어셈블리어 코드 기반의 Invalid Function Pointer Access Error 가능성 검출

김현수* · 김병만*

*금오공과대학교

Detection of Potential Invalid Function Pointer Access Error based on Assembly Codes

HyunSoo Kim* · Byeong Man Kim*

*Kumoh National Institute of Technology

E-mail : kim.hyunsoo@se.kumoh.ac.kr, bmkim@kumoh.ac.kr

요 약

일반적으로 컴파일러가 프로그램 번역시 메모리 사용 오류에 대한 검사도 병행하지만, 코드 레벨에서는 검사가 불가능한 함수 포인터는 정상적인 검사가 매우 힘들다. 이에 본 논문에서는 실행 프로그램을 역어셈블하여 만들어진 어셈블리 언어 프로그램을 구문 분석하여 함수 포인터 사용의 형태(패턴)를 어셈블리 명령어 전이도를 기반으로 "Invalid Function Pointer Access Error"에 대한 오류 가능성을 검출한다. 검사대상인 3개 프로그램은 약 10,000개의 함수와 1,000,000 개의 어셈블리 명령어로 구성되어 있으며, 본 논문에서 제안한 방법을 사용하여 함수 포인터의 사용 오류를 검출한 결과 1,100개의 함수 포인터 사용 중 약 500개의 비정상적 함수 포인터의 사용을 검출하였으며 검출에 걸린 시간은 총 82초 정도가 소요되었다.

ABSTRACT

Though a compiler checks memory errors, it is difficult for the compiler to detect function pointer errors in code level. Thus, in this paper, we propose a method for effectively detecting Invalid function pointer access errors, by analyzing assembly codes that are obtained by disassembling an executable file. To detect the errors, assembly codes in disassembled files are checked out based on the instruction transition diagrams which are constructed through analyzing normal usage patterns of function pointer access. When applying the proposed method to various programs having no compilation error, a total of about 500 potential errors including the ones of well-known open source programs such as Apache web server and PHP script interpreter are detected among 1 million lines of assembly codes corresponding to a total of about 10 thousand functions.

키워드

함수 포인터 오류 검출, 정적 프로그램 분석, 상태 전이도, 어셈블리어 기반 분석

Ⅰ. 서 론

일반적으로 프로그램 개발 시 함수를 많이 사용하고, 특히 로직상의 이유와 편의적 측면에서 함수 포인터를 사용하는 경우도 있다. 하지만 함수의 주소가 저장되지 않은 포인터를 호출할 경우 프로그램의 오작동이 발생한다. 이러한 오류를 소스 코드 기반으로 검출할 경우 각 개발자들의 코딩 스타일이 달라 고려해야 하는 경우의 수가 많아진다. 하지만 역어셈블을 통해 만들어진 어셈블리어의 구문을 분석하면 그 유형이 몇 가지로 함축 되고, 이에 따라 고려해야 하는 경우의 수가 줄어들어 분석이 용이해진다.

이에 본 논문에서는 실행 파일을 역어셈블하여 만든 어셈블리어를 분석하여 오류를 검출하는 방법을 사용하였다. 다양한 메모리 오류가 있지만

본 논문에서는 Invalid Function Pointer Access Error에 대해서만 다루었다. Invalid Function Pointer Access Error에 대한 유형을 파악한 후 이를 상태기반으로 검사하는 방법을 사용하였다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 메모리 오류 검출에 대한 기존 연구를 살펴보고, 3장에서는 Invalid Function Pointer Access Error의 가능성을 검출하는 방법을 제안하며 4장에서는 실제 대상 프로그램을 선정하여 제안한 검출 방법으로 검출한 결과를, 5장에서는 결론 및 향후 연구 과제에 대해 기술한다.

Ⅱ. 관련 연구

메모리 오류를 검출하기 위한 연구와 도구들은 표 1과 같이 코드 분석으로 오류를 검출하는 정적 프로그램 분석과 직접 실행이나 에뮬레이션을 통해 오류를 검출하는 방법으로 나누고 소스 코드에 기반을 두어 오류를 검출하는 방법과 실행 파일(어셈블리어 분석 포함)에 기반을 두어 오류를 검출하는 방법으로도 나눈다.

표 1 오류 검출 도구 분류

	정적 프로그램 분석	실행 시 검사
소스 코드 기반 검출	Airac[1]	Test Monitor[2]
실행 파일 기반 검출	제안 방법	Valgrind[3], MEDS[4]

이들 오류 검출 도구에서 분류한 메모리 관련 오류는 표 2와 같이 같다. 힙 영역의 할당과 해제에 관련 된 Memory Leak Error, Duplicate Free Error, Illegal Free Error 등은 테스트나 실행 시 오류를 검사하는 방법으로만 검출 가능하고, Local Memory Return Error와 일부 몇 가지 오류들은 정적 프로그램 분석 방법으로도 검출 가능하다.

Airac[1]은 프로그램이 실행 중에 가질 수 있는 결과 값을 계산하는 요약 해석[5]의 방법으로 C 언어 소스 코드를 분석하여 Out of Bound Error를 검출한다.

Test Monitor[2]는 [그림 1]과 같이 소스 코드에 로깅 시스템을 추가하고, Test Monitor에서 소스 코드가 컴파일되어 실행 파일이 생성된다. 생성된 실행 파일을 Test Monitor에서 실행하는 방법으로 [표 2]에 기술된 에러 중 Stack Overflow Error와 Invalid Function Pointer Access Error를 제외한 나머지 에러를 검출한다. Valgrind는 x86 기계어 해석기를 탑재하고 있고, 해석기를 통해 대상 프로그램의 기계 명령어를 실행하여 메모리 오류를 추적한다. Valgrind[3]은 Uninitialized Memory Access Error, Invalid Pointer Access Error, Out of Bound Error, Memory Leak Error 등의 메모리 오류를 검출한다. MEDS[4]는 두 단계에 걸쳐 메모리 오류를 검출한다. 이 시스템은 [그림 3]처럼 어셈블리어를 분석하고 변수 타입을 메타데이터로 기록을 하는 단계와 기록된 메타데이터를 기반으로 어셈블리어 코드를 가상으로 실행하여 메모리 오류를 검출하는 단계로 나누어 buffer overflow, Uninitialized data reads, double-free, 해제된 메모리 접근 및 취약성을 검출한다.

Ⅲ. 검출 방법

표 2 메모리 오류 종류

오류 명칭	설명
Out of Bound Error	할당된 영역을 벗어난 메모리를 이용하여 명령을 수행할 때 발생
Local Memory Return Error	Stack 메모리를 다른 함수에 전달하고 이를 이용하여 명령을 수행할 경우 발생
Null Pointer Access Error	직접적으로 Null이 할당되거나 메모리 블록 할당 실패로 인해 Null이 할당된 포인터를 참조하여 명령을 수행할 때 발생
Uninitialized Pointer Access Error	초기화 되지 않은 포인터를 통해 명령을 수행하는 경우 발생
Invalid Pointer Access Error	유효하지 않은 포인터를 이용하여 명령을 수행할 때 발생
Duplicate Free Error	이미 해제된 메모리 블록을 중복 해제할 경우 발생
Illegal Free Error	유효하지 않은 포인터를 해제할 경우 발생
Memory Leak Error	Heap영역을 동적 할당 받은 후 해제하지 않은 경우 발생
Stack Overflow Error	프로그램이 사용할 수 있는 Stack 메모리를 넘어서 사용하는 경우 발생
Invalid Function Pointer Access Error	함수의 시작 주소를 가리키지 않은 포인터를 호출할 때 발생

기존 메모리 오류 검출에 관련된 연구들에서는 메모리 사용에 있어 실제 오류가 발생해야 메모리 오류를 검출할 수 있다. 특정 메모리 오류는 그 발생 빈도가 매우 낮아 실행 기반의 방법인 기존 연구들에서는 검출 불가능할 수도 있다. 소스 코드를 분석하여 메모리 오류를 검출하는 경

우 개발자의 구현 스타일을 고려해야하기 때문에 구현 스타일에 대한 경우의 수가 많고, 소스 코드의 최적화 과정이 필요하다. 그러나 본 논문에서 제안한 방법은 실행 파일을 역어셈블하여 획득한 어셈블리어를 분석하여 오류를 검출한다. 이 방법은 컴파일러에 의해 코드가 최적화된 실행파일을 사용하기 때문에 구현 스타일에 대한 제약 사항이 줄어든다.

Invalid Function Pointer Access Error는 잘못된 포인터를 호출 할 경우 발생하고, 프로그램의 오작동을 유발한다. 그림 1은 일반적인 함수 호출의 C언어 코드이고, 그림 1의 (a), (b), (c) 함수가 선언되어 있지 않다면 컴파일 단계에서 컴파일러에 의해 검출이 된다. 그러나 그림 2와 같이 함수의 직접적인 호출이 아니라 필요에 의해 함수 포인터(그림 2의 (a))를 사용하여 함수를 호출해야 할 경우 호출될 함수 포인터에 저장된 주소 값이 정상적인 함수의 시작 주소가 아니면 프로그램의 오작동 또는 프로그램이 종료된다. 그림 3은 함수 포인터를 사용해 함수를 호출한 경우의 실행파일을 역어셈블한 어셈블리어 코드이다. 전역 변수로 선언된 함수 포인터인 0x804ee40 주소(그림 3의 (a))에 저장된 데이터를 EBX 레지스터(그림 3의 (b))에 저장하고, 그림 3의 (c)에서 그 주소를 포인터로 변환시켜 호출한다.

```
    while( true ){
        if( EIP == -1 || EIP == 0xffffffff ) break;
        command = (char *)malloc(strlen(CODE[EIP].assembly) + 1);
        memset(command, 0x00, strlen(CODE[EIP].assembly) + 1);
        sprintf(command, "%s", CODE[EIP].assembly);
#ifdef DEBUG
        printf("%5d:%6d:%s\n", EIP, CODE[EIP].offset, command);
#endif
        int ret = analizeASM(command);
    (a) asmFunction(CODE[EIP].codeLength, CODE[EIP].code, command);
        if( ASM_NO == ret ){
            EIP = EIP + CODE[EIP].offset;
            continue;
        }
```

그림 2 함수 포인터를 사용한 함수 호출의 C언어 코드

```
int main(int argc, char** argv){

    int loaded_word;                          //읽어들인 단어 수.
    voca** dict = NULL;                         //사전구조체포인터배열.
    player* rec_head = NULL;                    //사용자리스트헤더.
    player* user = NULL;                        //로그인사용자 포인터.

    //게임 준비//
(a) Check_Factor(argc);                       //실행 인자 값 확인.
    dict = Load_Dict(argv[1], &loaded_word);
(b) rec_head = Load_Record(argv[2])
(c) Sort_Record(rec_head);                    //사용자리스트 점수 정렬.
```

그림 1 직접적인 함수 호출의 C언어 코드

```
8049629:   8b 1d 40 ee 04 08    (a) mov    0x804ee40,%ebx (b)
804962f:   8b 0d 34 ee 04 08        mov    0x804ee34,%ecx
8049635:   8b 15 94 ee 04 08        mov    0x804ee94,%edx
804963b:   89 d0                    mov    %edx,%eax
804963d:   c1 e0 02                 shl    $0x2,%eax
8049640:   01 d0                    add    %edx,%eax
8049642:   c1 e0 02                 shl    $0x2,%eax
8049645:   8d 04 01                 lea    (%ecx,%eax,1),%eax
8049648:   8b 70 0c                 mov    0xc(%eax),%esi
804964b:   8b 0d 34 ee 04 08        mov    0x804ee34,%ecx
8049651:   8b 15 94 ee 04 08        mov    0x804ee94,%edx
8049657:   89 d0                    mov    %edx,%eax
8049659:   c1 e0 02                 shl    $0x2,%eax
804965c:   01 d0                    add    %edx,%eax
804965e:   c1 e0 02                 shl    $0x2,%eax
8049661:   8d 04 01                 lea    (%ecx,%eax,1),%eax
8049664:   8b 50 08                 mov    0x8(%eax),%edx
8049667:   8b 45 f0                 mov    -0x10(%ebp),%eax
804966a:   89 44 24 08              mov    %eax,0x8(%esp)
804966e:   89 74 24 04              mov    %esi,0x4(%esp)
8049672:   89 14 24                 mov    %edx,(%esp)
8049675:   ff d3                (c) call   *%ebx
8049677:   83 7d f4 ff              cmpl   $0xffffffff,-0xc(%ebp)
804967b:   75 2b                    jne    80496a8 <findInfluence+0x21c>
804967d:   8b 0d 34 ee 04 08        mov    0x804ee34,%ecx
8049683:   8b 15 94 ee 04 08        mov    0x804ee94,%edx
```

그림 3 함수 포인터를 사용한 함수 호출의 어셈블리어 코드

함수 포인터의 경우 프로그램 로직 상 변경 가능하므로 실제 로직의 구성을 알아야 함수 호출 스택을 역추적 하여 저장되는 값의 관찰이 가능하다. 본 논문에서는 어셈블리어 명령어 구문 분석을 이용하여 오류의 가능성을 판별하기 때문에 함수 포인터를 이용하는 방법 중 저장된 값들을 분석하여 오류를 판별하는 방법은 고려하지 않는다.

함수 포인터를 호출하기 전 해당 포인터에 대해 Null 검사를 하여 잘못된 함수 호출(Null 호출)을 피하는 코드가 포함 되어야 프로그램의 안정성이 높다. 그림 4는 함수 포인터 호출 전 Null 검사가 존재할 경우의 C언어 소스코드이고, 그림 4의 (a)에서 signal_server라는 함수 포인터에 대하 Null 검사를 하고 그림 4의 (b)에서 호출을 하기 때문에 안정성이 높은 코드이다. 그림 4의 코드를 컴파일 한 후 이를 역어셈블한 어셈블리어 코드는 그림 5의 코드이다. 그림 5의 (a)에서 0x8066748 번지의 apr_dynamic_fn_retrieve함수를 호출 후 그 반환 값에 대해 TEST 명령어(그림 5의 (b))로 Null 검사를 하고, 그 값을 EDX 레지스터(그림 5의 (c))에 복사한 후 그림 5의 (d)에서 EDX 레지스터에 대한 포인터를 호출한다.

```
    signal_server = APR_RETRIEVE_OPTIONAL_FN(ap_signal_server);
(a) if (signal_server) {
        int exit_status;

        if((b)signal_server(&exit_status, pconf)) != 0) {
            destroy_and_exit_process(process, exit_status);
        }
    }
```

그림 4 함수 포인터 호출 전 Null 검사가 존재하는 C언어 코드

```
0x806a9a9: e8 62 4b 01 00           call   807f510 <ap_process_config_tree>
0x806a9ae: 85 c0                    test   %eax,%eax
0x806a9b0: 89 c7                    mov    %eax,%edi
0x806a9b2: 0f 84 a5 03 00 00        je     806ad5d <main+0xbcd>
0x806a9b8: c7 04 24 b1 20 0e 08     movl   $0x80e20b1,(%esp)
0x806a9bf: e8 84 bd ff ff       (a) call   8066748 <apr_dynamic_fn_retrieve@plt>
0x806a9c4: 85 c0                (b) test   %eax,%eax
0x806a9c6: 89 c2                    mov    %eax,%edx (c)
0x806a9c8: 74 17                    je     806a9e1 <main+0x851>
0x806a9ca: 8b 45 90                 mov    -0x70(%ebp),%eax
0x806a9cd: 89 44 24 04              mov    %eax,0x4(%esp)
0x806a9d1: 8d 45 d0                 lea    -0x30(%ebp),%eax
0x806a9d4: 89 04 24                 mov    %eax,(%esp)
0x806a9d7: ff d2                (d) call   *%edx
0x806a9d9: 85 c0                    test   %eax,%eax
0x806a9db: 0f 85 2f 05 00 00        jne    806af10 <main+0xd80>
0x806a9e1: 85 ff                    test   %edi,%edi
```

그림 5 함수 포인터 호출 전 Null 검사가 존재하는 어셈블리어 코드

그림 6은 어셈블리어를 분석하여 도출한 명령어 전이도이며 함수 포인터에 대한 호출에서 시작하여 역순으로 Null 검사의 패턴 없이 함수의 시작 명령어가 나타날 경우 Invalid Function Pointer Access Error의 가능성이 있다고 판단한다.

start 상태에서 명령어로 TEST가 올 경우 A상태로 전이되고, Invalid Function Pointer Access Error의 가능성 검출이 시작된다. 각 상태에서 다음 상태로 전이하는 조건은 다음과 같다.

- 상태 start: 어셈블리어 명령어 TEST가 올 경우 상태 A로 전이한다.

- 상태 A: 어셈블리어 명령어 CALL이 올 경우 상태 B로 전이하고, MOV가 올 경우 상태 C로 전이한다.
- 상태 B: CALL 명령어의 함수명으로 이전 명령어의 목적 피연산자의 포인터가 올 경우 상태 end로 전이된다.
- 상태 C: MOV 명령어의 소스 피연산자로 이전 명령어의 목적 피연산자가 올 경우 상태 D로 전이한다.
- 상태 D: 목적 피연산자가 일반적인 메모리가 올 경우 상태 A로 전이한다.

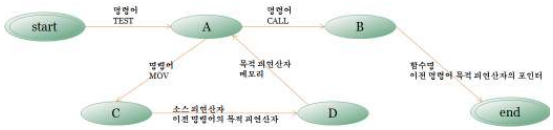

그림 6 함수 포인터 호출 전 Null 검사가 존재하는 경우의 명령어 전이도

Ⅳ. 검출 결과

본 논문에서 성능을 분석하기 위하여 실험한 데이터는 본 논문에서 구현한 검출 도구(대상 1), 오픈 소스 프로젝트인 아파치 웹 서버[6]의 실행 파일 중 httpd(대상 2)와 PHP 스크립트 해석기[7]의 실행 파일 중 php(대상 3)를 사용하였다.

대상 프로그램들은 C언어로 작성되었으며 x86 기반 리눅스 시스템에서 gcc 버전 4로 컴파일하여 실행파일을 만들었고, 동일 시스템의 objdump 툴로 역어셈블하였다.

Invalid Function Pointer Access Error 가능성 검출 결과는 표 4와 같다. 대상1에서는 함수 포인터의 사용이 존재하지 않았고, 대상 2에서 총 함수 포인터를 124회 호출 하지만 Null 검사가 이루어지지 않고 호출한 횟수가 59회로 47.6%의 발생 비율이고 대상 3에서는 987회의 함수 포인터 호출 중 445회가 Null 검사를 하지 않아 45.1%의 결과를 보이고 있다.

메모리 오류 가능성 검출에 소요된 시간은 표 5과 같고 대상 3이 82,586.08 ms로 가장 오랜 시간이 소요되었는데 그 이유는 대상 3의 실행 파일 크기가 50MB이상으로 다른 대상들에 비해 매우 크기 때문이다.

Ⅴ. 결 론

본 논문에서는 실행 파일을 역어셈블하여 만들어진 어셈블리어를 기반으로 “Invalid Function Pointer Access Error”에 대해 메모리 사용의 정상적 유형에 대한 명령어 전이도를 정의하였다.

오픈 소스인 아파치 웹 서버, PHP 프로그램을 컴파일 후 역어셈블하여 만들어진 어셈블리어 코드를 입력 값으로 정상적 유형의 명령어 전이도를 벗어나는지에 따라 메모리 오류의 가능성을 검출하였다.

표 4 Invalid Function Pointer Access Error 가능성 검출 결과

구분	함수 포인터 사용 수	Null 검사 없이 호출한 횟수	발생 비율
대상 1	0	0	0.0%
대상 2	124	59	47.6%
대상 3	987	445	45.1%
합계	1,111	504	

표 5 메모리 오류 검출에 소요된 시간

	함수 수	어셈블리어 명령어 수	검출 소요 시간
대상 1	36	4,079	0.77
대상 2	1,405	132,851	289.30
대상 3	8,101	896,108	82,586.08
계	9,542	1,033,038	82,876.15

단위: ms

참고문헌

[1] 정영범, “Airac", 마이크로 소프트웨어, pp. 178-186, 2005

[2] SureSoftTech, "TEST Monitor", http://suresofttech.com, 2007

[3] J. Seward, N. Nethercote, "Valgrind", http://www.valgrind.org, 2000

[4] Jason D. Hiser, Clark L. Coleman, Michele Co, and Jack W. Davison, "MEDS: The Memory Error Detection System", Engineering Secure Software and Systems, LNCS 5429, pp 164-179, 2009

[5] Patrick Cousot and Radhia Cousot, "Abstract interpretation: a unied lattice model for static analysis of programs by construction or approximation of xpoints", Proceedings of ACM Symposium on Principles of Programming Languages, pp 238-252, 1977

[6] Apache Web Server "http://httpd.apache.org"

[7] PHP Scripting Language "http://www.php.net"