석사 학위논문
Master's Thesis

가상화 기법으로 난독화된 실행 파일의 동적 분석 방법

A Dynamic Analysis of
Virtualization-Obfuscated Binary Executables

전 준 수 (全 浚 秀 Jeon, Joonsoo)
전산학과
Department of Computer Science

KAIST

2012

가상화 기법으로 난독화된 실행 파일의 동적 분석 방법

A Dynamic Analysis of
Virtualization-Obfuscated Binary Executables

A Dynamic Analysis of Virtualization-Obfuscated Binary Executables

Advisor : Professor Han, Taisook

by

Jeon, Joonsoo

Department of Computer Science

KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science . The study was conducted in accordance with Code of Research Ethics[1].

2011. 12. 19.

Approved by

Professor Han, Taisook

[Advisor]

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

가상화 기법으로 난독화된 실행 파일의 동적 분석 방법

전 준 수

위 논문은 한국과학기술원 석사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2011년 12월 19일

심사위원장 한 태 숙 (인)

심사위원 김 명 호 (인)

심사위원 류 석 영 (인)

MCS
20103567

전 준 수. Jeon, Joonsoo. A Dynamic Analysis of Virtualization-Obfuscated Binary Executables. 가상화 기법으로 난독화된 실행 파일의 동적 분석 방법. Department of Computer Science . 2012. 53p. Advisor Prof. Han, Taisook.

ABSTRACT

When a new malware is found, security researchers must first understand the behavior of the malware to prepare countermeasure. This is a difficult task because malware writers distort the contents of malwares using code obfuscation techniques. It is getting difficult to analyze the behavior of a malware because the obfuscation techniques are getting sophisticated and diverse. One of those advanced obfuscation techniques is virtualization-obfuscation that utilizes virtual machines for obfuscation. Since virtualization totally changes the structure and instruction set of an original program, the binary executable obfuscated with virtualization is extremely difficult to analyze. Furthermore, many variations of virtualization obfuscation are expected to be applied soon. So it is infeasible to develop reverse engineering techniques applicable to all obfuscation techniques. Nonetheless, since an obfuscated program inevitably has the same semantics as the original program, monitoring dynamic behaviors of a program will help the analyzers to figure out the inside. In other word, to disclose the semantics of an obfuscated program, it will be useful to analyze its run trace that is the record of executed instructions and changes of machine states. In this thesis, we present a dynamic analysis tool, Trudio, which provides functional modules to analyze the structure and semantics of a program based on its run traces. Moreover, our tool transforms the program into analyzable shape based on the information acquired from the run traces. We describe our observations on obfuscation techniques and the algorithms that are implemented in Trudio and show that the suggested methods can help analyzing virtualization-obfuscated binary executables and understanding the behavior of programs. We expect malware analyzers to save time in understanding an obfuscated malware with our tool.

목 차

표 목 차

그 림 목 차

제 1 장 서론

1.1 연구의 배경 및 필요성

난독화(難讀化; Obfuscation)란 실제로 전하거나 담고자 하는 내용을 제삼자가 파악하기 어렵게 하기 위해 그 형태를 바꾸는 것을 의미한다. 그 중에서도 특히 코드 난독화, 혹은 프로그램 난독화란 컴퓨터에서 실행하기 위해서 개발된 프로그램을, 동일한 기능을 하면서 분석자가 그 의미를 파악하기 어렵게 만드는 것을 의미한다. 하지만 프로그램의 의미 파악을 원천적으로 막는 것은 불가능하므로 분석 시간을 증가시키는 것이 코드 난독화의 목적이 된다.

코드 난독화는 소스 코드에 적용되는 경우와 기계어로 된 실행 파일에 적용되는 경우로 나누어 볼 수 있다. 소스 코드를 난독화하는 가장 대표적인 경우는 자바스크립트 언어로 작성된 프로그램을 난독화하는 경우이다. 자바스크립트 언어로 작성된 프로그램들은 소스코드의 형태로 배포되는 자바스크립트의 특성상 난독화를 적용하지 않으면 프로그램의 알고리즘 등을 쉽게 파악할 수 있기 때문에 저작권을 보호하기 위하여 난독화를 적용하는 경우가 흔히 있으며, 상용으로 공개되어 있는 난독화 도구도 있다[1].

한편, 실행 파일에 대해서는 실행 파일의 역공학(reverse engineering)을 어렵게 하여 원본 프로그램의 저작권을 보호하기 위한 목적으로 난독화가 적용된다. 프로그램 역공학이란 프로그램의 내용을 분석하여 동작 원리나 알고리즘 등을 파악해 내고, 궁극적으로는 프로그램의 원본 소스 코드를 유추해내는 것을 의미한다. 특히 자바 프로그램의 경우 역공학이 용이한 자바 바이트코드로 컴파일되기 때문에 저작권 보호를 위해 프로그램을 배포하기 전에 난독화를 적용한 사례들이 많고 난독화 도구도 다수 공개되어 있다[2].

하지만 코드 난독화는 저작권 보호라는 목적 외에 악성 코드에 적용되어 분석자가 악성 코드의 의미를 파악하는 데 소요되는 시간을 증가시키려는 목적으로 빈번하게 사용된다. 악성 코드란 악의적인 목적으로 사용자의 컴퓨터에 해를 가하는 프로그램을 의미하는 말로, 컴퓨터 바이러스, 웜, 트로이 목마 등이 이에 속한다. 2011년 상반기에만 전년도에 비해 15% 이상 증가한 120만개 이상의 신규 악성 코드가 발견된 것으로 보고[3]되는 등 갈수록 심각한 보안 문제로 부각되고 있다.

초기의 보안 프로그램들은 시그니쳐(signature) 기반으로 악성 코드를 탐지하였다. 시그니쳐란 하나의 프로그램에서 나타나는 바이트 시퀀스나 인스트럭션 시퀀스를 의미하는 것으로, 하나의 악성 코드나 해당 악성 코드에 감염된 프로그램에서 공통적으로 나타나는 시그니쳐를 탐색하여 악성 코드를 검출하였다.

하나의 프로그램에 난독화를 적용하면 적용되는 난독화 방법에 따라 프로그램의 시그니쳐가 변경될 수 있기 때문에 하나의 악성 코드로부터 변종을 만들어 내기 위해 난독화를 적용하였다. 하지만 보안 프로그램이 발전하면서 악성 코드의 의미를 파악하여 검출하는 기능 등이 추가되면서[4] 악성 코드 작성자들은 악성 코드가 스스로 시그니쳐를 바꾸어나갈 수 있는 다형성(polymorphic) 및 변성(metamorphic) 악성 코드를 개발함과 더불어, 악성 코드 분석자들이 그 내용을 파악하기 어렵도록 보다 복잡한 난독화 기법을 적용하게 되었다.

난독화 기법이 발전됨에 따라 난독화 해제 기법도 발전했고, 반대로 난독화 해제기법이 발전할수록 악성 코드 개발자들도 보다 복잡하고 해제하기 어려운 난독화 기법을 적용해 가고 있다[5]. 따라서, 프로그램의 난독화를 해제하는 것이 악성 코드 분석의 중요한 이슈가 되고, 악성 코드가 컴퓨터 시스템 보안의 매우 중요한 문제인 만큼 프로그램의 난독화를 해제하는 것이 보안 문제 해결의 중요한 과제가 된다.

코드 난독화의 아이디어는 매우 다양하다[6]. 가장 기본적인 아이디어는 불필요한 코드를 많이 삽입하는 것이다. 즉, 프로그램의 이곳 저곳에 실제로 실행되지 않거나, 아무 일도 하지 않거나, 혹은 실행되어도 결과적으로 프로그램의 실행 결과에 아무런 영향을 미치지 않는 코드를 삽입하는 것이다. 이러한 코드들을 죽은 코드(dead code)라고 부르기도 한다. 이러한 난독화 기법이 적용된 프로그램은 수행 속도가 원래의 프로그램에 비해 느려진다는 단점이 있고, 분석자가 난독화된 프로그램을 섬세하게 분석하면 결국에는 원본 프로그램의 내용을 파악할 수 있지만, 프로그램의 어떤 부분이 본래 프로그램의 내용을 담고 있고 어떤 부분이 불필요한 코드인지 파악하는 과정을 거쳐야 하기 때문에 프로그램의 분석을 어렵게 한다는 본래의 목적은 달성할 수 있다.

또 다른 난독화 기법은 프로그램 코드를 여기저기로 찢어서 흩뿌려 놓는 방법이 있다. 프로그램 내에서 분기나 반복 없이 실행되는 프로그램의 부분을 베이직 블록(basic block)이라고 하는데, 실질적으로 하나의 베이직 블록에 속하는 코드를 프로그램의 이곳 저곳으로 찢어 놓고 무조건 분기(unconditional jump)를 사용해 순차적으로 실행되도록 하는 기법이다.

앞서 소개한 방법들이 프로그램의 실행 경로를 분석하기 어렵게 만드는 방법이었다면, 그 외에 프로그램에서 사용할 데이터를 특정한 알고리즘으로 인코딩하였다가 사용하기 직전에 디코딩하여 사용하도록 난독화하여 프로그램에서 사용하는 데이터를 분석하기 어렵게 만드는 방법도 있다.

이러한 전통적인 난독화 기법 외에, 비교적 최근 소개된 기법으로 가상화(Virtualization)을 이용한 난독화 기법이 있다. 가상화란 본래 실행되어야 할 프로그램이 요구하는 환경과 실제 구동될 시스템 환경이 다른 경우, 실행될 대상과 실제 환경 사이에 가상화 계층(Virtualization layer)를 삽입하여 그 두 환경의 차이를 극복하는 것을 의미한다.

가상화 기술을 이용하여 프로그램을 구동하는 대표적인 기술로 자바를 들 수 있다. C나 C++와 같은 언어로 작성된 프로그램은 보통 곧바로 실제 프로그램이 구동될 시스템 환경에서 바로 실행될 수 있는 기계어로 컴파일 된다. 하지만 자바 프로그램은 자바 바이트코드로 컴파일되는데, 일반적인 PC나 스마트폰 등에 탑재되는 프로세서들은 자바 바이트코드를 실행할 수 없다. 때문에 자바 바이트코드로 컴파일된 프로그램은 실행될 PC나 스마트폰 등의 환경 위에서 구동되는 자바 가상 기계(Java virtual machine; Java VM)을 통해 실행된다. 자바 VM은 자바 바이트코드로 컴파일된 프로그램을 읽어 들인 뒤 그 의미를 해석하여 현재의 구동 환경의 프로세서가 실행할 수 있는 형태로 바꾸어 실행한다. 바이트코드라는 말은 원본 소스코드보다는 저수준(low-level)이지만 VM을 통해서만 실행될 수 있다는 의미로 기계어와 구분하기 위하여 사용된다.

가상화를 이용한 난독화 기법이란 난독화된 프로그램에 구조가 알려지지 않은 소형의 가상 기계와 이 가상 기계로 실행할 수 있도록 인코딩된 원본 프로그램을 담아서 원본 프로그램의 내용을 감추는 난독화 기법이다. 자바에 빗대어 설명하면, 자바 VM이 자바 바이트코드로 컴파일된 프로그램을 수행할 수 있다면, 자바 VM과 자바 바이트코드로 컴파일된 프로그램을 하나의 실행 파일에 넣어서 본래 의도한 프로그램을 수행하도록 할 수 있을 것이다. 이와 비슷하게, 원본 프로그램을 바이트코드로 바꾼 뒤, 하나의 실행 파일에 바이트코드로 인코딩된 원본 프로그램과 해당 바이트코드를 실행할 수 있는 가상 기계를 탑재한다면 본래 의도한 프로그램을 수행하도록 할 수 있을 것이다. 다른 점이라면, 자바 VM과 자바 바이트코드는 그 구조가 공개되어 있지만, 난독화를 위해 생성되는 VM과 바이트코드의 구조는 공개되어 있지 않아서 분석하기가 어렵다는 점이다.

이러한 난독화 기법은 두 개 이상의 프로그램 카운터를 사용하거나 두 개 이상의 프로그램 스토리지를 사용하여 더욱 복잡한 형태로 발전시킬 수도 있어 앞으로 분석이 더욱 어려워질 가능성이 높다.

난독화를 위한 VM은 흔히 다음에 실행할 바이트코드가 저장된 메모리 위치를 가리키는 가상 프로그램 카운터(Virtual Program Counter; VPC), 각 바이트코드에 대응되어 실제로 실행해야 할 명령을 저장한 바이트코드 핸들러, 그리고 바이트코드와 핸들러를 대응시키는 '바이트코드 대응 테이블(Bytecode mapping table)', 그리고 VPC를 이용해 다음에 실행할 바이트코드를 읽어와 대응 테이블에서 다음에 실행할 핸들러를 찾아 핸들러를 호출해 주는 가상 기계 중심 블록의 네가지 구성 요소로 되어 있다.

그림 1.1은 가상화를 이용한 난독화 기법이 적용된 프로그램의 동작 방법을 도식적으로 보이고 있다.

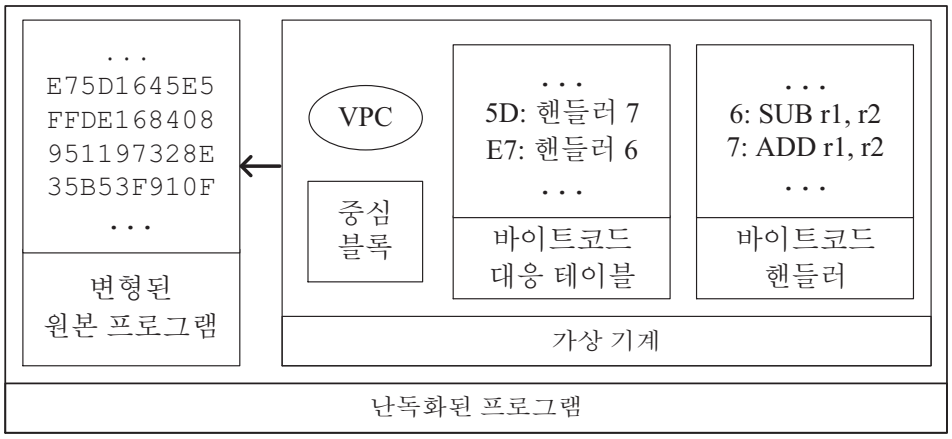


그림 1.1: 가상화를 이용한 난독화의 원리

상용 난독화 도구인 VMProtect[7]나 Code Virtualizer[8] 등을 이용하면 임의의 실행 파일에 가상화를 이용한 난독화를 자동으로 적용할 수 있다.

하지만 아무리 고도화된 코드 난독화 기술이 적용되더라도 난독화된 프로그램은 원본 프로그램과 동일한 실행 결과를 내야 한다. 다시 말해, 고도로 난독화된 프로그램도 결국 원본 프로그램과 동일한 동작(behavior) 혹은 의미(semantic)을 가져야 한다는 것이다. 그리고 어떤 프로그램의 의미는 그 프로그램이 수행될 때 실행되는 CPU 인스트럭션과 시스템의 상태(status)의 변화를 분석해서 알 수 있다.

이 점에 착안하여 실제로 프로그램이 수행되는 과정에서 실행되는 명령과 상태 변화를 기록한 실행 트레이스(run trace)를 바탕으로 프로그램의 의미를 분석하는 도구인 Trudio를 개발하였다. 본 논문은 도구에서 구현된 분석 알고리즘과 도구의 사용 방법 및 실제 적용 사례와 도구의 유용성을 증명하는 실험 결과를 포함한다.

1.2 관련 연구

기존에는 프로그램을 분석하기 위해 주로 IDA Pro[9]나 OllyDbg[10]와 같은 디버깅 도구들을 이용해 왔다. 이러한 디버깅 도구는 기본적으로 프로그램을 인스트럭션 단위로 실행하면서 프로그램이 동작하는 과정을 추적할 수 있게 돕는다.

하지만 디버깅 도구만을 이용해서는 난독화된 프로그램을 제대로 분석하기가 어려우므로 이를 해결하기 위한 다양한 난독화 해제 기법이 개발되었다. 이들 해제 기법들은 개별 프로그램의 형태나 디버깅 도구의 플러그인 형태로 개발되어 사용되어 왔다.

R. Rolles는 요약 해석(Abstract Interpretation) 기법을 이용해 제어 흐름 난독화 기법을 해제하는 방법을 제안하였다[11]. 불투명 술어(opaque predicate)란 변수의 값이 무엇이든 항상 같은 불리언 결과를 내는 식을 나타내는데, 이러한 불투명 술어를 이용하여 무조건 점프문을 분기문의 형태로 바꾸어 난독화하는 경우가 있다. Rolles는 이러한 분기문들에 사용되는 조건 비트의 값을 실제 값 대신 참, 거짓, 알 수 없음 세 가지의 상태로 요약하여 무의미한 분기문을 찾아서 제거하는 방법을 제안하였다.

B. Spasojević은 컴파일러의 최적화 기법들을 난독화 해제를 위해 사용하는 방법을 정리하였다[12, 13]. 그리고 이러한 기법들을 IDA Pro에서 사용할 수 있도록 개발한 플러그인 형태의 optimice를 공개하였다[14]. Y. Guillot과 A. Gazet도 이와 비슷한 기법들을 개발하고 [15], 이를 구현한 METASM[16]을 공개하였다. 그 외에 바이너리 인스트루멘테이션(Binary instrumentation)을 이용해 자바스크립트나 X86 코드의 난독화를 해제하려는 시도도 있었다[17, 18].

가상화를 이용한 난독화에 특화된 해제 기법들도 소개되었다. R. Rolles는 VMProtect[7]에서 사용하는 가상 기계의 구조를 파악하여 이를 역으로 이용해 난독화된 프로그램에서 바이트코드를 추출한 다음 이를 바탕으로 원본 프로그램을 생성하였다[19]. 여기서 바이트코드는 우선 중간 언어(intermediate language) 형태의 코드로 변환되는데, 여기에 컴파일러에서 사용되는 최적화 기법들을 적용하여 크기를 줄였다. 최적화 기법을 통해 중간 코드의 크기가 80% 가까이 감소하고 불필요한 내용이 없어졌음을 소개하고 있다. 하지만 이 논문에서 소개된 기법은 VMProtect가 업데이트되면서 사용될 수 없게 된 것으로 알려져 있다.

[20]은 비교적 최근의 연구로, 프로그램을 실행하여 얻은 실행 트레이스를 이용해 동적 분석을 수행하는 기법을 다루고 있으므로 본 논문과 유사한 접근 방법을 취하고 있다.

1.3 연구 방향

본 논문은 기하급수적으로 증가하는 난독화 기법에 일일이 대응하는 것이 불가능하다고 보고, 프로그램의 의미를 파악하는 데에 초점을 맞추어 프로그램을 실제로 수행하면서 얻어진 실행 트레이스를 분석하는 동적 분석 방법론에 대하여 진행한 연구에 대해 기술한 것이다. 이는 기존의 연구들이 하나의 난독화 기법을 설정하여 이를 정적인 방법으로 해제하는 데에 집중하였던 것과는 달리 동적인 방법으로 프로그램의 의미를 파악하는 데 집중하고 있다는 점이 다르다.

기존의 연구들이 채용한 정적 분석 방식은 난독화된 대상 프로그램에 적용된 난독화 기법에 대한 강력한 가정을 갖고 개발해야 하고, 가정에 맞지 않는 변종이나 새로운 난독화 기법이 등장하면 기존의 연구를 사용할 수 없게 된다. 또한 가상화를 이용한 난독화 기법처럼 난독화 기법이 복잡해지면 이를 해제하기 위한 분석 방식은 더욱더 복잡해 진다. 하지만 프로그램 전체를 분석하고 커버할 수 있다는 장점이 있다.

반면 동적 분석 방식은 프로그램 실행 과정에서 수행되지 않은 코드나 실행 경로는 분석할 수 없어서 프로그램 전체를 커버할 수 없다는 한계를 갖고 있는 한편 적용된 난독화 기법이 매우 복잡해지거나 기존에 알려지지 않았던 새로운 기법이 적용되더라도 프로그램이 실제로 실행된 기록을 바탕으로 분석을 수행하므로 실행된 내용에 대하여 분석할 수 있다는 장점이 있다.

본 논문은 프로그램이 수행되면서 실행된 인스트럭션의 목록과 레지스터와 메모리의 값 변화를 나타내는 시스템의 상태 변화를 저장하여 한 벌의 실행 트레이스라고 한다. 실행 트레이스의 구조는 2.2에서 자세하게 소개된다. 이러한 실행 트레이스를 바탕으로 다음의 세 가지 목적을 달성하기 위한 알고리즘을 개발하고 구현하였다.

첫째, 프로그램의 구조를 파악한다. 실행 트레이스로부터 프로그램의 제어 흐름 그래프를 얻어내거나, 각 인스트럭션이 접근하는 메모리 영역 등을 추적하거나, 적절한 형태의 통계를 보이는 등의 보조 기능을 통해 분석자가 프로그램의 구조를 파악할 수 있게 돕는다. 특히 난독화에 사용되는 가상 기계의 특징을 이용해 가상 기계의 구조를 파악할 수 있게 돕는다.

둘째, 프로그램의 의미를 파악한다. 프로그램의 실행 결과가 어떤 과정을 거쳐서 계산되었는지를 추적하고 시각화하여 보여서 분석자가 프로그램의 계산 과정과 알고리즘을 쉽게 파악할 수 있게 돕는다.

셋째, 프로그램의 실행에 불필요한 코드를 제거하여 보다 분석하기에 용이한 형태로 최적화한다. 동적 분석을 통하여 대상 프로그램의 명령 중 불필요한 것들이나 인코딩된 데이터 영역을 찾아내어 이를 제거할 수 있다. 이러한 방법으로 난독화 기법들을 해제하면 프로그램을 보다 용이하게 분석할 수 있다.

이러한 동적 분석 각각의 목적과 이를 위한 분석 알고리즘에 대하여 2장에서 상술한다.

제 2 장 알고리즘

2.1 표기법

$f(a:A) \to B$ 는 이름이 a인 타입 A의 인자를 받아 B 타입의 값을 반환하는 함수 f를 나타낸다.

그 외에 본 논문에 나타나는 알고리즘들은 일반적인 의사 코드(pseudocode)의 형태를 따라서 if, for, for all, repeat until 문 등을 사용하고 $a \leftarrow b$ 는 변수 a에 값 b를 넣음을 의미한다. 단, 두 개의 집합 A와 B에 대해 $A \Leftarrow B$와 같은 표기법이 사용될 때, 이는 B의 모든 원소를 A에 추가함을 의미한다. 즉, $A \Leftarrow B$는 $A \leftarrow A \cup B$와 같다.

본 논문에서는 몇 가지 특별한 자료 형을 사용한다. 쌍(pair)은 원소가 2개인 튜플을 의미하고 목록(list)은 같은 타입의 원소를 여러개 갖는 순서가 있는 집합이다. 목록은 모두 0부터 색인되는 것으로 가정하고 목록 뒤에 $[i]$가 붙으면 목록의 i번째 원소를 나타낸다. 예를 들어 어떤 목록 l에 대해 $l[0]$은 l의 첫번째 원소를 가리키고 $l[1]$은 l의 두번째 원소를 가리킨다.

또한 논문에서 딕셔너리라는 자료 구조를 사용한다. 딕셔너리란 키(key)로부터 값(value)로 대응시키는 자료 구조로, 딕셔너리의 키는 중복될 수 없으며 하나의 키에 하나의 값만 대응될 수 있다. 딕셔너리의 타입은 $\{(A \to B)\}$로 나타나며 이러한 딕셔너리는 A 타입의 키로부터 B 타입의 값을 대응해줌을 나타낸다. 딕셔너리는 사실상 키와 값의 쌍의 집합으로 나타낼 수 있다. 즉, $\{(A \to B)\}$ 타입의 딕셔너리는 $\{(A, B)\}$ 형태의 집합이다.

딕셔너리에 관련된 몇 가지 표기법과 함수를 다음과 같이 정의한다.

- D^k는 딕셔너리 D에서 키 k에 대응하는 값을 나타낸다. D^k의 값을 사용할 수도 있고 값을 여기에 대입할 수도 있다. 즉, $D^k \leftarrow v \equiv D \leftarrow D - \{(a,b) \in D | a = k\} \cup \{(k,v)\}$ 이다.
- $keys(\{(A \to B)\}) \to \{A\}$ 함수는 $\{(A \to B)\}$ 타입의 딕셔너리를 받아 $\{A\}$ 타입인 딕셔너리의 모든 키의 집합을 반환한다. 즉, $keys(D) = \{k, (k,v) \in D\}$ 이다.
- $values(\{(A \to B)\}) \to \{A\}$ 함수는 $\{(A \to B)\}$ 타입의 딕셔너리를 받아 $\{B\}$ 타입인 딕셔너리의 모든 값의 집합을 반환한다. 즉, $values(D) = \{v, (k,v) \in D\}$ 이다.

2.2 실행 트레이스의 구조

이 절에서는 본 논문에서 사용할 실행 트레이스의 구조를 설명하고 용어 및 표기법을 정의한다. 실행 트레이스는 대상 프로그램의 분석을 위해 필요한 프로그램의 실행에 관련된 정보의 기록으로 실행된 CPU 인스트럭션과 상태의 변화, 즉 레지스터와 메모리 값의 변화 등이 포함된다.

실행 트레이스에는 기본적으로 실행된 인스트럭션의 목록이 포함되어야 한다. 실행된 인스트럭션 하나를 하나의 트레이스 인스턴스라고 부르고 전체 트레이스 인스턴스의 목록을 전체 트레이스 인스턴스 목록이라고 부른다. 이들이 인스트럭션이 아닌 인스턴스라고 불리우는 것은 트레이스 인스트럭션과 구분하기 위해서이다. 루프나 재귀 함수가 호출되었을 경우 하나의 인스트럭션이 여러번 실행되는 경우가 있을 수 있는데, 이 때 실행되는 하나의 인스트럭션이 트레이스 인스트럭션이고, 해당 인스트럭션이 여러번 실행되면서 전체 트레이스 인스턴스 목록에 기록되는 각각의 기록을 트레이스 인스턴스라고 한다. 즉, 하나의 트레이스 인스트럭션은 한 개 이상의 트레이스 인스턴스를 생성하게 되며, 하나의 트레이스 인스턴스는 단 하나의 트레이스 인스트럭션으로부터 생성된다. 앞으로는 트레이스 인스트럭션과 트레이스 인스턴스를 줄여서 각각 인스트럭션과 인스턴스로 부르기로 한다.

각각의 인스트럭션은 내부(inner) 인스트럭션이거나 외부(external) 인스트럭션이다. 내부 인스트럭션은 분석 대상 프로그램으로부터 온 인스트럭션이고, 외부 인스트럭션은 대상 프로그램 외의 모듈에 속한 인스트럭션이다. 각 내부 인스트럭션은 하나의 CPU 인스트럭션에 대한 정보를 담고 있고, 각 외부 인스트럭션은 하나의 외부 루틴에 대한 정보를 담고 있다. 따라서 외부 인스트럭션 하나는 여러 개의 CPU 인스트럭션을 포함할 수 있다.

내부 인스트럭션에는 인스트럭션이 저장되어 있었던 메모리의 주소 정보와 해당되는 CPU 인스트럭션의 바이트 시퀀스를 포함한다. 외부 인스트럭션은 해당 인스트럭션의 외부 루틴 이름과 루틴이 로드된 원본 파일 경로를 포함한다.

하나의 인스턴스는 자신이 생성된 인스트럭션을 저장하고, 이에 더해 전체 트레이스 인스턴스 리스트에서 자신의 위치를 인덱스 번호로 저장한다.

또 각각의 인스턴스에는 인스턴스 액세스의 목록이 저장된다. 인스턴스 액세스는 해당 인스턴스가 읽거나 쓴 레지스터나 메모리에 대한 정보를 저장한다. 시스템의 레지스터와 메모리를 통틀어 시스템의 상태라고 부른다. 하나의 인스턴스 액세스는 읽기, 쓰기 전, 혹은 쓰기 후로 나뉘어지고, 이를 해당 인스턴스 액세스의 종류(type)라고 한다. 하나의 인스턴스 액세스의 종류가 읽기이면 이는 해당 인스턴스 액세스가 속한 인스턴스가 인스턴스 액세스가 가리키고 있는 위치를 읽었음을 나타내고, 읽어들인 값이 함께 저장된다. 인스턴스 액세스가 쓰기이면 해당 인스턴스가 인스턴스 액세스가 가리키고 있는 레지스터나 메모리의 값을 변경했음을 나타낸다. 쓰기 전은 인스턴스가 값을 변경하기 직전의 값을 나타내고, 쓰기 후는 인스턴스가 변경한 값을 저장하고 있다. 따라서 쓰기 전 액세스와 쓰기 후 액세스는 각각 대응된다.

내부 인스트럭션에는 인스트럭션 액세스 목록이 저장된다. 하나의 내부 인스트럭션은 하나의 CPU 인스트럭션을 나타내고, 하나의 CPU 인스트럭션이 시스템의 상태를 읽거나 쓰는 패턴은 일정하다. 따라서 하나의 내부 인스트럭션이 여러 개의 인스턴스를 생성했더라도 모든 인스턴스는 같은 길이, 같은 수의 읽기 및 쓰기 인스턴스 액세스를 갖는, 같은 형태의 인스턴스 액세스 목록을 갖게 된다. 달리 말하면, 인스트럭션 액세스는 해당 인스트럭션에서 생성된 여러 개의 인스턴스에 공통적으로 속한 인스턴스 액세스들의 추상화라고 할 수 있다. 반면 외부 인스트럭션은 여러 개의 CPU 인스트럭션을 가리킬 수 있으므로 인스턴스 액세스들의 형태가 다를 수 있어서 인스트럭션 액세스를 갖지 않는다.

그림 2.1은 실행 트레이스의 구조를 도식적으로 보인다.

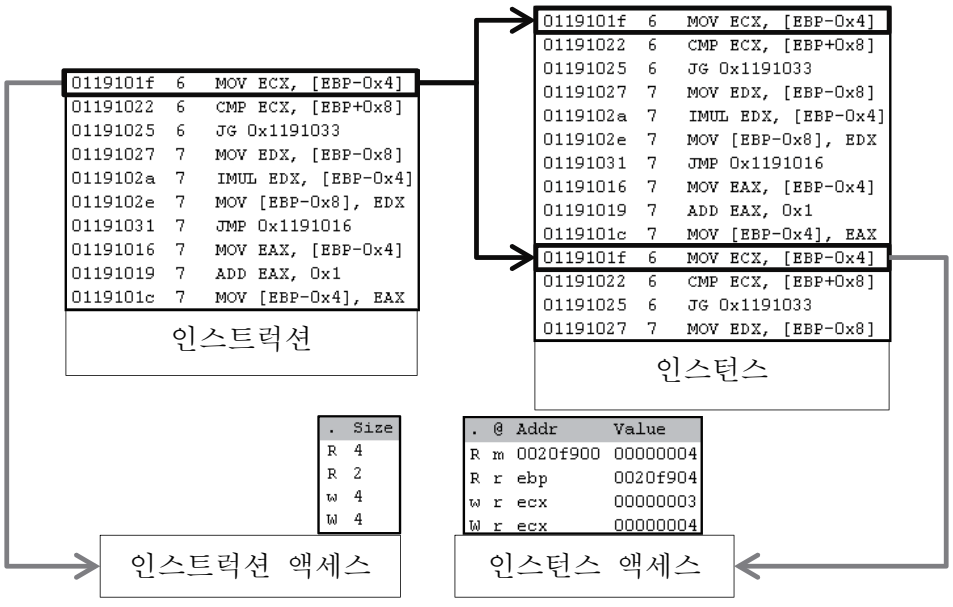


그림 2.1: 실행 트레이스의 구조

알고리즘을 기술하기 위해 실행 트레이스의 정보를 사용하기 위한 함수를 다음과 같이 정의하였다.

- $trace(k : Integer) \rightarrow Instance$ 함수는 음이 아닌 정수 k를 받아 전체 트레이스 인스턴스 목록에서 k번째 인스턴스를 반환한다. 이 때, 전체 트레이스 인스턴스는 0부터 색인되어 있는 것으로 가정한다.
- $index(i : Instance) \rightarrow Integer$ 함수는 인스턴스 i를 받아 i가 전체 트레이스 인스턴스 목록에서 몇 번째 인스턴스인지를 나타내는 음이 아닌 정수를 반환한다.
- $instruction(i : Instance) \rightarrow Instruction$ 함수는 인스턴스 i를 받아 i가 속한 인스트럭션을 반환한다.
- $place(I : Instruction) \rightarrow inner, external$ 함수는 인스트럭션 I를 받아 I가 내부 인스트럭션인 경우 상수 $inner$를 반환하고 외부 인스트럭션인 경우 상수 $external$을 반환한다.
- $instances(I : Instruction) \rightarrow [Instance]$ 함수는 인스트럭션 I를 받아 I로부터 생성된 인스턴스를 전체 트레이스 인스턴스 목록에서의 색인에 따라 오름차순으로 정렬한 목록을 반환한다.
- $opcode(I : Instruction) \rightarrow OPCODE$ 함수는 내부 인스트럭션 I를 받아 연산 코드(operation code; opcode)를 반환한다.
- $accesses(i : Instance) \rightarrow [InstanceAccess]$ 함수는 인스턴스 i를 받아 i에 해당하는 인스턴스 액세스 목록을 반환한다.
- $accesses(I : Instruction) \rightarrow [InstructionAccess]$ 함수는 내부 인스트럭션 I를 받아 I에 해당하는 인스트럭션 액세스 목록을 반환한다.
- $instance(a : InstanceAccess) \rightarrow Instance$ 함수는 인스턴스 액세스 a를 받아 a가 속한 인스턴스를 반환한다.
- $instruction(A : InstructionAccess) \rightarrow Instruction$ 함수는 인스트럭션 액세스 A를 받아 A가 속한 인스트럭션을 반환한다.
- $instructionaccess(a : InstanceAccess) \rightarrow InstructionAccess$ 함수는 인스턴스 액세스 a를 받아 이에 대응되는 인스트럭션 액세스를 반환한다.
- $instanceaccesses(A : InstructionAccess) \rightarrow \{InstanceAccess\}$ 함수는 인스트럭션 액세스 A를 받아 이에 대응되는 인스턴스 액세스를 각 액세스가 속한 인스턴스의 전체 트레이스 인스턴스 목록에서의 색인에 따라 오름차순으로 정렬한 목록을 반환한다.

인스턴스 액세스는 항상 연속적인 영역을 읽거나 쓰는 것으로 간주한다. 즉, 레지스터나 메모리로부터 연속된 배열을 읽거나 쓰는 경우에만 하나의 인스턴스 액세스로 기록되며, 그렇지 않은 경우 여러 개의 인스턴스 액세스로 나뉘어서 저장되어야 한다. 따라서 인스턴스 액세스는 읽거나 쓰는 영역의 첫 번째 주소와 길이를 저장하여 대상 주소를 가리키게 된다. 이러한 가정 하에 인스턴스 액세스의 정보를 읽어오기 위한 함수를 다음과 같이 정의한다.

- $type(a : InstanceAccess) \rightarrow read, writing, written$ 함수는 인스턴스 액세스 a를 받아 그 종류를 반환한다. 종류는 읽기인 경우 read, 쓰기 전은 writing, 쓰기 후는 written으로 반환된다.
- $address(a : InstanceAccess) \rightarrow Address$ 함수는 인스턴스 액세스 a를 받아 읽거나 쓰는 첫번째 주소를 반환한다.

- $size(a : InstanceAccess) \rightarrow Integer$ 함수는 인스턴스 액세스 a를 받아 읽거나 쓰는 길이를 반환한다.

- $addresses(a : InstanceAccess) \rightarrow \{Address\}$ 함수는 인스턴스 액세스 a를 받아 a가 읽거나 쓰는 주소들의 집합을 반환한다. 이 때, 인스턴스 액세스는 항상 연속된 배열을 읽거나 쓰므로 다음과 같이 나타낼 수 있다.

  $addresses(a) = \{address(a) + k | 0 \leq k < size(a)\}$.

- $value(a : InstanceAccess) \rightarrow Integer$ 함수는 인스턴스 액세스 a를 받아 a가 저장하고 있는 값을 정수의 형태로 반환한다. 이 함수는 대상 프로그램이 구동되는 컴퓨터의 엔디안(endian) 설정에 따라 달라질 수 있을 것이다.

- $value(a : InstanceAccess, b : Address) \rightarrow Byte$ 함수는 인스턴스 액세스 a가 주소 b에 읽거나 쓰는 값을 바이트의 형태로 반환한다. $b \in addresses(a)$의 조건을 만족시켜야 한다.

- $written(a : InstanceAccess) \rightarrow InstanceAccess$ 함수는 쓰기 후 인스턴스 액세스 a를 받아 이에 대응하는 쓰기 전 인스턴스 액세스를 반환한다.

- $writing(a : InstanceAccess) \rightarrow InstanceAccess$ 함수는 쓰기 전 인스턴스 액세스 a를 받아 이에 대응하는 쓰기 후 인스턴스 액세스를 반환한다.

- $type$, $size$, $written$, $writing$ 함수는 인스트럭션 액세스에 대해서도 유사하게 정의된다.

주소(address)는 시스템의 상태로 취급되는 위치를 나타내는 자료형이다. 즉, 시스템의 상태가 저장되고 변경될 수 있는 곳이 레지스터와 메모리라면 레지스터 공간과 메모리 공간을 통틀어 주소 공간이라고 하고, 하나의 주소는 레지스터와 메모리 상의 특정 위치를 나타내게 된다.

하나의 주소에는 해당 주소가 레지스터에 속한 주소인지 메모리에 속한 주소인지를 나타내는 메모리 공간 태그와 해당 공간에서의 주소를 나타내는 정수의 쌍으로 나타낸다. 하나의 메모리 공간은 연속된 바이트 배열로 가정하고, 한 메모리 공간에서의 주소는 이 배열에서의 색인으로 생각하며, 따라서 하나의 주소가 나타내는 위치에는 하나의 바이트 값이 저장된다.

본 논문에서는 주소 공간을 레지스터나 메모리 중의 하나로 가정했지만 이는 대상 컴퓨터 환경에 따라 달라질 수도 있다. 다만 각 주소 공간은 모두 완전히 분리된 것으로 가정한다.

주소와 관련된 함수를 다음과 같이 정의한다.

- $space(a : Address) \rightarrow register, memory$ 함수는 주소 a를 받아서 주소 공간을 반환한다. 본 연구에서는 주소가 레지스터나 메모리 중 하나를 나타내므로 $register$나 $memory$ 중 하나가 반환된다.

- $location(a : Address) \rightarrow Integer$ 함수는 주소 a를 받아서 메모리 공간에서의 주소를 나타내는 정수를 반환한다.

- 어떤 주소 a와 정수 k에 대해서, $a + k$는 $(space(a), location(a) + k)$를 의미한다.

레지스터의 위치가 정수로 바로 대응되지 않는 경우, 하나의 레지스터를 임의의 정수로 대응되는 것으로 가정한다. 물론 이 때 서로 다른 레지스터의 주소 영역이 겹쳐서는 안 된다. 예를 들어, 인텔 아키텍쳐에서의 eax 레지스터는 레지스터 주소 공간의 0부터 3까지의 공간을 차지하고, ebx 레지스터는 레지스터 주소 공간의 4부터 7까지의 공간을 차지하는 것으로 가정할 수 있다. 단, 본 논문에서 $(register, register_name e)$와 같은 표기법은 레지스터가 대응된 첫번째 주소 영역을 나타낸다. 예를 들어, $(register, eax) = (register, 0)$이다.

다음은 본 논문 전체에 걸쳐 공통적으로 사용될 변수를 정의한다.

- $T : [Instance]$는 전체 트레이스 인스턴스 목록이다.
- $T_n : Integer$은 T의 길이이다.
- Z는 모든 인스트럭션의 집합이다. 즉, $Z = \bigcup_{i \in T} \{instruction(i)\}$
- Z_i는 모든 내부 인스트럭션의 집합이다. 즉, $Z_i = \{I \in Z | place(I) = inner\}$

2.3 프로그램의 구조 파악

2.3.1 제어 흐름 그래프

제어 흐름 그래프(control flow graph; CFG)는 일반적으로 분기나 점프가 포함되지 않거나 마지막 인스트럭션으로만 포함되는 인스트럭션의 시퀀스로 구성된 베이직 블록(basic block)들을 정점(vertex)으로 하고, 하나의 베이직 블록이 실행된 이후에 실행될 수 있는 베이직 블록으로 간선(edge)을 연결하여 구성하는 그래프를 의미한다. 하지만 기존의 정적 제어 흐름 그래프에는 두 가지의 문제점이 있다.

첫째, 정적 제어 흐름 그래프는 간접 분기(indirect branch)를 정확하게 검출할 수 없다. 간접 분기 인스트럭션은 매 실행마다 다른 위치로 점프할 가능성이 있는 제어 인스트럭션이다. 매 실행마다 점프할 위치가 변경되므로 정적 분석을 통해 점프할 위치를 정확하게 찾아낼 수 없는 것은 당연하다.

둘째, 정적으로 생성된 제어 흐름 그래프는 제어 흐름 난독화에 영향을 받을 수 있다. 제어 흐름 난독화는 의미 없이 같은 곳으로 점프하는 명령을 프로그램 중간에 삽입하여 하나의 베이직 블록으로 만들어져야 할 인스트럭션 시퀀스를 여러 개로 쪼개는 것을 의미한다. 무조건 점프 외에도 항상 같은 결과를 내는 불투명 술어(opaque predicate)에 의해 분기가 결정되는 분기문이 사용되기도 한다. 정적 제어 흐름 그래프에서는 정의에 의해 점프나 분기문이 항상 베이직 블록의 마지막에 위치하기 때문에 제어 흐름 난독화가 적용된 프로그램의 경우 매우 복잡한 형태의 제어 흐름 그래프가 생성되게 된다.

동적 분석은 프로그램의 실제 수행 트레이스에 의해 분석을 수행하므로, 분석의 대상인 실행 트레이스를 통해 각 간접 분기문의 점프 위치 및 분기문의 결정을 이미 알고 있다. 이러한 관찰을 바탕으로 동적 분석에서 사용될 수 있는 새로운 제어 흐름 그래프를 정의할 수 있었다. 새로운 동적 제어 흐름 그래프는 정적 제어 흐름 그래프와 두 가지가 다르다. 첫째, 베이직 블록은 실제로 순차적으로 실행되어 외부로 점프하거나 외부로부터 점프된 적이 없는 트레이스 인스트럭션 시퀀스이다. 둘째, 정적 제어 흐름 그래프에서의 간선은 하나의 베이직 블록 사이의 제어 흐름 가능성을 나타내었지만, 동적 제어 흐름 그래프에서의 간선은 두 베이직 블록 사이에 실제로 제어 흐름 변경이 있었음을 나타낸다.

동적 제어 흐름 그래프 생성 알고리즘을 정의하기 전에, 먼저 *in* 함수와 *out* 함수를 정의하여야 한다.

- $in(I : Instruction)$ 함수는 인스트럭션 I를 받아서 I가 실행되기 직전에 실행된 적이 있는 모든 인스트럭션의 집합을 반환한다. 즉,

  $in(I) = \bigcup_{k \in g(I)} \{instruction(trace(k-1))\}$ where $g(I) = \left(\bigcup_{j \in instances(I)} \{index(j)\} \right) - \{0\}$
- $out(I : Instruction)$ 함수는 인스트럭션 I를 받아서 I가 실행된 직후에 실행된 적이 있는 모든 인스트럭션의 집합을 반환한다. 즉,

  $out(I) = \bigcup_{k \in g(I)} \{instruction(trace(k+1))\}$ where $g(I) = \left(\bigcup_{j \in instances(I)} \{index(j)\} \right) - \{T_n - 1\}$

동적 제어 흐름 그래프의 각 정점은 베이직 블록으로 나타나고, 각 간선은 두 정점의 쌍으로 나타난다. 얻어진 제어 흐름 그래프와 관련된 함수를 다음과 같이 정의하였다.

- $block(I : Instruction) \rightarrow BasicBlock$ 함수는 인스트럭션 I를 받아 I가 속한 베이직 블록을 반환한다.

- $instructions(b : BasicBlock) \rightarrow [Instruction]$ 함수는 베이직 블록 b를 받아 b에 속한 인스트럭션의 목록을 반환한다.

동적 제어 흐름 그래프를 생성하기 위해 전체 트레이스 인스턴스 목록을 처음부터 순차적으로 순회하면서, 실제로 중간에 점프해 나가거나 점프되어 들어온 적 없는 인스트럭션 시퀀스를 모아 하나의 베이직 블록으로 묶어서 이를 제어 흐름 그래프의 정점으로 등록한 다음, 인접한 두 인스트럭션이 서로 다른 베이직 블록에 속하는 경우 두 인스트럭션의 베이직 블록 사이에 간선을 만들어 제어 흐름 그래프에 등록하면 된다.

그림 2.2는 이러한 방법으로 동적 제어 흐름 그래프를 생성하는 알고리즘을 보인다.

$CFG_V \leftarrow \{\}$
$CFG_E \leftarrow \{\}$
$tmp \leftarrow \{\}$
for $i = 0 \rightarrow T_n - 1$ **do**
  $I \leftarrow instruction(trace(i))$
  if $i = 0$ **then**
    $new \leftarrow$ 새 블록
    $CFG_V \Leftarrow \{new\}$
    new에 I를 덧붙인다
  else
    $I_p \leftarrow instruction(trace(i-1))$
    if $I \notin tmp$ **then**
      if $(|in(I)| \neq 1) \vee (|out(I_p)| \neq 1)$ **then**
        $new \leftarrow$ 새 블록
        $CFG_V \Leftarrow \{new\}$
        new에 I를 덧붙인다
      else
        $block(I_p)$에 I를 덧붙인다
      end if
    end if
    if $block(I) \neq block(I_p)$ **then**
      $CFG_E \Leftarrow \{(block(I_p), block(I))\}$
    end if
  end if
  $tmp \Leftarrow \{I\}$
end for
$CFG = (CFG_V, CFG_E)$

그림 2.2: 동적 제어 흐름 그래프 생성 알고리즘

2.3.2 가상 기계의 특징

이 절은 가상화 난독화 기법에 대해 논의한다. 가상화 난독화 기법을 이용하여 난독화된 프로그램에는 무작위로 생성된 인스트럭션 집합을 실행하는 가상 기계(virtual machine; VM)과 원본 프로그램을 가상 기계가 실행할 수 있는 인스트럭션 집합으로 변환한 것이 포함되게 된다. 따라서 난독화된 프로그램은 가상적으로 변환된 원본 프로그램을 실행하므로 원본 프로그램과 동일한 동작을 하게 되지만, 난독화된 프로그램은 가상 기계를 실행하므로 그 형태는 원본 프로그램과는 완전히 다르게 된다.

난독화에 사용되는 가상 기계는 대체로 앞서 설명한 바와 같이 하나의 가상 프로그램 카운터(virtual program counter; VPC), 중심 루틴, 여러 개의 핸들러, 그리고 바이트코드와 핸들러를 대응하는 테이블을 갖는다. 가상 프로그램 카운터는 다음에 실행해야 할 가상 기계의 바이트코드가 저장되어 있는 메모리의 위치를 가리키고, 중심 루틴과 일부 핸들러에서 이 카운터를 변경시키며 프로그램을 진행시킨다. 중심 루틴은 가상 프로그램 카운터가 가리키고 있는 바이트코드를 읽어와 이를 해석하여 다음에 실행해야 할 핸들러가 저장되어 있는 메모리의 위치를 찾아내고 찾아낸 핸들러를 실제로 호출시키는 기능을 한다. 핸들러는 중심 루틴에서 바이트코드의 종류에 따라 호출되며 각 핸들러는 하나의 바이트코드에 대응하여 실제로 실행되어야 하는 내용을 수행한다.

가상 기계의 중심 루틴은 보통 내부에 분기를 갖지 않는다. 제어 흐름 난독화 과정에서 분기나 점프를 포함하게 되었더라도, 앞서 소개된 동적 제어 흐름 그래프를 생성하면 하나의 베이직 블록에 포함된다. 이를 가상 기계의 중심 블록이라고 하자. 가상 기계의 중심 블록은 가상 프로그램 카운터를 변경시키고 핸들러를 호출하는 가상 기계의 중심 역할을 하는 만큼 다음과 같은 특징을 가질 것으로 예상할 수 있다.

- 중심 블록의 실행 횟수는 다른 블록에 비해 많다.
- 동적 제어 흐름 그래프에서 중심 블록으로부터 나가는 간선의 개수는 다른 블록보다 많다.
- 중심 블록으로부터 나가는 간선이 가리키는 베이직 블록들은 바이트코드의 핸들러일 것이다.

이러한 특징들이 맞다면 이를 이용하면 가상 기계의 중심 블록을 찾아낼 수 있다. 중심 블록을 찾아낸 뒤엔, 가상 기계의 핸들러들도 찾아낼 수 있고, 핸들러를 호출하는 중심 블록의 마지막 인스트럭션에 영향을 미치는 인스트럭션을 역으로 추적하면 가상 프로그램 카운터가 저장되는 위치와 이를 이용해 바이트코드를 불러오는 인스트럭션을 찾을 수 있다.

4.3에서 실험을 통해 이들 특징이 실제로 맞는지를 확인한다.

2.3.3 액세스 맵

액세스 맵은 하나의 인스트럭션을 선택한 뒤 선택된 인스트럭션이 읽거나 쓰는 주소를 보이는 것을 말한다. 이는 가상화 난독화된 프로그램에서 가상 프로그램 카운터와 바이트코드를 읽어오는 인스트럭션을 찾아낸 뒤에 특히 유용하다. 바이트코드를 읽어오는 인스트럭션을 선택하고 해당 인스트럭션이 읽은 적이 있는 영역은 바이트코드가 저장되어 있는 영역일 가능성이 높기 때문이다.

액세스 맵을 보이기 위하여 함수 $access$를 다음과 같이 정의한다.

- $access(i : Instance) \rightarrow \{Address\}$ 함수는 인스턴스 i를 받아 i의 인스턴스 액세스들이 읽거나 쓴 주소의 집합을 반환한다. 즉,

  $access(i) = \bigcup_{a \in accesses(i)} addresses(a)$

- $access(I : Instruction) \rightarrow \{Address\}$ 함수는 앞의 $access$ 함수를 인스턴스에서 인스트럭션으로 추상화한 것이다. 즉,

  $access(I) = \bigcup_{i \in instances(I)} access(i)$

반대로 한 주소를 읽거나 쓰는 인스트럭션을 찾기 위한 함수도 다음과 같이 정의할 수 있다.

- $accessat(a : Address) \rightarrow \{Instruction\}$ 함수는 주소 a를 받아서 해당 주소를 읽거나 쓴 인스트럭션의 집합을 반환한다. 즉,

  $accessat(a) = \{I \in Z_i | a \in access(I)\}$

2.4 프로그램의 의미 파악

2.4.1 값 히스토리

대부분의 인스트럭션 아키텍처는 시스템의 상태를 변경하면서 프로그램을 실행하는 명령형으로 설계되어 있다. 따라서, 시스템의 상태는 프로그램이 실행되는 동안 지속적으로 변경된다. 이러한 상태 변경은 실행 트레이스에 인스턴스 액세스의 형태로 저장되며, 저장된 인스턴스 액세스를 이용하면 인스턴스 사이의 시스템 상태를 재현할 수 있다.

어떤 주소 a와 정수 $0 \leq n < T_n$에 대하여 $V_n^a : Byte$는 $trace(n)$이 실행된 직후에 주소 a에 저장된 바이트 값을 나타낸다. V_n은 $trace(n)$이 실행된 직후의 시스템 상태에서 트레이스에 저장된 정보로부터 저장된 값을 알아낼 수 있는 메모리 주소의 집합을 나타낸다.

그림 2.3은 V를 계산하기 위한 알고리즘을 보이고 있다. U_n은 $trace(n)$에 의해서 값이 변경된 주소의 집합을 나타낸다.

```
for n = 0 → T_n − 1 do
    U_n ← {}
    for all a ∈ accesses(trace(n)) do
        if type(a) = written then
            for all k ∈ addresses(a) do
                V_n^k ← value(a, k)
                U_n ⇐ {k}
            end for
        end if
    end for
    if n > 0 then
        for all k ∈ V_{n−1} − U_n do
            V_n^k ← V_{n−1}^k
        end for
    end if
end for
```

그림 2.3: 값 히스토리 계산 알고리즘

2.4.2 의존성 추적기

서로 다른 인스턴스에 속한 두 개의 인스턴스 액세스 a_0과 a_1가 있다고 하자. a_0은 먼저 실행된 쓰기 액세스이고 a_1는 나중에 실행된 읽기 액세스이면서, a_0이 쓴 값을 다른 인스턴스가 덮어쓰지 않고 a_1이 읽은 적이 있다면 우리는 a_1이 읽어서 사용한 값이 a_0에 의존적이다 혹은 a_0이 a_1에 영향을 준다고 말한다.

이러한 관계를 인스턴스 액세스들을 정점으로 하고 영향을 준 액세스에서 영향을 받은 액세스로의 간선을 가진 그래프로 나타낼 수 있고, 이러한 그래프를 의존성 그래프라고 한다. 의존성 그래프는 데이터의 흐름과 계산 과정을 보여주므로 프로그램의 알고리즘 혹은 의미를 파악하는 데 도움을 준다.

의존성 그래프 D를 다음과 같이 정의한다.

- $D = (D_V, D_E)$
- $D_V : \{InstanceAccess\}$는 모든 인스턴스 액세스의 집합이다. 즉, $D_V = \bigcup_{i \in T} accesses(i)$
- $D_E : \{(InstanceAccess, InstanceAccess)\}$는 인스턴스 액세스 사이의 단방향 간선을 나타내는 두 인스턴스 액세스의 쌍의 집합이다.
- $(f, s) \in D_E$는 f가 s에 영향을 줌을 의미한다. 다른 말로는 s가 f에 의존적임을 의미한다.
- 영향을 주는 인스턴스 액세스는 쓰기 액세스여야하고 영향을 받는 액세스는 읽기 액세스여야 한다. 즉, $\forall (f, s) \in D_E$, $type(f) = written$ 이고 $type(s) = read$ 이다.

의존성 그래프와 유사하게 덮어쓰기 그래프 W를 다음과 같이 정의한다.

- $W = (W_V, W_E)$
- $W_V : \{InstanceAccess\}$는 모든 쓰기 후 인스턴스 액세스의 집합이다. 즉, $W_V = \{a \in D_V | type(a) = written\}$
- $W_E : \{(InstanceAccess, InstanceAccess)\}$는 인스턴스 액세스 사이의 단방향 간선을 나타내는 두 인스턴스 액세스의 쌍의 집합이다.
- $(f, s) \in W_E$는 f에 의해 쓰여진 값을 s가 덮어썼음을 의미한다.
- 덮어씌워지는 값과 덮어쓰는 액세스는 모두 쓰기 후 인스턴스 액세스에 의해 기록되므로 $\forall (f, s) \in W_E$, $type(f) = type(s) = written$

의존성 그래프와 덮어쓰기 그래프를 생성하기 위해서는 우선 덮어쓰기 딕셔너리 O를 먼저 생성해야 한다. O_n은 $trace(n)$이 실행되기 직전에 각 주소 위치로부터 각 위치의 값을 가장 마지막으로 덮어 쓴 인스턴스 액세스로 대응하는 딕셔너리이다. 즉, $O_n : \{(Address \rightarrow InstanceAccess)\}$로, O_n^a는 $trace(n)$이 실행되기 바로 직전의 상태에서 주소 a에 가장 마지막으로 값을 덮어쓴 인스턴스 액세스를 나타내게 된다. 따라서, $type(O_n^a) = written$이다. 편의를 위해, $O_{-1} = \{\}$로 가정한다.

그림 2.4에서 덮어쓰기 딕셔너리 O와 덮어쓰기 그래프 W를 계산하는 알고리즘을 보이고 있다. 이어서 그림 2.5는 O를 이용해 의존성 그래프를 생성하는 과정을 보이고 있다.

for $n = 0 \rightarrow T_n - 1$ **do**
  for all $a \in accesses(trace(n))$ **do**
    if $type(a) = written$ **then**
      for all $k \in addresses(a)$ **do**
        if $k \in keys(O_n)$ **then**
          $W_E \Leftarrow \{(O_n^k, a)\}$
        end if
        $O_n^k \leftarrow a$
      end for
    end if
  end for
  for all $k \in keys(O_{n-1}) - keys(O_n)$ **do**
    $O_n^k \leftarrow O_{n-1}^k$
  end for
end for

그림 2.4: 덮어쓰기 딕셔너리 및 그래프 생성 알고리즘

$D_E \leftarrow \{\}$
for $n = 0 \rightarrow T_n - 1$ **do**
  for all $a \in accesses(trace(n))$ **do**
    if $type(a) = read$ **then**
      for all $k \in addresses(a)$ **do**
        if $k \in keys(O_n)$ **then**
          $D_E \Leftarrow \{(O_n^k, a)\}$
        end if
      end for
    end if
  end for
end for

그림 2.5: 의존성 그래프 생성 알고리즘

의존성 그래프 D와 덮어쓰기 그래프 W의 정보를 이용하는 함수를 다음과 같이 정의한다.

- $depends(a : InstanceAccess) \rightarrow \{InstanceAccess\}$ 함수는 인스턴스 액세스 a를 받아 a에 영향을 주는 모든 인스턴스 액세스들의 집합을 반환한다. 즉, $depends(a) = \bigcup_{(k,a)\in D_E}\{k\}$ 이다. 영향을 주는 액세스는 항상 쓰기 액세스이고, 영향을 받는 액세스는 항상 읽기 액세스이므로 a는 항상 읽기 액세스여야 한다. 만약 a가 읽기 액세스가 아니면 $depends(a) = \{\}$로 가정한다.

- $forwards(a : InstanceAccess) \rightarrow \{InstanceAccess\}$ 함수는 인스턴스 액세스 a를 받아 a가 영향을 주는 모든 인스턴스 액세스들의 집합을 반환한다. 즉, $forwards(a) = \bigcup_{(a,k)\in D_E}\{k\}$ 이다. 영향을 주는 액세스는 항상 쓰기 액세스이고, 영향을 받는 액세스는 항상 읽기 액세스이므로 a는 항상 쓰기 액세스여야 한다. 만약 a가 쓰기 액세스가 아니면 $forwards(a) = \{\}$로 가정한다.

- $overwrites(a : InstanceAccess) \rightarrow \{InstanceAccess\}$ 함수는 인스턴스 액세스 a를 받아 a가 a가 기록한 값을 덮어쓴 모든 인스턴스 액세스들의 집합을 반환한다. 즉, $overwrites(a) = \bigcup_{(a,k)\in W}\{k\}$ 이다. a는 반드시 쓰기 인스턴스 액세스여야 하고 만약 a가 쓰기 액세스가 아니면 $overwrites(a) = \{\}$로 가정한다.

인스트럭션 액세스에 대해서 추상화하는 $depends$, $forwards$, $overwrites$ 함수를 다음과 같이 정의할 수 있다.

- $depends(A : InstructionAccess) \rightarrow \{InstructionAccess\} =$

  $\bigcup_{a\in instanceaccesses(A)}[g(a)]$ where

  $g(a) = \bigcup_{d\in depends(a)}(instructionaccess(d))$.

- $forwards$ 함수와 $overwrites$ 함수도 이와 비슷하게 정의된다.

의존성 그래프가 분석 대상 프로그램의 계산 과정을 잘 나타내 주기는 하지만, 실제 분석에서 사용하기에 너무 크고 복잡할 뿐만 아니라, 분석자가 실제로 관심을 갖는 연산 과정은 한정되어 있기 때문에 특정 값과 연관된 계산 과정만을 포함하는 의존성 그래프의 부분집합을 추출하여 분석에 사용할 필요가 있다. 특히 일반적으로 외부 루틴들이 프로그램의 실질적인 동작을 결정하기 때문에 외부 루틴에 인자로 전달되는 값을 기록하는 인스턴스 액세스와 관련된 내용을 추출하여 그 값이 어떻게 계산되었는지 과정을 아는 것이 프로그램 분석에 도움이 될 때가 많다.

의존성 그래프의 부분집합을 부분 의존성 그래프(subdependency graph)라고 부르기로 하고, 특히 특정한 값에 직간접적으로 영향을 미치는 의존성 관계만을 포함하고 있는 부분 의존성 그래프를 특정 값에 연관된 부분 의존성 그래프(relevant subdependency graph)라고 부르기로 하자. 부분 의존성 그래프 S는 다음과 같은 특성을 갖는다.

- 정의에 따라, S는 의존성 그래프 D의 부분집합이다. 즉,

  $S = (S_V, S_E)$ 라고 할 때, $S_V \subseteq D_V$ 이고 $S_E \subseteq D_E$ 이다.

- $S_V = \bigcup_{(f,s)\in S_E}\{f, s\}$

구현된 의존성 추적기에서는 특정한 값을 지정하면 해당 값에 연관된 부분 의존성 그래프를 자동으로 추적하는 기능이 포함되어 있다.

2.4.3 수식 트리

분석자가 관심 있는 특정 값에 연관된 부분 의존성 그래프를 추출하더라도 난독화된 프로그램으로부터 얻어낸 그래프가 원본 프로그램의 그래프보다 일반적으로 더 복잡하게 나타난다. 예를 들면 난독화된 프로그램의 계산 과정에는 원본 프로그램보다 많은 횟수의 move 명령이 나타나는 등이다. 수식 트리는 이러한 경우에 부분 의존성 그래프에서 사용자가 관심 없는 것으로 설정한 인스트럭션에 속한 인스턴스의 액세스들은 제거하여 분석자가 대상 프로그램의 계산 과정을 보다 분명히 볼 수 있게 도울 수 있도록 고안된 구조이다.

수식 트리를 생성하기 위해서는 우선 부분 의존성 그래프 S와 더불어 관심 있는 인스트럭션의 집합 N을 설정해야 한다. 관심 있는 인스트럭션이란 분석자가 그 역할을 확인하기 위해 수식 트리에 포함시키고자 하는 인스트럭션을 의미한다.

수식 트리와 부분 의존성 그래프의 근본적 차이는 우회 간선(bypassing edge)의 존재 여부이다. 세 개의 인스턴스 액세스 a, b, c가 있고, a가 b에, b가 c에 영향을 준다고 하자. 이 때 b의 인스턴스가 관심 없는 인스트럭션으로 설정된다면 분석자는 b에 연관된 의존성 관계는 보고 싶지 않을 것이다. 따라서 부분 의존성 그래프의 (a, b)와 (b, c) 간선들 대신 수식 트리에는 우회 간선인 (a, c)가 포함되게 된다. 우회 간선의 존재로 인해 수식 트리는 의존성 그래프의 부분 집합이 아닐 수 있다.

어떤 부분 의존성 그래프 S와 관심 있는 인스트럭션의 집합 N에 대한 수식 트리 E를 다음과 같이 정의한다.

- $E = (E_V, E_E)$
- $E_V : \{InstanceAccess\} \subset S_V$
- $E_E : \{(InstanceAccess, InstanceAccess)\}$
- 우회 간선으로 인해 E_E 는 S_E 의 부분집합이 아닐 수 있다.

수식 트리를 생성하는 알고리즘에서는 우회 간선을 찾는 것이 기본이 되므로 우회 간선을 찾는 함수 $forwardbypass$를 다음과 같이 정의한다.

- $forwardbypass(i : InstanceAccess) \rightarrow \{InstanceAccess\}$ 함수는 인스턴스 액세스 i를 받아 수식 트리에서 i를 대체할 인스턴스 액세스들의 집합을 반환한다. 의존성 그래프에는 $depends(i)$의 각 원소로부터 i로 연결되는 간선이 포함되어 있지만, 수식 트리에서는 $depends(i)$의 각 원소로부터 $forwardbypass(i)$의 각 원소로 연결되는 우회 간선이 포함되게 된다. i는 반드시 읽기 인스턴스 액세스여야 하며, $forwardbypass$가 반환하는 인스턴스 액세스도 모두 읽기 액세스여야 한다.
- $backwardbypass$ 함수도 유사하게 정의될 수 있으나 여기서는 $forwardbypass$ 함수만 사용하므로 생략한다.

그림 2.6은 $forwardbypass$ 함수를 정의하고 있고, 이어서 그림 2.7은 $forwardbypass$ 함수를 이용해 부분 의존성 그래프 S 와 관심 있는 인스트럭션의 집합 N 으로부터 수식 트리 E 를 구하는 알고리즘을 보이고 있다.

수식 트리의 실제 구현에서는 사용을 간편하게 하기 위하여 관심 있는 인스트럭션을 직접 지정하는 대신 관심 있는 opcode를 지정하도록 단순하게 변경되었다. 즉, 관심 있는 인스트럭션의 집합 N은 관심있는 opcode의 집합 P로부터 다음의 수식에 의해 정해진다.

$N = \{i \in Z_i | opcode(i) \in P\}$.

procedure FORWARDBYPASS($i : InstanceAccess$)
  if $instruction(instance(i)) \in N$ **then**
    return $\{i\}$
  else
    $result \leftarrow \{\}$
    for all $j \in forwards(i)$ **do**
      for all $k \in accesses(j)$ **do**
        if $(type(k) = read) \wedge (k \in S_V)$ **then**
          $result \Leftarrow forwardbypass(k)$
        end if
      end for
    end for
    return $result$
  end if
end procedure

그림 2.6: $forwardbypass$ 함수

$E_V \leftarrow \{\}$
$E_E \leftarrow \{\}$
for all $(f, s) \in S_E$ **do**
  if $instruction(instance(f)) \in N_V$ **then**
    $bypasses \leftarrow forwardbypass(s)$
    $E_V \Leftarrow \{f\} \cup bypasses$
    $E_E \Leftarrow \bigcup_{k \in bypasses} \{(f, k)\}$
  end if
end for

그림 2.7: 수식 트리 생성 알고리즘

2.5 프로그램의 최적화

2.5.1 최적화

동적 분석을 통해 프로그램의 의미를 파악하는 것 외에 난독화된 프로그램을 최적화(Optimization)할 수 있다. 최적화란 보통 컴파일러 개발에서 보다 시간 및 공간 효율적인 코드를 생성해내는 기법들을 의미하지만, 본 논문에서는 이와 달리 난독화된 프로그램을 보다 분석하기 용이한 형태로 바꾸어주는 것을 의미한다. 최적화된 프로그램은 원래의 프로그램과 동일한 환경에서 동일한 실행 경로로 실행했을 때 같은 실행 결과를 내야 한다.

VMProtect나 Code Virtualizer와 같은 상용 도구들을 이용해 난독화를 적용하는 경우, 가상 기계를 삽입하는 것뿐만 아니라 다양한 코드 난독화 기법들을 기본적으로 적용하기 때문에 분석이 더욱 어려워진다. 이러한 부수적인 난독화 기법들을 제거하면 분석이 보다 용이해질 것이다. 정적 분석 방법은 난독화된 프로그램에 대한 가정 때문에 간단한 난독화 기법만 추가되어도 적용될 수 없게 되는 경우가 있었는데, 최적화 과정을 거치면 사용할 수 없게 되었던 기존 연구도 다시 사용할 수 있게 되는 경우도 있다.

프로그램 최적화 과정은 기본적으로 난독화 기법을 해제하는 과정이다. 논문에서 제시된 최적화 기법을 이용하면 죽은 코드 삽입이나 데이터 인코딩 난독화 기법 등을 실행 트레이스를 분석해서 해제할 수 있다. 하지만 프로그램 최적화는 한정된 난독화 기법만 해제할 수 있으므로 모든 난독화를 완전히 해제하여 난독화된 프로그램으로부터 원본 프로그램을 만들어낼 수는 없다. 예를 들어, 개발된 최적화 기법을 가상화 난독화를 이용하여 난독화된 실행 파일에 적용하면 일부 난독화 기법을 해제하여 분석자가 보다 용이하게 분석에 사용할 수 있는 실행 파일의 형태로 바꿀 수는 있지만 가상 기계를 제거하지는 못한다. 하지만 앞서 언급한 바와 같이 프로그램 최적화의 목적은 보다 분석에 용이한 실행 파일을 생성해내는 것이므로 한정된 난독화 기법만을 해제하는 것도 의의가 있다.

최적화는 기본적으로 없어도 무방한 인스트럭션을 찾아 제거하는 것이다. 인스트럭션을 제거한다는 것은 이를 NOP 인스트럭션(아무 일도 하지 않는 인스트럭션)으로 변경하여 분석자가 무시할 수 있도록 만드는 것이다. 다만 최적화 과정에서 외부 인스트럭션은 모두 반드시 필요한 것으로 가정하며, 따라서 제거할 수 있는 인스트럭션의 대상은 내부 인스트럭션으로 한정된다.

최적화가 적용된 프로그램은 적용되지 않은 프로그램과 동일한 실행 결과를 내야 하기 때문에 인스트럭션을 제거할 때 다른 인스트럭션을 변경하여 새로운 인스트럭션으로 대치하거나, 혹은 프로그램의 데이터 영역 일부를 새로운 데이터로 수정해야 하는 경우가 생긴다. 따라서 최적화는 (제거될 수 있는 인스트럭션 $Dropped$, 수정된 인스트럭션 $Modified$, 수정된 데이터 영역 $Data$)의 길이가 3인 튜플로 정의할 수 있다.

- $Optimization = (Dropped, Modified, Data)$
- $Dropped : \{Instruction\}$ 는 제거할 수 있는 인스트럭션의 집합이다. 외부 인스트럭션은 제거할 수 없으므로 $Dropped$에는 내부 인스트럭션만 포함된다. 어떤 인스트럭션, 혹은 인스트럭션의 집합을 제거한다는 것은 $Dropped$ 집합에 해당 인스트럭션이나 해당 인스트럭션 집합의 원소들을 추가한다는 의미이다.
- $Modified : \{(Instruction \rightarrow Instruction)\}$ 은 원본 인스트럭션으로부터 수정된 새 인스트럭션으로 대응하는 딕셔너리이다. 인스트럭션 I를 새 인스트럭션 J로 수정한다는 것은 $Modified^I \leftarrow J$ 를 의미한다.
- $Data : \{(Address \rightarrow Byte)\}$ 는 특정 주소로부터 수정된 새 바이트 값으로 대응하는 딕셔너리이다. 주소 a를 새 바이트 값 b로 수정한다는 것은 $Data^a \leftarrow b$ 를 의미한다.

최적화는 다음과 같이 일곱 단계로 구성되어 있다.

1. 마일스톤 설정
2. 의존성 분석을 통한 가지치기
3. 의미 없는 인스트럭션 제거
4. 효과 없는 인스트럭션 제거
5. 효용 없는 스택 연산 제거
6. 인스트럭션 체인 약분
7. 수정된 실행 파일 생성

1단계와 7단계는 반드시 수행되어야 하는 과정이며, 2단계부터 6단계까지는 선택적으로 적용할 수 있다. 7단계는 엄밀히 말해 최적화 단계가 아니라 앞서의 과정에서 얻어진 최적화 정보를 반영하여 새로운 실행 파일을 생성하는 과정이지만 최적화의 마지막 단계로 취급하였다.

2.5.2 마일스톤 설정

최적화 대상 프로그램의 내부 인스트럭션 중 어떤 것들은 프로그램 실행에 필수적이어서 절대로 삭제되어서는 안 된다. 이러한 인스트럭션을 영어로 이정표를 의미하는 마일스톤 인스트럭션이라고 부른다. 마일스톤 설정 과정은 수행되어야 하는 최적화의 첫번째 단계로 대상 프로그램 실행에 필수적이어서 삭제되어서는 안 되는 인스트럭션을 찾아 마일스톤 인스트럭션으로 설정하는 과정이다.

마일스톤은 세 가지 기준으로 설정된다. 외부 인스트럭션의 바로 앞에서 실행된 적이 있는 인스트럭션, 외부 루틴에 영향을 주는 인스트럭션, 그리고 각 베이직 블록의 마지막 인스트럭션이다. 외부 인스트럭션의 바로 앞에서 실행된 적이 있는 인스트럭션은 이 인스트럭션이 외부 루틴을 호출하는 역할을 했음을 의미하기 때문에 제거되어선 안 된다. 외부 루틴에 영향을 주는 인스트럭션은 외부 루틴에 인자를 제공하여 외부 루틴의 동작을 결정하기 때문에 제거되어선 안 된다. 각 베이직 블록의 마지막 인스트럭션은 실제로 프로그램의 제어 흐름을 변경하는 인스트럭션이므로 마일스톤으로 설정해서 보호해야 한다.

마일스톤을 찾는 알고리즘을 기술하기 전에 알고리즘을 간단하게 쓰기 위하여 $influences$ 함수를 다음과 같이 정의한다.

- $influences(I : Instruction) \rightarrow \{Instruction\}$ 함수는 인스트럭션 I의 쓰기 액세스들이 영향을 미치는 모든 액세스들의 인스트럭션의 집합이다. 따라서 다음과 같이 쓸 수 있다.

  $influences(I) = \bigcup_{A \in g(I)} \{instruction(A)\}$ where $g(I) = \bigcup_{k \in accesses(I)} forwards(k)$

  이 때, 마일스톤 인스트럭션의 집합 M은 다음과 같이 쓸 수 있다.
  $M = \{k \in Z_i |$
  $(out(k) \not\subseteq Z_i) \ \vee$
  $(\{j \in influences(k) | j \notin Z_i\} \neq \{\}) \ \vee$
  $(k = the\ last\ item\ of\ instructions(block(k)))\}$

2.5.3 의존성 분석을 통한 가지치기

마일스톤이 설정되면, 마일스톤에 직간접적으로 영향을 주는 인스트럭션을 찾을 수 있는데, 이들을 관계 있는 인스트럭션(relevant instruction)이라고 한다. 관계 있는 인스트럭션은 마일스톤에 영향을 주어 간접적으로 외부 루틴의 동작에 영향을 미치므로 삭제되어선 안 되지만, 그 외의 인스트럭션은 외부 인스트럭션에 전혀 영향을 주지 않으므로 제거해도 된다.

그림 2.8은 관계있는 인스트럭션을 찾는 알고리즘의 의사 코드를 보이고 있다. 여기서 $R : \{Instance\}$은 인스턴스들의 집합으로 여기 속한 인스턴스들의 인스트럭션은 모두 관계 있는 인스트럭션이다. 즉, $\{I \in Z_i | I \in \{instruction(i) | i \in R\}\}$는 관계 있는 인스트럭션이다. 따라서 R을 계산한 뒤에 이 집합에 포함되지 않는 내부 인스트럭션은 모두 제거할 수 있다.

procedure RELEVANT($i : Instance, R : \{Instance\}$)
  if $(i \notin R) \wedge (place(instruction(i)) = inner)$ **then**
    $R \Leftarrow \{i\}$
    for all $a \in accesses(i)$ **do**
      for all $d \in depends(a)$ **do**
        $R \Leftarrow relevant(d, R)$
      end for
    end for
  end if
  return R
end procedure
$R \leftarrow \{\}$
for all $I \in M$ **do**
  for all $i \in instances(I)$ **do**
    $R \leftarrow relevant(i, R)$
  end for
end for

그림 2.8: 의존성 분석을 통한 가지치기 알고리즘

2.5.4 의미 없는 인스트럭션 제거

가지치기를 거친 후에 의미 없는 인스트럭션을 제거할 수 있다. 의미 없는 인스트럭션에는 두 가지 종류가 있다. 하나는 점프나 분기문과 같은 제어 흐름 인스트럭션들 중 베이직 블록의 마지막에 위치하지 않는 것들이고, 또 다른 하나는 프로그램의 실행 과정에서 단 한번도 시스템의 상태를 변경한 적이 없는 인스트럭션들이다.

제어 흐름 인스트럭션이 블록의 마지막에 위치하지 않는다는 것은 실질적으로 제어 흐름을 바꾼 적이 없음을 의미하는데, 제어 흐름에 실질적인 영향을 미치지 못한 제어 흐름 인스트럭션은 당연히 무의미할 것이다. 다른 경우는 단 한번도 시스템의 상태, 즉 레지스터나 메모리의 값을 변경한 적이 없는 인스트럭션이다. 달리 말해, 어떤 내부 인스트럭션의 인스턴스들에 속한 모든 쓰기 전 인스트럭션과 그에 대응하는 쓰기 후 인스트럭션의 값이 항상 같으면 이 인스트럭션은 시스템의 상태를 실질적으로 바꾼 적이 없으므로 무의미하며 제거할 수 있다.

무의미한 제어 흐름 인스트럭션 $F_{control}$과 F_{effect}를 찾는 알고리즘을 그림 2.9에서 볼 수 있다.

$F_{control} \leftarrow \{\}$
for all $I \in Z_i$ **do**
  if I is a control flow instruction $\wedge$ $I \neq$ *the last item of* $instructions(block(I))$ **then**
    $F_{control} \Leftarrow \{I\}$
  end if
end for
$F_{effect} \leftarrow \{\}$
for all $I \in Z_i$ **do**
  if $\forall i \in instances(I), \forall a \in accesses(i), ((type(a) = written) \wedge value(writing(a)) = value(a))$ **then**
    $F_{effect} \Leftarrow \{I\}$
  end if
end for

그림 2.9: 의미 없는 인스트럭션 검색 알고리즘

2.5.5 효과 없는 인스트럭션 제거

최적화가 진행되어 인스트럭션이 제거되었을 때, 제거된 인스트럭션에만 영향을 미치는 인스트럭션이 남게될 수 있는데 이들을 효과 없는 인스트럭션이라 한다. 이들은 프로그램 실행에 영향을 미치지 않으므로 제거할 수 있다. 효과 없는 인스트럭션의 집합은 다음과 같이 쓸 수 있다.

$\{k \in Z_i | influences(k) \subseteq Dropped\}$

2.5.6 효용 없는 스택 연산 제거

이 절은 효용 없는 스택 연산을 찾고 제거하는 방법을 논의한다. 따라서 스택 연산 명령을 갖고 있는 시스템 환경만 고려하게 되는데, 본 절은 시스템의 인스트럭션 집합 아키텍쳐(Instruction Set Architecture; ISA)에 많은 영향을 받으므로 32비트 인텔 아키텍쳐, 즉 IA32를 가정하고 스택 연산 검색 및 제거를 논의한다.

IA32는 메모리의 일부 영역을 스택에 할당하고 PUSH나 POP과 같은 스택 연산을 제공하여 이 영역을 제어할 수 있게 한다. 스택의 꼭대기 값이 저장되어 있는 메모리 위치는 ESP 레지스터에 저장되고, 가장 아래 값이 저장되어 있는 메모리 위치는 EBP 레지스터에 저장된다. 또 스택은 주소가 낮은 쪽으로 쌓인다. 즉, PUSH 인스트럭션을 이용해 새로운 값이 스택에 쌓이면 ESP 값은 작아지고, EBP 레지스터의 값은 항상 ESP 레지스터의 값보다 크거나 같다.

하지만 IA32에서의 스택 메모리 영역은 PUSH와 POP 외의 명령으로도 제어할 수 있기 때문에 이들 스택 연산 명령은 단순한 래핑(wrapping) 명령이라고 생각해도 된다. 예를 들어, PUSH 명령은 스택으로 값을 복사하는 MOV 명령과 ESP 레지스터의 값을 변경하는 SUB 명령의 조합이라고 생각할 수 있다. 이처럼 스택을 이용하는 데 있어 PUSH와 POP 명령만 이용해야 한다는 등의 강력한 제약이 없기 때문에 스택에서 꼭대기로부터 4번째 값(16번째=0x10번째 바이트)을 EAX레지스터로 옮기기 위해 다음과 같은 명령을 이용할 수 있다.

```
MOV EAX, [ESP+0x10]
```

그래서 스택 연산 명령을 제거할 때 PUSH와 대응되는 POP 사이의 인스트럭션 중 일부를 수정해야 하는 경우가 발생할 수 있다. 예를 들어, PUSH와 대응되는 POP을 지우려고 하는데 PUSH에 바로 이어서 스택의 꼭대기에서 4번째 값을 읽거나 쓰는 명령이 나오는 상황을 가정하자. 이러한 경우, PUSH가 제거되면 스택의 꼭대기에서 4번째에 위치하던 값이 3번째에 위치하는 값이 될 것이므로 이 인스트럭션을 수정해야 한다. 이처럼 스택 연산을 제거하는 일은 상당히 많은 상황을 고려해야 한다.

하지만 이와 같이 스택 연산을 제거하는 것이 복잡한 반면, 효용 없는 스택 연산을 찾는 일은 상대적으로 간단하다. 스택 연산이 의미가 있는 경우는 두 가지로 나누어볼 수 있다. 값이 의미 있는(value-meaningful) 경우와 위치가 의미 있는(position-meaningful) 경우이다. 정상적인 PUSH라면 의미 있는 값, 즉 POP된 이후에 사용되거나 혹은 최소한 POP이 실행되기 전에 값으로 읽혀서 사용되는 값을 저장하게 된다. 이러한 경우 PUSH가 스택에 저장한 값을 읽는 인스트럭션이 존재하게 되고, 이러한 PUSH를 값이 의미 있는 스택 연산이라고 한다. 반면 PUSH가 스택에 저장한 값을 읽어서 사용하는 인스트럭션은 없지만 그 위치에 다른 값을 덮어 씌워서 사용하는 경우는 있는 경우, 이러한 PUSH를 위치가 의미 있는 스택 연산이라고 한다. 값도 의미가 없고 위치도 의미가 없는 스택 연산은 효용 없는 스택 연산이므로 제거할 수 있다.

스택 연산의 효용성을 확인하고 실제로 제거하기 전에 우선 PUSH와 이에 대응하는 POP을 검색해야 한다. IA32에서는 PUSH에 의해서 스택에 값이 푸시된 이후에 이 값이 팝될 때 반드시 POP을 이용하지 않아도 되기 때문에 이 또한 간단치 않다. 그래서 PUSH가 실행된 이후 처음으로 ESP 레지스터의 값이 PUSH가 실행되기 직전의 값보다 커지도록 하는, 즉 스택의 꼭대기가 PUSH가 값을 넣기 전보다 낮아지거나 같아지도록 바꾸는 인스트럭션을 이 PUSH에 대응되는 팝 인스트럭션으로 간주해야 한다. 이렇게 하기 위해 각 PUSH 인스트럭션에 대해, 각각의 인스턴스 이후에 실행되는 인스턴스를 검색해 보아야 한다.

PUSH와 이에 대응되는 팝 인스트럭션을 찾을 때, 두 인스트럭션 사이를 순회하면서 그들 중 수정되어야 할 인스트럭션들도 함께 검색한다. 두 인스트럭션 사이의 인스트럭션들을 내부자(insider) 인스트럭션들이라고 부르고, 그들 중 수정되어야 하는 인스트럭션들을 영향을 받는(affected) 인스트럭션들이라고 부른다.

이 과정에서 유효하지 않은(invalid) 스택 연산 쌍을 검색할 수 있다. 하나의 PUSH에 대응하는 팝 인스트럭션이 두 개 이상인 경우, 하나의 스택 연산 쌍의 내부자 인스트럭션이나 영향을 받는 인스트럭션들의 집합이 각 스택 연산 쌍의 인스턴스들에서 같지 않은 경우가 존재하는 경우에는 해당 스택 연산 쌍은 유효하지 않은 것으로 하고 무시한다.

집합 SO의 원소들은 하나의 유효한 스택 연산 쌍을 나타내는 길이가 4인 튜플로, (PUSH 인스트럭션, 대응하는 pop 인스트럭션, 푸시된 값의 크기, 영향을 받는 인스트럭션들의 집합)으로 구성된다. 내부자 인스트럭션들의 집합은 스택 연산 쌍의 유효성을 검증할 때 사용되고 이후에는 필요치 않으므로 저장되지 않는다.

유효한 스택 쌍을 검색하는 알고리즘을 정의하기 위해 필요한 함수들을 다음과 같이 정의하였다.

- $memop(i : Instance) \rightarrow \{InstanceAccess\}$ 함수는 인스턴스 i를 받아 i에 속한 읽기 혹은 쓰기 후 인스턴스 액세스들 중 메모리에 접근하는 액세스를 찾아서 반환한다. i의 인스트럭션은 반드시 내부 인스트럭션이어야 한다. 만약 인스트럭션에 메모리에 접근하는 액세스가 없다면 공집합을 반환한다. IA32의 명령 체계에서는 하나의 명령이 많아야 한 개의 메모리 오퍼랜드를 가질 수 있으므로 $memop$ 함수가 반환하는 집합의 크기는 최대 1이다. 수식으로 나타내면 다음과 같다.

  $$memop(i) = \{a \in accesses(i) | (type(a) \in \{read, written\}) \wedge (space(address(a)) = memory)\}$$

- $espaccess(i : Instance) \rightarrow \{InstanceAccess\}$ 함수는 인스턴스 i를 받아 i에 속한 읽기 인스턴스 액세스들 중 ESP 레지스터를 읽는 인스턴스가 있으면 찾아서 반환한다. ESP 레지스터를 읽는 인스턴스 액세스가 있다면 원소가 단 하나인 집합을 반환하게 되고 없는 경우 공집합을 반환한다. 수식으로 쓰면 다음과 같다.

  $$espaccess(i) = \{a \in accesses(i) | (type(a) = read) \wedge (address(a) = (register, esp))\}$$

- $pushedvalue(i : Instance) \rightarrow InstanceAccess$ 함수는 PUSH 인스트럭션의 인스턴스 i를 받아 i가 스택에 기록하는 쓰기 인스턴스 액세스를 찾아서 반환한다.

그림 2.10에서 유효한 스택 연산 쌍의 집합인 SO를 찾는 알고리즘을 보이고 있다.

$SO \leftarrow \{\}$
for all $I \in Z_i$ **do**
  if $opcode(I) \in \{PUSH, PUSHF, PUSHA\}$ **then**
    $pop \leftarrow \{\}$
    $psz \leftarrow nil$ ▷ 푸시된 값의 크기
    $ins \leftarrow nil$ ▷ 내부자 인스트럭션
    $aff \leftarrow nil$ ▷ 영향을 받는 인스트럭션
    for all $i \in instances(I)$ **do**
      $esp \leftarrow espaccesses(i)$ ▷ $|esp|$는 반드시 1이어야 함
      $k \leftarrow index(i)$
      $psz_i \leftarrow size(pushedvalue(i))$
      $ins_i \leftarrow \{\}$
      $aff_i \leftarrow \{\}$
      repeat
        $k \leftarrow k + 1$
        if $k \geq T_n$ **then**
          goto *invalid*
        end if
        $ins_i \Leftarrow \{instruction(trace(k))\}$
        $mem \leftarrow memop(trace(k))$
        if $(mem \neq \{\}) \wedge (address(mem)\ is\ in\ stack\ area) \wedge address(mem) \geq esp$ **then**
          $aff_i \Leftarrow \{instruction(trace(k))\}$
        end if
        $esp_n \leftarrow espaccess(trace(k))$
      until $(|esp_n| \neq 1) \wedge (value(esp_n) < value(esp))$
      if $(psz \neq nil \wedge psz \neq psz_i) \vee (ins \neq nil \wedge ins \neq ins_i) \vee (aff \neq nil \wedge aff \neq aff_i)$ **then**
        goto *invalid*
      end if
      $psz \leftarrow size(pushedvalue(i))$
      $ins \leftarrow ins_i$
      $aff \leftarrow aff_i$
      $op \Leftarrow \{instruction(trace(k))\}$
    end for
    if $|pop| \neq 1$ **then**
      goto *invalid*
    end if
    $SO \Leftarrow \{(I, pop, psz, aff)\}$
  end if
  invalid:
end for

그림 2.10: 스택 연산 쌍 검색 알고리즘

스택 연산 쌍의 집합을 구한 뒤엔 효용 없는 스택 연산을 찾아야 한다. 그러기 위해서 각 스택 연산에 대해 값이 의미 있는지 혹은 위치가 의미있는지를 확인한 뒤, 둘 다 아니면 해당 스택 연산은 제거할 수 있다. 그림 2.11에서 효용 없는 스택 연산을 검색하는 알고리즘을 보이고 있다.

$SO_{useless} \leftarrow \{\}$
for all $so \in SO$ **do**
  $(push, pop, psz, aff) \leftarrow so$
  $valmean \leftarrow \{\}$ ▷ $push$의 값을 사용하는 인스턴스 액세스
  $posmean \leftarrow \{\}$ ▷ $push$의 값을 덮어쓰는 인스턴스 액세스
  for all $i \in instances(push)$ **do**
    $pushed \leftarrow pushedvalue(i)$
    $valmean \Leftarrow forwards(pushed)$
    $posmean \Leftarrow overwrites(pushed)$
  end for
  if $valmean = \{\}$ and $posmean = \{\}$ **then**
    $SO_{useless} \Leftarrow so$
  end if
end for

그림 2.11: 효용 없는 스택 연산 검색 알고리즘

그림 2.12는 스택 연산을 제거하는 알고리즘을 간략히 보이고 있다. 여기서 pop을 제거하거나 수정하는 부분은 IA32 아키텍쳐에 지나치게 의존적이어서 생략하였다.

for $so \in SO_{useless}$ **do**
  $(push, pop, psz, aff) \leftarrow so$
  if $\exists m \in aff$, m을 지원하지 않음 **then**
    goto $invalid$
  end if
  for all $m \in aff$ **do**
    m의 메모리 오퍼랜드의 displacement에서 psz를 뺀다
  end for
  $push$를 제거한다
  pop을 제거하거나 수정한다
  $invalid$:
end for

그림 2.12: 스택 연산 제거 알고리즘

2.5.7 인스트럭션 체인 약분

데이터 인코딩은 매우 흔한 난독화 기법이다. 데이터 인코딩이 적용된 프로그램은 사용할 데이터를 특정 알고리즘으로 인코딩하여 저장하였다가 프로그램 실행시 디코딩하여 사용한다. 예를 들어 가상화 난독화가 적용된 프로그램의 경우 바이트코드를 인코딩할 수 있다. 인코딩된 바이트코드는 가상 기계의 중심 블록에서 디코딩되어 핸들러를 찾을 때 사용될 수 있다. 이것이 기존의 가상화 난독화 해제 기술이 최신의 가상화 난독화 도구를 이용하여 난독화된 프로그램에 적용될 수 없는 주된 이유이기도 하다.

인스트럭션 체인 약분는 난독화된 프로그램에서 인코딩된 데이터와 디코딩 루틴을 찾고, 인코딩된 데이터 영역을 수정하면서 디코딩 루틴을 제거하여 분석을 보다 용이하게 한다. 인스트럭션 체인 약분를 이용해 디코딩 루틴이 제거되고 인코딩된 데이터가 없어진 프로그램은 보다 분석하기에 용이할 뿐만 아니라 기존에 데이터 인코딩으로 인해 적용할 수 없던 난독화 해제 기술을 이용할 수 있게 될 수도 있다.

디코딩 루틴은 두 가지의 특징이 있다. 첫째는 인코딩된 데이터는 반드시 사용되기 전에 읽혀서 디코딩된다는 사실이고, 둘째는 디코딩 과정에 있는 값은 디코딩이 끝나기 전까지 의미가 없으므로 디코딩 루틴 외에서는 그 값을 사용하지 않는다는 사실이다. 이러한 발견으로부터 약분 가능한 인스트럭션 체인(reducible instruction chain)을 정의하고 약분 가능한 인스트럭션 체인을 검색하고 제거하는 알고리즘을 설계하였다.

하나의 인스트럭션 체인은 인스트럭션 액세스인 원본(source)과 목적지(destination), 인스트럭션의 집합인 구성 인스트럭션, 그리고 새로운 인스트럭션인 교체 인스트럭션의 네 요소로 구성된다. 인스트럭션 체인의 원본은 메모리 영역에서 값을 읽어오는 인스트럭션 액세스이고 목적지는 값을 쓰는 인스트럭션 액세스이다. 원본에서 읽혀진 값은 구성 인스트럭션을 거쳐 최종적으로 목적지에 저장되게 된다. 따라서 하나의 인스트럭션 체인이 하나의 디코딩 루틴을 가리킨다고 봤을 때, 원본의 값은 인코딩된 값이고 목적지의 값은 디코딩이 끝난 상태의 값이다. 교체 인스트럭션은 인스트럭션 체인을 약분할 때 사용되는 무브 인스트럭션이다.

하나의 약분 가능 인스트럭션 체인이 발견되면 다음의 과정을 거쳐 제거할 수 있다. 이 과정을 통해 디코딩 루틴을 제거하고 프로그램이 디코딩된 값을 바로 읽어서 사용하도록 할 수 있다.

1. 체인의 목적지에서 사용된 최종적으로 디코딩된 값을 체인의 원본이 읽어온 메모리 위치에 저장한다.

2. 체인의 목적지가 속한 인스트럭션을 원본의 위치에서 목적지의 위치로 값을 복사하는 무브 인스트럭션인 교체 인스트럭션으로 교체한다.

3. 교체된 인스트럭션을 제외한 모든 구성 인스트럭션을 제거한다.

하나의 인스트럭션 체인은 길이가 4인 튜플 $(source, destination, members, replacing)$로 표현할 수 있으며 각각은 다음과 같이 정의된다.

- *source* : *InstructionAccess*는 인코딩된 데이터가 처음으로 읽혀지는 인스트럭션 액세스이다. *source*는 메모리로부터 읽는 액세스여야 한다.

- *destination* : *InstructionAccess*는 디코딩된 데이터가 최종적으로 저장되는 인스턴스 액세스이다. *destination*은 쓰는 액세스여야 한다. 이 때, 최종적으로 *source*의 위치로부터 *destination*의 위치로 값을 복사하는 인스트럭션이 추가되어야 하는데 인텔 아키텍쳐의 경우 메모리 사이의 연산은 허용되지 않으므로 *destination*은 레지스터에 쓰는 액세스여야 한다.

- *members* : {*Instruction*}는 인스트럭션 체인에 포함되는 구성 인스트럭션의 집합으로 원소들은 모두 내부 인스트럭션이어야 한다. 원본의 인스트럭션과 목적지의 인스트럭션도 *members*에 포함되어야 한다.

- *replacing* : *OPCODE* 은 인스트럭션 체인을 제거할 때 사용되는 교체 인스트럭션이다. 인스트럭션 체인의 제거 과정에서 목적지의 인스트럭션은 교체 인스트럭션으로 교체되어 메모리에 저장된 디코딩된 값을 필요한 자리로 바로 복사하도록 수정된다. 이 때 디코딩된 값이 저장되는 곳은 원본이 저장되어 있던 위치이고 디코딩된 값이 필요한 자리는 목적지가 저장되던 위치이므로 우리는 이미 오퍼랜드를 알고 있고, 그 동작도 두 위치 사이의 무브 인스트럭션임을 알고 있다. 하지만

인텔 아키텍쳐의 경우 무브 인스트럭션이 일반적인 무브(MOV), 부호 확장 무브(MOVSX), 0으로 확장 무브(MOVZX) 등이 존재하며 이들 opcode가 *replacing*에 저장된다.

또한 약분 가능 인스트럭션 체인이 제거될 수 있으려면 다음의 조건을 만족시켜야 한다

- $SRC = instanceaccesses(source)$, $DST = instanceaccesses(destination)$, 각각의 $I \in members$에 대해 $INST_I = instances(I)$라고 하자.
- 이 때 $\forall I \in members, |SRC| = |DST| = |INST_I|$를 만족해야 하고, 이 값을 N이라 하자.
- $0 \leq n < N$인 모든 n에 대해, $\forall I \in members, index(SRC[n]) \leq index(INST_I[n]) \leq index(DST[n])$가 만족되어야 한다.
- $0 \leq n < N$인 모든 n에 대해, $index(SRC[n]) < k < index(DST[n])$인 k에서 $trace(k)$가 *source*가 메모리 위치를 지정하기 위해 사용하는 레지스터나 *source*가 읽어들이는 메모리 영역의 값을 변경하지 않아야 한다.

경우에 따라서는 이러한 조건들을 만족시키는 약분 가능 인스트럭션 체인이 발견되었더라도 제거될 수 없는 경우가 있다. 예를 들어 명령어의 길이가 모두 다를 수 있는 CISC 명령 체계의 경우, 목적지의 인스트럭션의 길이가 교체 인스트럭션의 길이보다 짧으면, 길이가 긴 인스트럭션으로 짧은 인스트럭션을 대체하게 할 수 없으므로 인스트럭션 체인을 약분할 수 없게 된다.

인스트럭션 체인을 검색하기 위해서, 우선적으로 인스트럭션 체인의 원본이 될 수 있는 인스트럭션 액세스를 찾는다. 체인의 가능한 원본 후보를 찾기 위하여 인코딩된 데이터를 해제하는 루틴은 프로그램 전체에 단 하나뿐이라고 가정하고, 따라서 인코딩된 데이터는 그 데이터를 읽는 인스트럭션이 단 하나뿐이어야 한다고 가정하였다. 그림 2.13에서 이러한 가정에 따라 가능한 인스트럭션 체인의 원본 후보를 찾는 알고리즘을 보이고 있다.

그림 2.13에서 검색된 약분 가능 인스트럭션 체인의 원본 후보의 집합인 RC_{first}의 원소들로부터 인스트럭션 체인을 찾기 위한 확장(propagation)을 수행할 수 있다. 앞서 디코딩 루틴 내부에서 발생한 값은 디코딩 루틴이 끝나기 전까지는 외부 인스트럭션에서 의미가 없으므로 오직 디코딩 루틴에서만 그 값을 읽어서 사용할 것이라고 가정하였다. 이러한 가정에 따라 인스트럭션 체인의 원본 후보로 지목된 인스트럭션 액세스로부터 영향을 받는 인스트럭션의 개수가 오직 한 개인 경우 그 한 개의 인스트럭션을 인스트럭션 체인에 추가할 수 있다. 그림 2.14에서 이러한 원리로 약분 가능한 인스트럭션 체인을 확장하는 알고리즘을 보이고 있다. 여기서 사용된 함수들은 다음과 같이 정의된다.

- $source(I : Instruction) \rightarrow InstructionAccess$ 함수는 인스트럭션 I를 받아 I의 원본 오퍼랜드(source operand)에 해당하는 인스트럭션 액세스를 반환한다. 원본 오퍼랜드가 없는 경우에는 *nil*을 반환한다.
- $destination(I : Instruction) \rightarrow InstructionAccess$ 함수는 인스트럭션 I를 받아 I의 대상 오퍼랜드(destination operand)에 해당하는 인스트럭션 액세스를 반환한다. 대상 오퍼랜드가 없는 경우에는 *nil*을 반환한다.

$RC_{first} \leftarrow \{\}$
for all $I \in Z_i$ **do**
  for all $A \in accesses(I)$ **do**
    if $\{a \in instanceaccesses(A) | type(a) = read \wedge space(address(a)) = memory\} = \{\}$ **then**
      goto *break*
    end if
    for all $a \in instanceaccesses(A)$ **do**
      for all $j \in addresses(a)$ **do**
        if $accessat(j) \neq \{I\}$ **then**
          goto *break*
        end if
      end for
    end for
  end for
  $RC_{first} \Leftarrow \{I\}$
  break:
end for

그림 2.13: 약분 가능 인스트럭션 체인의 원본 후보 검색 알고리즘

$RC \leftarrow \{\}$
for all $f \in RC_{first}$ **do**
  if $source(f) \neq nil$ **then**
    $next \leftarrow f$
    $chn \leftarrow \{f\}$
    $rpl \leftarrow nil$
    loop
      if $opcode(next)$ is move opcode **then**
        $rpl \leftarrow opcode(next)$
      end if
      $chn \Leftarrow next$
      if $|influences(next)| \neq 1$ **then**
        break loop
      end if
      $next \leftarrow influences(next)$
    end loop
    if $rpl \neq nil$ **then**
      $RC \Leftarrow (source(f), destination(next), chn, rpl)$
    end if
  end if
end for

그림 2.14: 약분 가능 인스트럭션 체인 확장 알고리즘

그림 2.15는 인스트럭션 체인을 제거하기 위한 알고리즘을 보인다. 알고리즘 내의 **assert**의 조건이 만족되지 않는 인스트럭션 체인은 제거할 수 없는 것이므로 해당 인스트럭션 체인에 대한 약분 처리를 취소하고 그때까지 처리된 내용을 모두 복구해야 한다.

for all $rc \in RC$ **do**
  $(src, dst, chn, rep) \leftarrow rc$
  $instruction(dst)$를 "$rep\, dst\, src$"로 수정한다
  $srcinst \leftarrow instanceaccesses(src)$
  $dstinst \leftarrow instanceaccesses(dst)$
  assert $|srcinst| = |dstinst|$
  for $k = 0 \rightarrow |srcinst| - 1$ **do**
    $s \leftarrow srcinst[k]$
    $d \leftarrow dstinst[k]$
    assert $size(s) = size(d)$
    $a_s \leftarrow address(s)$
    $a_d \leftarrow address(d)$
    for $l = 0 \rightarrow size(s) - 1$ **do**
      assert $a_s + l \notin keys(Data)$
      $(a_s + l)$ 주소의 값을 $value(d, a_d + l)$로 수정한다
    end for
  end for
  $(chn - \{instruction(dst)\})$를 제거한다
end for

그림 2.15: 인스트럭션 체인 약분 알고리즘

2.5.8 수정된 실행 파일 생성

수정된 실행 파일 생성은 최적화 과정을 통해 얻어진 정보, 즉 제거할 수 있는 인스트럭션의 목록과 이들을 제거하기 위해 수정되어야 하는 내용들을 반영한 새로운 실행 파일을 생성하는 것이다. 이 과정은 대상 프로그램에서 제거될 수 있는 인스트럭션의 영역을 아무 일도 하지 않는 NOP으로 바꾸고 수정된 인스트럭션과 데이터를 반영하면 되므로 큰 문제가 없다.

다만, 제어 흐름 인스트럭션이 제거된 경우에 주의가 필요하다. 제어 흐름 인스트럭션이 불필요한 것으로 판정되어 제거되었더라도 원래의 대상 프로그램에서 그 인스트럭션으로 인해 그 바로 뒤에 위치한 인스트럭션이 아닌 다른 인스트럭션이 실행되었을 수 있다. 이러한 경우에는 해당 제어 흐름 인스트럭션을 제거하면 안 되고 그 다음에 실행해야 할 인스트럭션이 위치한 곳으로 무조건 점프하는 인스트럭션으로 대체하여야 한다.

제 3 장 구현

3.1 대상 시스템 환경

본 논문에서 소개된 분석 방법 및 최적화 방법은 대상 프로그램이 실행되는 시스템 환경에 구애받지 않는 일반적인 방법들이다. 하지만 32비트 인텔 아키텍쳐(Intel Architecture 32bit, IA32) 컴퓨터에서 구동되는 마이크로소프트 윈도우즈 운영체제에서 실행되도록 만들어진 실행 파일을 대상으로 분석 및 최적화 알고리즘들이 구현되었다. 이러한 시스템 환경은 가장 널리 사용되는 환경이면서 동시에 가장 많은 악성 코드가 활동하는 환경이기 때문에 이 환경을 선택하였다.

앞서 소개된 알고리즘들을 구현하기 위해 우선 분석 및 최적화의 대상이 되는 프로그램의 실행 트레이스를 추출하는 실행 트레이스 추출 도구를 개발하였고, 추출된 실행 트레이스를 분석하여 구조와 의미를 파악하고 최적화 정보를 찾아내는 분석 도구를 개발하여 Trudio라고 이름 붙였다.

이 장에서는 각 도구의 구현 방법에 대해 간략히 논의하고 Trudio에 대해 소개한다.

3.2 실행 트레이스 추출

분석 도구를 개발하기 전에 우선적으로 분석의 대상이 되는 실행 트레이스를 추출하는 도구를 개발하였다. 트레이스 추출 도구를 개발하기 위해 디버거를 이용하여 싱글 스텝 수행하여 트레이스를 얻어내는 방법, x86 에뮬레이터를 이용하는 방법, Pin을 이용하는 방법이 고려되었다.

처음에는 IDA Pro[9]나 OllyDbg[10]와 같은 디버거들을 트레이스 추출 도구로 사용하는 것을 고려하였으나 이들 디버거들은 트레이스 추출을 위해 싱글 스텝 예외를 사용하는데, 이 때문에 일부 난독화된 프로그램들의 트레이스를 제대로 추출하지 못 하는 문제가 있어 반려되었다.

다음에는 x86 에뮬레이터를 이용하는 방법이 고려되었다. QEMU[21]나 Bochs[22]와 같은 훌륭한 에뮬레이터가 있었지만 이들 도구는 전체 시스템을 에뮬레이션 하는 반면 본 연구에서는 전체 시스템이 아닌 응용 프로그램 하나만을 에뮬레이션 하는 도구가 필요해서 사용할 수 없었다. 또 에뮬레이터를 개발하기에는 시간과 인력이 부족하다는 점 때문에 에뮬레이터는 결국 제외되었다.

최종적으로 Pin[23]을 사용하는 것으로 결정되었다. Pin은 동적 바이너리 인스트루멘테이션(Dynamic Binary Instrumentation) 도구로서, 프로그램의 실행 중간에 개입하여 시스템 상황 모니터링을 할 수 있게 해주는 도구이다. Pin을 이용하려면 Pin에서 제공하는 API를 이용하여 분석 대상이 될 프로그램의 어떤 부분에서 어떤 식으로 개입해야 할 지를 설정해주도록 도구를 개발해야 하며, 이들 도구들을 Pin tool이라고 한다.

트레이스 추출 도구는 Pin tool의 형태로 Pin이 제공하는 API를 이용하여 개발되었다. 트레이스 추출 도구는 모든 실행되는 명령(트레이스 인스턴스)를 기록하고 레지스터나 메모리를 읽거나 쓰는 액세스를 모두 기록한다. 한편 개발된 트레이스 도구는 외부 루틴에서 실행되는 명령은 기록하지 않고 첫번째 명령만 외부 인스트럭션으로 기록하여 트레이스의 크기를 줄였다. 그러나 외부 인스트럭션의 인스턴스 액세스는 분석을 위해 모두 기록되도록 하였다.

트레이스 추출 도구는 오직 대상 프로그램을 실행하여 그 트레이스만을 추출하며 분석 과정과는 완전히 분리되도록 설계되었다. 따라서 새로운 트레이스 추출 도구를 개발하여 분석 도구에서 사용할 수 있는 형태로 추출된 실행 트레이스를 저장한다면 분석 과정에 그대로 사용할 수 있다.

트레이스 추출 도구는 [24]에서 다운로드 받을 수 있다.

3.3 분석 도구

본 논문에서 소개된 분석 알고리즘들을 구현하고 하나의 분석 도구로 통합하여 Trudio라는 이름의 분석 도구를 개발하였다. Trudio라는 이름은 트레이스(trace)와 작업장을 의미하는 스튜디오(studio)라는 두 단어의 합성어로, 난독화를 자동으로 해제하는 도구가 아니라 분석자가 트레이스를 보다 쉽게 이해하고 분석에 사용할 수 있도록 돕는 도구라는 특징을 보여준다.

Trudio는 자바 언어로 개발되었고 Swing GUI toolkit 및 디스어셈블러 distorm3[25], 어셈블러 Netwide Assembler[26]가 사용되었다. Trudio는 [27]에서 다운로드 받을 수 있다.

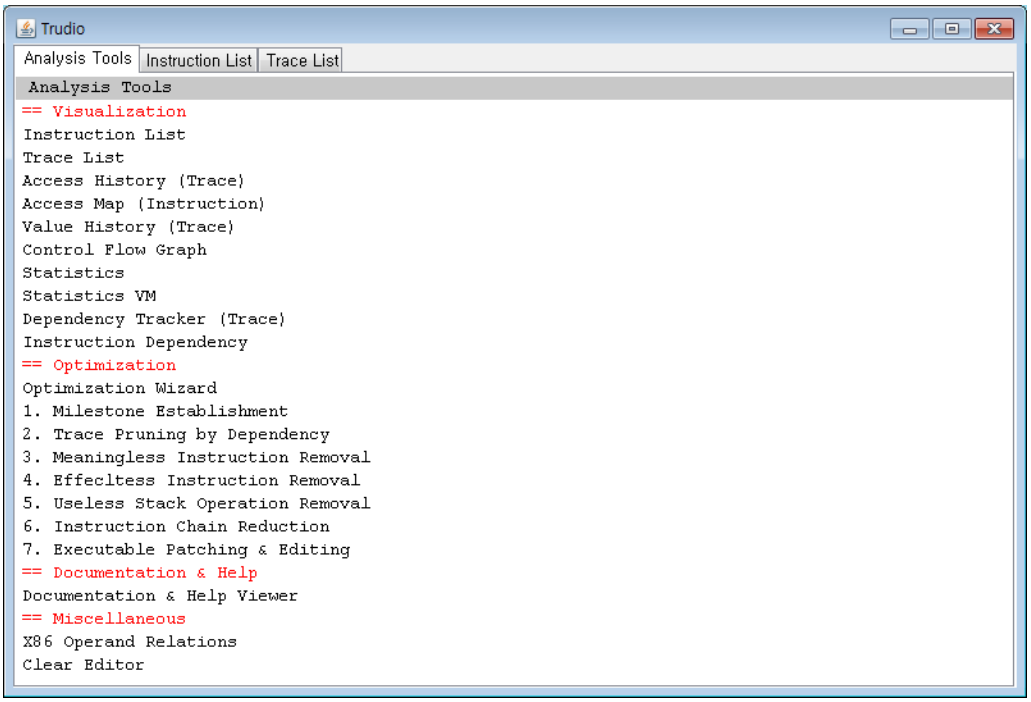


그림 3.1: Trudio의 첫 화면

Trudio를 추출된 트레이스와 대상 실행 파일을 설정하여 실행하면 첫 화면으로 그림 3.1과 같은 메뉴 화면이 나타난다. 이 화면에는 Trudio에 구현되어 포함되어 있는 분석 도구의 목록이 나타난다. 메뉴 목록의 앞부분에는 대상 프로그램의 구조와 의미를 파악하기 위한 시각화 도구들이 표시되고 있고 뒷부분에는 최적화 도구들이 표시되고 있다. 각 메뉴를 클릭하면 상단에 새로운 탭이 추가되며 해당 도구를 사용할 수 있게 된다. 이어서 각 도구들에 대해 간략히 소개한다.

3.3.1 트레이스의 기본 정보

인스트럭션 목록

그림 3.2에서 보이고 있는 인스트럭션 목록(Instruction List) 도구는 읽어들인 트레이스에 속한 인스트럭션들의 목록을 나타낸다.

도구의 좌측에는 트레이스에 속한 각 인스트럭션에 대해 인스트럭션의 주소, 인스트럭션이 속한 블록 번호, 디스어셈블된 인스트럭션 내용이 나타난다. 그 중 하나를 선택하면 도구 우측에 해당 인스트럭션의 인스턴스 개수, 해당 인스트럭션의 오퍼랜드 개수 및 목록과 더불어 인스트럭션 액세스의 목록이 나타난다. 각 인스트럭션 액세스 A에 대해서 액세스의 종류 $type(A)$, 읽히거나 쓰여진 주소 공간 $space(address(A))$, 액세스되는 주소가 레지스터인 경우 레지스터의 이름, 액세스가 오퍼랜드인 경우 몇 번째 오퍼랜드인지, 그리고 액세스가 읽거나 쓴 길이 $size(A)$가 표시된다.

트레이스 목록

그림 3.3에서 보이고 있는 트레이스 목록(Trace List) 도구는 읽어들인 트레이스에 속한 인스턴스들의 목록을 나타낸다. 따라서 이 도구에서 읽어들인 트레이스의 가공되지 않은 모든 정보를 볼 수 있다.

도구의 좌측에는 트레이스에 속한 각 인스턴스에 대해 인스턴스의 번호, 즉 몇 번째로 실행된 트레이스 인스턴스인지가 가장 좌측에 표시되고 해당 인스턴스가 속한 인스트럭션의 정보가 이어서 표시된다. 그 중 하나를 선택하면 도구 우측에 해당 인스턴스의 액세스 정보가 나타난다. 각 인스턴스 액세스 a에 대해, 액세스의 종류 $type(a)$, 읽히거나 쓰여진 주소 공간 $space(address(a))$와 실제 주소 $location(address(a))$, 액세스가 오퍼랜드인 경우 몇 번째 오퍼랜드인지, 그리고 읽혀지거나 쓰여진 값이 16진수로 인코딩되어 나타난다. 액세스되는 주소가 레지스터인 경우 주소가 숫자가 아닌 레지스터 이름으로 표시되고, 16진수로 인코딩된 값의 길이로부터 액세스된 길이를 알아낼 수 있으므로 길이는 별도로 표시하지 않는다.

3.3.2 프로그램의 구조 파악

제어 흐름 그래프

그림 3.4에서 보이고 있는 제어 흐름 그래프(Control Flow Graph) 도구는 2.3.1에서 정의한 동적 제어 흐름 그래프를 보인다.

기본적으로 표시되는 그래프의 형태 외에도 리스트의 형태로 보거나, 블록이 호출된 목록을 보거나, 한 블록에 속한 인스트럭션의 목록을 보거나, Graphviz[28]에서 사용되는 DOT 언어의 형태로도 볼 수 있다.

가상 기계 통계

그림 3.5에서 보이고 있는 가상 기계 통계(Statistics - VM) 도구는 2.3.2에서 기술한 가상화를 이용한 난독화에 사용되는 가상 기계의 일반적인 특징을 이용해 가상 기계를 추측하고 더 나아가 바이트코드의 종류를 찾아내는 데 사용될 수 있는 도구이다.

도구 상단 좌측에는 앞서 서술한 제어 흐름 그래프 도구가 나타나고, 상단 우측에는 각 블록이 실행된 횟수가 횟수가 많은 것부터 나타난다. 도구 하단에는 특정 블록이 가상 기계인지를 확인하고 바이트코드의 종류를 찾아낼 수 있는 기능이 구현되어 있다. 도구 하단 좌측에서 특정 블록을 선택하면 (기본적으로 가장 실행된 횟수가 많은 블록이 선택됨) 해당 블록의 실행이 종료된 직후에 실행된 적이 있는 블록의 목록과 선택된 블록에 속한 인스트럭션의 목록이 나타난다. 블록에 속한 인스트럭션들 중 하나를 선택하면 해당 인스트럭션에 속한 인스트럭션 액세스의 목록이 그 우측에 나타나고, 액세스들

중 하나를 선택하면 해당 액세스의 값에 따라 다음에 실행된 블록이 어떤 것이었는 지를 표시한다. 만약 여기서 해당 액세스의 값에 따라 그 다음에 실행된 블록이 하나로 한정된다면 가상 기계의 바이트코드로 사용되었을 가능성이 높다.

액세스 맵

그림 3.6에서 보이고 있는 액세스 맵(Access Map) 도구는 2.3.3에서 기술한 액세스 맵 분석을 구현한 것이다.

도구 좌측에는 인스트럭션 목록이 나타나고, 그 중 한 인스트럭션 I를 선택하면 도구 중앙의 메모리 맵에 해당 인스트럭션이 읽거나 쓰는 메모리 영역, 즉 $access(I)$가 초록색 배경으로 강조되어 표시된다. 반대로 중앙의 메모리 맵에서 메모리 주소 a를 선택하면 도구 우측에 해당 메모리 영역을 읽거나 쓴 적이 있는 인스트럭션의 목록, 즉 $accessat(a)$가 나타나고 도구 좌측의 인스트럭션 목록에도 해당 인스트럭션들이 강조되어 표시된다.

3.3.3 프로그램의 의미 파악

값 히스토리

그림 3.7에서 보이고 있는 값 히스토리(Value History) 도구는 2.4.1에서 기술한 값 히스토리를 구현한 도구이다.

도구 좌측 상단에는 트레이스 인스턴스 목록이 나타나고 그 중 하나를 선택하면 도구 좌측 하단에 해당 인스턴스의 액세스 목록이 나타난다. 도구의 우측에는 선택된 인스턴스가 실행된 직후에 메모리 영역의 값, 레지스터의 값, 스택의 값 등 시스템 상태가 나타난다.

의존성 추적기 및 수식 트리

그림 3.8에서 보이는 의존성 추적기(Dependency Tracker) 도구는 2.4.2에서 소개한 의존성 추적기를 구현한 것이다.

도구 좌측의 트레이스 인스턴스 목록에서 하나의 인스턴스 i를 선택하면 우측에 해당 인스턴스의 액세스 목록과 각 인스턴스 액세스 $a \in accesses(i)$에 대해 설정에 따라 $depends(a)$나 $forwards(a)$가 표시되고, 그 중 하나를 선택하면 의존성을 반복적으로 추적하여 값이 계산되는 과정을 살펴볼 수 있다.

의존성 추적 도구에는 2.4.3에서 소개한 수식 트리도 함께 구현되어 있으며 그림 3.9에 나타나 있다. 수식 트리 도구의 좌측 상단에서 관심 있는 인스트럭션의 집합 N에 포함될 opcode의 집합 P를 설정할 수 있고, 이 설정에 따라 도구 우측에 수식 트리가 그래프 형태로 시각화되어 나타난다. 도구 좌측 하단에는 같은 그래프를 Graphviz에서 그릴 수 있도록 DOT 언어의 형태로도 표시한다.

그림 3.10은 인스트럭션 의존성 추적(Instruction Dependency) 도구를 보이고 있다. 이 도구는 2.4.2에서 정의한 인스트럭션 액세스에 대한 $depends$ 함수와 $forwards$ 및 $overwrites$ 함수를 구현한 것이다.

도구 좌측의 인스트럭션 목록 중 하나를 선택하고 우측 상단에서 forwards, backwards, overwrites 중 하나를 선택하면 선택된 인스트럭션 I에 속한 각 액세스에 대해, 즉 $A \in accesses(I)$에 대해, $forwards(A)$, $backwards(A)$, 혹은 $overwrites(A)$를 보인다.

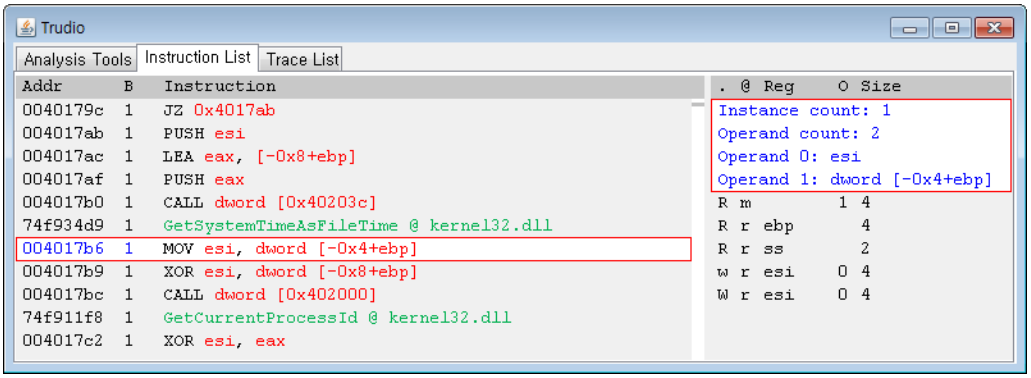


그림 3.2: 인스트럭션 목록 도구

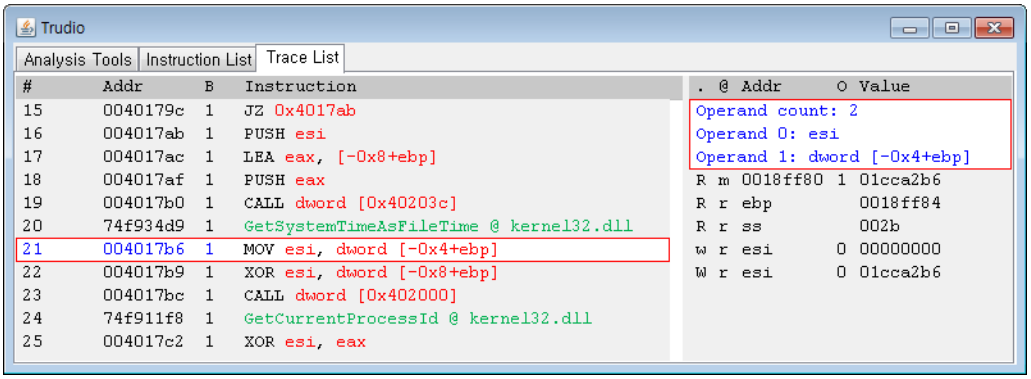


그림 3.3: 트레이스 목록 도구

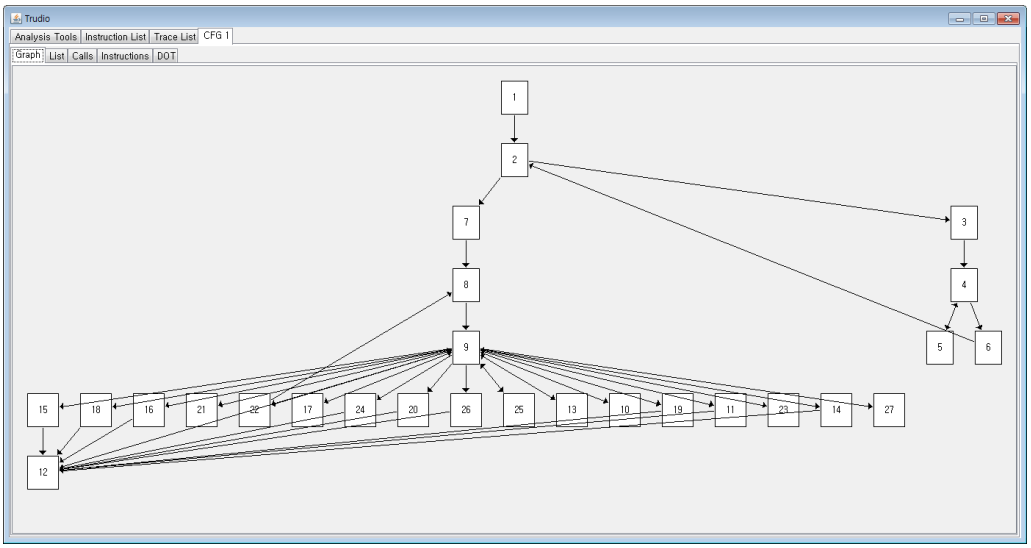


그림 3.4: 제어 흐름 그래프 도구

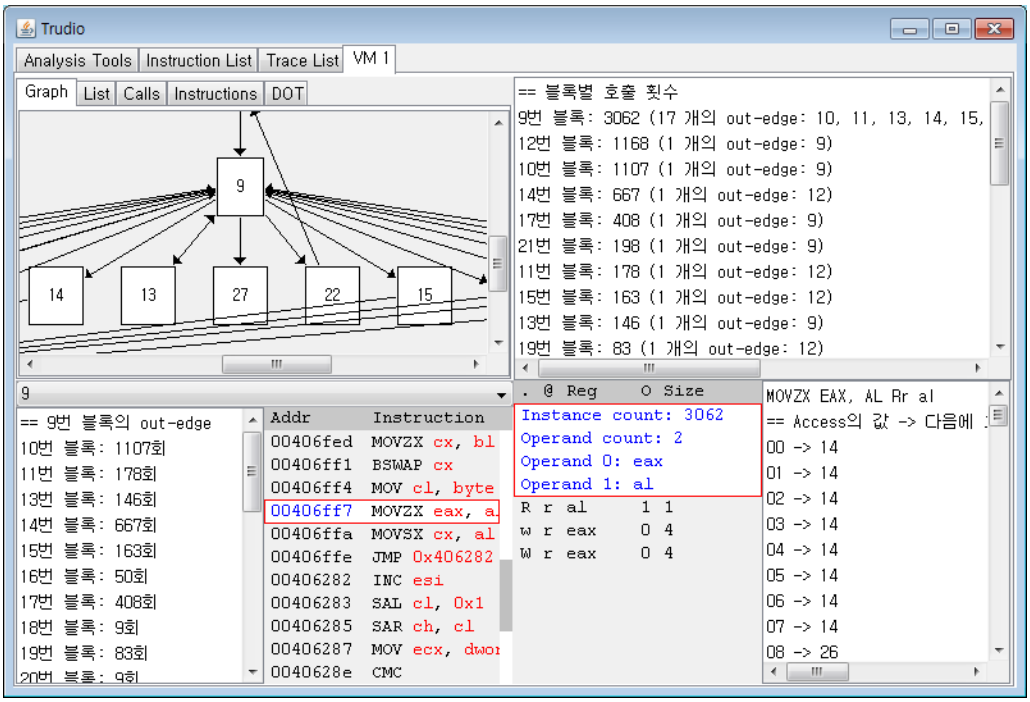


그림 3.5: 가상 기계 통계 도구

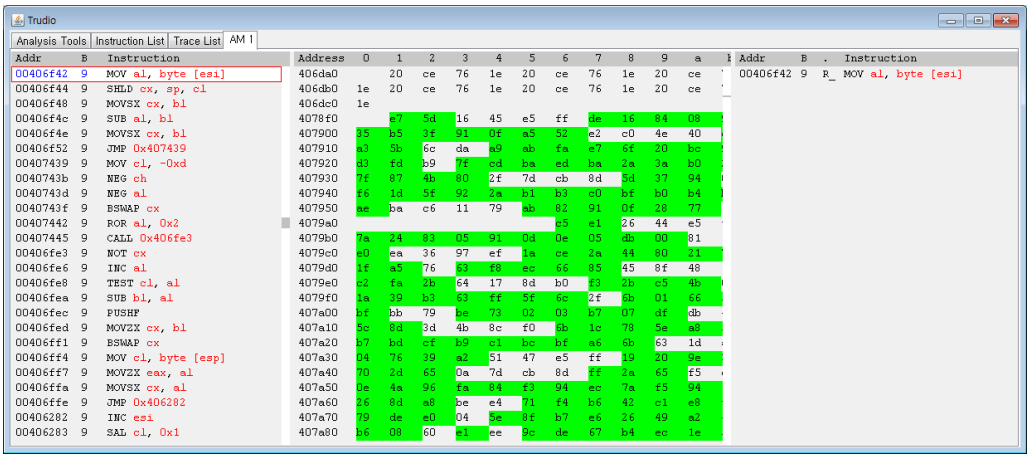


그림 3.6: 액세스 맵 도구

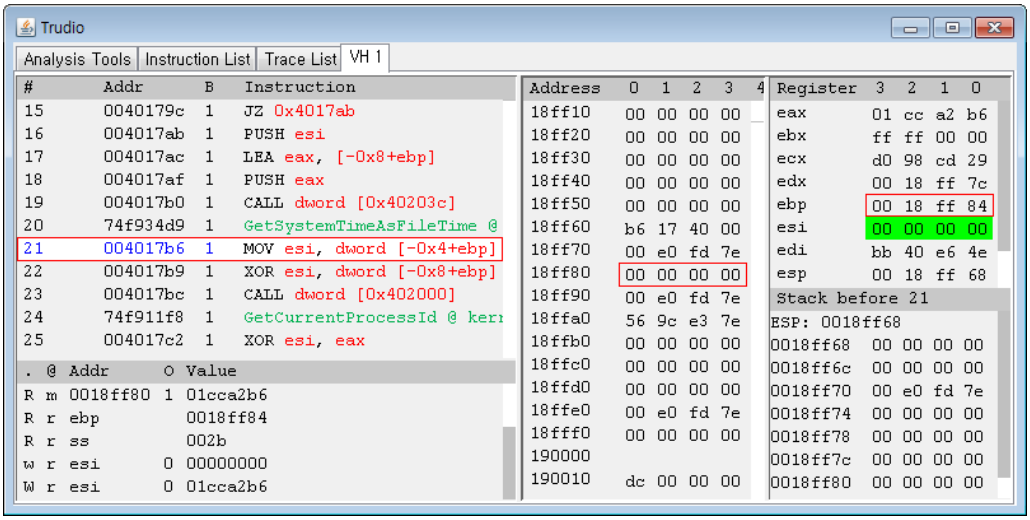


그림 3.7: 값 히스토리 도구

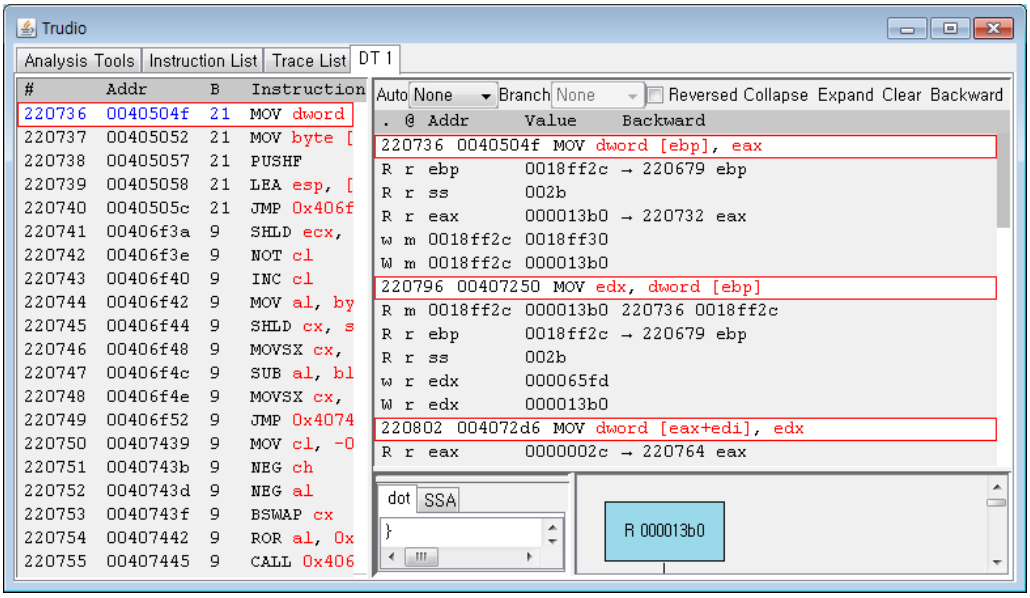


그림 3.8: 의존성 추적 도구

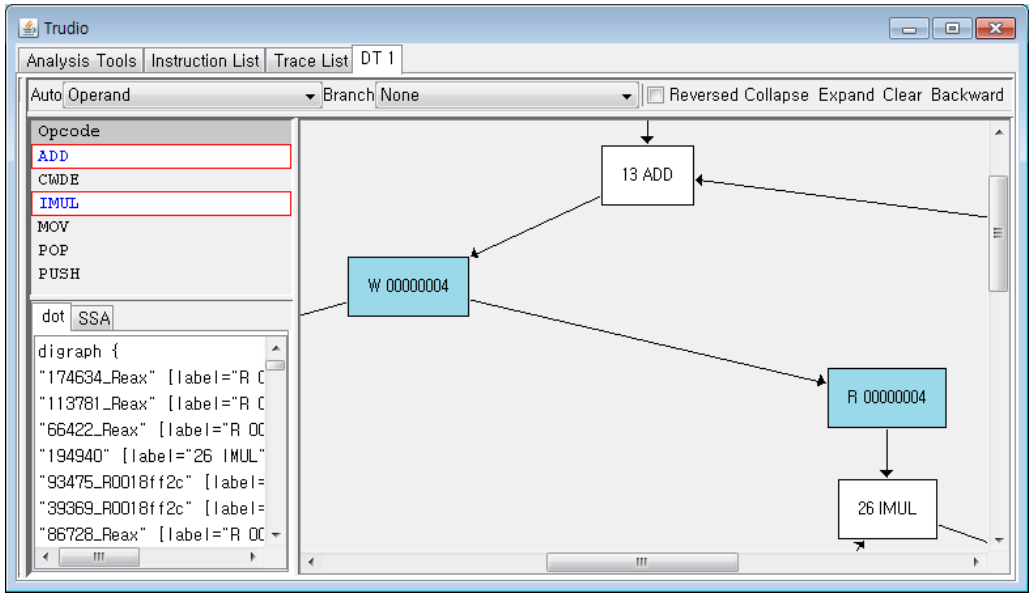


그림 3.9: 수식 트리

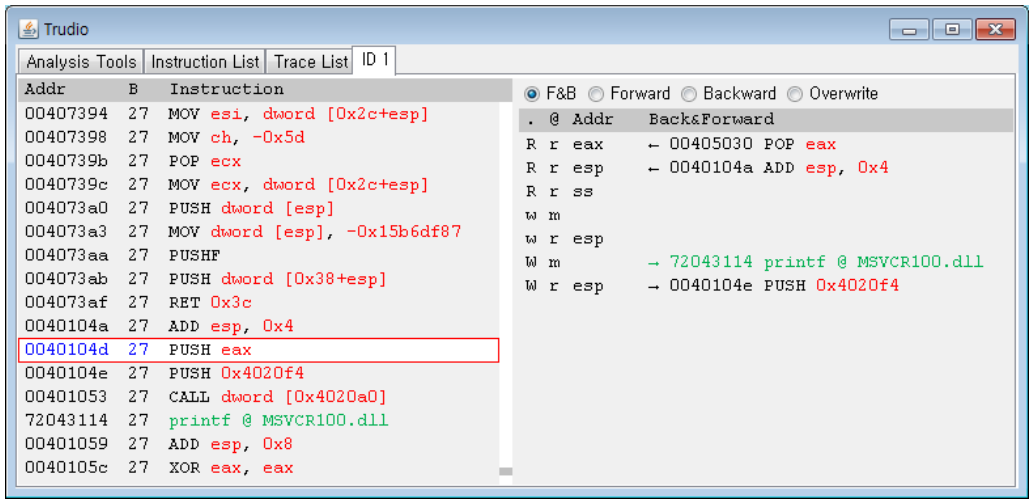


그림 3.10: 인스트럭션 의존성 추적 도구

3.3.4 프로그램의 최적화

앞서 2.5에서 소개한 바와 같이 최적화는 총 7단계로 구성되어 있다. 각 단계는 사실상 완전히 자동화되어 있어서 Trudio는 최적화 마법사(Optimization Wizard) 도구를 제공하여 모든 단계를 한번에 적용할 수 있지만 이에 대한 설명은 생략하고 각 단계의 구현만 간략하게 소개한다.

마일스톤 설정

그림 3.11이 보이고 있는 마일스톤 설정 도구는 2.5.2에서 소개한 마일스톤 선택 기준에 맞추어 검색된 마일스톤 인스트럭션 집합 M을 표시하고 마일스톤으로 설정하는 기능을 한다. 도구 중앙에 나타난 인스트럭션 목록에서 강조되어 나타나는 인스트럭션이 마일스톤으로 선택될 인스트럭션이다.

의존성 분석을 이용한 가지치기

그림 3.12는 2.5.3에서 소개한 의존성 분석을 이용한 가지치기를 구현한 도구의 실행 화면을 보이고 있다.

도구의 좌측에는 전체 트레이스 인스턴스 목록이 표시되고 있고, 우측에는 그 중 관계 있는 인스트럭션의 인스턴스가 나타난다. 따라서 우측의 목록에 나타나지 않는 인스트럭션은 제거된다.

의미 없는 인스트럭션 제거

그림 3.13은 2.5.4에서 소개된 의미 없는 인스트럭션 제거 최적화 단계를 구현한 도구를 보이고 있다. 앞서 정의된 의미 없는 제어 흐름 인스트럭션의 집합 $F_{control}$과 상태를 바꾼 적이 없는 인스트럭션의 집합 F_{effect} 중 화면에 표시할 집합을 도구 상단의 콤보 박스로 선택할 수 있고, 선택된 집합에 포함된 인스트럭션이 도구 중앙의 인스트럭션 목록에서 강조되어 나타난다.

효과 없는 인스트럭션 제거

그림 3.14는 2.5.5에서 소개한 효과 없는 인스트럭션 제거 단계를 구현한 도구의 실행 화면이다. 도구의 좌측에 나타난 인스트럭션 목록에 강조되어 나타난 인스트럭션이 현재 효과가 없는 것으로 나타난 인스트럭션이다.

효용 없는 스택 연산 제거

그림 3.15는 2.5.6에서 소개한 효용 없는 스택 연산 제거 단계를 구현한 도구의 실행 화면을 보인다. 도구 중앙의 목록은 그림 2.10에서 정의한 스택 연산 쌍 검색 알고리즘에 의해 검색된 스택 연산 쌍, 즉 프로그램이 실행되면서 수행된 PUSH와 POP의 쌍의 목록을 볼 수 있다. 목록의 스택 연산 하나를 선택하면 해당 스택 연산에 의해 영향을 받는 인스트럭션 목록을 볼 수 있다.

각 스택 연산이 값이 의미 있는 경우에는 가장 왼쪽에 V, 위치가 의미 있는 경우 P, 둘 다 의미가 없는 경우 회색으로 R이 표시된다. R로 표시된 스택 연산 중 스택 연산을 제거하기 위한 모든 조건을 만족해서 제거할 수 있는 스택 연산 쌍에는 Apply라는 이름의 버튼이 표시되고, 제거할 수 없는 스택 연산에는 해당 스택 연산을 제거할 수 없는 이유가 표시된다. 스택 연산을 표시할 수 없는 이유는 표 3.1에 정리되어 있다.

인스트럭션 체인 약분

그림 3.16은 2.5.7에서 소개된 인스트럭션 체인 약분 단계를 구현한 도구의 실행 화면이다. 도구 좌측에는 제거할 수 있는 인스트럭션 체인의 목록이 나타난다. 각 인스트럭션 체인에 대해서는 앞서

문자	의미
P	제거할 수 없는 종류의 PUSH 인스트럭션이다.
p	제거할 수 없는 종류의 POP 인스트럭션이다.
C	PUSH 인스트럭션의 인스턴스와 POP 인스트럭션의 인스턴스가 일관되지 않는다. 즉, PUSH와 POP의 인스턴스의 개수가 다르거나 PUSH의 인스턴스보다 대응되는 POP의 인스턴스가 먼저 나오는 경우가 있었다.
M	PUSH와 POP 인스트럭션 사이의 수정되어야 할 인스턴스가 일관되지 않는다. 즉, PUSH-POP 인스턴스 사이에서 수정되어야 할 인스턴스의 개수가 다르거나 수정되어야 할 인스턴스가 각 인스턴스에 대해 일치하지 않는 경우가 있었다.
S	PUSH와 POP 사이에 수정해야 하는 인스트럭션 중 수정할 수 없는 것이 포함되어 있다. 수정해야 하는 인스트럭션은 모두 내부 인스트럭션이고, 지원되는 op코드를 갖고, ESP 이외의 레지스터를 이용해 스택 메모리 영역에 접근하지 않아야 하는데, 이 조건을 만족하지 않는 인스트럭션을 수정해야 하는 경우가 있다.

표 3.1: 스택 연산을 제거할 수 없는 경우

정의된 source, destination, replacing instruction과 member instruction가 표시된다. 각 인스트럭션 체인이 인스트럭션 체인을 제거하기 위한 조건을 모두 만족하지 못하는 경우에는 그 배경을 흰 색이 아닌 다른 색으로 표시하여 나타나며 최적화에 적용되지 않는다.

실행 파일 패치

그림 3.17은 2.5.8에서 소개된 최적화의 마지막 단계인 실행 파일 패치 단계를 구현한 도구의 실행 화면을 보인다. 도구의 좌측에는 인스트럭션 목록이 나타나고 우측에는 생성될 실행 파일의 내용이 원본 실행 파일에서 변경될 부분이 초록색 배경으로 강조되어 나타난다. 최적화 과정에서 변경된 내용 외에 사용자가 직접 인스트럭션을 변경하거나 제거하는 것도 가능하다. Trudio는 IA32 프로그램을 대상으로 하므로 제거된 인스트럭션들은 IA32의 NOP 인스트럭션의 op코드인 90으로 대치된다.

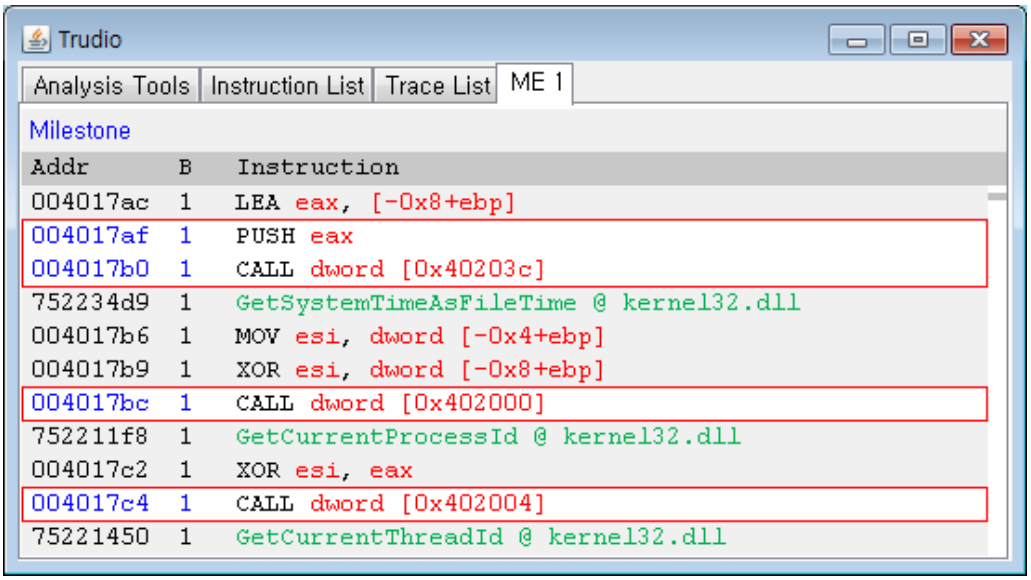


그림 3.11: 마일스톤 설정 도구

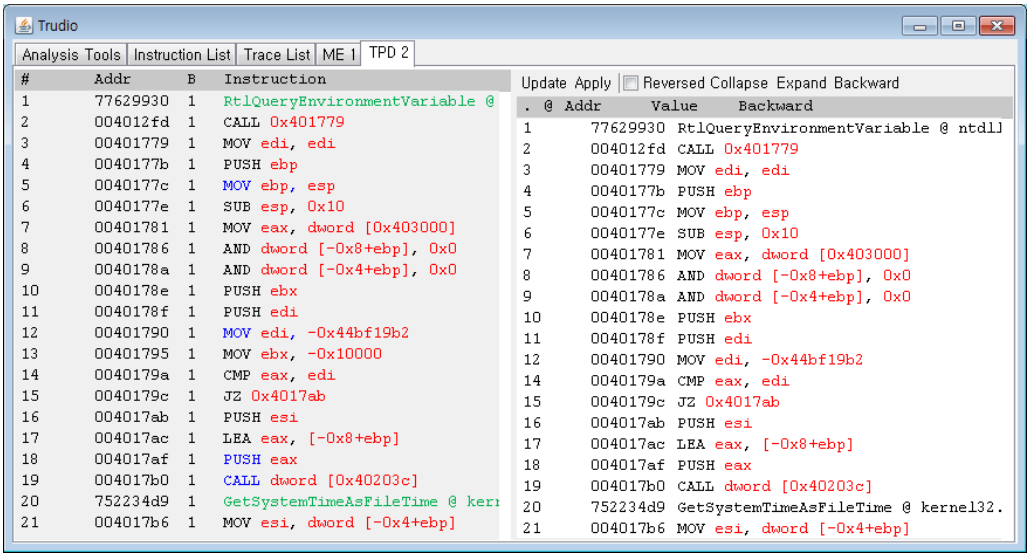


그림 3.12: 의존성 분석을 이용한 가지치기 도구

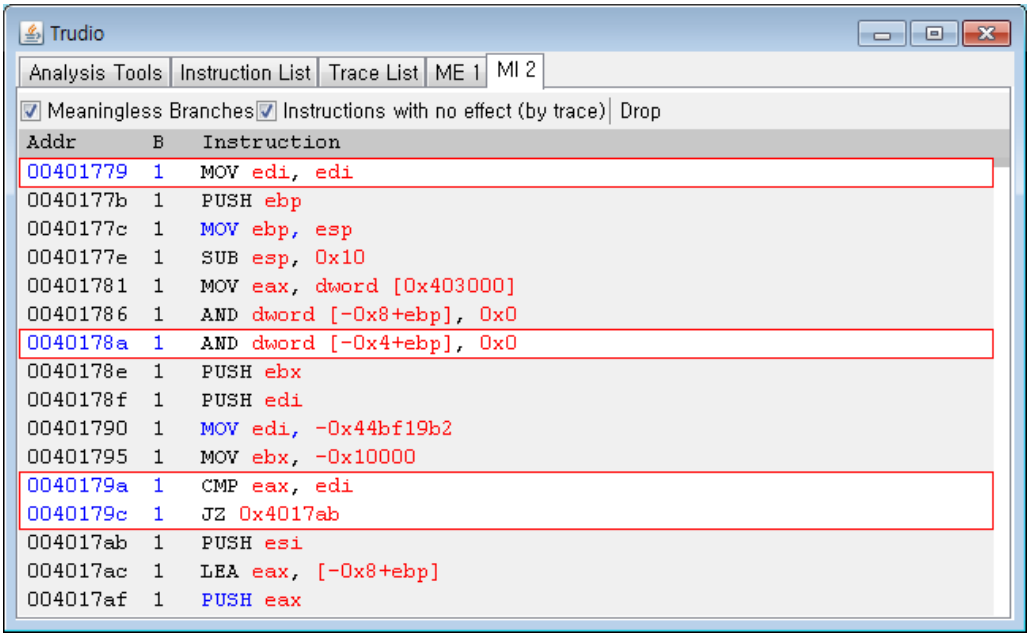


그림 3.13: 의미 없는 인스트럭션 제거 도구

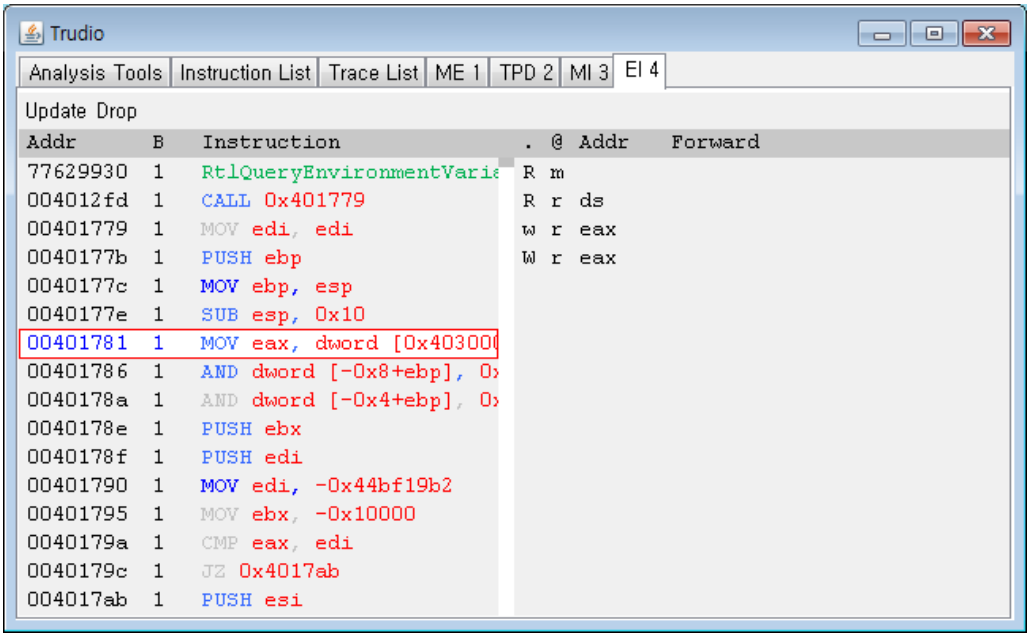


그림 3.14: 효과 없는 인스트럭션 제거 도구

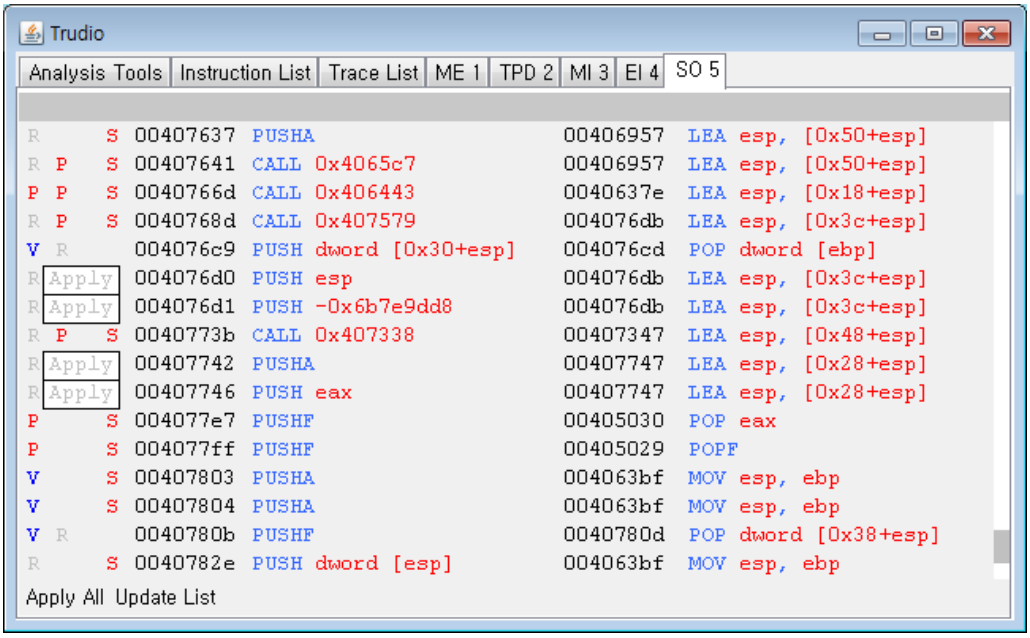


그림 3.15: 효용 없는 스택 연산 제거 도구

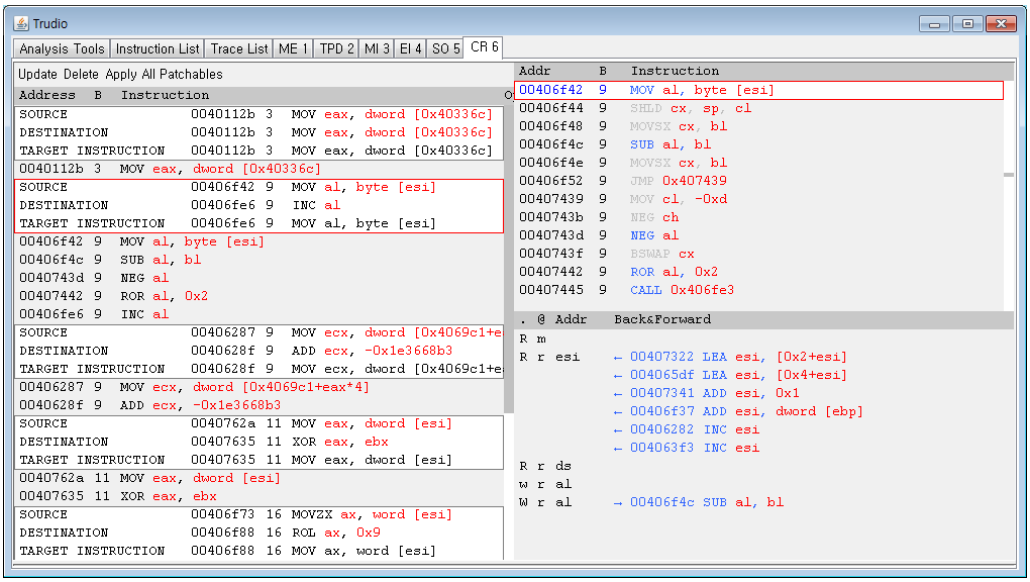


그림 3.16: 인스트럭션 체인 약분 도구

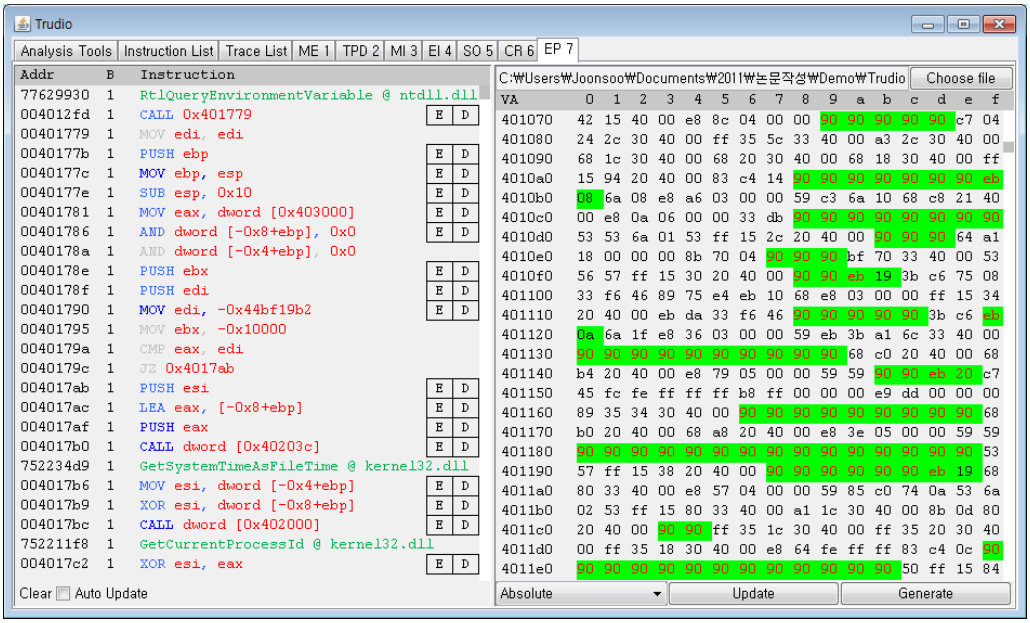


그림 3.17: 실행 파일 패치 도구

제 4 장 실험 및 결과

개발된 분석 방법들의 효과성을 검증하기 위해 일련의 실험을 설계하였다. 그 전에 실제 분석 사례를 소개하여 도구의 효용성을 보이고 실험 방법을 설명한다.

4.1 분석 사례

개발된 분석 방법의 효율성을 보이기 위해 재귀적인 방법으로 피보나치 수열의 6번째 값을 계산하는 간단한 프로그램을 C 언어로 작성하였다. 이 프로그램을 컴파일하여 이 중 실제 계산을 수행하는 재귀 함수를 VMProtect를 이용하여 난독화한 뒤 원본 프로그램과 분석 결과를 비교하였다. 분석 대상이 된 프로그램의 소스 코드는 그림 4.1에서 보이고 있다.

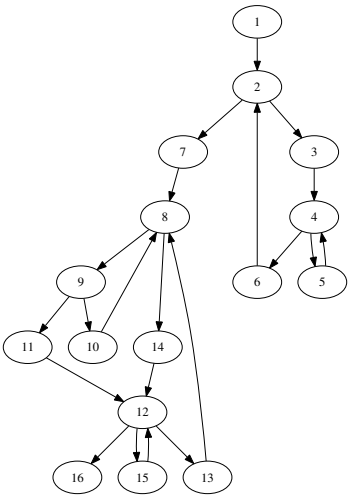
```
#include <stdio.h>

int recfibo(int n) {
        if (n == 0)
                return 0;
        if (n == 1)
                return 1;
        return recfibo(n-1) + recfibo(n-2);
}

int main() {
        printf("%d\n", recfibo(6));
        return 0;
}
```

그림 4.1: 재귀형 피보나치 프로그램의 소스 코드

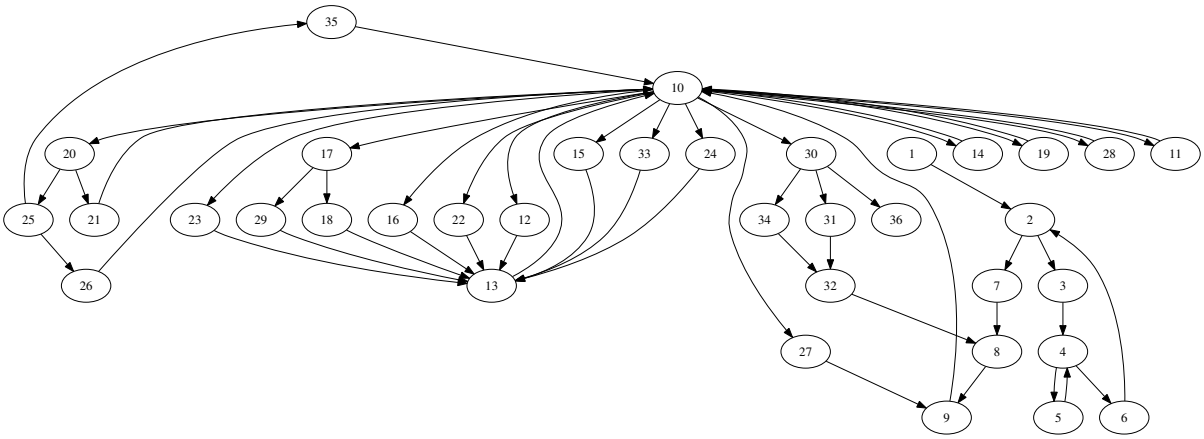
우선, 원본 프로그램과 난독화된 프로그램의 제어 흐름 그래프를 비교해 보자. 그림 4.2a는 원본 프로그램의 제어 흐름 그래프, 그림 4.2b는 난독화된 프로그램의 제어 흐름 그래프를 보인다.

원본 프로그램의 8번 블록부터 실제 계산을 수행하는 함수의 영역이다. 8번 블록은 재귀 함수로 전달된 변수의 값이 0이면 14번 블록으로 진행하여 0을 반환하고, 0이 아니면 9번 블록으로 진행한다. 9번 블록은 변수의 값이 1이면 11번 블록으로 진행하여 1을 반환하고, 1이 아니면 10번 블록으로 진행한다. 10번 블록에서는 변수의 값에서 1을 빼서 다시 8번 블록을 호출한다. 원본 소스 코드와 일치하는 간단한 구조의 프로그램이다.

반면 난독화된 프로그램은 그 구조가 원본 프로그램과 완전히 다른 것을 제어 흐름 그래프에서 볼 수 있다. 여기서 가상 기계의 중심 블록은 이들이 다른 블록에 비해 여러 차례 실행되며, 제어 흐름 그래프 상에서 많은 수의 블록과 연결되어 있다는 특징을 이용하여 찾아낼 수 있다. 통계를 통해 10번 블록이 10769회 실행되어 가장 많이 실행되었고, 두번째로 많이 실행된 13번 블록의 4271회에 비해 훨씬 많았다. 다른 블록과의 연결성을 보면 10번 블록으로부터 나가는 간선은 총 15개로 두번째로 나가는



(a) 원본 프로그램



(b) 난독화된 프로그램

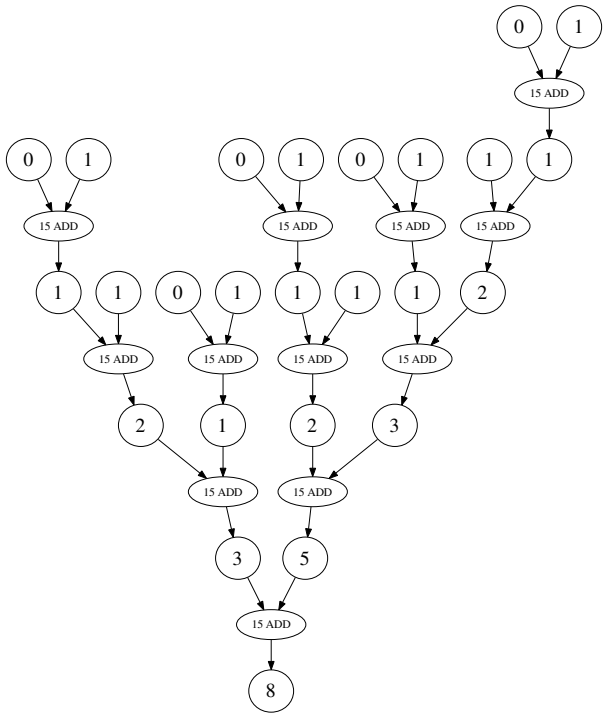
그림 4.2: 재귀형 피보나치 프로그램의 제어 흐름 그래프

간선이 많은 30번 블록의 3개에 비해 훨씬 많았다. 이러한 결과로부터 실행 횟수와 나가는 간선의 수가 가장 많은 10번 블록이 중심 블록으로 추측할 수 있었다. 또한 10번 블록의 내용을 살펴보면 ESI 레지스터에 저장된 메모리 위치로부터 한 바이트를 읽어와 이를 디코딩하여 핸들러를 호출하는 전형적인 구조를 취하고 있음을 볼 수 있다.

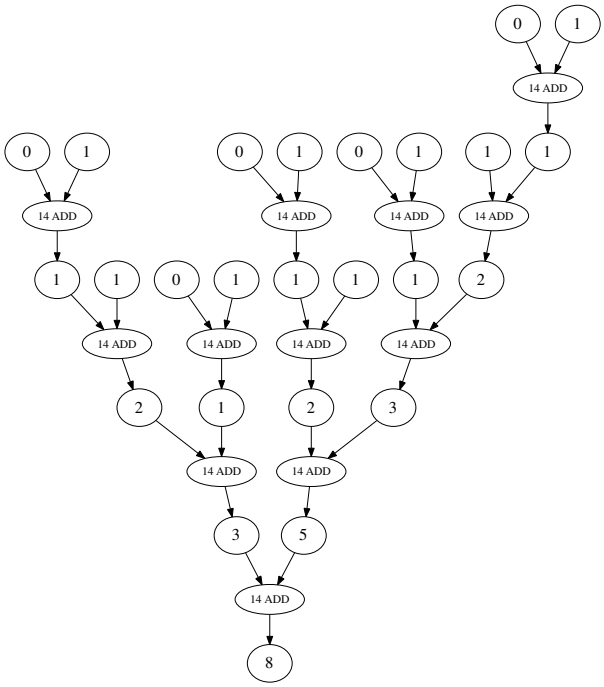
한편 정적으로 생성된 난독화된 프로그램의 제어 흐름 그래프에서는 중심 블록인 10번 블록 이후를 정확히 추적하지 못 하였다.

다음으로 난독화된 프로그램의 의미를 분석하였다. 대상 프로그램은 하나의 숫자를 계산하여 화면에 출력하는 간단한 구조로 되어 있기 때문에, 화면에 출력된 값이 어떤 과정을 거쳐서 계산된 값인지를 알아낼 수 있다면 프로그램의 알고리즘을 이해했다고 볼 수 있다. 화면에 숫자를 출력하는 printf 함수는 난독화된 프로그램에서도 공통적으로 나타났으며, printf 함수에 계산된 숫자를 인자로 전달하는 코드도 같은 형태로 나타났다. printf 함수에 출력을 위해 인자로 전달되는 숫자로부터 시작하여 연관된 부분 의존성 그래프를 추출하였다. 원본 프로그램에서 추출된 부분 의존성 그래프에는 ADD, MOV, POP, PUSH, XOR 등의 opcode를 가진 인스트럭션의 인스턴스가 포함되었고, 난독화된 프로그램에서 추출된 부분 의존성 그래프에는 ADD, AND, CWDE, MOV, NOT, PUSH 등의 opcode를 가진 인스트럭션의 인스턴스가 포함되었다. 이 중 MOV, POP, PUSH는 데이터를 이동시키는 것 외에 역할을 하지 않고, XOR은 변수를 0과 비교하기 위하여 사용된 것이므로 그 중 ADD만을 취하여 원본 프로그램의 수식

트리를 그렸다. 난독화된 프로그램에서도 마찬가지로 ADD만을 취하여 수식 트리를 그렸다. 이렇게 그려진 수식 트리는 그림 4.3에 나타난다. 그림 4.3a는 원본 프로그램의 수식 트리를 보이고 있고, 그림 4.3b는 난독화된 프로그램의 수식 트리를 보이고 있다. 이 둘이 완전히 일치하는 것을 볼 수 있으며, 그려진 수식 트리로부터 프로그램이 피보나치 수를 계산하고 있음을 확인할 수 있었다.



(a) 원본 프로그램



(b) 난독화된 프로그램

그림 4.3: 재귀형 피보나치 프로그램의 수식 트리

마지막으로 난독화된 프로그램을 최적화하여 그 효율성을 확인하였다. 앞서 소개된 최적화 7단계 중 최적화를 반영하는 실행 파일을 생성하는 마지막 단계를 제외한 1단계부터 6단계까지를 자동으로 수행하였다. 그 결과 가상 기계 중심 블록인 10번 블록에 속해 있던 36개의 인스트럭션들 중 단 14개만이 남는 것을 볼 수 있었다. 이 중에는 불러온 바이트코드를 디코딩하는 부분과 디코드한 바이트코드를 실제 핸들러 위치에 매핑하는 과정의 디코딩 루틴이 모두 포함되어 메인 블록에 포함된 모든 디코딩 루틴을 제거한 것을 확인할 수 있었다.

원래의 난독화된 프로그램에서는 994개의 인스트럭션이 784807개의 인스턴스를 생성하였었지만 최적화된 프로그램에서는 그 중 576개의 인스트럭션이 353445개의 인스턴스만을 생성하는 것을 볼 수 있었다. 또한 실제로 프로그램을 생성한 결과 최적화되기 전의 난독화된 프로그램과 동일한 실행 결과를 출력하는 것을 확인하였다. 다만 최적화된 프로그램의 인스트럭션 및 인스턴스 개수는 실제 생성된 프로그램으로부터 측정된 것이 아니라 제거된 인스트럭션과 그에 속한 인스턴스들의 개수를 제외한 것으로, 실제로 최적화된 프로그램을 생성하는 과정에서 일부 점프 인스트럭션을 제거할 수 없는 경우가 발생할 경우 이보다 높은 수치가 나타날 가능성이 있다. 인스트럭션은 약 42%, 인스턴스는 약 55%가 감소된 것으로 상용 난독화 도구로 난독화된 프로그램에 대한 최적화의 효과성을 확인할 수 있었다. 여기서 인스턴스 수의 감소율이 인스트럭션 수의 감소율에 비해 높게 나타나는 것은 제거된 인스트럭션이 중심 블록과 같은 여러 차례 실행되는 부분에 집중되어 있었기 때문이다.

4.2 실험 설계

실험은 크게 구조 분석, 의미 분석, 최적화의 유용성을 검증하기 위한 세 가지로 구성되었다. 대상 프로그램으로는 명령형 팩토리얼 계산(impfact), 명령형 피보나치 계산(impfibo), 재귀형 팩토리얼 계산(recfact), 재귀형 피보나치 계산(recfibo) 등 총 4개의 프로그램을 사용하였다. 이 중 재귀형 피보나치 계산 프로그램은 앞서 분석 사례에서 소개된 프로그램이기도 하다.

모든 대상 프로그램은 팩토리얼이나 피보나치 수열을 계산하고, 계산된 값을 출력한 뒤 종료하는 형태로 간단하게 구성되어 있다. 이는 의도적으로 프로그램이 사용자로부터 입력을 받지 않고 단일의 실행 경로를 갖도록 설계된 것이다. 한 프로그램이 여러 개의 실행 경로를 가질 수 있는 경우 여러 개의 실행 경로를 포함할 수 있는 테스트 케이스를 설계하는 것만으로도 커다란 문제가 되며, 이는 본 논문의 범위를 벗어나기 때문이다.

이들 각각의 대상 프로그램에서 계산 루틴을 VMProtect를 이용하여 난독화하여 원본 프로그램과 비교하여 실험을 진행하였다. 실험 대상으로 사용되는 프로그램의 크기가 작으나 가상화를 이용한 난독화를 적용하면 원본 프로그램에 비해 실행 속도가 최소 10배 이상 느려지기 때문에 실제로 적용된 사례에서도 프로그램의 많은 부분에 적용되기보다는 특별히 코드의 내용을 감출 필요가 있는 작은 부분에만 적용되기 때문에 실험의 정확도에 큰 영향을 미치지 않는다.

표 4.1은 각 프로그램의 원본과 난독화된 프로그램의 실행 속도 및 트레이스 추출에 걸린 시간을 정리한 것이다. 이 표에서는 원본 프로그램과 난독화된 프로그램의 실행 시간이 비슷한 것으로 나타나는데, 이는 프로그램의 크기가 작기 때문에 외부 함수를 호출하는 데 걸리는 시간이 더 커서 난독화에 의한 시간 증가가 잘 나타나지 않은 것으로 보인다. 또 트레이스를 추출할 때에는 원래의 실행시간에 비해 적게는 50배에서 많게는 500배 이상 느려지는 것을 볼 수 있었다. 이어서 표 4.2에서 대상 프로그램의 원본과 난독화된 대상 프로그램의 인스트럭션의 개수, 인스턴스의 개수를 볼 수 있다.

그림 4.4는 난독화에 의해 명령형 팩토리얼 계산 프로그램의 구조가 어떻게 변경되었는지를 보여준다. 그림 4.4a는 원본 프로그램의 제어 흐름 그래프이고 그림 4.4b는 난독화된 프로그램의 제어 흐름 그래프이다. 원본 프로그램에서 실제 계산이 수행되어 난독화가 적용된 부분은 8번과 9번 블록에 해당되는데, 이 부분이 난독화된 프로그램에서 완전히 변경된 것을 확인할 수 있다.

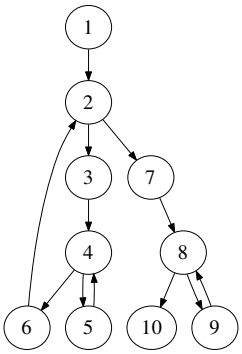
실험에서 사용된 프로그램과 실험 결과는 [29]에서 다운로드할 수 있다.

대상 프로그램	원본 프로그램		난독화된 프로그램	
	실행 시간	트레이스 추출 시간	실행 시간	트레이스 추출 시간
impfact	0.1초	26.8초	0.1초	28.5초
impfibo	0.1초	26.3초	0.1초	29.0초
recfact	0.1초	27.1초	0.1초	28.9초
recfibo	0.1초	27.4초	0.6초	31.6초

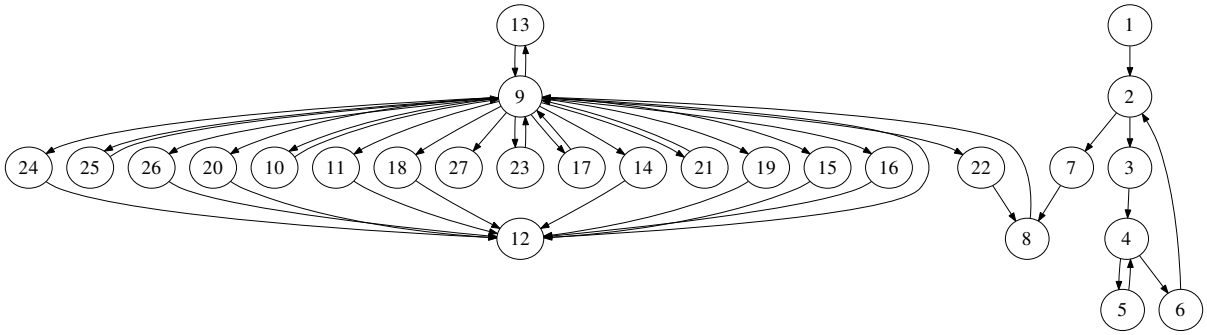
표 4.1: 실험 대상 프로그램

대상 프로그램	원본 프로그램		난독화된 프로그램	
	인스트럭션 수	인스턴스 수	인스트럭션 수	인스턴스 수
impfact	322	407	896	222036
impfibo	324	465	899	294173
recfact	316	428	952	248648
recfibo	328	734	994	784807

표 4.2: 실험 대상 실행 트레이스



(a) 원본 프로그램



(b) 난독화된 프로그램

그림 4.4: 명령형 팩토리얼 프로그램의 제어 흐름 그래프

4.3 프로그램의 구조 파악

정적 분석을 통해 생성된 제어 흐름 그래프는 일반적으로 가상 기계의 중심 블록 이후를 정확히 예측해내지 못했던 반면 동적 분석을 통해 생성된 제어 흐름 그래프는 실제 실행된 인스트럭션을 바탕으로 생성되므로 모든 핸들러가 반영되었다. 때문에 동적 분석을 통해 생성된 제어 흐름 그래프가 정적 분석을 통해 생성된 제어 흐름 그래프에 비해 훨씬 유용함이 자명하여 이 둘을 비교하는 것은 의미가 없을 것으로 판단하여 실험에서는 제외하였다.

한편 가상 기계의 중심 블록의 특징을 바탕으로 동적 제어 흐름 분석을 통해 중심 블록을 찾아내는 방법에 대한 실험을 진행하였다. 2.3.2에서 가상 기계의 중심 블록은 보통 다른 블록에 비해 여러 차례 실행되며 많은 수의 나가는 간선을 가질 것으로 예상하였다. 이를 확인하기 위하여 각 실험 대상 프로그램에 대해 가장 많이 실행되는 블록과 두번째로 많이 실행되는 블록을 찾고 각 블록이 실행된 횟수를 찾았다. 이와 더불어 나가는 간선을 가장 많이 가진 블록과 두번째로 많이 가진 블록을 찾고 각 블록이 가진 나가는 간선의 개수를 찾았다. 표 4.3은 이렇게 찾은 정보를 보이고 있다.

실험 결과에서 실행 횟수가 가장 많은 블록과 나가는 간선의 수가 가장 많은 블록이 항상 일치하는 것을 볼 수 있고, 실행 횟수와 나가는 간선의 수가 두번째로 많은 블록에 비해 그 수가 압도적으로 많은 것을 확인할 수 있다. 또한 해당 블록의 내부를 분석하여 이들 블록들이 가상 기계의 중심 블록인 것을 알 수 있었다. 이처럼 예상했던 가상 기계 중심 블록의 특징을 실례에 적용하여 중심 블록을 찾는 데 이용할 수 있음을 볼 수 있었다.

대상 프로그램	실행 횟수		나가는 간선의 수	
	첫째	둘째	첫째	둘째
	블록 / 횟수	블록 / 횟수	블록 / 개수	블록 / 개수
impfact	9 / 3062	12 / 1168	9 / 17	2, 4 / 2
impfibo	9 / 3755	13 / 1424	9 / 16	2, 4, 10, 13 / 2
recfact	10 / 3239	13 / 1274	10 / 16	2, 4, 25 / 2
recfibo	10 / 10769	13 / 4271	10 / 15	30 / 3

표 4.3: 가상 기계의 중심 블록

4.4 프로그램의 의미 파악

분석 사례에서 보인 것과 유사한 방법으로 하나의 프로그램에 대해 이 프로그램이 출력한 값이 어떻게 계산되는 지를 추적하여 출력된 값과 연관된 부분 의존성 그래프를 추출하였다. 그리고 그 중 값 이동이나 스택 연산 인스트럭션을 제외한 인스트럭션을 관심 있는 인스트럭션으로 하여 수식 트리를 그렸다. 표 4.4는 각 실험 대상 프로그램에 대하여 원본 프로그램에서 추출한 출력된 값에 연관된 부분 의존성 그래프에 나타난 OPCODE, 난독화된 프로그램에서 추출한 출력된 값에 연관된 부분 의존성 그래프에 나타난 OPCODE들의 목록, 그리고 마지막으로 원본 프로그램에서와 난독화된 프로그램의 수식 트리를 그리기 위해 공통적으로 사용된 OPCODE들의 목록을 보이고 있다.

실험 대상 프로그램 impfact의 원본 프로그램에서는 MOV와 PUSH가 각각 값 이동 및 스택 연산 명령이기 때문에 제외되고 ADD와 IMUL 명령만 수식 트리를 그릴 때 사용되었다. 난독화된 프로그램에는 그 외에 CWDE와 POP 명령이 추가로 나타났는데, POP은 역시 스택 연산 명령이므로 제외하였고 CWDE는 인코딩된 상수 값을 디코딩하는 루틴의 마지막 인스트럭션으로 무시하여도 좋다. CWDE 명령은 다른 난독화된 프로그램들에서도 공통적으로 나타나는데, 항상 인코딩된 상수 값을 디코딩하는 루틴의 마지막 인스트럭션이어서 무시하였다. impfibo에서도 마찬가지로 MOV, PUSH, POP, CWDE

대상 프로그램	OPCODE	
impfact	원본	ADD, IMUL, MOV, PUSH
	난독	ADD, CWDE, IMUL, MOV, POP, PUSH
	사용	ADD, IMUL
impfibo	원본	ADD, MOV, PUSH
	난독	ADD, CWDE, MOV, POP, PUSH
	사용	ADD
recfact	원본	IMUL, MOV, PUSH, SUB
	난독	ADD, AND, CWDE, IMUL, MOV, NOT, POP, PUSH
	사용	IMUL, SUB / ADD, IMUL
recfibo	원본	ADD, MOV, POP, PUSH, XOR
	난독	ADD, AND, CWDE, MOV, NOT, PUSH
	사용	ADD

표 4.4: 연관된 부분 의존성 그래프에 나타난 OPCODE 및 수식 트리 생성에 사용된 OPCODE

은 제외하고 ADD만 사용하였다. recfact에서는 원본 프로그램의 SUB가 난독화된 프로그램에서는 사라지고 ADD가 생겼다. 이는 난독화 과정에서 양의 정수인 초기 변수 값을 음수로 바꾸고 1씩 감소시키는 대신 1씩 증가시키고, 실제 계산 직전에 여기에 NOT을 적용하여 사용하도록 수정되었기 때문이다. 때문에 수식 트리를 그릴 때에도 원본 프로그램에선 곱하기 명령인 IMUL과 빼기 명령인 SUB가 사용된 반면 난독화된 프로그램에선 IMUL과 더하기 명령인 ADD가 사용되었다. recfibo의 원본 프로그램에서 MOV, POP, PUSH는 값 이동 및 스택 연산이라 제외되고, XOR은 확인 결과 변수의 값이 0과 같은 지를 확인하기 위해 사용되었으므로 제외하고 ADD만 이용하여 수식 트리를 그렸다. 난독화된 프로그램에서도 마찬가지로 ADD만 사용하여 수식 트리를 그렸다.

앞서 그림 4.3에서 recfibo 프로그램의 원본 프로그램과 난독화된 수식 트리를 보여 두 개가 일치하는 것을 보였다. 다른 프로그램들도 이와 같은 방법으로 수식 트리를 그렸을 때 거의 일치하는 형태로 나타나는 것을 볼 수 있어서 본 논문의 프로그램 의미 파악 방법이 유용함을 확인하였다.

4.5 프로그램의 최적화

마지막으로 동적 분석을 통해 프로그램을 최적화하는 알고리즘의 유용성을 확인하였다.

프로그램 최적화의 모든 단계를 자동으로 수행하였으며, 이러한 과정을 거쳐 생성된 최적화된 프로그램이 오류 없이 실행되고 최적화되기 전의 프로그램과 동일한 결과를 출력하는 것을 먼저 확인하였다. 그리고 최적화되기 전의 프로그램과 최적화된 프로그램의 인스트럭션과 인스턴스의 수를 비교하였다.

최적화된 프로그램의 인스트럭션 수는 최적화되기 전의 프로그램에서 제거된 인스트럭션의 수를 빼서 계산하였고, 최적화된 프로그램의 인스턴스 수는 최적화되기 전의 프로그램에서 제거된 인스트럭션에 의해 생성된 인스턴스들의 수를 빼서 계산하였다. 즉, 최적화된 프로그램의 인스턴스 수는 다음과 같이 계산된 것이다.

$$\left|\bigcup_{I\in\{k\in Z_i | k\notin Dropped\}} instances(I)\right|$$

이는 실제 최적화된 프로그램 생성 과정에서 인스트럭션의 주소를 보존하기 위해 추가되는 점프 명령을 고려하지 않은 것이다. 때문에 최적화된 프로그램을 실제로 생성하면 표에 표시된 것보다 많은 수의 인스트럭션과 인스턴스가 생성될 수 있다.

최적화되기 전의 난독화된 프로그램과 최적화된 프로그램의 인스트럭션 수와 인스턴스의 수를 비교하여 표 4.5에 나타내었다.

대상 프로그램	인스트럭션 수			인스턴스 수		
	원본	난독	%	원본	난독	%
impfact	896	500	-44.2%	222036	87322	-60.7%
impfibo	899	489	-45.6%	294173	108589	-63.1%
recfact	952	533	-44.0%	248648	123569	-50.3%
recfibo	994	576	-42.1%	784807	353445	-55.0%

표 4.5: 최적화 점수

표에서 볼 수 있는 바와 같이 인스트럭션의 수는 42%에서 45%까지 감소하는 것을 볼 수 있었고, 인스턴스의 수는 50%에서 63%까지 감소하는 것을 볼 수 있었다. 특히 많은 수의 스택 연산과 디코딩 루틴 및 인코딩된 데이터가 제거된 것을 볼 수 있었다.

인스턴스의 감소율이 인스트럭션의 감소율에 비해 높게 나타나고 있는데 이는 제거된 인스트럭션이 여러 차례 실행되는 가상 기계의 중심 블록이나 핸들러에 주로 나타나고 있기 때문이다.

제 5 장 결론

본 논문은 분석 대상 프로그램을 실제로 실행하고 실행 과정을 기록하여 그 내용을 바탕으로 프로그램을 분석하는 동적 프로그램 분석에 대하여 논하였고, 프로그램 분석에 사용될 수 있는 다양한 알고리즘을 설계하고 기술하여 분석 대상 프로그램의 구조 및 내용을 분석하고 더 나아가 보다 분석하기에 용이한 형태로 최적화하는 알고리즘을 소개하였다. 또한 이러한 알고리즘을 구현하고 하나의 분석 도구로 통합한 Trudio에 대해 기술하였고, 실험을 통하여 이러한 접근 방법의 유용성을 증명하였다.

하지만 본 논문은 다음과 같은 한계를 지니고 있다. 첫째, 최적화 알고리즘의 안전성을 수학적으로 증명하지 않았다. 때문에 최적화된 프로그램이 원본 프로그램의 실행 경로를 완전히 구현한다고 보장할 수 없다. 따라서 최적화 알고리즘의 안전성을 증명하는 것이 추가로 연구되어야 할 것이다. 둘째, 동적 분석의 태생적 한계로 커버리지(coverage) 문제가 있다. 즉, 동적 분석 알고리즘들은 모두 분석의 대상이 되는 실행 경로만 고려하므로 그 외의 실행 경로에 대해서는 분석할 수 없다는 것이다. 분석 대상 프로그램을 다양한 테스트 케이스로 실행하여 보다 많은 코드를 포함하는 실행 트레이스를 얻어내면 이 문제를 해결할 수 있다. 그러나 이는 본 논문의 범위를 벗어나므로 자세히 논하지 않는다. 셋째, 개발된 분석 도구가 실제 분석자들의 분석 시간을 얼마나 줄일 수 있는 지 실험을 통해 정량적인 분석을 수행하지 못 하였다.

이러한 단점에도 불구하고 본 논문에서 제시된 아이디어와 알고리즘 및 구현된 분석 도구들을 이용하여 난독화된 프로그램의 구조와 의미를 보다 쉽게 파악할 수 있게 되었다. 특히 가상화를 이용한 난독화 기법이 적용된 실행 파일의 경우 가상 기계에서 일반적으로 나타나는 특징을 이용하여 그 구조를 보다 심도있게 분석할 수 있는 도구를 제공하게 되었다. 이를 통해 난독화가 적용된 악성 코드들에 선제적으로 대응할 수 있게 될 것으로 기대된다.

참 고 문 헌

[1] "Javascript obfuscator," http://javascript-source.com/, 2009.

[2] E. Lafortune, "Proguard alternatives," http://proguard.sourceforge.net/alternatives.html, 2011.

[3] R. Benzmüller and S. Berkenkopf, "G data malware report, half-yearly report january - june 2011," G Data SecurityLab, Tech. Rep., 2011.

[4] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, "Malware normalization," University of Wisconsin, Madison, Wisconsin, USA, Tech. Rep. 1539, Nov. 2005.

[5] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Network and Distributed System Security Symposium*, 2008.

[6] G. Wroblewski, "General method of program code obfuscation," in *Software Engineering Research and Practice*, 2002.

[7] VMProtect Software, "Vmprotect software protection," http://vmpsoft.com/, 2011.

[8] Oreans Technologies, "Code virtualizer," http://www.oreans.com/codevirtualizer.php, 2011.

[9] Hex-Rays SA, "IDA Pro," http://www.hex-rays.com/products/ida/index.shtml, 2011.

[10] O. Yuschuk, "OllyDbg," http://ollydbg.de/, 2011.

[11] R. Rolles, "Control flow deobfuscation via abstract interpretation."

[12] B. Spasojević, "Code deobfuscation by optimization," in *27th Chaos Communication Congress*, 2010.

[13] ——, "Using optimization algorithms for malware deobfuscation," Diploma Thesis, UNIVERSITY OF ZAGREB, 2010.

[14] ——, "optimice - deobfuscation plugin for ida," http://code.google.com/p/optimice/.

[15] Y. Guillot and A. Gazet, "Automatic binary deobfuscation," *Journal in Computer Virology*, pp. 261–276, 2010.

[16] Y. Guillot, "The METASM assembly manipulation suite," http://metasm.cr0.org/.

[17] J.-Y. Marion and D. Reynaud, "Dynamic binary instrumentation for deobfuscation and unpacking," in *In-Depth Security Conference Europe 2009 (Deepsec09)*, 2009.

[18] D. Reynaud, "tartetatintools - a bunch of experimental pintools for malware analysis," http://code.google.com/p/tartetatintools/.

[19] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX conference on Offensive technologies*, ser. WOOT'09. Montreal, Canada: USENIX Association, 2009.

[20] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 275–284.

[21] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX 2005 Annual Technical Conference*, 2005, pp. 41–46.

[22] K. Lawton, "bochs: The open source ia-32 emulation project," http://bochs.sourceforge.net/, 2011.

[23] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[24] J. Jeon, "Tracing tool," http://trudio.googlecode.com/files/tracing.zip.

[25] G. Dabah, "diStorm3," http://code.google.com/p/distorm/, 2011.

[26] S. Tatham and J. Hall, "The netwide assembler," http://www.nasm.us/, 2011.

[27] J. Jeon, "Trudio," http://trudio.googlecode.com/files/Trudio.zip.

[28] "Graphviz - graph visualization software," http://www.graphviz.org/.

[29] J. Jeon, "Trudio experiments," http://trudio.googlecode.com/files/experiments.zip.

감 사 의 글

주의의 많은 분들의 도움이 없었다면 이 논문을 완성할 수 없었을 것입니다. 먼저 항상 따뜻한 격려와 조언을 아끼지 않고 지도해주신 한태숙 교수님께 깊이 감사드립니다. 또한 바쁘신 와중에 논문 심사를 해 주신 김명호과 류석영 교수님께도 감사드립니다. 항상 연구에 대해 함께 고민해 주신 연구실 선배 윤경 누나와 준형이 형께도 감사드립니다. 프로그래밍 언어 연구실의 현익이 형, 재준이 형, 지응이 형, 창희 형, 성훈이 형, 홍기 형, 명희 형께도 감사의 말씀을 드립니다. 먼저 졸업하신 정한이 형, 신형이 형, 철우 형, 성건이 형, 그리고 올 한 해 제 연구를 이끌어 주신 석우 형께도 감사의 말씀을 드립니다. 모든 분들 덕분에 즐겁게 배우고 연구할 수 있었습니다. 기숙사 생활을 함께해 준 승우와 용훈이와 함께 했던 시간도 잊지 못할 것입니다. 제게 항상 활력소가 되어준 친구들 이재, 동수, 연준, 평화, 민영, 건하, 경민, 동우, 규성, 동근에게도 고맙다는 말을 전합니다. 각자의 자리에서 군생활을 하고 있는 선일, 태웅, 강민, 성혁, 동환에게도 감사와 응원을 전하고 싶습니다. 그리고 마지막으로 언제나 저를 믿고 응원해 주시는 부모님과 동생 민수에게 항상 고맙다는 말을 전하고 싶습니다.

이 력 서

이 름 : 전 준 수

생 년 월 일 : 1988년 7월 27일

E-mail 주 소 : jeon.joonsoo@gmail.com

학 력

2004. 3. – 2007. 2. 경기과학고등학교

2007. 2. – 2010. 2. 한국과학기술원 전산학과 (B.S.)

2010. 2. – 2012. 2. 한국과학기술원 전산학과 (M.S.)

경 력

2010. 9. – 2011. 5. 한국과학기술원 전산학과 조교