ISSN : 1226-7244 (Print)
ISSN : 2288-243X (Online)
논문번호 23-01-14
j.inst.Korean.electr.electron.eng.Vol.27,No.1,93~102,March 2023
https://doi.org/10.7471/ikeee.2023.27.1.93

동적 분석을 이용한 난독화 된 실행 프로그램의 함수 호출 그래프 생성 연구

The Generation of the Function Calls Graph of an Obfuscated Execution Program Using Dynamic

천 세 범*, 김 대 엽**★

Se-Beom Cheon*, DaeYoub Kim**★

Abstract

As one of the techniques for analyzing malicious code, techniques creating a sequence or a graph of function call relationships in an executable program and then analyzing the result are proposed. Such methods generally study function calling in the executable program code through static analysis and organize function call relationships into a sequence or a graph. However, in the case of an obfuscated executable program, it is difficult to analyze the function call relationship only with static analysis because the structure/content of the executable program file is different from the standard structure/content. In this paper, we propose a dynamic analysis method to analyze the function call relationship of an obfuscated execution program. We suggest constructing a function call relationship as a graph using the proposed technique.

요 약

악성코드 분석을 위한 기술 중 하나로 실행 프로그램의 함수 호출 관계를 시퀀스 또는 그래프 작성한 후, 그 결과를 분석하는 기술이 제안되었다. 이러한 기술들은 일반적으로 실행 프로그램 파일의 정적 분석을 통해 함수 호출 코드를 분석하고, 함수 호출 관계를 시퀀스 또는 그래프로 정리한다. 그러나 난독화 된 실행 프로그램의 경우, 실행 프로그램 파일의 구성이 표준구성과 다르기 때문에 정적분석 만으로는 함수 호출관계를 명확히 분석하기 어렵다. 본 논문에서는 난독화 된 실행 프로그램의 함수 호출관계를 분석하기 위한 동적 분석 방법을 제안하고, 제안된 기술을 이용하여 함수 호출관계를 그래프로 구성하는 방법을 제안한다.

Key words : Malware, Static analysis, Dynamic analysis, SW Packing, Function Call Graph

Ⅰ. 서론

지능화된 랜섬웨어와 같은 악성코드의 피해는 매년 증가하는 추세이며, 다양한 IoT(Internet of Things) 기기들과 스마트 차량(Connected Vehicle)이 새로운 공격 대상이 될 경우, 그 피해는 더욱 심각해질 것으로 예상된다[1]. 악성코드를 신속하게 탐지하고 효율적으로 대응하기 위한 연구는 지속적으로 진행되어 왔으나 기존의 블

* Student, Dept. of Information Security, Suwon University
** Professor, Dept. of Information Security, Suwon University
★ Corresponding author
E-mail : daeyoub69@suwon.ac.kr, Tel : +82-31-229-8284
※ 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 Nov. 24, 2022; revised Dec. 14, 2022; accepted Mar. 20, 2023.

랙리스트 기반의 코드 시그니처(Signature) 분석/탐지와 같은 전통적인 기술은 빠르게 진화하는 악성코드에 신속하게 대응하기 어렵다는 한계를 갖고 있다[2-4]. 예를 들어, 블랙리스트 기반의 악성코드 탐지 기술을 사용하여 신종 악성코드를 탐지하는 경우, 해당 악성코드로 인한 피해자가 발생한 후 코드 정보가 수집/분석되어야 대응이 가능하다. 스마트 시티 또는 스마트 차량의 경우, 이와 같은 피해는 대규모 참사로 이어질 수 있기 때문에 효율적인 사전 탐지 기술이 반드시 요구된다. 이러한 전통적인 사후 탐지 기술의 한계를 극복하기 위하여 머신러닝을 활용한 API 기반 악성코드 탐지 모델 연구, 행위 기반 랜섬웨어 탐지 연구와 같이 악성코드가 갖고 있는 일반적인 특성을 분석하고 정보화 하여 신종 악성코드를 사전에 탐지하려는 연구가 진행되고 있다[5-8]. 또한, 최근에는 악성코드에서 발생하는 함수 호출의 특성을 분석하고 탐지에 활용하려는 시도가 주목 받고 있다[9-11]. 실행파일을 정적으로 분석하여 함수 호출 관계를 그래프로 표현하고, 그 특성을 분석하는 연구인 GCG(Generating Call Graph for PE file)도 그 중 하나이다[12].

악의적인 목적으로 악성코드를 배포하는 해커들은 자신들이 개발/배포한 악성코드가 분석되는 것을 방해하기 위해 종종 실행파일을 압축하거나 인코딩/암호화 기술을 이용하여 난독화를 수행해서 배포한다. 이와 같은 난독화 기법을 패킹(Packing)이라고 하며, 난독화 된 프로그램을 해독하여 실행 코드를 추출하는 기법을 언패킹(Unpacking)이라고 한다. 이러한 패킹 기술은 악성코드 분석을 방해할 목적으로 이용되고 있으며, 실제로 표 1에서 볼 수 있는 것처럼 온라인을 통해 배포되는 악성코드의 30% 정도가 패킹 되어 배포되고 있다. GCG를 사용하여 패킹 된 실행 파일을 분석 할 경우, 정적 분석만으로는 정확한 분석의 한계가 있다. 예를 들어, 윈도우즈 운영체제에서 사용되는 실행 파일구조인 PE 파일구조에는 프로그램이 실행 중 동적 연결 라이브러리를 통해 호출하는 함수 정보를 저장하는 IAT(Import_Address_Table)가 존재한다. 특정 패킹 프로그램을 사용하여 패킹 된 실행파일의 경우, IAT가 압축되어 원시 실행파일의 IAT와 큰 차이를 보인다. 또한, 실행파일의 최초 실행 메모리 주소인 EP(Entry Point)의 경우, 패킹 된 실행파일은 언패킹 작업을 우선 수행하기 때문에 패킹 된 실행파일에서의 EP는 언패킹의 메모리 시작주소를 가리킨다. 그러므로 패킹 된 실행파일의 EP는 패킹되지 않은 원시 실행파일의 EP(OEP, Original EP)와 서로 다른 주소를 가리킨다. 그러므로 패킹 된 실행파일을 그대로 정적분석 할 경우, 언패킹 이후의 원시 프로그램 데이터를 획득할 수 없으므로 원하는 그래프를 획득할 수 없다.

이와 같은 정적분석 기반의 GCG의 문제점을 해결하기 위하여 본 논문에서는 실행 파일을 동적으로 분석하여 원시 프로그램의 데이터를 확보하고, 이 정도를 바탕으로 함수의 호출 관계를 그래프로 표현해내는 Improved GCG(iGCG)를 제안한다. 악성코드의 특성 상 동적 분석 시 발생할 수 있는 피해를 방지하기 위하여 본 논문에서는 VMware 가상환경을 사용하여 패키징 된 실행코드를 실행한 후, 실행중인 프로그램의 메모리 정보를 확보하였다. 본 논문에서 제안하는 방법은 패킹 된 실행 파일이 실행되어 언패킹 과정을 완료한 이후의 정보를 분석하는 것이기 때문에 패커의 종류에 관계없이 유사하게 적용이 가능하다.

본 논문은 다음과 같이 구성되어 있다. 2절은 GCG와 패킹 된 실행 파일에 대해 설명한다. 3절은 패킹 된 실행파일을 동적 분석하고 iGCG를 이용하여 그래프를 표현해내는 방법에 대해 설명한다. 4절은 iGCG에 대한 분석 및 결과를 설명한다. 마지막으로 5절은 본 논문의 결론에 대해 설명한다.

II. PE 파일 구조 및 GCG 개요

2.1 PE 파일구조

PE(Portable Executable) 파일구조는 윈도우즈 운영시스템에서 프로그램을 실행하기 위해 필요한 정보를 저장하기 위한 데이터 구조이다. 그림 1은 PE 파일구조를 보여준다. GCG에서 PE 파일구조를 분석하여 그래프로

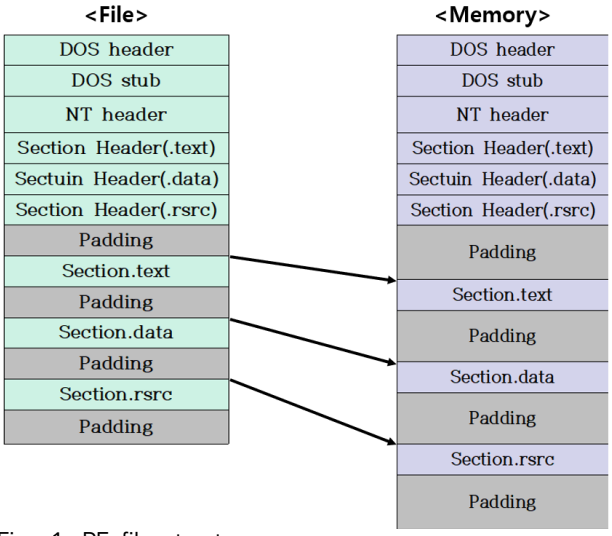


Fig. 1. PE file structure.
그림 1. PE 파일구조

표현하기 위해 필요한 정보는 다음과 같다.

- Import_Address_Table : 프로그램이 실행하면서 동적 연결 라이브러리를 통해 호출하는 함수들이 저장된 구조체이다.
- 바이너리 실행 코드 : PE 파일구조에는 프로그램의 실행 코드가 바이너리 형태로 저장되어있다. 일반적인 PE 파일구조에서 Section.txt 영역에 존재한다.
- EP : 프로그램이 실행되는 메모리 시작 주소 값이다. 일반적인 PE 파일구조에서 NT Header의 IMAGE_OPTIONAL_HEADER 구조체에 저장된다.

2.2 GCG 개요

GCG는 PE 파일구조를 정적으로 분석하여 프로그램의 함수 호출 관계를 그래프로 표현한다[12]. PE 파일구조를 분석하여 획득한 정보들을 기반으로 함수블록을 생성하고, 각 함수블록들의 호출 관계를 그래프로 표현한다. 함수블록은 일반적으로 다음과 같은 네 가지형태로 구성된다.

(1) 스택구조를 갖는 함수블록

함수블록이 PUSH EBP 명령과 MOV EBP, ESP 명령으로 시작되고, 함수의 리턴 처리를 위해 RET 또는 RETN 명령으로 끝이 구성된다.

(2) 스택구조를 갖지 않는 함수블록

최적화를 위하여 함수가 스택구조를 활용하지 않는 경우가 존재한다. 이 경우, 함수의 끝은 스택구조를 갖는 함수블록과 동일하지만, 함수의 시작을 판별하기 위해서는 분해된 코드를 다음 세 가지 방법으로 분석하여 함수블록의 시작 부분을 찾는다.

- 방법 1 : 함수블록 생성 절차를 시작하기 전에 실행코드의 함수 호출 명령들을 모두 검색하여 피호출자 리스트(Callee List)를 생성한다. 피호출자 리스트에는 함수블록의 시작주소들이 저장된다. 이 후, 피호출자 리스트에 저장된 정보를 이용하여 함수블록의 시작주소를 판별할 수 있다. 그러나 이 경우만을 사용하면, 스택구조를 사용하는 함수와 그렇지 않은 함수를 모두 검색할 수 있기 때문에 실행코드내의 모든 함수를 함수블록으로 구성하기 어려운 문제가 존재한다. 그러므로 다음과 같은 추가적인 경우에 대해서도 고려해야 한다.
- 방법 2 : 일반적으로 함수블록은 RET 또는 RETN 명령으로 끝나기 때문에 이전에 구성된 함수블록의 끝을 가리키는 RET 또는 RETN 명령 다음에 수행되는 명령부터 새로운 함수의 시작으로 간주한다.
- 방법 3 : 일반적으로 실행코드 내의 함수 크기를 8이나 16의 배수로 맞추기 위해 부족한 코드를 1바이트 크기의 INT3으로 패딩 처리하는 경우가 존재한다. 그러므로 하나 또는 연속해서 INT3 명령이 발견되면, 마지막 INT3의 다음 명령부터 새로운 함수의 시작으로 간주한다.

(3) 여러 개의 RET 명령을 갖는 함수블록

함수블록 내에서 조건문으로 분기가 발생한 경우 여러 개의 RET명령을 포함할 수 있다. 이 경우, 함수의 끝을 판별하기 위해 마지막 RET 명령을 식별해야 한다. GCG는 함수블록 내의 JMP 같은 분기명령의 피연산자와 RET 명령이 저장된 주소 값을 비교하여 RET 명령이 해당 함수블록의 마지막 명령인지를 식별하여 함수의 끝을 판별한다.

(4) RET 명령이 없는 함수블록

CALL/JMP 명령의 목적지 주소로 명시된 일부 함수블록의 경우, RET 명령이 포함되어 있지 않는 경우도 존재한다. 이와 같은 경우를 위해 다음과 같은 두 가지의 분석 방법이 존재한다.

- 함수블록 내에서 오류 처리기를 호출할 때 RET 명령이 존재하지 않을 수 있다. 이 경우, 함수블록의 크기를 8 또는 16의 배수로 맞추려는 INT3 패딩 방식이 이용된다. 즉, RET 명령의 포함 여부와 관계없이 분해된 코드를 분석하는 도중 INT3 명령이 발견되면, 그 직전 명령까지를 함수블록으로 판별한다.
- 실제 함수블록을 호출하기 위해 경유하는 코드블록의 경우 코드블록 내에 RET 명령이 존재하지 않는다. 이와 같은 코드블록은 정상적인 함수블록의 호출을 위해 JMP 명령으로 구성된다. 이와 같은 경우, 다음과 같은 두 가지 경우를 구분하여 함수블록을 결정한다. 첫 번째는 CALL/JMP 명령의 목적지 주소가 가리키는 코드의 명령이 JMP 명령 하나로 구성된 경우이다. 이 경우, 해당 코드블록을 함수블록으로 간주하지 않고, CALL/JMP 명령어의 목적지 주소가 실제 실행되는 함수블록을 가리키도록 코드를 수정한다. 두 번째는 CALL/JMP 명령의 목적지 주소에 해당하는 코드블록이 두 개 이상의 명령으로 구성되어 있고, 코드블록의 마지막 명령이 JMP 명령인 경우이다. 이 경우, GCG에서는 해당 코드블록

을 함수블록으로 간주하고 함수의 끝을 INT3 패딩 정보로 판별한다.

분석된 함수블록의 정보를 노드(Node) 집합에 저장한 후, 분석된 함수블록마다 함수를 호출하는 호출자(Caller)와 함수가 호출하는 피호출자(Callee)를 분별한다. 함수블록의 코드를 메모리 주소순서에 따라 분석하여 호출되는 피호출자를 확인하고 호출자와 피호출자의 관계를 방향간선(Edge) 집합에 저장하여 최종 그래프를 생성한다.

그림 2는 GCG를 이용하여 실행 프로그램을 그래프로 표현한 예를 보여준다. 함수블록들을 노드로 설정하고, 함수블록들의 호출 관계를 방향간선으로 설정하여 그래프로 표현했기 때문에 호출자와 피호출자의 관계가 명확하다. 예를 들어 그림 2의 0×401396은 MessageBo×A 함수를 호출하고 있으므로 해당 프로그램에서 MessageBo×A 함수를 사용함을 판단할 수 있다.

2.3 GCG의 문제점

GCG를 사용하여 함수 호출 관계를 그래프로 생성하는 경우, 다음과 같은 두 가지 문제점이 발견된다.

(1) GCG를 사용하여 그래프를 생성하기 위해서는 PE 파일구조를 정적으로 분석하여 함수 호출과 관련된 정확한 정보를 획득해야만 한다. 그러나 패킹 된 프로그램의 경우, PE 파일구조를 정적으로 분석하여 획득한 정보는 패킹 된 프로그램과 언패킹을 위한 프로그램 정보이므로 실제 실행 프로그램의 정확한 정보를 확보할 수 없다. 그러므로 패킹 된 프로그램을 언패킹한 후에 사용되는 함수 호출 관계에 대해서는 분석할 수 없기 때문에 정확한 그래프를 생성할 수 없다.

(2) GCG는 표준적인 PE 파일 구조를 따르는 프로그램에 대해서만 정확하게 작동한다. 예를 들어, C#으로 작성된 닷넷(.NET) 프레임워크 기반 프로그램의 경우, 프로그램의 어셈블리를 메모리에 로드하기 위한 API (Assembly.Load)가 닷넷 메커니즘 내에서 사용된다[13]. 프로그램에서 동적 연결 라이브러리를 통해 호출하는 함수들 또한 Assembly.Load가 실행 코드를 어셈블리로 변환하면서 메모리에 로드할 때 같이 로드되기 때문에 정적인 PE 파일 분석으로 획득할 수 있는 데이터가 제한적이다.

이와 같이 실행 프로그램이 전통적인 PE 파일 구조를 따르지 않고, 실행 코드가 별도의 프로세스를 거쳐 메모리에 로드되는 경우, GCG를 이용하여 함수 호출 관계를 그래프로 표현할 수 없다.

Ⅲ. 패킹 된 PE 파일의 동적 분석

본 절에서는 앞 절에서 언급한 GCG를 사용하여 그래프를 생성할 때 발생하는 문제점 중 첫 번 째 문제점에 대한 해결 방안을 제안한다.

3.1 패킹 된 PE 파일 구조

패킹 된 프로그램의 PE 파일구조는 패킹되기 전 원시 프로그램의 PE 파일구조와 매우 다르다. 다양한 종류의 패커가 존재하지만 본 논문에서는 UPX 패커를 기준으로 설명한다. 이 경우, 크게 다음과 같은 세 가지 차이점을 발견할 수 있다.

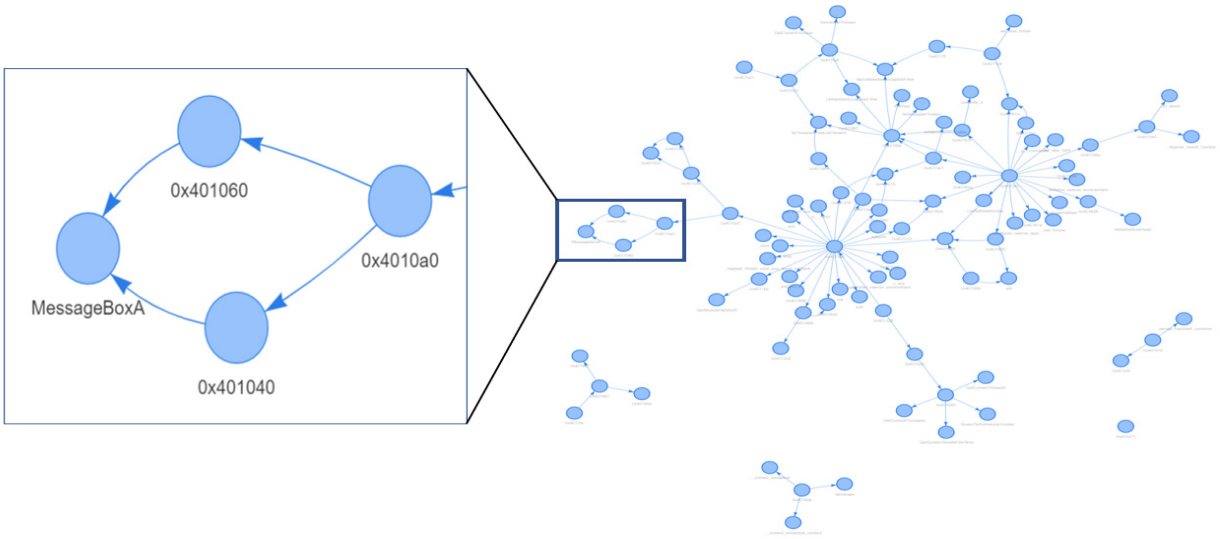


Fig. 2. Result of GCG.
그림 2. GCG 결과

(1) EP

그림 3은 동일한 프로그램에 대해 패킹되기 전과 후의 EP 비교 결과이다. 그림 3-(a)는 Address of Entry Point의 Data가 각각 패킹되기 전은 0×1336, 패킹 된 후는 0×6010임을 알 수 있다. 패킹되기 전의 0×1336은 원시 프로그램의 정확한 EP를 가리키지만, 패킹 된 후의 0×6010은 언패킹 작업을 위한 EP를 가리키고 있다. 이 경우, GCG를 이용하기 위해서는 언패킹 작업이 종료된 후 존재하는 원시 프로그램의 OEP를 찾아야 한다.

(2) IAT

그림 3-(b)와 (c)는 동일한 프로그램의 패킹되기 전과 후의 IAT 비교 결과이다. 원시 프로그램의 IAT는 프로그램이 실행 중 동적 연결 라이브러리를 통해 호출하는 함수를 포함하고 있다. 하지만 패킹 된 후의 IAT는 언패킹 작업을 위한 동적 연결 라이브러리 정보위주로 구성되어 있으므로 원시 프로그램의 실제 동적 연결 라이브러리의 개수보다 일반적으로 매우 적다. 패킹 된 프로그램을 실행하면, 언패킹 작업을 하면서 반복문을 통해 기존의 IAT를 복구한다. 그러므로 GCG를 이용하여 그래프를 표현하기 위해서는 언패킹 작업이 종료된 후 복구된 IAT를 분석해야 된다.

(3) 바이너리 코드

그림 3-(d)는 패킹 된 후의 바이너리 실행코드 비교 결과이다. 패킹되기 전의 PE 파일구조에서는 Section.txt 영역에 프로그램의 실행코드가 존재한다. 그러나 패킹 된 후의 PE 파일구조에서는 Section.txt 영역에 있는 Section.UPX0 영역에 어떠한 데이터도 존재하지 않는다. UPX 패커의 경우, 복호화 작업이 끝나면 반복문을 통해 기존 프로그램의 실행코드를 UPX0 영역에 적재한다. 그러므로 패킹 된 PE 파일구조를 정적으로 분석하면, 실행코드가 언패킹 되어 적재되기 전이므로 UPX0 영역은 빈 공간으로 존재하게 된다.

이와 같이 패킹 된 프로그램을 정적으로 분석하여 획득할 수 있는 정보에는 한계가 있다. 또한, 함수블록을 만들기 위해 필요한 실행 코드 영역이 빈 공간이므로 함수블록을 생성하는 단계에서 문제가 발생한다. 이 경우, GCG가 함수블록을 생성하기 위해 EP 값을 이용하여 정렬하는 과정에서 EP 값에 해당하는 데이터가 없기 때문에 오류가 발생한다. 만일 해당 오류가 발생하지 않아 그래프가 생성되어도 생성된 그래프는 패킹 된 프로그램의 IAT를 기반으로 생성하기 때문에 정확한 그래프라고 할 수 없다.

pFile	Data	Description
00000108	010B	Magic
0000010A	0E	Major Linker Version
0000010B	1D	Minor Linker Version
0000010C	00000E00	Size of Code
00000110	00001200	Size of Initialized Data
00000114	00000000	Size of Uninitialized Data
00000118	00001336	Address of Entry Point

pFile	Data	Description
00000108	010B	Magic
0000010A	0E	Major Linker Version
0000010B	1D	Minor Linker Version
0000010C	00002000	Size of Code
00000110	00001000	Size of Initialized Data
00000114	00004000	Size of Uninitialized Data
00000118	00006010	Address of Entry Point

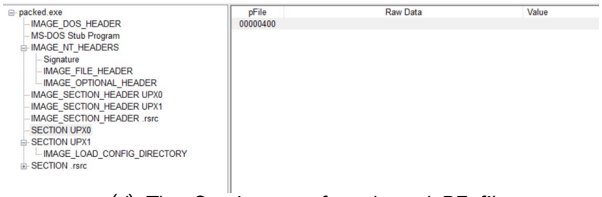
(a) The EP of two PE files

pFile	Data	Description	Value
00001200	00002A7E	Hint/Name RVA	0000 UnhandledExceptionFilter
00001204	00002AFC	Hint/Name RVA	0000 TerminateProcess
00001208	00002AE8	Hint/Name RVA	0000 GetCurrentProcess
0000120C	00002AD4	Hint/Name RVA	0000 GetModuleHandleW
00001210	00002AB8	Hint/Name RVA	0000 IsProcessorFeaturePresent
00001214	00002A9A	Hint/Name RVA	0000 SetUnhandledExceptionFilter
00001218	000029F4	Hint/Name RVA	0000 QueryPerformanceCounter
0000121C	00002A6A	Hint/Name RVA	0000 IsDebuggerPresent
00001220	00002A54	Hint/Name RVA	0000 InitializeSListHead
00001224	00002A3A	Hint/Name RVA	0000 GetSystemTimeAsFileTime
00001228	00002A24	Hint/Name RVA	0000 GetCurrentThreadId
0000122C	00002A0E	Hint/Name RVA	0000 GetCurrentProcessId
00001230	00000000	End of Imports	KERNEL32.DLL
00001234	00002700	Hint/Name RVA	0000 MessageBoxA
00001238	00000000	End of Imports	USER32.dll
0000123C	0000271A	Hint/Name RVA	0000 __current_exception
00001240	00002758	Hint/Name RVA	0000 _except_handler4_common
00001244	00002730	Hint/Name RVA	0000 __current_exception_context
00001248	0000274E	Hint/Name RVA	0000 memset
0000124C	00000000	End of Imports	VCRUNTIME140.dll
00001250	000028CE	Hint/Name RVA	0000 _set_new_mode
00001254	00000000	End of Imports	api-ms-win-crt-heap-l1-1-0.dll
00001258	000028B8	Hint/Name RVA	0000 _configthreadlocale
0000125C	00000000	End of Imports	api-ms-win-crt-locale-l1-1-0.dll
00001260	000027B0	Hint/Name RVA	0000 __setusermatherr
00001264	00000000	End of Imports	api-ms-win-crt-math-l1-1-0.dll
00001268	000028EE	Hint/Name RVA	0000 _initialize_onexit_table
0000126C	0000290A	Hint/Name RVA	0000 _register_onexit_function
00001270	0000288A	Hint/Name RVA	0000 _register_thread_local_exe_atexit_callback
00001274	00002934	Hint/Name RVA	0000 _controlfp_s
00001278	00002944	Hint/Name RVA	0000 terminate
0000127C	00002844	Hint/Name RVA	0000 _exit
00001280	00002800	Hint/Name RVA	0000 _get_initial_narrow_environment
00001284	0000283C	Hint/Name RVA	0000 exit
00001288	00002880	Hint/Name RVA	0000 _c_exit
0000128C	00002876	Hint/Name RVA	0000 _cexit
00001290	000027DE	Hint/Name RVA	0000 _initialize_narrow_environment
00001294	000027C4	Hint/Name RVA	0000 _configure_narrow_argv
00001298	0000285A	Hint/Name RVA	0000 p argc

(b) The IAT of not package PE file

pFile	Data	Description	Value
00001A90	000073A6	Hint/Name RVA	0000 _set_new_mode
00001A94	00000000	End of Imports	api-ms-win-crt-heap-l1-1-0.dll
00001A98	000073B6	Hint/Name RVA	0000 _configthreadlocale
00001A9C	00000000	End of Imports	api-ms-win-crt-locale-l1-1-0.dll
00001AA0	000073CC	Hint/Name RVA	0000 __setusermatherr
00001AA4	00000000	End of Imports	api-ms-win-crt-math-l1-1-0.dll
00001AA8	000073DE	Hint/Name RVA	0000 exit
00001AAC	00000000	End of Imports	api-ms-win-crt-runtime-l1-1-0.dll
00001AB0	000073E4	Hint/Name RVA	0000 _set_fmode
00001AB4	00000000	End of Imports	api-ms-win-crt-stdio-l1-1-0.dll
00001AB8	0000740E	Hint/Name RVA	0000 LoadLibraryA
00001ABC	000073F0	Hint/Name RVA	0000 ExitProcess
00001AC0	000073FE	Hint/Name RVA	0000 GetProcAddress
00001AC4	0000741C	Hint/Name RVA	0000 VirtualProtect
00001AC8	00000000	End of Imports	KERNEL32.DLL
00001ACC	0000742C	Hint/Name RVA	0000 MessageBoxA
00001AD0	00000000	End of Imports	USER32.dll
00001AD4	0000743A	Hint/Name RVA	0000 memset
00001AD8	00000000	End of Imports	VCRUNTIME140.dll

(c) The IAT of packaged PE file



(d) The Section.txt of packaged PE file

Fig. 3. Comparison between original and packed program of PE file structure.

그림 3. 원시 프로그램과 패킹 된 프로그램의 PE 파일구조 비교

3.2 동적 분석을 위한 메모리 분석

본 절에서는 패킹 된 프로그램의 특징을 이용하여 언패킹 이후의 원시 프로그램 정보를 메모리에서 추출하고, 해당 정보를 바탕으로 GCG를 사용해서 그래프를 생

성하는 방법을 제안한다. 본 논문에서는 UPX 패커를 사용하여 구현 결과를 설명하였으나, 패커의 종류에 관계없이 패킹 된 프로그램은 실행직후 언패킹 과정을 수행하고 언패킹 된 데이터를 메모리에 저장한다. 그러므로 언패킹 된 데이터가 저장된 섹션의 시작점과 크기를 분석한다면, GCG를 이용하여 정확한 그래프를 표현할 수 있다.

본 논문에서는 메모리에서 정보를 추출하기 위한 응용프로그램으로 ProcDump를 사용하였으며 [14], 추출된 메모리 정보를 분석하기 위한 응용프로그램으로 Windbg를 사용하였다[15]. 또한, ×32dbg를 사용하여 OEP를 찾았다.

3.2.1 OEP 탐색

패킹 된 프로그램은 실행된 후 최초로 자신을 언패킹하는 작업을 수행한다. 그러므로 패킹 된 프로그램에서 EP는 메모리에 로딩 된 코드에서 언패킹 작업을 수행하는 시작점을 의미한다. GCG는 실행코드를 코드 분해(Dissembly) 한 후, EP를 기준으로 메모리 주소에 따라 순차적으로 분석을 진행하기 때문에 패킹 된 프로그램이 언패킹 후 메모리에 로딩 되어 있는 원시 프로그램의 EP, 즉 OEP를 찾아야 한다. OEP 정보를 패킹 된 코드에서 찾는 방법은 패커마다 서로 다를 수 있다. 본 논문에서는 표 1에서와 같이 악성코드 개발에 많이 사용되는 패커의 OEP 획득 방법을 분석하였다. 일반적으로 언패킹 작업 후, 원시코드를 메모리의 특정 영역에 저장하고, 이 후 해당 영역의 실행 코드로 JMP 하여 원시 코드를 실행하게 된다. 이러한 방법은 UPX, MPRESS, RLPack에서 공통적으로 분석되었다. 즉, 이러한 패커의 특징은 언패킹 작업의 시작 코드 부분에서 PUSHAD(EP) 명령이 수행되고, 언패킹 작업 종료 코드 부분에서 POPAD 명령이 수행된다. 이와 같은 특징을 활용하여 언패킹 코드 블록을 추측할 수 있다. 또한, UPX처럼 언패킹 된 코드가 별도의 영역에 저장되어 있다면, 언패킹 작업이 완료된 후, 반복문을 통해 코드 저장 영역(예를 들어, UPX의 경우 UPX0 영역)에 언패킹 된 코드들을 적재한다. 그리고 JMP 명령어를 이용하여 OEP로 분기 한다. 그러므로 JMP 명령에서 사용되는 주소 값이 OEP 값으로 추측된다.

GCG는 프로그램의 실행 코드를 코드 분해하여 EP를 기준으로 메모리 주소에 따라 순차적으로 분석을 진행한다. 패킹 된 프로그램에서 원시 프로그램의 OEP를 발견하였으나 언패킹 하는 실행 코드를 코드 분해하여 OEP

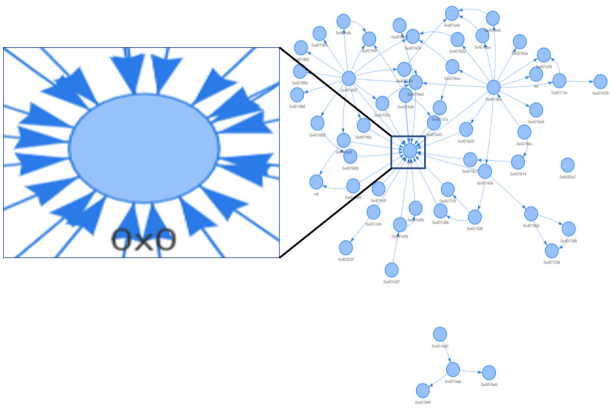


Fig. 4. Result of GCG without IAT.
그림 4. IAT없는 GCG의 그래프 표현 결과

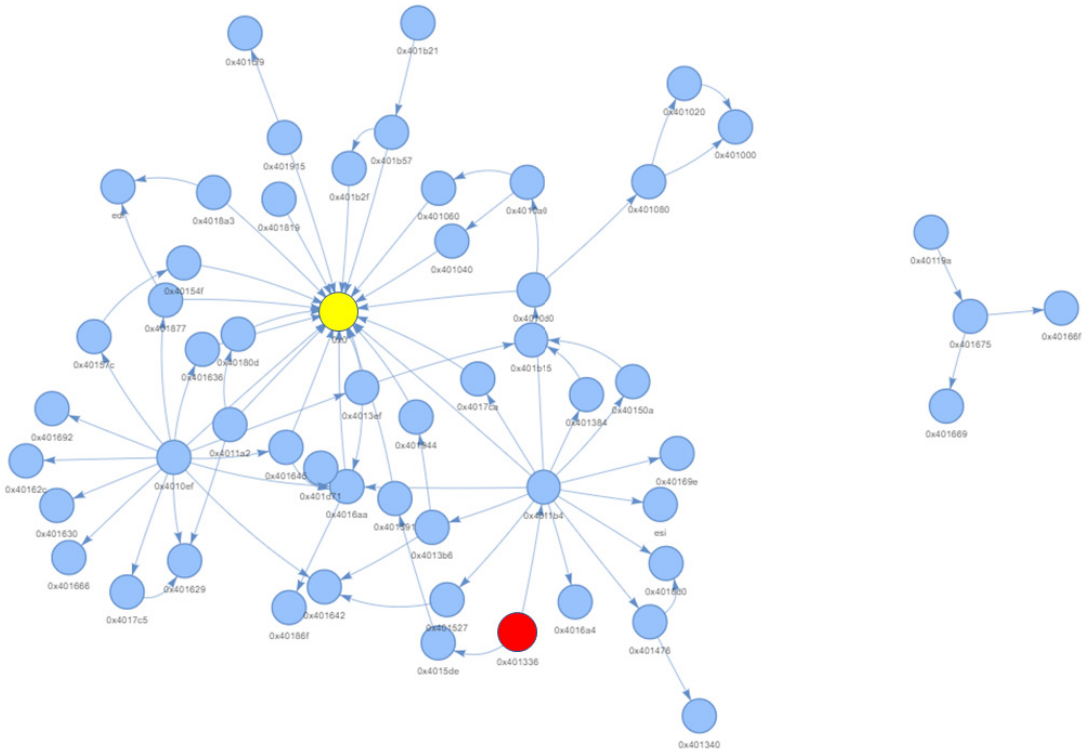
를 지정해주면 오류가 난다. 이는 언패킹 작업이 종료된 이후에 OEP가 등장하기 때문에 언패킹 하는 실행 코드 영역의 범위에는 OEP가 존재하지 않기 때문이다. 이외에도 본 논문에서 앞서 설명한 것처럼 GCG를 통해 정확한 그래프를 표현하기 위해서는 언패킹이 종료된 이후의 실행 코드가 요구되므로 해당 영역의 실행 코드를 찾아야 한다.

3.2.2 원시 실행 코드 획득

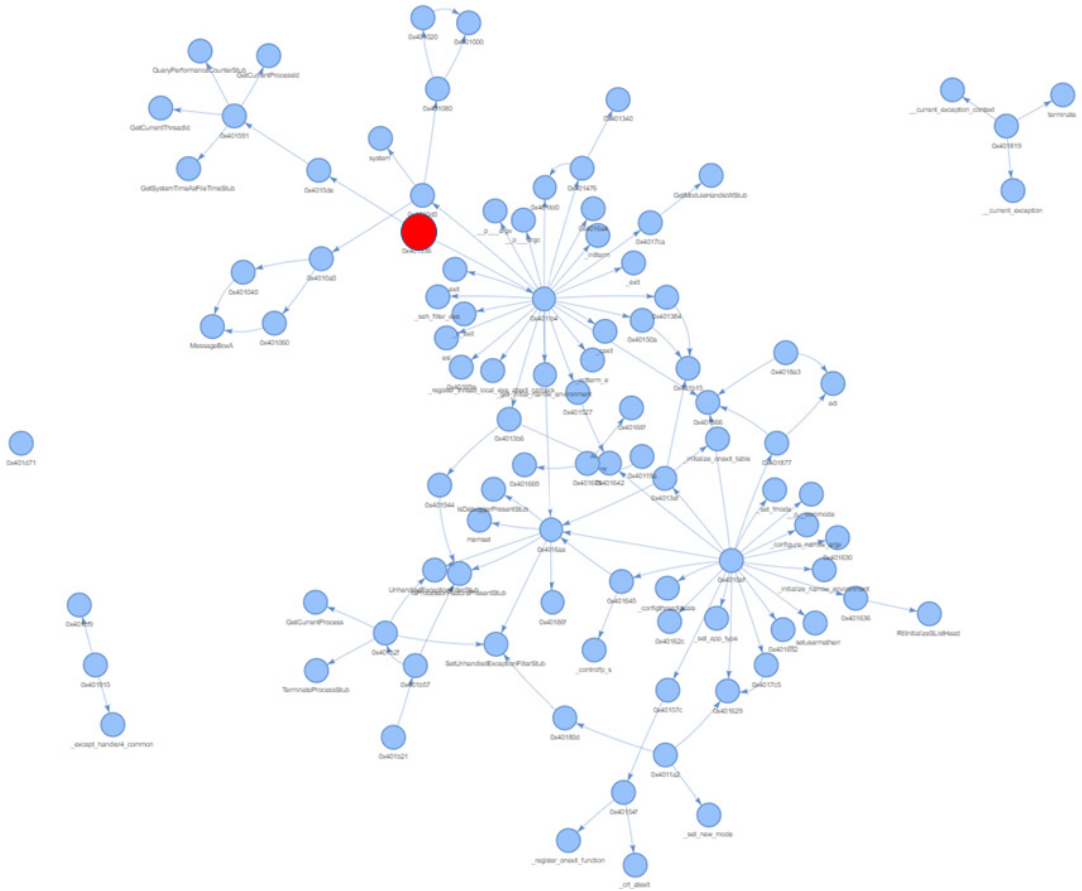
본 논문에서 앞서 설명한 것처럼 패커는 언패킹이 종료된 이후의 실행 코드를 원시코드 저장 영역(예를 들어, UPX 패커는 UPX0 영역)에 적재한다. 그러므로 추출한 메모리 정보를 분석하면 원시 실행 코드를 확보할 수 있다. UPX0 영역 정보는 패킹 된 PE 파일의 Section Header를 통해서도 확보 할 수 있으며, PE 파일의 ImageBase 값과 연동하여 메모리에 로딩 된 UPX0 영역을 찾을 수 있다. 이렇게 확보한 원시 실행 코드에 대하여 디어셈블을 수행한다.

3.2.3 원시 코드의 IAT 구성

획득한 OEP와 원시 실행 코드를 기반으로 GCG를 사용하여 함수 호출 그래프를 생성할 수 있다. 그러나 그림 4에서 많은 수의 함수블록들이 하나의 함수블록을 호출하고 있음을 볼 수 있다. 이는 실제로 해당 함수블록을 호출하는 것이 아니라 호출되는 함수블록의 주소 값을 찾지 못하는 경우, GCG가 임의로 호출되는 함수블록의 주소 값에 0을 저장하도록 오류 처리 한 가상 노드이다. 이와 같은 오류가 다수 발생하는 이유는 정상적인 IAT를 확보하지 못했기 때문이다. 이 문제를 해결하기 위하여 본 절에서는 UPX와 같은 절차를 수행한다.



(a) A GCG call graph of a packaged PE file



(b) A iGCG call graph of a packaged PE file

Fig. 5. The comparison of call graphs of a packaged PE file.
그림 5. 패킹된 PE 파일의 함수 호출 그래프 비교

(1) UPX 패커는 언패킹 된 실행 코드를 UPX0 영역에 저장하기 때문에 언패킹 된 IAT 또한 UPX0 영역에 저장된다. 그러나 언패킹 된 IAT의 시작주소와 크기를 찾기 어렵기 때문에 UPX0 영역에서 IAT를 추출하는 것은 많은 시간이 요구된다. 본 논문에서는 이러한 비효율성을 개선하기 위하여 코드 분해된 실행코드를 분석하면서 피연산자의 주소 값을 UPX0 영역에서 명시하는 방법을 제안한다. 이를 위하여 코드 분해된 실행코드를 분석할 때, 피연산자의 주소를 IAT 리스트가 아닌 UPX0 영역에서 찾도록 수정한다.

(2) 원시 코드의 IAT가 반영된 그래프를 생성하더라도 앞서 IAT 구성 절차가 메모리 주소 값을 기반으로 수행되었기 때문에 동적 연결 라이브러리를 통해 사용되는 함수의 경우, 그래프에는 함수 이름이 아닌 주소 값이 명시된다. 본 논문에서는 이와 같은 문제를 해결하기 위해 추출한 메모리에서 프로그램이 호출하는 동적 연결 라이브러리들과 해당 라이브러리에 속해있는 함수의 주소 값을 추출한 후, 코드 분해된 실행코드를 분석 시 피연산자의 주소 값과 추출한 주소 값을 비교하여 함수 이름으로 지정해준다.

Ⅳ. Improved GCG의 분석 및 결과

그림 5는 패킹된 PE 파일을 GCG와 iGCG를 이용하여 분석한 후 함수 호출 그래프를 생성한 결과이다. 그림에서 붉은 색 노드가 프로그램 EP를 나타낸다. (a)는 패킹된 PE 파일을 GCG로 분석하여 그래프를 생성한 결과이다. 노란색 노드는 앞서 설명한 것처럼 실제 함수가 아니라 호출되는 함수를 메모리 위에서 찾지 못하는 경우를 의미하는 가상 노드이다. (b)는 패킹된 PE 파일을 iGCG로 분석한 것으로, (a)에서와 같은 가상 노드가 존재하지 않는다.

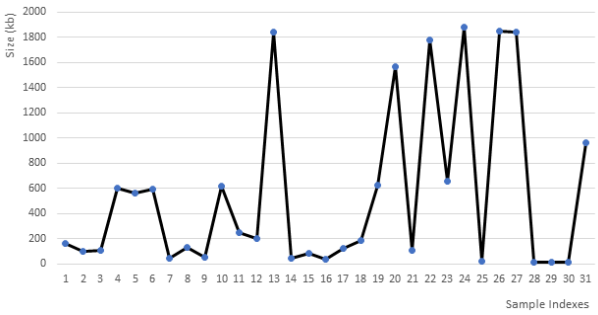
표 1은 MalwareBazaar 사이트에서 실제 배포된 악성코드를 분석한 결과이다[16]. 분석을 위해 다운받은 악성코드는 윈도우즈 운영 시스템에서 작동하는 랜섬웨어로 제한하였다. 그 결과, 305개의 샘플을 다운받을 수 있었다. 그 중 C#으로 제작된 DeepSea Obfuscator를 제외한 패커는 iGCG를 적용할 수 있었다. 즉, 패킹 된 악성코드라도 정상적으로 함수 블록을 구성하고 그래프로 표현할 수 있었다.

그림 6은 305개의 랜섬웨어 중 UPX로 패킹 된 31개의 프로그램에 대한 실행 분석 결과이다. 그림 6-(a)는 31개의 프로그램 각각의 크기를 보여준다. 그림 6-(b)는 31개의 프로그램에 대해 iGCG를 적용하기 위해 데이터를 추출하는 시간에 대한 분석을 보여준다. TMD(Time of Memory Dump)는 실행 코드를 실제 실행하여 언패킹 된 원시 프로그램이 메모리에 언패킹 된 후, 원시 코드를 메모리를 추출하기 위한 시간을 보여준다. 그림에서 보여주듯이 메모리에서 원시 코드를 추출하는 TMD는 실행 프로그램의 크기와 상관없이 모두 유사하다.

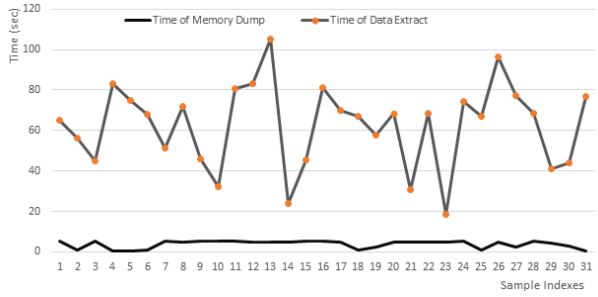
TDE(Time of Data Extract)는 추출한 메모리에서 프로그램의 실행 코드와 IAT를 추출하기 위한 시간을 보여준다. 메모리에서 추출한 실행코드와 IAT를 추출하는 시간은 해당 프로그램에서 사용하는 API의 개수에 따라 다소 차이가 있으나 평균 62초가 소요되었다. 표본 13과 14를 비교 할 경우, 표본 13의 크기는 1,839KB이며 IAT 데이터 추출 시간은 110초 이다. 표본 14의 크기는 47KB이며 데이터 IAT 추출 시간은 29초 이다. 이와 같이 일반적으로 샘플의 크기와 데이터 추출 시간은 비례한다. 그러나 표본 16의 경우, 크기는 37KB인 반면에 IAT 데이터 추출 시간은 86초이다. 이와 같은 경우는, 추출된 IAT 데이터의 개수가 프로그램 크기에 비해 상대적으로 많기 때문에 발생하는 경우이다.

Table 1. Ratio Packaged Ransomware and iGCG Availability.
표 1. 랜섬웨어의 패키징 비율과 iGCG의 적용가능성

Kind of packer	Number of packer	Ratio of packer	GCG	iGCG	iGCG Data	
					OEP	Original Code
UPX	31/305	10.16%	X	O	PUSHAD/ POPAD frame	UPX0
MPRESS	13/305	4.26%	X	O		MPRESS1
RLPack	3/305	0.98%	X	O		PACKED
DeepSea Obfuscator	32/305	10.49%	X	X	-	-
Unknown Packer	11/305	3.60%	X	X	-	-
Not Packed	201/305	71.49%	-	-	-	-



(a) Size of 31 UPX packed programs



(b) Time of Data Extraction on 31 UPX packed programs

Fig. 6. Computation Overheads of iGCG.
그림 6. iGCG 계산량 평가

V. 결론

악성코드를 분석하기 위한 기술로 실행 파일 내의 함수 호출 관계를 시퀀스나 그래프로 표현하는 연구가 제안되어 왔다. GCG도 악성코드를 탐지하기 위해 실행코드에 대한 함수 호출 관계를 그래프로 표현하였다. 이와 같은 함수 호출 관계에 대한 연구들은 PE 파일을 분석하여 획득한 정확한 함수 호출 관련 데이터가 필요하다. 그러나 패킹 된 PE 파일을 정적으로 분석하여 함수 호출 관계를 시퀀스나 그래프로 표현하는 경우, 원시 프로그램의 함수 호출 관련 그래프를 생성하지 못한다. 그러므로 악성코드 탐지 연구를 위해 GCG 그래프를 이용하려면 패킹 된 PE 파일의 함수 호출 관계 그래프 생성을 위한 방안이 필요하다.

본 논문에서는 패킹 된 PE 파일을 동적으로 분석하여 원시 프로그램의 함수 호출 데이터를 추출하고, 이를 기반으로 함수 호출 그래프를 생성하는 iGCG를 제안하였다. 이를 위하여 패킹 된 실행파일을 가상환경에서 실행하여 언패킹 한 후, 메모리에 로딩 된 원시 코드를 직접 획득한다. 이렇게 획득한 코드에서 원시 실행코드와 IAT를 추출하여 GCG 입력에 필요한 데이터로 가공하면 패킹 된 실행 파일이라도 원시 코드의 정확한 그래프를 표현할 수 있다.

본 논문에서 앞서 언급한 바와 같이, GCG는 PE 파일의 실행코드가 그대로 메모리에 로딩 되는 경우만 분석이 가능하다. 그러므로 C#과 같이 중간처리기를 이용하여 실행코드를 메모리에 로드하는 실행코드에 대해서는 그래프 표현이 불가능하다. 이러한 한계는 iGCG에도 여전히 존재한다. 악성코드 탐지 연구를 위한 데이터의 도메인을 넓히기 위해서는 향후 다양한 환경에서 개발된 악성코드를 그래프로 표현하기 위한 연구가 필요하다.

본 연구 결과는 향후 패킹 된 실행 파일로부터 생성된 함수 호출 그래프와 악성코드의 함수 호출 그래프의 유사도를 측정하여 악성코드를 탐지하기 위한 연구에 적용할 수 있을 것으로 판단된다.

References

[1] P. Bajpai and R. Enbody, “Preparing Smart Cities for Ransomware Attacks,” *2020 3rd International Conference on Data Intelligence and Security (ICDIS)*, pp.127-133, 2020.
DOI: 10.1109/ICDIS50059.2020.00023

[2] O. A. Aslan and R. Samet, “A Comprehensive Review on Malware Detection Approaches,” *IEEE Access*, vol.8, pp.6249-6271, 2020.
DOI: 10.1109/ACCESS.2019.2963724

[3] S. R. Davies, R. Macfarlane and W. J. Buchanan, “Review of Current Ransomware Detection Techniques,” *2021 International Conference on Engineering and Emerging Technologies (ICEET)*, pp.1-6, 2021.
DOI: 10.1109/ICEET53442.2021.9659643

[4] H. K. Lee, J. H. Seong, Y. C. Kim, J. B. Kim, and G.-Y. Gim, “The Automation Model of Ransomware Analysis and Detection Pattern,” *Journal of the Korea Institute of Information and Communication Engineering*, vol.21, no.8, pp.1581-1588, 2017. DOI: 10.6109/jkiice.2017.21.8.1581

[5] M. Almousa, S. Basavaraju and M. Anwar, “API-Based Ransomware Detection Using Machine Learning-Based Threat Detection Models,” *2021 18th International Conference on Privacy, Security and Trust (PST)*, pp.1-7, 2021.
DOI: 10.1109/PST52912.2021.9647816

[6] B. Wang, H. Liu, X. Han and D. Xuan, “RanPAS:

A Behavior-based System for Ransomware Detection," *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*, pp.309-314, 2021. DOI: 10.1109/DSC53577.2021.00049
[7] U. Urooj, B. A. S. Al-rimy, A. Zainal, F. A. Ghaleb, and M. A. Rassam, "Ransomware Detection Using the Dynamic Analysis and Machine Learning: A Survey and Research Directions," *Applied Sciences*, vol.12, no.1, 2021.
DOI: 10.3390/app12010172
[8] S. H. Lee and J. S. Hwang, "A study on variable selection and classification in dynamic analysis data for ransomware detection," *The Korean Journal of Applied Statistics*, Vol.31, No4, pp.497-505, 2018.
[9] H. S. Kang, S. R. Kim, "Offline Based Ransomware Detection and Analysis Method using Dynamic API Calls Flow Graph," *Journal of Digital Contents Society,* vol.19, no.2, pp.363-370, 2018.
DOI: 10.9728/dcs.2018.19.2.363
[10] D. H. Choi, (2021) "Graph Database Design and Implementation for Ransomware Detection," *Journal of Convergence for Information Technology*, Vol.11, no.6, pp.22-32, 2021.
DOI: 10.22156/CS4SMB.2021.11.06.024
[11] J. H. Kwon, J. H. Lee, H. C. Jeong, and H. J. Lee, "Metamorphic Malware Detection using Subgraph Matching," *Journal of the Korea Institute of Information Security & Cryptology*, vol.21, no.2, pp.37-47, 2011.
DOI: 10.1109/ICCKE.2015.7365862
[12] D. Y. Kim, "Generating Call Graph for PE file," *Journal of IKEEE*, vol.25, no.3, pp.451-461, 2021. DOI: 10.7471/ikeee.2021.25.3.451
[13] M. Manna, A. Case, A. Gombe, G. Richard, "Memory analysis of .NET and .Net Core applications," *Forensic Science International: Digital Investigation*, vol.42, 2022.
DOI: 10.1016/j.fsidi.2022.301404
[14] "ProcDump v.11.0,"https://learn.microsoft.com/ko-kr/sysinternals/downloads/procdump
[15] "Windows 디버깅 도구(WinDbg),"https://learn.microsoft.com/ko-kr/windows-hardware/drivers/debugger/
[16] "MalwareBazaar Database," https://bazaar.abuse.ch/browse

BIOGRAPHY

Se-Beom Cheon (Member)

2016~ : Student, Dept. of Information Security, Suwon Univ.

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.