Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.27, NO.1, Feb. 2017 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2017.27.1.59

역난독화를 위한 바이너리 프로그램 슬라이싱*

목 성 균,† 전 현 구, 조 은 선‡
충남대학교

Program Slicing for Binary code Deobfuscation*

Seong-Kyun Mok,† Hyeon-gu Jeon, Eun-Sun Cho‡
Chungnam National University

요 약

해커들이 자신들이 만든 악성코드의 분석을 어렵게 하기 위하여 코드 난독화 기법을 적용하고 있다. 최근의 난독화 기법은 가상화 난독화 기법을 통해 원래의 코드를 바이트코드로 만들고 가상머신이 이를 실행시키는 방식으로, 실행시키기 전에는 원래의 코드를 알 수가 없다. 프로그램 슬라이싱은 프로그램 분석기술 중 하나로 슬라이싱 기준을 정하고 그와 관련된 문장을 추출해내는 기술이다. 본 논문에서는 슬라이싱 기법을 사용하여 난독화를 해제하는 방법을 제시한다.

ABSTRACT

Hackers have obfuscated their malware to avoid being analyzed. Recently, obfuscation tools translate original codes into bytecodes to use virtualized-obfuscation, so that bytecodes are executed by virtual machines. In such cases, malware analysts fail to know about the malware before execution of the codes. We found that program slicing is one of promising program analysis techniques to solve this problem. The main concepts of program slice include slicing criteria given by analysts and sliced statements according to the slicing criteria. This paper proposes a deobfuscation method based on program slicing technique.
Keywords: Obfuscation, Deobfuscation, Program Slicing, Dynamic binary analysis

I. 서 론

난독화는 프로그램 원래의 의미는 유지하면서 코드의 역공학을 어렵게 바꾸는 기법을 의미한다. 이러한 기법은 소프트웨어의 핵심 알고리즘을 보호하기 위해 사용하기도 하지만 악성코드의 제작자가 자신들의 코드를 분석하기 어렵게 하기 위해서 사용하기도 한다.

난독화는 비교적 쉬운 명령어를 이해하기 어려운 명령어로 바꾸거나 의미 없는 코드를 삽입하는 방식으로 이루어진다. 예를 들면 'mov eax, ecx'를 'push ecx, pop eax'와 같이 변환한다. ecx의 값을 eax에 대입하는 코드이다. 'push ecx, pop eax'는 스택을 거치지만 결국은 ecx의 값이 eax로 대입하는 코드이다. 이와 같은 방식으로 잘 사용되지 않는 방식으로 코드를 변환하여 분석을 어렵게 한다. 하지만, 최근에는 이러한 난독화 방식을 유지하면서 가상화 난독화 방식이나 코드 자가 변경(self modifying code) 기법을 사용한다. 가상화 난독화는 원래의 프로그램 코드를 바이트 코드로 만들고, 프로그램에 삽입된 가상머신이 이를 실행한다. 코드 자가 변경 기법은 실행 중에 코드를 변경하여 변경한 코드를 실행한다. 즉, 다음에 실행할 코드 주소에 특정 값을 연산하여 원래의 실행할 코드로 바꾼다. 두 가지 기법의

Received(09. 23. 2016), Modified(01. 03. 2017), Accepted(01. 04. 2017)
* 본 논문은 2016년도 하계학술대회에 발표한 우수논문을 개선 및 확장한 것임
* 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학 ICT연구센터육성지원사업의 연구결과로 수행되었음. (IITP-2016-R2718-16-0003)
* 본 연구는 국가 보안 기술 연구소 과제의 연구결과로 수행되었음.
† 주저자, mok7764@cnu.ac.kr
‡ 교신저자, eschough@cnu.ac.kr(Corresponding author)

공통점은 정적 분석으로는 프로그램 원래의 코드를 알 수 없다는 것이다. 정적 분석을 사용하여 역난독화를 진행한 연구가 있긴 하지만 특정 난독화 도구에서 사용하는 가상머신의 구동 방법을 파악하여 난독화를 해제하는 방법이다[14]. 즉, 언급한대로 특정 난독화 도구에만 해당이 되므로, 만약 난독화 도구가 해당 연구를 회피하기 위한 패치가 된다면 해당 역난독화 방법은 사용할 수 없다.

난독화 해제 연구 중 가장 두드러진 연구는 시스템 콜을 중심으로 테인트 분석을 하여 의미 있는 명령어를 추출하는 연구이다. 본 논문에서는 이와 유사한 접근 방법을 사용해서 프로그램 슬라이싱을 통한 난독화 해제 방법을 제시하고자 한다.

II. 슬라이싱

슬라이싱은 프로그램의 분석 기술 중 하나로 프로그램의 일부 컴포넌트에 초점을 맞추는 방법이다[3]. 슬라이싱 기준(slicing criterion)을 정하고 그와 관련되는 프로그램 문장(statement)들을 추출해 내는 것이다. 여기서 슬라이싱의 기준은 변수 또는 특정한 문장이 될 수 있다. 기준을 정한 뒤, 기준에 영향을 주는 문장들을 추출한다. 어떤 지점의 변수가 기준일 경우, 변수에 영향을 주는 모든 변수들을 추출한다. 다음 표는 파이썬으로 작성된 슬라이싱 예제 코드이다. 슬라이싱의 기준은 S10의 y 변수이다. 표 1이 슬라이싱 대상 프로그램이면, 표 2는 슬라이싱 된 결과이다.

Table 1에서 S10을 슬라이싱 기준으로 하고, 프로그램의 역으로 가면서, S10에 영향을 준 모든 문장들을 추출해낸다. S8, S6과 S3에서 x가 y에 영향

Table 1. Before slicing example code[8]

```
S1      x = input
S2      if (x < 0):
S3          y = f1(x)
S4          z = g2(x)
        else:
S5          if ( x = 0):
S6              y = f2(x)
S7              z = g2(x)
            else:
S8              y = f3(x)
S9              z = g3(x)
S10     print(y)
S11     print(z)
```

Table 2. Sliced result of Table 1[8]

```
S1   x = input
S2   if (x < 0):
S3       y = f1(x)

     else:
S5       if ( x = 0):
S6           y = f2(x)


S8           y = f3(x)

S10  print(y)
```

을 주었기 때문에, 여기서 부터는 y가 아닌 x에 대해서 추출해낸다. 여기서, 주목할 점은 제어 의존성이다. S6와 S8은 S5의 결과에 따라 실행 여부가 결정된다. 즉, S6와 S8은 S5에 제어 의존성 관계를 가지고 있다. S3 역시 S2에 대해서 제어 의존성 관계를 가지고 있다. 그러므로 Table 2와 같은 슬라이싱 결과가 나오게 된다.

2.1 슬라이싱 전처리 작업

슬라이싱은 전처리 작업으로 PDG(Program Dependece Graph)를 만든다[4]. PDG는 DDG(Data Dependency Graph)와 CDG(Control Dependency Graph)가 합쳐진 형태이다. DDG는 한 명령어가 있을 때, 이 명령어에서 정의된 데이터가 다른 데이터에 영향을 주는 명령어들을 나타내게 된다. DDG는 한 명령어가 있을 때, 이 명령어에 영향을 주는 다른 명령어들을 찾을 때 사용된다. DDG에서 노드는 프로그램의 문장이 되고, 엣지는 데이터 의존성을 나타낸다. CDG는 제어 흐름의 의존성을 나타낸다. 노드는 베이직블록 또는 프로그램의 문장이 되고, 엣지는 제어 의존성을 나타낸다. 따라서 PDG는 데이터 의존성과 제어 의존성을 모두 볼 수 있다는 장점이 있다. PDG에서도 노드는 프로그램의 문장이 되고, 엣지는 데이터 의존성 또는 제어 관계 의존성이 된다.

Fig 1은 Table1의 코드에 대한 PDG이다. `S10`을 기준으로 슬라이싱 한다고 하면, `S10`의 엣지를 따라가는 방식으로 슬라이싱을 할 수 있다. 결과로는 Fig 2와 같이 {`S1, S2, S3, S5, S6, S8, S10`}을 얻어낼 수 있다.

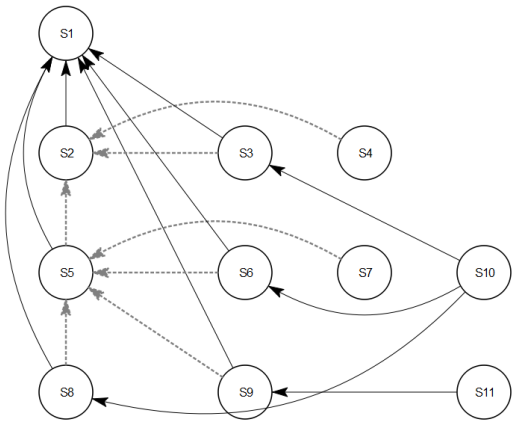


Fig. 1. PDG of Table 1[8]. Solid line is data dependency. Dotted line is control dependency.

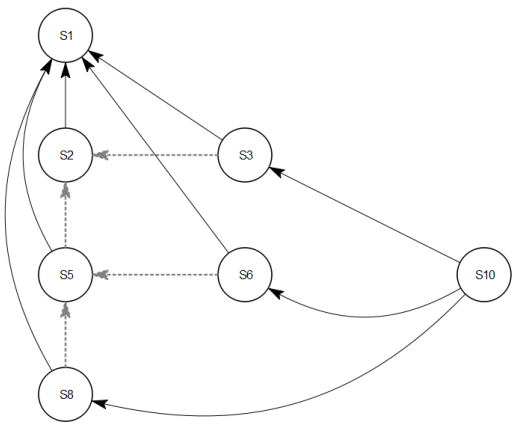


Fig. 2. Result of Fig1's slicing[8]

2.2 정적 슬라이싱

정적 슬라이싱은 PDG를 바탕으로 기준이 되는 문장으로부터 영향을 받은 모든 문장을 추출한다. 정적 슬라이싱을 바이너리에 적용할 경우, 점프 주소가 보이지 않는 간접적인 제어 전이, 메모리 주소를 알 수 없어 부정확한 분석이 될 확률이 크다. 난독화된 프로그램은 코드 자가 변경 기법이 적용되어 프로그램 원래의 코드가 보이지 않기 때문에 정적 슬라이싱은 난독화된 프로그램에는 적용할 수 없다.

2.3 동적 슬라이싱

동적 슬라이싱은 프로그램 실행 동안에 슬라이싱을 하기 때문에, 정적 분석과 달리 제어 전이 주소, 메모리 주소 같은 정보 부족 현상이 없다[8]. 프로그램 실행 경로는 이미 정해져 있기 때문에 제어 흐름에 관한 정보도 불필요 하다.

동적 슬라이싱은 실행동안에 이루어지기 때문에 하나의 실행경로에 대해서만 이루어진다. Table 1의 코드에서 x가 1이라고 하면 실행 경로는 {S1, S2, S3, S4, S10, S11}과 같다. 그리고 여기에서 S10에 대해 슬라이싱을 하면 Fig 3과 같이 {S10, S3, S2, S1}이 나온다. 하지만 정적 슬라이싱의 결과인 Fig 2와 비교했을 때 하나의 실행경로에 대해서만 이루어지기 때문에 커버리지를 만족하지 못한다.

동적 슬라이싱의 단점은 행하면서 또는 수행 중 추출한 트레이스를 바탕으로 분석하기 때문에 정적 슬라이싱에 비해서 더 많은 시간이 필요하고, 커버리지가 작다. 하지만, 난독화는 수행하기 전에는 코드를 알 수 없기 때문에, 동적 슬라이싱이 적합하다.

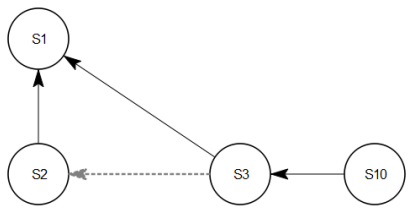


Fig. 3. If x =1, result of Fig 1's dynamic slicing[8]

2.4 슬라이싱의 분석 방향

슬라이싱은 분석 방향에 따라 정방향 슬라이싱과 역방향 슬라이싱으로 구분할 수 있다. 정방향 슬라이싱은 슬라이싱의 기준이 되는 지점과 변수에 영향을 받는 문장들을 추출하는 것이다. 반면에 역방향 슬라이싱은 슬라이싱의 기준이 되는 지점과 변수에 영향을 준 문장들을 찾아내는 것을 의미한다. 두 방법은 분석 방향의 차이가 있지만 프로그램의 일부를 추출하기 위한 방법이라는 공통점이 있다.

2.5 테인트 분석과 슬라이싱의 비교

테인트 분석은 어떤 특정 데이터를 추적하고, 영향을 받는 명령어를 마킹한다. 마킹된 명령어는 추적하려는 데이터가 유효한지를 의미한다. 테인트 분석에서 데이터를 추적하는 것은 슬라이싱에서 특정 데이터에

관련된 명령어를 추출하는 것에 유사한 개념이다.

그러나 프로그램 슬라이싱은 앞서 말한 것처럼 PDG를 구성하고 여기에서 관련된 문장들을 추출해 내는 기술인 반면에 테인트 분석은 사용되는 기술보다는 분석의 목적을 규명하는 용어이다. 따라서, 테인트 출처, 테인트 전파 정책 그리고 테인트 싱크와 같이 3가지를 정의해야 한다. 테인트 출처는 처음 테인트가 시작되는 지점의 데이터를 의미한다. 테인트 전파 정책은 오염된 데이터가 어떻게 퍼져나가는지를 정하는 것을 의미한다. 테인트 싱크는 마킹된 명령어를 어떻게 표현할 것인가를 의미한다.

슬라이싱에서 PDG를 통해 제어 의존성을 반영하는 것처럼 테인트 분석에서도 제어 의존에 관해서 테인트 분석을 한다. 테인트 분석에서는 정적 분석을 통해 CFG(Control Flow Graph)와 pdom (postdominator) 트리를 만든 뒤, 이를 바탕으로 제어 의존에 관한 테인트 분석을 제안한다. 하지만 슬라이싱 기술에서 오래전부터 PDG를 통해 제어 의존성을 반영한 분석을 하고 있다.

테인트 분석은 슬라이싱과 다르게 주로 정방향 분석으로 이루어지고 있다. 특히 바이너리 코드에 대한 역방향 테인트 분석은 매우 드물다. VDT[12]와 ARM-Analyzer[13]에서 역방향 테인트 분석을 하지만, 이것이 슬라이싱과 달리 정확성에 대해 이론적으로 증명이 되지 않았다.

III. 동적 슬라이싱 구현

슬라이싱은 난독화된 프로그램의 의미 없는 코드를 제거할 수 있게 해준다. 난독화된 프로그램은 가상머신의 코드와 자가 변경 코드를 갖고 있기 때문에, 프로그램 원래의 의미를 아는 것은 어렵다. 슬라이싱 기법은 프로그램의 의미와는 상관없는 프로그램에 삽입된 가상머신의 코드와 자가 변경 코드를 제거할 수 있게 해준다.

본 논문에서는 역난독화를 위해서 프로그램 동적 슬라이싱 기법을 이용하였다. 난독화된 프로그램은 프로그램을 실행시키기 전에는 본래 수행되는 코드를 볼 수 없다. 실행하면서 얻어지는 트레이스에는 난독화 되면서 삽입된 코드와 프로그램 본래의 코드가 섞여 있다. 비록 동적 슬라이싱은 정적 슬라이싱에 비해 코드 커버리지가 작고 시간이 많이 걸리지만, 정적 슬라이싱으로는 얻을 수 없는 트레이스, 시스템콜 정보 같은 런타임 정보를 이용하여 분석할 수 있게 해준다.

동적 슬라이싱의 전처리 작업으로 먼저, Fig 4와 같이 동적 트레이스를 추출해야 한다. 추출한 동적 트레이스는 저수준 중간 언어를 이용하여 분석과정을 거친다. 중간언어는 REIL (Reverse Engineering Intermediate Language)[5]를 사용한다. 실제 기계어수준의 명령어는 매우 많기 때문에 이에 대해 모두 처리하는 것은 어렵다. 중간 언어는 바이너리 분석을 보다 쉽게 할 수 있게 해준다. 분석과정에서는 Fig 4.에서 보는 것과 같이 데이터 흐름 분석 (Data flow analysis)을 하게 된다. 그 다음으로 역방향 슬라이싱을 한다. 분석 과정을 마친 후에는 슬라이싱한 코드로 부프로그램을 구성한다.

본 논문에서는 DDG만을 구현하였다. 왜냐하면 프로그램이 가상화 난독화가 되면 프로그램의 제어흐름은 바이트 코드화 되어 원래의 제어 흐름을 알 수 없게 만들어서 노이즈가 많이 생기므로 제어 의존을 제외하고 프로그램 슬라이싱을 구현하였다. 향후 의미

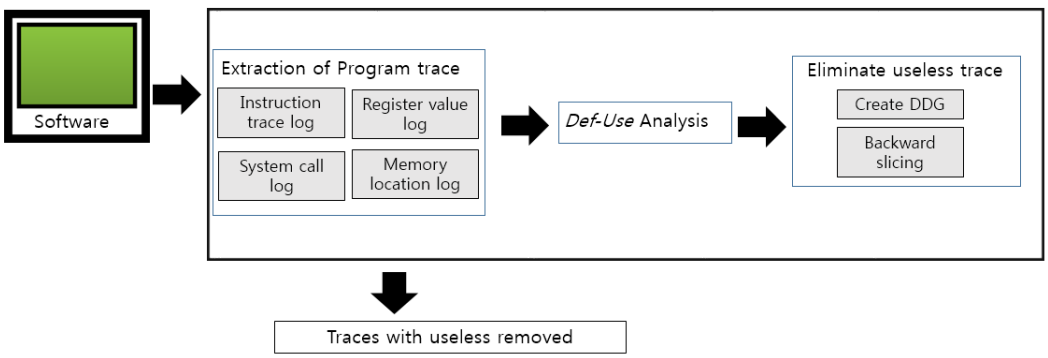


Fig. 4. Flow of Propsed Analysis

있는 제어흐름을 미리 파악하는 연구를 할 예정이다.

3.1 동적 트레이스 추출

트레이스 로깅은 프로그램의 실행시의 정보를 얻기 위한 단계이다. 여기서 얻을 정보는 바이너리 트레이스, 시스템콜, 레지스터가 가지는 값과 메모리 주소를 얻게 된다.

기존의 동적 오염 분석의 경우, 바이너리 트레이스를 얻는 것이 아닌 제어 흐름만을 기록함으로써, 트레이스의 크기를 줄인다. 하지만 본 논문에서는 바이너리 트레이스 전부를 기록 한다. 그렇게 하는 이유는 난독화 도구가 자가 변경(self-modifying) 기법을 이용하기 때문에 실행 전에는 실제 코드를 알 수 없기 때문이다.

로깅할 때는 트레이스 카운트(Trace Count)라는 별도의 킷 값을 만들어서 로깅한다. 여기에 트레이스 카운트를 만들어서 키로 사용하였다. 트레이스 카운트는 프로그램이 실행되고 처음으로 실행되는 명령어는 0으로 로깅한다. 그 다음 실행되는 명령어는 1을 더한 값이 로깅된다. 즉 n번째 실행되는 명령어는 n-1을 트레이스 카운트로 가지게 된다. 트레이스 카운트는 같은 주소의 명령어가 여러 번 실행되었을 때, 이를 구분할 수 있게 해준다. 같은 주소의 같은 명령어 일지라도 실행 시점에 따라 상태가 다르다. 트레이스 카운트는 이를 구분할 수 있게 해준다.

난독화 해제를 목적으로 하는 슬라이싱은 시스템콜로부터 시작하여 원래의 코드로 복원하게 된다. 시스템콜 호출 정보는 시스템콜의 이름, 트레이스 카운트와 PC값을 기록한다. 시스템콜 정보는 리눅스의 경우 시스템콜 테이블 정보를 가지고 할 수 있다. 난독화 해제가 아니더라도 시스템콜 호출 정보는 동적수행을 통해서 알 수 있으므로 매우 중요한 정보이다.

레지스터가 가지는 값과 메모리 위치의 경우, 실행시에만 얻을 수 있는 정보로 정적 분석보다 정확한 분석을 할 수 있게 해준다.

Table 3. Example of binary trace result

```
1. 0x4000 MOV EAX, [EBP+10h]
2. 0x4004 ADD EAX, EDX
3. 0x4008 JMP 0x4000
4. 0x4000 MOV EAX, [EBP+10h]
```

Table 4. Example of system call log

Name	Read
Trace Count	89512
PC	0x08045782

3.2 *def-use* 분석

데이터 분석은 먼저 명령어별 *def-use*분석을 정의한 후 이루어진다. 각 명령어는 *def* 집합과 *use* 집합을 가진다. *def*는 명령어에서 새롭게 정의되는 변수를 말한다. 변수는 레지스터 또는 메모리 주소가 될 수 있다. '`mov eax, 0x5000`'이라는 명령어가 있다고 하면, 이 명령어에서 `eax` 레지스터는 새로운 값이 정의된다. 따라서 `eax`는 명령어에서 *def*가 된다. *use*는 명령어에서 사용되는 변수를 의미한다. '`add eax, ecx`'를 예로 보면, `eax`, `ecx` 레지스터는 이 명령어에서 사용되는 변수이다. 즉, *use*로 분류할 수 있다. 하지만 `eax` 레지스터의 경우에는 연산의 결과가 저장되기 때문에 이 명령어에서는 *def*가 될 수 있다.

3.3 핵심 트레이스 추출

K.Coogan은 시스템콜 중심의 난독화 해제를 제시했었다[2]. 시스템콜의 입력에 영향을 주고 결과에 영향을 받는 명령어는 프로그램에 의미적인 연관이 있다고 보고 그와 관련된 명령어들을 추출해낸다.

시스템콜에 영향을 주고받는 것을 파악하기 위한 데이터 분석은 위의 *def-use* 분석의 결과를 바탕으로 분석한다. 이 분석을 통해 데이터 의존 그래프(DDG)를 만들어 낸다. DDG는 역방향 슬라이싱에 사용된다.

역방향 슬라이싱은 이러한 DDG를 바탕으로 분석을 한다. 역방향 슬라이싱의 기준은 시스템콜의 호출 지점과 인자가 된다. DDG에서 시스템콜 호출 지점 노드를 갖고 오고 이 노드로부터 이어지는 부분 그래프를 추출하면 시스템콜과 관련된 슬라이싱을 추출해 낼 수 있다.

IV. 실험 및 결론

본 논문에 제시된 방법은 실행 트레이스는 Pin[6]을 이용하여 추출하고, 분석 프로그램은 Python으로

작성하였다. 실험 방법은 비교적 간단한 프로그램을 난독화한 뒤, 프로그램의 시스템콜 중심으로 슬라이싱 하여, 의미 없는 트레이스를 제거한 후, 제거하기 전과 후의 트레이스 개수를 비교하였다.

실험 환경은 VMware[9]로 가상머신을 구축하여 실험하였다. 가상머신 사양은 CPU i7-2600, RAM 2GB, Ubuntu 15.10 32bit로 구성하였다.

분석 대상 프로그램의 경우, 일단 난독화를 하기 위해서는 소스코드가 필요하기 때문에 실제 악성코드를 난독화 하여 실험할 수 없기 때문에 랜섬웨어와 같이 파일을 암호화 시키는 프로그램을 작성하여 실험하였다. 이 프로그램은 특정 파일을 읽어 그 파일에 암호화 알고리즘[10]을 적용하여 새로운 파일을 만들어 낸다. 난독화 도구는 Code Virtuailizer[7]를 사용하였다. 거기에 난독화 옵션을 다르게 만든 후, 각각을 난독화 1과 난독화 2로 명명한 뒤, 실험하였다.

Table 5은 각각의 프로그램의 트레이스의 크기를 나타내는 표이다. 난독화 버전에서는 추출한 트레이스의 크기는 난독화 이전의 원본 트레이스 보다 대폭 늘어났다. 난독화 이전에는 트레이스의 크기가 약 24만개지만, 가상머신을 하나 삽입했을 때는 트레이스의 크기가 67%가 늘어나고, 두 개를 삽입한 옵션의 경우에는 114%가 증가하였다.

Table 6에서 슬라이스를 한 후에는, 대폭 (약 80%) 트레이스를 감소시켰다. 가상머신이 하나만 삽입된 난독화 1의 경우, 76094로 약40만개의 트레이스에서 81%를 감소시켰다. 가상머신이 2개가 삽입된 난독화 2의 경우 79%를 감소시킴으로써 감소율에서 난독화 1과 크게 차이를 보이지 않았다. 즉, 가상머신이 2개임에도 비슷한 비율로 감소시키는 것을 볼 수 있다. 난독화된 프로그램을 슬라이싱 후에 남은 트레이스가 난독화 되기 전의 프로그램의 트레이스보다 작은 것을 볼 수 있다. 이는 시스템콜과 관련된 문장들만을 뽑아내기 때문에 함수의 프롤로그와 에필로그가 포함되지 않는다.

추가적으로 난독화와 비슷한 기능을 하는 패킹 도구에서도 효과가 있는지 오픈 소스인 UPX[11]와 자체 제작한[1] 패킹 도구로 실험 하였다. 앞서 난독화 실험과 같은 대상 프로그램으로 실험 결과, 트레이스의 크기는 각각 409,945개와 290,122개로 약 67%, 약18%로 증가하였다. 슬라이스 후의 결과는 두 가지

1) migetpack[15]를 참고하였다.

Table 5. Trace Size of Each Program

	Original Program	Obfuscation 1	Obfuscation 2
Trace Size	244210	409895	523188
Rate of increment		67.85%	114.24%

Table 6. Result of Program Slice

	Obfuscation 1	Obfuscation 2
Size of sliced trace	76094(409895)	105295(523188)
Rate of decrement	81.43%	79.88%

모두 2,044개로 동일한데, 패킹 도구는 원래의 코드를 분할하여 숨기는 등 분석을 어렵게 하는 부분이 없기 때문으로 보인다.

V. 결 론

본 논문에서는 프로그램 동적 슬라이싱 기법을 사용하여 난독화된 코드를 해제했을 때 트레이스 수에 대해 비교 실험한 결과를 소개하였다. 특히 가상화 난독화와 같이 트레이스 수가 너무 많고 복잡하여 난독화 해제에 어려움이 있는 경우를 주요 대상으로 하였다. 제안된 방법은 시스템 호출 기반의 트레이스 추출을 기본으로 하였으며, 패킹을 비롯한 다른 난독화 기법에서도 트레이스 크기를 줄이는 효과가 있을 것으로 보인다. 향후에는 제어 의존 슬라이싱 등을 통해 원본 프로그램의 의미적 복원에 중점을 둔 연구를 진행할 계획이다.

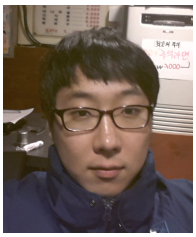
References

[1] K.Coogan, G.Lu and S.Debray, "Deobfuscation of Virtualization-Obfuscated Software," Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 275-284, Oct, 2011.

[2] B.Yadegari and S.Debray, "Bit-level Taint Analysis," 14th IEEE International Working Conference on Source Code

Analysis and Manipulation, pp. 255-264, Sep, 2014
〔3〕 N.Sasirekha, A. Edwin Robert and M.Hemalatha, "Program slicing techniques and its applications," International Journal of Software Engineering and Applications, vol. 2 no. 3, pp. 50-64, July, 2011
〔4〕 X.Zhang, R.Gupta and Y.Zhang, "Precise Dynamic Slicing Algorithms," Proceedings of the 25th International Conference on Software Engineering, pp. 319-329, May, 2003
〔5〕 T.Dullien and S.Porst, "REIL: A platform-independent intermediate representation of dissassembled code for static code analysis," CanSecWest 2009, 2009
〔6〕 Pin, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
〔7〕 CodeVirualizer, http://oreans.com/codevirtualizer.php
〔8〕 H.Agrawal and Joseph R. Horgan, "Dynamic Program Slicing," Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pp. 246-256, June, 1990
〔9〕 VMware, http://www.vmware.com/kr.html
〔10〕 https://github.com/tarequeh/DES
〔11〕 UPX, https://upx.github.io/
〔12〕 VDT, https://github.com/rrbranco/VDT
〔13〕 D.Kim and S.Kim, "Triaging Crashes with Backward Taint Analysis for ARM Architecture," Black Hat EUROPE 2015, Nov, 2015
〔14〕 M.Sharif, A.Lanzi, J.Giffin and W.Lee, "Automatic Reverse Engineering of Malware Emulators," 30th IEEE Symposium on Security and Privacy, pp.94-109, May, 2009
〔15〕 midgetpack, https://github.com/arisada/midgetpack

〈저자소개〉



목 성 균 (Seong-Kyun Mok) 학생회원
2014년 2월: 충남대학교 컴퓨터공학과 졸업
2014년 3월~현재: 충남대학교 컴퓨터공학과 석박통합 과정
〈관심분야〉 프로그래밍언어, 프로그램 분석

전 현 구 (Hyeon-gu Jeon) 학생회원
2014년 2월: 충남대학교 컴퓨터공학과 졸업
2014년 3월~2016년 2월: 충남대학교 컴퓨터 석사과정
〈관심분야〉 프로그래밍언어, 역공학

조 은 선 (Gil-dong Hong) 정회원
1991년 2월: 서울대학교 계산통계학과 졸업
1993년 2월: 서울대학교 전산과학과 석사
1998년 3월: 서울대학교 전산과학과 박사
1999년~20100년: 한국과학기술원 연구원
2000년~2001년: 아주대학교 정보통신전문대학원 조교수 대우
2002년~2006년: 충북대학교 조교수
2006년~ 현재: 충남대학교 컴퓨터공학과 교수
〈관심분야〉 프로그래밍언어, 프로그램 분석, 이벤트 처리