韓國컴퓨터情報學會 論文誌
第16卷 第6號, 2011. 6.

2011-16-6-1-6

이진 코드의 정적 실행 흐름 추적을 위한 프레임워크 설계 및 구현

백 영 태*, 김 기 태**, 전 상 표***

Design and Implementation of Framework for Static Execution Flow Trace of Binary Codes

Baek YeongTae*, Kim ki Tae**, Jun Sang Pyo***

요 약

국내에는 바이너리 코드에 대한 분석 기술이 많이 부족한 상태이다. 일반적으로 컴퓨터에 설치되는 실행 파일은 소스 코드 없이 단지 바이너리로 된 실행 파일만 주어지는 경우가 대부분이다. 따라서 위험하거나 알 수 없는 동작이 수행되는 경우가 발생할 수 있다. 따라서 이 논문에서는 바이너리 수준에서 정적으로 프로그램 분석을 수행할 수 있는 프레임워크를 설계 및 구현한다. 이 논문에서는 바이너리 실행 파일로부터 실행 순서 및 제어 흐름 등의 정보를 표현할 수 있는 제어 흐름 그래프를 작성하여 실행 흐름과 위험한 함수의 호출 여부를 동시에 파악하고 개발된 프레임워크를 통해 바이너리 파일에 대한 분석을 용이하게 한다.

▸ Keyword : 이진 코드, 제어 흐름 분석, 프레임워크

Abstract

In domestic, the binary code analysis technology is insufficient. In general, an executable file that is installed on your computer without the source code into an executable binary files is given only the most dangerous, or because it is unknown if the action is to occur. In this paper, static program analysis at the binary level to perform the design and implementation framework.

In this paper, we create a control flow graph. We use the graph of the function call and determine whether dangerous. Through Framework, analysis of binary files is easy.

▸ Keyword : binary code, control flow analysis, framework

• 투고일 : 2010. 12. 21, 심사일 : 2011. 03. 02, 게재확정일 : 2011. 5. 24
* 김포대학 멀티미디어과 부교수(Dept. of Multimedia, Kimpo College)
** (주)나루기술 선임연구원(Naru Technology Co. Ltd.)
*** 남서울대학교 교양학부 교수(School of General Educaton, Namseoul University)
※ 이 논문은 2011학년도 김포대학의 연구비 지원에 의하여 연구되었음
※ 이 논문은 2010년 한국컴퓨터정보학회 제42차 하계학술대회에 발표한“바이너리 코드의 정적 제어 흐름 분석을 위한 프레임워크”을 확장한 것임.

I. 서 론

현재 국내에는 바이너리 코드에 대한 분석 기술이 많이 부족한 상태이다. 하지만 점점 더 인터넷을 통해 검증되지 않은 실행 파일을 내려 받아 컴퓨터에 설치하여 사용하는 경우는 증가하고 있으며, 외부에서 제공되는 API에 대해서도 특별한 검증 과정 없이 사용하여 새로운 프로그램이 개발되고 있는 실정이다. 일반적으로 바이너리 코드는 소스 코드를 컴파일하여 생성한다. 하지만 사용된 컴파일러와 컴파일 과정이 수행된 환경에 따라 다른 형태의 바이너리 코드가 생성될 수 있다. 게다가 바이너리 코드의 동작은 작성된 원래 소스 코드를 확인 할 수 없는 경우가 빈번하기 때문에 프로그램의 의도를 정확히 알 수 없는 경우가 많이 발생한다[1].

이러한 문제를 해결하기 위해서는 바이너리 수준의 프로그램 분석이 요구된다. 프로그램 분석이란 프로그램의 행동이나 동작에 대한 분석을 자동으로 수행하는 기술로 최근엔 이러한 프로그램 분석 기술이 여러 소프트웨어 공학 분야와 컴퓨터 보안 분야에 적용되고 있는 추세이다. 특히 컴퓨터 보안 분야에서 프로그램의 버그나 코드의 취약점을 찾기 위한 연구 분야에 많이 시도되거나 적용되고 있다[2]. 프로그램 분석 시 소스 코드가 제공되지 않는 경우라면, 실행 파일인 바이너리 코드를 역어셈블[3]할 수 있고, 이에 대한 제어 흐름 그래프를 작성할 수 있고, 또한 정적으로 실행 흐름 분석을 할 수 있는 새로운 자동 도구가 필요하게 된다.

기존의 도구들은 대부분 특정 문제에 국한되어서만 동작하거나 단순한 형태의 그래프만을 제공하는 경우가 대부분이다. CodeSurfer/x86[4]의 경우엔 추상 해석(abstract interpretation)을 적용한 경우로 Value-set Analysis에 의해 예상되는 값에 대해 어림짐작되는 값을 평가해 처리한다. 이 경우 어림짐작되는 값을 사용하기 때문에 가능성에 대한 여부확인에는 좋지만, 정확한 그래프 작성은 어려운 방식이다. Digital Genome Mapping 방법[5]은 그래프를 작성한 후 그래프의 유사도를 이용하여 바이너리 코드의 유사성을 및 차이점을 찾아내는 방식이다. 이 경우 함수 이름을 바탕으로 유사도를 찾기 때문에 바이너리 코드 자체에 대한 분석보다는 코드의 변종 여부를 판단하는 용도로 사용된다. IDA Pro[6]의 경우에도 제어 흐름 그래프를 보여주기는 하지만 특정 프로시저에 대해서만 확인이 가능하고 전체적인 흐름은 파악할 수 없다는 단점이 존재하였다.

따라서 이 논문에서는 새로운 도구를 설계 및 개발하여 바이너리 실행 파일로부터 함수 간의 실행 순서 및 제어 흐름 등의 정보를 표현할 수 있는 실행 흐름 그래프를 작성한다. 사용자는 작성된 실행 흐름 그래프를 이용하여 바이너리 파일의 실행 흐름과 위험한 함수의 호출 여부를 파악할 수 있게 되고, 그래프를 이용하여 바이너리 파일에 대한 분석을 수행한다. 또한, 특정 실행 흐름에서 임의의 경로에 대한 자동 탐색 방법을 구현한다[7].

이 논문의 구성은 다음과 같다. 2장에서는 정적 분석과 바이너리 코드 분석에 대한 관련 연구를 설명하고, 3장에서는 바이너리 코드 분석을 위한 프레임워크를 설계 및 구현한다. 4장에서는 예제를 통해 실제 구현된 도구와 도구를 통한 실험 결과를 보여주고, 5장에서는 결론 및 향후 연구계획에 대해 설명한다.

II. 관련 연구

1. 동적 분석과 정적 분석

프로그램 분석은 분석이 수행되는 시점에 따라 크게 프로그램을 실행하기 전 수행하는 정적 분석(Static Program Analysis)과 프로그램을 실행하면서 수행하는 동적 분석(Dynamic Program Analysis)으로 구분할 수 있다[8].

동적 분석은 프로그램이 실행 중에 가질 수 있는 성질이나 행동을 실행 중에 자동으로 분석하는 방법을 말한다. 실행 시간에만 정확히 알 수 있는 메모리에 대한 사용이나 사용자의 입력에 대해 분석할 경우엔 정확한 결과를 얻을 수 있다. 그러나 동적 프로그램 분석의 경우, 실제 프로그램을 실행하면서 수행되기 때문에 실행 시간 안전성이 요구되는 중요한 프로그램에 대해 적용하기에는 부적절하다는 단점이 존재한다.

반면 정적 분석은 프로그램이 실행 중에 가지는 성질을 프로그램 실행 전에 자동으로 안전하게 어림잡는 방법이다. 컴파일 시간에 분석이 이루어지기 때문에 컴파일시간 분석이라고도 한다. 정적 분석 방법으로는 현재까지 여러 기술들이 개발되고 있으며 대표적으로는 프로그램의 의미 구조를 구문적으로 분석하는 타입 시스템, 프로그램 실행 중 의미 있는 속성들을 요약 또는 추상화하여 분석하는 요약 해석, 프로그램을 명세에 따라 작성되었는지 검증하는 프로그램 검증, 프로그램을 상태에 따라 변하는 모델로 구성하여 검사하는 모델 검사 등의 기술이 사용된다.

2. 소스 코드 분석과 바이너리 코드 분석

프로그램의 실행 전과 실행 후 뿐만 아니라 분석을 소스

코드 수준에서 할지 아니면 바이너리 코드 수준에서 수행할지도 고려된다. 소스 코드 수준의 분석은 분석 후 취약점이 발생했을 때 프로그램의 수정이 용이하다는 장점이 있지만 다음과 같은 몇 가지 단점이 존재한다.

소스코드 수준의 분석의 경우 컴파일러가 하는 일에 대해서는 신경을 쓰지 않는다. 즉, 컴파일 과정 중 포함되는 정적 라이브러리나 프로그램이 실행될 때 동적으로 호출하는 DLL 등에 대해서는 고려할 수가 없다는 것이다. 또한 컴파일러의 최적화 과정 중 머신에 따라 코드가 변형되는 경우나 여러 프로그램 언어로 작성된 프로그램일 경우에도 문제가 발생한다. 웜이나 바이러스 같은 경우에는 소스 코드가 존재 하지 않기 때문에 이러한 경우에는 소스 코드 수준의 분석이 불가능하다는 단점이 존재한다.

반면, 바이너리 수준의 분석은 몇 가지 장점을 가진다. 첫째, 우선 바이너리 코드 수준의 프로그램 분석은 메모리 레이아웃, 레지스터, 실행 순서 등 머신에 종속적인 사항들도 고려할 수 있다. 둘째, 인라인 어셈블리 코드가 포함된 경우에도 분석이 가능하다. 셋째, 정확하고 넓은 범위의 분석이 가능하다는 장점을 갖는다[11]. 물론 바이너리 파일도 몇 가지 문제점을 포함한다. 첫째, 명령어가 다양한 길이를 갖고 코드와 데이터가 섞여 있는 형태의 명령어가 많아 명령어의 경계를 결정하기 어렵다. 둘째, 발견된 명령어에 대해서 동작에 대한 분석이 요구되고 특별한 조작이 필요하다. 셋째, 함수에 대한 경계를 찾기 어렵다는 것이다. 하지만 바이너리 파일 내에 존재하는 몇 가지 정보는 이러한 문제를 해결할 수 있도록 도와준다. 우선 심벌 정보는 바이너리 코드 내의 코드와 데이터를 분리하고 함수 정보를 찾는데 도움을 제공한다. 재배치 정보는 간접 함수 호출의 목표와 모호한 제어 전송을 결정하도록 도움을 준다. 하지만 심벌 정보와 재배치 정보는 제공되지 않는 경우도 존재한다. 최근 바이너리 분석 기법들은 이러한 정보가 없는 경우에도 분석을 수행할 수 있는 다양한 방법과 도구가 제공된다.

III. 프레임워크 설계 및 구현

1. 프레임워크 개요

분석을 수행하는 프레임워크인 Flow Graph Analyzer의 전체 구성도는 그림 1과 같다[12]. 프레임워크는 크게 IDA Pro, 분석기, 그리고 그래프 시각화 도구 이렇게 세 부분으로 나뉜다.

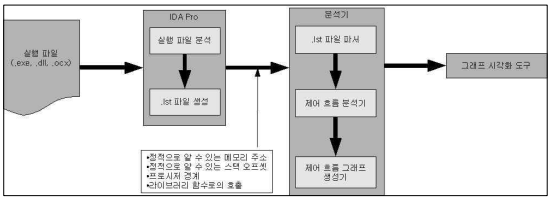

그림 1. 프레임워크
Fig. 1 Framework

첫 번째 부분은 IDA Pro 처리 부분이다. IDA Pro는 Hex-rays에서 개발한 상용 역어셈블러로 라이브러리 함수 인식, 디버깅 등의 기능을 제공하여 바이너리 코드 분석에 많이 이용리 코도구이다. 이 논문에s에서IDA Pro를 이용해 .exe, .dll, .ocx와 l, 확장자를 가진 바이너리 파일들을 역어셈블한다. 이 버깅에서 IDA Pro로부터 정 IDA알 수 있는 메모리의 주소와 스택 오프셋의 정보를 획득한다. IDA Pro를 이용해 .lst 확장자를 갖는 파일을 생성한다. .lst 파일, 그래프 생성을 위해 필요한 오프셋 정보들을 갖고 있너리 파일, 다음 버깅인 분석기의 입력으로 사용한다.

두 번째 부분은 입력으로 들어오는 .lst 파일을 분석하는 과정이다. 다양한 정보를 포함하고 있는 .lst파일을 이용해 제어 흐름 그래프를 만들기 위해서 우선 .lst 파일을 파싱한다. .lst 파일을 파싱하는 과정에서 각 명령어의 주소와 프로시저의 경계 정보를 이용하여 각 프로시저를 구분하고 명령어의 분기 정보, 호출 정보, 그리고 라이브러리 함수에 관한 정보 등을 추출한다. 제어 흐름 분석기는 파싱을 통해 얻은 정보와 기본 블록에 따라 제어 흐름 그래프를 생성한다.

설계의 마지막 부분은 그래프 시각화 도구이다. 분석기 부분을 통해 구성된 제어 흐름 그래프는 그래프 시각화 도구를 통해 사용자의 요구에 따라 원하는 정보를 보여주는 역할을 수행한다.

2. 프레임워크 설계 및 구현

이 논문에서 다루는 바이너리 파일 형식은 윈도우 실행 파일인 PE 파일이다.

2.1 .lst 파일의 파싱과 정보 추출

분석할 바이너리 파일인 PE 파일을 분석하기 위해서는 PE 파일을 파싱하고, 역어셈블, 라이브러리 함수 찾기와 같은 추가적인 분석이 필요하다. 이 중 역어셈블 과정을 쉽게 하기위해 IDA Pro를 이용한다. IDA Pro를 통해 함수의 위치나 함수들의 정보는 쉽게 얻을 수 있다. 보통의 경우 함수는 주소를 사용하여 다루어야 한다는 단점이 존재하여 주소에 의한 검색이 힘들다는 문제점이 발생한다. 하지만 IDA Pro는 함수에 대

해 이름 정보를 제공해주기 때문에 특정 함수를 이름으로 쉽게 다룰 수 있게 해준다.

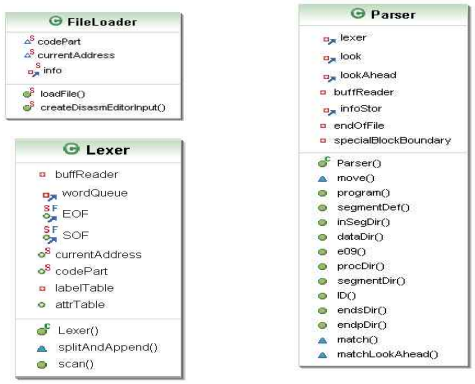

그림 2. FileLoader, Lexer, Parser 클래스
Fig. 2 FileLoader, Lexer, Parser classes

IDA Pro를 통해 생성한 .lst 파일을 통해서 각 명령어의 주소를 알 수 있고, 프로시저의 시작 주소와 끝 주소를 쉽게 알 수 있게 된다. .lst 파일로부터 필요한 정보를 추출하기 위해서는 파일 로딩과 어휘 분석 그리고 구문 분석 과정이 필요하다. 그림 2의 첫 번째는 .lst 파일을 읽기 위한 FileLoader 클래스이다. FileLoader 클래스는 loadFile() 메소드를 이용해 .lst 파일을 로딩한다. 이 때 loadFile() 메소드 안에서는 어휘 분석을 위해 Lexer 클래스를 이용하고, 구문 분석을 위해서는 Parser 클래스를 이용한다. Lexer클래스는 Parser 클래스에 의해서 이용되는데 Parser에서 Lexer의 scan() 메소드를 이용해 역어셈블리 구문에 맞는 어휘를 인식한다.

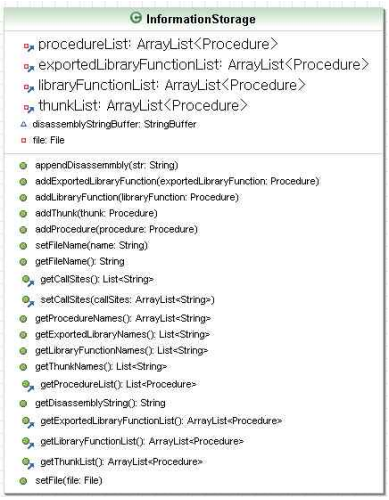

그림 3. InformationStorage 클래스
Fig. 3 InformationStoage class

Parser 클래스는 FileLoader가 Parser 클래스의 program() 메소드를 호출하면서 역어셈블리 구문에 맞게 구문 분석을 수행하고, 구문 분석 수행 과정 중 .lst파일에서 그래프 생성과 분석을 위한 정보를 추출하여 Procedure 단위로 인식한 정보를 그림 3의 InformationStorage에 저장하는 역할을 한다.

그림 3은 .lst 파일을 로드할 때 얻은 정보를 저장하기 위한 InformationStorage 클래스를 나타낸다. InformationStorage는 각각 프로시저를 위한 리스트, 엑스포티드(exported) 함수를 위한 리스트, 라이브러리 함수를 위한 리스트, thunk를 위한 리스트로 procedureList, exportedLibraryFunctionList, library FunctionList 그리고 thunkList를 포함한다. 그 밖에 로드된 파일을 나타내는 file, 전체 역어셈블리를 나타내는 disassembly StringBuffer 등을 가지고 있다.

2.2 그래프의 생성

FileLoader 클래스를 통해 InformationStorage에 정보를 모두 뽑아낸 후 그래프를 생성할 수 있다. 그래프를 생성하기 위해서는 그래프 생성 시 포함될 프로시저가 정해져야 하고 그 프로시저를 구성하는 기본 블록을 이용해 그래프를 생성해야 한다. 우선, 기본 블록을 정의하고 기본 블록 사이의 제어 흐름을 정의해야 한다.

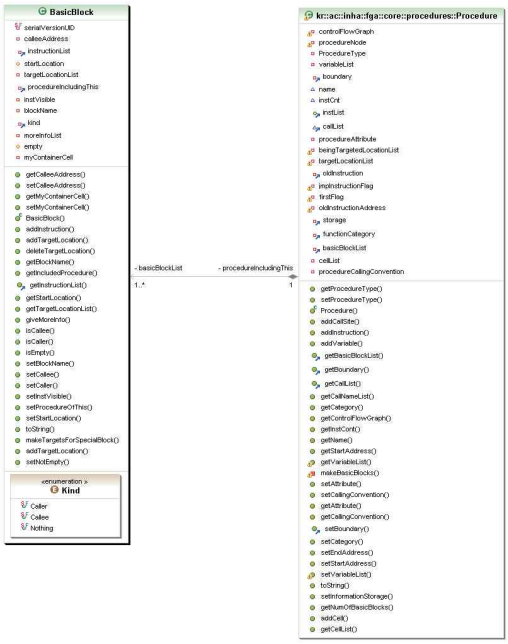

그림 4. Procedure와 BasicBlock 간의 관계
Fig. 4 Relation of Procedure and BasicBlock

기본 블록을 나누는 기준으로 기본 블록을 생성한 후 제어의 흐름에 따라 기본 블록을 간선으로 연결해야 한다. Jmp 명령어를 포함한 기본 블록부터 호출의 대상이 되는 주소를

갖는 기본 블록으로 제어 흐름이 존재하며, Call 명령어를 포함한 기본 블록부터 호출의 대상이 되는 라이브러리 함수나 프로시저로 제어 흐름이 존재한다.

그림 4는 그래프를 구성하는 기본 블록을 나타내는 BasicBlock 클래스와 프로시저와의 관계를 나타낸다. 각 프로시저 마다 기본 블록의 리스트가 존재한다. BasicBlock 클래스는 kind 변수를 이용해 기본 블록의 종류를 나타낸다. 기본 블록은 Caller, Callee, 그리고 이 둘에 속하지 않는 Nothing으로 나누어 진다. kind가 Caller인 기본 블록은 Call 명령어를 포함하는 기본 블록이고, kind가 Callee인 기본 블록은 Call 명령어의 대상이 되는 기본 블록이다.

BasicBlock 클래스는 기본 블록의 시작 주소와 기본 블록이 속한 프로시저에 관한 정보 기본 블록에서 제어가 향할 수 있는 대상 주소의 리스트에 대한 정보를 포함한다. Procedure는 프로시저의 이름, 프로시저의 종류, 프로시저를 구성하는 기본 블록의 리스트에 대한 정보를 포함한다.

BasicBlock 클래스와 Procedure 클래스를 이용해 실제 프로시저의 그래프를 생성하기 위해서는 기본 블록의 연결 관계에 대한 정보를 생성해야 한다. 이는 각 프로시저를 나타내는 Procedure 클래스의 getBasicBlockList() 메소드에서 이루어진다. getBasicBlockList() 메소드는 처음에 기본 블록간의 연결 관계가 설정되지 않은 때에는 기본 블록간의 연결 관계를 기본 블록의 종류와 제어 흐름에 따라 설정하고 처음 호출되지 않는 경우에는 처음 설정한 정보를 얻는 역할을 한다.

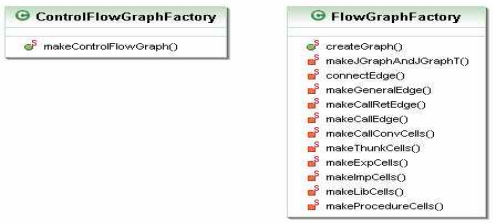

그림 5. ControlFlowGraphFactory 와 FlowGraphFactory
Fig. 5 ControlFlowGraphFactory and FlowGraphFactory

그림 5는 ControlFlowGraphFactory 클래스와 FlowGraphFactory 클래스이다. 생성된 모델을 그래픽으로 보여주기 위해 ControlFlowGraphFactory 클래스와 FlowGraph Factory 클래스를 사용한다. ControlFlowGraphFactory 클래스는 호출 그래프를 생성하기 위한 클래스로 makeControl FlowGraph() 메소드로써 특정 프로시저의 제어 흐름 그래프를 생성한다. FlowGraphFactory는 모든 프로시저를 포함한 흐름 그래프의 생성을 위한 클래스로 createGraph() 메소드로써 모든 프로시저의 기본 블록의 제어 흐름 관계에 따라 기본 블록들을 연결하여 그래프를 생성한다. FlowGraph Factory는 실제로 보여주기 위한 뷰를 생성하는 클래스이다. makeCallConvsCells(), makeThunkCells(), makeExpCells(), makeImpCells(), makeLibCells(), makeProcedureCells() 메소드들을 이용해 그래프 셀을 생성한다. connectEdge()메소드 안의 makeGeneralEdge(), makeCallEdge(), makeCallRet Edge() 메소드를 이용해 간선을 연결한다.

2.3 그래프의 시각화

그래프의 시각화는 현재까지 구성한 그래프를 실제로 보여주는 것을 의미한다. 이를 위해 그래프를 프로시저 단위로 그룹화 하여 여러 기본 블록을 하나의 노드로 나타낼 수 있어야 하고, 여러 그래프와 관련된 정보를 보이기 위해 그래프의 이벤트를 처리해야 한다.

ControlFlowGraphFactory 클래스와 FlowGraphFactory 클래스에 의해 기본 블록들 간의 연결 관계가 형성되면 프로시저의 제어 흐름 그래프는 CFGEditor를 통해 나타낼 수 있고 흐름 그래프는 GraphEditor를 통해 나타낼 수 있다. 호출 그래프를 생성할 때 모든 프로시저의 제어 흐름 그래프를 보여주면 기본 블록의 수가 많아지므로 매우 복잡하다. 이러한 문제점을 해결하기 위해서는 사용자의 클릭에 따라 보고자 하는 프로시저의 제어 흐름 그래프만 나타내는 것이 필요하다. 이를 위해 하나의 프로시저의 제어 흐름 그래프를 모두 나타내지 않고 하나의 노드로 만드는 방법이 필요하다. 이러한 요구로 인해 프로시저의 왼쪽 구석의 +, -표시를 클릭하여 프로시저의 제어 흐름 그래프를 숨기거나 나타나게 하는 기능을 구현한다.

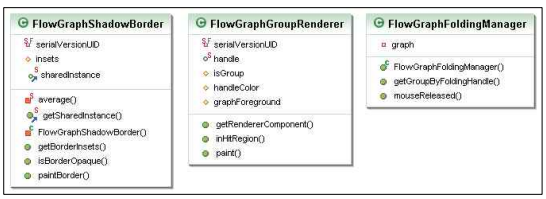

그림 6. 그룹화를 위해 필요한 클래스들
Fig. 6 Classes for grouping

그림6은 그룹화를 하기 위해 필요한 클래스들이다. FlowGraphShadowBorder 클래스는 하나의 프로시저의 경계를 알기 위해 점선으로 표시하는 역할을 하는 클래스이다. FlowGraph GroupRenderer 클래스는 그룹을 묶거나 펼칠 때 컴포넌트들을 그려주는 역할을 하는 클래스이다. FlowGraphFoldingManager 클래스는 실제 마우스 이벤트를 받아서 펼치고 접는 일을 수행하는 클래스이다.

2.4 그래프에서 실행 흐름

시각화된 그래프에서 실행 흐름을 찾기 위해서는 특정 기본 블록에서 특정 기본 블록까지의 경로찾기를 수행해야 한다. 이를 위해, 고려할 수 있는 알고리즘으로 Dijkstra의

Shortest Path 알고리즘이 있는데, 이 알고리즘은 가장 짧은 경로만을 찾기 때문에 특정 시작 블록에서 타겟 블록까지의 존재할 수 있는 모든 경로를 찾아야 하는 실행 흐름 분석에는 적합하지 않다. 따라서 KShortestPaths 알고리즘을 이용하여 가능한 K개의 경로들을 찾아낸다. 이 논문에서는 이를 위해 JGraphT 라이브러리에서 제공하는 KShortestPaths 알고리즘을 변형하여 이용한다.

```
Paths = ∅
PathsVector = ∅
For each node s ∈ N
   For each node d ∈ N - {s}
      Find the shortest path SP between s and d with
Dijkstra algorithm
      PathsVector = PathsVector ∪{SP}
      n = 1
      While (n < K) and (PathsVector ≠ ∅)
         Take the first path p of PathsVector
         PathsVector = PathsVector - {p}
         Paths = Paths ∪{p}
         Search L(p)
         While (L(p) ≠ ∅) and (n < K)
            Take the less-cost link l of L(p)
            L(p) = L(p) - {l}
            Remove the link l of the graph and search the
            new shortest path SPbetween s and d (Dijkstra)
            If SP'is found
                  PathsVector = PathsVector ∪ {SP'}
                  Paths = Paths ∪ {SP'}
                  n = (n +1)
            End If
            Reinsert l in the graph
         End While
      End While
   End For
End For
```

그림 7. KShortestPaths 알고리즘의 Pseudocode
Fig. 7 Pseudocode of KShortestPaths Algorithm

그림 7은 KShortestPaths 알고리즘의 pseudocode이다. Paths는 검색되는 n (n ≤ K) 개의 shortest paths 집합이고, pathsVector 는 가능한 경로들의 집합을 나타낸다. PathsVector의 각각의 p에 대해 L(p)는 p의 연결들의 집합을 의미한다.

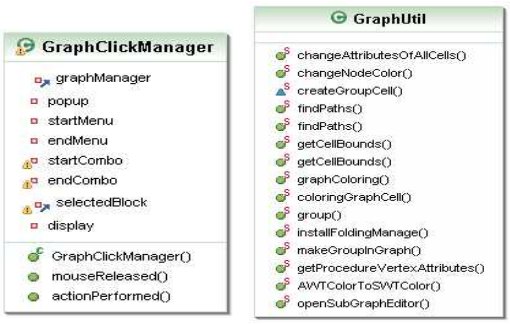

그림 8 GraphClickManager와 GraphUtil 클래스
Fig.8 GraphClickManager and GraphUtil classes

그림 8의 GraphClickManager 클래스는 그래프에서 시작 노드와 타겟 노드를 선택하기 위한 이벤트를 처리하는 클래스이다. GraphClickManager는 실제 그래프의 컨트롤러인 JGraph 클래스의 인스턴스에 설치된다. 그래프를 선택하거나 마우스 오른쪽 버튼을 클릭할 때 mouseReleased() 메소드에서 그래프의 선택과 관련된 일을 수행한다.

GraphUtil은 그래프에서 선택된 시작 노드와 타겟 노드를 이용해 KShortestPaths 알고리즘의 결과를 보여주고 서브 그래프를 그리는 GraphUtil 클래스를 나타낸다. GraphUtil 클래스는 노드의 색을 변경하기 위한 changeNodeColor() 메소드, 그래프에서 경로를 찾고 경로에 색을 표시하기 위한 findPaths() 메소드, 찾은 경로를 서브 그래프로 보이기 위한 openSubGraphEditor() 메소드 등을 포함한다. 찾아낸 경로들을 통해 그 경로에 포함되는 프로시저들만을 이용해 서브 그래프를 구성하고 그리게 되는데 이는 GraphUtil 클래스의 openSubGraphEditor() 메소드에서 수행되며 서브 그래프를 생성하는 클래스는 SubGraphFactory 클래스이다. SubGraphFactory의 getProcedureInPaths() 메소드에서 경로들을 이용해 그 경로에 포함되는 프로시저들을 구하고 createSubGraph() 메소드를 통해 구한 프로시저들을 이용해 제어 흐름에 따라 그래프를 생성한다. 이렇게 생성된 서브 그래프는 그림 9의 SubGraphEditor Input 클래스의 인스턴스를 통해 SubGraphEditor 클래스로 보여진다.

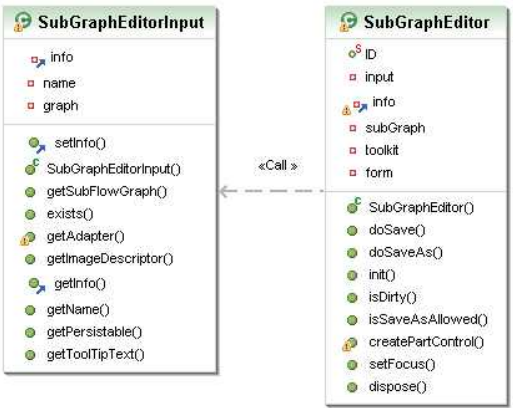

그림 9. SubGraphEditorInput과 SubGraphEditor
Fig. 9 SubGraphEditorInput and SubGraphEditor

IV. 구현 및 실험

1. Flow Graph Analyzer의 기능

그림 10의 caseTest.lst파일은 main 함수에서 switch, case, if, else 등의 분기문을 통해 whenTrue 함수나 whenFalse 함수를 호출하는 예제이다. whenTrue 함수내에서는 strcpy 함수와 printf 함수가 사용되며 whenFalse 함수 내에서는 printf 함수가 사용된 경우이다. 이 파일은 그

래프를 시각화하고 strcpy 함수까지의 경로를 찾는데 사용된다.

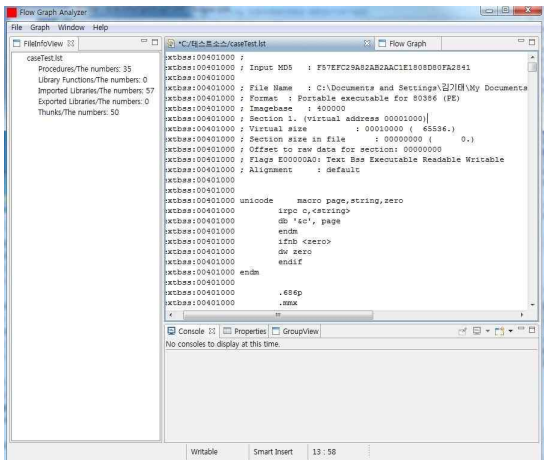

그림 10. caseTest.lst 예제
Fig. 10 Example of caseTest.lst

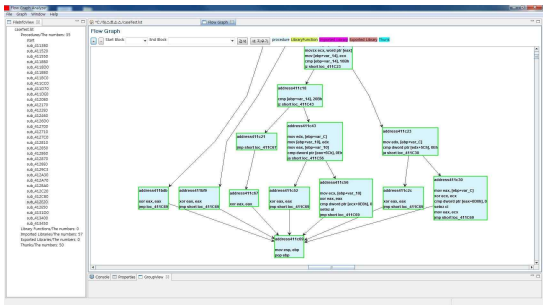

그림 11. 제어 흐름 그래프
Fig. 11 Control Flow Graph

생성된 프레임워크에서 열기 버튼을 클릭하면 그림 11과 같이 프로시저의 호출 그래프가 생성된다. 프로그램의 상단에는 메뉴가 나타나고 왼쪽에 파일 정보를 내타내는 FileInfoView에는 프로시저의 개수와 이름이 나열되어 있으며 오른쪽의 Call Graph 에디터에는 제어 흐름 그래프가 나타난다.

하나의 프로시저를 하나의 노드로 그룹핑해서 나타내기 위해서는 그림 12과 같이 프로시저 그래프의 왼쪽 구석의 - 버튼을 클릭하고 반대로 한 프로시저의 전체 제어 흐름 그래프를 나타내기 위해서는 하나의 프로시저 노드의 왼쪽 구석의 + 버튼을 클릭한다. 줌 기능을 이용하면 한눈에 전체적인 제어 흐름을 파악할 수 있다. IDA Pro의 경우에는 이러한 전체 그래프를 그리는 기능은 없으며 이를 대체하기 위해 call graph에 대한 그림이 제공된다. Flow Graph Analyzer에서는 그룹핑 기능을 잘 활용하면 제어 흐름을 파악하는데 더욱 효과적으로 이용할 수 있다.

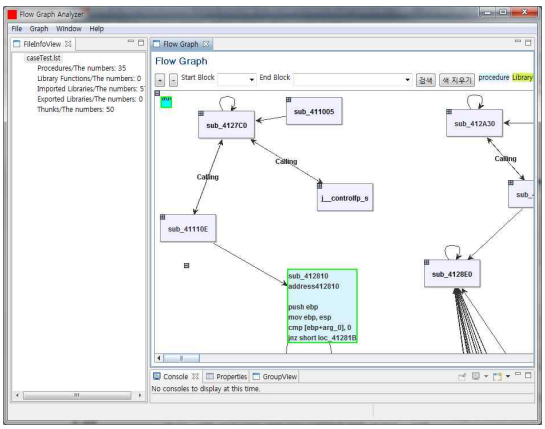

그림 12. 그룹화
Fig. 12 Grouping

그림 13은 특정 시작 위치에서 내부에 존재하는 정보를 복사 할 수 있는 기능을 가진 strcpy와 같이 위험할 수 있는 함수로의 경로가 존재하는지를 확인한다. 경로 찾기에 있어 중요한 점은 가장 짧은 경로 하나를 찾는 것이 아니라 존재하는 K 개의 경로를 찾는 것이다. 이론적으로는 인접 행렬을 이용하여 존재하는 모든 경로를 찾으면 되지만 생성된 노드의 개수가 많아지면 시간이 너무 많이 걸린다는 단점이 존재한다. 따라서 K 개의 경로를 알고리즘을 적용하여 찾아낸다.

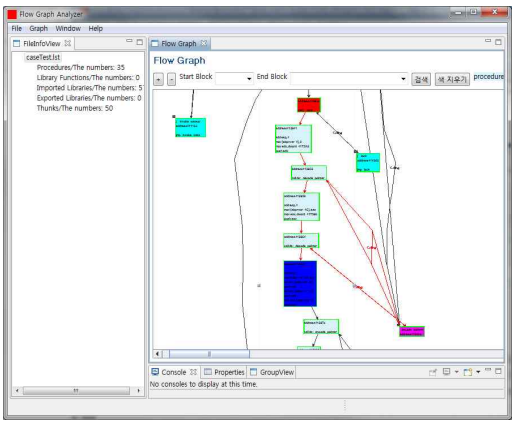

그림 13. 경로 탐색
Fig. 13 Path Search

2. 실험

Flow Graph Analyzer에 대한 실험 환경은 Interl(R) Core(TM) i7 CPU Q720 @1.60GHz와 4GB RAM을 사용하였고 Window 7 Home Premium K에서 수행하였다. Flow Graph Analyzer를 작성하기 위해 jdk1.0.0_21와 eclipse-jee-helios-SR1-win32를 사용하였다. Flow

Graph Analyzer에서 그래프 작성과 배치를 위해 JGraphT 0.7.3과 JGraph 5.9.2.0을 사용하고, .lst 파일 생성을 위해 IDA Pro 5.0.0.879 버전을 사용하였다.

표 1 변환 결과
Table. 1. Result of translate

프로그램	파일 크기	변환 시간	변경 후 크기
comctl32.dll	597KB	414sec	13.7MB
faad2	57.6KB	5.9sec	693KB
GDI32.dll	271KB	314sec	7.61MB
MPD	243KB	131.2sec	2.74MB
Mplyer	14.2KB	1sec	36.8KB
QEMU	341KB	54.6sec	12.8MB
Xine	966KB	590sec	12.9MB

표 1은 바이너리 파일을 Flow Graph Analyzer를 통해 변환 결과이다. 표 1에서 첫 번째 열은 사용된 프로그램을 나타낸다. 두 번째 파일 크기는 변환하기 전 바이너리 파일의 크기를 나타낸다. 세 번째 열은 IDA Pro를 통해 작성된 .lst 파일로부터 제어 흐름 그래프를 생성하는데 걸리는 변환 시간이다. 이 과정에서 많은 정보를 Flow Graph Analyzer가 직접 생성하고, 이를 바탕으로 그래프를 메모리에 직접 작성하기 때문에 시간이 많이 걸리는 결과를 보인다. 네 번째 열은 제어 흐름 그래프를 작성한 후의 파일 크기를 나타낸다. 결과처럼 파일의 크기가 상당히 커지게 된다. 이유는 JGraph를 통하여 비주얼한 정보를 생성하기 때문이다.

표 2 경로 탐색 결과
Table. 2. Result of Traversing

프로그램	진입함수	전체 경로	의심 경로	탐색 시간 (sec)
comctl32.dll	DSA_SetItem	3	2	0.1
faad2	decodeMP4file	36	3	8.3
GDI32.dll	CopyMetaFile	452	3	57.4
MPD	mp4_decode	2	1	0.1
Mplyer	_LoadPNG	4	1	0.3
QEMU	vmdk_open	20	2	25.8
Xine	ff_audio_decode_data	10	1	2.7

표 2는 Flow Graph Analyzer의 경로 탐색 결과이다. 첫 번째 열은 프로그램을 나타내고, 두 번째 열은 경로 탐색을 위해 사용한 진입함수 이름이다. 그림 15와 같이 필요에 따라서는 다양한 위치에서 결과를 확인할 수 있지만, 표 2는 각 프로그램의 대표적인 함수를 나타낸 것이다. 세 번째 열은 정해진 함수에서 위험한 함수라고 지정한 함수들까지의 모든 전체 경로를 나타낸다. 특히 GDI32.dll의 경우에는 내부적으로 strcpy를 많이 사용하고, 다른 CopyMetaFile 함수가 내부적으로 참조하는 함수들이 많아 상대적으로 많은 수의 경로가 나타나고 있다. 네 번째 열은 전체 경로 중에 위험할 수 있는 경로의 수를 나타내고 있다. 이 경로들을 추적하면 위험성에 대한 분석을 용이하게 할 수 있게 된다. 마지막 열은 진입함수로부터 위험성을 가질 수 있는 함수들까지 탐색하는데 걸리는 시간을 나타낸다. 전체 경로의 수가 많고 확인해야 하는 위험 함수가 많은 경우 상대적으로 많은 시간이 걸리는 것을 확인할 수 있다.

표 3 기존 시스템과 비교
Table. 3. Compared with the existing system

	IDA Pro	CodeSuffer/ x86	Flow Graph Analysis
IDA Pro 사용	O	O	O
CFG 작성	O	O	O
Call Graph	O	O	O
Group 기능	O		O
Zoom 기능			O
경로탐색 기능			O

표3은 기존 시스템들과의 비교이다. 우선 바이너리 파일을 위해 내부적으로 IDA Pro를 사용하는지 여부이다. 세 가지 시스템 모두 내부적으로 IDA Pro를 사용하며, 각각 CFG와 Call Graph를 작성하는 것을 확인할 수 있다. IDA Pro의 경우는 간단한 형태의 제어 흐름 그래프를 보여주며, Call Graph의 경우엔 단순한 이미지 형태로 제공한다. 따라서 실제 호출 관계를 이해하기 어렵다는 단점이 존재한다. CodeSurfer에서는 그래프 자체 보다는 분석을 위해 시스템 의존 그래프(SDG)를 작성하여 분석을 위해 사용한다. 반면 본 논문에서 제안한 Flow Graph Analysis의 경우엔 상대적로 자세하며 다양한 기능을 그래프 형태로 제공한다. 특히 추가적으로 Group, Zoom, 경로 탐색 기능이 제공된다. Group과 Zoom 기능으로 전체적인 구조와 경로에 대한 요약 그리고 필요에 따라서는 세부적인 내용을 직접확인 할 수 있다는 장점을 가진다.

V. 결 론

이 논문에서는 특정 실행 파일이 주어지면, 우선 IDA Pro를 통해 역어셈블과 정적 라이브러리 함수 찾기 과정을 수행하였고, 제어 흐름 그래프를 생성하였다. 이를 통해 실행 파일의 흐름 정보를 볼 수 있다. 또한 경로를 찾기 위해 KShortestPaths 알고리즘을 이용하여 특정 프로시저나 주소까지의 경로를 쉽게 탐색하고 파악할 수 있었다. 또한 효율적인 프레임워크를 위해 그룹핑, 탐색, 컬러링, 속성창 등을 제공하였다. 그러나 현재 개발된 프레임워크는 분석하는 파일의 크기가 커지고 처리해야 그래프의 노드의 양이 많아지면

상대적으로 속도가 매우 느려지는 단점이 발생하였다. 이는 그래프 생성에 기인하는 문제이며, 이러한 문제점을 해결하기 위한 성능 향상에 대한 향후 연구가 필요하다.

향후 추진되어야 할 연구로는 보다 정확한 분석을 위해 제어 흐름 분석뿐만 아니라 데이터 흐름 분석이 이루어져야 한다. 이 논문의 제어 흐름 분석만으로도 의미 있는 결과를 얻어낼 수 있지만 데이터로 인한 흐름의 영향까지는 현재 고려하지 않았다. 그러므로 흐름에 영향을 미치는 데이터의 분석까지 포함하는 데이터 흐름 분석을 추가하면 보다 정확한 분석이 가능해진다. 그렇지만 데이터 흐름 분석은 매우 복잡하고 많은 메모리 공간을 소모하므로 많은 노력이 요구된다.

참고문헌

[1] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. "Static Detection of Vulnerabilities in x86 Executables", In Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2006.

[2] Byoungyoung Lee , Yuna Kim , Jong Kim, binOb+: a framework for potent and stealthy binary obfuscation, Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, April 13-16, 2010, Beijing, China

[3] C. Kruegel , W. Robertson , F. Valeur , G. Vigna, "Static disassembly of obfuscated binaries", Proceedings of the 13th conference on USENIX Security Symposium, pp.18-18, 2004.

[4] Gogul Balakrishnan , Thomas Reps, WYSINWYX: What you see is not what you eXecute, ACM Transactions on Programming Languages and Systems (TOPLAS), v.32 n.6, p.1-84, August 2010

[5] E. Carrera et al, "Digital Genome Mapping : Advanced Binary Malware Analysis", Proceedings of 15th Virus Bulletin International Conference, pp.187-197, 2004.

[6] IDA Pro. http://www.datarescue.com/

[7] Yeong-Tae Baek, Ki-Tae Kim, "Framework for Static Control Flow Analysis of Binary Codes," Proc. of The Korea Society of Computer and Information, Vol. 18, No. 2, pp. 67-70, Jul. 2010.

[8] M. D. Ernst. "Static and Dynamic Analysis: Synergy and Duality", In WODA 2003: ICSE Workshop on Dynamic Analysis, Portland, OR, May 9, 2003.

[9] Denis Gopan , Thomas Reps, Low-level library analysis and summarization, Proceedings of the 19th international conference on Computer aided verification, July 03-07, 2007, Berlin, Germany

저 자 소 개

백 영 태(Baek, Yeong Tae)

1993년: 인하대학교 전자계산공학과 석사

2002년: 인하대학교 전자계산공학과 박사

1993년-1998년: 대상정보기술 정보통신연구소

1998년 - 현재 : 김포대학 부교수

관심분야 : 웹교육시스템, 프로그래밍언어

E-mail : hannae@kimpo.ac.kr

김 기 태(Kim, Ki Tae)

2001년: 인하대학교 전자계산공학과 석사

2008년 : 인하대학교 정보공학과 박사

2008년-2010년: 인하대학교 강의전임강사

2010년 - 현재 : (주)나루기술 선임연구원

관심분야 : 컴파일러, 프로그래밍언어

E-mail : aqua0405@gmail.com

전 상 표(Jun, Sang Pyo)

1987년 : 인하대학교 수학과 석사

2000년 : 인하대학교 통계학과 박사

현재 : 남서울대학교 교양학부 교수

관심분야 : 컴퓨터활용 수학교육, 프로세스 분석

E-mail : spjun7129@dreamwiz.com