한국컴퓨터정보학회 하계학술대회 논문집 제18권 제2호 (2010. 7)

바이너리 코드의 정적 제어 흐름 분석을 위한 프레임워크

백영태°, 김기태**
*김포대학 멀티미디어과
**인하대학교 컴퓨터공학부
e-mail : hannae@kimpo.ac.kr, kimkitae@inha.ac.kr

Framework for Static Control Flow Analysis of Binary Codes

Yeong-Tae Baek°, Ki-Tae Kim**
*Dept. of Multimedia, Kimpo College
**Dept. of Computer Science, Inha University

● 요 약 ●

본 논문은 바이너리 코드 수준에서 정적인 프로그램 분석을 수행하는 프레임워크를 설계 및 구현한다. 정적으로 바이너리 코드 수준에서 분석을 수행하려는 이유는 일반적으로 컴퓨터에 설치되는 실행 파일은 소스 코드 없이 단지 바이너리로 된 실행 파일만 주어지는 경우가 대부분이고, 정적 제어 흐름 분석을 통해 수행 전에 동작을 파악하기 위해서이다. 본 논문에서는 바이너리 실행 파일로부터 실행 순서 및 제어 흐름 등의 정보를 표현할 수 있는 제어 흐름 그래프를 작성하여 바이너리 파일의 실행 흐름과 위험한 함수의 호출 여부를 동시에 파악할 수 있도록 하며, 그래프 시각화를 통해 바이너리 파일의 분석을 용이하게 한다. 또한 실행 흐름에 대한 자동 탐색 방법을 제공한다.

키워드: 이진 코드(binary code), 제어 흐름 분석(control flow analysis), 프레임워크(framework)

I. 서론

일반적으로 바이너리 실행 코드는 소스 코드가 컴파일되어 생성되는데, 컴파일 과정을 수행하는 컴파일러와 컴파일이 수행되는 환경에 따라 다른 머신 코드가 생성된다. 따라서 바이너리 코드의 동작은 소스 코드가 제공되지 않는 경우에는 프로그램의 의도를 알 수 없는 경우가 발생한다. 이러한 문제를 해결하기 위해서는 바이너리 코드 수준에서 프로그램 분석이 요구된다[1,2]

현재 국내에는 바이너리 코드에 대한 분석 기술이 많이 부족한 상태이다. 하지만 현재 인터넷을 통해 검증되지 않은 실행 파일을 내려 받아 컴퓨터에 설치하여 사용하고 있으며, 외부에서 제공되는 API를 검증없이 사용하여 프로그램을 개발하고 있다. 사실 이러한 실행 파일이나 API들이 컴퓨터 내에서 어떠한 동작을 하는지는 보통의 경우 알 수 없는 경우가 대부분이다. 따라서 바이너리 코드에 대한 제어 흐름 분석 기술을 통해 프로그램 분석 분야와 보안 분야에 많이 활용할 수 있게 된다[3]. 그리고 정적인 제어 흐름 분석을 통해 프로그램이 수행 시 어떤 동작을 수행하게 될지를 수행 전에 파악할 수 있기 때문에 안전한 소프트웨어 개발을 가능하게 한다.

프로그램의 행동이나 동작을 자동으로 분석하는 기술인 프로그램 분석 기술을 이용하여 정적으로 바이너리 파일의 실행 흐름을 분석한다. 최근 이러한 프로그램 분석 기술은 프로그래밍 언어나 소프트웨어 공학, 그리고 컴퓨터 보안 분야에 많이 적용되고 있다. 특히 컴퓨터 보안 분야에서는 프로그램의 버그나 취약점을 찾기 위해 많이 적용되고 있다.

소스 코드를 구하기 힘든 상황에서는 실행 파일로 제공된 바이너리 코드를 분석하여 역어셈블하고, 제어 흐름 그래프를 작성하여, 정적으로 실행 흐름을 분석할 수 있는 자동 도구가 요구된다[4]. 본 연구에서는 개발된 바이너리 코드 정적 분석 프레임워크를 통해 바이너리 실행 파일로부터 함수간의 실행 순서 및 제어 흐름 등의 정보가 표현되는 실행 흐름 그래프를 작성하여 사용자가 바이너리 파일의 실행 흐름과 위험한 함수의 호출 여부를 동시에 파악할 수 있도록 하여 바이너리 파일의 분석을 용이하게 한다. 또한 실행 흐름에 대한 자동 탐색 방법을 제공하여 수행될 프로그램의 안전성을 보장하고, 수행 전에 외부에서 다운받아 설치할 프로그램이 안전한지를 판단할 수 있도록 한다.

본 논문의 구성은 다음과 같다. 2장에서는 정적 분석과 바이너리 코드에 대한 분석에 대한 관련 연구를 설명하고, 3장에서는 바이너리 코드 분석을 위한 프레임워크를 설계한다. 4장에서는 예제를 통해 실제 구현된 도구들을 보여주고, 5장에서는 결론 및 향후 연구계획에 대해 설명한다.

II. 관련 연구

1. 정적분석

프로그램 분석은 분석이 수행되는 시점에 따라 정적 분석과 동적 분석으로 구분할 수 있다. 프로그램을 실행하기 전 수행하는 정적 프로그램 분석이 있고, 프로그램을 실행하며 수행하는 동적 프로그램 분석으로 구분한다[5].

동적 프로그램 분석(Dynamic Program Analysis)은 프로그램이 실행 중에 가질 수 있는 성질이나 행동을 실행 중에 자동으로 분석하는 방법이다. 실행 시간에만 정확히 알 수 있는 메모리에 대한 사용이나 사용자의 입력에 대해 분석을 수행하여 정확한 분석이 수행될 수 있다. 그러나 동적 프로그램 분석은 실제 프로그램을 실행하여 수행되므로 실행 시간에 안전성이 중요하게 생각되는 프로그램에 적용하기에는 위험한 경향이 있다.

반면, 정적 프로그램 분석(Static Program Analysis)은 프로그램이 실행 중에 가지는 성질을 "실행 전에 자동으로 안전하게 어림잡는 방법"이다. 컴파일 시간에 프로그램이 분석되므로 컴파일 시간 분석이라고도 한다. '실행 전'은 프로그램을 실행시키지 않고 분석한다는 뜻이고, 자동으로는 사람이 손으로 분석하는 것이 아니라 프로그램이 분석을 해 준다는 뜻이다. '안전하게'란 프로그램이 실행 중에 가지는 모든 상황을 빠짐없이 고려한다는 뜻이다. '어림잡는다'는 것은 정확하게 할 수 없으므로 대략 어림잡는다는 뜻이다. 정적인 방법은 프로그램을 실행하여 실행시간에 프로그램의 취약점을 찾아내는 방법에 비해 안전하고 프로그램의 모든 실행을 분석할 수 있으므로 동적 분석보다 강력하다는 특징을 가진다.

2. 바이너리 코드 분석

프로그램의 분석은 실행 전과 실행 후 뿐만 아니라 소스 코드 수준에서 할지 아니면 바이너리 코드 수준에서 수행할지도 고려해야 한다. 소스 코드 수준에서의 분석은 분석 후 취약점이 발생했을 때 프로그램의 수정이 용이하다는 장점이 있지만 다음과 같은 몇 가지 단점이 존재한다. 소스코드 수준의 분석에서는 컴파일 과정 중 포함되는 정적 라이브러리나 프로그램이 실행될 때 동적으로 호출하는 DLL 등에 대해서는 고려할 수가 없다. 또한 컴파일러의 최적화 과정 중 머신에 따라 코드가 변형되는 경우나 여러 프로그램 언어로 작성된 프로그램일 경우에도 문제가 생긴다. 웜이나 바이러스 같은 코드를 분석하는 경우에도 소스 코드가 없는 경우가 있기 때문에 소스 코드 수준의 분석이 불가능하다. 반면, 바이너리 수준의 분석은 머신의 플랫폼에 의존적인 사항을 다룰 수 있게 한다. 즉, 메모리 레이아웃, 레지스터의 사용 방법, 실행 순서 등에 대한 정보도 다룰 수 있다. 인라인 어셈블리가 포함된 코드도 분석할 수 있다는 장점이 있다.

III. 프레임워크 설계

분석을 수행하는 프레임워크는 그림 1과 같다.

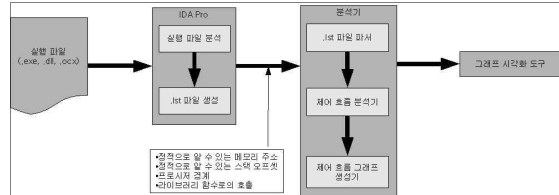

그림 1. 프레임워크 개요
Fig. 1. Frame Overview

그림 1은 프레임워크의 전체적인 구성을 나타낸다. 프레임워크는 크게 IDA Pro, 분석기, 그리고 그래프 시각화 도구 이렇게 세 부분으로 나눈다.

첫 번째 부분인 IDA Pro는 Hex-rays에서 개발한 상용 역어셈블러로 라이브러리 함수 인식, 디버깅 등의 기능을 제공하여 바이너리 코드 분석에 많이 이용되고 있다. 본 논문에서는 상용도구인 IDA Pro를 이용해 .exe, .dll, .ocx와 같은 확장자를 가진 실행 파일들을 분석하고 역어셈블한다. 이 과정을 통해 IDA Pro로부터 정적으로 알 수 있는 메모리의 주소와 스택 오프셋의 정보를 얻는다. 또한, 프로시저 경계에 관한 정보와 특정 라이브러리 함수를 인식하고, 그 함수로의 호출에 관련된 정보도 얻는다. 획득한 정보를 .lst 확장자를 갖는 리스트 파일로 생성한다. .lst 파일은 그래프 생성을 위해 필요한 정보들을 갖고 있기 때문에, 다음 과정인 분석기의 입력으로 쓰인다.

두 번째 부분인 분석기는 입력으로 들어오는 .lst 파일을 분석하는 과정을 나타낸다. 앞 과정에서 생성된 .lst 파일을 이용해 그래프를 만들기 위한 정보를 추출하기 위해서 분석기에서는 먼저 .lst 파일을 파싱한다. .lst 파일을 파싱하면서 각 명령어의 주소와 프로시저의 경계 정보를 통해 프로시저를 구분하고 명령어의 분기 정보, 호출 정보, 그리고 라이브러리 함수에 관한 정보 등을 획득한다. 제어 흐름 분석기는 파싱을 통해 얻은 정보와 기본 블록 구분 기준에 따라 제어 흐름 그래프를 생성하기 위한 준비를 한다. 제어 흐름 그래프 생성기는 제어 흐름에 따라 제어 흐름 그래프를 구성하는 역할을 수행한다.

마지막 부분은 그래프 시각화 도구이다. 분석기 부분을 통해 구성된 제어 흐름 그래프는 그래프 시각화 도구를 통해 사용자의 요구에 따라 원하는 정보를 보여준다. 효율적인 프레임워크를 위해 전체 흐름을 파악할 수 있는 그룹핑, 위험 경로에 대한 탐색, 컬러링, 속성창 등을 제공한다.

IV. 프레임워크 적용 사례

```
#include <stdio.h>
#include <string.h>
void whenTrue( char *buf ){
        char targetBuf[200];
        strcpy(targetBuf,buf);
        printf("Input Value is
%s₩n",targetBuf);
}

void whenFalse(){
        printf("in False");
}

int main(void){
        bool check = true
        char buf[2000];
        scanf("%s",buf);
        if(check) whenTrue(buf);
        else      whenFalse();
        printf("checking is completed");
        return 0;
}
```

그림 2. TestStrcpy.c 예제
Fig. 2. TestStrcpy.c

그림 2의 TestStrcpy.c파일은 C언어로 작성되어 있고, main 함수에서 if, else의 분기문을 통해 whenTrue 함수나 whenFalse 함수를 호출한다. whenTrue 함수내에서는 strcpy 함수와 printf 함수가 사용되며 whenFalse 함수내에서는 printf 함수가 사용된다. 이 소스 파일은 그래프를 시각화하고 strcpy 함수까지의 경로를 찾는데 사용되는 예제 파일이다.

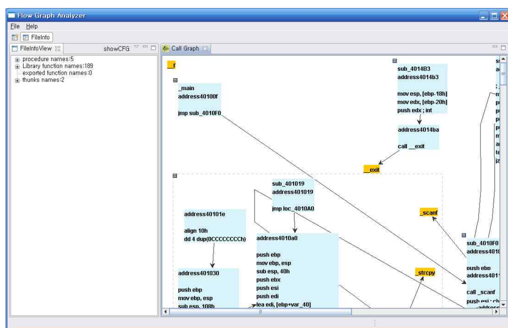

그림 3. 제어 흐름 그래프
Fig. 3. Control Flow Graph

생성된 프레임워크에서 열기 버튼을 클릭하면 그림 3과 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 4과 같이 프로시저 그래프의 왼쪽 구석의 - 버튼을 클릭하고 반대로 하나의 프로시저의 전체 제어 흐름 그래프를 나타내기 위해서는 하나의 프로시저 노드의 왼쪽 구석의 + 버튼을 클릭한다.

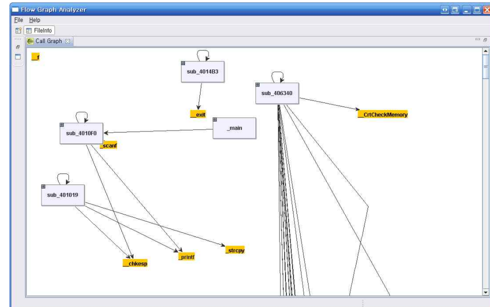

그림 4. 그룹화
Fig. 4. Grouping

그 밖에 원하는 프로시저의 제어 흐름 그래프만 한 화면에 보기 위해서는 원하는 프로시저에서 마우스 오른쪽 버튼을 눌러 팝업 메뉴의 show CFG를 클릭하면 그림 5와 같은 한 프로시저의 제어 흐름 그래프를 보일 수 있다.

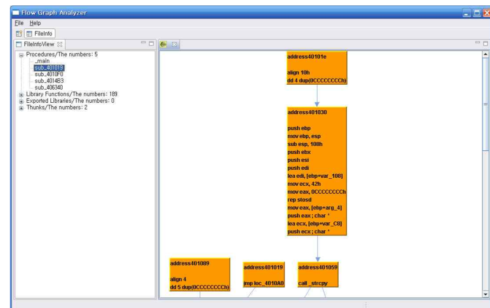

그림 5. show CFG 수행 후
Fig. 5. After show CFG

그림 5와 같이 한 프로시저의 내용을 분석할 수도 있고 그림 6과 같이 줌 기능을 통해 전체흐름을 파악할 수도 있다.

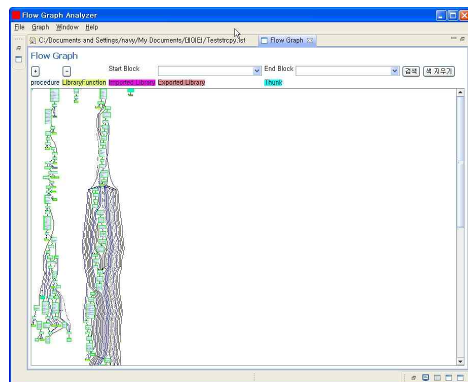

그림 6. 줌
Fig. 6. Zoom

그림 6과 같이 줌 기능도 추가하여 전체적인 제어 흐름을 파악할 수 있다. 이 경우 그룹핑 기능을 잘 활용하면 제어 흐름을 파악하는데 더욱 효과적이다.

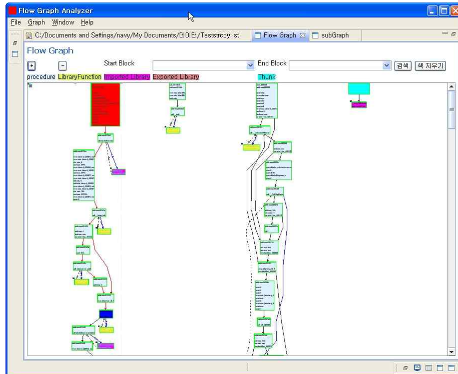

그림 7. 경로 탐색
Fig. 7. Path Search

그림 7은 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 모든 경로를 찾는 것이다. 이론적으로는 인접 행렬을 이용하여 존재하는 모든 경로를 찾으면 되지만 생성된 노드의 개 수가 많아지면 시간이 너무 많이 걸린다는 단점이 존재하였다. 따라서 KShortestPaths 알고리즘을 적용하여 가능한 경로를 찾는다.

V. 결론

본 논문에서는 특정 실행 파일이 주어졌을 때, 우선 IDA Pro를 통해 역어셈블과 정적 라이브러리 함수 찾기를 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 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.

참고문헌

[1] 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.

[2] A. Lanzi , L. Martignoni , M. Monga , R. Paleari, “A Smart Fuzzer for x86 Executables”, Proceedings of the Third International Workshop on Software Engineering for Secure Systems, pp.1-7, 2007.

[3] 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.

[4] B. Schwarz, S. K. Debray, and G. R. Andrews. “Disassembly of Executable Code Revisited, In 9th Working Conference on Reverse Engineering, pp. 45–54, IEEE Computer Society, 2002.

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