(pkt)	**FP**(%)
BitTorrent		47,182	0	103,809	0	54,914	0
HTTP	S1	724,013	0	342,521	0	684,644	0
	S2	2,071,450	0	1,472,522	0	1,710,488	0
	S3	28,888	0	1,279	0	18,927	0
GNUTELLA		766	0	2,267	0	444	0
eDonkey	S4	243,605	0	575,493	0.0103	286,631	0
	S5	67,995	3.6209	54,995	0.0012	57,151	4.1277

표 5.4 어플리케이션 False Positive

“ 0xe3 0x68 0x00 0x00 0x00”, “ 0xe3 0x42 0x00 0x00 0x00” 등의 “ 0xe3”가 offset 0위치에 나오는 서로 같은 패킷에서 나오지 않는 패턴들로

구성되어 있다. Application signature 자동 생성기에 의해서 가장 occurrence가 많은 패킷은 “ 0xe3”이나 패턴이 1byte면 signature가 되기에 부족하다고 판단하고 그 패턴을 포함시키는 다른 패턴들을 “ 0xe3” occurrence를 만족시킬 때까지 시그네쳐로 추가시킨다. 1byte가 signature가 되기 부족하다고 판단한 이유는, 1 byte pattern이 실제로 많이 나왔을 수도 있지만, 우연히 다른 패킷에도 그 1byte pattern이 많이 나올 수도 있기 때문에, 그래서 더 정확한 signature를 생성하기 위해 1byte pattern이 signature가 될 수 있는지 여부를 판단하였다. 실제 생성된 시그네쳐로 데이터 트래픽을 분류한 결과 Dataset1과 Dataset3에서는 0을 Dataset2에서는 0.0103으로 낮은 false positive를 보이고 있다.

또 다른 시그네쳐는 “ 0xC5”로, “ 0xC5”에 대해서도 application signature 자동 생성기에서 전체 패킷을 다 분류할 때까지 occurrence가 나온 패턴을 보면서 시그네쳐를 추가시키는데, “ 0xC5”가 가장 많은 occurrence를 보였지만, 1 byte이므로 그것을 포함하는 패턴들을 추가시키고, “ 0xC5”를 추가시켰는데도 아직 전체를 분류할 수 없어, “ ch” “ xX”의 offset이 28, 25인 패턴이 시그네쳐로 추가되었고, 이로 인해 S5로 분류된 패킷들은 다른 것에 비해 높은 false positive를 갖는다. Dataset1에서는 3.6209, Dataset2에서는 0.0012, Dataset3에서는 4.1277의FP를 갖는다. eDonkey application이 FP가 다른 어플리케이션보다 높은 이유는 1byte pattern을 signature로 보기 어렵다고 판단하고 그것을 포함하는 다른 패턴들을 시그네쳐에 추가시킴으로써 패턴 하나하나의 정확성은 높아지나 너무 많은 종류의 signature가 추가될 것이며, 충분한 data training set이 필요하다. 아니면 또 다른 1 byte pattern을 처리하는 방법이 필요하다.

Application signature 자동 생성기에 의해 만들어진 시그네쳐는 max 4.15정도의 FP를 가지며, 1byte signature를 가진 어플리케이션이 아닌 다른 어플리케이션에서는 시그네쳐에 대한 False positive는 0이다. 즉,

생성된 시그네쳐가 한 개의 어플리케이션만을 분류하는데 아주 정확하다는 것이다. FP로 인해, 우리는 우리가 생성한 시그네쳐가 얼마나 정확한지 설명하였다. 표5은 첫 번째 열은 어플리케이션을, 두 번째 열은 Dataset1 중 application signature 자동 생성기를 통해 생성된 시그네쳐로 분류된 전체 트래픽의 양을 패킷으로 표시하였고, 그 중 다른 어플리케이션이 얼마나 있는지 False Positive의 값을 표현하였다.

(ㄴ) False Negatives

False negative는 순수한 어플리케이션 A 트래픽 중 시그네쳐 Sa로 분류하지 못한 트래픽의 양을 말한다. 순수한 어플리케이션 트래픽(분류하고자 하는 어플리케이션만 있는 트래픽)을 구하기 위해서, 이미 논문에서 밝혀지고 정확성을 평가하여 믿을 수 있는 시그네쳐들의 목록을 가지고 데이터 트래픽을 각 어플리케이션 별로 분류한다. 그리고 4.1의 시그네쳐로 얻은 트래픽을 100% 신뢰한다는 가정 하에, 모아 놓은 순수한 어플리케이션 트래픽을 우리가 생성한 시그네쳐로 분류해 봄으로써 실제 트래픽의 얼마를 분류해 내지 못하는 지 평가한다. 표6의 첫 번째 열은 실험에 사용된 어플리케이션, 두 번째 열은 위에서 언급했듯이 논문에서 정확성을 평가 받거나 알려진 시그네쳐를 사용해 얻은 순수한 어플리케이션 트래픽의 양(패킷)이다. 세 번째 열은 False negative 값이다. 실험 결과, eDonkey의 경우를 제외하고 1%도 안 되는 False Negative를 가지고 있다.

실제로 FN이 나온 어플리케이션 트래픽을 살펴 본 결과, Dataset2의 경우, 알려진 시그네쳐로 “ GET”이라는 것을 사용하였는데, 실제로 우리는 “ GET0x20”로, 0x20은 공백문자를 의미한다. “ GET”이라는 이미 알려진 시그네쳐로 분류한 순수한 어플리케이션에는 GETO라는 다른 어플리케이션이 21개 포함이 되어 있어서 False negative가 0.0014% 나타난 것이다. 실제로 HTTP 어플리케이션이 아닌 것을 제외 했을 경우는

FN이 0이 되는 것이다.

eDonkey 어플리케이션은 “ 0xE3 0x6C 0x00 0x00 0x00” 시그네쳐가 상당히 많은 트래픽에서 나타나기 때문에 시그네쳐를 잘 못 생성할 확률이 매우 낮아져 FN이 “ 0xC5” 와 관련된 시그네쳐의 FN보다 낮은 것이다. 어플리케이션 시그네쳐 자동 생성기에 의해 전체 트래픽을 분류하는 시그네쳐들을 생성할 때, “ 0xC5”를 포함하는 패턴들을 생성 시, 실제로 signature는 “ 0xC5”이기 때문에 “ 0xC5” 다음에 다양한 byte sequence가 더 나올 수 있다. 어플리케이션 시그네쳐 자동 생성기에 의해 시그네쳐로 추가되지 못한 “ 0xC5 .. ..” 때문에 실제로 eDonkey application이지만 eDonkey로 분류되지 못한 트래픽이 다른 어플리케이션에 비해 많게 된다. 이를 통해 우리는 1 byte pattern이 signature가 될 수 있으며 이것에 대한 보안을 마련해야 할 것이다. 그러나 다른 논문들의 나온 accuracy보다 훨씬 좋은 accuracy를 보이고 있다.

Application		**Dataset 1**		**Dataset 2**		**Dataset 3**	
		Total(pkt)	**FN**(%)	**Total**(pkt)	**FN**(%)	**Total**(pkt)	**FN**(%)
BitTorrent		47182	0	103,809	0	54,914	0
HTTP	S1	724,013	0	3,432,521	0	684,644	0
	S2	2,071,450	0	1,472,539	0.0014	1,710,488	0
	S3	28,896	0.0277	1279	0	18,931	0.0211
GNUTELLA		766	0	2267	0	444	0
eDonkey	S4	243,629	0.0021	575,434	0.0290	286,700	0.0220
	S5	65,533	0.0015	55,476	0.8670	54,792	0.0018

표 5.5 어플리케이션 별 False Negative

(2) 알려지지 않는 (unknown) 시그네쳐 생성

1)포트별 시그네쳐 생성

2절에서 본 바와 같이 본 연구에서 제안한 시그네쳐 생성 알고리즘은 한 어플리케이션에 대한 순수한 트래픽 데이터를 기반하여 시그네쳐를 생성할 경우 시그네쳐 생성에 높은 정확도를 보이고 있다. 하지만 실제로 한 어플리케이션의 대한 순수한 트래픽을 패시브 한 모니터링 환경에서 분별해서 모으기는 힘든 작업이다. 이러한 패시브 모니터링 환경에서 본 연구에서 제안한 시그네쳐 자동 생성 알고리즘을 적용하기 위해서는 한 개의 어플리케이션에 대한 순수한 트래픽 데이터로 최대한 클리닝을 하여야 한다. 이를 위해 기존에 알려진 어플리케이션 시그네쳐 리스트로 트래픽 데이터를 필터링 한 후 남은 트래픽에 대해서 포트 번호에 따라 트래픽을 나누어 보았다. 각 포트 트래픽에는 알려진 시그네쳐 트래픽을 포함하지 않으면서 포함된 어플리케이션의 수가 처음 bulk하게 모여 있을 때보다 줄어 들기 때문에 어플리케이션 시그네쳐를 생성하기에 이전 보다 나은 환경을 제공하게 된다. 트래픽 데이터에 대해 상위 Top10에 있는 포트들로부터 시그네쳐를 생성하여 각 시그네쳐가 대표 하는 어플리케이션에 대해 조사하여 보았다. Top10 포트들은 443(SSL), 80(HTTP), 1863(MSN), 25(SMTP), 8080(HTTPS), 554(RTSP), 10100(unknown), 995(PoP3 over TLS/SSL), 8008(HTTP alternative), 110(PoP3 version3)등이 였다. 이 중 대표적으로 443, 80, 1863, 25, 10100을 시그네쳐 생성 후 패킷들과 비교해 보았다.

(ㄱ) 443 PORT : SSL

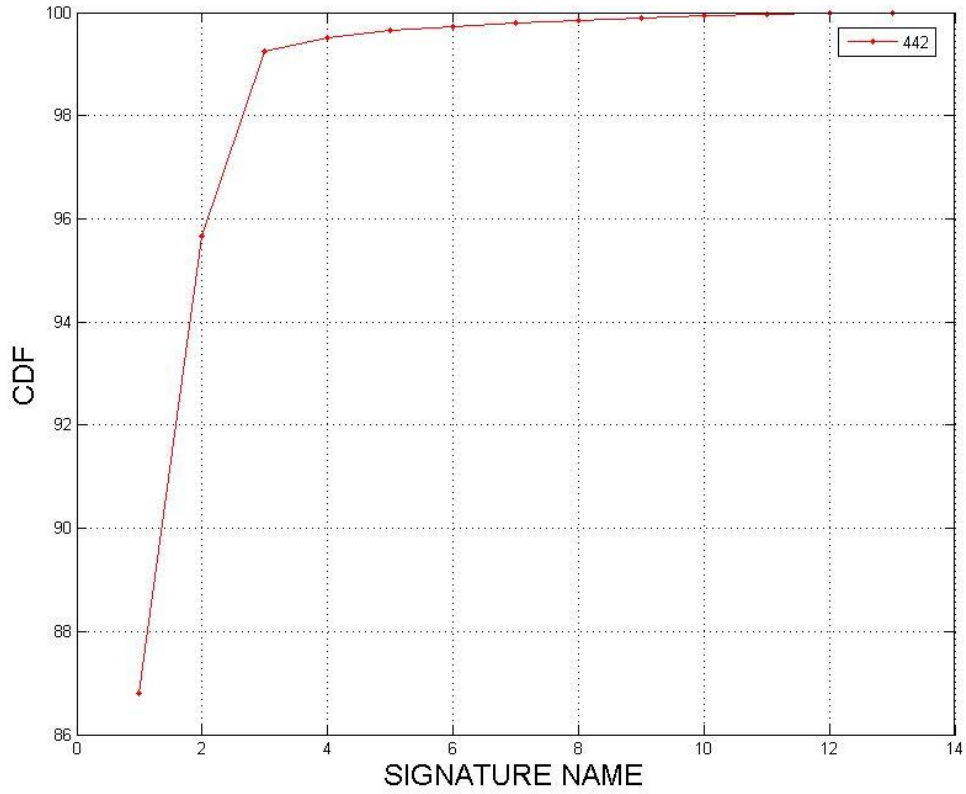

그림 5.2 Port 443 트래픽에서 생성된 시그네쳐의 패킷 coverage CDF

<그림 5.2>에서 보면 443포트에 존재하는 패킷들에 대해 시그네쳐를 생성하여 3개의 시그네쳐를 선택함으로 해서 99%의 패킷들을 대표하는 시그네쳐를 생성해 낼 수 있다. 실제로 트래픽 데이터를 확인한 결과 3가지의 시그네쳐를 가지는 패킷들은 SSL(secure socket layer) 어플리케이션의 패킷들이었다. 12개의 시그네쳐 중 11개의 오프셋이 0에 위치하였다.

(ㄴ) 80 : HTTP

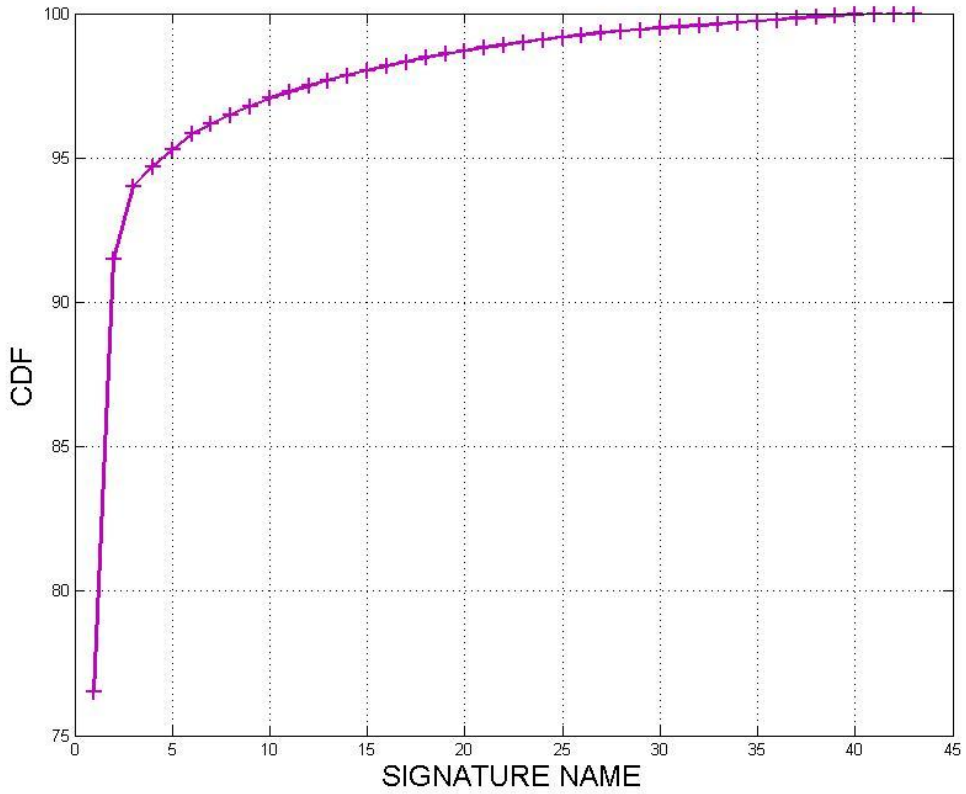

그림 5.3 Port 80 트래픽에서 생성된 시그네쳐의 패킷 coverage CDF

443번 포트와 마찬가지로 3개의 시그네쳐를 선택 시 94%의 패킷들을 대표하는 시그네쳐를 생성해 낼 수 있었다. 시그네쳐 1은 “ HEAD” 문자열로서 HTTP의 리퀘스트 메쏘드 중 하나이다. 시그네쳐 2는 “ PROPFIND” 문자열로 마찬가지로 HTTP의 리퀘스트 메쏘드였다. 시그네쳐 3은 0x022A000160BF 라는 비트 스트림이였으며 어떠한 어플리케이션인지 알 수 없었으나 많은 수의 패킷들에서 발견된 공통적인 메시지였다.

(ㄷ) 1863 : MSN

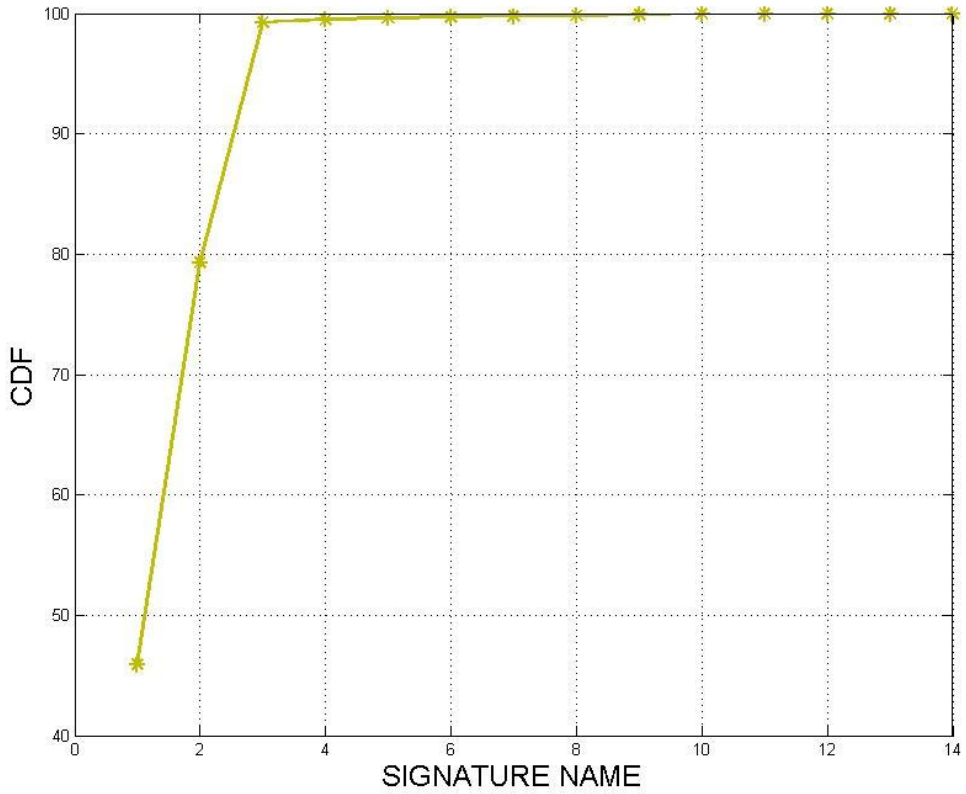

그림 5.4 Port 1863 트래픽에서 생성된 시그네쳐의 패킷 coverage CDF

MSN 메신저 포트로 알려져 있으며 3번째 시그네쳐까지 선택 시 99퍼센트 이상을 대표할 수 있는 시그네쳐를 생성해 낼 수 있다. 3개의 시그네쳐는 오프셋 0의 위치에 “ USR”, “ VER”, “ ANS”의 문자열을 가졌다.

(ㄹ) 25 : SMTP

3개의 시그네쳐를 생성시 99.5퍼센트 이상의 패킷 coverage를 가진다. 3개의 시그네쳐는 오프셋 2의 “ LO”, 오프셋 0의“ QUIT”, 오프셋 0의 “ MAIL F”이였다. 이 중 LO는 SMTP의 메시지 중 HELO와 EHLO의 공통적인 패턴이며 QUIT과 MAIL F 또한 SMTP의 중요 커맨드 중에 하나였다.

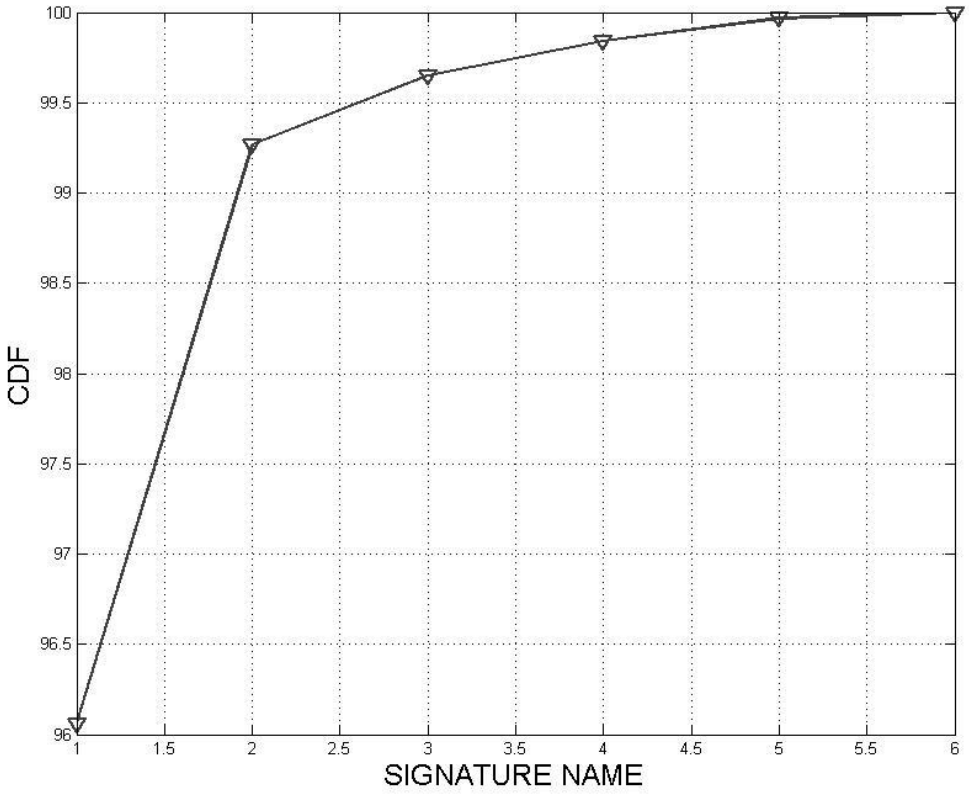

그림 5.5 Port 25 트래픽에서 생성된 시그네쳐의 패킷 coverage CDF

(ㅁ) 19101 : clubbox

2개의 시그네쳐가 생성되었고, 한가지 시그네쳐가 99.9퍼센트를 차지하고 있었다. 오프셋이 0인 " 0x00 00 00 03 14 05 00 00 08 00 00 00 1C 05 00 03"이였다. 어플리케이션을 확인해 본 결과 클럽박스라는 어플리케이션이 동작하고 있었으며 그 어플리케이션의 시그네쳐 임을 확인하였다.

(ㅂ) 5004 : nateon

5004번 포트에 모인 트래픽 데이터를 기반으로 시그네쳐를 생성시 총 3개의 시그네쳐가 생성되었고 그 중 한 개의 시그네쳐로 99.9퍼센트의 패킷들을 대표할 수 있었다. 그 시그네쳐는 오프셋이 0이며 " ATHC 0"라는 문자열이었다. 패킷을 확인한 결과 네이트 온이라는 메신저 어플리케이션의 시그네쳐였다.

디폴트 포트를 가지고 있는 어플리케이션의 경우 지정된 포트로 통신하는 경향이 동적 포트 할당 어플리케이션보다 강하므로 하나의 포트에 모여있는 경향을 보이게 된다. 이러한 포트 트래픽에서 시그네쳐를 생성하면 하나의 어플리케이션이 응집된 경향이 강해서 1~3개 정도의 시그네쳐 만으로 90퍼센트 이상의 패킷들을 대표할 수 있다.

(2) Unknown bulk 데이터에서 시그네쳐 생성

이렇게 포트 별로 시그네쳐를 생성하는 접근 방식이 아닌 필터 되어 여러 어플리케이션이 많이 섞여 있는 bulk한 트래픽 데이터을 기반으로 한 시그네쳐 생성에 대해 고려해 보자. 생성된 시그네쳐의 패킷 coverage count에 대한 CDF는 <그림 11>와 같다. 이 그래프에서 볼 수 있듯이 7~10개 정도의 주요한 시그네쳐가 존재하며 그 시그네쳐 만으로 Unknown 트래픽 데이터의 80퍼센트 정도의 패킷들을 분류해 낼 수 있다. 이 때 시그네쳐를 28개가 90퍼센트의 패킷들에서 공통적으로 나타나고 있다. 이러한 결과는 시그네쳐를 총 30여 개 안팎으로 구하고 있을 때 TCP connection을 기반으로 트래픽을 분류 할 때 전체의 10%정도를 차지하는 Unknown트래픽에 대해서 9%정도까지 분류해 낼 수 있고 트래픽 전체의 1%만이 Unknown트래픽으로 남는 결과를 얻어 낼 수 있다.

<그림 5.7>의 그래프는 이렇게 만들어진 시그네쳐의 offset에 따른 패킷 coverage count에 대한 CDF그래프이다. 각각의 시그네쳐는 패킷 페이로드에서 시작하는 offset위치가 정해져 있다. 또한 각 시그네쳐들이 대표하고 있는 패킷들의 수가 coverage count이다. 오프셋이 같은 시그네쳐들끼리 coverage count를 합산하여 전체에 대해 cumulative하게 그린 그래프이다. 이 그래프에서 보면 알 수 있듯이 생성된 시그네쳐의 85퍼센트의 offset이 0에서 생성되었다. 한 시그네쳐와 coverage하는 패킷이 동일하면서 오프셋과 시그네쳐를 구성하는 바이트 시퀀스(byte sequence)가 다른 시그네쳐가 존재할 수도 있지만, 이 시그네쳐가 사용될 환경 – 게이트웨이나 방화벽에서 필터링 – 에서나 트래픽을 패시브 하게 모으는 환경 – 페이로드까지 수집하더라도 처음 16바이트나 32바이트까지만 수집- 을 고려하여 오프셋이 0인 시그네쳐를 생성하는 것이 유리하다. 이 그래프 분포에서도 볼 수 있듯이 시그네쳐가 오프셋 0에 근접경향이 있다. 이는 시그네쳐의 특성이 어플리케이션에서 반복적으로 나타나는 메시지이며 이러한 메시지들 중에서 많은 수가 프로토콜 정의에

의한 것이 많고 이러한 프로토콜 정의는 메시지 초기에 들어있어서 프로토콜에 벗어난 통신을 미연에 방지한다. 이러한 이유로 시그네쳐는 오프셋 0에 가깝게 존재하는 경향이 있다.

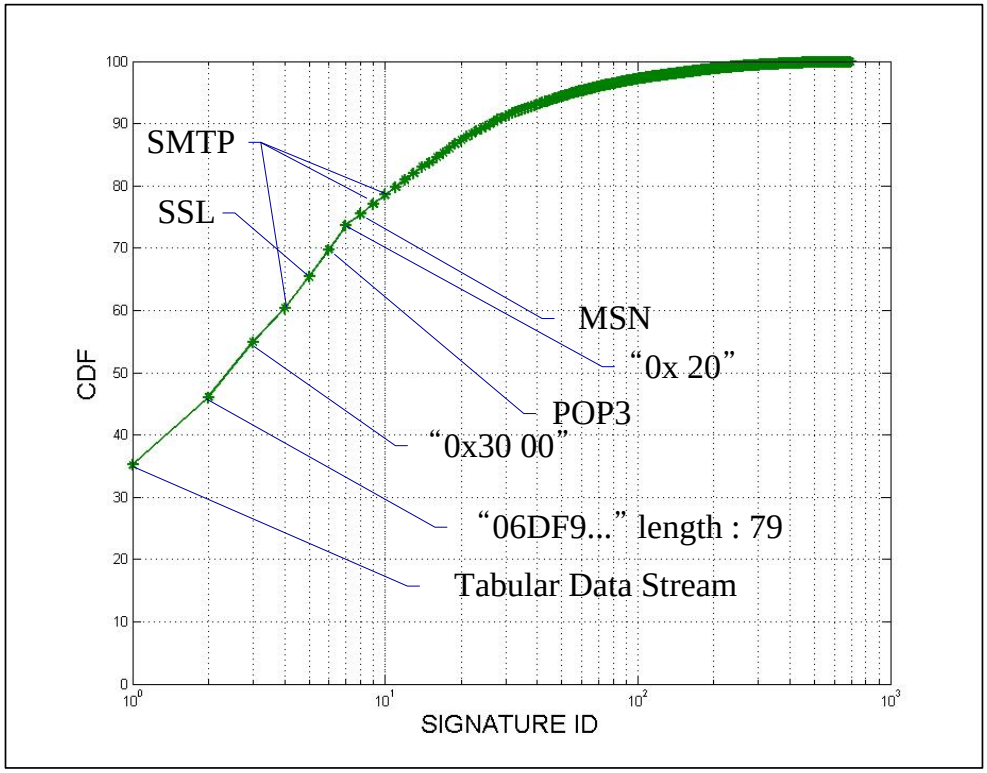

그림 5.6 Unknown bulk 트래픽에서 생성된 시그네쳐의 패킷 coverage CDF

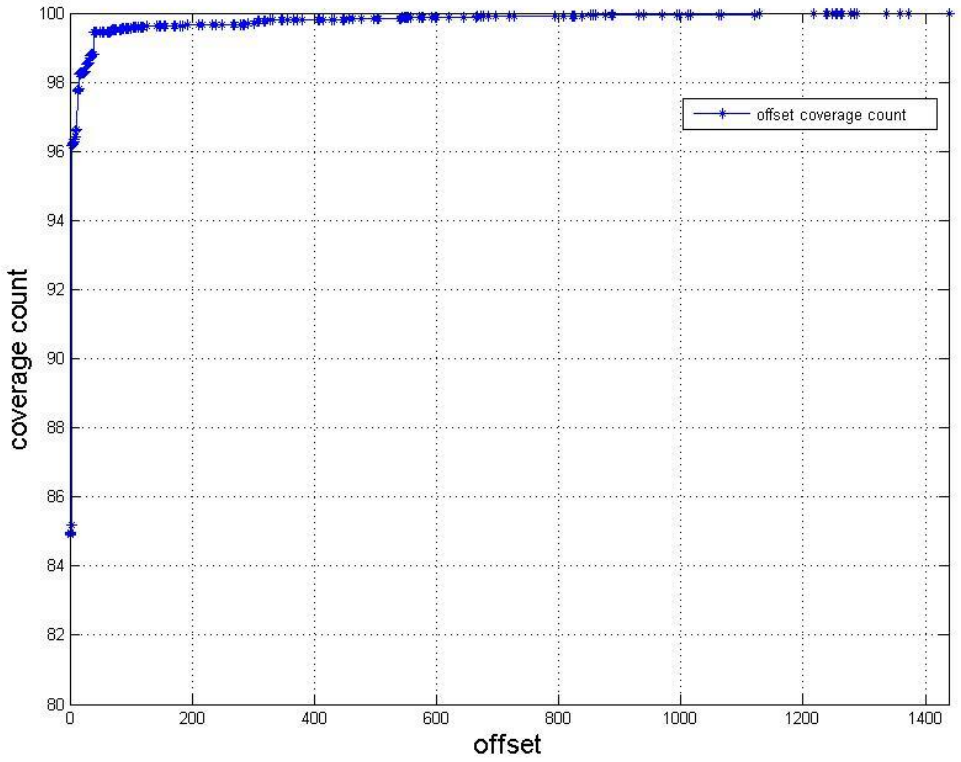

그림 5.7 Unknown bulk 트래픽에서 생성된 시그네쳐의 offset에 따른 패킷 coverage CDF

Chapter 6. Conclusions and Future Works

(1) Summary

본 연구에서는 어플리케이션 별 인터넷 트래픽을 분류를 위하여 시그네쳐 자동으로 생성하여 분류하는 방식을 제안하였다. 이 방식은 인터넷 트래픽을 기존에 알려진 시그네쳐를 통하여 분류한 후 남은 트래픽에 대하여 각 포트 별로 모인 트래픽에 대하여 어플리케이션 시그네쳐 자동 생성 알고리즘(automatic application signature algorithm)을 적용하여 새로운, 알려지지 않은 어플리케이션 시그네쳐(application signature)를 오프라인으로 생성하였다. 이렇게 생성된 시그네쳐는 패킷 필터에 피드백 되며 이 후 나머지 트래픽에서의 시그네쳐 생성 전에 필터로서 동작하게 된다.

이렇게 생성된 시그네쳐는 시그네쳐를 생성시킨 트래픽 데이터에 기반을 두고 있으며 이 트래픽 데이터에 존재하는 패킷들 사이에 공통적으로 존재하는 시그네쳐를 생성하여 트래픽 데이터 내의 모든 패킷들을 생성된 시그네쳐만으로 판별할 수 있었다.

시그네쳐 자동 생성 알고리즘은 특정 어플리케이션의 패킷만 존재하지 않고 다른 어플리케이션의 패킷들이 존재하더라도 그 트래픽 데이터를 기반하여 시그네쳐를 생성하였으며 정형화 표현(regular expression)에서도 하나의 어플리케이션이 다수의 시그네쳐를 가질 경우에도 각 시그네쳐들의 생성이 가능하였다.

(2) Contribution

본 연구에 있어서 이 분야 연구에 기여한 바는 다음과 같다.

오프라인 트래픽 데이터 기반 시그네쳐 자동 생성기 개발

- 알려지지 않은 시그네쳐 생성(msn 메신저, nateon 메신저)

→ 생성된 시그네쳐를 통해 전체 TCP 트래픽의 98%를 분류해 낼 수 있었다.

(3) Future work

1) 어플리케이션 별 시그네쳐 구분

4.4절에서 Unknown traffic을 어플리케이션 시그네쳐 자동 생성기로 생성된 시그네쳐 별로 분류하는 것을 보았다. 한 개의 큰 unknown 트래픽을 여러 개의 세분화된 같은 byte sequence를 가지는 트래픽으로 분류하였지만, 각각 생성된 시그네쳐 별로 분류된 트래픽을 어떤 시그네쳐들이 한 어플리케이션을 구성하는지 파악하고, Unknown traffic을 signature 별로 분류하는 것이 아닌, 어플리케이션 별로 분류하는 방법에 대한 연구를 할 수 있다.

2)variable offset 시그네쳐 생성

현재 Application signature 자동 생성기는 fixed offset 기반으로, 주어진 offset에 나오는 byte sequence를 비교하여 패킷을 분류하고, 어플리케이션을 분류하는 시그네쳐를 생성하고 있다. 공통된 byte sequence가 정해진 offset 위치가 아니라 variable offset을 사용할 때, 시그네쳐를 만드는 방법에 대해 연구가 필요하다.

요약문

어플리케이션 시그네쳐 자동 생성에 관한 연구

네트워크 관리자들은 자신들이 관리하고 있는 네트워크의 현재 상태를 알고자 한다. 이를 위하여 그들은 네트워크 트래픽을 어플리케이션 종류별로 분류한다. 네트워크 트래픽을 어플리케이션 별로 분류하는 방식에는 어플리케이션이 사용하는 포트를 기반으로 분류하는 포트 기반 분류 방식이 있다. 이 방식은 정적 포트 부여 방식을 사용하는 서버 기반 어플리케이션들(HTTP, FTP, SMTP)에 적용될 수 있다. 하지만 동적 포트를 사용하는 P2P, 게임, 스트리밍 어플리케이션의 사용이 늘어남에 따라 포트 기반 방식으로는 분류의 정확도가 떨어지는 한계성을 가지게 되었다. 이를 보안하기 위해서 어플리케이션이 통신할 때 패킷 내에 공통적으로 나타나는 시그네쳐를 기반으로 분류하는 시그네쳐 분류 방식이 제안되었다. 하지만 이 방식은 자동화 되어 있지 않아 인터넷에 존재하는 수천의 어플리케이션의 시그네쳐를 생성하는 데 있어서 비효율적이다. 본 논문에서는 어플리케이션 시그네쳐를 자동으로 생성하기 위한 이슈들을 살펴보고 어플리케이션 시그네쳐 자동생성기를 구현하였다. KAIST 캠퍼스 DAGMON장비를 통해서 네트워크 트래픽을 수집하여 이를 오프라인으로 분석하여, 이미 알려져 있는 시그네쳐들(BitTorrent, edonkey, gnutella, HTTP)뿐 아니라 알려지지 않은 시그네쳐들(MSN 메신저, nateon 메신저)을 생성해 내었다. 실제의 네트워크 트래픽을 기반으로 이를 평가해 보았으며, 생성된 시그네쳐들은 트래픽 분류에 있어서 낮은 폴스 포지티브(4퍼센트 이하)와 낮은 폴스 네가티브(0.8퍼센트 이하)를 보여주었다.

Reference

[1] Internet Assigned Numbers Authority (IANA).

http://www.iana.org/assignments/port-numbers

[2] Subhabrata Sen, Oliver Spatcheck and Dongmei Wang, “ Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures”, World Wide Web Conference, May 2004

[3] CISCO SYSTEMS. Network-Based Application Recognition.

http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newf%t/122t/122t8/dtnbarad.htm.

[4] J. Cleary, S. Donnelly, I. Graham, A. McGregor and M. Pearson, “Design Principles for Accurate Passive Measurement”, in Proc. of PAM Workshop, Hamilton, New Zealand, April 2000.

[5] Andrew W. Moore and Konstantina Papagiannaki, "Toward the Accurate Identification of Network Applications," Passive & Active Measurement Workshop, Boston, USA, March 2005

[6] Subhabrata Sen, Oliver Spatcheck and Dongmei Wang, "Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures", World Wide Web Conference, May 2004

[7] Sumeet Sigh, Cristian Estan, George Varghese and Stefan Savage, "Automated Worm Fingerprinting", ACM Symposium on Operating System Design and Implementation(OSDI), December 2004

[8] Hyang-Ah Kim and Brad Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection", USENIX Security Symposium, 2004

[9] Patrick Haffner, Subhabrata Sen, Oliver Spatcheck and Dongmei Wang, "ACAS:

Automated Construction of Application Signatures", SIGCOMM'05 Workshops, August 2005

[10] Thomas Karagiannis, Andre Broido, Micheal Faloutsos, Kc claffy, “Transport Layer Identification of P2P Traffic”, IMC04 Workshops, October 2004

[11] Extensible Record Format (ERF)

http://www.endance.com/support/EndaceRecordFormat.pdf

[12] Andrew W. Moore and Konstantina Papagiannaki, “ Toward the Accurate Identification of Network Applications”, Passive & Active Measurement Workshop, March 2005

[13] Cooperative Association for Internet Data Analysis (CAIDA)

http://www.caida.org

감 사 의 글

KAIST에 입학한지 어언 9년이 지났습니다. 저의 20대의 대부분을 KAIST에 서 보내고 떠나가게 되어 감회가 새롭습니다. 학부시절에는 대학에서 뭘 배워야 할지도 모르고 그저 수업만 듣고 친구들과 어울리곤 했던 것 같습니다. 대학원에 와서는 이제까지 배워온 것들을 통해서 새로운 배움을 얻을 수 있었지만 만족스럽게 하지는 못한 것 같습니다. 이제 다시 대학교, 대학원시절을 다시 다니라고 하면 잘 할 자신이 생기는데 사회로 떠나야 할 시간이 된 것 같습니다. 전혀 지금과는 다른 사회라는 곳으로 나아가려니 호기심 반 두려움 반입니다.

이 논문이 완성되기까지 아낌없는 관심과 조언, 세심한 지도로 저를 이끌어 주신 문수복 교수님께 진심으로 감사 드립니다. 석사 1년의 시간 동안 연구자의 갈 길과 자세를 보여주신 전길남 교수님께도 감사의 말씀을 전하고 싶습니다. 아울러 본 논문의 논문 심사를 맡아 주시고 좋은 조언과 의견을 주신 황규영 교수님과 최기선 교수님께도 감사 드립니다.

제 석사 기간 동안 도움 주신 모든 분들께 감사 드립니다. 이미 졸업하였지만 석사 기간 내내 많은 도움과 조언 준 승준이에게 고맙다는 말을 하고 싶습니다. 본 논문을 같이 연구하면서 힘든데도 불평하나 하지 않으면서 따라와 주고 이끌어 준 태희에게 정말 고맙다는 말 전하고 싶습니다. 비슷한 분야 연구하면서 많은 의견 주시고 연구실 일도 많이 도와주신 종건씨 에게도 감사 드립니다. 처음 연구실 생겼을 때 헤매는 우리를 이끌어 준 우리 랩 랩짱 미영이, 외국에서 열심히 공부하고 있는 민경이형, 항상 썰렁한 유머로 우릴 기쁘게 해줬던 동기, 연구도 열심히 하면서 잘 놀아주기도 한 승엽이, 항상 듬직하게 도와주셨던 두영씨, 얼마 함께 없었지만 정이 든 또앙, 프레드릭, 니콜라에게도 감사 드립니다. 석사 1년의 기간이었지만 함께 있으면서 많은 도움을 준 전길남 교수님 연구실 학생들에게도 감사 드립니다. 이미 졸업했지만 함께 입학 함께 해서 동고동락한 대원씨, 형준이, son, 랩 선배로서 잘 이끌어 준 준복형, 썰렁한 유머의 원조로서 우리를 기쁘게 해 준 유성이, 항상 재미난 일거리를 만들어 심심치 않게 해 준 덕희, 항상 듬직한 모습의 상호, 열심히 연구하는 모범의 모습을 보여준 정호씨 모두에게 감사 드립니다.

카이스트에 있는 내내 함께 있어준 고등학교 동기들, 창효, 재완, 대길, 화신, 수호, 문식, 병재, 명식, 창수, 세완, 흥길, 권수, 호, 상은, 종호에게도 고맙다는 말 전하고 싶네요. 항상 마음의 안식처가 되어 준 고향 친구들 상훈, 경주, 정우, 그리고 어린 시절 많은 도움 주신 김광열 선생님, 변희 선생님께도 감사 드립니다.

마지막으로 연구하는 동안 물심양면으로 도와 주신 가족들과 친척분들께도 감사 드립니다. 항상 절 사랑해 주신 엄마, 아빠 그리고 동생 경아에게 이 논문을 바칩니다.

이 력 서

이　　름 : 김 태 호

생 년 월 일 : 1979년 6월 17일

출　생　지 : 전라남도

주　　소 : 전남 고흥군 도양읍 봉암리

E-mail 주 소 : taestone@gmail.com

학　　력

1997. 3. ~ 2002. 8.	한국과학기술원 전산학과 (B.S.)
2003. 3. ~ 2006. 2.	한국과학기술원 전산학과 (M.S.)
	한국과학기술원 정보통신학제전공