DOI: 10.3745/KIPSTD.2011.18D.1.035

어셈블리어 코드 기반의 메모리 오류 가능성 검출

김 현 수[†] · 김 병 만[††] · 배 현 섭[†††] · 정 인 상[††††]

요 약

메모리 사용에 관련한 오류는 해당 프로그램뿐 아니라 시스템의 오작동을 유발할 수 있다. 특히 발생 빈도가 매우 낮은 일부 메모리 오류의 경우 제대로 된 동작 테스트를 할 수 없어, 오류에 대한 파악 및 수정이 힘들다. 이에 본 논문에서는 실행 프로그램을 역어셈블(Dis-Assemble)하여 만들어진 어셈블리어 코드를 구문 분석하여 명령어 전이도를 도출하고 이에 기반을 두어 메모리 사용 오류 가능성을 검출하는 방법을 제안하였다. 몇 가지 프로그램을 검사대상으로 선정하여 Local Memory Return Error, Null Pointer Access Error, Uninitialized Pointer Access Error를 검출하였으며 그 중 오픈 소스 프로젝트(Open Source Project)인 아파치 웹 서버와 PHP 스크립트 해석기에서도 메모리 사용 오류의 가능성이 있는 코드가 검출되었다.

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

Detection of Potential Memory Access Errors based on Assembly Codes

HyunSoo Kim[†] · Byeong Man Kim[††] · HyunSeop Bae[†††] · In Sang Chung[††††]

ABSTRACT

Memory errors can cause not only program malfunctions but also even unexpected system halt. Though a programmer checks memory errors, some memory errors with low occurrence frequency are missed to detect. In this paper, we propose a method for effectively detecting such memory errors using instruction transition diagrams through analyzing assembly codes obtained by disassembling an executable file. Out of various memory errors, local memory return errors, null pointer access errors and uninitialized pointer access errors are targeted for detection. When applying the proposed method to various programs including well-known open source programs such as Apache web server and PHP script interpreter, some potential memory errors are detected.

Keywords : Detection Of Function Pointer Error, Static Program Analysis, State Transition Diagram, Analysis Based On Assembly Code

1. 서 론

프로그램 개발 시 버그로 인해 결함이 발생할 수 있고, 특히 메모리 사용에 관련한 오류는 프로그램에 치명적인 결함으로 해당 프로그램뿐만 아니라 전체 시스템의 오작동까지 유발할 수 있다. 하지만 이들 오류는 발생 빈도가 극히 낮아 실행 시 문제점으로 부각되지 않을 수 있는 코드가 있어 단순한 테스트로 알 수 없는 경우가 많다.

이러한 메모리 오류를 검출하기 위한 여러 연구들에서 사용하는 방법을 분류해보면 크게 두 가지 방법으로 나눌 수 있다. 첫 번째는 원시 소스 코드를 분석하여 오류를 검출하는 방법이고 두 번째는 테스트용 코드를 원시 코드에 삽입 후 컴파일한 뒤 직접 실행하는 방법이다. 전자의 경우 각 개발자들의 코딩 스타일이 달라 고려해야 하는 경우의 수가 많아져 검출에 어려움이 있고, 후자의 경우 전체 프로그램의 모든 경로에 대해 검사할 수 있다는 보장도 없을뿐더러 검출에 소요되는 시간이 매우 길다.

하지만 실행 파일을 역어셈블하여 만들어진 어셈블리어 코드를 사용하여 메모리 오류를 검출할 경우 컴파일러에 의해 최적화되어 고려해야 하는 경우의 수가 줄어들어 검출이 용이해짐은 물론 프로그램의 모든 경로에 대한 검사 및 검사에 소요되는 시간도 줄어든다. 추가적으로 소스 코드가

※ 본 연구는 금오공과대학교 학술연구비에 의하여 연구되었음.
† 준 회 원 : 금오공과대학교 소프트웨어공학과 박사과정
†† 정 회 원 : 금오공과대학교 컴퓨터공학부 교수
††† 비 회 원 : 슈어소프트테크(주) 대표이사
†††† 정 회 원 : 한성대학교 컴퓨터공학과 교수
논문접수 : 2010년 6월 21일
수 정 일 : 1차 2010년 9월 8일, 2차 2010년 9월 17일
심사완료 : 2010년 9월 18일

제공되지 않는 라이브러리의 경우에도 역어셈블 기법을 사용하여 혹시나 있을지 모르는 라이브러리 내의 메모리 사용 오류를 검출할 수 있다.

이에 본 논문에서는 테스트 과정에서 발생 확률이 낮아 검출이 어려운 메모리 사용에 관련한 오류를 검출하기 위해 실행 파일을 역어셈블하여 그 결과를 분석한 후 메모리 사용의 정상적 경우와 비정상적 경우의 명령어 전이도를 정의하고 명령어 전이도를 기반으로 메모리 오류를 검출하였다. 다양한 메모리 오류가 있지만 Local Memory Return Error와 Null Pointer Access Error, Uninitialized Pointer Access Error에 대해서 다루었다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 메모리 오류 검출에 관련한 기존 연구를 살펴보고 3장에서는 각 메모리 오류별 명령어 전이도를 도출하며 4장에서는 대상 프로그램을 선정하여 3장에서 도출한 명령어 전이도를 기반으로 실험한 결과를 기술하고 5장에서는 결론 및 향후 연구 과제에 대해 기술한다.

2. 관련연구

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 나누고 소스 코드에 기반을 두어 오류를 검출하는 방법과 실행 파일(어셈블리어 분석 포함)에 기반을 두어 오류를 검출하는 방법으로 나눈다. <표 1>에서 보는 것과 같이 정적 프로그램 분석을 이용하면서 소스 코드 기반으로 오류를 검출하는 방법 Airac[1], 프로그램 실행 시 소스 코드 기반으로 오류를 검출하는 방법 Test Monitor[2], 프로그램 실행 시 실행 파일 기반으로 오류를 검출하는 방법 Valgrind[3]과 MEDS[4]등으로 나누고 본 논문에서 새롭게 정의하는 정적 프로그램 분석으로 실행 파일 기반 오류 검출 방법이 있다.

〈표 1〉 오류 검출 도구 분류

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

Airac[1]은 C언어 소스 코드의 구문을 분석하여 메모리 오류를 검사하는 방법이고, Test Monitor[2]는 소스 코드에 로깅 시스템을 추가하고 직접 실행을 통해 메모리 오류를 검사하는 방법이며, Valgrind[3]과 MEDS[4]는 실행 파일의 CPU 기계어를 가상 머신으로 실행하여 오류를 검사하는 방법이다. 이러한 방법들에서 검출하는 메모리에 관련된 오류는 Out of Bound Error, Local Memory Return Error, Null Pointer Access Error, Uninitialized Pointer Access Error, Invalid Pointer Access Error, Duplicate Free Error, Illegal Free Error, Memory Leak Error, Stack Overflow Error, Invalid Function Pointer Access Error등이다.

이 중 Memory Leak Error, Duplicate Free Error, Illegal Free Error 등은 힙 영역의 할당과 해제에 관련되어 테스트나 실행 시 오류를 검사하는 방법으로만 검출 가능하고, Local Memory Return Error와 일부 몇 가지 오류들은 정적 프로그램 분석 방법으로도 검출 가능하다.

2.1 소스 코드 기반 구문 분석 오류 검출 도구

Airac[1]은 프로그램이 실행 중에 가질 수 있는 결과 값을 계산하는 요약 해석[5]의 방법으로 C언어 소스 코드를 분석하여 Out of Bound Error를 검출한다. 요약 해석에서는 a보다 크고 b보다 작거나 같은 수들의 집합을 나타내는 [a,b], 모든 수의 집합을 나타내는 [-∞,∞], 인터벌 I가 J에 포함됨을 의미하는 $I \subseteq J$, 모든 인터벌 보다 작은 인터벌을 의미하는 ⊥ 등의 기호를 써서 오류를 검출한다. 여기서 인터벌이란 소스 코드에 있는 수식 값들의 요약을 의미한다.

배열은 시작 주소, 배열의 크기 인터벌, 현재 포인터가 가리키고 있는 배열의 오프셋 인터벌을 가지는 값으로 요약되고, 메모리는 메모리가 할당된 프로그램 상의 위치로 요약된다. Airac에서는 프로그램의 모든 식, 명령문, 선언 등에 교유한 레이블을 할당한다.

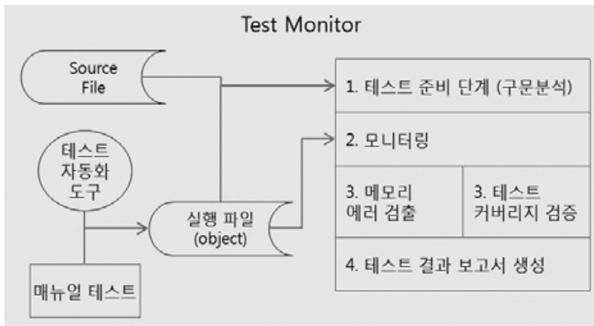
<표 2>는 Airac에서의 요약 해석이다. l은 배열 선언 $a[10]$에 할당된 레이블이다. Airac은 인덱스 인터벌이 크기 인터벌 외의 값을 가질 가능성이 있을 때 버퍼 오버플로(Out of Bound Error)가 발생 할 수 있다고 판단한다.

〈표 2〉 Airac에서의 요약 해석

C언어 소스 코드	int a[10], *p; p = a + 3;
요약 해석 도메인	$a = \langle l, [10,10], 0 \rangle$ $p = \langle l, [10,10], [3,3] \rangle$

2.2 소스 코드 기반 실행에 의거한 오류 검출 도구

Test Monitor[2]는 (그림 1)과 같이 소스 코드에 로깅 시스템을 추가하고, Test Monitor에서 소스 코드를 컴파일하여 실행 파일을 생성한다. 생성된 실행 파일을 Test Monitor에서 실행하여 Stack Overflow Error와 Invalid Function Pointer Access Error를 제외한 나머지 에러를 검출한다. 테스터가 프로그램을 실행하여 각 기능을 테스트하면 그 결과를 분석하여 메모리 에러와 테스트 커버리지를 계산하여 결과 보고서를 생성한다.



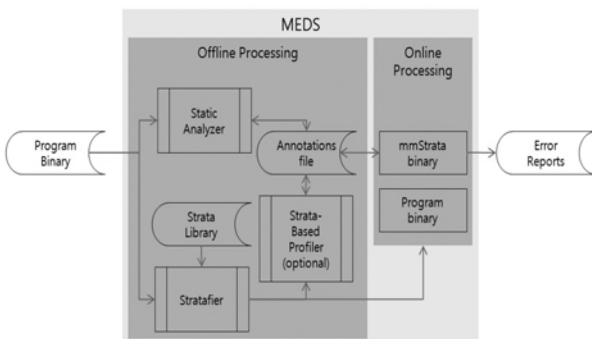
(그림 1) Test Monitor 동작 순서

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 오류 수정에 도움을 준다.

2.3 실행 파일 기반의 실행에 의거한 오류 검출 도구

Seward의 Valgrind[3]는 프로그램을 직접 실행하여 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 추적하여 Uninitialized Memory Access Error, Invalid Pointer Access Error, Out of Bound Error, Memory Leak Error 등의 메모리 오류를 검출한다.

실행에 의거하여 메모리 오류를 검출하는 다른 방법은 Hiser의 MEDS(Memory Error Detection System)[4]가 있다. MEDS는 두 단계에 걸쳐 메모리 오류를 검출한다. 이 시스템은 (그림 2)처럼 어셈블리어를 분석하고 변수 타입을 메타데이터로 기록을 하는 단계와 기록된 메타데이터를 기반으로 어셈블리어 코드를 가상으로 실행하여 메모리 오류를 검출하는 단계로 나뉜다.



(그림 2) MEDS 개요

MEDS의 첫 번째 단계는 어셈블리어를 분석하여 레지스터, 메모리 위치 등 모든 저장 위치마다 숫자형인지 포인터형인지를 구분하여 메타데이터로 기록한다. 포인터와 숫자에 대해 연산을 할 경우 각 연산자에 따라 (그림 3)에 의거하여 데이터 타입을 구분한다. (그림 3)의 P는 포인터형 데이터를 의미하고 N은 숫자형 데이터를 의미한다. 그리고 -, &, |, +, *, /, %, ^, ~, <<, >>는 연산자를 의미한다.

+	p	n
p	n	p
n	p	n

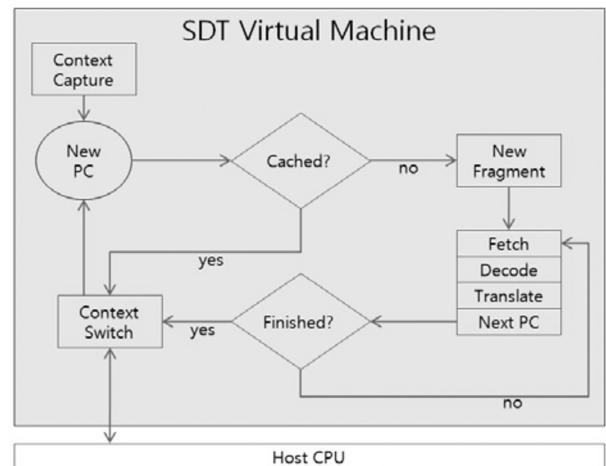
-	p	n
p	n	p
n	n	n

&,|	p	n
p	p/n	p/n
n	p/n	n

*, /, %, ^, ~, <<, >>	p	n
p	n	n
n	n	n

(그림 3) 메타데이터 구분 방법

MEDS의 두 번째 단계에서는 첫 번째 단계에서 구분한 메타데이터와 프로그램의 바이너리 코드를 입력 값으로 (그림 4)의 Strata[6]을 사용하여 오류를 검출한다. Strata는 바이너리 코드를 동적으로 변환하여 호스트 CPU에서 실행한다. 새로운 명령어 주소가 시작되면 먼저, 변환된 적이 있는 주소(fragment cache)인지 검사하고, fragment cache인 주소라면 호스트 CPU에서 명령을 실행한다. fragment cache가 아닌 주소라면 fragment cache로 변환한 후 호스트 CPU에서 명령을 실행한다. MEDS는 buffer overflow, Uninitialized data reads, double-free, 해제된 메모리 접근 및 취약성을 검출하는 반면 운영체제에서 발생하는 시그널 처리, 커널 레벨 멀티쓰레딩 등을 무시한다.



(그림 4) Strata의 개요

3. 메모리 오류 가능성 검출을 위한 명령어 전이도 도출

본 장에서는 메모리 오류 가능성 검출을 위해 실행 파일을 역어셈블한 어셈블리어 코드에 기반을 두어 각 메모리

오류별 유형을 파악하고, 명령어 전이도를 도출한다. 3.1절에서는 Local Memory Return Error의 명령어 전이도를, 3.2절에서는 Null Pointer Access Error의 명령어 전이도를, 3.3절에서는 Uninitialized Pointer Access Error의 명령어 전이도를 각각 도출한다.

3.1 Local Memory Return Error의 명령어 전이도 도출

Local Memory Return Error는 함수 내 메모리 영역의 주소 값을 반환하는 것으로 함수가 끝난 후 반환된 주소는 유효하지 않은 영역이 되어 해당 주소에서 읽은 데이터를 사용할 경우 의도하지 않은 데이터로 인해 프로그램의 오작동을 유발할 수 있다.

어셈블리어 코드에서 함수의 반환 값을 처리하는 방법은 callee 함수의 반환 값을 MOV 명령어를 사용하여 EAX 레지스터에 저장을 하고, RET 명령어에 의해 함수가 종료된 후 해당 함수를 호출한 함수(caller)에서 EAX 레지스터에 있는 값을 MOV 명령을 이용하여 임의의 메모리 공간으로 복사 후 이를 사용하는 것이다.

Local Memory Return의 전형적인 형태의 C언어 코드는 (그림 5)과 (그림 6)이다. (그림 5)의 (a)는 배열의 특정 위치 주소 값을 직접 반환하는 형태이고, (그림 6)의 (a)는 배열의 특정 위치의 주소 값을 다른 포인터 변수에 저장한 후 반환하는 형태로 Local Memory Return Error가 발생한다.

```
char *LocalMemory(void){
    char a[32] = "local memory return";
    return &a[0]; (a)
}
```

(그림 5) 직접적인 Local Memory Return의 C언어 코드

```
char *LocalMemory2(void){
    static char a[32] = "local memory return";
    char *b = &a[5]; (a)
    return b;
}
```

(그림 6) 간접적인 Local Memory Return의 C언어 코드

(그림 7)은 (그림 5)의 코드를 컴파일한 후 이를 역어셈블한 어셈블리어 코드이고, (그림 8)는 (그림 6)의 코드를 컴파일한 후 이를 역어셈블한 어셈블리어 코드이다. (그림 7)에서는 함수가 종료하는 명령어인 (그림 7)의 (d)의 RET 명령어 이전에 (그림 7)의 (a)의 LEA 명령어를 이용해 직접적으로 (그림 7)의 (b)의 지역 변수 주소를 (그림 7)의 (c) EAX 레지스터에 할당하고, (그림 8)에서는 (그림 8)의 (b)의 지역 변수 주소를 (그림 8)의 (a) LEA 명령어를 이용하여 (그림 8)의 (c) 임의의 지역 변수에 입력 후 그 지역 변수의 값을 (그림 8)의 (d) EAX 레지스터에 할당 후 (그림 8)의 (e) RET 명령어로 함수를 종료한다.

```
80484f6:   8d 45 e0    (a) lea  (b) -0x20(%ebp),%eax (c)
80484f9:   c9              leave
80484fa:   c3          (d) ret
```

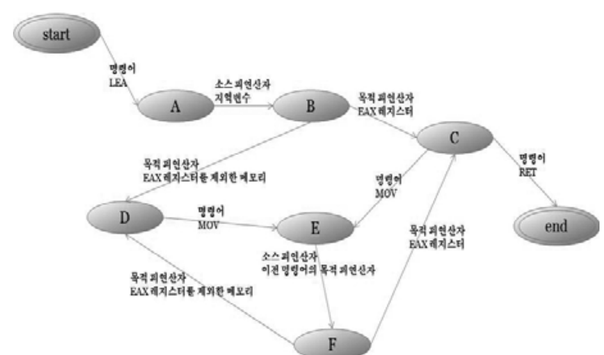
(그림 7) 직접적인 Local Memory Return의 어셈블리어 코드

```
8048445:   8d 45 dc    (a) lea  (b) -0x24(%ebp),%eax
8048448:   83 c0 05        add      $0x5,%eax
804844b:   89 45 fc        mov      %eax,-0x4(%ebp) (c)
804844e:   8b 45 fc        mov      -0x4(%ebp),%eax (d)
8048451:   c9              leave
8048452:   c3          (e) ret
```

(그림 8) 간접적인 Local Memory Return의 어셈블리어 코드

Local Memory Return이 발생하는 유형을 정리하면 (그림 9)의 명령어 전이도로 정의할 수 있다. 상태 start에서 명령어로 LEA가 올 경우 상태 A로 전이되면서 Local Memory Return Error 가능성 검출을 시작한다. 각 상태에서 다른 상태로의 전이에 대한 조건은 다음과 같다.

- 상태 start: 어셈블리어 명령어 중 LEA가 올 경우 상태 A로 전이한다.
- 상태 A: LEA 명령어의 소스 피연산자로 지역 변수가 올 경우 상태 B로 전이한다.
- 상태 B: LEA 명령어의 목적 피연산자로 EAX 레지스터가 올 경우 상태 C로 전이하고, EAX 레지스터를 제외한 다른 피연산자가 올 경우 상태 D로 전이한다.
- 상태 C: 어셈블리어 명령어 중 RET 명령어가 올 경우 상태 end로 전이하고, MOV 명령어가 올 경우 상태 E로 전이한다.
- 상태 D: 어셈블리어 명령어 중 MOV 명령어가 올 경우 상태 E로 전이한다.
- 상태 E: MOV 명령어의 소스 피연산자로 이전 명령어의 목적 피연산자가 올 경우 상태 F로 전이한다.
- 상태 F: MOV 명령어의 목적 피연산자로 EAX 레지스터가 올 경우 상태 C로 전이하고, EAX 레지스터를 제외한 메모리 영역이 올 경우 상태 D로 전이한다.
- 상태 end: 종료 상태이며, 상태 C에서 전이된 경우 Local Memory Return Error의 가능성이 있다고 판단한다.



(그림 9) Local Memory Return의 명령어 전이도

3.2 Null Pointer Access Error의 명령어 전이도 도출

Null Pointer Access Error는 포인터 변수를 Null(0x0)로 초기화한 후 해당 변수를 사용하는 경우에 발생한다. 일반적으로 Null Pointer Access Error는 직접적인 Null 참조보다는 동적 메모리 할당, 파일 핸들링 등에서 반환된 값을 Null 검사 없이 사용함으로써 발생할 확률이 높다. Null Pointer Access Error가 발생하면 해당 프로그램이 종료하거나 컴퓨터의 전체 시스템이 멈춘다.

(그림 10)과 (그림 11)는 각각 직접적인 Null 참조의 C언어 소스 코드와 어셈블리어 코드이다. (그림 10) 및 (그림 11)의 (a)에서 지역 변수에 Null을 직접 할당하고, (b)에서 Null이 할당된 지역 변수를 참조하여 에러가 발생한다. (그림 11)의 (a)에서 지역 변수에 저장된 값(Null)을 (그림 11)의 (b)에서 ESP 레지스터에 저장된 값이 가리키는 주소(callee 함수의 첫 번째 인자에 해당하는 주소)에 할당하고 (그림 11)의 (c)에서 puts 함수를 호출한다. 이 시점에 발생하는 Null에 대한 직접적인 참조로 인해 프로그램의 작동이 멈추게 된다.

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

(그림 10) 직접적인 Null Pointer Access의 C언어 코드

```
080485f8 <nullPointerAccess>:
 80485f8:   55                          push   %ebp
 80485f9:   89 e5                       mov    %esp,%ebp
 80485fb:   83 ec 18                    sub    $0x18,%esp
 80485fe:   c7 45 fc 00 00 00 00    (a) movl   $0x0,-0x4(%ebp)
 8048605:   8b 45 fc                (b) mov    -0x4(%ebp),%eax
 8048608:   89 04 24                    mov    %eax,(%esp)
 804860b:   e8 04 fe ff ff          (c) call   8048414 <puts@plt>
 8048610:   c9                          leave
 8048611:   c3                          ret
 8048612:   90                          nop
 8048613:   90                          nop
```

(그림 11) 직접적인 Null Pointer Access의 어셈블리어 코드

일반적으로 프로그램 작성 시 Null Pointer에 대한 직접적인 접근보다 동적 메모리 할당 함수인 malloc 함수와 파일 제어를 위해 호출하는 fopen 함수와 같은 반환 값이 포인터형인 함수의 호출 이후 그 반환 값에 대해 접근하는 것이 더 빈번하다. 실제 동적 메모리 할당은 OS에서 제공하는 라이브러리를 사용하는데 이 라이브러리는 메모리를 100% 할당해 줄 수 있도록 설계되어 있어 개발자가 의도적으로 Null 검사를 하지 않고 사용하는 경우가 있다. 메모리 블록의 오류 또는 메모리 누수로 인하여 힙 영역의 메모리 여유 공간이 부족하면 메모리 할당 실패로 인해 Null이 반환되고 이를 참조할 경우 Null Pointer Access Error가 발생한다. 목적 파일이 부재인 상태에서 fopen 함수를 호출하고 그 반환 값을 Null 검사 없이 사용할 경우 역시 Null Pointer Access Error가 발생한다.

어셈블리어 언어에서 Null 검사에 사용하는 명령어는 CMP 명령어와 TEST 명령어이다. CMP 명령어는 두 피연산자의 차를 이용하여 목적 피연산자와 소스 피연산자를 비교하는 명령어이고 지역 변수와의 비교에 주로 사용한다. TEST 명령어는 소스 피연산자와 목적 피연산자를 논리 곱 연산 후 결과만 FLAGS 레지스터에 기록을 하는 명령어이고 레지스터와의 비교에 주로 사용한다.

CMP 명령어를 사용하여 Null 검사를 하는 유형은 일반적으로 많이 사용하는 형태인데 C언어 코드 상에서 반환 값을 지역변수에 저장하고 이에 대해 Null 검사를 하는 코드이다. (그림 12)는 그 유형을 나타내는 C언어 코드이다. (그림 12)의 (a)에서 malloc 함수로 할당받은 포인터를 dict 변수에 대입하고, (그림 12)의 (b)에서 dict 변수를 Null과 비교한다. (그림 12)의 코드를 역어셈블하여 어셈블리어 코드로 변환하면 (그림 13)의 코드가 된다. (그림 13)의 (a)에서 malloc 함수를 호출 후 지역 변수((그림 13)의 (b))에 그 결과를 저장하고, (그림 13)의 (c)에서 CMPL 명령어를 이용하여 Null(0x0)과 비교한다.

```
(a) dict = (voca**)malloc(sizeof(voca*) * max_line);
(b) if(dict == NULL){puts("[오류] 메모리확보가 불가능합니다.."); exit(1);}
```

(그림 12) 동적 메모리 할당 후 Null 검사 유형 C언어 코드

```
804883f:   e8 a0 fe ff ff    (a) call   80486e4 <malloc@plt>
8048844:   89 45 f4              mov    %eax,-0xc(%ebp)  (b)
8048847:   83 7d f4 00       (c) cmpl   $0x0,-0xc(%ebp)
```

(그림 13) 동적 메모리 할당 후 Null 검사 유형 어셈블리어 코드

TEST 명령어를 사용하여 Null 검사를 하는 유형은 구조체 내의 특정 위치나 포인터 배열을 사용하는 유형으로 Null 검사의 목적 주소를 특정 레지스터에 저장하고, 그 레지스터를 TEST 명령어로 검사한다. (그림 14)는 TEST 명령어를 사용하여 Null 검사를 하는 첫 번째 유형의 C언어 코드이다. (그림 14)의 (a)에서 malloc 함수로 할당 받은 메모리를 dict 배열의 i번째 위치에 저장하고 (그림 14)의 (b)에서 Null과 비교한다. (그림 15)는 (그림 14) 코드의 어셈블리어 코드이다. (그림 15)의 (a)와 같이 포인터 배열 내의 특정 위치를 계산하기 위한 과정을 거친 후 그 주소를 (그림 15)의 (a) 마지막에 위치하는 EBX 레지스터에 저장하고, (그림 15)의 (b)에서 호출한 malloc 함수의 반환 값을 (그림 15)의 (c)에서 EBX 레지스터가 가리키는 주소에 대입한 후 다시 (그림 15)의 (d) 과정을 통해 주소를 계산하고 (그림 15)의 (e)에서 TEST 명령어로 Null 검사를 한다.

```
(a) dict[i] = (voca*)malloc(sizeof(voca)); //메모리 할당.
(b) if(dict[i] == NULL){puts("[오류] 메모리확보가 불가능합니다.."); exit(1);}
```

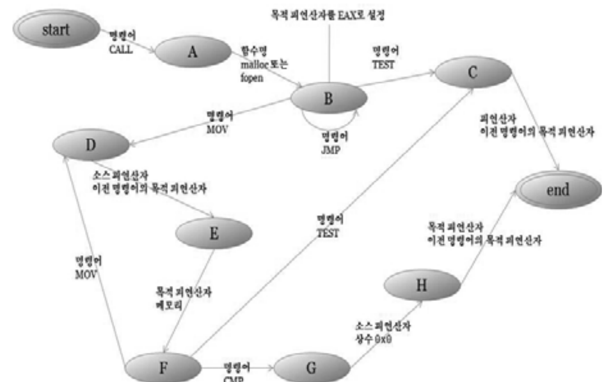
(그림 14) TEST 명령어 유형 C언어 코드

```
8048884:  8b 45 e4              (a) mov    -0x1c(%ebp),%eax
8048887:  c1 e0 02                  shl    $0x2,%eax
804888a:  89 c3                     mov    %eax,%ebx
804888c:  03 5d f4                  add    -0xc(%ebp),%ebx
804888f:  c7 04 24 44 00 00 00      movl   $0x44,(%esp)
8048896:  e8 49 fe ff ff        (b) call   80486e4 <malloc@plt>
804889b:  89 03                 (c) mov    %eax,(%ebx)
804889d:  8b 45 e4              (d) mov    -0x1c(%ebp),%eax
80488a0:  c1 e0 02                  shl    $0x2,%eax
80488a3:  03 45 f4                  add    -0xc(%ebp),%eax
80488a6:  8b 00                     mov    (%eax),%eax
80488a8:  85 c0                 (e) test   %eax,%eax
```

(그림 15) TEST 명령어 유형 어셈블리어 코드

파일제어를 목적으로 포인터를 확보하기 위해 사용하는 fopen 함수의 경우도 malloc 함수의 사용과 차이점이 존재하지 않는다. 즉 malloc 함수 이후에 이루어지는 유형과 동일한 과정을 통해 Null을 검사한다. 동적 메모리 할당 함수인 malloc 함수 또는 파일 열기 함수인 fopen 함수의 반환값에 대해 Null 검사를 한 경우의 어셈블리어의 유형을 분석하여 도출된 명령어 전이도는 (그림 16)과 같다. 역어셈블된 어셈블리어를 함수의 시작부분부터 순차적으로 검사하여 명령어 전이도의 패턴을 벗어나는 경우를 Null Pointer Access Error의 가능성이 있다고 판단한다. 상태 start에서 명령어로 CALL이 올 경우 상태 A로 전이되면서 Null 검사가 존재하는지에 대한 검출을 시작한다. 각 상태에서 다음 상태로의 전이 조건은 다음과 같다.

- 상태 start: 어셈블리어 명령어 CALL이 올 경우 상태 A로 전이한다.
- 상태 A: 호출될 함수 명이 malloc 또는 fopen 일 경우 상태 B로 전이한다.
- 상태 B: 목적 피연산자로 EAX 레지스터를 설정한 이후 다음 명령어가 JMP, TEST, MOV일 경우 각각 상태 B, 상태 C, 상태 D로 전이한다.
- 상태 C: TEST 명령어의 피연산자가 이전 명령어의 목적 피연산자가 올 경우 상태 end로 전이한다. 이때 이전 명령어의 목적 피연산자는 대부분 EAX 레지스터가 온다.
- 상태 D: MOV 명령어의 소스 피연산자로 이전 명령어의 목적 피연산자가 올 경우 상태 E로 전이한다.
- 상태 E: MOV 명령어의 목적 피연산자로 일반적 메모리 영역이 올 경우 상태 F로 전이한다.
- 상태 F: 어셈블리어 명령어로 MOV가 올 경우 상태 D로 전이하고, CMP가 올 경우 상태 G로 전이한다.
- 상태 G: CMP 명령어의 소스 피연산자로 상수 0x0(Null)이 올 경우 상태 H로 전이한다.
- 상태 H: CMP 명령어의 대상 피연산자로 이전 명령어의 목적 피연산자가 올 경우 상태 end로 전이한다.
- 상태 end: 상태 C 또는 상태 H에서 전이된 경우 정상적 Null 검사로 보고 프로그램 코드 상 해당 메모리 영역에 접근하는 경우 정상적 접근으로 판단한다.



(그림 16) Null 검사가 존재하는 경우의 명령어 전이도

3.3 Uninitialized Pointer Access Error의 명령어 전이도 도출

Uninitialized Pointer Access Error는 초기화를 하지 않았거나 데이터가 설정이 되지 않은 메모리 영역에 접근하는 경우에 발생한다. 할당받은 힙 영역의 메모리나 스택 내 메모리(지역 변수)를 초기화 또는 값 설정 없이 데이터를 읽을 경우 의도하지 않은 데이터가 읽힐 수 있고, 잘못된 데이터가 원인이 되어 비즈니스 로직상의 오작동이 발생할 수 있다.

메모리의 초기화는 bzero함수나 memset 함수 같은 초기화 함수를 이용하는 방법과 직접 데이터를 초기화하는 방법이 있다. (그림 17)는 초기화 함수를 이용하여 메모리를 초기화하는 방법의 C언어 코드이고 (그림 18)는 이를 컴파일한 후 역어셈블한 어셈블리어 코드이다. (그림 17)에서 bzero함수를 이용하여 size 크기만큼 temp를 초기화한다. (그림 18)의 (a)에서 MOV 명령어를 이용하여 지역변수에 저장된 값을 EAX 레지스터에 저장하고, (b)에서는 (a)에서 EAX 레지스터에 저장된 값을 호출할 callee 함수의 두 번째 인자(ESP 레지스터 기준 +0x4 위치)에 복사한다. (c)에서 마찬가지로 지역 변수의 값을 EAX 레지스터로 복사한 후 (d)에서 호출할 함수의 첫 번째 인자(ESP 레지스터 기준 +0x0 위치)에 복사하고 (e)에서 bzero 함수를 호출하여 초기화를 수행한다. bzero 함수는 첫 번째 인자에 입력된 주소를 시작주소로 하여 두 번째 인자의 크기만큼을 초기화하는 함수이다.

```
bzero(temp, size);
```

(그림 17) 메모리 초기화 유형

```
804bb81:  8b 45 fc         (a) mov    -0x4(%ebp),%eax
804bb84:  89 44 24 04      (b) mov    %eax,0x4(%esp)
804bb88:  8b 45 f8         (c) mov    -0x8(%ebp),%eax
804bb8b:  89 04 24         (d) mov    %eax,(%esp)
804bb8e:  e8 19 cd ff ff   (e) call   80488ac <bzero@plt>
```

(그림 18) 메모리 초기화 유형 1 어셈블리어 코드

메모리를 초기화하는 다른 방법으로는 (그림 19)의 코드와 같은 데이터를 직접 설정하는 방법이 있다. (그림 19)에서 지역 변수 word_cnt에 0을 직접 설정하는 것으로 초기화를 한다. 이를 역어셈블한 어셈블리어 코드인 (그림 20)의 (a)에서는 0x0을 복사하여 지역변수를 초기화한다.

```
int word_cnt = 0;    //단어복사가 성공한 단어를 갯수를 카운트.
```

(그림 19) 메모리 초기화 유형 2

```
0x804880b: c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%ebp)
```

(그림 20) 메모리 초기화 유형 2 어셈블리어 코드

초기화 함수를 사용하는 유형에서 초기화 대상인 메모리가 malloc 함수로 동적 할당 받은 힙 영역의 메모리 이외에 (그림 21)와 같이 배열을 초기화하는 경우도 있다. (그림 21)의 (a)에서 배열을 할당한 다음 (b)에서 bzero 함수를 이용해 초기화한다. 이를 역어셈블한 어셈블리어 코드는 (그림 22)이다. (그림 22)의 (a)에서 LEA 명령어를 이용하여 지역변수를 주소를 (b)에서 함수의 첫 번째 인자로 설정 후 (c)에서 bzero 함수를 호출하여 초기화한다. 앞의 2가지 유형과 비교해보면 명령어만 MOV 명령어에서 LEA 명령어로 교체가 되었고, 그 외의 다른 부분은 MOV 명령어를 사용할 때와 동일하다.

```
(a) char savedArea[64];
(b) bzero(savedArea, 64);
```

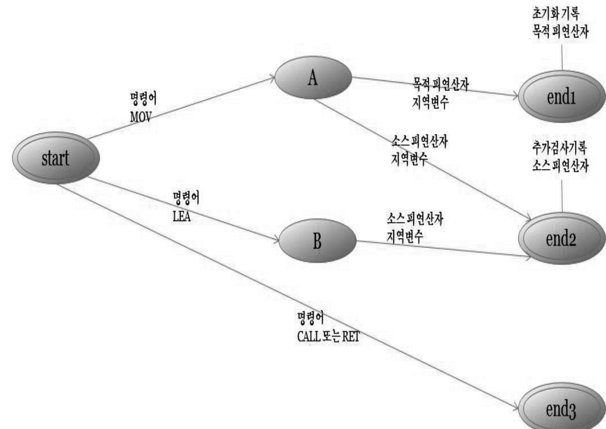
(그림 21) 지역 변수의 초기화 유형

```
0x804b1ed: 8d 85 18 ff ff ff    (a) lea    -0xe8(%ebp),%eax
0x804b1f3: 89 04 24             (b) mov    %eax,(%esp)
0x804b1f6: e8 b1 d6 ff ff       (c) call   80488ac <bzero@plt>
```

(그림 22) 지역 변수의 초기화 어셈블리어 코드

2가지의 메모리 초기화 유형을 기반으로 도출한 명령어 전이도는 (그림 23)과 같다. 상태 start에서 명령어로 CALL, LEA, MOV가 올 경우 각각 상태 A, 상태 G, 상태 J로 전이되고, Uninitialized Pointer Access Error의 가능성 검출이 시작한다. 각 상태에서 다음 상태로 전이되는 조건은 다음과 같다.

- 상태 start: 어셈블리어 명령어 MOV가 올 경우 상태 A로 전이되고, LEA가 올 경우 상태 B로 전이되고, CALL 또는 RET가 올 경우 상태 end3으로 전이된다.
- 상태 A: 어셈블리어 명령어 MOV의 목적 피연산자가 지역변수가 올 경우 상태 end1로 전이되고, 소스 피연산자가 지역변수인 경우 상태 end2로 전이된다.
- 상태 B: 어셈블리어 명령어 LEA의 소스 피연산자가 지역변수인 경우 상태 end2로 전이된다.
- 상태 end1: 본 상태에 도달하게 되면 목적 피연산자에 초기화했다고 기록한다.
- 상태 end2: 본 상태에 도달하게 되었을 경우 소스 피연산자에 추가 검사가 필요하다고 기록한다.
- 상태 end3: 본 상태에 도달한 경우 추가 검사가 필요하다고 기록한 지역 변수들에 대해 검사를 실시한다.



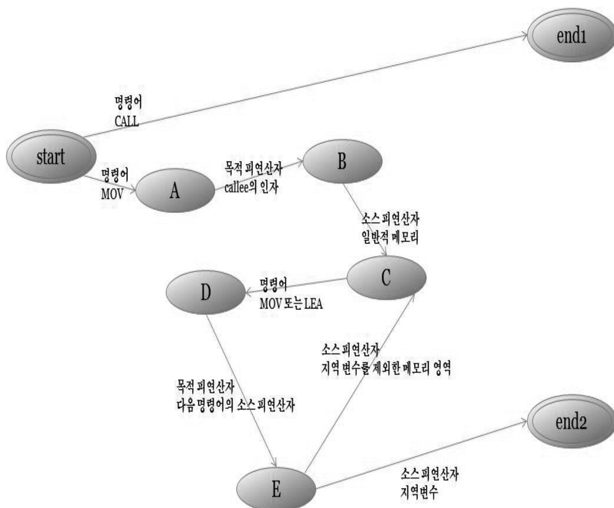
(그림 23) Uninitialized Pointer Access Error에서의 명령어 전이도

(그림 23)의 상태 end3은 함수를 호출하여 메모리를 초기화하는 경우에 도달하는 상태이다. 이때 초기화 대상으로 선택되는 인자는 호출되는 함수가 무엇인지에 따라 달라지며 초기화 대상이 되는 인자를 결정하고 초기화 되었다고 기록을 하는 단계가 추가적으로 필요하다. (그림 24)는 초기화 대상이 되는 인자를 결정하기 위해 추가적인 검사에서 사용하는 명령어 전이도이다. 각 상태에서 다음 상태로 전이하는 조건은 다음과 같다.

- 상태 start: 어셈블리어 명령어 CALL이나 RET 또는 함수의 시작부분이 올 경우 상태 end1로 전이하고 MOV가 올 경우 상태 A로 전이한다.
- 상태 A: 어셈블리어 명령어 MOV의 목적 피연산자가 callee의 인자일 경우 상태 B로 전이한다.
- 상태 B: 어셈블리어 명령어 MOV의 소스 피연산자가 일반 메모리 영역일 경우 상태 C로 전이한다.
- 상태 C: 어셈블리어 명령어 MOV 또는 LEA가 올 경우 상태 D로 전이한다.
- 상태 D: 어셈블리어 명령어 MOV 또는 LEA의 목적 피연산자가 다음 명령어의 소스 피연산자일 경우 상태 B로 전이한다.
- 상태 E: 어셈블리어 명령어의 소스 피연산자가 지역 변수일 경우 상태 end2로 전이하고, 지역 변수가 아닐 경우 상태 F로 전이한다.
- 상태 end1: 본 명령어 전이도를 시작할 때 명령어에 따라 몇 가지 동작을 행한다.
  - 명령어 RET: 추가 검사로 기록된 포인터에 대해

Uninitialized Pointer Access Error의 가능성이 있다고 판단한다.

- 명령어 CALL: 추가 검사로 기록된 포인터 중 end2 상태에서 함수의 인자로 설정한 지역 변수를 제거하고 남은 지역 변수에 대해 Uninitialized Pointer Access Error의 가능성이 있다고 판단한다. 이 때 함수가 무엇이냐에 따라 제거하는 인자의 위치가 달라진다.
  - 첫 번째 인자에 데이터를 기록하는 함수: memset, strcpy와 같은 함수의 경우 첫 번째 인자로 기록된 지역 변수를 제거한다.
  - 두 번째 인자 이후에 데이터를 기록하는 함수: scanf 함수의 경우 두 번째부터 n번째까지의 인자로 기록된 지역 변수를 제거한다.
  - 세 번째 인자 이후에 데이터를 기록하는 함수: sscanf함수 또는 fscanf함수의 경우 세 번째부터 n 번째까지의 인자로 기록된 지역 변수를 제거한다.



(그림 24) Uninitialized Pointer Access Error 추가 검사 명령어 전이도

4. 실험 결과

본 논문에서 성능을 분석하기 위하여 실험한 데이터는 컴퓨터 관련 학과에 재학 중인 학생들이 과제로 제출한 소스코드 중 일부(대상 1~5)와 본 논문에서 구현한 검출 도구(대상 6), 오픈 소스 프로젝트인 아파치 웹 서버[7]의 실행 파일 중 httpd(대상 7)과 htpasswd(대상 8), PHP 스크립트 해석기[8]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 툴로 역어셈블하였다. 실험에 사용한 시스템의 상세한 사양은 <표 3>와 같다.

〈표 3〉 실험에 사용한 시스템의 상세 사양

프로세서	인텔 Centrino Duo T2300 @ 1.66 GHz
메모리	Cache: 2048 KB System: 2048 MB Swap: 1012 MB
하드디스크	SATA 80 GB 5400 RPM
운영체제	Fedora 10 (Kernel: 2.6.27.38-170)

Local Memory Return Error는 어셈블리어 명령어가 3.1절에서 정의한 (그림 9)의 명령어 전이도를 따르는 경우 오류라고 판단한다. <표 4>와 같이 학생 대상의 발생 비율 중 대상 4번이 5.56%로 가장 높은 비율을 차지하고, 나머지의 경우 0%의 발생 비율을 차지하였다. 오픈 소스의 경우는 Local Memory Return Error가 발생한 함수가 없다.

〈표 4〉 Local Memory Return Error 가능성 검출 결과

분류	구분	함수 수 (a)	Local Memory Return 발생 수 (b)	발생 비율 (b / a * 100)
학생 대상	대상 1	45	0	0.00%
	대상 2	80	0	0.00%
	대상 3	18	1	5.56%
	대상 4	14	0	0.00%
	대상 5	53	0	0.00%
	대상 6	65	0	0.00%
오픈 소스 대상	대상 7	1,405	0	0.00%
	대상 8	4	0	0.00%
	대상 9	8,101	0	0.00%
계		9,785	1	

Null Pointer Access Error는 3.2절에서 정의한 (그림 16)의 명령어 전이도에서 상태 end3에 도달하지 못한 메모리 영역이 소스 피연산자가 될 경우 오류라고 판단한다. Null Pointer Access Error의 검출은 malloc 함수와 fopen 함수에 대해 실험을 진행하였으며, malloc 함수에 대한 Null Pointer Access Error의 검출 결과는 <표 5>와 같다. 학생 대상에서 높은 비율의 메모리 오류 가능성이 발생한 원인은 malloc 함수의 호출 후 Null 검사가 이루어지지 않은 상태에서 대상 포인터에 대해 여러 차례 접근을 하였기 때문이다. 즉 발생 비율은 100%라 할 수 있다.

〈표 5〉 Null Pointer Access Error 가능성 검출 결과 (malloc)

분류	구분	malloc 함수 호출 수 (a)	Null 검사 없이 접근한 횟수 (b)	발생 비율 (b / a * 100)
학생 대상	대상 1	10	3	30.00%
	대상 2	5	11	220.00%
	대상 3	2	4	200.00%
	대상 4	6	29	483.33%
	대상 5	29	18	62.07%
	대상 6	20	2	10.00%
오픈 소스 대상	대상 7	12	2	16.67%
	대상 8	0	0	0.00%
	대상 9	177	161	90.96%
계		261	230	

fopen에 대해 Null Pointer Access Error를 검출한 결과는 <표 6>과 같다. 학생 대상 프로그램 중 대상 3과 4에서 fopen 함수 호출 후 Null 검사 없이 접근을 하였고, 대상 1, 대상 2, 대상 5, 대상 6의 경우는 fopen 함수의 호출 이후 Null 검사를 하였다. 오픈 소스 프로그램의 경우 대상 7과 대상 8에서는 fopen 함수를 사용하지 않았고, 대상 9에서만 15번의 fopen 함수를 호출하였고, 그 중 9번을 Null 검사 없이 접근하였다.

〈표 6〉 Null Pointer Access Error 가능성 검출 결과 (fopen)

분류	구분	fopen 함수 호출 수 (a)	Null 검사 없이 접근한 횟수 (b)	발생 비율 (b / a * 100)
학생 대상	대상 1	1	0	0.00%
	대상 2	6	0	0.00%
	대상 3	6	2	33.33%
	대상 4	5	6	120.00%
	대상 5	3	0	0.00%
	대상 6	3	0	0.00%
오픈 소스 대상	대상 7	0	0	0.00%
	대상 8	0	0	0.00%
	대상 9	15	9	60.00%
계		39	17	

Uninitialized Pointer Access Error는 3.3절에서 정의한 (그림 23)의 명령어 전이도에서 상태 end1에 도달하지 못한 메모리 영역이 소스 피연산자로 올 경우와 상태 end2에 도달 후 (그림 24)의 명령어 전이도에 의해 추가 검사를 하여 적합한 초기화 과정을 거치지 않고 소스 피연산자로 올 경우 오류가 발생할 수 있다고 판단한다. Uninitialized Pointer Access Error 가능성 검출 결과는 <표 7>과 같다. 전체 대상 프로그램에서 초기화 없이 메모리를 읽은 횟수가 5천여 회 나왔으며, 평균적으로 함수 당 0.5개의 오류 발생 가능성이 검출되었다.

〈표 7〉 Uninitialized Pointer Access Error 가능성 검출 결과

분류	구분	함수 수 (a)	초기화 없이 메모리를 읽은 횟수(b)	함수 당 오류 수 (b / a)
학생 대상	대상 1	45	34	0.8
	대상 2	80	164	2.0
	대상 3	18	26	1.4
	대상 4	14	37	2.6
	대상 5	53	127	2.4
	대상 6	65	46	0.7
오픈 소스 대상	대상 7	1,405	1,939	1.4
	대상 8	4	40	10.0
	대상 9	8,101	2,714	0.3
계		9,785	5,127	0.5

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 오류에 의해 프로그램의 중단되거나 시스템의 오작동알 유발하여 사용자에게 피해를 발생시킬 수 있다.

본 논문에서는 실행 파일 및 라이브러리 파일을 역어셈블하여 만들어진 어셈블리어를 기반으로 Local Memory Return Error, Null Pointer Access Error, Uninitialized Pointer Access Error에 대해 메모리 사용의 정상적 유형 및 비정상적 유형에 대한 명령어 전이도를 정의하고, 이를 기반으로 학생들의 소스코드, 오픈 소스인 아파치 웹 서버, PHP 프로그램을 대상 프로그램으로 정상적 유형의 명령어 전이도를 벗어나거나 비정상적 명령어 전이도를 따르는지에 따라 메모리 오류의 가능성을 검출하였다.

향후 본 논문에서 다루지 못한 오류 중 프로그램 분석 기법을 통해 검출이 가능한 오류를 검출하는 방법에 대해 추가적인 연구를 할 계획이고, 어셈블리어 분석에 기반을 둔 보안 관련 취약성 검출에 대한 연구를 계속할 계획이다.

참 고 문 헌

[1] 정영범, 김재황, 신재호, 이광근, “아이락: C 프로그램의 메모리 오류 정적 분석기", 한국정보과학회 2005 가을 학술발표대회 문집 32(2), pp. 964~966 2005.11

[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] Kevin Scott and Jack Davidson, “Strata: A Software Dynamic Translation Infrastructure”, In Proceedings of the Workshop on Binary Translation (WBT '01), Barcelona, Spain, September 2001.

[7] Apache Web Server, "http://httpd.apache.org", 1995.

[8] PHP Scripting Language, "http://www.php.net", 1995.

김 현 수

e-mail : kim.hyunsoo@se.kumoh.ac.kr

2008년 금오공과대학교 컴퓨터공학부 (학사)

2010년 금오공과대학교 대학원 소프트웨어 공학과(공학석사)

2010년~현 재 금오공과대학교 대학원 소프트웨어공학과 박사과정

관심분야 : 인공지능, 소프트웨어공학, 디자인 패턴

김 병 만

e-mail : bmkim@kumoh.ac.kr

1987년 서울대학교 컴퓨터공학과(학사)

1989년 한국과학기술원 전산학과(공학석사)

1992년 한국과학기술원 전산학과(공학박사)

1992년~현 재 금오공과대학교 교수

1998년~1999년 미국 UC, Irvine 대학 방문교수

2005년~2006년 미국 콜로라도 주립대학 방문교수

관심분야 : 인공지능, 정보검색, 정보보안

배 현 섭

e-mail : hsbae@suresofttech.com

1993년 KAIST 전산학과 졸업(학사)

1995년 KAIST 전산학과 졸업(공학석사)

1999년 KAIST 전산학과 졸업(공학박사)

1999년~2000년 한국전자통신연구원 (ETRI) 선임연구원

2001년~2002년 매크로임팩트 책임연구원

2002년~현 재 슈어소프트테크(주) 대표이사

관심분야 : 소프트웨어 테스팅, 소프트웨어 안전성, 신뢰성 등

정 인 상

e-mail : insang@hansung.ac.kr

1987년 서울대학교 공과대학 컴퓨터공학과(학사)

1989년 KAIST 전산학과 석사(소프트웨어 공학전공)

1993년 KAIST 전산학과 박사(소프트웨어 공학전공)

1994년~1999년 한림대학교 교수

1999년~현 재 한성대학교 컴퓨터공학과 교수

관심분야 : 소프트웨어 테스팅