Journal of the Korea Industrial Information Systems Research Vol. 20 No.5, Oct. 2015 :13-23
http://dx.doi.org./10.9723/jksiis.2015.20.5.013 ISSN:1229-3741

역어셈블에 기반한 포인터 참조 오류 검출 방법

(A Method of Detecting Pointer Access Error based on Disassembled Codes)

김 현 수[1], 김 병 만[2], 허 남 철[3], 신 윤 식[4]*
(Hyunsoo Kim, Byeong Man Kim, Nam Chul Huh, and Yoon Sik Shin)

요 약 본 논문에서는 소프트웨어 구동 시 발생 가능한 메모리 오류 중 발생 빈도가 매우 낮은 일부 메모리 오류에 대해 실행 파일을 역어셈블하여 만들어진 어셈블리어의 구문을 분석하여 메모리 오류 가능성을 검출하는 방법을 제안한다. 몇 개의 프로그램을 대상으로 선정하고, 제안한 방법을 이용하여 메모리 오류 가능성을 검출한 결과, 약 만 개의 함수, 백만 라인의 어셈블리어 명령어에서 750여 개의 메모리 오류 가능성을 검출하였고, 검출에 걸린 시간은 총 90초 정도가 소요되었다.

핵심주제어 : 역어셈블, 함수 포인터 참조 오류, 메모리 참조 오류, 정적 분석

Abstract This thesis proposes a method for effectively detecting memory errors with low occurrence frequency that may occur depending on runtime situation by analyzing assembly codes obtained by disassembling an executable file. When applying the proposed method to various programs having no compilation error, a total of about 750 potential errors taken about 90 seconds are detected among 1 million lines of assembly codes corresponding to a total of about 10 thousand functions.

Key Words : Disassembly, Invalid Function Pointer Access Error, Memory Access Error, Static Analysis

1. 서 론

소프트웨어 인수 단계에서는 소프트웨어가 실제 운영될 시스템에 설치를 하고, 각 기능을 점검하는 인수 테스트가 이루어진다. 이 때 숨겨진 오류들로 인해 여러 가지 결함이 발생하게 되면 소프트웨어에 치명적인 영향을 미치게 되고, 그에 따라 해당 소프트웨어뿐만 아니라 해당 소프트웨어가 운영되는 시스템의 오작동까지 유발할 수 있다. 일부 오류들은 발생 빈도가 극히 낮아 테스트 단계에서의 점검이나 인수 테스트에서의 점검에서 발생하지 않고 인수가 된 후 한 달 또는 일 년 이상 운영이 된 후에나 발생할 수 있다.

특히 메모리 사용에 관련한 오류는 소프트웨어에 치명적인 영향으로 해당 소프트웨어의 오작동이나 중지 뿐 아니라, 그 소프트웨어가 운영되는 시스템에 악영향을 미쳐 전체 시스템의 오작동

* Corresponding Author : ysshin@kumoh.ac.kr
† 이 연구는 금오공과대학교학술연구비에 의하여 지원된 논문임.
Manuscript received May 7, 2015 / revised March 10, 2015 / accepted March 29, 2015
1) 디투이모션(주) 선임연구원, 제1저자
2) 금오공과대학교 컴퓨터소프트웨어공학과, 제2저자
3) 대구미래대학교 부사관과, 제3저자
4) 금오공과대학교 컴퓨터소프트웨어공학과, 교신저자

Table 1 Classification of error detection tools

	Static program analysis	dynamic program analysis
Detection based on source code	Airac	Test Monitor
Detection based on executable file	Kim at el.	Valgrind, MEDS

또는 시스템 다운의 원인이 되기도 한다. 이러한 메모리 사용에 관련된 오류가 높은 발생빈도를 가져 테스트 단계나 인수 테스트 중에 발견되어 수정이 되면 다행이지만, 메모리 사용에 관련된 오류들은 대부분 발생 빈도가 낮아 유닛 테스트나 기능 테스트로 발견 할 수 없는 경우가 많다.

개발이 진행되는 동안 각 단계에서 테스트가 이루어지고, 특히 마지막 인수 단계 이전에 테스트 단계를 두어 여러 기능에 대해 집중적으로 테스트를 한다. 그 동안 테스트 단계에서 오류를 검출하기 위해 여러 연구가 진행되었고, 테스트 도구들도 개발되어 현재 시판중인 것도 있다. 이러한 연구와 도구들은 코드 분석으로 오류를 검출하는 정적 프로그램 분석과 직접 실행이나 에뮬레이션을 통해 오류를 검출하는 방법으로 나눌 수 있고, 소스 코드에 기반을 두느냐 실행 파일(어셈블리어 분석 포함)에 기반을 두느냐에 따라 나눌 수도 있다.

정적 프로그램 분석을 이용하면서 소스 코드 기반으로 오류를 검출하는 방법에는 Airac[1], 프로그램 실행 시 소스 코드 기반으로 오류를 검출하는 방법에는 Test Monitor[2], 프로그램 실행 시 실행 파일 기반으로 오류를 검출하는 방법에는 Valgrind[3]과 MEDS[4], 그리고 실행 파일 기반으로 정적 분석을 하는 방법에는 [5]가 있다.

소프트웨어의 신뢰도[6-8]에 치명적인 영향을 주어 결함의 원인이 될 수 있는 메모리 오류를 찾기 위해 Airac[1]처럼 소스 코드의 구문을 분석하거나 Test Monitor[2]처럼 소스 코드 사이에 오류 점검 코드를 추가한 후 직접 실행하는 방법 등으로 검출할 수 있다. 하지만 전자의 경우 각 개발자의 코딩 스타일이 달라 고려해야 하는 경우가 수가 많아지고, 후자의 경우 소프트웨어의 실행 속도가 느려지고 필요로 하는 메모리양이 많아져 소프트웨어의 정상적 실행을 방해할 수 있다.

하지만 [5]의 방법은 컴파일러에 의해 최적화된 실행 파일을 역어셈블하여 만들어진 어셈블리어 코드를 분석하기 때문에 코딩 유형이 몇 가지로 압축되고, 이에 따라 고려해야 하는 경우의 수가 줄어들어 분석이 용이해짐은 물론, 소프트웨어를 직접 실행하지 않고 구문의 패턴을 분석하기 때문에 Valgrind[3]과 MEDS[4]처럼 직접 실행하는 것보다 빠른 시간에 오류를 검출할 수 있다. 추가적으로 소스 코드가 제공되지 않는 라이브러리의 경우에도 역어셈블 기법을 사용하여 혹시나 있을지 모르는 라이브러리 내의 오류를 검출할 수 있다.

본 논문에서는 [5]의 연구를 확장하여 invalid function pointer access error(함수를 가리키지 않은 포인터를 호출할 때 발생)를 추가로 검출하는 방법을 제안하였으며, 추가로 null pointer access error(직접적으로 null이 할당되거나 메모리 블록 할당 실패로 인해 null이 할당된 포인터를 참조하여 명령을 수행할 때 발생)와 invalid function pointer access error 검출 시간을 비교 분석하였다.

2. 관련 연구

오류 검출 도구는 소스의 형태와 검사 시기에 따라 Table 1과 같이 분류할 수 있다. 이번 장에서는 각 분류에 해당하는 대표 시스템에 대해서 간략히 살펴본다.

2.1 소스 코드 기반 구문 분석 오류 검출 도구

Airac[1]은 소스 코드의 구문을 분석하여 메모리 오류를 검사하는 방법으로 C언어 소스 코드를 분석하여 out of bound error를 검출한다. 이 방법에서는 프로그램이 실행 중에 가질 수 있는

결과 값을 요약 해석[9]의 방법으로 분석하여 그 결과를 인터벌(interval)이라는 형태로 표현하고 이를 이용하여 오류 가능성을 검출하게 된다. Airac은 인덱스 인터벌이 크기 인터벌 이외의 값을 가질 가능성이 있을 때 버퍼 오버플로우(out of bound error)가 발생 할 수 있다고 판단한다.

2.2 소스 코드 기반 실행에 의거한 검출 도구

Test Monitor[2]는 소스 코드에 로깅 시스템을 추가하고, Test Monitor에서 소스 코드가 컴파일 되어 실행 파일이 생성된다. 생성된 실행 파일을 Test Monitor에서 실행하는 방법으로 다양한 오류를 검출한다. 하지만 이 방법에서는 invalid function pointer access error는 검출하지 못한다. 이 방법은 테스터가 프로그램을 실행하여 각 기능을 테스트하고, 그 결과를 조합하고 분석하여 메모리 오류와 테스트 커버리지를 계산하여 결과 보고서를 생성한다.

Test Monitor의 주요 기능은 수행 경로 추적, 커버리지 모니터링, 메모리 모니터링 등이 있다. 수행 경로 추적은 프로그램이 실행되는 동안의 수행 과정을 소스 코드 수준으로 재현하여 오류 발생 위치와 원인 정보를 제공한다. 커버리지 모니터링은 전체 소스에서 분기에 따른 커버리지 측정율을 제시하여 테스트 완료 기준점 달성 여부를 판단할 수 있고, 메모리 모니터링은 프로그램이 실행되는 동안에 발생한 메모리 오류, 발생 위치, 메모리 사용량 정보를 제공하여 프로그램 개발 시 개발자의 오류수정에 도움을 준다.

2.3 실행 파일 기반의 에뮬레이팅 오류 검출 도구

Seward의 Valgrind[3]는 가상 실행 기반으로 메모리 오류 검사, 캐시 검사, 호출 그래프 검사, 힙 프로파일링 등을 할 수 있다. Valgrind는 x86 기계어 해석기를 탑재하고 있고, 해석기를 통해 대상 프로그램의 기계 명령어를 실행하여 메모리 오류를 추적한다. Valgrind는 uninitialized memory access error, invalid pointer access error, out of bound error, memory leak error 등의 메모리 오류를 검출한다.

에뮬레이션에 기반하여 메모리 오류를 검출하는 다른 방법은 Hiser의 MEDS(Memory Error Detection System)[4]가 있다. MEDS는 두 단계에 걸쳐 메모리 오류를 검출한다. 이 시스템은 어셈블리어를 분석하고 변수 타입을 메타데이터로 기록을 하는 단계와 기록된 메타데이터를 기반으로 어셈블리어 코드를 가상으로 실행하여 메모리 오류를 검출하는 단계로 나뉜다. 이 방법에서는 buffer overflow, uninitialized data reads, double-free, 해제된 메모리 접근 및 취약성을 검출하고, 운영체제에서 발생하는 시그널 처리, 커널 레벨 멀티쓰레딩 등을 무시한다.

2.4 실행 파일 기반의 정적 분석 오류 검출 방법

Kim et al.[5]은 실행파일을 역어셈블링하여 얻어진 어셈블러 코드를 분석한 후 메모리 사용의 정상적인 경우와 비정상적인 경우의 명령어 전이도를 정의하고 이를 기반으로 메모리 오류를 검출하였다. 즉, 전이도의 시작 상태에서 어셈블리 코드를 읽고, 전이조건에 맞으면 다음 상태로 전이하면서 목표 상태에 도달하면 오류를 판단하는 방법을 사용한다. 검출 가능한 메모리 오류에는 local memory return error(stack 메모리를 다른 함수에 전달하고 이를 이용하여 명령을 수행할 경우 발생)와 null pointer access error 그리고 uninitialized pointer access error(초기화 되지 않은 포인터를 통해 명령을 수행하는 경우 발생)가 있다.

3. 메모리 오류 가능성 검출 방법

본 논문에서는 실행 파일을 역어셈블한 어셈블리어를 분석하고, 분석된 패턴을 이용하여 메모리 오류의 가능성을 검출하는 방법을 제안한다. 본 논문에서는 다양한 메모리 오류 중 null pointer access error와 invalid function pointer access error를 검출한다.

제안 방법의 전체적인 구조는 Fig. 1과 같다.

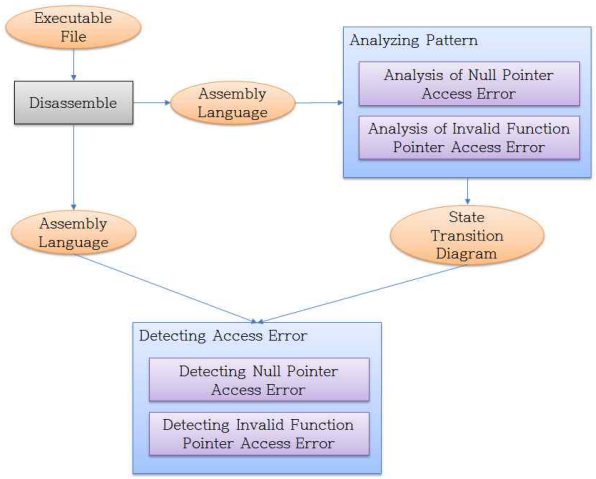

Fig. 1 Structure of memory access error detection

메모리 오류 가능성을 검출하기 위해 메모리 사용의 정상적 패턴을 분석하여 명령어 전이도를 도출하고 도출된 명령어 전이도를 기반으로 invalid function pointer access error와 null pointer access error를 검출한다.

3.1 Null Pointer Access Error 가능성 검출 방법

Null pointer access error는 포인터 변수를 null(0x0)로 초기화한 후 해당 변수를 사용하는 경우에 발생하고, null pointer access error가 발생하면 프로그램이 종료하거나 시스템이 멈춘다. Fig. 2는 직접적인 null 참조의 간단한 예인데 프로그램 실행 시 null pointer access error가 발생하여 해당 프로그램이 중지되면서 segmentation fault가 발생할 수 있다. 하지만 null pointer access error는 직접적인 참조보다는 동적 메모리 할당, 파일 핸들링 등에서 반환된 값을 Null 검사 없이 사용함으로써 발생할 확률이 높다.

```
void nullPointerAccess(void){
  (a) char *a = NULL;
  (b) printf("%s\n", a);
}
```

Fig. 2 Direct null pointer access error in C

Null pointer의 직접적인 접근보다 포인터 형이 반환 값인 함수의 호출 이후 그 반환 값에 대한 접근이 프로그램 코드에서 사용될 수 있지만, 반환 값에 대하여 null 검사가 없는 것보다 검사가 있는 것이 null pointer access error를 방지하기에 좋은 방법이다. 동적 메모리 할당 함수인 malloc 함수와 파일 제어 전 호출하는 fopen 함수의 반환 값이 null인 경우 프로그램에 악영향을 미칠 수 있다.

실제 동적 메모리 할당은 OS에서 제공하는 라이브러리를 사용하는데 이 라이브러리는 null 검사를 하지 않고 사용하는 경우 메모리 블록의 오류 또는 메모리 누수로 인하여 힙 영역의 메모리 여유 공간이 부족하면 메모리 할당 실패로 인한 null pointer access error가 발생한다. 목적 파일이 부재인 상태에서 fopen 함수로 호출하고 null 검사 없이 파일 포인터에 접근할 경우에도 null pointer access error가 발생한다.

malloc 함수와 fopen 함수의 반환 값에 대한 null 검사의 존재 유무에 따라 null pointer access error의 가능성을 판단할 수 있도록, null 검사가 존재할 경우의 패턴을 어셈블리어 상에서 분석하면 CMP 명령어 또는 TEST 명령어로 구성된다.

CMP 명령어는 목적 피연산자와 소스 피연산자의 차를 이용하여 목적 피연산자와 소스 피연산자를 비교하는 명령어이고, TEST 명령어는 소스 피연산자와 목적 피연산자를 논리 곱 연산 후 결과만 FLAGS 레지스터에 기록을 하는 명령어이다.

```
//개행문자의 수를 과악해 사전단어 수 과악.
max_line = Count_Line_Num(file);
(a) dict = (voca**)malloc(sizeof(voca*) * max_line);
(b) if(dict == NULL){puts("[오류] 메모리확보가 불가능
```

Fig. 3 Checking null after dynamic memory allocation in C

CMP 명령어를 사용하여 null 검사를 하는 유형은 일반적으로 가장 많이 사용하는 형태인데 반환 값을 지역변수에 저장하는 유형의 코드이다. Fig. 3은 그 유형을 나타내는 C언어 코드이다. Fig. 3의 (a)에서 malloc 함수로 할당받은 포

인터를 dist 변수에 대입하고, (b)에서 dist 변수를 Null과 비교한다. Fig. 3을 어셈블리어로 변환하면 Fig. 4가 된다.

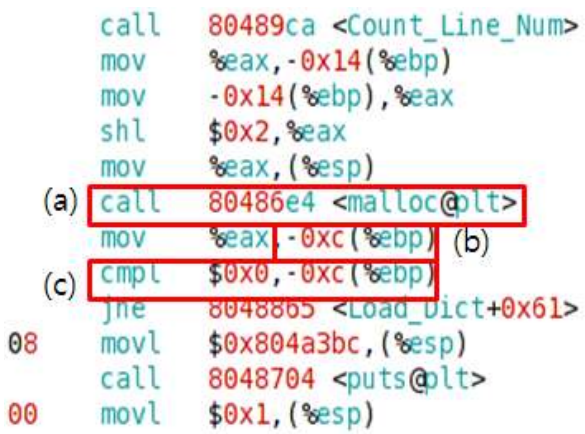

Fig. 4 Checking null after dynamic memory allocation in assembly

Fig. 4의 (a)에서 malloc 함수를 호출 후 지역변수(Fig. 4의 (b))에 그 결과를 저장하고, Fig. 4의 (c)에서 CMPL 명령어를 이용하여 null(0x0)과 비교한다.

CMP 명령어 대신 TEST 명령어를 사용하여 Null 검사를 하는 유형은 3가지이다. 구조체 내의 특정 위치나 배열을 사용하는 유형, 할당된 메모리에 여러 번 사용할 경우에 나타나는 유형, 마지막으로 다른 분기에서 할당된 포인터를 같은 위치에서 검사하는 유형이다. 지면 관계 상 여기서는 첫 번째 유형에 대해서만 기술토록 한다. 나머지 유형은 Kim et. al.[5]를 참고하기 바란다.

TEST 명령어의 첫 번째 유형은 구조체 내의 특정 위치나 포인터 배열을 사용하는 유형이며 이러한 경우 null 검사의 목적 주소를 특정 레지스터에 저장하고, 그 레지스터를 TEST 명령어로 검사한다. Fig. 5는 TEST 명령어를 사용하여 null 검사를 하는 첫 번째 유형의 C언어 코드이다. Fig. 5의 (a)에서는 malloc 함수로 할당 받은 메모리를 dist 배열의 i번째 위치에 저장하고 Fig. 5의 (b)에서 null과 비교한다. Fig. 6은 Fig. 5의 어셈블리어 코드이다.

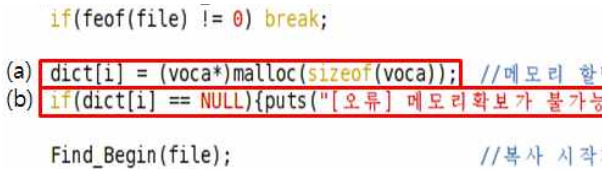

Fig. 5 The first type of 'TEST' command in C

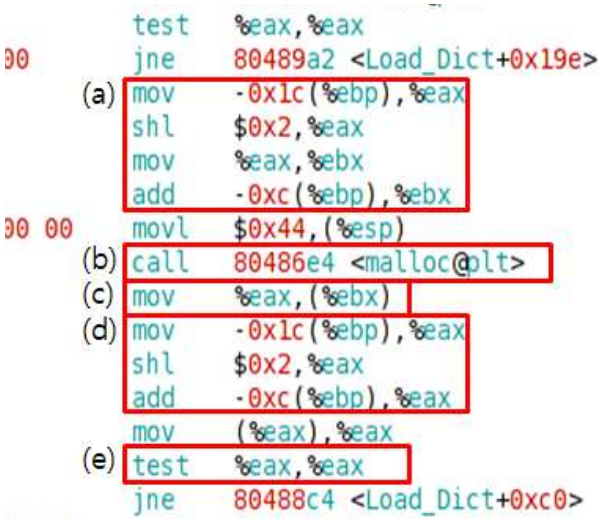

Fig. 6 The first type of 'TEST' command in assembly

Fig. 6의 (a)와 같이 포인터 배열 내의 특정 위치를 계산하기 위한 과정을 거친 후 그 주소를 Fig. 6의 (a) 마지막에 위치하는 EBX 레지스터에 저장을 하고, Fig. 6의 (b)에서 호출한 malloc 함수의 반환 값을 Fig. 6의 (c)에서 EBX 레지스터가 가리키는 주소에 대입한 후 다시 Fig. 6의 (d) 과정을 통해 주소를 계산하고 Fig. 6의 (e)에서 TEST 명령어로 null 검사를 한다.

동적 메모리 할당 함수인 malloc 함수 또는 파일 열기 함수인 fopen 함수의 반환 값에 대해 null 검사를 한 경우의 어셈블리어의 유형을 분석하여 도출된 명령어 전이도는 Fig. 7과 같다. 역어셈블된 어셈블리어를 함수의 시작부분부터 순차적으로 검사하여 명령어 전이도의 패턴을 벗어나는 경우를 null pointer access error의 가능성이 있다고 판단한다. 즉, 상태 start에서 명령어로 CALL이 올 경우 상태 A로 전이되면서 null 검사가 존재하는지에 대한 검출을 시작한다.

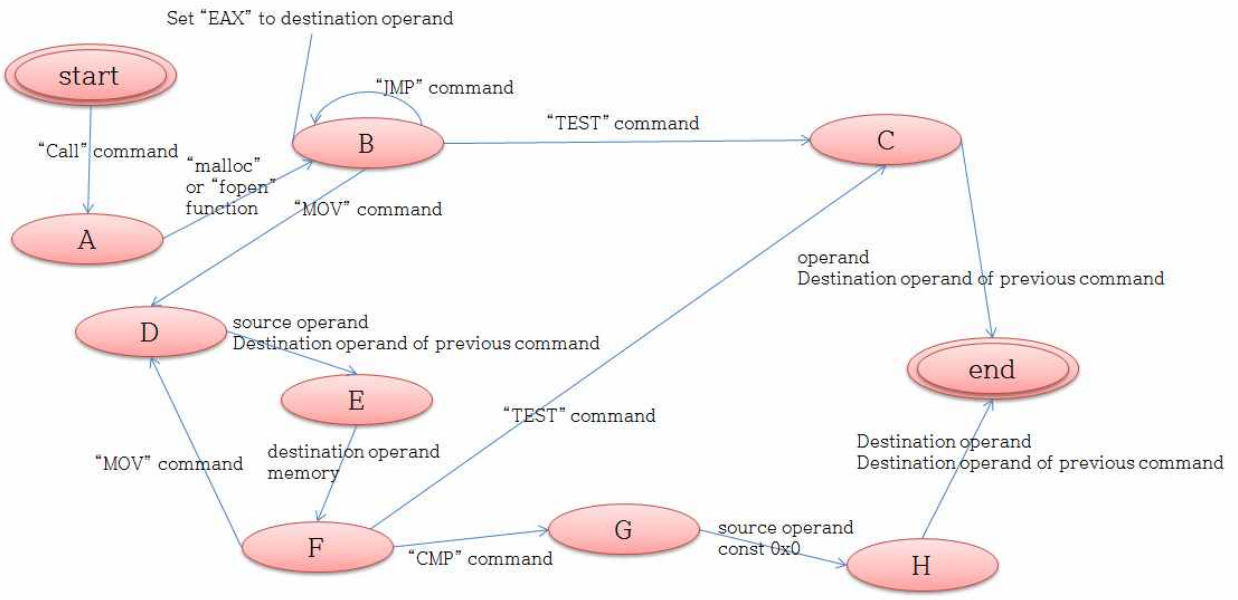

Fig. 7 State transition diagram for detecting null pointer access error

각 상태에서 전이 조건에 따라 다음 상태로 전이하면서 end 상태에 도달하면 정상적이고 그렇지 않으면 null pointer access error가 발생 가능한 것으로 판단한다.

3.2 Invalid Function Pointer Access Error 가능성 검출 방법

Invalid function pointer access error는 잘못된 포인터를 호출 할 경우 발생하고, 프로그램의 오작동을 유발한다. Fig. 8은 일반적인 함수 호출의 C언어 코드이고, Fig. 8의 (a), (b), (c) 함수가 선언되어 있지 않다면 컴파일 단계에서 컴파일러에 의해 검출이 된다. 그러나 Fig. 9와 같이 함수의 직접적인 호출이 아니라 필요에 의해 함수 포인터(Fig. 9의 (a))를 사용하여 함수를 호출해야

```
player* user = NULL;                                    /

//게임 준비//
(a) Check_Factor(argc);                                 //실
    dict = Load_Dict(argv[1], &loaded_word);
(b) rec_head = Load_Record(argv[2])
(c) Sort_Record(rec_head);                              //사
```

Fig. 8 Direct function call in C

할 경우 호출될 함수 포인터에 저장된 주소 값이 정상적인 함수의 시작 주소가 아니면 프로그램의 오작동 또는 프로그램이 종료된다.

Fig. 10은 함수 포인터를 사용해 함수를 호출한 경우의 실행파일을 역어셈블한 어셈블리어 코드이다. 전역 변수로 선언된 함수 포인터인 0x804ee40 주소(Fig. 10의 (a))에 저장된 데이터를 EBX 레지스터(Fig. 10의 (b))에 저장하고, Fig. 10의 (c)에서 그 주소를 포인터로 변환시켜 호출한다.

```
#ifdef DEBUG
        printf("%5d:%6d:%s\n", EIP, CODE[EIP].offse
#endif
        int ret = analizeASM(command);
    (a) asmFunction(CODE[EIP].codeLength, CODE[EIP]
        if( ASM_NO == ret ){
            EIP = EIP + CODE[EIP].offset;
            continue;
        }
```

Fig. 9 Function call using function pointer in C

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 분

```
8b 1d 40 ee 04 08      (a) mov    0x804ee40,%ebx (b)
8b 0d 34 ee 04 08          mov    0x804ee34,%ecx
8b 15 94 ee 04 08          mov    0x804ee94,%edx
89 d0                      mov    %edx,%eax
c1 e0 02                   shl    $0x2,%eax
01 d0                      add    %edx,%eax
c1 e0 02                   shl    $0x2,%eax
8d 04 01                   lea    (%ecx,%eax,1),%ea
8b 70 0c                   mov    0xc(%eax),%esi
8b 0d 34 ee 04 08          mov    0x804ee34,%ecx
8b 15 94 ee 04 08          mov    0x804ee94,%edx
89 d0                      mov    %edx,%eax
c1 e0 02                   shl    $0x2,%eax
01 d0                      add    %edx,%eax
c1 e0 02                   shl    $0x2,%eax
8d 04 01                   lea    (%ecx,%eax,1),%ea
8b 50 08                   mov    0x8(%eax),%edx
8b 45 f0                   mov    -0x10(%ebp),%eax
89 44 24 08                mov    %eax,0x8(%esp)
89 74 24 04                mov    %esi,0x4(%esp)
89 14 24                   mov    %edx,(%esp)
ff d3                  (c) call   **%ebx
83 7d f4 ff                cmpl   $0xffffffff,-0xc(
```

Fig. 10 Function call using function pointer in assembly

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

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

Fig. 11 Checking null before function call via function pointer in C

석을 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. Fig. 11은 함수 포인터 호출 전 null 검사가 존재할 경우의 C언어 소스 코드이고, Fig. 11의 (a)에서 signal_server라는 함수 포인터에 대하여 null 검사를 하고 Fig. 11의 (b)에서 호출하기 때문에 안정성이 높은 코드이다. Fig. 11의 어셈블리어 코드는 Fig. 12이다.

Fig. 12의 (a)에서 0x8066748 번지의 apr_dynamic_fn_retrieve함수를 호출 후 그 반환 값에 대해 TEST 명령어(Fig. 12의 (b))로 null 검사를 하고, 그 값을 EDX 레지스터(Fig. 12의 (c))에 복사한 후 Fig. 12의 (d)에서 EDX 레지스터에 대한 포인터를 호출한다.

Fig. 13은 Fig. 12의 어셈블리어를 분석한 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.

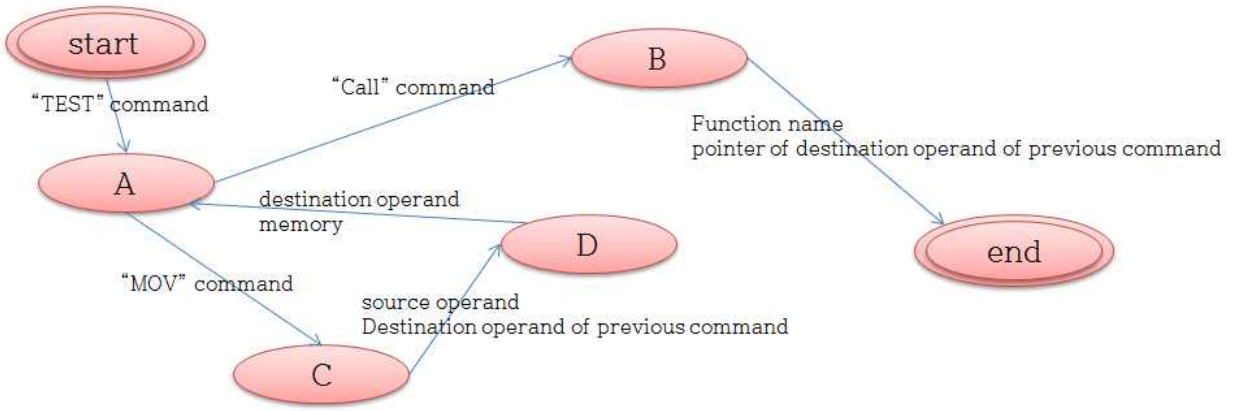

Fig. 13 State transition diagram of existing null check

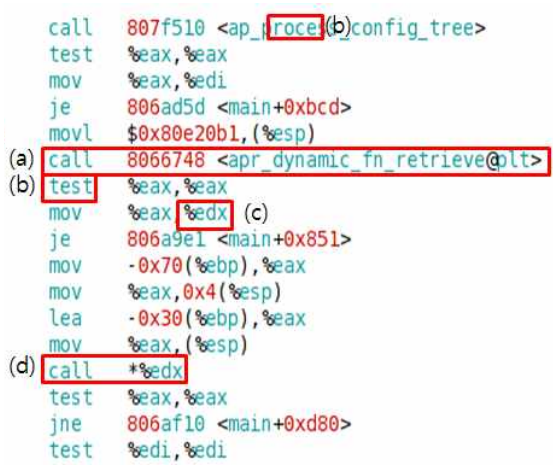

Fig. 12 Checking null before function call via function pointer in assembly

4. 메모리 오류 가능성 검출 결과

본 논문에서 성능을 분석하기 위하여 실험한 데이터는 컴퓨터 관련 학과에 재학 중인 학생들이 과제로 제출한 소스 코드 중 일부(대상 1~5)와 본 논문에서 구현한 검출 도구(대상 6), 오픈 소스 프로젝트인 아파치 웹 서버[8]의 실행 파일 중 httpd(대상 7)과 htpasswd(대상 8), PHP 스크립트 해석기[9]의 실행 파일 중 php(대상 9)를 사용하였다.

대상 프로그램들은 C언어로 작성되었으며 x86 기반 리눅스 시스템에서 gcc 버전 4로 컴파일하여 실행파일을 만들었고, 동일 시스템의 objdump 프로그램으로 역어셈블하였다. 실험에 사용한 시스템의 상세한 사양은 Table 2와 같다.

Table 2 Specification of system for experiments

Type	Specification
Processor	Intel Centrino Duo T2300 @ 1.66GHz
Memory	Cache: 2048 KB System: 2048 MB Swap: 1012 MB
HDD	SATA 80 GB 5400 RPM
OS	Fedora 10 (Kernel: 2.6.27.38-170)

4.1 Null Pointer Access Error 가능성 검출 결과

Null pointer access error는 모든 대상에서 평균적으로 많은 양이 발견되었다. Null pointer access error 중 malloc 함수에 의한 오류는 Table 3과 같고, 학생 대상의 발생 비율 중 대상 5번의 경우 483.3%로 가장 좋지 않은 구현 결과이다. 483.3%의 비율이 발생한 원인은 malloc 함수의 호출 수는 6번이지만 Null 검사가 이루어지지 않은 상태에서 여러 번 대상 포인터에 접근하였기 때문이다. 대상 6번이 20개 중 2개가 발생하여 10%로 가장 좋은 구현 결과이다.

Table 3 Result of detecting candidates of null pointer access error (malloc)

a: number of calling "malloc"
b: number of accessing without Null check
c: occurrence rate(b / a * 100)

Type	Target	a	b	c
Student	1	29	18	62.07%
	2	10	3	30.00%
	3	5	11	220.00%
	4	2	4	200.00%
	5	6	29	483.33%
	6	20	2	10.00%
Open Source	7	12	2	16.67%
	8	0	0	0.00%
	9	177	161	90.96%
Total		261	230	

오픈 소스의 경우 대상 9번이 177회의 malloc 함수 호출 중 161개가 발생하여 90.96%로 가장 좋지 않은 구현 결과를 가지는 반면 대상 8의 경우 0%의 발생 비율이지만 malloc 함수를 호출하지 않았기 때문에 대상 7의 발생 비율 16.67%가 가장 좋은 구현 결과라 할 수 있다.

Null pointer access error 중 fopen 함수는 Table 4와 같고, 학생 대상 프로그램 중 대상 5가 5회의 fopen 호출 중 Null 검사 없이 해당 포인터에 접근한 횟수 6회로 120%의 발생 비율로 가장 좋지 않은 구현 결과이고, 대상 1, 대상 2,

Table 4 Result of detecting candidates of null pointer access error (fopen)

a: number of calling "fopen"
b: number of accessing without Null check
c: occurrence rate(b / a * 100)

Type	Target	a	b	c
Student	1	3	0	0.00%
	2	1	0	0.00%
	3	6	0	0.00%
	4	6	2	33.33%
	5	5	6	120.00%
	6	3	0	0.00%
Open Source	7	0	0	0.00%
	8	0	0	0.00%
	9	15	9	60.00%
Total		39	17	

Table 5 Result of detecting candidates of invalid function pointer access error

a: number of using pointer
b: number of accessing without Null check
c: occurrence rate(b / a * 100)

Type	Target	a	b	c
Student	1	0	0	0.00%
	2	0	0	0.00%
	3	0	0	0.00%
	4	0	0	0.00%
	5	0	0	0.00%
	6	0	0	0.00%
Open Source	7	124	59	47.58%
	8	0	0	0.00%
	9	987	445	45.09%
Total		1,111	504	

대상 3, 대상 6의 경우 발생 비율이 0%이다. 그 중 가장 좋은 구현 결과라 할 수 있는 것은 대상 3번의 경우로 fopen 함수를 6회 호출하였지만 모든 호출에 대해 null 검사를 하였기 때문에 가장 안정적인 구현 결과이다.

오픈 소스의 경우 대상 9가 15회의 fopen 호출 중 null 검사 없이 해당 포인터에 접근한 횟수가 9회가 발생하여 발생 비율이 60%로 가장 좋지 않은 구현 결과를 가지고 있지만 대상 7번과 대상 8번의 경우 fopen 함수의 호출 수가 0회이기 때문에 오픈 소스에서는 대상 9가 가장 좋은 구현 결과이면서 좋지 않은 구현 결과라 할 수 있다.

4.2 Invalid Function Pointer Access Error 가능성 검출 결과

Invalid function pointer access error 가능성 검출 결과는 Table 5와 같고, 학생 대상 프로그램의 경우 함수 포인터의 호출 횟수가 0이기 때문에 결과를 비교하지 않는다.

오픈 소스의 경우 대상 7에서 함수 포인터를 124회 호출 하지만 null 검사가 이루어지지 않은 횟수가 59회로 47.58%의 발생 비율로 가장 좋지 않은 구현 결과를 가지는 반면, 대상 9에서는 987회의 함수 포인터 호출 중 445회가 null 검사를 하지 않아 45.09%의 결과로 가장 좋은 구현 결과를 가지고 있다. 대상 8 역시 학생 대상의 경우와 마찬가지로 함수 포인터의 호출이 없기 때문에 비교 대상에서 제외하였다.

Table 6 Execution time for detecting memory errors

a: number of functions
b: number of assembly command
c: spending time for detecting null pointer access error
d: spending time for detecting invalid function pointer access error
e: total time
Unit: ms

Target	a	b	c	d	e
1	53	4,381	9.9	0.7	10.6
2	45	2,003	2.8	0.3	3.1
3	80	4,802	7.0	0.7	7.7
4	18	978	2.4	0.2	2.6
5	14	1,111	0.3	0.3	0.6
6	36	4,079	10.1	0.8	10.9
7	1,405	132,851	485.6	289.3	774.9
8	4	892	6.9	0.1	7
9	8,101	896,108	6,155.2	82,586.1	88,741.3
Total	9,756	1,047,205			89,558.7

4.3 메모리 오류 가능성 검출 시간

메모리 오류 가능성 검출에 소요된 시간은 Table 6과 같고 대상 9가 88,743.3 ms로 가장 오랜 시간이 소요되었는데 그 이유는 대상 9의 실행 파일 크기가 50MB이상으로 다른 대상들에 비해 매우 크기 때문이다. 본 논문에서는 대상 9가 가장 좋지 않은 구현 결과를 가지지만 파일 사이즈가 크고 어셈블리어 명령어 수가 많기 때문이라 할 수 있고, 대상 9를 제외한 나머지의 경우에도 파일 사이즈와 어셈블리어 명령어 수가 늘어난다면 대상 9와 같은 결과를 초래할 수 있다.

5. 결론 및 향후 연구

메모리 오류 중 일부는 발생 빈도가 극히 낮은 오류이다. 시스템 테스트 시 검출을 하지 못하는 경우가 발생할 수 있고 차후 이 후 이 오류에 의해 프로그램의 중단되거나 시스템의 오작동을 유발하여 사용자에게 피해를 발생시킬 수 있다.

본 논문에서는 실행 파일 및 라이브러리 파일을 역어셈블하여 만들어진 어셈블리어를 기반으로 invalid function pointer access error와 null pointer access error에 대해 메모리 사용의 정상적 유형에 대한 명령어 전이도를 정의하고, 이를 기반으로 학생들의 소스코드, 오픈 소스인 아파치 웹 서버, PHP 프로그램을 역어셈블하여 만들어진 어셈블리어 코드를 입력 값으로 메모리 오류의 가능성을 검출하였다. 적용 결과, 학생들 코드뿐만 아니라 상용화 시스템 코드에서도 많은 양의 메모리 오류 가능성이 있음을 확인할 수 있었다.

향후 본 논문에서 다루지 못한 오류 중 프로그램 분석 기법을 통해 검출이 가능한 오류를 검출하는 방법에 대해 추가적인 연구를 할 계획이고, 동적 검사를 추가하여 완벽한 메모리 오류 검출에 대한 연구 및 보안 관련 취약성 검출에 대한 연구를 계속할 계획이다.

References

[1] Yungbum Jung, Jaehwang Kim, Jaeho Shin and Kwangkeun Yi, “Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis”, SAS'05: International Static Analysis Symposium, London, 2005.

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

[3] J. Seward and 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] HyunSoo Kim, Byeong Man Kim, HyunSeop Bae and In Sang Chung, “Detection of Potential Memory Access Errors based on Assembly Codes”, The KIPS Transactions:PartD, Vol 18D, Issue 1, pp. 35-44, 2011.

[6] Gye-Tak Yang, “A Study on the Reliability of S/W during the Developing Stage”, Journal of the Korea Society Industrial Information System, Vol. 14, No. 5, 2009.

[7] Jeong Hyang An and Sang Chul Yoon, “Estimation of Reliability for a Parallel System with Dependent Exponential Components”, Journal of the Korea Society Industrial Information System, Vol. 8, No. 4, 2003.

[8] Yong-Kyong Kim, “A Study on the Characteristics of the Small and Medium System Integration Companies in Performing IT Projects”, Journal of the Korea Society Industrial Information System, Vol. 15, No. 5, 2010.

[9] Patrick Cousot and Radhia Cousot, "Abstract interpretation: a united 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.

[10] Apache Web Server "http://httpd.apache.org", 1995, using source from 2007.

[11] PHP Scripting Language "http://www.php.net", 1995, using source from 2007.

김 현 수 (Hyunsoo Kim)

- 정회원
- 금오공과대학교 컴퓨터공학부 공학사
- 금오공과대학교 소프트웨어공학과 공학석사
- 금오공과대학교 소프트웨어공학과 공학박사
- 디투이모션(주) 선임연구원
- 관심분야 : 인공지능, 역공학, 소프트웨어테스팅

김 병 만 (Byeong Man Kim)

- 정회원
- 서울대학교 컴퓨터공학과 공학사
- 한국과학기술원 전산학과 공학석사
- 한국과학기술원 전산학과 공학박사
- 국립금오공과대학교 교수
- 미국 UC, Irvine 대학 방문교수
- 미국 콜로라도 주립대학 대학 방문교수
- 관심분야 : 인공지능, 정보검색, 정보보안

허 남 철 (Nam Chul Huh)

- 정회원
- 계명대학교 전자계산학과 공학사
- 한국과학기술원 전산학과 공학석사
- 한국과학기술원 전산학과 공학박사
- 산업과학기술연구소(현 포항산업과학연구원) 주임연구원
- 대구미래대학교 교수
- 관심분야 : 정보검색, 정보보안

신 윤 식 (Yoon Sik Shin)

- 정회원
- 경북대학교 공학사
- 한국과학기술원 전산학과 공학석사
- 국립금오공과대학교 교수
- 관심분야 : 소프트웨어공학