Journal of The Korea Institute of Information Security & Cryptology(JKIISC) ISSN 1598-3986(Print)
VOL.23, NO.4, August 2013 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2013.23.4.709

가상화 난독화 기법이 적용된 실행 파일 분석 및 자동화 분석 도구 구현*

석 재 혁,† 김 성 훈, 이 동 훈‡
고려대학교 정보보호대학원

Analysis of Virtualization Obfuscated Executable Files and Implementation of Automatic Analysis Tool*

Jae Hyuk Suk,† Sunghoon Kim, Dong Hoon Lee‡
Graduate School of Information Security, Korea University

요 약

가상화 난독화 기법은 보호하고자 하는 코드영역에 가상화 기법을 적용함으로써 코드의 분석을 어렵게 하는 기법이다. 상용 가상화 난독화 도구로 보호된 프로그램은 가상화된 코드가 원본코드로 복원되는 시점이 존재하지 않고 다양한 난독화 기법으로 가상화 영역이 보호되어 있어 분석하기 어렵기로 잘 알려져 있다. 그러나 이러한 가상화 난독화 기법이 악성코드 보호에 악용되면서 악성코드의 분석 및 대응에 어려움을 겪고 있는 현실이다.

본 논문에서는 상용 가상화 난독화 기법의 핵심 요소들을 자동으로 추출하고 실행 과정을 트레이스 할 수 있는 도구를 구현함으로써 상용 가상화 난독화 도구로 보호되어 있는 악성코드의 분석 및 대응에 활용할 수 있도록 한다. 이를 위하여 가상화 난독화 기법의 기본 구조와 동작 과정을 정리하고, 상용 가상화 난독화 기법으로 보호된 실행 파일을 대상으로 프로그램 분석 기법 중 하나인 Equation Reasoning System을 활용한 분석 결과를 제시한다. 또한 상용 가상화 난독화 도구로 보호되어 있는 실행 파일에서 가상화 구조를 추출하고 프로그램 실행 순서를 도출할 수 있는 자동화 분석 도구를 구현한다.

ABSTRACT

Virtualization obfuscation makes hard to analyze the code by applying virtualization to code section. Protected code by common used virtualization obfuscation technique has become known that it doesn't have restored point and also it is hard to analyze. However, it is abused to protect malware recently. So, It is been hard to analyze and take action for malware.

Therefore, this paper's purpose is analyze and take action for protected malware by virtualization obfuscation technique through implement tool which can extract virtualization structure automatically and trace execution process. Hence, basic structure and operation process of virtualization obfuscation technique will be handled and analysis result of protected malware by virtualization obfuscation utilized Equation Reasoning System, one kind of program analysis. Also, we implement automatic analysis tool, extract virtualization structure from protected executable file by virtualization obfuscation technique and deduct program's execution sequence.

Keywords: Virtualization Obfuscation, Program Analysis, Automatic Analysis Tool

접수일(2013년 4월 18일), 수정일(2013년 6월 19일), 게재확정일(2013년 6월 27일)

* 이 논문은 2012년도 정부(교육과학기술부)의 재원으로 한국연구재단-차세대정보컴퓨팅기술개발사업의 지원을 받아 수행된 연구임(No. 2012-0006419).

† 주저자, sjh2268@nate.com

‡ 교신저자, donghlee@korea.ac.kr(Corresponding author)

I. 서 론

난독화 기법은 프로그램 본래의 기능성은 유지하면서 내부 구조를 변환시킴으로써 역공학을 어렵게 하기 위해 사용되는 기법이다. 최근 악성코드 제작자들이 악성코드 분석가로부터 분석을 지연시키기 위한 목적으로 난독화 기법을 사용하고 있어 문제가 되고 있다. 난독화 기법 중 하나인 가상화 난독화 기법은 보호하고자 하는 코드영역을 잘 알려지지 않은 기계어로 변환(가상화 코드)함으로써 변환된 코드에 대한 기능성을 분석하기 어렵게 만드는 기법이다.

기존 실행코드 암호화 기법이나 패킹(packing) 기법[7]과는 달리 가상화 난독화 기법에 의해 변환된 코드는 원본코드로 복원되는 시점이 존재하지 않으므로 다른 난독화 기법에 비해 분석하기 어렵다. 상용 소프트웨어 보호도구인 Themida[9], Code Virtualizer[8], VMProtect[11] 등에서 사용하는 가상화 난독화 기법은 가상화 구조가 공개되어 있지 않으며 가상화 요소 자체에 적용된 난독화 기법은 분석을 더욱 어렵게 하는 요소로 작용하고 있다.

상용 소프트웨어 보호도구를 이용하면 누구라도 프로그램에 가상화 난독화 기법을 적용할 수 있기 때문에 보호 대상 프로그램을 쉽게 보호할 수 있는 장점이 있다. 그러나 이러한 도구가 악성코드 보호에 악용되면서 문제가 되고 있다. 악성코드에 가상화 난독화 기법이 적용되면 이를 역난독화하기 전까지는 분석이 거의 불가능하며 악성코드 분석에 걸리는 시간이 최대 수백 배까지 증가할 수 있다. 악성코드에 상용 가상화 난독화 기법을 적용하는 것은 쉬운 반면, 이를 역난독화하는 것은 어렵기 때문에 악성코드에 대응하기가 점점 더 어려워지고 있다. 따라서 가상화 난독화 기법이 적용된 악성코드에 대응하기 위한 역난독화 기법 및 분석 기법의 연구는 중요한 이슈가 되고 있다.

기존 연구에서 제시하는 가상화 난독화 분석 방법은 아래와 같이 크게 두 가지로 유형화할 수 있다[3].

- outside-in approach: VM 인터프리터(interpreter) 영역을 우선적으로 분석하는 방법이다. 인터프리터를 역공학한 후 바이트코드의 구조를 분석할 수 있다. 이 방법은 가상화 난독화에 사용되는 인터프리터의 구조를 알고 있는 경우에 적용할 수 있으며 효율적으로 분석하는 것이 가능하지만 인터프리터의 구조를 정확하게 알지 못하는 경우에는 적용할 수 없다는 단점이 있다. R.Rolles[6]의 연구와 M.Sharif[5]의 연구가 이에 해당한다.
- inside-out approach: 프로그램 행위를 우선적으로 분석한다. 이 분석방법은 VM 인터프리터의 구조를 정확히 알지 못해도 적용할 수 있으며, 적용할 때 특별한 가정이 필요하지 않아 어떠한 프로그램에도 적용할 수 있는 일반성이 장점이지만, 행위를 기반으로 분석하기 때문에 분석의 정확도가 높지 않다는 단점이 있다. K.Coogan[3,4]의 연구가 이에 해당한다.

R.Rolles[6]는 가상화 난독화 기법이 적용된 악성코드에도 기존 악성코드 분석기법을 적용할 수 있게 하기 위하여 가상화된 코드를 원본코드 형태로 변환할 수 있는 절차를 소개하였다. 이 연구[6]에서는 원본코드와 구조가 다른 가상화 코드를 IR(Intermediate Representation)로 변환하고 최적화 기법을 적용하여 원본코드와 유사한 형태로 복원함으로써 기존의 분석기법을 적용할 수 있게 한다. M.Sharif[5]는 가상화 난독화 기법이 적용된 악성코드에 대해서 가상화 코드의 구문(syntax)과 의미(semantic)를 추출하고 악성코드의 행위 분석을 위해 제어흐름 그래프를 추출하는 자동화된 역공학 도구인 Rotalume을 구현하였다. Rotalume은 동적 에뮬레이터인 QEMU[2]를 기반으로 제작한 도구로써 우선 동적 에뮬레이터 상에 악성코드를 실행하여 악성코드에서 실행하는 인스트럭션을 모두 로그로 기록한다. 추출한 명령어 로그를 기반으로 메모리 접근 패턴을 분석하고, 이를 통해 VPC(Virtual Program Counter : 가상프로세서상의 프로그램 카운터)를 찾아낸다. 다음으로 바이트코드와 핸들러 코드의 실행 패턴을 분석하여 바이트코드의 구문과 의미를 추출한다. Rotalume의 실행결과를 이용하여 악성코드를 분석할 경우 기존의 제어흐름 그래프를 이용한 행위 기반 분석기법을 적용할 수 있다는 점에서 기여도를 갖는다. 그러나 [5]에서 제안하는 기법은 동적 분석 기법을 기반으로 하였기 때문에 모든 바이트코드의 정보를 추출할 수 있는 것은 아니며, 실행되는 경로에 대해서만 분석이 가능하다는 한계점이 있다. 또한 [5]에서 분석할 수 있는 가상화 난독화 기법은 바이트코드를 해석하는 루틴이 루프(loop) 형태로 핸들러 코드의 외부에 따로 존재하는 디코드-디스패치(decode-dispatch) 방식에 한정된다. K.Coogan[3]은 가상화 난독화 기법이 적용된 악성코드를 분석하기 위해서 어떤 종류의 프로그램이라도 운영체제에서 제공하는 시스템 자원을 이용한다는 사실을 이용한다. [3]에서 제안하는 방법은 시스

템 호출의 입력 값으로 들어가는 인자를 식별하고 분석하는 것이다. 가상화 난독화 기법이 적용된 프로그램이라도 시스템 호출은 일어나기 때문에 그 입력인자 값을 분석함으로써 프로그램의 악성 여부를 진단하는 방법을 제안하였다. 그러나 입력되는 인자 값도 난독화 기법에 의해 근원지를 파악하기 어렵게 되어있기 때문에 K.Coogan은 위의 사실과 함께 Equation Reasoning System[4] 기법을 적용한다. 어셈블리어를 등식과 인덱스로 표현하는 기법인 Equation Reasoning System을 이용하여 시스템 호출에서 사용되는 인자의 근원지를 밝혀내고 분석한 결과를 제시한다. 시스템 호출의 분석과 Equation Reasoning System 기법을 적용하기 위해서는 많은 가정이 필요하지 않기 때문에 다양한 종류의 악성코드 파일에 대해서 제안하는 기법을 적용하는 것이 가능하다는 장점이 있다. 그러나 이 제안 기법은 동적 분석 기법을 기반으로 분석하기 때문에 실행되지 않은 경로의 시스템 호출에 대해서는 분석 기법을 적용할 수 없다는 한계점이 있다.

본 논문에서는 가상화 난독화 기법이 적용된 프로그램의 구조와 동작 방식을 파악하기 위하여 가상화 구조에 대해서 알아보고 가상화 난독화 기법을 적용한 테스트 파일의 분석 결과를 제시한다. 또한 핸들러 코드가 실행되는 순서를 정적으로 계산할 수 있게 한다. 그리고 자동화 분석 도구를 구현하여 가상화 요소와 핸들러 코드를 로그로 추출한다. 가상화 요소와 핸들러 코드 로그를 추출하는 과정은 디스패치 루프 영역을 역공학한 후 이에 동적 분석을 적용하였다. 그리고 핸들러 코드의 실행순서 도출을 정적 분석으로 가능하게 하는 과정은 Equation Reasoning System 기법을 적용함으로써 일반성을 가지게 하였다.

본 논문의 기여도는 아래와 같이 요약할 수 있다.

- 가상화 난독화 기법이 적용된 실행 파일을 분석하기 위하여 상용 가상화 난독화 도구 중 최신버전의 Themida[9]로 보호된 실행 파일을 분석하고, 자동화 분석 도구 구현 결과를 제시한다. 논문에서 제시하는 결과는 악성코드 분석가의 분석 진입장벽을 낮추는 데 기여할 것으로 보인다.
- 본 연구는 분석 대상 파일이 가지고 있는 모든 가상화 요소의 위치를 정확히 파악해내고 실행되는 모든 핸들러 코드를 실행 순서대로 로그 형태로 출력한 결과를 제시한다.
- 가상화 난독화 기법에서는 가상화 코드를 해석하는 루틴을 찾기 어렵게 하기 위하여 여러 단계의 패킹 기법이 적용되어 있다. 이에 본 논문에서 구현한 자동화 분석 도구는 분석 대상 파일을 동적 에뮬레이터인 QEMU[2] 상에서 실행하여 가상화 난독화에서 사용하는 여러 개의 언패킹 루틴을 모두 실행하고 언패킹이 완료된 가상화 요소를 추출하여 제시한다. 이는 분석가가 직접 언패킹 루틴을 실행하지 않아도 분석을 시작할 수 있게 하며, 분석해야할 정보를 우선적으로 제공하기 때문에 분석시간을 크게 줄일 수 있다.
- 가상화 코드를 해석하는 루틴인 디스패치 루프 영역은 분석하기 어렵게 하기 위해 난독화 기법이 적용되어 있으며, 해당 영역에 적용되는 난독화 기법은 프로그램에 가상화 난독화 기법을 적용할 때마다 그 종류가 달라지기 때문에 디스패치 루프는 분석할 때마다 매번 다른 역난독화 기법을 적용하여야 한다. 이에 난독화 기법이 적용된 디스패치 루프에 Equation Reasoning System 기법을 적용함으로써 이의 기능을 한 줄의 등식 표현으로 축약하고, 정적 분석으로 핸들러 코드의 실행순서 도출이 가능하게 하였다. 특히, Equation Reasoning System의 경우 적용할 때 일반성을 가지기 때문에 매번 달라지는 난독화 기법에 대해서도 동일하게 한 줄의 등식 표현으로 축약하는 것이 가능하다.
- 일반적으로 핸들러 코드 영역을 찾는데 성공하더라도 코드 자체가 jmp문에 의해 산재되어 있기 때문에 디버거로 분석할 경우 엄청난 집중력이 요구된다. 본 논문에서는 산재된 핸들러 코드를 연속된 형태의 로그로 출력하여 악성코드 분석가가 분석하기 용이하게 하였다.

본 논문의 구성은 다음과 같다. 2장에서는 가상화 난독화를 분석하기 위해 적용한 Equation Reasoning System 기법에 대해 설명하고, 3장에서는 가상화 난독화의 개념과 가상화 구조를 구성하는 요소에 대하여 설명한다. 4장에서는 가상화 난독화 기법이 적용된 테스트 파일을 분석한 결과와 구현한 가상화 난독화 분석 도구를 이용하여 가상화 요소를 추출한 결과를 제시한다. 5장에서 구현된 결과물의 특징을 제시하며 끝으로 6장에서 결론을 맺는다.

II. 배경지식

2.1 Equation Reasoning System

x86 어셈블리어의 분석을 용이하게 할 수 있는 기

법으로, 기본 원리는 어셈블리어를 등식과 인덱스로 표현하는 것이다. 이 기법은 K.Coogan이 악성코드에서 사용되는 난독화된 조건문 및 점프문을 분석하기 위하여 제안하였으며, 하이레벨 소스코드가 주어지지 않은 환경에서 x86 어셈블리어를 보다 용이하게 분석하기 위해 적용하는 기법이다. 본 논문에서는 이 기법을 디스패치 루프 영역의 분석에 이용한다. Equation Reasoning System 기법은 사용하는 각 레지스터에 인덱싱을 하여 데이터 흐름을 파악하기 쉽게 하며, 해당 인스트럭션이 메모리에 접근하는 경우에는 ValueAt(Memory Address)의 형식을 사용하여 등식으로 표현한다. 이 기법은 인덱스가 높은 인스트럭션에서부터 값을 대입해나가면 여러 줄로 표현되는 어셈블리어도 한 줄의 등식으로 표현이 가능하다는 특징이 있다.

	eax_{31}=0xb
/*I_{31}*/ mov eax, 0xb	$eflags_{32}$=flag(ebx_6 cmp ecx_{15})
/*I_{32}*/ cmp ebx, ecx	esp_{33}=esp_7-4
/*I_{33}*/ pushf	ValueAt(M3000)$_{33}$=$eflags_{32}$
/*I_{34}*/ pop ebx	ebx_{34}=ValueAt(M3000)$_{33}$
/*I_{35}*/ and ebx, 0x1	ebx_{35}=ebx_{34}&0x1
/*I_{36}*/ add eax, ebx	eax_{36}=eax_{31}+ebx_{35}
/*I_{37}*/ jmp [eax*4+0x8]	$target_{37}$=eax_{36}*4+0x8
	➔ $target_{37}$=(0xb+flag(ebx_6 cmp ecx_{15})&0x1)*4+0x8

〔그림 1〕 Equation Reasoning System의 예

〔그림 1〕에서 37번째 인스트럭션(I_{37})은 eax 레지스터를 사용하는 간접점프문이다. 간접점프의 대상주소를 계산하기 위해서는 eax 레지스터의 값을 알아야 하는데 31~36번째 인스트럭션에 난독화가 적용되어 있어 eax의 값을 알기 어렵게 한다. 즉, 점프 대상주소를 알기 위해서는 먼저 31~36번째 인스트럭션의 분석이 이루어져야 하는 것이다. 그러나 〔그림 1〕의 어셈블리어 영역에 Equation Reaso- ning System을 적용하면 대상주소(〔그림 1〕에서 $target_{37}$에 해당)를 한 줄로 표현할 수 있다. 간접점프문이 있는 어셈블리어 영역에 Equation Reasoning System 기법을 적용함으로써 대상주소를 보다 쉽게 계산할 수 있는 것이다.

III. 가상화 난독화

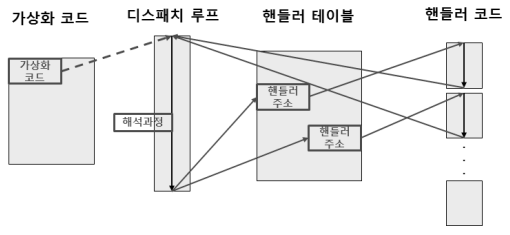
가상화 난독화 기법은 보호하고자 하는 코드영역을 잘 알려지지 않은 다른 종류의 기계어로 변환한다. 다른 종류의 기계어로 변환된 코드를 가상화 코드라고 하며 이러한 가상화 코드를 처리할 수 있는 가상프로세서를 프로그램 내부에 구현한다. 가상화 코드는 CPU가 해석할 수 없기 때문에 프로그램 내부에 소프트웨어적으로 구현되는 가상프로세서를 통해 해석되고 실행된다.

가상화 난독화를 구현하기 위해 필요한 요소는 아래와 같이 크게 네 가지로 분류할 수 있다.

1) 가상화 코드(VM code) 영역: 원본코드를 다른 종류의 기계어로 변환한 가상화된 명령어 집합(instruction set)이 있는 영역이다.
2) 디스패치 루프(dispatch loop) 영역: 가상화 코드를 해석하는 영역으로, 해석된 결과에 따라 해당하는 핸들러 코드로 점프한다.
3) 핸들러 테이블(handler table) 영역: 핸들러 코드의 시작주소가 저장되어 있는 영역이다.
4) 핸들러 코드(handler code) 영역: 실제 실행할 명령을 구현한 핸들러 코드가 있는 영역이다.

가상프로세서가 제어권을 갖게 되었을 때 핸들러 코드가 실행되는 과정은 다음과 같다. 디스패치 루프는 VPC가 가리키는 위치의 가상화 코드를 읽어 들여 해석하고, 해석한 결과를 이용하여 어떤 핸들러 코드를 실행할지 결정한다. 프로세서는 핸들러 테이블을 참조하여 핸들러 코드의 시작주소로 간접점프하며 해당 핸들러 코드를 실행한다. 가상화 코드에 해당되는 핸들러 코드의 실행은 실제 프로세서가 하기 때문에 가상화 코드가 원본코드로 복원되는 시점이 존재하지 않는다. 가상화 요소와 가상화 난독화 동작 과정은 〔그림 2〕와 같이 표현된다.

다른 난독화 기법과 비교할 때 가상화 난독화 기법이 적용된 실행 파일을 분석하는 것은 더욱 어렵다. 가상화 코드의 실행순서나 핸들러 코드를 분석하기 위해서는 가상프로세서를 분석해야 하는데 가상프로세서 내에서 가상화 요소의 위치를 파악하는 것 자체가 쉽지 않기 때문이다. 또한 가상화 요소를 찾아내도 이에 난독화 기법이 적용되어 있거나 가상화 요소의 코드가 jmp문에 의해 여기저기 산재되어 있는 경우도



〔그림 2〕 가상화 난독화 동작 과정 및 가상화 요소

존재한다. 따라서 가상화 요소를 디버거를 통해 분석하는 것은 매우 어렵고 엄청난 집중력이 요구된다. 그리고 가상화 요소에 적용되는 난독화 기법은 실행 파일에 가상화 난독화 기법을 적용할 때마다 매번 그 종류가 달라진다. 그렇기 때문에 가상화 난독화 기법이 적용된 실행 파일을 분석할 때 가상화 요소의 역난독화에 초점을 맞출 경우 하나의 파일을 분석한 결과가 다른 파일을 분석하는데 큰 도움이 되지 않는다. 위와 같은 이유로 가상화 난독화 기법은 다른 난독화 기법에 비해 더욱 분석하기 어렵다.

IV. 가상화 난독화 분석 도구 구현

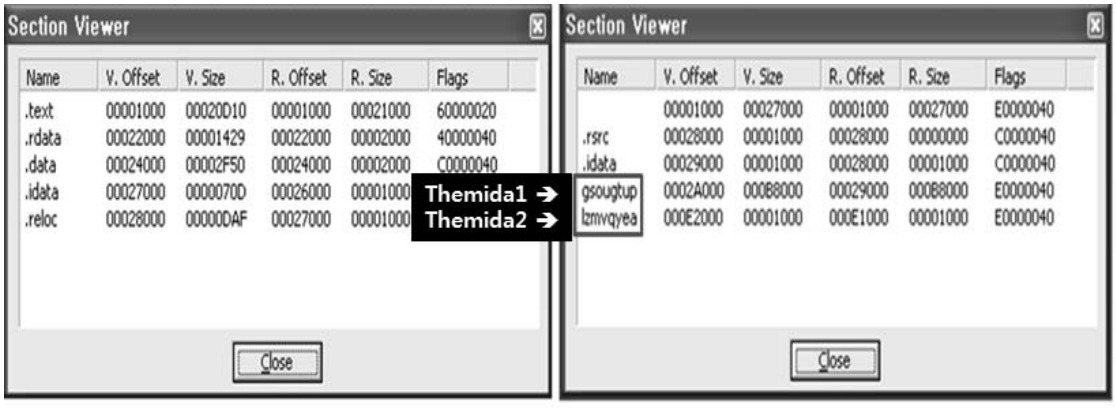
본 장에서는 기존에 제안된 Equation Reasoning System 기법을 가상화 난독화 기법이 적용된 실행 파일에 적용하여 분석하고, 가상화 구조를 자동으로 추출할 수 있는 가상화 난독화 분석 도구를 구현하여 실행 파일을 분석한 결과를 제시한다.

가상화 난독화 분석 및 구현한 분석 도구 테스트환경 구성을 위한 과정은 다음과 같다. 우선 C++을 이용하여 간단한 연산(덧셈, 뺄셈, XOR)을 하는 테스트 프로그램을 작성하였다. 그 다음 테스트 프로그램에 상용 소프트웨어 보호 도구인 Themida v2.1.8로 덧셈, 뺄셈, XOR의 연산을 하는 부분에 CISC 가상화 난독화 옵션을 적용하였다. Themida는 상용 소프트웨어 보호 도구로써 가상화 난독화 이외에도 프로그램을 보호하기 위한 많은 옵션이 존재하지만, 가상화 난독화 분석 도구의 테스트를 위해서 가상화 난독화 기법 이외의 다른 난독화 옵션은 선택하지 않았다. 본 논문에서 테스트하는 가상화 난독화의 옵션은 Mutable CISC Processor, 1 CPU, Static Opcode, Disabled의 최저 옵션으로 적용하였으며, 가상화 난독화의 옵션 강도가 높아져도 전체적인 코드의 분량이 증가할 뿐 분석을 위한 접근 방법 자체에는 큰 차이가 없음을 확인하였다.

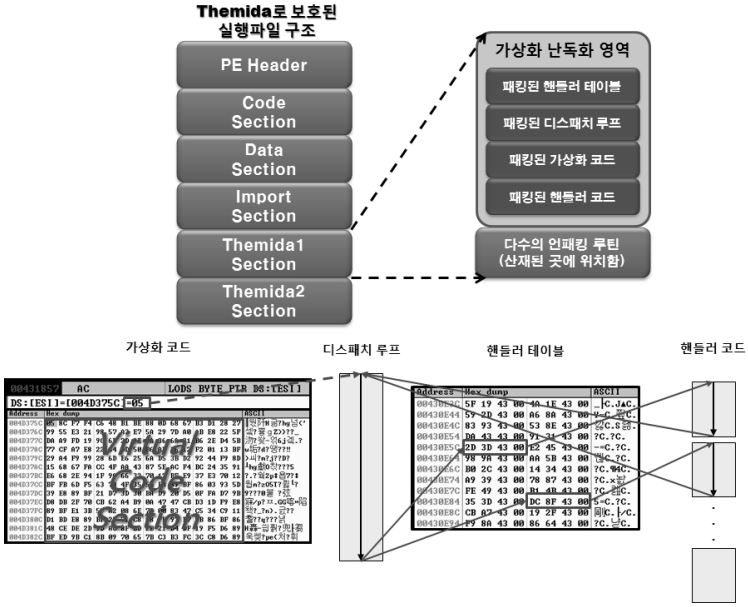
4.1 가상화 난독화 기법의 가상화 구조 분석

가상화 난독화 적용 전과 후의 실행파일의 특징 및 차이를 알아보기 위하여 PEiD[10]로 외형 분석을 실행하였으며 그 결과는 [그림 3]과 같다.

[그림 3]에서 볼 수 있듯이 가상화 난독화 기법 적용 전과 후의 실행 파일은 동일한 개수의 섹션을 가진다는 공통점이 있지만, 적용 후의 파일은 제일 위 섹션의 이름(적용 전 실행 파일에서의 .text 섹션에 해당하며 가상화 난독화 기법이 적용되지 않은 코드가 존재하는 섹션)이 공백이 되고 아래의 두 섹션의 이름이 변형된다. 또한 전체적으로 섹션 크기가 변하는 차이점이 있다. 아래의 두 섹션([그림 3]의 gsougtup 섹션과 lzmvqvea 섹션)은 가상화 난독화 기법을 적용할 때마다 그 이름이 임의로 변한다. 본 논문에서는 두 섹션의 이름을 Themida1([그림 3]에서의 gsougtup 섹션)과 Themida2([그림 3]에서의 lzmvqvea 섹션)라고 하겠다. 디버거를 이용하여 프로그램의 실행 경로를 트레이스 한 결과, Themida2 섹션은 프로그램 실행 시 가장 먼저 실행되는 EP(Entry Point) 섹션임을 알 수 있었다. 이 섹션에는 예외(exception) 코드들이 산재하며, 예외 코드가 실행되면 디버거(debugger)의 동작이 멈추게 되어 분석이 지연된다. 그러나 이러한 예외 코드는 디



[그림 3] 분석 대상 파일의 가상화 난독화 적용 전 PE 파일구조(좌)와 가상화 난독화 적용 후 PE 파일구조(우)



〔그림 4〕 Themida에 의해 가상화 난독화 기법이 적용된 실행 파일의 가상화 구조(위)와 가상화 난독화 동작 과정(아래)

버거에서 제공하는 옵션을 통해 우회할 수 있었으며, Themida2 섹션은 대부분이 분석 지연을 위한 예외 코드로 구성되어있음을 확인하였다.

Themida2 섹션의 실행이 모두 끝나면 Themida1 섹션으로 제어권이 넘어가게 된다. Themida2 섹션은 그 크기가 작고 또한 섹션의 대부분이 예외 코드로 구성되어 있기 때문에, Themida2 섹션에는 가상화 난독화에 필요한 가상화 요소(가상화 코드, 디스패치 루프, 핸들러 테이블, 핸들러 코드)가 존재하지 않음을 알 수 있다. 따라서 가상화 요소는 섹션의 크기가 가장 큰 Themida1 섹션에 존재할 것이라고 가정하고 분석하였다. 최초의 Themida1 섹션에는 패킹 기법이 적용되어 있었으며, Themida1 섹션 내에 존재하는 여러 개의 언패킹 루틴이 실행된 후에 분석할 수 있는 형태가 되었다.

언패킹이 완료된 후, 디버거 상에서 Themida1 섹션을 분석한 결과, 반복적으로 실행되는 구간(A 구간)이 있음을 확인하였다. A 구간이 실행되는 시점에 발견한 특징은 다음과 같다.

- A 구간의 첫 번째 명령어는 esi 레지스터에 저장된 주소에서 1바이트를 읽어 들여 eax 레지스터에 저장하는 lods byte ptr ds:〔esi〕 명령어이다(esi 레지스터 값 1 증가).
- A 구간의 첫 번째 명령어와 마지막 명령어를 제외한 나머지 명령어들은 eax 레지스터에 저장되어 있는 값을 변형하는 명령어들로 구성되어 있으며, 난독화 기법이 적용되어 있다.
- A 구간의 마지막 명령어는 edi 레지스터 값을 base로 하고, 변형된 eax 레지스터 값을 offset으로 하여 특정 구간으로 간접점프를 하는 jmp dword ptr ds:〔edi + eax*4〕 명령어이다(edi 레지스터의 값은 변하지 않음).
- A 구간의 마지막에서 특정 구간으로 점프하고, 구간내의 코드가 실행되고 나면 다시 A 구간의 첫 번째 명령어로 되돌아온다(A 구간으로 되돌아오는 jmp문이 존재).

3장에서 언급하였듯이, 가상화 난독화 기법을 구현하기 위해서는 가상화 코드, 디스패치 루프, 핸들러 테이블, 핸들러 코드의 네 가지 요소가 필요하다. 그 중에서 위 A 구간의 특징을 가지면서 반복적으로 실행되는 구간은 디스패치 루프이다. 디스패치 루프는 가상화 코드를 읽어 들여 이를 해석하고 해석 결과에 따라 핸들러 코드로 점프하여 실행하는 인터프리터(interpreter) 역할을 한다. 가상화 코드 해석이 완료되면 핸들러 코드의 시작주소가 저장되어 있는 핸들러 테이블 영역을 참조하여 간접점프를 하게 된다.

```
00431857   AC          LODS BYTE PTR DS:[ESI]
00431858   30D8        XOR AL,BL
0043185A   83EC 04     SUB ESP,4
0043185D   891424      MOV DWORD PTR SS:[ESP],EDX
00431860   B6 67       MOV DH,67
00431862   FECE        DEC DH
              .
              .
              .
00431954   00CB        ADD BL,CL
00431956   59          POP ECX
00431957   0FB6C0      MOVZX EAX,AL
0043195A   FF2487      JMP DWORD PTR DS:[EDI+EAX*4]
```

〔그림 5〕 Themida1 섹션 내에서 반복적으로 실행되는 구간 (A 구간, 디스패치 루프)

위와 같은 특징에 근거하여 A 구간이 디스패치 루프 영역임을 알 수 있으며 아래와 같은 사실을 알 수 있다.

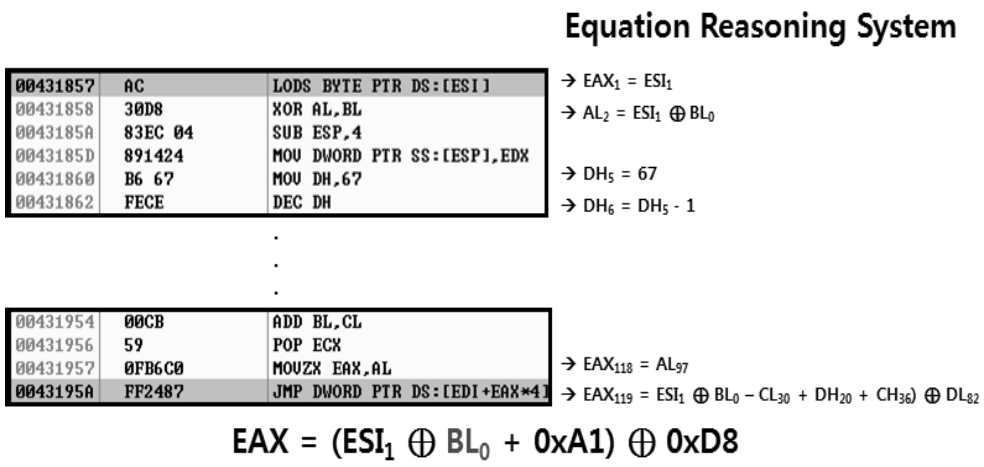
- 디스패치 루프의 첫 번째 명령어는 가상화 코드(esi 레지스터에 가상화 코드 영역 주소가 저장됨)를 읽어 들여서 eax 레지스터에 저장하는 명령어이다.
- 디스패치 루프의 마지막 명령어는 핸들러 테이블을 참조하여 간접점프를 하는 명령어이다.
- esi 레지스터에는 가상화 코드 영역의 주소가 저장되어 있다.
- 가상화 코드는 1바이트로 이루어져 있으며, 이를 디스패치 루프가 해석하는 과정에는 난독화가 적용되어 있다.
- edi 레지스터에는 핸들러 테이블 영역의 base address가 저장되어 있다.
- eax 레지스터에는 읽어 들인 가상화 코드가 저장되어 있으며 이를 해석하여 핸들러 테이블 참조 시 offset으로 사용한다.
- 디스패치 루프에서 간접점프 하는 구간은 핸들러 코드 영역으로, 핸들러 코드의 실행이 끝나면 디스패치 루프로 되돌아간다.

〔그림 4〕는 위의 사실을 근거로 하여 그려낸 가상화 구조와 가상화 난독화 동작 과정이다. 가상화 난독화의 동작 과정과 가상화 구조를 분석하기 위해서는 디스패치 루프의 위치를 정확하게 찾는 것이 중요하다. 디스패치 루프에서 가상화 코드를 읽어 들이고 핸들러 테이블을 참조하여 핸들러 코드로 간접점프하기 때문에, 디스패치 루프의 위치를 정확하게 알게 된다면 나머지 가상화 요소의 위치를 파악하는 것이 가능하다.

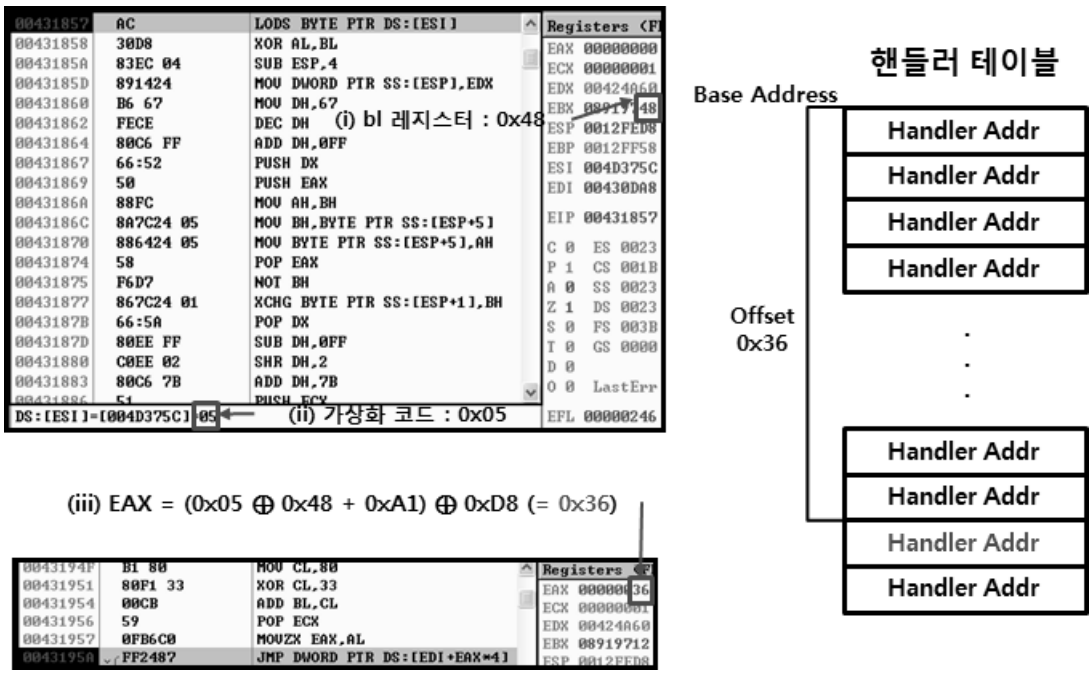
그러나 디스패치 루프에는 난독화가 적용되어 있어서 가상화 코드를 해석하는 과정을 이해하기 어렵게 되어있다. 즉, 디스패치 루프와 가상화 요소의 위치를 찾아내더라도 가상화 코드를 읽어 들이는 시점에서 어떤 핸들러 코드를 실행할지 사전에 계산하는 것은 어렵다는 의미이다. 본 논문에서는 Equation Reasoning System 기법을 디스패치 루프에 적용하여 매번 실행될 핸들러 코드 시작주소를 사전에 계산할 수 있게 해주며 이 결과를 이용하여 가상화 난독화의 핸들러 코드 실행순서를 도출할 수 있다.

4.2 Equation Reasoning System 적용 결과

2장의 배경지식에서 어셈블리어를 등식과 인덱스로 표현하는 Equation Reasoning System 기법을



〔그림 6〕 디스패치 루프에 Equation Reasoning System 기법을 적용한 결과



〔그림 7〕 Equation Reasoning System을 적용하여 핸들러 테이블 offset을 계산한 결과(위)와 실제 실행 결과(아래)

설명하였다. Equation Reasoning System 기법을 난독화된 간접점프문이 있는 코드영역에 적용하면 점프 대상주소를 계산하기 용이하며 본 논문에서는 이 기법을 디스패치 루프에 적용하였다.

〔그림 6〕에서 볼 수 있듯이 Equation Reasoning System 기법을 적용하면 간접점프의 offset (즉, 핸들러 테이블의 offset)으로 사용되는 eax 레지스터 값은 처음에 읽어 들인 가상화 코드(esi 레지스터)와 bl(ebx 레지스터의 하위 1바이트)의 값, 그리고 0xA1, 0xD8의 고정된 hex값에 의해 결정됨을 알 수 있다.

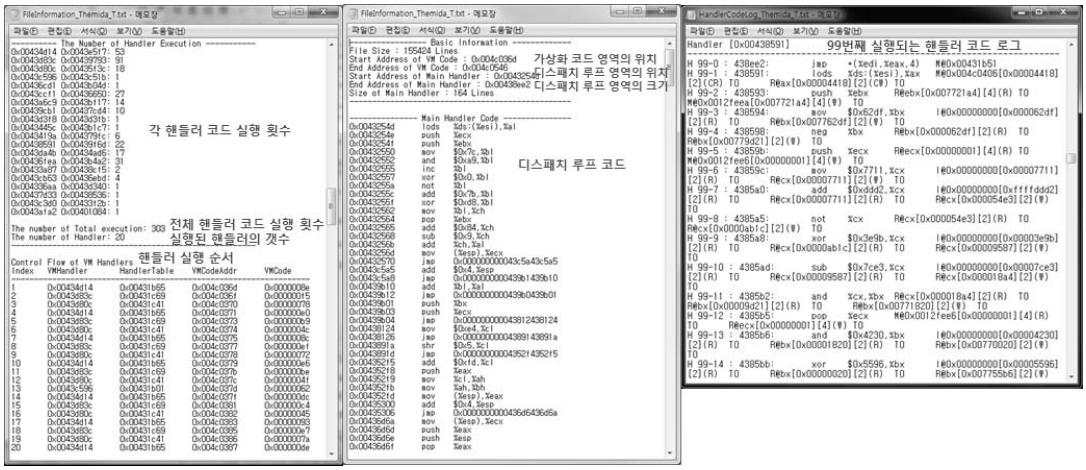
〔그림 7〕을 보면 (i)디스패치 루프 시작시점에서 셋팅 된 bl의 값이 0x48이고 (ii)읽어 들인 가상화 코드가 0x05임을 알 수 있다. (iii)이를 Equation Reasoning System 기법으로 도출한 eax 계산식에 대입하면 다음에 실행할 핸들러 코드 시작주소는 핸들러 테이블 base address(edi 레지스터에 저장된 값)에서 0x36의 offset만큼 떨어진 지점에 저장되어 있음을 계산할 수 있다.

Equation Reasoning System을 디스패치 루프에 적용하여 eax 레지스터의 계산식을 도출하게 되면 매번 실행될 핸들러 코드의 시작주소를 디스패치 루프 영역을 실행하지 않아도 사전에 계산할 수 있다는 장점이 있다. 그리고 이러한 장점을 이용하여 핸들러 코드의 실행 순서를 도출할 수 있다.

4.3 가상화 난독화 분석 도구

가상화 난독화 기법이 적용된 실행 파일을 분석하기 어려운 이유 중 하나는 가상화 요소 자체에도 난독화 기법이 적용되어 있어 이를 분석하는데 많은 시간이 소요된다는 점이다. 테스트 파일 내의 가상화 요소 중 디스패치 루프 영역과 핸들러 코드 영역에 난독화가 적용되어 있는데, Themida를 이용하여 가상화 난독화를 적용할 때마다 이 영역에 적용되는 난독화 기법의 종류가 달라지는 것을 확인하였다. 그렇기 때문에 가상화 요소의 역난독화에 중점을 둘 경우 Themida 적용 결과로 생성되는 파일에 대해서 매번 다른 종류의 역난독화를 적용하여야 하기 때문에 하나의 파일을 분석한 결과가 다른 파일을 분석하는데 크게 도움이 되지 않는다는 문제점이 있다.

위에서 적용한 Equation Reasoning System 기법은 특정한 역난독화 기법과는 달리 diversified obfuscation 기법이 적용된 디스패치 루프에 대해서도 공통적으로 적용할 수 있어서 분석하는데 일반성을 가진다는 장점이 있다. 그러나 Themida에 의해 가



〔그림 8〕 가상화 요소의 위치, 디스패치 루프 코드, 핸들러 실행 횟수 및 실행 순서, 핸들러 코드 로그 출력 결과

상화 난독화 기법이 적용된 실행 파일의 경우, 핸들러 코드 영역과 디스패치 루프 영역에 난독화 기법이 적용되어 있을 뿐 아니라 해당 영역의 코드들이 jmp문에 의해 여기저기 산재되어 있음을 확인하였다. 따라서 분석가가 디버거를 이용하여 분석할 경우 조각난 형태의 코드를 분석하게 되기 때문에 분석에 어려움을 느끼게 되며 Equation Reasoning System 기법을 적용하기 위해 디스패치 루프의 위치를 파악하는 것도 어려워진다.

본 논문에서는 위와 같은 어려움을 해결하기 위해 가상화 요소의 위치와 코드를 자동으로 추출하고 실제로 실행되는 핸들러 코드를 로그로 남기는 가상화 난독화 분석 도구를 구현하였다. 분석 도구의 구현을 위해 동적 에뮬레이터인 QEMU를 사용하였다. QEMU를 이용하여 프로그램 실행시간에 실행되는 명령어를 로그로 생성하고, 가상화 난독화 분석 도구는 이 로그를 입력 값으로 받아 가상화 코드, 디스패치 루프, 핸들러 테이블 영역을 추출하며, 실행되는 핸들러 코드를 로그로 추출한다. 가상화 난독화 분석 도구를 이용하여 추출할 수 있는 정보는 〔그림 8〕과 같다.

〔그림 8〕은 가상화 난독화 분석 도구를 이용하여 가상화 구조의 정보를 추출한 로그를 보여준다. 디버거 상에서 확인할 수 있는 디스패치 루프 코드는 jmp문에 의해 산재되어 있기 때문에 그만큼 분석하는 것이 어렵다. 그러나 가상화 난독화 분석 도구는 동적 에뮬레이터를 사용하였기 때문에 디스패치 루프 코드가 연속적으로 출력되며 디버거로 분석할 때보다 높은 가독성을 제공한다. 또한 핸들러 코드의 실행 순서를 출력함으로써 프로그램의 제어 흐름을 직관적으로 파악할 수 있다.

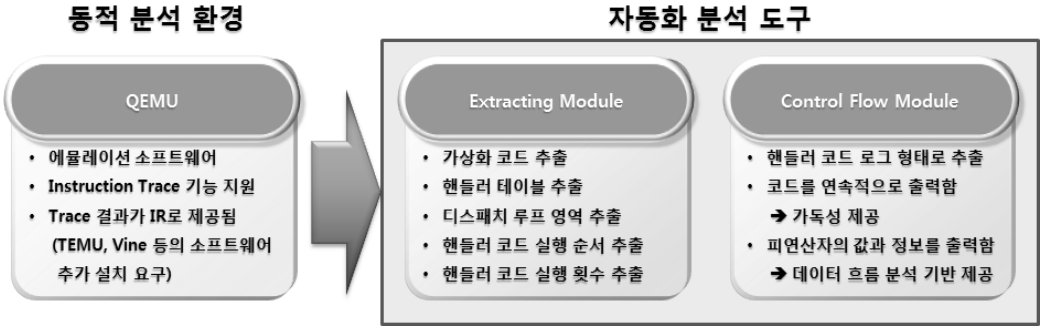
〔그림 8〕의 오른쪽에 보이는 텍스트 파일은 테스트하는 프로그램 내에서 99번째에 실행되는 핸들러 코드 로그를 출력한 결과이다. 핸들러 코드 또한 jmp문에 의해 산재되어 있기 때문에 핸들러 코드 영역의 위치를 정확하게 파악하더라도 이를 분석하는데 어려움이 존재한다. 구현 도구에서 출력하는 로그를 통해 핸들러 코드를 분석하게 되면 디버거 상에서 분석하는 것보다 더욱 높은 가독성을 제공한다.

V. 구현 도구 특징

이 장에서는 실행 파일 분석 결과를 활용하여 구현한 자동화 분석 도구의 특징을 제시한다. 우선 분석 대상 파일을 QEMU 상에서 실행하고 실제로 실행되는 명령어를 로그로 기록한다. 구현한 자동화 분석 도구는 QEMU를 이용하여 생성한 명령어 로그를 입력으로 하여 가상화 요소와 핸들러 코드를 로그 형태로 출력하여 보여준다.

자동화 분석 도구는 추출 모듈(extracting module)과 제어흐름 모듈(control flow module)로 구성되어 있다. 〔그림 10〕과 〔그림 11〕은 추출 모듈 및 제어흐름 모듈에서 제공하는 정보를 보여주는 그림이다.

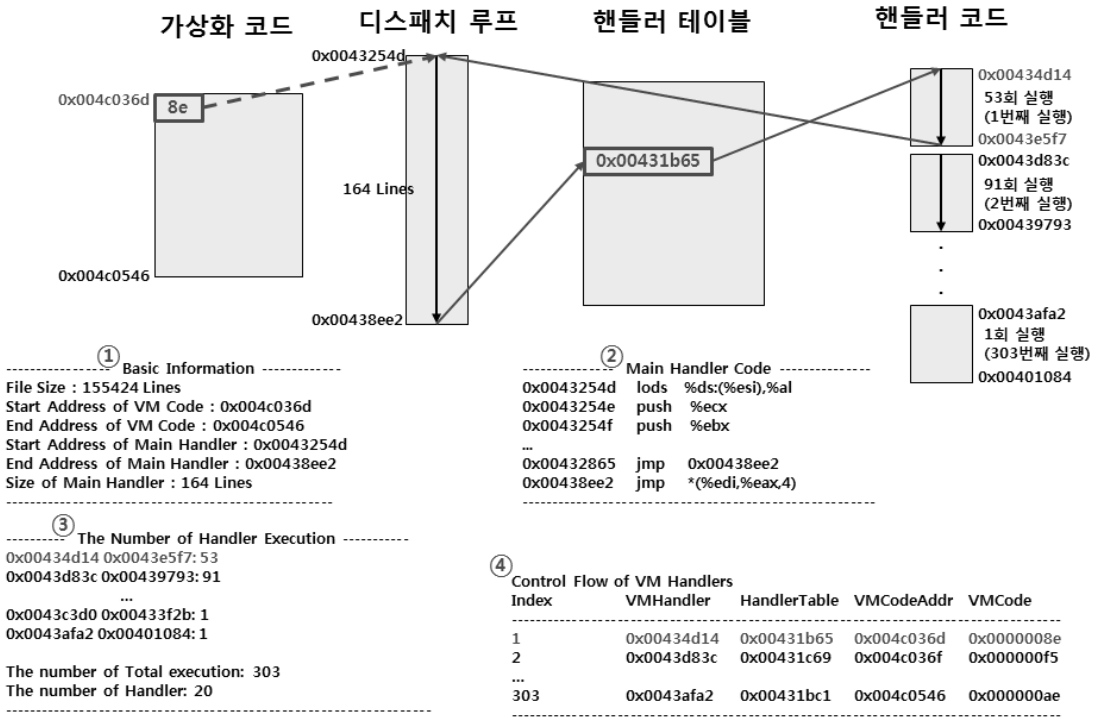
〔그림 10〕에서 볼 수 있듯이 추출 모듈에서는 난독화된 실행파일의 기본 정보 이외에도 디스패치 루프 코드, 핸들러 실행 횟수, 핸들러 실행 순서를 출력한다. 가상화 코드와 디스패치 루프 코드, 실행되는 핸들러 코드 영역을 출력하여 분석가에게 우선적으로 분



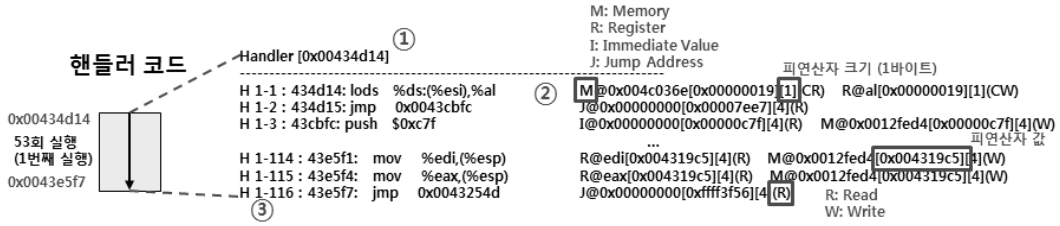
〔그림 9〕 구현한 자동화 분석 도구에서 제공하는 정보

석해야 할 정보를 제공하는 것 또한 구현 도구의 특징이다. 〔그림 10〕의 ③에서 볼 수 있듯이 각 핸들러 코드의 실행 횟수를 출력함으로써 분석가는 많이 실행되는 핸들러 코드를 우선적으로 분석할 수 있다. 또한 〔그림 10〕의 ④에서처럼 핸들러 코드가 실행되는 순서를 제시함으로써 해당 프로그램의 제어흐름을 간략하게 파악할 수 있다.

〔그림 11〕의 제어흐름 모듈에서는 프로그램이 실행되는 순서에 따라 핸들러 코드 로그를 생성한다. 핸들러 코드 또한 디스패치 루프 코드처럼 jmp문에 의해 산재되어 있기 때문에 이를 연속적으로 출력함으로써 분석가가 핸들러 코드의 행위를 분석할 때 높은 가독성을 제공할 수 있다는 것이 장점이다. 또한 핸들러 코드의 주소 뿐 아니라 실행순번 및 라인 수를 함께 출력함으로써 제어흐름을 직관적으로 파악할 수 있다. 예를 들어, 〔그림 11〕은 테스트하는 실행 파일에서 첫 번째로 실행되는 핸들러 코드를 출력한 결과이다. 첫 번째 핸들러 코드는 총 116줄로 구성되어 있으며, 2



〔그림 10〕 추출 모듈에서 출력하는 정보 (① 기본 정보 ② 디스패치 루프 코드 ③ 핸들러 실행 횟수 ④ 핸들러 실행 순서)



〔그림 11〕 제어흐름 모듈에서 출력하는 정보 (① 핸들러 시작주소 ② 명령어 피연산자 정보 ③ 핸들러 실행순번)

번째 줄의 명령어가 jmp 0x0043cbfc임을 확인할 수 있다. 디버거로 분석할 경우 해당 jmp 명령어를 실행하면 점프를 함과 동시에 디버거의 화면이 변하게 되지만, 로그를 이용하여 분석하면 바로 다음 줄인 3번째 줄에서 분석이 가능하게 된다. 또한 핸들러의 실행 순번 및 라인 수가 앞에 있기 때문에 핸들러 코드의 시작과 끝을 쉽게 파악할 수 있다는 점도 가독성을 높여주는 요소이다. 〔그림 11〕의 ②에서는 해당 명령어가 사용하는 피연산자의 타입과 주소, 크기 및 실제 값을 출력하기 때문에 값의 변화를 통해 핸들러 코드의 실행 결과를 분석하는 데이터 흐름 분석(data flow analysis) 기법을 적용하기 용이하다는 장점을 가진다.

VI. 결 론

본 논문에서는 가상화 난독화 기법이 적용된 프로그램에 Equation Reasoning System 기법을 적용하여 가상화 구조를 밝혀내고 이를 분석한 결과를 제시하였으며 가상화 구조를 자동으로 추출할 수 있는 도구를 구현하였다. 본 논문에서 구현한 도구는 가상화 난독화로 보호되어 있는 악성코드를 분석할 때 필요한 정보를 추출하며, 분석가가 직접 복잡한 언패킹 루틴을 거치지 않아도 분석을 시작할 수 있다. 또한 실행되는 핸들러 코드를 산재되지 않은 형태로 제공한다는 점에서 악성코드 분석가에게 큰 도움이 될 것으로 기대된다. 그러나 가상화 구조와 핸들러 코드의 실행 순서, 실제 실행되는 핸들러 코드의 추출에 성공하였어도 추출된 핸들러 코드는 분석가가 따로 분석해야 한다. 핸들러 코드에도 난독화 기법이 적용되어 있어 코드의 양이 최소 수 줄에서 최대 수천 줄이 되기 때문에 이를 분석하는 것은 쉽지 않은 일이다. 따라서 난독화된 핸들러 코드에 역난독화 기법을 적용하여 원본코드와 유사한 형태로 복원할 수 있게 연구하는 것이 향후 과제이다.

참고문헌

〔1〕 C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, July. 1997.

〔2〕 F. Bellard, "QEMU, a fast and portable dynamic translator," In USENIX Annual Technical Conference. USENIX, pp. 41-46, April 2005.

〔3〕 K. Coogan, G. Lu, and S.K. Debray. "Deobfuscation of virtualization obfuscated software: a semantics-based approach," ACM Conference on Computer and Communications Security. ACM, pp. 275-284, Oct. 2011.

〔4〕 K. Coogan, G. Lu, and S.K. Debray. "Equational reasoning on x86 assembly code," Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on. IEEE, pp. 75-84, Sep. 2011.

〔5〕 M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," In Proc. of the 30th IEEE Symposium on Security and Privacy, pp. 94-109, May 2009.

〔6〕 R. Rolles, "Unpacking virtualization obfuscators," In Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT '09), pp. 1-1, Aug. 2009.

〔7〕 M.V. Yason, "The Art of Unpacking," Blackhat USA 2007.

〔8〕 Oreans Technologies. Code virtualizer:

Total obfuscation against reverse engineering, Dec. 2008. http://www.oreans.com/codevirtualizer.php.
〔9〕 Oreans Technologies. Themida: Advanced Windows Software Protection System, Jul. 2012. http://www.oreans.com/themida.php.
〔10〕 PEiD. 2009. http://www.peid.info,
〔11〕 VMProtect Software. VMProtect software protection, 2008. http://vmpsoft.com/.

〈저자소개〉

석 재 혁 (Jae Hyuk Suk) 학생회원
2012년 2월: 서울시립대학교 전자전기컴퓨터공학부 졸업
2012년 2월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 소프트웨어 분석, 소프트웨어 난독화

김 성 훈 (Sung Hoon Kim) 학생회원
2006년 8월: 서울시립대학교 수학과 졸업
2009년 2월: 고려대학교 정보보호대학원 석사 졸업
2009년 1월~2011년 2월: (주)알티캐스트 CAS개발본부 전임연구원
2011년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 소프트웨어 보안, 소프트웨어 난독화

이 동 훈 (Dong Hoon Lee) 정회원
1983년 8월: 고려대학교 경제학사 졸업
1987년 12월: Oklahoma University 전산학과 석사 졸업
1992년 5월: Oklahoma University 전산학과 박사 졸업
1993년 3월~1997년 2월: 고려대학교 전산학과 조교수
1997년 3월~2001년 2월: 고려대학교 전산학과 부교수
2001년 3월~현재: 고려대학교 정보보호대학원 교수
〈관심분야〉 암호프로토콜, 암호이론, USN이론, 키 교환, 익명성 연구, PET 기술