바이너리 코드-SIL 중간언어 변환 검증을 위한 시각화 도구 구현

임정호*, 이태규*, 백도우*, 손윤식**, 정준호***, 최진영*, 고광만****, 오세만**

*고려대학교 사이버국방학과
**동국대학교 컴퓨터공학과
***동국대학교 경주캠퍼스 전자성거래 연구소
*****상지대학교 컴퓨터정보공학부
e-mail: smoh@dongguk.edu

A Visualization Tool Implementation for Evaluation of Binary Code to Smart Intermediate Language Conversion

Do-Woo Baik*, Tae-Gue Lee**, Jung-Ho Lim*, Yunsik Son**, Junho Jeong***, Jin-Young Choi*, Kwangman Ko****, Seman Oh**
*Dept of Cyber Defense, Korea University
**Dept of Computer Engineering, Dongguk University
***Electronic Commerce Institute, Dongguk University Gyeongju Campus
****Dept. of Computer and Information Engineering, SangJi University

요 약

최근 소프트웨어에 내장된 취약점 분석을 위한 자동화 도구 개발 연구가 각 분야에서 활발히 연구되고 있다. 그 중 바이너리 코드를 대상으로 바로 보안취약점을 분석하는 방법이 아닌 중간언어를 활용하여 분석하는 방법이 대두되고 있으며 이를 위한 다양한 중간언어가 제시되었다. 그 중 하이레벨 언어 수준의 내용의 기술이 가능하며 명령어 자체적으로 자료형을 유지하여 보안 취약점 분석에 효과적인 언어로 SIL 중간언어가 재조명 받고 있다. 따라서 본 논문에서는 이를 위해서 x86/64 기반 어셈블리어를 SIL 로 효과적으로 변환하며 프로그램의 의미가 변하지 않는 것을 확인하기 위해서 프로그램의 제어흐름을 시각화하는 기능을 가진 시스템을 제안한다.

1. 서론

최근 소프트웨어에 내장된 취약점 분석을 위한 자동화 도구 개발 연구가 각 분야에서 활발히 연구되고 있다. 전통적인 보안약점 및 취약점 분석은 소프트웨어의 소스코드를 기반으로 분석이 이루어지는 것이 일반적이었으나 최근에는 바이너리 코드를 대상으로 보안약점 및 취약점을 분석하고 하는 요구가 증가하고 있다.[1]

이는 소프트웨어 개발시 외부에서 제작된 라이브러리의 사용과 밀접한 연관이 있는데 외부에서 제작된 라이브러리의 경우 소스코드가 공개되지 않고 바이너리의 형태로 제공되는 것이 일반적이기 때문이다.

따라서 바이너리 코드를 대상으로 보안취약점을 분석하기 위한 다양한 연구가 진행되고 있다.[2-5] 그 중 바이너리 코드를 대상으로 바로 보안취약점을 분석하는 방법이 아닌 중간언어를 활용하여 분석하는 방법이 대두되고 있으며 이를 위한 다양한 중간언어가 제시되었다. 그 중 하이레벨 언어 수준의 내용의 기술이 가능하며 명령어 자체적으로 자료형을 유지하여 보안 취약점 분석에 효과적인 언어로 SIL 중간언어가 재조명 받고 있다.[6]

하지만 기존의 SIL 중간언어는 소스코드를 기반으로 중간언어로 변환은 가능하였으나 바이너리 코드를 중간언어로 변환하는 것은 불가능하였다. 따라서 윈도우 환경에서 C/C++로 생성된 대부분의 dll 파일과 실행 파일과 같은 바이너리 코드로부터 보안약점을 분석하기 위해서는 바이너리 코드를 x86/64 어셈블리어로 변환하고 이 어셈블리를 SIL로 변환할 필요가 있다.

따라서 본 논문에서는 이를 위해서 x86/64 기반 어셈블리어를 SIL 로 효과적으로 변환하며 프로그램의 의미가 변하지 않는 것을 확인하기 위해서 프로그램의 제어흐름을 시각화 해주는 기능을 가진 프로그램을 개발하였다. 본 논문의 구성은 다음과 같다. 2 장에서 제안 시스템을 소개하고 3 장에서 그 결과를 분석하며 마지막으로 결론을 맺고 마친다.

"본 연구는 방위사업청과 국방과학연구소 (계약번호 UD160035ED)의 연구비 지원에 의한 연구 결과임"

2. 바이너리 코드-SIL 중간언어 변환 시각화 도구

본 연구에서 제안하는 시각화 도구는 C/C++로 작성된 바이너리 코드를 SIL 중간언어로 변환하며 해당 코드의 시맨틱을 잃지 않고 변환이 이루어지는지 효과적으로 확인하기 위해서 제어흐름을 시각화 하는 도구이다. dll 파일로 빌드하고, 그것을 SIL 중간언어로 변환하여, 그 변환된 중간언어에서 취약점을 찾아내기 위해 사전작업으로 각 변수들의 타입을 1 차 추론한다. 그 과정은 아래의 (그림1) 과 같이 진행이 이루어진다.

1) C/C++ 고급언어로 작성된 소스코드로부터 PE 구조의 바이너리 코드를 생성한다.
2) dll 파일의 PE 구조를 파싱하여 각 함수를 추출하고, 추출된 함수에서 Normal Block을 구성하여 제어흐름 그래프(CFG/Control Flow Graph)를 추출한다.
3) CFG의 각각의 Normal Block의 x86 Assembly를 SIL 코드로 변환한다.

(그림 1) 바이너리 코드로부터 중간언어 변환 시각화 과정

2.1 제안 시스템 세부 수행

제안 시스템은 크게 네 단계로 진행되며 그 첫 번째는 대상이 되는 파일의 입력부터 시작한다. 그 입력은 C/C++로 작성된 PE 형태의 바이너리 코드로 가정하며 C 언어로 만들어진 바이너리 파일과 C++ 언어로 만들어진 파일의 구조가 다르기 때문에, 이 둘을 구분하여 처리하기 위한 전처리가 1 차 수행된다.

두 번째는 입력받은 파일의 구조에 따라 바이너리 코드를 파싱하여 코드로부터 export 하는 함수들을 추출한다. 함수들은 각각 어셈블리의 형태로 추출되며, 메모리에 저장되어 함수에서 함수로 전달될 뿐 별도의 파일로 저장은 하지 않는다.

세 번째는 파싱된 파일을 바탕으로 제어흐름을 분석하여 그래프화하는 것이다. 즉, CFG 를 작성하는 과정으로 프로그램을 실행하면서 통과 할 수 있는 모든 경로를 그래프 표기법을 사용하여 표현한 형태를 말한다. 어셈블리 명령어와 SIL 명령어는 수행되는 기계의 차이가 있으므로 이를 해결하여 효과적인 언어 변환을 위해서 어셈블리 명령어를 통해 CFG를 사전 작성하는 것이 중요하다. 분기문(branch)을 기준으로 분석하여 CFG를 형성하며 완성된 CFG는 분기문을 기준으로 블록과 간선을 구분하여 JSON 형태로 출력하고 이를 Dagre 오픈소스[7]를 활용하여 제작한 시각화 도구를 통해 확인할 수 있다.

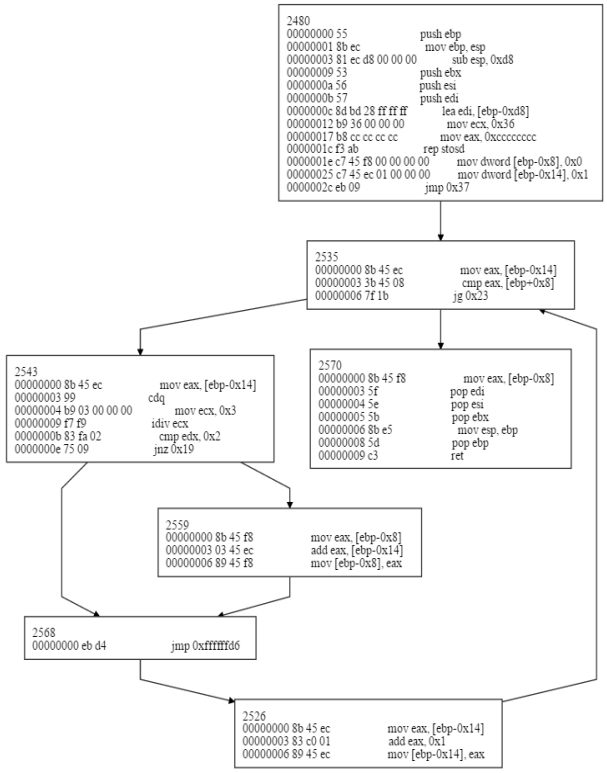
마지막으로 어셈블리 명령어를 기반으로 작성된 CFG로부터 기본적인 SIL 언어로 변환이다. 어셈블리 명령어와 SIL 명령어의 근본적인 차이점인 flag와 분기문을 해결하기 위해 분기문을 기준으로 블록으로 나누었으며 이 나누어진 블록을 입력받아 한 블록을 SIL로 변환하고 이 변환된 블록을 다시 연결하여 CFG를 만들어 출력하는 작업을 반복하며 그 출력의 형식은 json을 활용한 동일한 형태로 표현된다.

3. 실험 결과 분석

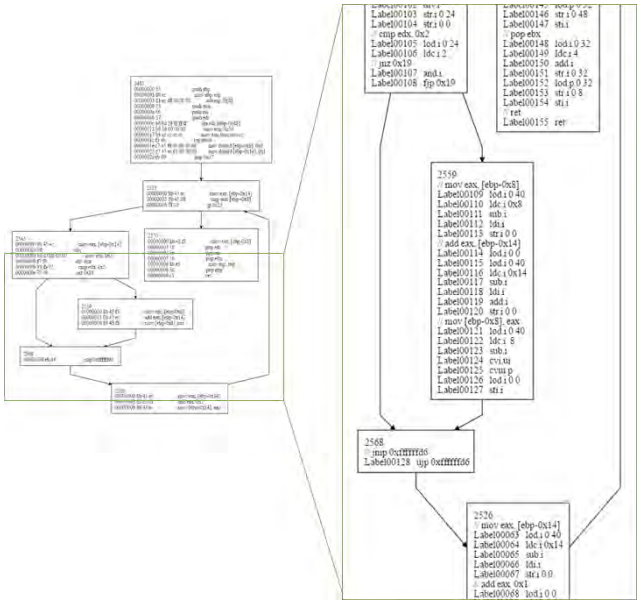
(그림 2) 는 제안시스템을 테스트하기 위해 사용된 많은 테스트 케이스 중 하나이며 CFG를 생성하기 위한 프로그램의 예제이다. 이를 통해 생성된 어셈블리 코드 CFG는 (그림 3) 과 같고, 실제 프로그램의 제어흐름을 반영한 CFG가 작성된 것을 확인 할 수 있다. (그림 4)는 이를 바탕으로 작성된 SIL 코드 CFG의 일부분이다. 동일한 형태의 CFG가 작성되며 각 어셈블리 동작에 대해서 그에 대응되는 SIL 명령어로 변환됨을 확인할 수 있다.

```
DLL_EXPORTING_API int DLL6(int n)
{
    int sum = 0;
    int i;
    for (i = 1; i <= n; i++)
    {
        if (i % 3 == 2)
            sum += i;
    }
    return sum;
}
```

(그림 2) 분석대상 프로그램



(그림 3) 어셈블리 코드 CFG



(그림 4) SIL 코드 CFG

4. 결론

우리는 바이너리 코드를 SIL 중간언어로 변환하는데 있어 그 의미가 훼손되지 않는지를 효과적으로 검증하기 위해서 CFG를 작성하는 시각화 도구를 구현하였다. 향후 연구로 자료흐름그래프(Data Flow Graph) 를 구현하여 중간언어 변환에 대한 효과적인 검증을 추가하고자 한다.

참고문헌

[1] GRAMMATECH, “Eliminating Vulnerabilities in Third-party Code with Binary Analysis”, WhitePaper http://www.grammatech.com/products/codesonar

[2] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., & Vigna, G. (2015). Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In NDSS.

[3] Feist, J., Mounier, L., & Potet, M. L. (2014). Statically detecting use after free on binary code. Journal of Computer Virology and Hacking Techniques, 10(3), 211-217.

[4] Zhang, B., Wu, B., Feng, C., Zhang, X., & Tang, C. (2015, December). Statically detect invalid pointer dereference vulnerabilities in binary software. In Progress in Informatics and Computing (PIC), 2015 IEEE International Conference on (pp. 390-394). IEEE.

[5] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., ... & Saxena, P. (2008, December). BitBlaze: A new approach to computer security via binary analysis. In International Conference on Information Systems Security (pp. 1-25). Springer Berlin Heidelberg.

[6] Son, Y., & Lee, Y. (2014). Smart Virtual Machine Code based Compilers for Supporting Multi Programming Languages in Smart Cross Platform. International Journal of Software Engineering and Its Applications, 8(5), 249-260.

[7] https://github.com/cpettitt/dagre