JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 25 No. 11, pp. 131-138, November 2020
https://doi.org/10.9708/jksci.2020.25.11.131

Automatic malware variant generation framework using Disassembly and Code Modification

Jong-Lark Lee*, Il-Yong Won**

*Professor, Faculty of Cyber Security, Yeungnam University College, Daegu, Korea
**Professor, Dept. of Cyber Hacknig Security, Seoul Hoseo Technical College, Seoul, Korea

[Abstract]

Malware is generally recognized as a computer program that penetrates another computer system and causes malicious behavior intended by the developer. In cyberspace, it is also used as a cyber weapon to attack adversary. The most important factor that a malware must have as a cyber weapon is that it must achieve its intended purpose before being detected by the other's detection system. It requires a lot of time and expertise to create a single malware to avoid the other's detection system.

We propose the framework that automatically generates variant malware when a binary code type malware is input using the DCM technique. In this framework, the sample malware was automatically converted into variant malware, and it was confirmed that this variant malware was not detected in the signature-based malware detection system.

▸ **Key words**: Malware, Variant, Auto generation, DCM, Cyber weapon

[요 약]

멀웨어는 일반적으로 다른 사용자의 컴퓨터시스템에 침입하여 개발자가 의도하는 악의적인 행위를 일으키는 컴퓨터프로그램으로 인식되지만 사이버 공간에서는 적대국을 공격하기 위한 사이버 무기로써 사용되기도 한다. 사이버 무기로서 멀웨어가 갖춰야 할 가장 중요한 요소는 상대방의 탐지시스템에 의해 탐지되기 이전에 의도한 목적을 달성하여야 한다는 것인데, 하나의 멀웨어를 상대방의 탐지 시스템을 피하도록 제작하는 데에는 많은 시간과 전문성이 요구된다.

우리는 DCM 기법을 사용하여, 바이너리코드 형태의 멀웨어를 입력하면 변종 멀웨어를 자동으로 생성해 주는 프레임워크를 제안한다. 이 프레임워크 안에서 샘플 멀웨어가 자동으로 변종 멀웨어로 변환되도록 구현하였고, 시그니쳐 기반의 멀웨어 탐지시스템에서는 이 변종 멀웨어가 탐지되지 않는 것을 확인하였다.

▸ **주제어**: 멀웨어, 변종, 자동생성, DCM, 사이버무기

• First Author: Jong-Lark Lee, Corresponding Author: Il-Yong Won
*Jong-Lark Lee (jllee@ync.ac.kr), Faculty of Cyber Security, YeongNam University College
**Il-Yong Won (clccclcc@shoseo.ac.kr), Dept. Cyber Hacking Security, Seoul Hoseo Technical College
• Received: 2020. 10. 06, Revised: 2020. 10. 19, Accepted: 2020. 10. 19.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

멀웨어(Malware)란 일반적으로 사용자의 컴퓨터 시스템이나 모바일 디바이스 등에 침투하여 개발자가 의도하는 악의적인 행위를 수행하는 소프트웨어를 말한다. 반면 제5전장이라고 일컬어지는 사이버 공간에서는 적대국의 네트워크나 컴퓨터 시스템에 침투하여 시스템을 교란시키거나 주요 데이터를 수집, 파괴, 변조하기 위한 목적으로 사용되기도 한다. 이 경우에는 일반적으로 멀웨어라는 용어보다는 사이버 무기라는 용어로 사용된다.

사이버 무기 운용자의 입장에서는 자신이 확보한 소프트웨어가 상대방의 탐지 시스템에 탐지되기 이전에 컴퓨터 시스템에 침투하여 의도한 목적을 달성하도록 노력할 것이며, 반면 탐지 시스템 개발자는 공격자가 보낸 멀웨어를 빠르게 탐지하여 제거하고자 할 것이다.

멀웨어 변종을 자동으로 생성하는 문제에 대한 연구는 두 가지 관점에서 그 의의를 찾을 수 있는데, 첫째는 사이버 무기 운용자의 관점이다. 사이버 전장에서는 상대방이 사용하는 탐지 시스템에 탐지되지 않는 사이버 무기를 빠르게 만들어 내어 확보하는 것이 관건인데 이를 위해서는 많은 시간과 높은 전문성이 요구된다. 따라서 하나의 무기를 확보했을 때 탐지되지 않으면서도 원하는 목적을 수행할 수 있는 공격용 무기 수백 만개를 대량으로 생성할 수 있다면 공격자의 입장에서는 매우 유용하다고 할 수 있다.

둘째는 탐지 시스템 개발자의 입장이다. 탐지 시스템 개발자는 하나의 멀웨어가 발견되면 이를 분석하여 효율적으로 탐지하기 위한 시그니처를 추가한다. 또한 이 멀웨어의 변종이 발견되면 자신의 탐지 시스템이 이에 대응할 수 있는지를 테스트한 후 불완전하다고 판단되는 경우 이를 완벽하게 대응할 수 있도록 시그니처를 추가한다. 하나의 멀웨어에 대하여 매우 다양한 변종이 발생된다는 관점에서 볼 때 이는 매우 소모적인 일이며, 예방적 차원에서 볼 때 늦은 대처일 수밖에 없다. 만약 한 개의 멀웨어가 발견되었을 때, 이로 인해 발생할 수 있는 다양한 변종 생성을 미리 고려하여 시그니처를 개발할 수 있다면 보다 빠르게 대응할 수 있을 것이다.

국내의 멀웨어 변종 생성에 관한 연구는 주로 이를 어떻게 효율적으로 탐지할 것인지에 대한 문제에 국한되어 있다. [1]은 변종 멀웨어들이 공통으로 사용하는 코드의 재사용 여부를 통해 이를 탐지하는 분석기법을 연구하였고, [2]는 자동화된 도구에 의해 생성된 변종 멀웨어의 7가지 공통 속성을 선정한 후 이에 매칭되는지의 여부를 가지고 변종 멀웨어를 탐지하는 방법에 대해 연구하였다. 그 외에도 다양한 변종 멀웨어 탐지 방법이 연구되고 있으나 탐지 시스템에 의해 탐지되지 않으면서도 자동으로 변종을 생성하는 사이버 무기의 관점에서의 연구는 없다.

각종 공격용 도구들을 모아놓은 Kali Linux 안에는 정상 프로그램 내에 악성코드를 삽입하여 정상 프로그램 실행 시 악성코드가 함께 실행되도록 하는 멀웨어 자동생성 도구가 포함되어 있으나 이는 변종 멀웨어 생성을 위한 도구와는 차이가 있다.

국외에서는 비교적 활발한 연구가 이루어지고 있는데 주로 진화 알고리즘(Evolution Algorithm) 등을 사용하여 기존의 소스 코드를 다른 소스 코드로 자동으로 변화시키거나 상이한 두 멀웨어의 소스 코드를 추출하여 선택, 교차, 변이 연산을 사용하여 새로운 변종 멀웨어를 생성하는 연구가 진행되고 있다[3],[4].

멀웨어의 변종 개발은 실제 소스 코드를 가지고 있는 경우와 바이너리 코드의 실행파일만을 가지고 있는 경우로 구분할 수 있다.

소스 코드를 가지고 있는 경우라면 소스 코드 레벨에서 해당 소스 코드를 어떻게 하면 자동으로 변화시켜 탐지되지 않는 변종을 개발할 수 있을 것인가의 문제가 된다. [3],[4]가 이러한 경우에 해당한다. 하지만 그렇지 않은 경우, 즉 코드의 수정이 불가능한 바이너리 코드만 가지고 있다면, 이를 다시 사람이 식별 가능한 어셈블리어 코드의 형태로 변환시킨 후 어셈블리어 코드 레벨에서 코드를 어떻게 자동으로 수정할 것인가의 문제가 된다. 뿐만 아니라 수정된 어셈블리어 코드를 다시 바이너리 코드로 변환시키는 문제를 해결해야 한다.

본 연구에서는 멀웨어의 소스 코드를 가지고 있지 않은 경우에 멀웨어의 변종을 자동으로 생성하는 문제를 다룬다. 즉, 바이너리 코드 형태의 원본 멀웨어를 입력하면 변종 멀웨어가 자동으로 생성되기 위한 시스템의 프레임워크로서 DCM(Disassembly and Code modification) 방법을 제안한다. 그리고 이 프레임워크를 구현한 후 실험용 샘플을 적용하여 해당 프레임워크가 동작 가능함을 보여줄 것이다.

본 논문의 구성은 다음과 같다. 2장에서는 본 연구와 관련된 연구를 설명한다. 3장에서는 멀웨어를 자동으로 생성하기 위해 제안하는 DCM 프레임워크의 전체적인 구조와 DCM 프레임워크의 세부 구현과정 그리고 샘플 파일을 이용하여 적용한 실험 결과를 설명한다. 그리고 4장에서는 연구 내용의 요약과 함께 향후 연구 방향에 대해 설명하는 결론으로 끝을 맺는다.

II. Related Works

1. Malware Detection and Obfuscation

일반적으로 멀웨어 탐지를 위한 분석 방법은 정적 분석과 동적 분석으로 분류된다. 정적 분석은 멀웨어를 실행시키지 않고 분석하는 방법으로써 바이너리 파일을 디스어셈블하여 어셈블리어 코드로 변환한 후 프로그램의 구조나 동작 원리 등을 파악하여 해당 멀웨어의 특징을 시그니처로 추출하게 된다. 실행파일만을 가지고 프로그램을 역분석하는 기술을 리버스엔지니어링이라고 부르며, Ollydbg, IDA pro 등의 툴을 사용한다. 이에 반해 동적 분석은 특정한 실행 환경을 만들어 멀웨어를 실행시킨 후 멀웨어가 수행하는 행위를 분석하는 방법으로써 Function Call 모니터링, Function Parameter 분석, 프로그램이 처리하는 데이터 추적, 프로그램 실행시에 부가적으로 자동실행되는 프로그램 설치 여부 등을 분석한다[5]. 대표적인 프로그램으로는 Process Explorer, vmmap, autoruns 등의 프로그램이 있다. 최근에는 멀웨어 탐지 오류를 줄이면서도 빠른 탐지를 위해 Machine -Learning을 활용한 탐지 방법에 관한 연구도 활발히 이루어지고 있다[6].

멀웨어의 난독화(obfuscation)란 멀웨어 분석가가 리버싱을 통해 이를 분석하는 것을 어렵게 만들기 위한 과정을 의미한다. 원본 멀웨어와 동일한 기능을 수행하지만 코드의 형태는 새로운 버전으로 변환시키는 기법들이 사용되는데, 다형성 기법과 패킹 기법이 그 예이다. 다형성 기법이란 코드의 기능은 동일하게 유지되지만 코드의 내용은 변경됨으로써 시그니처 기반의 백신에서는 탐지되지 못하도록 생성된 멀웨어를 의미한다. 주로 악성 페이로드 부분의 암호화, 코드의 이동, 주요 변수명의 변경, 유사 코드로 대체, 레지스터 재할당, 공백삭제, 코드축소 등을 통해 수행된다[7].

패킹이란 압축이나 암호화를 사용하여 실제 멀웨어의 악의적인 코드를 분석하기 어렵게 만드는 기법을 말하며, 이를 위한 도구를 패커라 부른다. 원래는 저작권 보호나 민감정보의 불법적인 복사를 방지하기 위한 목적으로 사용되었으나 멀웨어 공격자가 자신의 코드를 숨기거나 탐지를 지연시키기 위한 목적으로 사용된다. 패킹의 과정은 다음과 같이 진행된다.

(1) 원본 코드가 패커에 업로드되면 압축이나 암호화를 위한 프로세스가 수행된다.

(2) 원본 PE(Portable Executable) Header와 코드는 압축되거나 암호화되어 새로운 실행파일의 Packed Section에 저장된다.

(3) 패킹된 파일에는 다음이 추가된다.

① 새로운 PE Header
② Packed Section
③ 언패킹을 위한 Section

(4) 패킹과정에서 OEP(Original entry point)는 Packed Section 내에 존재하게 되므로 읽을 수 없다.

(5) 새로운 PE 파일에서는 새로운 OEP를 할당하여 언패킹을 위한 루틴 영역을 가리키게 하여 언패킹이 먼저 이루어진 후 멀웨어가 실행된다.

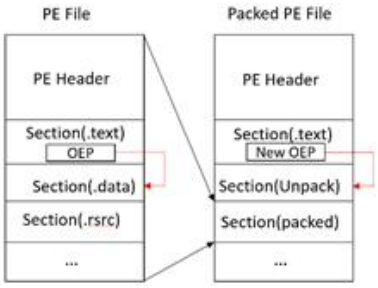

Fig. 1. Packing Process of PE file

2. Disassembly and Assembly

수집된 멀웨어는 대부분 바이너리 코드 형태의 실행파일이므로 해당 프로그램의 소스 코드를 알 수는 없다. 멀웨어 분석가든 사이버 무기 제작자든 간에 바이너리 코드 상태로는 할 수 있는 일이 없으므로 사람이 읽을 수 있는 형태의 코드로 변환해야 할 필요가 있다. 디컴파일러(decompiler)는 사람이 읽을 수 있는 소스 코드 형태로 변환시켜주는 도구를 의미하나 바이너리 코드를 완벽한 소스 파일의 형태로 변환시켜주는 디컴파일러는 아직 존재하지 않는다. 반면 비록 소스 코드에 비해 이해하기 쉽지는 않지만 사람이 읽을 수 있는 어셈블리어 코드 형태로 변환시켜주는 디스어셈블러(disassembler)가 있다. 바이너리 코드를 소스 코드로 변환하는 것은 어렵지만 이를 어셈블리어 코드로 변환하는 일은 그리 어려운 작업이 아니므로 이를 위한 다양한 도구들이 존재한다. 이를 위한 도구로는 NASM(Netwide Assembler), MASM(Microsoft Macro Assembler), TASM(Turbo Assembler), FASM(Flat Assembler) 등이 존재하며 본 연구에서는 x86 프로세서용으로서 빠른 속도, 코드크기 최적화 및 운영체제 호환성의 특징을 갖고 있는 FASM을 사용한다[8].

일반적으로 멀웨어를 분석하는 것만이 목적이라면 바이너리 코드를 어셈블리어 코드로 변환시키는 디스어셈블러만 사용하면 되지만, 어셈블리어 코드를 수정하여 멀웨어의 변종을 만들기 위한 목적이라면 이를 다시 바이너리코드로 변환하는 어셈블 과정도 필요하다.

3. Malware Variant Generation Framework

멀웨어 탐지 방법이 계속 발전하고 있는 것과 동시에 이에 탐지되지 않도록 하기 위한 다양한 멀웨어의 변종 생성 방법들도 지속적으로 발전하고 있다. 이를 위한 일반적인 Framework를 요약하면 Fig.2와 같다.

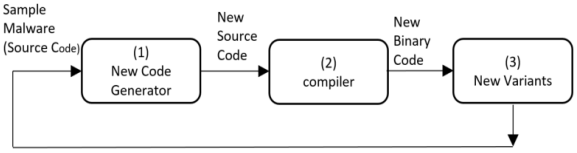

Fig. 2. General Malware Variant Auto Generating process

멀웨어 자동생성을 위한 기존의 연구들은 원본 멀웨어의 소스 코드가 존재하는 경우가 대부분이다. [3][4]는 Fig.2에서 (1)의 과정을 EA(Evolvable Algorithm)알고리즘을 이용하여 새로운 코드를 생성하는 프레임워크를 제안하였으며, Bagle이라는 웜과 Timid라는 바이러스의 소스 코드를 활용하여 백신에 탐지되지 않는 새로운 변종을 생성하는 데 성공하였다. 하지만 이는 완전한 멀웨어 소스 코드를 확보한 경우에 해당된다.

[6]은 최근에 유전자알고리즘을 이용한 머신러닝 기법으로 변종 멀웨어를 자동으로 생성하는 Framework인 AMVG를 제안하였다. 샘플 멀웨어가 입력되면 해당 소스 코드를 parser와 Modifier모듈을 이용하여 수정한 후 탐지 여부를 평가하는 Score모듈을 통해 탐지되면 0, 탐지되지 않으면 1로 처리하는 시스템을 제안 후 Python과 C 언어로 만들어진 멀웨어의 소스 파일을 이용하여 구현된 결과를 실험하였다.

III. The Proposed Framework

1. DCM Framework

바이너리 코드 형태의 멀웨어에 대한 변종을 생성하기 위해 우리가 제안하는 DCM Framework는 Fig. 3과 같다.

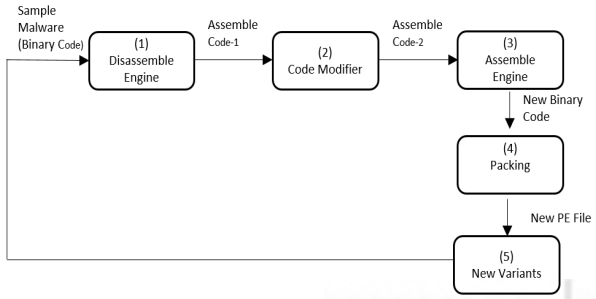

Fig. 3. DCM Framework

(1) **Disassemble Engine** : 바이너리 형식의 멀웨어를 Disassemble Engine을 통해 어셈블리언어 코드 형식으로 변환하는 과정이다. 우리의 목적은 원본과 바이너리 코드는 상이하지만 기능은 동일한 변종을 생성하는 것이므로 바이너리 코드로는 이를 달성하기 어렵다. 따라서 코드의 수정이 가능한 어셈블리어 코드로 변환시킨다. 바이너리 코드 형태의 파일이 어셈블리어 형태로 변환되면 원래의 소스 코드가 가지고 있던 변수의 타입, 데이터의 구조나 크기 등의 정보를 상실한다. 또한 이때 어떤 Disassembler를 사용했는가에 따라 어셈블리어 코드의 형태도 달라진다. 이러한 정보의 상실로 인해 기존의 시그니처 방식의 탐지 도구로는 탐지가 어렵다.

(2) **Code Modifier** : 변환된 어셈블리어의 코드를 수정하는 과정이다. 어셈블리어 코드의 재배치, 다른 코드로의 대체, 특정한 알고리즘을 활용한 코드의 재생성 등을 수행한다. 본 연구에서는 새로운 알고리즘을 적용한 코드의 재수정은 적용하지 않았으나 μGP 를 활용한 어셈블리 코드의 재생성을 적용해 볼 수 있다[9].

(3) **Assemble Engine** : 어셈블리어 코드를 입력받아 새로운 바이너리 코드 형태의 실행파일을 생성하는 과정이다. 이를 위해서는 멀웨어가 동작하기 위해 사용하는 라이브러리 함수의 정보 알고 있어야 한다. 이 정보는 PE 헤더와 IAT(Import Address Table)에 저장되어 있는데 우리는 이를 자동으로 찾아 재배치하였다.

(4) **Packing** : 새롭게 생성된 바이너리 실행파일을 입력받아 압축이나 암호화 과정을 통해 난독화하는 과정이다. 이를 통해 어셈블리어 코드로의 변환을 어렵게 하여 시그니처 기반의 탐지 시스템으로부터의 탐지를 바이패스 시키도록 한다.

(1)에서부터 (4)의 과정이 모두 종료되면 새로운 PE 파일이 생성된다. 위의 과정이 반복되면 탐지 시스템은 이를 더욱 탐지하기 어렵게 된다.

2. Implementation and Result

DCM의 구현 및 실험을 위해 사용한 샘플 멀웨어는 KeyLogger이다. 이것을 사용한 이유는 다른 멀웨어 프로그램에 비해 동작 과정이 단순하며 소스가 공개되어 있어 프로그램의 동작 과정을 쉽게 분석할 수 있기 때문이다. 다만 해당 소스 코드는 프로그램의 동작 과정을 분석하기 위한 용도로만 사용했으며 실제 DCM 과정에서는 바이너리 코드를 사용했다.

2.1 Disassembly and New Assembler Code Generation

바이너리 코드 형태의 멀웨어를 어셈블리어 코드 형태로 변환하기 위해서 고려해야 할 것은 변환된 어셈블리어 코드를 다시 바이너리 코드로 만드는 어셈블러의 문법을 준수하는 것이다. 우리는 공개 어셈블러인 FASM 을 사용하였다.

DCM의 첫 단계인 바이너리 코드를 이용하여 어셈블리어 코드를 생성하기 위해서는 바이너리 파일을 디스어셈블하여 실행 코드(instruction)들과 변수들을 복원해야만 한다. 이때 디스어셈블러들은 각각 고유한 자신만의 문법 형태로 디스어셈블하게 되는데, 우리는 FASM 문법에 가장 유사한 문법 형태를 제공하는 Pydasm 디스어셈블러를 사용하였다. 여기서 주의해야 하는 것은 어떤 디스어셈블러도 완벽한 어셈블리 언어 즉, 100% 원본으로 변환하지 못한다는 것이다.

바이너리 코드를 어셈블리어 코드로 변환하기 위해서는 컴파일 당시 컴파일러에 의해 추가된 부수적인 코드를 피해 실행의 핵심 부분만을 추출해야 한다. 추출해야 하는 내용은 크게 실행 코드 부분, 데이터 부분, 외부 라이브러리 연결부분으로 요약할 수 있다.

실험하는 멀웨어는 윈도우 실행 바이너리(PE) 형태이므로, 이 파일의 PE 헤더에 있는 정보를 이용하여 프로그램이 시작하는 주소를 추출한 후 디스어셈블을 시작한다. Fig. 4는 샘플 멀웨어를 Pydasm을 이용하여 디스어셈블한 코드의 일부이다. 이를 이용하여 실행 파일의 소스 코드가 존재하는 .text 부분의 코드를 복원하였다.

```
address: 00000f80
    push ebp
address: 00000f81
    mov ebp,esp
address: 00000f83
    sub esp,0x48
address: 00000f86
    push ebx
address: 00000f87
    push esi
address: 00000f88
    push edi
address: 00000f89
    push byte 0x0
address: 00000f8b
    push dword 0x80
address: 00000f90
    push byte 0x2
address: 00000f92
    push byte 0x0
address: 00000f94
    push byte 0x0
address: 00000f96
    push dword 0x40000000
address: 00000f9b
    push dword 0x404820
address: 00000fa0
    call [0x407004]
address: 00000fa6
    mov [ebp-0x4],eax
address: 00000fa9
    mov eax,0x1
address: 00000fae
    test eax,eax
address: 00000fb0
    jz 0x1009
address: 00000fb2
    mov dword [ebp-0x8],0x0
address: 00000fb9
    jmp 0xfc4
```

Fig. 4. Disassembly Code of keylogger

데이터 영역 또한 PE 구조를 이용하여 추출할 수 있는데, PE파일을 통해 얻을 수 있는 정보는 데이터 영역의 시작 주소와 크기뿐이며, 구체적인 변수의 이름이나 유형은 코드 부분을 이용하여 유추할 수 밖에 없다. 아래 그림은 추출한 .data 섹션의 일부이다.

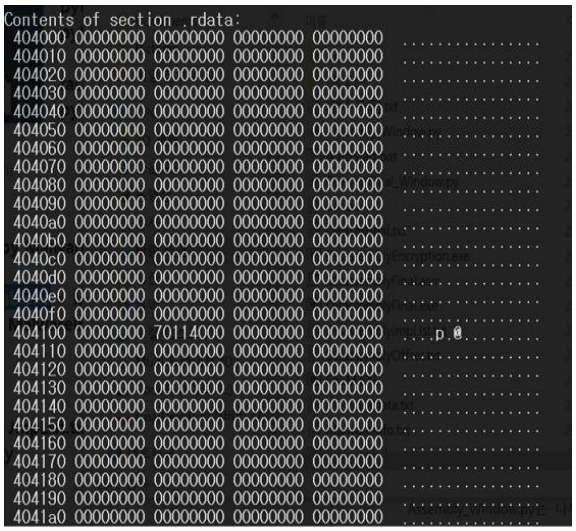

Fig. 5. Disassembly Code of keylogger .data Section

디스어셈블된 .text 부분의 실행 코드에서 변수는 모두 주소로 표현되므로 이 주소와 .data 영역을 매핑하여 새로운 어셈블리 코드에 변수를 정의하는 것이 필요하다. 매핑의 핵심은 .text 코드에서 jmp 명령어를 추출하여 .data 영역의 주소로 분기하는 것을 변수로 복구하는 것이다. Fig. 6은 이 기능을 구현한 결과이다.

```
0x404820 = dat.txt = msg1
0x404828 = %c = msg2 = 0ah
```

Fig. 6. Variable mapping in the assembly code

IAT에는 외부에서 사용하는 각종 라이브러리의 주소가 저장되어 있는데, 운영체제의 종류와 상태에 따라 외부 라이브러리의 주소는 변경된다. 따라서 원본 멀웨어에서 IAT의 주소를 확인한 후, 변환된 어셈블리어 코드에서는 이를 함수 이름으로 변경해 주어야 한다. 그래야만 복구된 소스를 다시 바이너리 코드로 어셈블 할 때 외부 라이브러리를 다시 할당할 수 있기 때문이다. Fig. 7은 이를 위해 시스템에서 사용한 IAT의 주소와 함수의 대응표 중 일부이다.

```
'0x406160 UnhandledExceptionFilter',
'0x406164 GetCurrentProcess',
'0x406168 TerminateProcess',
'0x40616c GetSystemTimeAsFileTime',
'0x406170 GetCurrentProcessId',
'0x406174 GetCurrentThreadId',
'0x406178 GetTickCount',
'0x40617c QueryPerformanceCounter',
'0x406180 DecodePointer',
'0x406184 SetUnhandledExceptionFilter',
'0x406188 HeapSetInformation',
'0x40618c InterlockedCompareExchange',
'0x406190 Sleep',
'0x406194 InterlockedExchange',
'0x406198 EncodePointer',
'0x40619c IsDebuggerPresent',
'0x4061d8 ?terminate@@YAXXZ',
'0x4061dc _controlfp_s',
'0x4061e0 _invoke_watson',
```

Fig. 7. Correspondance of Address and Function

우리는 이를 이용하여 샘플멀웨어에 대한 어셈블리어 코드를 생성하였으며, Fig. 8은 그 일부이다.

```
address: 00000f80
    push ebp
address: 00000f81
    mov ebp,esp
address: 00000f83
    sub esp,0x48
address: 00000f86
    push ebx
address: 00000f87
    push esi
address: 00000f88
    push edi
address: 00000f89
    push byte 0x0
address: 00000f8b
    push dword 0x80
address: 00000f90
    push byte 0x2
address: 00000f92
    push byte 0x0
address: 00000f94
    push byte 0x0
address: 00000f96
    push dword 0x40000000
address: 00000f9b
    push dword 0x404820
address: 00000fa0
    call [0x407004]
address: 00000fa6
    mov [ebp-0x4],eax
address: 00000fa9
    mov eax,0x1
address: 00000fae
    test eax,eax
address: 00000fb0
    jz 0x1009
address: 00000fb2
    mov dword [ebp-0x8],0x0
address: 00000fb9
    jmp 0xfc4
address: 00000ff3
    push byte 0x0
address: 00000ff5
    push byte 0x1
address: 00000ff7
    lea eax,[ebp-0x8]
address: 00000ffa
    push eax
address: 00000ffb
    mov ecx,[ebp-0x4]
address: 00000ffe
    push ecx
address: 00000fff
    call [0x407028]
address: 00001005
    jmp 0xfbb
address: 00001007
    jmp 0xfa9
address: 00001009
    mov eax,[ebp-0x4]
address: 0000100c
    push eax
address: 0000100d
    call [0x407000]
address: 00001013
    xor eax,eax
address: 00001015
    pop edi
address: 00001016
    pop esi
address: 00001017
    pop ebx
address: 00001018
    mov esp,ebp
address: 0000101a
    pop ebp
address: 0000101b
    ret
address:
address: 00000fbb
    mov eax,[ebp-0x8]
address: 00000fbe
    add eax,0x1
address: 00000fc1
    mov [ebp-0x8],eax
address: 00000fc4
    cmp dword [ebp-0x8],0x100
address: 00000fcb
    jnl 0x1007
address: 00000fcd
    mov eax,[ebp-0x8]
address: 00000fd0
    push eax
address: 00000fd1
    call [0x407110]
address: 00000fd7
    movsx ecx,ax
address: 00000fda
    and ecx,0x1
address: 00000fdd
    jz 0x1005
address: 00000fdf
    mov eax,[ebp-0x8]
address: 00000fe2
    push eax
address: 00000fe3
    push dword 0x404828
address: 00000fe8
    call [0x4070d4]
address: 00000fee
    add esp,0x8
address: 00000ff1
    push byte 0x0
address: 00000ff3
    push byte 0x0
address: 00000ff5
    push byte 0x1
address: 00000ff7
    lea eax,[ebp-0x8]
```

Fig. 8. Assembler Code generated by DCM

2.2 Code Modification and New Binary Code Generation

새로 만들어진 멀웨어는 원본 멀웨어와 동일한 기능을 수행하지만, 어셈블리어 코드 레벨에서의 변화로 인해 바이너리가 코드의 형태는 달라진다. 아래 그림은 원본 바이너리와 변경된 바이너리의 비교 그림으로 차이가 존재함을 알 수 있다.

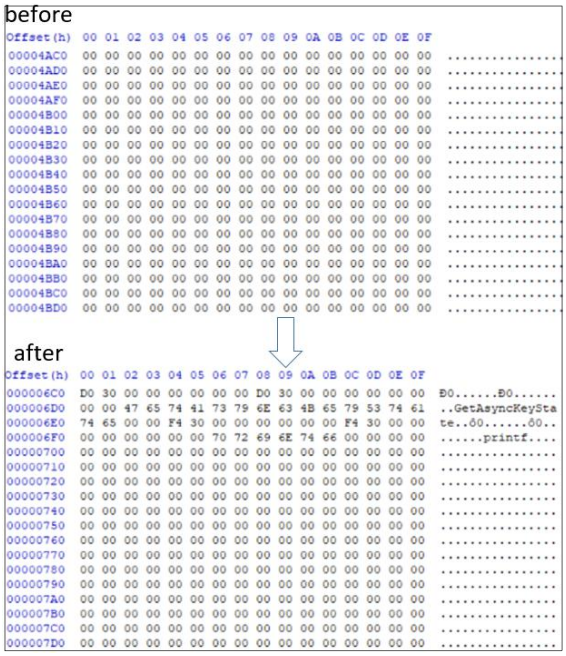

Fig. 9. Comparison of Original binary and Variant

만들어진 바이너리의 실행 코드 주소는 변경되었지만, 로직이 동일하므로 유사 패턴이 발견될 수 있다. 이에 우리는 패턴을 제거하기 위해 새로 만들어진 바이너리 파일에서 패턴이 발견될 수 있는 .text 영역을 VirtualProtect 함수를 이용하여 XOR로 암호화하여 코드를 변경시켰다. 아래 그림은 이렇게 변경된 바이너리 코드의 일부분이다.

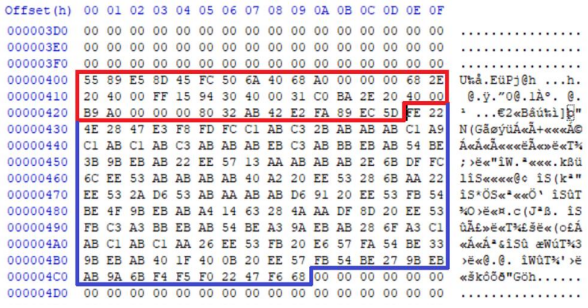

Fig. 10. Binary code after Applying Obfuscation

2.3 Detection Test

우리는 DCM에 의해 생성된 새로운 변종 멀웨어에 대한 탐지 테스트를 두 가지 방법으로 수행하였다. 첫째는 시그니처 기반의 멀웨어 검사 도구를 이용한 방법으로써 VirusTotal(www.virustotal.com)을 이용하였다. 이는 Google에서 제공하는 멀웨어 검사용 사이트로서 업로드된 파일에 대한 멀웨어 여부를 V3, 알약, 카스퍼스키 등 전 세계의 유명 백신 프로그램 70 여종을 이용하여 빠르게 검사해 준다. 전세계에서 개발된 유명 멀웨어 탐지 소프트웨

어 대부분을 이용하여 검사를 수행한다는 장점이 있지만, 행위기반의 검사는 하지 않는 것으로 알려져 있다[10].

둘째는 행위기반의 탐지 도구를 이용한 방법으로써 독일에서 개발된 Hybrid-Analysis(www.hybrid-analysis.com) 사이트를 이용하였다. 이 사이트는 업로드된 파일을 자사에서 개발한 Falcon Sandbox 아래에서 실행시켜 그 행위를 분석한 후 위협점수(Threat Score)를 100점 만점으로 레포팅 해 준다. 이때 이 위협점수는 파일이 실행되면서 사용하는 다양한 행위를 검사하여 위협이 되는 각 행위의 수준에 따라 가중치를 주어 계산된 점수이다[11].

변종 멀웨어에 대한 이 두 가지 방법의 탐지 결과는 Table 1과 같다.

Table 1. Result of Detector Test

Detector type	Original Malware	Variant Malware
Signature based detector (Virustotal)	detected	**not detected**
Signature and Behavior based detector (Hybrid Analysis)	detected	detected with Threat Score : 60

원본 멀웨어는 시그니처기반의 탐지 도구인 Virustotal과 행위기반의 탐지도구인 Hybrid-Analysis 모두에서 탐지되었다. 그러나 DCM 기법에 의해 생성된 변종 멀웨어는 시그니처 기반의 탐지도구에서는 전혀 탐지되지 못한 반면, 행위기반의 탐지도구에서는 60점의 위협점수(Threat Score)로 탐지되었다.

IV. Conclusions

본 연구에서는 사이버 무기의 관점에서 기존에 만들어진 바이너리 코드 형태의 멀웨어를 활용하여 탐지시스템이 탐지하기 어려운 변종 멀웨어를 자동으로 생성하는 DCM프레임워크를 제안하였다. 이를 위해 Disassemble엔진을 이용하여 어셈블리어 코드를 생성하였고 이 코드를 수정 후 다시 바이너리 코드 형태의 실행파일을 생성한다. 그리고 여기에 난독화를 위한 패킹기법을 적용하는 프레임워크를 제안하고 구현하였다. 그리고 이 프레임워크에 의해 생성된 변종 멀웨어에 대한 탐지 테스트 실험을 진행한 결과 시그니쳐기반의 탐지도구에서는 전혀 탐지가 되지 않는 것도 확인할 수 있었다. 물론 행위 기반의 탐지도구에서는 위협점수 60점으로 탐지되었으나 관건은 시간이다. 행위기반의 탐지 도구에 의해 탐지되는 시간보다 변종 멀웨어가 생성되는 시간이 빠르다면 탐지는 의미 없기 때문이다.

이 연구는 다음과 같이 확장 가능하다. 첫째, Disassemble Engine에서 생성되는 어셈블리어 코드를 활용하여 새로운 코드를 자동으로 생성하는 연구이다. 이미 이를 위한 μGP라는 Python 라이브러리가 만들어져 있으므로 이를 활용하여 적용할 수 있다.

둘째, 본 연구에서 구현된 프레임워크는 하나의 바이너리 멀웨어를 이용하여 하나의 변종을 생성하기 위한 것이었으나 후속 연구에서는 하나의 멀웨어를 이용하여 다중 변종 멀웨어를 생성하는 것으로 확장할 수 있다. 이 또한 새로운 코드를 생성하는 알고리즘을 생성한다면 구현 가능할 것으로 판단된다.

셋째, 본 연구에서는 모든 멀웨어에 적용 가능한 프레임워크를 구현한 것이 아니라 하나의 샘플 멀웨어에 적용가능한 프레임워크를 구현한 것이었다. 이후의 연구에서는 다양한 형태의 보다 많은 멀웨어에 적용가능한 프레임워크로 구현하도록 지속적인 연구가 필요하다.

ACKNOWLEDGEMENT

This work was supported by the Yeungnam University College Research Grants in 2017

REFERENCES

[1] Taeguen Kim, EulGyu Im, “Code reuse analysis method for detecting malicious code variants”, Korea Institute of Information Security and Cryptology, Vol. 24, No 1. pp. 32-38, Feb. 2014, DOI:KIISC.2014.24.1.32

[2] Sungbin park, Minsu Kim, Bongnam Noh, “Detection Method Using Common Features of Malware Variants Generated by Automated Tools”, Journal of Korean institute of information technology Vol. 18 No.8, pp. 81-91, Sep .2020, DOI:10.30693/SMJ.2019.8.4.25

[3] Sadia Noreen, Shafaq Murtaza, M.Zubair, Muddassar Farooq, “Evolvable Malware”, Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp. 1569-1576, Jul. 2009, DOI:10.1145/1569901.1570111

[4] Andrea Cani, Carco Gaudesi, Ernesto Sanchez, “Towards automated malware creation:code generation and code integration”, Proceedings of the 29th Annual ACM Symposium on Applied Computing, pp. 157-160, March. 2014. DOI:

10.1145/2554850.2555157

[5] Manuel Egele, Theodoor Scholte, Engin Kirda, Christopher Kruegel, “A Survey on Automated Dynamic Malware Analysis Techniques and Tools”, ACM Computing Surveys, Vol.. 44, No. 2, pp. 1-42, Feb. 2012, DOI:10.1145/2089125.2089126

[6] Jusop Choi, Dongsoon Shin, Hyoungshick Kim, Jason Seotis, Jin B.Hong, “AMVG:Adative Malware Variant Generation Framework Using Machine Learning”, 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing, DOI:10.1109/PRDC47002.2019.00055

[7] Ilsun You, Kangbin Yim, “Malware Obfuscation Techniques”, 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, Nov. 2010, DOI:10.1109/BWCCA.2010.85

[8] Tomasz Grysztar, “Flat Assembler Documentation and tutorials”, https://flatassembler.net/docs.php

[9] Riccardo Poli, William B. Langdon, Nicholas F.Mcphee, “A Field Guide to Genetic Programming, Jan. 2008, http://www.gp-field-guide.org.uk

[10] Virustotal Service, https://www.virustotal.com

[11] Hybrid-Analysis Service, https://www.hybrid-analysis.com

Authors

Jong-Lark Lee received Ph.D. degree in Statistics from SungKyunKwan University, Korea, in 2012. He is currently an Assisstent Professor in the Faculty of Cyber Security, YeungNam University College.

He is interested in Network Security, Machine Learnig, and Cyber Weapon.

Il-Yong Won received Ph.D. degree in Computer Science and Engineering from Konkuk University, Korea, in 2006. He is currently a Professor in the Department of Cyber Security, Seoul-Hoseo Technical College. He is interested in auto binary analysis and auto malware detection.