情報保護學會論文誌
第19卷 第4號, 2009. 8

고속 정적 분석 방법을 이용한 폴리모픽 웜 탐지

오 진 태,[1†] 김 대 원,[1] 김 익 균,[1] 장 종 수,[1] 전 용 희[2‡]
[1]한국전자통신연구원, [2]대구가톨릭대학교

Polymorphic Worm Detection Using A Fast Static Analysis Approach

Jin-tae Oh,[1†] Dae-won Kim,[1] Ik-kyun Kim,[1] Jong-soo Jang,[1] Yong-hee Jeon[2‡]
[1]ETRI, [2]Catholic Univ. of Daegu

요 약

통신망을 통하여 자동적으로 확산되는 악성 프로그램인 웜에 대응하기 위하여, 웜 관련 패킷들을 분석하여 시그니처를 생성하여 웜을 탐지하는 방법들이 많이 사용되고 있다. 그러나 이런 시그니처-기반 탐지 기법을 회피하기 위하여, 변형된 폴리모픽 형태의 공격 코드 사용이 점차 증가하고 있다. 본 논문에서는 폴리모픽 공격 코드의 복호화 루틴을 탐지하기 위한 새로운 정적 분석 방법을 제안한다. 제안된 방법은 통신망 플로에서 폴리모픽 공격 코드에 함께 포함되어 암호화된 원본 코드를 복호화하는 역할을 수행하는 코드 루틴을 탐지한다. 실험 결과를 통하여 제안된 방법이 정적 분석 방지 기법들이 사용된 폴리모픽 코드들도 탐지할 수 있는 것을 보여준다. 또한 처리 성능에서 에뮬레이션 기반 분석 방법보다 효율적인 것으로 나타났다.

ABSTRACT

In order to respond against worms which are malicious programs automatically spreading across communication networks, worm detection approach by generating signatures resulting from analyzing worm-related packets is being mostly used. However, to avoid such signature-based detection techniques, usage of exploits employing mutated polymorphic types are becoming more prevalent. In this paper, we propose a novel static analysis approach for detecting the decryption routine of polymorphic exploit code. Our approach detects a code routine for performing the decryption of the encrypted original code which are contained with the polymorphic exploit code within the network flows. The experiment results show that our approach can detect polymorphic exploit codes in which the static analysis resistant techniques are used. It is also revealed that our approach is more efficient than the emulation-based approach in the processing performance.

Keywords: Exploit, shellcode, polymorphism, malware detection, static analysis

I. 서 론

인터넷의 급격한 성장으로 이를 대상으로 하는 공격들로 인한 피해 규모가 기하급수적으로 증가하고 있고, 공격의 형태 또한 점점 복잡해지고 다양화되고 있다. 이러한 공격들을 탐지하기 위한 기술들이 지속적으로 발전하고 있지만, 이를 회피하기 위한 공격 방법들도 병행하여 꾸준히 진화되고 있다. 통신망에 적용되고 있는 대부분의 네트워크 기반 공격 탐지 시스템들은 Snort[1]와 같이 공격 시그니처(signature)를 탑재해야 한다는 근본적인 한계를 가지고 있다. 따라서 폴리모픽(polymorphic) 형태와 메타모픽(metamorphic) 형태와 같이 특정 시그니처를 생성하기 힘든 공격들에 대해서는 취약할 수 밖에 없다. 특히 폴리모픽 형태의 경우에는 공격 코드 제작에 대한 특별한 지식이 없어도 쉽고 다양한 형태의 공격 코드를 만들 수 있는 많은 자동화된 엔진들[2-5]이 존재하기 때문에 폴리모픽 공격에 대한 연구가 더욱 활

접수일(2009년 2월 4일), 수정일(2009년 5월 27일), 게재확정일(2009년 6월 22일)
† 주저자, showme@etri.re.kr
‡ 교신저자, yhjeon@cu.ac.kr

발히 진행될 필요가 있다. 실제로 폴리모픽 방법은 대상 호스트(Target Host)의 취약점을 악용하여 제어권을 얻기 위해, 공격 호스트가 전송하는 쉘 코드(Shell Code)를 특정 키 값을 이용하여 암호화하고, 그 쉘 코드 내에 복호화 코드를 같이 삽입하게 된다. 단순한 폴리모픽 엔진들은 탐지가 쉬운 암호화 코드를 생성하지만, 최신 엔진[2,3,5,6] 들은 메타모픽 방법을 결합하여 사용하기 때문에 복호화 코드를 발견하는 것을 더욱 어렵게 만들고 있다.

폴리모픽 공격을 탐지하기 위한 여러 분야들 중의 한 가지로서, 패킷의 페이로드에 대한 바이너리 분석 방법들이 연구되어 왔다. 이런 시도들로서 다양한 정적 분석(Static Analysis) 방법들[7-9]이 연구되었지만 역어셈블 방지 기법(Disassembly Thwarting Technique), 자기 수정 코드 기법(Self-modifying Code Technique)과 같은 정적 분석 방지 기법(Static Analysis Resistant Technique)[10]이 폴리모픽 공격 코드에 새롭게 적용되어감에 따라 탐지에 점점 어려움이 따르게 되었다. 정적 분석 방지 기법들의 영향을 받지 않기 위해 최근에는 패킷의 페이로드를 직접 실행해 보는 동적 분석 방법에 대한 많은 연구들이 진행 되어왔다. 이 연구들로는 CPU 에뮬레이션을 이용하는 동적 분석 방법[11,12]과 정적·동적 분석을 함께 이용하는 하이브리드 분석 방법[13]이 있다.

동적 분석 방법은 대부분의 폴리모픽 코드들을 찾을 수 있는 장점이 있지만, 실제 CPU와 같이 패킷 내의 페이로드를 하나씩 실행해 보아야 하기 때문에 처리 능력이 떨어지며, 이로 인해 통신망에 직접적으로는 적용이 힘들다는 단점이 있다. 하이브리드 분석 방법은 정적 분석을 통해 페이로드에서 에뮬레이션을 시작할 위치를 우선적으로 선정한다는 점에서 전자와 비교해서는 성능적인 오버헤드가 작지만, 정적 분석과 에뮬레이터 구현에 대한 복잡성 및 처리 성능 상의 오버헤드 측면 때문에 통신망에 적용하는 것이 역시 만족스럽지는 않다. 따라서 본 논문에서는 정적 분석 방법을 통해서도 역어셈블 방지 기법과 자기 수정 코드 기법이 사용된 폴리모픽 코드를 탐지할 수 있으며, 하이브리드 방법 수준의 탐지 성능을 가지면서도 구현이 간단한 고속 정적 분석 방법을 제안하고자 한다.

일반적으로 폴리모픽 코드를 제작하는 공격자는 원격 호스트에 전송된 암호화된(encrypted) 원본 코드의 주소를 예측하기 어렵기 때문에 복호화 루틴(Decryption Routine)을 통해 원격 호스트의 현재 프로그램 카운터(Program Counter) 값을 스택 상에 저장하고, 그 값을 암호화된 코드의 메모리를 액세스하기 위한 주소로 사용되도록 공격 코드를 작성한다. 본 논문에서는 이 프로그램 카운터 값의 이동을 정적으로 추적하여 폴리모픽 코드의 행동 패턴을 탐지하는 방법을 제안한다. 본 논문의 나머지 내용은 다음과 같다. 제 2 장에서는 관련연구로서 역어셈블 방지와 자기 수정 코드 기법에 대하여 알아본다. 제 3 장에서는 본 논문에서 제안하는 탐지 방법을 제시하고, 제 4 장에서는 제안된 방법에 대하여 성능평가를 수행하고 그 결과를 제시하며, 마지막으로 제 5장에서 끝을 맺는다.

II. 관련 연구

Snort[1]와 같은 시그니처-기반 네트워크 침입탐지 시스템들은 기존에 알려진 쉘 코드의 특정 부분, 연속된 NOP 슬레드(sled) 등과 같이 공통적인 실제 콘텐츠를 시그니처로 만들어서 공격을 탐지한다. 그러나 시그니처-기반 탐지 기법들을 회피하기 위하여 쉘 코드들은 단순한 폴리모픽 기법으로 암호화 되었으며, 연속된 NOP 슬레드 부분들은 메타모픽 기법에 의해 다양하게 변형되었다.

새로운 쉘 코드를 탐지하기 위해 Polygrah[14], PAYL[15], PADS[16] 등과 같이 통계적인 방법으로 공통적인 스트링(Prevalent String)들을 분석하는 방법들이 연구되었지만, 단순히 변형된 쉘 코드들은 탐지가 가능하지만 복잡한 형태로 변형된 쉘 코드들에게는 여전히 취약한 요소를 가지고 있다. 또한 네트워크 패킷들에서 공통적인 스트링의 발생 빈도를 측정하는 방법을 사용하기 때문에 공격 대상이 정해져 있는 소규모의 공격들에는 효과적이지 못한 점이 있다.

변형된 슬레드 및 암호화된 쉘 코드에 대하여 시그니처 기반 및 공통적인 스트링들의 발생 빈도를 이용한 탐지 방법들의 한계가 드러나면서, 보다 최근의 연구들은 쉘 코드를 찾기 위해 패킷의 페이로드에 대해 정적 바이너리 코드 분석 방법들[7,9,17]을 적용하기 시작하였다. 그러나 이런 방법들은 정상적인 데이터와 공격 데이터에 대한 학습에 따라 탐지 오차가 클 수 있으며, 자기 수정 코드와 같은 정적 분석 방법에 대항하는 기법에 대하여 무력화 될 수 있는 문제가 있다. 다음에 관련 연구로서 정적인 바이너리 코드 분석을 회피할 수 있는 최근 기술들에 대하여 서술한다.

2.1 역어셈블 방지(Thwarting Disassembly)

기계코드(Machine Code)는 보통 여러 가지의 부분으로 구성되는데, 프로그램에 대한 여러 가지 정보를 포함하는 섹션들과 헤더 섹션들이 있다. 헤더는 프로그램 진입점과 명령어의 전체 크기에 대한 정보를 포함한다. 역어셈블은 읽을 수 있는 문서 형태로 된 파일로부터 어셈블리 코드 명령어의 순서를 복구하는 과정을 말한다. 일반적으로 사용되는 방법으로는 두 가지가 존재 한다: Linear Sweep과 Recursive Traversal.

선형 역어셈블 알고리즘은 무효한(invalid) 명령어나 데이터의 끝에 도달할 때까지 차례대로 각각의 명령어들을 해석한다. [그림 1]은 Metasploit 프레임워크 프로그램의 Countdown 암호화 엔진으로 생성된 쉘 코드의 복호화 코드 부분의 역어셈블된 부분을 보여준다.

```
0000   6A7F         push 0x7F
0002   59           pop  ecx
0003   E8FFFFFFFF   call  0x7
0008   C15E304C     rcr[esi+0x30],
                    0x4C
000C   0E           push cs
000D   07           pop  es
000E   E2FA         loop 0xA
0010
           <encrypted payload>
           ...
008F
```

[그림 1] 선형 역어셈블의 예

이 알고리즘의 주요 약점은 명령어 스트림 내에 내장된 데이터의 번역 오류로 인하여 역어셈블 에러가 발생하기 쉽다는 것이다. 0x0003 주소의 call 명령어는 실제 수행을 하게 되면 0x0007 주소로 분기되고, 그 주소부터 명령어가 해석된다. 선형 역어셈블의 경우는 분기를 반영하지 않기 때문에 call을 해석한 후, 0x0008주소에서 rcr 명령어를 해석하는 오류를 범하게 된다. 이 방법은 GNU 유틸리티 objdump[18]와 같은 프로그램에서 사용된다.

재귀 역어셈블 기법은 분기문의 목적지 주소를 따라 해석을 하게 된다. [그림 2]는 재귀 역어셈블의 예를 보여준다.

재귀 역어셈블의 경우에도 항상 정확한 지점에서 해석을 한다는 보장이 없다. 쉘 코드는 역어셈블을 방

```
0000   6A7F         push 0x7F
0002   59           pop  ecx
0003   E8FFFFFFFF   call  0x7
0007   FFC1         inc  ecx
0009   5E           pop esi
000A   304C0E07   xor[esi+ecx+0x7], cl
000E   E2FA         loop 0xA
0010
           <encrypted payload>
           ...
008F
```

[그림 2] 재귀 역어셈블의 예

지하기 위하여 분기의 목적지 주소 부분에 절대 값이 아닌 상대적인 주소 값을 이용할 수 있기 때문이다. 이 경우는 상대 주소 값을 가지고 있는 레지스터에 대한 연산이 먼저 이루어져야 하고, 분기문의 실행에 반영되어야 한다.

2.2 자기 수정 코드

역어셈블의 이런 단점을 보완하기 위하여 쉘 코드의 대략적인 제어 흐름 그래프(CFG: Control Flow Graph)를 추출하는 방법들이 연구되었다[7,8]. 이 방법은 쉘 코드의 구조적인 특징을 분석하는 것으로서 상대적인 주소를 이용하는 분기에 대해 대략적인 분석을 수행하여 명령어들의 흐름에 대한 특징을 추출하게 된다.

그러나 이런 분석 방법도 자기 수정 코드 기법을 사용하는 쉘 코드들을 탐지하는데 한계가 있다. 이 기법은 스스로의 코드를 구동 시(run-time)에 동적으로 변경하게 된다. 따라서 CFG에 의해 추출된 흐름과

```
0000   6A7F         push 0x7F
0002   59           pop  ecx
0003   E8FFFFFFFF   call  0x7
0007   FFC1         inc  ecx
0009   5E           pop esi
000a   80460AE0     add[esi+0xA],
                    0xE0
000e   304C0E0B     xor
               [esi+ecx+0xB], cl
0012   02FA         add bh, dl
0014
           <encrypted payload>
           ...
0093   80460AE0     add[esi+0xA],
                    0xE0
```

[그림 3] 폴리모픽 쉘 코드 생성 형태

구동 시의 흐름과는 차이가 있게 된다. [그림 3]은 수정된 Countdown 엔진으로 작성된 폴리모픽 쉘 코드에 대해 생성된 폴리모픽 쉘 코드 모습을, [그림 4]는 구동 시의 실제 코드 모습에 대한 예를 보여준다.

[그림 3]에서는 폴리모픽 쉘 코드의 특징을 쉽게 찾을 수 없다.

```
0000        6A7F              push 0x7F
0002        59                pop  ecx
                              ;ecx=0x7F
0003        E8FFFFFFFF        call 0x7
                              ;PUSH 0x8
0007        FFC1              inc  ecx
                              ;ecx=0x80
0009        5E                pop esi
                              ;esi=0x8
000a        80460AE0          add
[esi+0xA], 0xE0;ADD [0012] 0xE0
000e        304C0E0B          xor
[esi+ecx+0xB], cl;XOR [0093] 0x80
0012        E2FA              loop 0xE
                              ;ecx=0x7F
000e        304C0E0B          xor
[esi+ecx+oxB], cl;XOR [0092] 0x7F
0012        E2FA              loop 0xE
                              ;ecx=0x7E
            ...
```

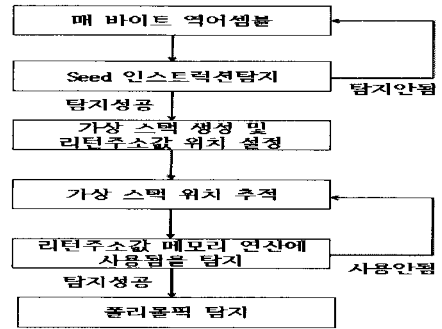
[그림 4] 구동 시의 실제 코드 형태

[그림 4]는 [그림 3]이 실제 실행되어 자기 수정을 한 후의 실제 쉘 코드 모습을 보여준다. [그림 3]의 0x000a 주소의 add [esi+0xA], 0xE0는 0x0012 주소의 add bh, dl을 수정하여 [그림 4]에서는 loop 0xE를 생성한다. 이 loop에 의해 폴리모픽 쉘 코드 내의 암호화된 실행코드가 복호화 된다.

III. 제안된 탐지방법

3.1 동작 개요

일반적으로 폴리모픽 공격 코드는 제작의 용이성을 위해 원격 호스트에 삽입된 복호화 루틴이 원격 호스트의 현재 프로그램 카운터 값을 스택에 저장하고, 이 값을 암호화된 데이터의 메모리를 액세스하는데 사용한다. 이 사실을 이용하여, 제안된 방법에서는 프로그램 카운터 값의 이동을 정적으로 추적하여 폴리모픽 코드의 행동 패턴을 탐지한다. [그림 5]는 제안된 방법의 전체적인 흐름을 보여준다.

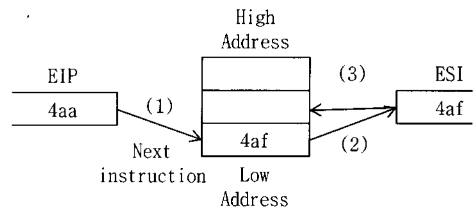


[그림 5] 제안된 방법의 흐름도

[그림 6]은 본 논문에서 제안된 방법에 의해 2008년 3월 실제 국내 통신망에서 탐지된 복호화 루틴을 역어셈블한 코드이고, [그림 7]은 [그림 6] 코드의 동작 과정을 나타낸다.

```
491 5E               pop esi                              (2)
492 31C9             xor ecx,ecx
494 81E9 89FFFFFF    sub ecx,-77
49a 8136 80BF3294    xor dword ptr [esi],9432BF80          (3)
4a0 81EE FCFFFFFF    sub esi,-4
4a6 E2 F2            loopd short 0000049A
4a8 EB 05            jmp short 000004AF
4aa E8 E2FFFFFF      call 00000491                        (1)
```

[그림 6] 폴리모픽 쉘 코드의 복호화 루틴 부분



[그림 7] 프로그램 카운터 값을 악용하는 복호화 루틴의 스택 상의 동작 과정

[그림 6]과 [그림 7]의 경우에, (1)에서 특정 명령어([그림 5]에서는 call)이 다음 명령어의 주소 값을 스택(stack)의 최상위(top)에 저장한다. (2)에서 특정 명령어([그림 5]에서는 pop)이 스택에 저장된 값을 임의의 레지스터([그림 6]에서는 esi)로 로드(load)한다. (3)에서 그 레지스터 값을 이용하여 복호화를 반복적으로 수행([그림 6]에서는 xor과

loopd)하게 된다.

본 논문에서 제안하는 방법은 복호화 루틴의 위와 같은 동작 특성을 정적으로 추적하게 된다. 본 방법은 이와 같은 추적을 하기 위해서 다음과 같은 과정으로 수행 된다:

- (1 단계) 프로그램 카운터 값을 스택에 저장하는 특정 명령어(이하 seed 명령어)를 매 바이트마다 역어셈블을 수행하여 찾는다. 본 방법은 매 바이트 역어셈블을 수행함으로써 역어셈블 방지 기법에 유효하게 된다. seed가 탐지되면 복호화 루틴의 스택 상의 동작 과정을 추적하기 위한 가상 스택(Virtual Stack)을 생성한다.
- (2 단계) 명령어들을 정적으로 분석해 가면서 가상 스택에 저장된 프로그램 카운터 값의 스택 상의 위치를 추적하여, 그 값을 임의의 레지스터로 로드하는 과정을 탐지한다.
- (3 단계) 그 레지스터에 저장된 값이 여러 다른 레지스터들을 거쳐 가면서 최종적으로 암호화된 원본 데이터가 있는 메모리를 액세스하는 명령어에 사용되는 과정을 탐지한다. 폴리모픽 코드를 생성한 공격자의 입장에서는 자기 수정 코드 기법은 3 단계 이후로 적용이 가능하다.

따라서 본 방법은 그 전에 복호화 루틴의 특징을 이미 파악하게 되는 것이다. 자기 수정 코드 기법들이 사용된 폴리모픽 코드들은 대상 호스트에 삽입된 자신의 현재 메모리 위치를 다시 읽어 들여야 암호화된 원본 데이터의 메모리 주소를 계산할 수 있기 때문에, 제안된 방법은 현재 메모리 위치를 다시 읽어 들이는 시점에서 폴리모픽 코드임을 최종판단하는 것이다.

3.2 복호화 루틴 탐지

3.2.1 1 단계: seed 명령어 탐지

seed 명령어는 현재 프로그램 카운터 값을 스택에 저장하는 역할을 하며, 그 값은 자기 수정 코드 기법이나 암호화된 코드의 주소를 찾는데 사용된다. 공격자가 원격 호스트의 메모리 상태를 추측하는 데는 많은 시간적, 환경적 어려움이 따르기 때문에 폴리모픽 코드 제작의 용이성을 위해서 일반적으로 seed 명령어를 복호화 루틴 내에 포함한다.

seed 명령어는 call, fsave, fnsave, fstenv, fnstenv가 주로 사용되고 있기 때문에 본 논문의 방법은 매 바이트 역어셈블 수행하여 우선적으로 위 seed 명령어들을 놓치지 않고 탐지하게 된다. seed가 탐지되면 가상 스택에 프로그램 카운터 값이 들어간 위치가 기록이 되며, call의 경우는 가상 스택의 top에, 나머지 명령어의 경우는 오퍼랜드(operand) 부분을 정적 분석하여 프로그램 카운터 값이 저장될 스택 상의 위치를 계산하게 된다.

예를 들어, fnstenv [esp-0c]의 경우, 프로그램 카운터 값은 스택의 top에 위치되며, -0c가 아닌 다른 값인 경우 -0c를 기준으로 다른 위치를 계산하여 가상 스택에 기록하게 된다. fnstenv [esp-0c]와 같은 예의 경우는 프로그램 카운터가 저장되는 위치가 스택을 나타내는 esp를 기준으로 하고 있지만, call이 아닌 다른 네 개의 f 명령어들의 경우는 esp가 사용되지 않으면 복호화 루틴이 아닌 것으로 판단한다. 그 이유는 공격자는 원격 호스트의 메모리 및 레지스터 상태들을 파악하는 것이 힘들기 때문에 임의의 메모리를 액세스하여 발생되는 위반(violation)을 피하기 위해 스택 주소를 주로 이용하기 때문이다.

3.2.2 2 단계: 프로그램 카운터 값을 로드하는 레지스터 탐지

복호화 루틴에서 폴리모픽 탐지 방법들을 회피하기 위해 때때로 많은 더미(dummy) 명령어들을 삽입하는 경우들이 있다. 따라서, 2 단계에서는 가상 스택에 저장된 프로그램 카운터 값이 어떤 레지스터로 로드되는지를 정적 분석으로 추적하게 된다. 만약, 로드된 프로그램 카운터 값을 이용하지 않고, 임의의 주소 값을 이용하여 메모리를 액세스 하려는 명령어가 나타난다면 복호화 루틴이 아닐 확률이 높다.

본 논문에서 제안하는 방법은 더미 명령어용도 및 스택을 통한 연산에 자주 이용되는 명령어들로서 push/pop, inc/dec/sub/add와 같은 기본 명령어들에 대해 정적 분석 방법을 사용하여 가상 스택에서 프로그램 카운터 값의 위치 변화 및 그 값을 로드하는 레지스터를 추적 및 탐지하게 된다. 2단계에서 위 명령어들 중 추적 대상이 되는 것들은 오퍼랜드 부분을 분석하여 가상 스택의 위치를 변경하는 경우들 만이다. 예를 들어, push의 경우는 가상 스택의 top 포인터의 주소를 감소시키고, pop의 경우는 증가시키는 것으로 간단히 분석이 가능하다. 또한 inc/dec/sub/add의 경우는 출력 오퍼랜드 쪽에 esp 레지스터가 선언되어 있는 경우만을 추적의 대상으로 한다.

[그림 5]를 예로 들어 설명하면 (2)에 의해 esi에

프로그램 카운터 값이 저장이 되지만, 만약 (1)의 call 후에 inc esp, 4와 같은 명령어가 먼저 실행되고 (2)가 실행된다면, 가상 스택 top 포인터가 4바이트 증가해 버리므로 (2)는 다른 값이 로드 된다는 것을 탐지할 수 있는 것이다.

3.2.3 3단계: 프로그램 카운터 값을 이용한 메모리 액세스 탐지

공격자는 〔그림 5〕와 같이, 프로그램 카운터 값을 저장한 레지스터를 암호화된 원본 코드를 액세스 하는데 직접 사용할 수도 있지만, 폴리모픽 탐지 방법들을 회피할 목적으로 프로그램 카운터 값을 여러 레지스터들을 거친 후에 이용할 수도 있다. 프로그램 카운터 값이 레지스터들 간에 이동되는 것을 연결 관계로 표현하고, 연결된 레지스터들 중에 메모리를 액세스하는 명령어에 사용되는 것이 있다면 결국 프로그램 카운터 값을 이용하여 암호화된 메모리를 액세스 하는 것이다.

공격자는 분석을 방해할 목적으로 2 단계와 같이 더미 명령어들을 삽입할 수도 있고, 프로그램 카운터 값을 첫 번째로 로드한 레지스터 값을 다시 push 한 후 다른 레지스터로 pop 할 수도 있으며, 산술 혹은 논리 연산 명령어를 통해서 다른 레지스터로 값을 이동시킬 수도 있다. 제안 방법에서는 전자의 경우는 2 단계와 같이 가상 스택 추적을 통해서 연결 관계를 찾을 수 있고, 후자의 경우는 명령어 오퍼랜드 부분을 입력과 출력으로 구분하고 입력 쪽에 연결 관계에 있는 레지스터가 존재하고 출력 쪽에 새로운 레지스터가 존재한다면 새 레지스터도 연결 관계에 포함시키게 된다.

예를 들어, 프로그램 카운터 값을 처음 로드한 레지스터가 ecx이고, 다음 명령어로 mov eax, ecx가 나타난다면, eax도 ecx와 연관성을 가지게 되는 것이다. 이와 같은 연결 관계 생성은 정적 분석으로도 간단히 수행된다. 최종적으로 연결 관계에 있는 레지스터들 중에 하나라도 〔그림 5〕의 (3)과 같이 메모리를 액세스하는 명령어에 사용된다면 폴리모픽 공격 코드의 복호화 루틴이 존재한다고 판단하게 된다.

3.2.4 분기문 처리

복호화 루틴들은 폴리모픽 탐지 방법들의 분석을 방해할 목적으로 여러 분기를 거쳐서 추적해야 그 특징을 잡아낼 수 있도록 구성될 수도 있다. 분기들을 추적하기 위해 본 방법은 〔그림 8〕 및 〔그림 9〕와 같은 상황들에 대한 처리를 수행할 수가 있다.

```
124f E839000000 call 0000128D
1254 8A01       mov al,[ecx]

      •••

128d 59         pop ecx
128e 51         push ecx
128f C3         retn
```

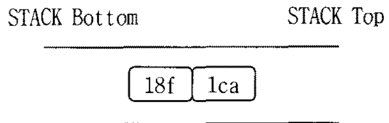
〔그림 8〕 call, ret에 의한 분기의 예제 코드

```
18d 75 39 jnz short 000001C8

      •••

1c8 E1 39 loopde short 00000181

      •••
```



〔그림 9〕 조건 분기문에 의한 분기의 예제 코드 및 가상 분기문 스택의 모습

〔그림 8〕과 〔그림 9〕는 제안된 방법에 의해 2007년 11월에 탐지된 복호화 루틴의 일부분들이다. 〔그림 6〕의 경우, 탐지 1단계에서 call을 탐지한 후, call다음 명령어 주소 0x1254를 가상 리턴 주소 배열에 저장한 후, 0x128d 주소로 분기한다. 탐지 2단계에서 ecx가 프로그램 카운터 값을 저장한 레지스터로 판단되면 가상 리턴 주소에 저장된 값을 초기화한다. 탐지 3 단계에서 그 값이 다시 가상 스택의 top으로 입력되는 것이 탐지되면 가상 리턴 주소 배열에0x1254를 다시 저장하게 된다. 0x128f 주소의 retn에 의해 가상 리턴 주소 배열에 저장된 값이 가리키는 주소인 0x1254로 이동하고, ecx가 메모리 액세스 연산에 사용된다는 것이 최종적으로 탐지된다.

〔그림 9〕는 조건 분기문 처리 방식을 보여주기 위한 예제이다. 제안된 방법은 조건 분기문을 만나면 깊이 우선 탐색을 수행하게 된다. 0x18d주소에서 조건 분기문을 만나게 되면, 다음 명령어 주소인 0x18f와 현재까지의 탐지 상태를 가상 분기문 스택의 top에

저장한다. 0x1c8주소로 분기되면 다시 일종의 조건 분기문인 loopde 명령어가 나타나게 되고, 다음 명령어 주소인 0x1ca를 top에 저장한다. 분석 도중 복호화 루틴이 아님이 판단되면, 가상 분기문 스택에 저장해 둔 정보를 로드 하여 그 시점부터의 분석을 계속 수행하게 된다. 이와 같은 과정을 조건 분기문 스택에 저장된 내용이 없거나, 복호화 루틴으로 최종 판단될 때까지 반복하게 된다.

3.2.5 탐지 과정의 예

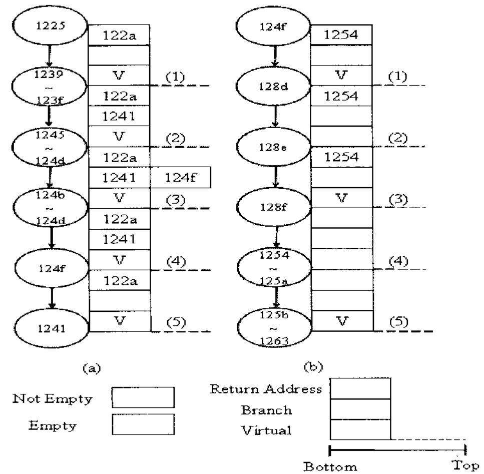
〔그림 10〕은 제안된 방법에 의하여 실제 하니넷에서 탐지된 폴리모픽 공격코드의 복잡한 복호화 루틴을 보여준다.

```
1225 E80F000000    call 00001239                       (continued…)
122a E862000000    call 00001291          1268 8801           mov [ecx],al
122f 6764FF360000  push dword ptr fs:[0]  126a 83C101         add ecx,1
1235 5D            pop ebp                126d 83EF01         sub edi,1
1236 8D6D08        lea ebp,[ebp+8]        1270 83FF00         cmp edi,0
1239 29C0          sub eax,eax            1273 77 EE          ja short 00001263
123b FEC8          dec al                 1275 59             pop ecx
123d 08C0          or al,al               1276 67648B260000   mov esp,fs:[0]
123f 7404          je short 00001245      127c 64678F060000   pop dword ptr fs:[0]
1241 75F8          jnz short 0000123B     1282 C9             leave
1243 EB67          jmp short 000012AC     1283 894C2418       mov [esp+18],ecx
1245 29F6          sub esi,esi            1287 61             popad
1247 29C9          sub ecx,ecx            1288 FFE1           jmp ecx
1249 B175          mov cl,75              128a 90             nop
124b 46            inc esi                128b 90             nop
124c 49            dec ecx                128c 90             nop
124d 75FC          jnz short 0000124B     128d 59             pop ecx
124f E839000000    call 0000128D          128e 51             push ecx
1254 81C14A000000  add ecx,4A             128f C3             retn
125a 51            push ecx               1290 90             nop
125b 31FF          xor edi,edi            1291 8B442410       mov eax,[esp+10]
125d 81CF3C240000  or edi,243C            1295 8F80B8000000   pop dword ptr[eax+B8]
1263 8A01          mov al,[ecx]           129b 31C0           xor eax,eax
1265 6629F0        sub ax,si              129d C3             retn
        (continued…)                      129e 90             nop
```

〔그림 10〕 실제 하니넷에서 탐지된 복호화 루틴

〔그림 11〕은 〔그림 10〕에 대하여 제안된 방법에서의 (a) 분기 추적과 (b) 탐지 성공에 대한 과정 흐름을 보여준다.

〔그림 10〕과 11에서, (1) 주소 0x1225의 *call*은 *seed*로 탐지된다. 다음 명령어의 주소 0x122a는 귀환(return) 주소 스택으로 푸쉬되며 가상 스택 상의 *base* 위치를 가리키게 된다. 현재 프로그램 카운터는 0x1239로 변경된다. (2) 0x1239부터 0x123d까지, *base*를 적재(load)하는 레지스터가 없다. 0x123f에서, 다음 명령어의 주소 0x1241과 현재 탐지 상태가



〔그림 11〕 (a) 분기 추적과 (b)탐지 성공에 대한 과정 흐름

분기 스택 상에 푸쉬된다. 프로그램 카운터는 0x1245로 변경된다. (3) 주소 0x124d의 명령어 *jnz*는 forward와 jump의 두 경로를 가지기 때문에, 제안된 방법은 먼저 forward 0x124f를 분기 스택으로 푸시 한다. (4) 0x124b로 점핑한 후, 0x124d가 다시 방문된다면, 현재 경로의 프로세스가 정지되고 분기 스택에 저장된 주소 0x124f가 프로그램 카운터로 팝 된다. 따라서 제안된 방법은 루프 트랩에 빠지지 않는다. (5) 0x124f에서, *seed*가 다시 나타나기 때문에, 현재 경로의 프로세스가 정지되고 프로그램 카운터는 가상 스택으로부터 0x1241로 갱신된다.

〔그림 11(a)〕에서 주소 0x1225의 seed로부터 시작한 현재 탐지 과정은 위의 단계와 같은 분기를 제어하기 위하여 규칙에 따라 0x1241부터 또한 계속된다. *seed*로부터 시작한 탐지 과정은 여러 경로를 포함하며, 제안된 방법의 현재 구현에서는 경로가 정지되는 때는 프로세스 경로가 다음과 같을 경우 이다:

a) 같은 주소에 있는 같은 분기 명령어를 다시 방문할 때

b) 다른 *seed*를 만날 때,

c) *base*를 포함하지 않는 메모리 액세스 명령어를 만날 때,

d) 원 거리 혹은 긴 점프 명령어를 만날 때,

e) 표본에서 무효한 주소를 만날 때.

〔그림 11(b)〕에서, 주소 0x124f의 *call*은 *seed*로 탐지된다. (2) 0x128d에서, 명령어 *pop*는 *base*를 *ecx*로 적재한다. 가상 스택 상의 *base*에 대한 위치 점검이 없어지고 *base*에 대한 연결 관련 그래프가 구

축된다. (3) 0x128e에서, 그래프 내의 *base*가 다시 가상 스택 상으로 푸시 된다. (4) *retn*에 의하여, 현재 프로그램 카운터는 귀환 주소 스택으로부터 0x1254로 갱신된다. (5) 마지막으로, 0x1263에서 *base*가 *mov al, [ecx]*를 위하여 메모리 액세스를 위하여 사용된 것이 확인되었기 때문에, 제안된 방법은 표본에 복호화 루틴이 포함되었다는 것을 결정할 수 있다.

IV. 성능 평가

4.1 탐지율

본 논문에서 제안한 방법을 이용하여 대표적인 13개의 폴리모픽 코드 생성 엔진들을 대상으로 복호화 루틴의 탐지율을 측정하였다. 이 엔진들은 동적 분석 방법 및 하이브리드 분석 방법 등 여러 다른 논문들을 통해서도 같은 목적으로 주로 사용된 것들이다. 특히 Countdown과 JmpCallAdditive는 역어셈블 방지 기법이 적용된 것이며, Alpha2와 ShikataGaNai는 자기 수정 코드 기법이 사용된 복호화 루틴을 생성하는 엔진들이다. 각 엔진들마다 10개씩의 폴리모픽 코드들을 생성하였고, 제안된 방법에 의해 100% 탐지가 이루어졌다. 따라서 제안된 방법은 여러 정적 분석 방지 기법을 사용한 폴리모픽 공격 코드에도 유효하며, 에뮬레이션을 수행하는 다른 방법들과도 유사한 탐지율을 나타낸다는 것을 알 수 있었다. [표 1]은 13개의 폴리모픽 엔진들에 대한 평균 명령어 카운트와 탐지율을 보여준다.

[표 1] 주요 폴리모픽 쉘 코드 생성 엔진들에 대한 탐지율

폴리모픽 생성 엔진		평균 명령어 카운트	탐지율 (%)
ADMmutate		7	100
Clet		3	100
Alpha2		26	100
Metasploit	Countdown	4	100
	JmpCallAdditive	4	100
	ShikataGaNai	4	100
	Call4DwordXor	4	100
	FnstenvMov	3	100
	NonAlpha	9	100
	PexFnstenvSub	3	100
	Pex	4	100
	PexAlphaNum	20	100
	PexFnstenvMov	3	100

평균 명령어 카운트는 여러 경로들 중에서 최종 탐지 경로에서 seed로부터 메모리 액세스 명령어 까지 거치게 되는 평균 명령어의 수를 의미한다. 표 1에서는 평균 명령어 수는 일반적으로 아주 작은 것을 보여준다. 이것은 본 논문에서 제안된 방법이 복호화 루틴의 의심스러운 명령어 범위 안에서 에뮬레이션-기반 기법보다 더 빠른 결정 속도를 가진다는 것을 보여준다. 게다가 높은 수는 제안된 방법이 ADMmutate, Alpha2, NonAlpha와 PexAlphaNum과 같은 복잡한 복호화 루틴의 경우에도 또한 탐지가 가능하다는 것을 보여준다.

4.2 오탐율

2005년 7월 28일 오전 11시부터 국내 모 대학에서 하루 동안 수집한 트래픽을 대상으로 오탐율을 측정하였다. 그 대학의 통신망은 대학 구성원의 요청에 의해서만 포트 및 IP 주소를 개방하고 있으며, 게이트웨이 포인트 및 통신망 내부적으로도 침입방지시스템(IPS: Intrusion Prevention System)를 중간에 적용하여 실시간으로 공격을 차단하는 일종의 클린 네트워크로 운영되고 있었다. 수집한 트래픽을 대상으로 2008년 4월 26일자 V3와 Snort 2.8.1을 운영한 결과에서도 클린한 트래픽임을 확인하였다. 따라서 본 논문에서 제안된 방법의 오탐율을 측정하는데 적당한 트래픽으로 판단된다. 오탐율은 아래와 같이 계산되었다.

$$\text{오탐율} = \frac{\text{탐지된 (클린)패킷 수}}{\text{전체 (클린)패킷 수}}$$

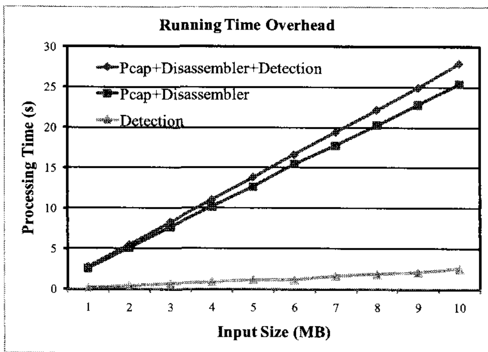
전체 수집된 트래픽의 파일 크기는 107GB이고, 총 패킷 수는 31,297,002개, 이 중 138개가 본 방법에 의해 폴리모픽 공격 코드로 탐지되었다. 따라서 본 방법의 오탐율은 0.0004%로서 거의 0%에 가까운 결과를 보여주었다. 138개를 하나씩 수작업으로 분류한 결과 모두 공격과 관련된 것이 아님이 판명되었고, 한 패킷 내에서 같은 패턴에 의해 여러 번 탐지된 것을 고려한다면 실제 패킷 당 계산된 오탐율은 0.0004% 보다 적어지게 된다.

4.3 처리부하

본 방법이 실행되는 컴퓨팅 환경은 리눅스 커널 2.6.9, 펜티엄 3.2GHz, 4GB의 메인 메모리를 가진

시스템으로서, 수집한 트래픽을 관리하고 있는 NAS (Network Area Storage)로부터 pcap형태로 저장된 해당 파일로부터 패킷을 하나씩 읽어 들여서 실험이 이루어졌다.

본 방법이 적용된 실험 프로그램은 기본적으로 pcap 처리, 역어셈블 처리, 탐지 알고리즘 수행의 세 가지 부하가 존재한다. 본 실험 프로그램 내에서는 gettimeofday()를 사용하여 각각의 처리 시간들을 측정하였다.



〔그림 12〕 입력 데이터의 크기 별 처리시간 측정결과

〔그림 12〕에서 중간 그래프(Pcap+ Disassembler)는 pcap 처리와 탐지 1단계만 수행할 경우의 수행 시간을 나타내고, 가장 위쪽 그래프 (Pcap+Disassembler+Detection)는 탐지 3단계까지 모두 수행했을 경우의 시간이다. 마지막 그래프는 두 그래프의 시간차로 구해지는 일종의 탐지 알고리즘만의 수행 시간으로 예상되는 결과 그래프이다.

실험 결과로부터 주목할 만 한 사실은 제안된 방법은 입력되는 데이터의 크기에 따라 선형적으로 부하가 증가한다는 사실이다. 또한 실험 프로그램은 아직 최적화가 이루어지지 않은 상태이며, 실험 데이터가 실험 프로그램이 수행되는 로컬 스토리지가 아닌 원격지에서 통신망을 통해 전송되는 환경이었지만, 절대적인 값으로만 보더라도 본 방법은 대략 3Mbits/s의 처리성능을 나타내었고, 대부분은 공개 역어셈블러와 pcap의 오버헤드이다.

V. 결 론

본 논문에서는 폴리모픽 공격 코드를 탐지하기 위한 에뮬레이션 기반의 방법들의 복잡성 및 성능 문제로 인해 통신망에 직접적으로 적용이 힘들다는 점을 주목하였다. 이 문제를 해결하고자 정적 분석 방법을 이용한 새로운 폴리모픽 공격 코드 탐지 방법을 제안하였다. 본 논문에서 제안된 방법은 대부분의 복호화 루틴들이 스택 상에 원격 호스트의 프로그램 카운터 값을 저장하고, 그 값을 암호화된 원본 코드가 위치하는 메모리를 액세스하기 위한 주소로 이용 한다는 사실에 착안하였다. 제안된 방법은 정적 분석 방법을 이용하여 복호화 루틴의 운용 특징을 추적한다.

최근의 폴리모픽 공격코드들은 정적 분석 방법을 회피하기 위하여 역어셈블 방지 기법과 자기 수정 코드 기법을 적용하고 있다. 본 논문에서는 실험 결과를 통해 제안된 방법이 역어셈블 방지 방법과 자기 수정 코드 기법을 포함하고 있는 주요한 13 개의 폴리모픽 엔진으로 생성된 변형된 코드들에 대하여도 100%의 탐지율을 보여주었다. 또한 기존의 정적 분석 방법들의 한계로 여겨지던 정적 분석 방지 기법들이 사용된 폴리모픽 코드들에도 본 방법은 유효하게 동작함을 보여주었다. 마지막으로 에뮬레이션 기반 방법들에 대해 오탐율 면에서는 유사한 수준이며, 성능과 복잡성 면에서 좀 더 좋은 결과를 보여 주었다. 본 논문에서 제안된 기법을 이용한 프로토타입 제품이 현재 국내망에서 실제 운용 중이며, 향후 기술이전을 통하여 상용제품으로 개발될 예정이다.

참 고 문 헌

〔1〕 Snort, http://www.snort.org

〔2〕 Metasploit, http://www.metasploit.com

〔3〕 K2, ADMmutate, http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz

〔4〕 T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk, "Polymorphic shellcode engine using spectrum analysis," Phrack Magazine, vol. 11, no. 61, Aug. 2003.

〔5〕 B.J. Wever, "Alpha2," http://www.edup.tudelft.nl/~bjwever/src/alpha2.c

〔6〕 TAPiON, http://pb.specialised.info/all/tapion/

〔7〕 R. Chinchani and E. Berg, "A Fast Static Analysis Approach To Detect Exploit Code Inside Networks Flows," Proc. 8th International Symposium on Recent Advances in Intrusion Detection (RAID),

pp. 284-308, Sep. 2005.

[8] C. Krugel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," Proc. 8th International Symposium on Recent Advances in Intrusion Detection (RAID), pp. 207-226, Sep. 2005.

[9] X. Wang, C. Pan, P. Liu, and S. Zhu, "SigFree: A Signature-free Buffer Overflow Attack Blocker," Proc. 15th USENIX Security Symposium, pp. 225-240, July 2007.

[10] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," CCS'03, pp. 290-299, Oct. 2003.

[11] M. Polychronakis, K. Anagnostakis, and E. Makatos, "Network-Level Polymorphic Shellcode Detection Using Emulation," Proc. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), pp. 54-73, July 2006.

[12] M. Polychronakis, K. Anagnostakis, and E. Makatos, "Emulation-based Detection of Non-self-contained Polymorphic Shellcode," Proc. International Symposium on Recent Advances in Intrusion Detection (RAID), pp. 87-106, Sep. 2007.

[13] Q. Zhang, D.S. Reeves, P. Ning, and S.P. Lyer, "Analyzing network traffic to detect self-decrypting exploit code," Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS), pp. 20-22, Mar. 2007.

[14] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms," In Proc. of the IEEE Security & Privacy Symp., pp. 226-241, May 2005.

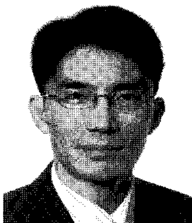
[15] K. Wang and S.J. Stolfo, "Anomalous Payload-based Network Intrusion Detection," In Proc. of the 7th Int'l Symp. on Recent Advances in Intrusion Detection(RAID), pp. 201-222, Sep. 2004.

[16] Y. Tang and S. Chen, "Defending Against Internet Worms: a Signature-based Approach," in Proc. of the 24th Annual Joint Conf. of IEEE Computer and Comm. Societies(INFOCOM), pp. 1384-1394, Mar. 2005.

[17] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis, "STRIDE: Polymorphic Sled Detection through Intrusion Sequence Analysis," In Proc. of the 20th IFIP Int'l Information Security Conf.(SEC'05), pp. 375-392, June 2005.

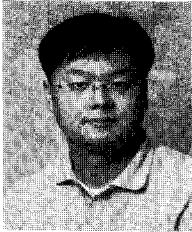
[18] Objdump, GNU Manuals Online, GNU Project-Free Software Foundation, http://www.gnu.org/manual/binutils-2.10.1/html_chapter/binutils-4.html

〈著者紹介〉

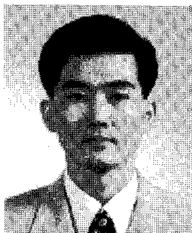


오 진 태 (Jintae Oh) 정회원
1990년 2월: 경북대학교 전자공학과 공학사
1992년 2월: 경북대학교 전자공학과 석사
1992년 2월 ~ 1998년 2월: 한국전자통신연구원 책임연구원
1998년 3월 ~ 1999년 1월: 미국 MinMax Tech. 연구원
1999년 2월 ~ 2001년 10월: 미국 Engedi Networks, Director
2001년 10월 ~ 2003년 1월: 미국 Winnow Tech. Co-founder, CTO 부사장
2003년 3월 ~ 현재: 한국전자통신연구원 지식정보보안연구부 책임연구원
2008년 1월 ~ 현재: 한국정보보호학회 학회지 편집위원(간사)
〈관심분야〉 네트워크보안, 비정상행위탐지기술, 공격 시그니처 자동생성기술, 보안하드웨어기술

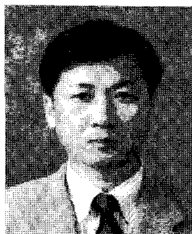
김 대 원 (Dae Won Kim) 정회원
2003년 2월: 경북대학교 전자공학과 공학사
2005년 2월: KAIST 전자공학과 석사
2005년 3월 ~ 현재: 한국전자통신연구원 지식정보보안연구부 연구원
〈관심분야〉 네트워크보안, 멀웨어탐지기술



김 익 균 (Ik Kyun Kim) 정회원
1994년 2월: 경북대학교 컴퓨터공학과 공학사
1996년 2월: 경북대학교 컴퓨터공학과 석사
2009년 2월: 경북대학교 컴퓨터공학과 박사
1996년 ~ 1999년: 한국전자통신연구원 연구원
2000년 ~ 2001년: Paxcomm Ltd. 연구원
2001년 ~ 현재: 한국전자통신연구원 지식정보보안연구부 선임연구원
2004년 ~ 2005년: 미국 Purdue University 방문 연구원
〈관심분야〉 DDoS 보안 메커니즘, 고속망 보안 시스템 및 네트워크 프로세서 설계



장 종 수 (Jong Soo Jang) 종신회원
1984년: 경북대학교 전자공학과 학사
1986년: 경북대학교 대학원 전자공학과 석사
2000년: 충북대학교 대학원 컴퓨터공학과 박사
1989년 7월~현재: 한국전자통신연구원 지식정보보안연구부 책임연구원
2008년 1월 ~ 현재: 한국정보보호학회 부회장, 학회지 편집위원
2008년 1월 ~ 현재: 한국정보처리학회 이사, 논문지 편집위원
2009년 1월 ~ 현재: 대한전자공학회 통신소사이티 이사
〈관심분야〉 Network Security, 정책기반보안관리, 비정상트래픽탐지, 유해정보차단



전 용 희 (Yong Hee Jeon) 종신회원
1971년 3월 ~ 1978년 2월: 고려대학교 전기전자전파공학부
1985년 8월 ~ 1987년 8월: 미국 플로리다 공대 대학원 컴퓨터공학과
1987년 8월 ~ 1992년 12월: 미국 노스캐롤라이나주립 대학원 Elec. and Comp. Eng. 석사, 박사
1978년 1월 ~ 1978년 11월: 삼성중공업(주)
1978년 11월 ~ 1985년 7월: 한국전력기술(주)
1979년 6월 ~ 1980년 6월: 벨기에 벨가톰사 연수
1989년 1월 ~ 1989년 6월: 미국 노스캐롤라이나주립대 Dept of Elec. and Comp. Eng. TA
1989년 7월 ~ 1992년 9월: 미국 노스캐롤라이나주립대 부설 CCSP (Center For Comm. & Signal Processing) RA
1992년 10월 ~ 1994년 2월: 한국전자통신연구원 광대역통신망연구부 선임연구원
1994년 3월 ~ 현재: 대구가톨릭대학교 컴퓨터·정보통신공학부 교수
2001년 3월 ~ 2003년 2월: 대구가톨릭대학교 공과대학장 역임
2004년 2월 ~ 2005년 2월: 한국전자통신연구원 정보보호연구단 초빙연구원
2007년 1월 ~ 2007년 12월: 한국정보보호학회 학회지 편집위원장
2008년 1월 ~ 현재: 한국정보보호학회 부회장
2009년 1월 ~ 현재: 한국정보과학회 정보보호연구회 위원장
〈관심분야〉 네트워크 보안, DDoS 대응 기술, 지능형 전력망(Smart Grid) 보안, 산업 제어 시스템 보안