ISSN : 1226-7244 (Print)
ISSN : 2288-243X (Online)
논문번호 21-03-07
40
j.inst.Korean.electr.electron.eng.Vol.25,No.3,451~461,September 2021
https://doi.org/10.7471/ikeee.2021.25.3.451

PE 파일 분석을 위한 함수 호출 그래프 생성 연구

Generating Call Graph for PE file

김 대 엽*★

DaeYoub Kim*★

Abstract

As various smart devices spread and the damage caused by malicious codes becomes more serious, malicious code detection technology using machine learning technology is attracting attention. However, if the training data of machine learning is constructed based on only the fragmentary characteristics of the code, it is still easy to create variants and new malicious codes that avoid it. To solve such a problem, a research using the function call relationship of malicious code as training data is attracting attention. In particular, it is expected that more advanced malware detection will be possible by measuring the similarity of graphs using GNN. This paper proposes an efficient method to generate a function call graph from binary code to utilize GNN for malware detection.

요 약

다양한 스마트 기기의 보급으로 인하여 악성코드로 인한 피해를 더욱 심각해지면서 머신러닝 기술을 활용한 악성코드 탐지 기술이 주목 받고 있다. 그러나 코드의 단편적인 특성만을 기반으로 머시러닝의 학습 데이터를 구성할 경우, 이를 회피하는 변종 및 신종 악성코드는 여전히 제작하기 쉽다. 이와 같은 문제를 해결하기 위한 방법으로 악성코드의 함수호출 관계를 학습 데이터로 사용하는 연구가 주목받고 있다. 특히, GNN을 활용하여 그래프의 유사도를 측정함으로써 보다 향상된 악성코드 탐지가 가능할 것으로 예상된다. 본 논문에서는 GNN을 악성코드 탐지에 활용하기 위해 바이너리 코드로부터 함수 호출 그래프를 생성하는 효율적인 방안을 제안한다.

Key words : FCF, FCG, Disassembly, PE file, Malware, GNN

Ⅰ. 서론

랜섬웨어와 같은 악성코드의 다양성과 지능화로 인한 피해 규모가 급속도로 증가하고 있다. 특히 개인 모바일 기기 보급의 증가와 함께 스마트 홈과 같은 IoT(Internet of Things) 기술을 활용한 서비스가 보편화 되면서 그 피해가 기하급수적으로 증가할 것으로 예상 된다[1]. 그러나 악성코드의 코드 특성(Signature)을 기반으로 수집된 악성코드를 정적으로 분석하고, 분석 결과를 바탕으로 악성코드를 탐지하는 전통적인 기술은 점차 지능화 및 다양화 되고 폭발적으로 증가하는 악성코드에 대응하기 역부족이다. 특히, 신종 또는 변종과 같은 알려지지 않은 악성코드 탐지에는 전통적 탐지 방법의

* Dept. of Information Security, Suwon University
★ Corresponding author
E-mail : daeyoub69@suwon.ac.kr, Tel : +82-31-229-8352
※ Acknowledgment
This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT)(No. NRF-2021R1F1A1062954).
Manuscript received Aug. 18, 2021; revised Sep. 13, 2021; accepted Sep. 23, 2021.

한계가 분명히 드러난다. 알려지지 않은 신종/변종 악성코드의 위험은 연결형 자동차(Connected Vehicle)와 스마트 홈(Smart Home)과 같은 지능형 기기의 보급과 함께 악성코드의 피해가 곧 인명 피해로도 연결될 수 있기 때문에 보다 실질적인 대응 방안이 필수적으로 요구 된다[2].

악성코드 특성을 정적으로 분석하거나 동적 기능 분석을 통하여 악성코드의 특징을 규정하는 기술의 한계를 개선하기 위하여 다양한 연구들이 진행되고 있다. 예를 들어, 악성코드에서 사용하는 API 함수 호출 패턴을 분석하는 새로운 연구 방향도 지속적으로 진행되어 왔다[3][4]. 또한 머신러닝과 딥러닝 기술을 활용하여 악성코드를 예측/탐지하는 다양한 기술들이 연구되고 있다[5]. 특히, 악성코드 탐지에 머신러닝 기술을 활용하기 위한 악성코드 특성들에 대한 연구도 지속적으로 진행되고 있다[6].

최근 들어 악성코드의 함수 호출을 스퀀스(FCS, Function Call Sequence)나 그래프(FCG, Function Call Graph)로 구성한 후, 머신러닝 기술을 적용하는 연구가 주목을 받고 있다[7][8][9][10]. 그러나 이와 같은 연구들은 주요 API 함수 호출의 연관 관계만을 그래프로 표현하여 분석하거나 IDA Pro와 같은 코드 분해기(Dissembler)를 이용하기 때문에 응용 프로그램의 함수 호출을 전체적으로, 그리고 상세하게 그래프로 표현하는데 한계가 있다. 특히, GNN(Graph Neural Networks)을 적용하여 악성코드를 탐지하기 위해서는 보다 정밀한 그래프가 요구된다[11].

뿐만 아니라 악성코드 제작에 사용된 프로그램 언어나 악성코드가 동작하는 운영체제 마다 컴파일러의 특성이 있기 때문에 일반적으로 적용하기 어렵다는 한계도 존재 한다[12][13][14]. 그럼에도 불구하고 실행코드의 단편적 특성만을 기반으로 악성코드를 탐색할 경우 쉽게 변종을 생성하고 탐색 우회 기술을 적용할 수 있기 때문에 이와 같은 프로그램의 전체적인 구조와 흐름을 기반으로 악성코드를 탐지하는 기술은 향후에도 지속적으로 연구될 분야라 할 수 있다.

프로그램 코드의 FCG를 생성하기 위해 일반적으로 사용되는 방법은 함수의 호출 흐름(FCF, Function Call Flow)을 순차적으로 분석하는 것이다. 그러나 함수 호출 흐름만을 사용하여 FCG를 구성하는 경우, 코드의 정적 분석 시 함수 호출 흐름으로는 분석되지 않는 경우들이 존재한다. 함수 호출을 위해 사용된 CALL 명령의 피연산자가 특정 함수의 시작주소가 아니라 레지스터(CPU Register)로 표현된 경우가 그 대표적인 예라 할 수 있다. 뿐만 아니라 FCF 기반으로 FCG를 생성하면 함수의 흐름은 대략적으로 이미지화 할 수 있지만, 함수 자체의 특성은 FCG에 포함하기 어렵다는 단점이 있다. 이와 같은 FCF 기반의 FCG 생성의 문제점을 해결하기 위하여 본 논문에서는 프로그램 코드를 정적으로 분석하여 함수블록들을 파싱(Parsing)한 후, 생성된 함수블록들을 기반으로 함수 호출 흐름을 분석하여 FCG를 생성하는 절차를 제안한다. 특히, 본 논문에서 제안하는 방법은 프로그램 코드에 포함된 모든 함수를 식별하여 함수 블록으로 구성하는 것이기 때문에 프로그램 코드에서 사용하는 외부 API 함수 호출뿐만 아니라 내부 함수 호출도 FCG로 구성할 수 있다. 함수블록 생성 시, DLL과 같은 외부 함수 호출의 경우 함수 호출을 위한 함수 테이블을 경유하여 호출되는 특성이 있기 때문에 FCG에 의미 없는 함수 노드가 생성된다. 이와 같은 비효율성을 개선하기 위하여 함수 호출 경로에서 함수 호출 테이블을 분석하여 직접 연결하도록 코드를 재구성하였다. 또한 악성코드 분석과 그래프 유사도 분석의 정밀도를 높이기 위하여 함수의 매개변수 개수, 함수의 크기, 호출자(Caller)의 수, 피호출자(Callee)의 수와 같은 함수 자체의 특성을 그래프에 추가할 수 있도록 하였다.

제안된 기술은 윈도우즈 PE(Portable Executable) 파일 구조분석을 기반으로 파이썬(Python 3.8)으로 구현하였으며, 코드 분해(Dissembly)를 위하여 capstone 라이브러리를 이용하였다.

II. 함수 호출 그래프(FCG)

1. PE 파일 구조

코드를 정적으로 분석해 함수블록들을 구성하기 위해서는 PE(Portable Executable) 파일구조 분석이 필요하다. PE 파일구조는 실행파일, 오브젝트 코드, DLL과 같은 윈도우즈 운영 시스템에서 사용되는 파일의 구조로 윈도우즈 운영 시스템에서 프로그램을 실행하기 위해 필요한 정보를 저장하기 위한 데이터 구조이다. 그림 1은 일반적인 PE 파일 구조를 설명한다. PE 파일구조는 프로그램 실행 코

드뿐만 아니라 DLL 참조 정보, API 입출력 테이블, 리소스 데이터, 쓰레드 데이터와 같은 다양한 정보를 포함한다. PE 파일구조는 PEView와 같은 응용프로그램을 이용하여 쉽게 확인할 수 있으나 난독화가 적용된 경우, Stud_PE와 같은 별도의 프로그램이 필요하다.

코드를 정적으로 분석해 함수블록들을 구성하기 위해서는 PE(Portable Executable) 파일구조 분석이 필요하다. PE 파일구조는 실행파일, 오브젝트 코드, DLL과 같은 윈도우즈 운영 시스템에서 사용되는 파일의 구조로 윈도우즈 운영 시스템에서 프로그램을 실행하기 위해 필요한 정보를 저장하기 위한 데이터 구조이다. 그림 1은 일반적인 PE 파일구조를 설명한다. PE 파일구조는 프로그램 실행 코드뿐만 아니라 DLL 참조 정보, API 입출력 테이블, 리소스 데이터, 쓰레드 데이터와 같은 다양한 정보를 포함한다. PE 파일구조는 PEView와 같은 응용프로그램을 이용하여 쉽게 확인할 수 있으나 난독화가 적용된 경우, Stud_PE와 같은 별도의 프로그램이 필요하다.

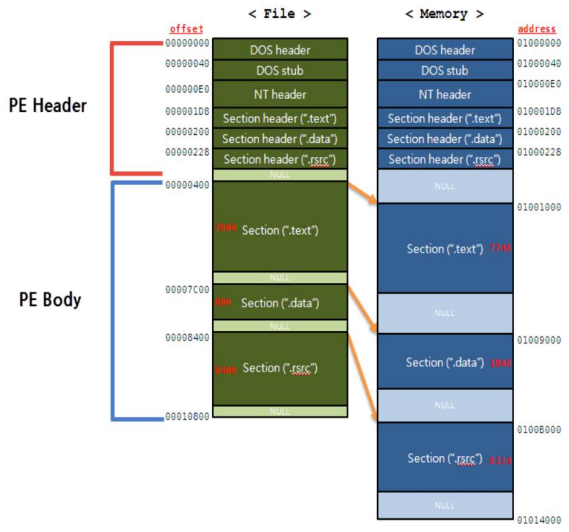

Fig. 1. PE File Structure.
그림 1. PE 파일 구조

FCG 구성을 위해서는 다음과 같은 PE 파일구조의 요소에 대한 분석이 필요하다:

- IMAGE_OPTIONAL_HEADER: 프로그램의 [Address of Entry Point]와 [Image Base] 정보를 제공한다. 이 정보들을 사용하여 실행 코드의 위치 및 함수의 시작 주소 등을 확인한다. 또한 DLL API 함수 정보를 저장하고 있는 테이블의 위치 정보를 획득할 수 있는 [Number of Data Directories] 정보를 제공한다.
- Section.rdata: 프로그램 내에서 호출되는 외부 API 정보를 저장하는 IMPORT_Address_Table, IMPORT_Directory_Table, IMPORT_Name_Table 등을 제공한다. 이 정보를 사용하여 실행 코드 내에서 DLL 함수의 호출을 확인할 수 있다.
- Section.txt: 프로그램의 바이너리 실행 코드(Binary Code)를 제공한다.

2. 함수 분석

이 절에서는 코드 내 함수의 특성을 분석하기 위하여 함수의 매개변수 분석과 호출관계 분석 방법을 설명한다. 또한, 바이너리 실행코드를 정적으로 분석하여 함수블록을 생성하는 방법을 설명한다.

가. 함수 특성 분석

(1) 함수 매개변수 분석

PE 파일을 정적으로 분석할 때, 함수의 매개변수 구성을 관찰하기 위한 방법은 다음과 같은 두 가지 방법을 고려할 수 있다.

- 경우 1: CALL 명령 수행 직전에 함수 매개변수를 스택에 저장하는 코드를 분석한다. 함수 매개변수 구조를 분석하기 위해서는 분해된 코드를 코드주소에 따라 순차적으로 분석할 때 CALL 명령이 발견되면 해당 CALL 명령의 앞부분을 코드주소 역순으로 재확인하며 스택에 매개변수를 저장하는 명령을 분석한다. 그러나 일반적으로 FCG 구성을 위해서는 분해된 코드를 코드주소에 따라 순차적으로 명령(Instruction)을 분석하기 때문에 코드주소 역순으로 이미 분석된 명령들을 재확인하는 것은 매우 비효율적이다. 또한, 매개변수를 스택에 저장하기 위해 사용되는 PUSH 명령이 매개변수 저장에만 사용되는 것이 아니기 때문에, 코드주소 역순으로 코드를 분석한 결과에 오류가 포함될 가능성이 있다. 즉, 실제 매개변수가 아님에도 불구하고 매개변수로 간주하는 오류가 발생할 수 있다.
- 경우 2: 실행코드의 함수블록들이 분석/구성되어 있다면, 함수블록 내부에서 실제로 사용되는 매개변수를 검색한다. 함수 내부에서 매개변수에 접근하기 위해서는 스택 프레임 시작주소(EBP)

를 기반으로 매개변수가 저장된 스택의 위치를 계산한다. 즉, 함수블록 내의 명령에서 사용되는 피연산자(Operands)가 [EBP+n]과 같은 형식으로 매개변수에 접근한다. 그러므로 함수블록 내에서 피연산자가 [EBP+n] 형식으로 표현된 명령을 모두 검색한다.

경우 1은 비효율적이고 오류 가능성이 있지만 스택에 저장되는 매개변수의 데이터 유형(Type)을 대략적으로 알 수 있다. 경우 2는 피연산자가 가리키는 스택에 저장 된 데이터유형을 식별할 수 없다는 단점이 있다. 본 논문에서는 FCG 생성의 효율성을 높이기 위하여 경우 2를 사용한다. 이 경우, 매개변수의 개수와 해당 변수가 포인터 변수인지 여부만을 확인할 수 있다.

(2) 함수 호출관계 분석

함수의 특성을 규정하거나 FCG 생성을 위해서는 호출자와 피호출자의 관계 분석이 필요하다. 분석하려는 함수(목표함수)의 피호출자 조사는 목표함수의 함수블록을 구성하는 명령들을 분석하는 것으로 충분하다. 그러나 코드의 특성을 FCG를 이용하여 보다 정확하게 표현하기 위해서는 API 함수 호출뿐만 아니라 코드의 내부 함수 호출관계도 분석되어야 한다. 또한, FCG 분석의 정확도를 향상시키기 위해서는 경유 호출과 같은 불필요한 호출에 사용되는 노드를 가급적 제거하고 직접 호출관계로 표현하는 것이 필요하다.

목표함수의 호출자를 분석하는 것은 다음과 같은 두 가지 경우를 모두 고려해야한다.

- 경우 1(타 함수에 의한 호출): 타 함수에 의한 목표함수 호출은 목표함수 자체를 조사해서는 분석할 수 없다. 이를 위해서는 실행 코드의 모든 함수블록을 구성한 후, 함수블록의 피호출자들과 목표함수를 비교해야 한다.
- 경우 2(재귀호출): 목표함수의 함수 호출 명령을 분석하여 목표함수를 스스로 호출하는지 조사한다.

실행코드 내에서 함수를 호출하기 위해서는 일반적으로 CALL 또는 JMP 명령을 사용한다. 다음과 같은 다섯 가지 방법이 일반적으로 관찰되었다.

- 경우 1: CALL 명령, 피연산자 = 직접주소
- 경우 2: CALL 명령, 피연산자 = 간접주소
- 경우 3: CALL 명령, 피연산자 = 레지스터 (ESI 또는 EDI)

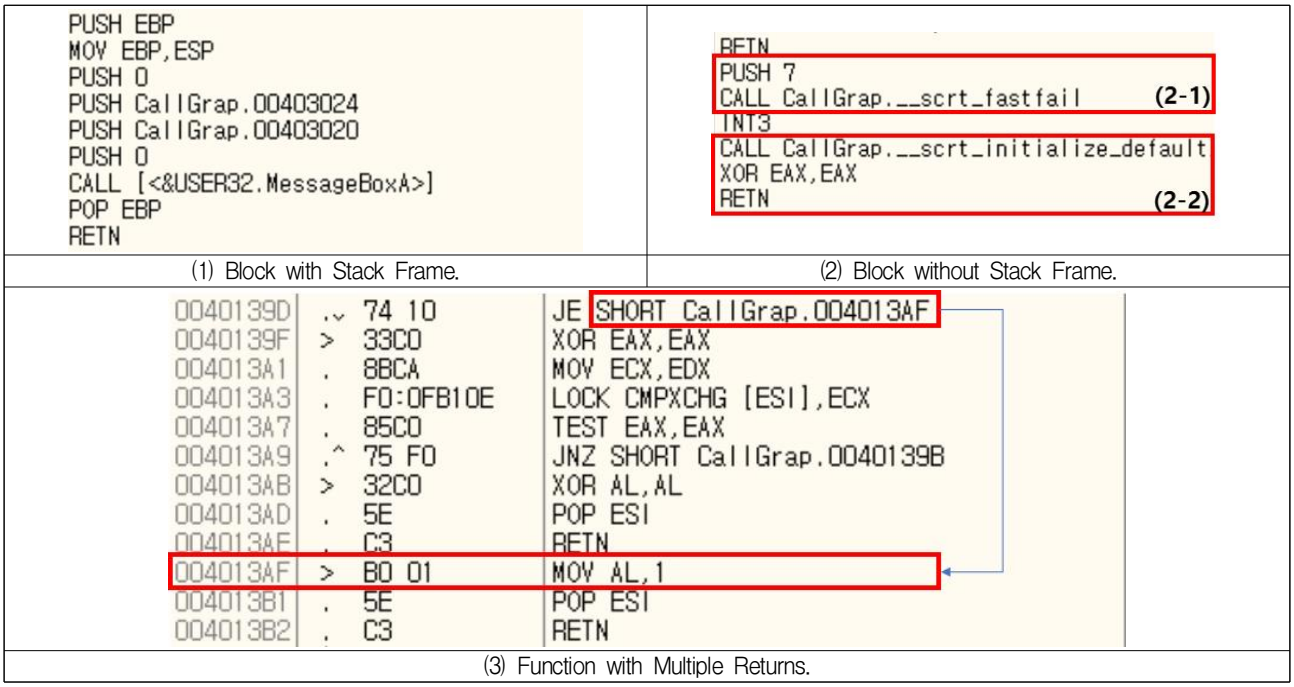

Fig. 2. Function Block Cases.
그림 2. 함수 블록의 종류

- 경우 4: bnd JMP 명령, 피연산자 = 직접주소
- 경우 5: JMP 명령, 피연산자 = 간접주소

경우 2와 경우 5의 피연산자에는 함수블록의 시작주소 값을 저장하고 있는 메모리의 주소(간접주소)가 저장된다. 특히, PE 파일구조의 IAT 정보를 활용하여 DLL 함수를 호출할 때에는 경우 5의 방법이 일반적으로 사용된다. 이 경우, 피연산자에 저장된 간접주소는 IAT 정보에 의해 구성된 코드를 가리키고 있으며, 해당 코드는 JMP 명령을 사용하여 실제 메모리에 로딩 된 DLL 함수의 시작주소로 프로그램 제어를 이동시킨다. 그러므로 경우 5를 그대로 적용하여 FCG를 구성할 경우, 실제 함수블록이 아닌 연결 (또는 경유) 경로를 그래프의 노드로 간주하여 분석에 지장을 줄 수 있다. 이와 같은 문제를 해결하기 위해서는 FCG 생성을 위해 분해된 어셈블리 코드를 분석하는 과정에서 경우 5 형식의 명령이 발견되면, 해당 명령의 피연산자에 저장된 간접주소가 가리키는 메모리의 명령을 분석하여 실제 호출되는 함수블록의 시작주소 값으로 수정하는 작업이 필요하다.

경우 3과 같이 CALL 명령의 피연산자가 특정 직/간접 주소 값이 아니라 ESI/EDI와 같은 CPU 레지스터인 경우, 분해된 실행코드를 정적으로 분석하여 CALL 명령에 의해 호출되는 함수를 특정 짓기 어렵다. 실제로 CALL 명령의 피연산자로 지정된 CPU 레지스터에 저장된 정보를 분석하여 함수블록의 시작주소 값을 획득하기 위해서는 코드주소 역순으로 코드를 재분석하여 레지스터의 값을 확인하고, 이 값이 함수블록의 주소인지 여부를 판단해야 한다. 본 논문에서는 FCG 생성 효율성을 고려하여 코드주소 역순 분석을 사용하지 않는다. 또한, 이와 같은 역순 분석을 수행할 때에도 실제 레지스터에 저장된 값이 함수블록의 시작주소인지를 확인하기 위해서는 코드 내의 함수블록이 모두 분석된 후에 가능하다.

나. 함수블록 생성

분해된 실행코드를 PE 파일의 EP(Entry Point) 값을 기준으로 해서 코드주소의 순서에 따라 차례로 분석하여 함수블록들을 생성한다. 이렇게 생성된 함수블록은 FCG의 노드에 각각 대응한다. 함수블록을 생성하기 위해서는 실행코드 내에서 함수부분을 분리해야 한다. 즉, 실행코드 내에서 함수의 시작주소와 끝주소를 모두 분석해야 한다. 이를 위하여 본 논문에서는 다음과 같은 경우들을 모두 고려한다.

(1) 스택구조(Stack Frame)를 갖는 함수블록

그림 2-(1)은 전형적인 스택구조를 이용하는 함수블록의 예를 보여준다. 일반적으로 함수 호출 시 함수블록을 구성하기 위하여 스택구조를 사용하는 경우, 함수블록은 PUSH EBP 명령과 MOV EBP, ESP 명령으로 시작된다. 또한, 함수블록의 끝은 함수 리턴 처리를 위해 RET 명령 또는 RETN 명령으로 구성된다. 실행코드를 분석하며, 이와 같은 규칙에 따라 함수블록의 시작과 끝을 판단한다.

(2) 비 스택구조를 사용하는 함수블록

그림 2-(2)와 같이 최저화를 위하여 함수가 스택구조를 활용하지 않게 컴파일 될 수 있다. 이와 같은 경우, 분해된 코드를 순차적으로 분석해서 함수블록을 구성하기 위한 별도의 분석이 필요하다. 비 함수 스택구조를 사용하는 함수블록의 구성을 위해서 함수블록의 시작주소를 판단한다.

- 경우 1: 함수블록 생성 절차를 시작하기 전에 실행코드의 함수 호출 명령들을 모두 검색하여 피호출자 리스트(Callee List)를 생성한다. 피호출자 리스트는 함수블록의 시작주소들을 저장한다. 이 후, 피호출자 리스트에 저장된 함수블록 시작주소를 이용하여 함수블록을 구성한다.

이 경우, 스택구조를 활용하는 함수와 그렇지 않은 함수를 모두 검색할 수 있다. 그러나 경우 1만을 기준으로 함수블록을 생성하면, 앞서 설명한 것처럼 함수 호출 여부만으로 실행코드내의 모든 함수를 함수블록으로 구성하기 어렵다. 그러므로 다음과 같은 추가적인 상황을 고려해야 한다.

- 경우 2: 일반적으로 함수블록은 RET 명령으로 끝나기 때문에 직전에 구성된 함수블록의 마지막 RET 명령 다음에 수행되는 명령부터 새로운 함수의 시작으로 간주한다. 그림 2-(2)의 (2-1)과 같은 경우가 이에 해당한다.
- 경우 3: INT3 명령 다음에 수행되는 명령을 함

수의 시작으로 간주한다. 그림 2-(2)의 (2-2)처럼, 일반적으로 실행코드 내의 함수 크기를 8이나 16의 배수로 맞추기 위하여 부족한 코드를 1 바이트 크기의 INT3 명령으로 패딩 처리하는 경우가 있다. 그러므로 실행코드를 분석하며 하나 또는 연속해서 복수개의 INT3 명령이 발견되면, 마지막 INT3 명령까지를 함수로 간주하고, 그 다음 명령부터 새로운 함수의 시작으로 판단한다.

(3) 복수개의 RET 명령 포함하는 함수블록

함수에 여러 개의 RET 명령이 포함된 경우, 함수블록의 끝을 설정하기 위해서는 복수개의 RET 명령들 중에서 마지막 RET 명령을 식별해야 한다. 복수개의 RET 명령이 포함되는 경우는 함수블록 내에서 조건문을 사용하여 제어 분기가 발생한 경우이다. 그러므로 본 논문에서는 함수블록 내의 JMP 명령과 같은 분기명령의 피연산자와 RET 명령이 저장된 주소 값을 비교하여 RET 명령이 해당 함수블록의 마지막 명령인지 여부를 판단한다. 예를 들어 그림 2-(3)에서 0x40139D의 분기명령인 JE 명령의 목적지 주소는 0x4013AF이다. 그러므로 0x4013AE의 RET 명령은 이전 분기명령의 목적지 주소 0x4013AF 보다 작으므로, 해당 RET 명령이 함수블록의 끝이 아니라고 판단한다. 본 논문에서는 이와 같은 제안을 효과적으로 구현하기 위하여 함수블록 시작주소부터 순차적으로 코드 분석하며, 분기명령이 파싱될 때마다 분기명령의 피연산자에 저장된 목적지 주소를 리스트에 저장한 후, RET 명령이 파싱되면 해당 리스트와 비교한다. 만약 해당 RET 명령이 저장된 주소가 해당 리스트에 있는 모든 주소들보다 뒤에 위치하면 해당 RET 명령을 함수블록의 마지막 명령으로 간주한다.

(4) RET 명령을 사용하지 않는 함수블록

그림 2-(2)의 (2-1)이나 IAT 처리를 위하여 CALL/JMP 명령의 목적지 주소로 명시된 일부 코드블록의 경우, RET 명령이 포함되어 있지 않는 코드블록들도 존재한다. 이와 같은 경우는 다음과 같은 두 가지로 분석된다.

- 경우 1: 일부 치명적인 실행오류의 경우, 오류(Event) 발생 시 오류 처리를 위한 처리기(Handler) 호출 후 실행 코드를 강제 종료한다. 함수블록 내에서 이와 같은 오류 처리기를 호출하는 경우, RET 명령이 발견되지 않는 경우가 있다. 이 경우, 본 논문에서는 앞서 설명한 것처럼 함수블록의 끝을 판단하기 위해서 함수블록의 크기를 8 또는 16의 배수로 조정하려는 컴파일러의 일반적인 특징을 이용한다. 즉, 컴파일러는 함수블록의 크기를 조정하기 위하여 INT3 명령으로 패딩 처리한다. 그러므로 RET 명령의 포함 여부와 관계없이 분해된 코드를 순차적으로 분석하는 도중 INT3 명령이 발견되면, 그 직전 명령까지를 함수블록으로 간주한다.
- 경우 2: DLL 함수 호출 과정에서와 같이 실제 함수블록을 호출하기 위해 경유하는 코드블록의 경우 RET 명령이 코드블록 내에 존재하지 않는다. 이와 같은 코드블록의 공통적인 특징은 코드블록의 마지막 명령이 다른 정상적인 함수블록을 호출하기 위하여 JMP 명령으로 구성된다. 이와 같은 경우는 다음 두 가지 경우를 구분해서 처리한다.
  (a) 앞서 호출명령 분석에서 설명한 것처럼, CALL/JMP 명령의 목적지 주소가 가리키는 코드의 명령이 JMP 명령 하나로 구성된 경우이다. 본 논문은 이 경우처럼 JMP 명령 하나로만 구성된 경유를 위한 코드블록은 함수블록으로 간주하지 않고, CALL/JMP 명령의 목적지 주소가 실제 실행되는 함수블록을 가리키도록 분해된 코드를 수정한다.
  (b) CALL/JMP 명령의 목적지 주소에 해당하는 코드블록이 두 개 이상의 명령으로 구성되어 있고, 코드블록의 마지막 명령이 JMP 명령인 경우이다. 본 논문에서는 이 경우, 해당 코드블록을 함수블록으로 간주한다. 단, 이와 같은 함수블록은 RET 명령으로 함수블록의 끝을 확인할 수 없기 때문에, 앞서 설명한 것처럼 함수블록의 크기를 조정하기 위한 패딩 정보를 바탕으로 함수 끝을 확인한다.

3. FCG 생성

FCG의 노드(Node)는 함수블록에 대응되며, 함수블록의 시작주소를 노드 식별자 (Node Identity)로 사용한다. 각 노드의 노드 값(Node Value)은 매개변

Table 1. Pseudo Code for Generating FCG.
표 1. FCG 생성 절차

INPUT	PE_file
OUTPUT	G=<N=node_list, E=edge_list>

```
//Step 1 and 2: Parsing dll function address from PE_file
    IAT[] <- Generate dll function address list from PE_file;
    binCode <- Read Section.txt from PE_file;
    assCode <- Disassembly binaryCode;
    Translate the object address of CALL/JMP instruction of assCode to RVA;
    Sort assCode according to code address;
//Step 3: Change the indirect object address of function call
    jmpList[] <- Extract the code address of JMP dword ptr []
    FOR code in assCode:
        IF the instruction of code is CALL THEN
            oa <- Read code.operand;
            IF oa is in jmpList THEN code.operand <- jmpList[oa].operand;
            callees[] <-code.operand;
//Step 4: Generate Function Block (Generate Nodes of Graph)
    FOR code in assCode:
        IF code.instruction is INT3 or code.address is in jmpList THEN continue;
        IF code is PUSH EBP or last_code is INT3 or code.address is in callees[] THEN
            b_start <- code.address;
            node_list[b_start] <- (null, null, null, null);
        IF code.instruction is related to JMP THEN
            IF code.operand > block_start THEN jmp_addresses[] <- code.operand;
        IF code.instruction is either RET or bnd RET THEN
            IF code.address > jmp_addresses[] THEN
                b_end <- code.address;
                b_size <- b_end-b_start+1;
                node_list[b_start] <- (b_end, b_size, null, null);
        last_code = code;
//Step 5: Construct both callees and callers of each function
    FOR b_start IN node_list:
        block <- Read block from b_start to node_list[b_start].b_end;
        b_callees[] <- search callees of block;
        FOR c_start IN node_list:
            c_block <- Read block from c_start to node_list[c_start].b_end;
            FOR code IN c_block(c_start, c_block[c_start]):
                IF code.instruction is in {CALL, JMP, bnd JMP} THEN
                    IF b_start is equal to code.operand THEN b_caller[] <-c_start;
        Add size of b_caller and b_callees to node_list[b_start];
//Step 6: Generate Call Relation of functions (Generate Edges of Graph)
    FOR b_start IN node_list:
        block <- Read block from b_start to node_list[b_start].b_end;
        b_callees[] <- Search callees of block;
        FOR callee IN b_callees:  edge_list[] <- (b_start, callee, null);
    Return (node_list, edge_list);
```

수의 개수, 피호출자(Callee)의 개수, 호출자(Caller)의 개수로 구성한다. FCG의 간선 (Edge)는 호출자와 피호출자의 관계로 설정한다. Table 1는 PE 파일을 입력 받아 FCG를 생성하는 절차를 설명한다.

- 단계 1: 입력된 PE file의 IAT 정보를 분석하여 DLL 함수의 코드 내에서 함수블록 시작주소를 IAT[] 리스트에 저장한다.
- 단계 2: PE_file의 Section.txt를 코드해석(Disassembly) 과정을 통하여 분석하고, CALL/JMP 명령에서 사용된 목적지 주소를 RVA 값으로 설정한다. 또한, 필요한 경우 해석된 코드를 코드의 주소에 따라 정렬한다.
- 단계 3: 해석된 코드 내에서 JMP dword ptr[] 형식으로 간접 호출되는 명령을 검색하여 목적지 주소를 직접 가리키도록 수정한다.
- 단계 4: 해석된 코드를 코드주소 순서에 따라 분석하여 함수블록을 생성한다. 함수블록 생성을 위하여 코드블록의 시작과 끝을 판단한다. 생성된 함수블록 정보는 노드(Node) 집합에 저장한다.
- 단계 5: 해석된 코드에서 분석된 함수블록마다 해당 함수를 호출하는 호출자(Caller)와 해당 함수가 호출하는 피호출자(Callee)를 확인한다.
- 단계 6: 각각의 함수블록의 코드를 주소순서에 따라 분석하여 해당 함수가 호출하는 피호출자를 확인하고 호출자-피호출자 관계 정보를 간선(Edge) 집합에 저장한다.

III. FCG 분석

FCG를 분석하기 위해서 그래프를 구성하는 노드(Node)와 간선(Edge)의 특성으로 표시된 함수블록 호출관계를 분석한다. 또한, 각 노드들의 특성 값을 분석한다. 그림 3은 샘플 코드를 앞서 제안한 Table 1 절차에 따라 분석하여 FCG를 생성한 결과이다.

1. 호출관계 분석

이 그래프는 함수블록 간 호출 관계를 표시한다. 그림 3에서 다음과 같은 특징을 갖는 노드들을 쉽게 찾을 수 있다.

- 붉은색 노드(0x401333)로 표시된 함수블록은 입력된 PE 파일의 실행 시작 함수(EP, Entry Point)로 지정된 함수블록이다. 해당 노드의 특징은 호출자가 없기 때문에 진입간선이 존재하지 않는다.
- 연두색 노드(esi, edi)로 표시된 함수블록은 CALL/JMP 명령의 목적지 주소를 저장하는 피연산자가 ESI/EDI와 같은 레지스터인 경우이다. 실제 정적분석 만으로는 CPU의 레지스터에 저장된 함수의 시작주소 값을 분석하기 어렵기 때문에 실제 호출되는 함수를 파악하기 어렵다.
- 노란색 노드(0x4010ec, 0x401197, 0x40119f, 0x

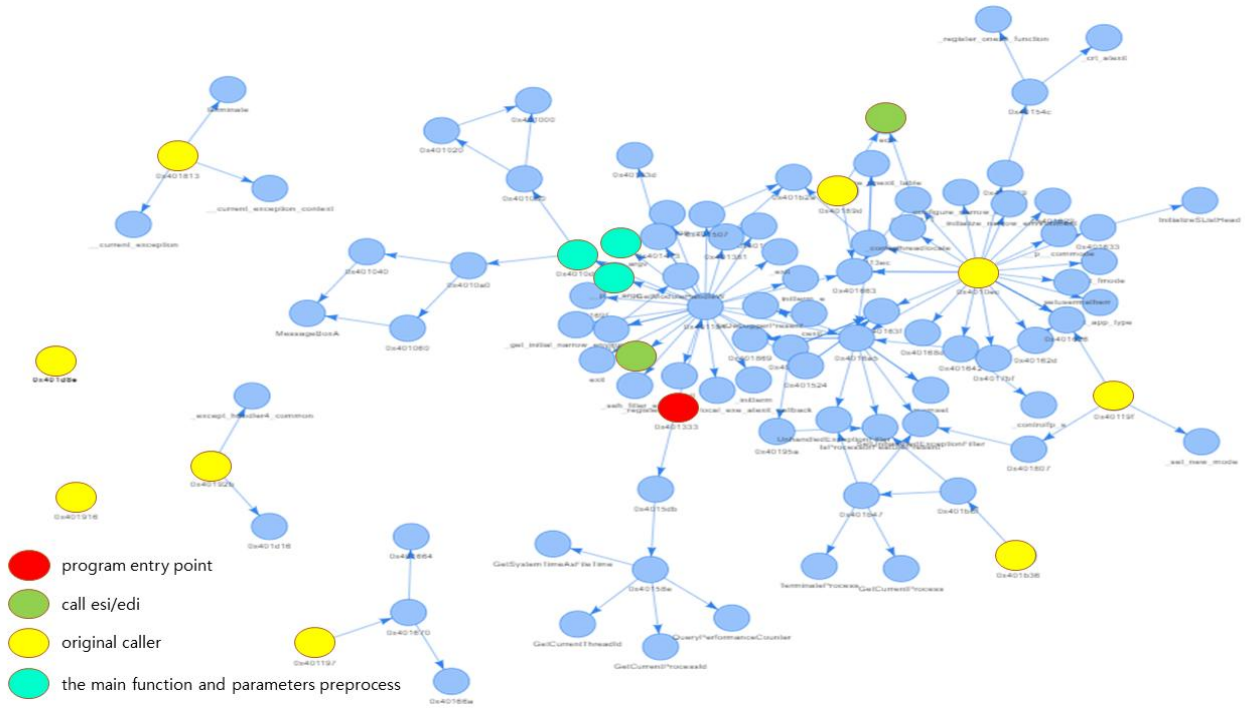

Fig. 3. FCG Generation Result and Analysis.
그림 3. FCG 생성 결과 및 분석

401813, 0x40189d, 0x40192b, 0x401b36)로 표시된 함수블록은 프로그램 코드에 포함되어 있으나 해당 함수를 호출하는 호출자가 존재하지 않는 함수블록을 의미한다. EP값으로 지정된 붉은색 노드(0x401333)도 역시 이런 노드 중 하나라 할 수 있다. 즉, 붉은색 노드와 노란색 노드들은 FCG를 생성할 때 독립된 서브 그래프로 표현이 가능하다. 단, 연두색 노드들이 결정되지 않은 상태이므로 연두색 노드가 실제로는 노란색 노드들 중 하나일 수 있기 때문에, FCG 분석을 위해서는 이와 같은 미결정 노드들을 고려할 필요가 있다. 또한 노란색 노드들 중, 간선을 하나도 포함하지 않는 노드가 존재한다. 0x401916과 0x401d8e가 이와 같은 노드들이다. 이와 같은 노드들에 대응하는 함수블록의 특징은 정적 코드 분석만으로는 호출자와 피호출자가 발견되지 않는다는 것이다.

- 하늘색 노드는 메인함수(0x4010D0) 호출과 관련된 노드들이다. 프로그램의 실제 실행과 직접적인 관련이 있기 때문에 프로그램의 주된 동작을 관찰할 수 있는 서브 그래프를 추출할 때 유용하게 활용될 수 있다. 메인함수는 붉은색 노드의 호출 경로를 분석하면 찾을 수 있다. 실제 C로 구현된 코드의 경우, 메인 함수가 호출되기 전에 메인 함수의 매개변수인 argv와 argc를 처리하는 절차가 선행된다. 이를 위하여 코드는 CALL __p_argv와 CALL __p_argc 명령을 사용하며, 피연자자인 __p_argv와 __p_argc는 IAT 정보에서 확인할 수 있다. 그러므로 이 두 명령을 수행한 후, 호출되는 함수가 메인함수이다.

그림 4는 함수블록의 추가 속성을 표시하기 위해 함수 호출을 트리 형태로 표현한 것이다. 생성된 그래프의 함수 호출 관계를 파악하면 그림 4와 같이 함수블록 내에서 동일한 함수가 복수 번 호출되는 경우가 발견된다. 본 논문에서는 이와 같이 함수 호출 회수를 간선의 가중치로 부여하여 사용할 수 있다. 그러나 명시적으로 호출된 회수만을 가중치로 사용하기 때문에 반복문을 사용한 다중 호출의 경우는 추가적인 분석 절차가 필요하다.

2. 노드 특성 분석

본 논문에서는 앞서 설명한 것처럼 그래프의 각

```
└ 0x4011b1() // 385 bytes, caller(1), callee(24)
    └ 0x4018d0() // 70 bytes, caller(2), callee(0)
    └ 0x4013b3(a) // 57 bytes, caller(1), callee(2)
        └ 0x40195a() // 464 bytes, caller(1), callee(1)
            └ IsProcessorFeaturePresent
        └ 0x40163f() // 3 bytes, caller(3), callee(0)
        └ 0x40163f() // 3 bytes, caller(3), callee(0)
        └ 0x40163f() // 3 bytes, caller(3), callee(0)
```

```
└ 0x401473(a) // 148 bytes, caller(1), callee(2)
    └ 0x4018d0() // 70 bytes, caller(2), callee(0)
    └ 0x40133d(a, b) // 68 bytes, caller(1), callee(0)
```

Fig. 4. Function Call Sub-Tree.
그림 4. 함수 호출 부분트리

노드는 매개변수의 개수, 함수블록의 크기, 호출자의 개수, 피호출자의 개수를 노드 값으로 관리한다. 노드의 특성을 추가함으로써 그래프를 분석하고 비교할 때 정확성을 높일 수 있을 것으로 기대한다. 그림 4는 함수 호출 트리의 일부이다. 그림 4의 각 라인은 다음과 같이 표시 한다: 노드 식별자 (매개변수의 개수 정보) // 블록의 크기, 호출자 개수, 피호출자 개수. 그림 4에서 0x401473 노드의 경우 매개변수의 개수가 1개이므로 (a)로 표시되었고, 0x40133d는 매개변수가 2개 이므로 (a, b)로 표시되었다. FCF 분석만으로 FCG를 구성하면 이와 같은 노드 특성 값을 분석하기는 어렵다. 본 논문에서는 코드 내의 모든 함수블록을 선행분석한 후, 분석된 함수블록들을 사용하여 FCF를 분석했기 때문에 가능하다.

본 논문에서 생성한 FCG는 프로그램에서 사용하는 주요 API 호출 관계뿐만 아니라 코드 내의 모든 함수들을 노드로 표현할 수 있다. 이 경우, 발생할 수 있는 그래프의 복잡성을 개선하기 위하여 간접호출 코드를 검색한 후, 직접호출로 검색된 코드를 수정한 결과를 바탕으로 FCG를 생성했다. 그 결과 DLL 함수를 호출하기 위하여 경유하는 코드를 피호출자로 처리하는 경우, 예제 코드에서는 114개의 함수블록이 생성되는 반면에 본 논문에서 제안한 것처럼 함수 호출 시 함수블록의 주소를 직접 가리키도록 수정하여 적용하면 83개의 함수블록만으로 그래프를 축소시킬 수 있다. 이는 생성된 그래프의 의미를 왜곡시키지 않고 그 구조를 보다 간결하게 표현함으로써 분석의 효율성과 정확성을 높일 수 있다.

표 2는 샘플 코드의 함수 호출 시퀀스/그래프 생

Table 2. Generated Graph Characteristics Comparison.
표 2. 생성 그래프 특성 비교

Scheme	Source	D(Node)	D(Edge)	N(Node)	N(Edge)	LOOK UP
[3]	binary code	API	Fuction Call	13	21	CAN NOT
[14]	binary code	API	Fuction Call	41	52	CAN NOT
[15]	program code	API/API pair	Fuction Call	70	8	N.A
Pro_1	binary code	Code Block	Fuction Call	114	136	CAN NOT
Pro_2	binary code	Code Block	Fuction Call	83	105	CAN

성 결과를 비교한 것이다. D(*)는 노드와 간선의 정의를 나타내고, N(*)는 노드와 간선의 수를 나타낸다. [3]과 [14]은 주요 API 함수 호출관계만을 그래프로 표현하고 있기 때문에 노드와 간선의 수가 비교적 작다. 반면에 [15]는 주요 API 간 호출관계를 분석하기 때문에 노드의 수는 많으나 간선의 수가 매우 적다. [3]과 [14]의 경우, API 함수 호출 주소를 테이블로 구성하고, 이를 검색하여 호출하도록 수정할 경우, 그래프를 정확하게 구성하지 못하는 단점이 있다. Pro_1과 Pro_2는 본 논문에서 제안한 절차를 적용한 결과이다. Pro_1은 경유경로 코드를 노드로 간주한 것이고, Pro_2는 경유경로를 분석하여 직접 호출로 수정한 것이다. Pro_1의 경우, [3]/[14]과 같이 함수 호출 주소를 테이블로 구성하면, 그래프의 노드와 간선이 증가하고 정확하게 구성되지 못했다. 그러나 Pro_2는 경유경로 분석을 통하여 이와 같은 단점을 해결하였다.

Ⅳ. 결론

악성코드 탐지하기 위해 실행코드에 대한 FCG를 활용하는 방안이 지속적으로 제안되고 있다. 실제로 기존 악성코드의 시그니처 비교탐지와 같은 기술을 우회하기 위한 코드 위/변조 기술의 경우, 전형적으로 코드의 일부분을 수정하거나 불필요한 코드를 추가한다. 그러나 FCG의 유사도 측정을 우회하기 위해서는 프로그램 코드의 구조를 전반적으로 수정해야만 한다. 즉, 전형적인 위/변조 방법보다 많은 비용이 요구된다. 그러므로 FCG 기반의 악성코드 분석 및 탐지는 효과적인 대응 방안 중 하나라 할 수 있다. 그러나 폭발적으로 증가하는 악성코드에 대응하고, 보다 정확한 분석을 위해서는 효율적이고 정확한 FCG 생성이 요구된다.

본 논문에서는 윈도우즈 시스템의 PE 파일구조를 갖는 실행파일을 분석하여 함수 호출자와 피호출자의 관계를 그래프로 표현하기 위한 효과적인 방법을 제안한다. 본 논문에서 제안하는 절차는 다음과 같은 의미를 갖는다.

첫째, 기존에 함수 호출 관계 위주의 그래프 생성 기술들이 코드 내의 모든 함수를 포함하지 못하는 문제점을 해결하기 위하여 본 논문에서는 프로그램 코드를 정적으로 분석하여 코드 내의 모든 함수를 찾아 블록으로 구성한 후, 함수 호출 관계를 적용시킴으로써 분석 시 제외된 함수가 없도록 하였다.

둘째, 기존 FCG 기술들이 프로그램의 주요 API 함수 위주로 그래프나 시퀀스를 생성한 반면에, 본 논문에서는 코드 내의 모든 함수를 포함하는 그래프를 제공함으로써 프로그램의 전체적인 양상을 분석할 수 있게 하였다.

셋째, 본 논문에서는 프로그램 내의 함수 호출 관계뿐만 아니라 각 함수의 특성을 노드 특성 값으로 활용하여, 보다 심도 있는 FCG 분석이 가능하도록 하였다.

넷째, 본 논문에서는 노드의 호출 특성에 따라 서브 그래프를 생성할 수 있는 방법을 제안하였으며, 특히 프로그램의 실제 주된 동작과 관련된 서브 그래프를 지정할 수 있는 방법을 제안하였다.

본 논문에서 제안된 FCG 생성 기법을 보다 정밀하게 개선하기 위해서는 함수 호출 시 ESI/EDI와 같은 레지스터를 사용하는 경우에 대한 처리와 함수 내에서 반복문을 사용하여 동일 함수를 여러 번 호출하는 경우, 그리고 멀티 쓰레드/프로세스를 생성하는 경우들에 대한 추가 연구가 필요하다. 향후 FCG 생성 기법을 활용하여 악성코드 샘플 데이터를 그래프화 한 후, 이를 GNN을 활용하여 학습하고 악성코드 탐지에 적용하는 연구의 진행이 필요하다.

References

[1] "Malware hidden site detection trend report in the second half of 2020," *Korea Internet & Security Agency*, 2021. online: https://www.boho.or.kr/data/reportView.do?bulletin_writing_sequence=35872

[2] M. Singh and S. Kim, "Security analysis of intelligent vehicles: Challenges and scope," *2017 International SoC Design Conference (ISOCC)*, pp.5-8 2017. DOI: 10.1109/ISOCC.2017.8368805

[3] E. Amer and I. Zelinka, "A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence," *Computers & Security,* 2020.
DOI: 10.1016/j.cose.2020.101760

[4] A. Ahmed, E. Elhadi1, M. A. Maarof1 and B. I. A. Barry, "Improving the Detection of Malware Behaviour Using Simplified Data Dependent API Call Graph," International Journal of Security and Its Applications, vol.7, no.5, pp.29-42, 2013.
DOI: 10.14257/ijsia.2013.7.5.03

[5] Lee, Taejin, "Trends in intelligent malware analysis technology using machine learning," *Korea Institute of Information Security and Cryptology,* Vol.28, No.2, pp.12-19, 2018.

[6] "Malware characteristic information for using artificial intelligence technology," *Korea Internet & Security Agency*, 2021. online: https://krcert.or.kr/data/reportView.do?bulletin_writing_sequence=36076

[7] P. Deshpande, "Metamorphic Detection Using Function Call Graph Analysis," *Master's Theses and Graduate Research*, 2013, online: https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1334&context=etd_projects

[8] D. Rajeswaran, "Function Call Graph Score for Malware Detection," Master's Theses and Graduate Research, 2015, online: https://core.ac.uk/download/pdf/70424797.pdf

[9] D. Rajeswaran, F. D. Troia, T. H. Austin and M. Stamp, "Function Call Graphs Versus Machine Learning for Malware Detection," *In book: Guide to Vulnerability Analysis for Computer Networks and Systems,* pp.259-279, 2018.
DOI: 10.1007/978-3-319-92624-7_11

[10] J. Bai ,Q. Shi, and S. Mu, "A Malware and Variant Detection Method Using Function Call Graph Isomorphism," Security and Communication Networks, vol.2019. 2019. DOI: 10.1155/2019/1043794

[11] Z. Liu and J. Zhou, "Introducation to Graph Neural Networks," *Morgan & Claypool Publishers,* 2020.

[12] M. Caia, Y. Jiangab, C. Gaoa, H. Lia, and W. Yuan, "Learning features from enhanced function call graphs for Android malware detection," *Neurocomputing,* vol.423, pp.301-307, 2021.
DOI: 10.1016/j.neucom.2020.10.054

[13] T. Toma and M Islam, "An efficient mechanism of generating call graph for JavaScript using dynamic analysis in web application," *International Conference on Informatics, Electronics & Vision (ICIEV)*, 2014. DOI: 10.1109/ICIEV.2014.6850807

[14] D. Andriesse, X. Chen, V. Veen, A. Slowinska, and H. Bos, "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries," *the Proceedings of the 25th USENIX Security Symposium,* pp.583-600, 2016. DOI: 10.5555/3241094.3241140

[15] S. Yang, S. Li, W. Chen, and Y. LIU, "A Real-Time and adaptive-Learning Malware Detection Method Based On API-Pair Graph," *IEEE Access,* vol.8, pp.120-135, 2020.
DOI: 10.1109/ACCESS.2020.3038453

BIOGRAPHY

DaeYoub Kim (Member)

1994 : BS degree in Math., Korea University.
1997 : MS degree in Math., Korea University.
2000 : PhD degree in Math., Korea University.
2000~2002 : Research Engineer, SECUI
2002~2012 : Senior Researcher and Project Manager, Samsung Electronics.
2012~ : Professor, Dept. of Information Security, Suwon Univ.