# 가상화 기법으로 난독화된 실행 파일의 동적 분석을 통한 핸들러 간략화

Simplification of Handler by Dynamic Analysis of Virtualization-Obfuscated Binary Executables


저자
(Authors)

원지혜, 한태숙
Jihye Won, Taisook Han



출처
(Source)

한국정보과학회 학술발표논문집 , 2014.6, 1621-1623 (3 pages)

발행처
(Publisher)

한국정보과학회
KOREA INFORMATION SCIENCE SOCIETY

URL

http://www.dbpia.co.kr/Article/NODE02444475

APA Style

원지혜, 한태숙 (2014). 가상화 기법으로 난독화된 실행 파일의 동적 분석을 통한 핸들러 간략화. 한국정보과학회 학술발표논문집, 1621-1623.

# 가상화 기법으로 난독화된 실행 파일의 동적 분석을 통한 핸들러 간략화†

원지혜○‡ 한태숙‡
‡한국과학기술원 전산학과
jhwon@kaist.ac.kr han@cs.kaist.ac.kr

# Simplification of Handler by Dynamic Analysis of Virtualization-Obfuscated Binary Executables

Jihye Won○‡ Taisook Han‡
‡Department of Computer Science, KAIST

**요 약**

최근 악성코드는 난독화 뿐만 아니라 가상화 기법을 사용하여 임의의 가상머신에서 해석되는 바이트코드로 변환된다. 가상화로 난독화된 실행 파일은 원본 프로그램과 동일한 의미를 가지므로 이를 기반으로 실행 파일 내부의 가상머신 구조 분석에 관한 연구가 진행되었다. 하지만 바이트코드의 의미를 정확하게 분석하기 힘들다는 한계가 있기 때문에 본 논문에서는 기존 연구를 바탕으로 바이트코드에 대응되는 가상화 바이트코드 핸들러를 간략화하는 기법을 소개한다. 따라서 분석자가 바이트코드의 의미를 분석하는 과정에서 단순 반복적인 작업을 줄여줌으로써 분석 시간을 줄일 수 있을 것이라 기대한다

## 1. 서 론

코드 난독화란 원본 프로그램의 기능은 유지하되 그 의미를 해석하기 어렵게 만드는 기법을 의미한다. 코드 난독화는 프로그램의 저작권 보호를 위해 사용될 수 있으나, 악성코드 제작자는 이를 악용하여 보안 분석가의 해석을 방해하는데 사용한다. 최근에는 상용 난독화 도구인 VMProtect[1]와 Code Virtualizer[2]가 코드 가상화와 난독화를 혼용하는 기법을 제공하고 있어 악성코드 분석을 더욱 어렵게 만들고 있다.

가상화 기법은 원본 실행 파일에 알려지지 않은 구조의 가상머신과 해당 가상머신에서 실행되도록 인코딩된 바이트코드를 끼워 넣어 난독화된 실행 파일을 생성하는 기법이다. 그림 1은 가상화 기법으로 난독화된 실행 파일 구조의 예를 보여준다. 가상화 섹션(VM Section)에는 각 바이트코드에 대응되어 실제로 실행될 명령을 저장할 가상화 바이트코드 핸들러, 이 핸들러의 주소가 저장된 가상머신 Import Address Table(VM IAT), 인코딩된 바이트코드, 인터프리터가 있다. 인터프리터가 바이트코드를 에뮬레이션(emulation)하는 과정은 다음과 같다. 인터프리터에서 다음에 실행할 바이트코드가 저장된 메모리 위치를 가리키는 Virtual Program Counter(VPC)로부터 바이트코드를 읽어오고, 이를 이용하여 IAT로부터 다음에 실행할 핸들러의 주소를 가져 온 후, 디스패치 포인트에서 핸들러로 점프를 수행한다. 따라서 난독화에 사용되는 가상머신의 구조를 파악하고 가상화 바이트코드 핸들러를 복구하는 과정은 분석가가 가상화 기법으로 난독화된 실행 파일의 의미를 해석하는 데 필요한 과정이다.

가상머신 분석을 기반으로 한 난독화 해제 기법과 관련된 선행 연구로는 Rolles[3]와 Sharif[4]의 연구가 있다. Rolles는 분석자가 직접 내부 가상머신의 구조를 파악하고, 프로그램의 바이트코드와 그에 대응되는 가상머신의 핸들러를 알아낸다. 그리고 핸들러에 컴파일러 최적화 기법을 적용하여 x86코드를 생성하는 기법을 소개하였다. 하지만 모든 가상머신 구조의 특성을 이해해야 하고 대부분의 분석 과정이 수동적이기 때문에 많은 시간과 노력이 요구된다. Sharif 연구팀은 프로그램의 실행 트레이스를 기록하여 본 연구와 같은 에뮬레이션 모델을 가정하고 바이트코드의 구문과 의미를 분석하여 제어 흐름 그래프를 복구하는 방법을 제시하였다. 그리고 이성호[5]는 이를 바탕으로 중간언어 변환 모듈을 사용하여 인터프리터와 VPC, 바이트코드 핸들러 등을 추출하였다. 본 연구는 선행 연구를 기반으로 가상화 핸들러 간략화하는 기법들을 제안하고자 한다. 간략화된 핸들러를 활용한다면 바이트코드의 의미를 요약하여 x86어셈블리코드로 변환하는 추후 연구에 도움을 줄 수 있을

† 본 연구는 2013년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (NRF-2011-0023847)

것이다.

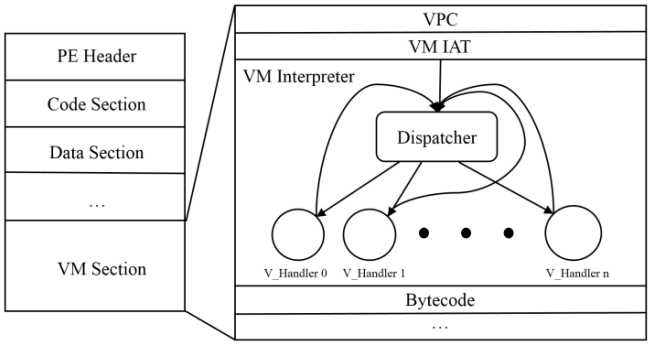


그림 1 가상화 기법으로 난독화된 실행 파일 구조

## 2. 분석 과정

본 연구에서는 다음과 같은 분석 과정을 수행한다.

1) 동적 실행 트레이스를 추출하여 실행된 명령어와 가상머신 구조 분석에 사용할 데이터를 기록한다.

2) x86-mBIL 변환 모듈을 개발하여 추출된 명령어를 Binary Analysis Platform(BAP)[6] 수정된 중간언어(modified BIL)로 변환한다.

3) 동적 제어 흐름 그래프를 생성하여 인터프리터, VPC, 핸들러 리스트 후보를 찾는다.

4) 의미 없는 명령어를 제거하고 임시 변수를 정리하여 난독화된 핸들러를 간략화한다.

## 3. 동적 실행 트레이스 추출

인텔 Pin[7]을 기반으로 개발된 트레이스 추출 도구는 실행 파일의 동적 실행 트레이스 파일을 만든다. 결과 파일에는 실행된 x86 명령어 헥사코드와 주소, 제어 흐름 변경시 타겟 주소, 메모리 입출력시 메모리 주소와 값을 담고 있다.

## 4. BAP 중간 언어 변환 모듈

이전 단계에서 기록한 헥사코드 형태의 x86 명령어는 어셈블리코드로 변환하는 디스어셈블 과정을 거치고, 다시 mBIL Abstract syntax tree로 변환된다. 수정된 BAP 중간 언어는 하나의 x86 명령어를 여러 개의 단순한 구문의 표현한다. 또한 메모리 주소와 메모리 값을 문법에 포함하여 분석에 사용할 수 있도록 수정하였고 시스템 상태의 변화를 명시적, 세부적으로 표현한다. 자세한 내용은 선행 연구 논문[5]에서 살펴볼 수 있다.

## 5. 난독화된 실행 파일 구조 분석

실행 트레이스로부터 동적 제어 흐름 그래프를 생성하고 보정하여 나가는 선(out edge)이 가장 많은 노드를 중심으로 인터프리터를 정한다. 그런 다음 인터프리터 베이직 블록을 mBIL 변환하고, 불필요한 명령어를 제거하기 위해 역방향 슬라이싱을 적용하여 VPC를 추출한다. 핸들러는 원본 실행 파일의 의미와 동일하게 실행되는 바이트코드에 대응되어 수행하며 특정 값을 변경한 후 다시 인터프리터로 실행 흐름을 옮긴다. 따라서 동적 제어 흐름 그래프에서 인터프리터를 기준으로 사이클(cycle)을 이루는 베이직 블록의 집합을 수집하고 VPC와 바이트코드 변형 키 값에 대한 명령어만을 기록하여 핸들러 리스트 후보를 검출한다.

## 6. 핸들러 간략화

### 6.1 Copy propagation

Copy propagation 기법에서 copy 란 x = y 형태의 식으로 x는 임시 변수이고 y라는 값이 할당된 것이다. x에 대한 다른 정의가 없을 때까지 변수 x가 사용된 표현식에 y값으로 대체하는 작업을 반복한다. 예로 그림 2는 가상화 핸들러의 일부분이다. 그림 2의 핸들러에 Copy propagation 기법을 적용하면 그림 3과 같이 간단해진다.

```
T_t:u32 = R_EDX:u32
mem:u32 = mem:u32 with [R_ESP:u32, 18fe14 (1),
e_little]:u32 = T_t:u32
T_t1:u32 = R_ESI:u32
T_t2:u32 = 0x3e677818:u32
R_ESI:u32 = T_t1:u32 + T_t2:u32
```

그림 2 Copy propagation 기법을 적용하기 전 핸들러

```
mem:u32 = mem:u32 with [R_ESP:u32, 18fe14 (1),
e_little]:u32 = R_EDX:u32
R_ESI:u32 = R_ESI:u32 + 0x3e677818:u32
```

그림 3 Copy propagation 기법을 적용한 후 핸들러

### 6.2 의미 없는 스택 연산 제거

의미 없는 스택 연산이란 그림 4에서와 같이 레지스터 ESP 에 4를 더하고 빼는 연산을 의미한다. 다른 명령어가 스택의 메모리 영역을 사용하지 않으므로 제거할 수 있다. 그림 4의 핸들러는 가상화 핸들러의 일부분이다. 그림 5는 그림 4의 핸들러에서 의미 없는 스택 연산을 제거한 후의 핸들러이다. 또한 스택 포인터가 가리키는 위치는 의미 없는 스택 연산을 제거하기 이전과 동일하다.

```
R_ESP:u32 = R_ESP:u32 - 0xbc:u3
R_ESP:u32 = R_ESP:u32 - 0x4:u32
R_ESP:u32 = R_ESP:u32 - 0x4:u32
R_ESP:u32 = R_ESP:u32 + 0x4:u32
R_ESP:u32 = R_ESP:u32 + 0x28:u32
R_ESP:u32 = R_ESP:u32 + 0x4:u32
```

그림 4 의미 없는 스택 연산 제거하기 전 핸들러

```
R_ESP:u32 = R_ESP:u32 - 0xbc:u3
R_ESP:u32 = R_ESP:u32 + 0x28:u32
```

그림 5 의미 없는 스택 연산 제거한 후 핸들러

## 7. 구현 및 실험

### 7.1 구현

본 논문에서 소개된 트레이스 추출 도구와 BAP 중간 언어 변환 모듈, 분석 알고리즘은 32비트 인텔 아키텍쳐 컴퓨터에서 구동되는 윈도우7 운영체제에서 실행되도록 만들어진 실행 파일을 대상으로 구현하였다.

트레이스 추출 도구는 Pin version 2.12.56759 로 개발되었고, 분석하고자 하는 실행 파일을 실행시켜 특정 정보가 담긴 실행 트레이스를 결과로 얻을 수 있다.

BAP 중간 언어 변환 모듈내 디스어셈블러는 distorm3.3[8]을 사용하였다.

### 7.2 실험

핸들러 간략화의 유용성을 측정하기 위해 그림 6과 같이 이중 반복문과 하나의 분기문이 있는 프로그램을 작성하여 컴파일하였다. 이 프로그램을 Code Virtualizer로 난독화한 프로그램을 첫 번째 실험 프로그램으로, VMProtect로 난독화한 프로그램을 두 번째 실험 프로그램으로 사용하였다.

첫 번째 실험 프로그램을 선행 연구 방법으로 검출한 모든 핸들러 리스트의 명령어 파일의 크기는 50KB, 명령어의 수는 1,010개였다. 하지만 본 연구에서 구현한 핸들러 간략화 기법을 수행하여 검출한 모든 핸들러 리스트의 명령어 파일의 크기는 42KB, 명령어 수는 711개였다. 두 번째 실험 프로그램을 선행 연구 방법으로 검출한 모든 핸들러 리스트의 명령어 수는 116개, 핸들러 간략화 기법을 수행하여 검출한 모든 핸들러 리스트의 명령어 수는 81개였다. 두 핸들러 리스트의 명령어 파일의 크기는 6KB였다.

결과적으로 두 실험 프로그램의 난독화된 핸들러 명령어의 약 30%가 감소했음을 알 수 있었고, 간략화 이전의 핸들러의 명령어와 의미적으로 동일하다는 것을 확인하였다.

```
void main(){
 int i, re=0;
 for(i=0; i<10; i++){
         for(int j=0; j<10; j++){
           if(i%2 == 0)      re += j;
           else    re -= j;
           }
  }
 printf("%d\n",re);
}
```

그림 6 실험 프로그램의 원본 소스 코드

## 8. 결론 및 향후 연구

본 논문은 가상화 기법으로 난독화된 실행 파일의 동적 분석을 바탕으로 가상화 바이트코드 핸들러의 의미는 유지하면서 간략화하는 기법을 제안하였다. 하지만 간략화 알고리즘을 개선하여 핸들러 명령어 감소율을 높여야 할 것이다. 예로 Constant propagation[9]과 같은 컴파일러 최적화 기법을 적용할 수 있다.

향후 간략화된 핸들러를 활용하여 원본 프로그램의 바이트코드 의미를 정확하게 요약 분석할 수 있을 것이라 기대한다. 또한 그 결과로 x86어셈블리코드로 변환하는 기술을 추가적으로 연구할 계획이다. 본 연구는 향후 연구를 수행하는 데에 있어서 분석자가 바이트코드의 의미를 분석하는 과정에서 단순 반복적인 작업을 줄여줌으로써 분석 시간을 단축할 수 있을 것이라 기대한다.

## 참고문헌


[1] VMProtect Software, VMProtect Software.

[2] Oreans Technologies, Code Virtualizer.

[3] Rolf Rolles, “Unpacking Virtualization Obfuscators,” in *Proceedings of the 3rd USENIX conference on Offensive technologies*, 2009.

[4] M. Sharif, et al., “Automatic Reverse Engineering of Malware Emulators,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[5] 이성호, “가상머신 기반으로 난독화 된 실행파일의 구조 및 원본의미 추출 동적 방법,” 석사학위논문, 한국과학기술원, 대전, 대한민국, 2014.

[6] David Brumley, et al., “The BAP Handbook”, 2014.

[7] C.-K. Luk, et al., “Pin: building customized program analysis tools with dynamic instrumentation,” in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 190-200, 2005.

[8] Gil Dabah. distorm.

[9] A. Aho, et al., “Compilers: Principles, Techniques, and Tools,” pp.583-638, Pearson/Addison Wesley, 2007.