C언어 기반 프로그램의 동적 메모리 접근 오류 테스트 자동화 도구 설계

(Design of an Automated Testing Tool to Detect Dynamic Memory Access Errors in C Programs)

조 대 완 † 오 승 욱 †† 김 현 수 †††
(Dae Wan Cho) (Seung Uk Oh) (Hyeon Soo Kim)

요 약 메모리 접근연산으로부터 발생되는 프로그램 오류는 C언어로 작성된 테스트 대상 프로그램에서 가장 빈번하게 발생하는 오류이다[1,2]. 기존연구를 통해 이런 문제점을 해결하기 위한 다양한 메모리 오류 자동검출 방법들이 제안되었다. 하지만 기존의 오류검출방법은 테스트 대상 프로그램에 가해지는 부가적인 오버헤드가 매우 크거나 검출할 수 있는 메모리 접근오류의 종류가 제한적이다. 또한 메모리 할당 함수의 내부구현에 의존성을 갖고 있기 때문에 플랫폼 간 이식성(portability)이 떨어지는 단점을 갖고 있다. 본 연구에서는 이러한 문제점을 해결하기 위해 새로운 메모리 접근오류 검출기법을 제안하고 테스트 자동화 도구를 설계하였다.

키워드 : 메모리 접근 오류, 메모리 오손, 메모리 테스트, 테스트 자동화

Abstract Memory access errors are frequently occurred in computer programs written in C programming language [1,2]. Accordingly, a number of research works have suggested a wide variety of methods to detect such errors automatically. However, they have one or more of the following problems: inability to detect all memory errors, changing the memory allocation mechanism, and excessive performance overhead. To cope with these problems, in this paper we suggest a new and automated tool to detect dynamic memory access errors in C programs.

Key words : memory access error, memory corruption, memory test, test automation

1. 서 론

C언어로 작성된 테스트 대상 프로그램에서 동적 메모리 접근 오류를 자동으로 검출할 수 있는 다양한 테스트 자동화 도구가 기존의 연구를 통해 소개된 바 있다. 이런 도구들은 소스코드의 확장을 통해 포인터 연산의 유효성을 검사하거나 컴파일 된 이진코드의 분석을 통해 프로그램에 나타나는 로드/스토어 명령의 유효성을 검사한다.

소스코드의 의미 분석 및 코드확장을 통한 방법은 이진 코드 분석을 수행하는 방법에 비해 다양한 종류의 동적 메모리 접근 오류를 검사할 수 있다. 그러나 기존의 구현방법들은 메모리 할당 함수의 내부구현에 의존성을 갖고 있기 때문에 테스트 대상 프로그램의 플랫폼 변경에 대해 효과적으로 대처할 수 없다[3-7]. 또한 포인터 변수에 대한 메타데이타를 유지하기 위해 테스트 대상 프로그램에 가해지는 실행파일 크기 및 실행시간의 공간 및 성능 오버헤드가 크다.

이진 코드의 분석을 통한 방법은 비록 앞서 소개한 방법에 비해 테스트 대상 프로그램에 가해지는 실행시간 오버헤드가 적지만 바이트 단위의 메모리 접근에 대한 유효성 검사를 수행하기 때문에 검출할 수 있는 메모리 접근 오류가 제한적이다. 또한 이진 코드의 분석을 통한 방법 역시 메모리 할당 함수의 내부구현에 의존성을 갖기 때문에 테스트 대상 프로그램의 플랫폼 변경에 대해 효과적으로 대처할 수 없다[8,9].

뿐만 아니라, 메모리 할당함수의 내부구현에 의존적인 기존의 방법들은 시스템의 메모리 구성방법이 매우 다양한 내장형 시스템 등에서 각 시스템에 맞는 메모리 할당함수를 별도로 구현해야 하기 때문에 도구의 플랫

† 학생회원 : 충남대학교 컴퓨터공학과
oopmania@cnu.ac.kr
†† 정 회 원 : 슈어소프트테크(주) 연구소
suoh@suresofttech.com
††† 종신회원 : 충남대학교 컴퓨터공학과 교수
hskim401@cnu.ac.kr
논문접수 : 2007년 4월 25일
심사완료 : 2007년 6월 28일

폼간 이식성(portability)이 떨어지는 단점을 갖고 있다.

본 연구에서는 앞서 살펴본 두 가지 접근방법을 효과적으로 혼용하여 이진코드 분석 방법에 비해 오류검출 성능이 뛰어나고, 소스코드 분석 방법에 비해 테스트 대상 프로그램에 가해지는 실행파일 크기 및 실행시간의 공간 및 성능 오버헤드를 최소화 할 수 있는 동적 메모리 접근오류 테스트 자동화 도구를 설계한다. 또한 본 연구에서 설계한 테스트 자동화 도구는 메모리 할당 함수의 내부구현에 독립적이기 때문에 플랫폼 변경에 따른 적용이 매우 간편하다.

본 논문의 구성은 다음과 같다. 2장에서는 기존연구를 통해 소개된 테스트 자동화 도구에 대해 살펴본다. 3장에서는 본 연구에서 설계한 메모리 테스트 자동화 도구에 대해 기술한다. 4장에서는 본 연구결과에 대한 실험 및 평가를 기술한다. 마지막으로 5장에서는 결론 및 향후 연구과제를 기술한다.

2. 관련연구

2.1 R.Jones와 P.Kelly의 메모리 오류 검사기법

R. Jones와 P. Kelly가 제안한 메모리 오류 검사기법은 소스코드의 의미 분석 및 확장을 통해 C언어로 작성된 테스트 대상 프로그램의 메모리 접근 오류를 검사한다[5]. 이 도구에서는 테스트 대상 프로그램의 동적 메모리 할당요청에 대해 그림 1과 같은 형태로 동적 메모리 영역을 관리하도록 메모리 할당 함수를 구현하였다.

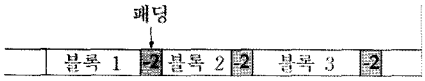


그림 1 패딩이 삽입된 메모리 영역

패딩이 삽입된 메모리 영역에 대한 접근연산은 경계를 벗어난 메모리 접근 오류로 간주된다. 그러나 이 방법에서는 패딩 영역을 넘어 의도하지 않은 인접 메모리 블록에 접근하는 연산 오류는 검출할 수 없다. 또한 패딩의 삽입을 위해 메모리 블록 당 최소 4바이트의 공간 오버헤드를 감수해야 한다.

이 방법에서는 각각의 동적 메모리 블록에 대해 그림 2와 같은 레코드로 구성된 메타데이타를 스플레이(splay) 트리에 저장한다.

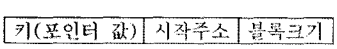


그림 2 메모리 블록의 메타데이타

그림 2에서 레코드의 키 값은 해당 메모리 블록을 가리키는 포인터 변수의 최근 값이다. 또한 시작주소와 블록크기는 메모리 할당함수를 통해 생성된 동적 메모리 블록의 주소영역을 나타낸다. 그림 2의 레코드는 테스트 대상 프로그램의 실행시간에 메모리 블록 당 최소 12바이트의 공간 오버헤드를 일으킨다. 또한 키 값의 유지를 위해 산술연산에 의한 포인터 값 변경이 일어날 때(pointer arithmetic)마다 레코드 값 갱신을 수행해야 한다. 만약 테스트 대상 프로그램에서 포인터 변수 간의 앨리어싱(aliasing)이 일어나는 빈도가 매우 높은 경우 각 메모리 블록에 대한 메타데이타는 해당 블록을 가리키는 포인터 변수마다 모두 생성되어야 하기 때문에 테스트 대상 프로그램에 가해지는 성능 및 공간 오버헤드는 더욱 증가한다.

2.2 O.Ruwase와 M.Lam의 메모리 오류 검사기법

2.1절에서 살펴본 R.Jones와 P.Kelly의 오류검사기법은 포인터 산술연산에 의해 영역을 벗어난 포인터 변수에 패딩 값(-2)을 저장한다. 만약, 패딩 값이 저장된 포인터 변수가 이후에 일어나는 산술연산의 r-value로 사용된다면 해당 연산의 결과로 인해 테스트 대상 프로그램의 오동작을 일으킬 수 있다. O.Ruwase와 M.Lam은 영역을 벗어난 포인터 변수가 그림 3과 같이 OOB(out-of-bounds)객체라 불리는 데이타 블록을 가리키도록 기존의 방법을 확장함으로써 앞의 문제점을 해결하였다[6].

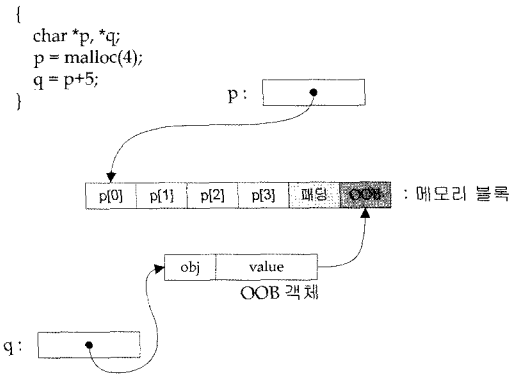


그림 3 O.Ruwase와 M.Lam의 메모리 오류 검사기법

2.3 Rational Purify

IBM사의 Rational Purify는 C언어로 작성된 테스트 대상 프로그램으로부터 다양한 동적 메모리 오류를 자동으로 검사할 수 있는 대표적인 상용화 도구이다. 이 도구에서는 테스트 대상 프로그램의 목적코드에 대한 분석을 통해 메모리 영역에 대한 로드/스토어 명령에 대해 바이트 단위 접근 오류를 검사한다[8].

Rational Purify는 각각의 바이트 단위 메모리 블록마다 2비트 코드 값으로 해당 블록의 상태를 저장한다. 테스트 대상 프로그램에 할당된 메모리 블록의 바이트 단

위 상태를 저장하기 위해 Rational Purify는 별도의 메모리 할당 함수를 구현하였다.

비록 Rational Purify가 다른 도구에 비해 다양한 메모리 오류를 검출할 수 있지만 동적 메모리에 대한 접근 오류 검출에서는 소스코드의 의미 분석을 통해서만 알 수 있는 포인터 변수의 명시적인 형 변환에 의한 메모리 오손(memory corruption)과 의도하지 않은 포인터 변수에 의한 메모리 접근(abuse of pointer) 오류를 검출할 수 없다[5].

2.4 GNU Valgrind

C언어로 작성된 테스트 대상 프로그램에서 메모리 오류를 검사하는 Valgrind는 이진 코드에 대한 분석을 통해 테스트 대상 프로그램의 메모리 오류를 검사한다는 점에서 IBM사의 Rational Purify와 매우 유사한 테스트 자동화 도구이다[9]. 따라서 Valgrind 역시 소스코드의 의미 분석을 통해서만 알 수 있는 포인터 변수의 명시적인 형 변환에 의한 메모리 오손(memory corruption)과 의도하지 않은 포인터 변수에 의한 메모리 접근(abuse of pointer) 오류를 검출할 수 없으며 별도로 구현된 메모리 할당 함수를 사용해야 하는 단점을 극복하지 못했다.

3. 메모리 테스트 자동화 도구 설계

그림 4는 본 연구에서 설계한 메모리 테스트 자동화 도구의 구성이다.

그림 4에서 코드변환도구는 구문삽입을 통해 입력으로 주어진 테스트 대상 프로그램의 소스코드를 확장한다. 확장된 소스코드는 컴파일 이후 오류검사를 수행하는 공유 라이브러리 파일과 링크되고, 테스트환경하에서 사용자 입력 또는 테스트데이타 생성기에 의한 입력에 의해 수행되고 실행시간에 동적 메모리 접근오류를 보고한다.

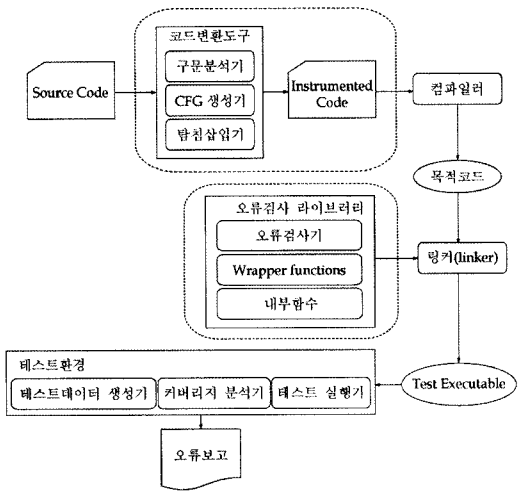


그림 4 메모리 테스트 자동화 도구의 구성

3.1 코드변환도구

코드변환도구는 테스트 대상 프로그램의 소스코드에 대한 전처리 과정을 통해 메모리 접근 오류를 검사하기 위한 추가적인 구문을 삽입한다. 먼저, 코드변환 과정에서는 테스트 대상 프로그램의 각 포인터 변수 p에 대응되는 추가적인 포인터 변수 'origin of p'(이하, o_p)의 선언 및 정의 구문을 삽입한다. 추가된 포인터 변수 o_p는 포인터 변수 p에 할당된 동적 메모리 블록의 시작주소를 가리킨다. 만약 포인터 변수 p에 동적 메모리 블록의 주소 값이 할당되지 않았거나 이전에 가리키던 메모리 블록이 해제된 이후에는 널 값을 갖도록 유지된다. 포인터 변수 o_p의 선언 및 정의 구문을 삽입하기 위해 구문분석기는 포인터 변수 p의 상태를 추적한다. 이 때 각 포인터 변수 p의 상태는 표 1과 같이 정의한다.

표 1 포인터 변수 p의 상태정의

상태	설명
$P_{uninitialized}$	초기화되지 않은 포인터
$P_{initialized}$	명시적으로 값이 지정된 포인터
$P_{allocated}$	동적 메모리 블록의 시작주소로 값이 지정된 포인터
$P_{modified}$	포인터 변수에 대한 산술연산에 의해 값이 변경된 포인터

구문분석기는 포인터 변수 p의 상태변화에 따라 아래 상태 다이어그램의 액션(action) 및 액티비티(activity) 부분에 해당하는 구문을 삽입한다(그림 5).

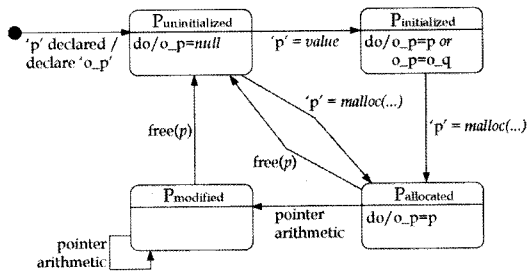


그림 5 'origin of p' 선언 및 정의 규칙

구문 삽입 규칙에 따라 테스트 대상 프로그램에 코드 삽입을 수행한 결과는 그림 6과 같다.

그림 7은 포인터 변수의 앨리어싱(aliasing)이 일어난 경우의 코드변환 방법을 보이고 있다.

그림 7(b)에서 포인터 변수 q의 값이 p에 할당될 때 포인터 변수 p의 상태는 $P_{initialized}$ 상태로 전이되면서

```
int *p;
...
p = (int*)malloc(...);
...
p++;
...
free(p);
```

```
int *p;
int *o_p = null;
...
p = (int*)malloc(...);
o_p = p;
...
p++;
...
free(p);
o_p = null;
```

(가) 원본코드 (나) 변환된 코드

그림 6 'origin of p'의 선언 및 정의

```
int *q=(int*)malloc(...);
int* p;
...
p = q
...
p++;
...
free(p);
```

```
int *q=(int*)malloc(...);
int *o_q = q;
int *p;
int *o_p = null;
...
p = q
o_p = o_q;
...
p++;
...
free(p);
o_p = null;
o_q = null;
```

(a) 원본코드 (b) 변환된 코드

그림 7 앨리어싱(aliasing)에 따른 코드 변환

o_q의 값이 o_p의 값에 복사되고 포인터 변수 q의 상태가 $P_{allocated}$상태이므로 포인터 변수 p의 상태 역시 $P_{allocated}$상태로 전이된다.

테스트 대상 프로그램의 소스코드에 대한 전처리 과정에서 삽입되는 또 다른 구문은 메모리 접근 연산의 직전에 삽입되는 메모리 오류 검사 API 함수의 호출구문이다. 실행시간에 수행되는 메모리 오류 검사 API 함수는 표 2와 같다.

첫 번째로 _check_memory()함수는 포인터 변수를 이용한 메모리 블록 접근 연산의 직전에 삽입되며 해당 연산이 메모리 오류를 일으키는 연산인지 검사한다. 테스트 대상 프로그램의 소스코드에 대한 전처리 과정에서 표 3과 같은 형태의_check_memory()함수 호출구문을 메모리 접근 연산의 직전에 삽입한다.

표 2 메모리 오류 검사 API함수

함수명	인자(argument) 목록	
	인자명	의미
_check_memory	o_p	포인터 변수 p에 할당된 메모리 블록의 시작주소
	p	메모리 접근 연산의 대상 주소
_check_free	o_p	포인터 변수 p에 할당된 메모리 블록의 시작주소
	p	메모리 해제함수(free)에 전달되는 주소 값

표 3 _check_memory() 함수 호출구문의 삽입

연산구문	API함수 호출구문
array[i]	_check_memory(o_array, array+i);
*(++p)	_check_memory(o_p, p+1);
*(--p)	_check_memory(o_p, p-1);
*(p+i)	_check_memory(o_p, p+i);

for(..;*p;p++){…}	_check_memory(o_p, p);
	for(..; *p; p++)
	{ …
	_check_memory(o_p, p+1);
	}

strcpy(buf, "hello")	_check_memory(o_buf, buf); _check_memory(o_buf, buf+strlen("hello"));

_check_free()함수는 메모리 블록을 해제하기 위한 free()함수 호출구문 직전에 삽입되며 free()함수에 전달되는 메모리 블록의 주소 값이 메모리 해제 오류를 일으키는지 검사한다(그림 8).

```
void *p = malloc(...);
free(p);
```

```
void *p = malloc(...);
void *o_p = p;

_check_free(o_p, p);
free(p);
o_p = null;
```

(a) 원본코드 (b) 변환된 코드

그림 8 _check_free() 함수 호출구문의 삽입

함수 간 호출에서 포인터 변수의 값이 전달되는 경우 o_p의 선언 및 유지 구문과 API 함수 호출구문은 그림 9와 같이 삽입된다.

그림 9에서 함수 foo()는 메모리 블록의 주소 값을 인자 값으로 전달받는다. 이 때 메모리 오류 검사 API함수의 호출구문은 인자 값으로 메모리 주소 값을 전달받는 함수의 호출 직전에 삽입된다. 함수의 호출직전에 메모리 오류를 검사하고 나면 호출된 함수 내부에서는 전달받은 포인터 변수 p를 함수 내에서 선언된 지역변수와 동일하게 취급할 수 있다. 따라서 전달받은 포인터 변수 p에 대해 o_p를 선언하고 포인터 변수 p의 값을 할당한다.

3.2 오류검사 라이브러리

오류검사 라이브러리는 테스트 대상 프로그램의 메모리 할당 및 해제요청을 감시하기 위해 표준 C라이브러리의 메모리 할당 및 해제함수를 래핑(wrapping)하는

```
foo (int *arr, int idx) {
  if (idx > 0)
    foo(arr,idx-1);
  arr[idx] = idx;
}

goo(void) {
  int *buf = malloc(...);
  ...
  foo(buf, 4);
  ...
}
```

(a) 원본코드

```
foo (int *arr, int idx) {
  int *o_arr = arr;
  if (idx > 0)
  {
    _check_memory(o_arr, arr);
    foo(arr,idx-1);
  }
  _check_memory(o_arr, arr+idx);
  arr[idx] = idx;
}

goo(void) {
  int *buf = malloc(...);
  int *o_buf = buf;
  ...
  _check_memory(o_buf, buf);
  foo(buf, 4);
  ...
}
```

(b) 변환된 코드

그림 9 형식인자로 선언된 포인터 변수

표 4 래퍼(wrapper) 함수

함수명	형식인자
cm_malloc	size_t size, const void* caller
cm_free	void* ptr, const void* caller

표 4의 함수 군을 제공한다.

표 4의 cm_malloc/cm_free함수는 테스트 대상 프로그램 및 링크된 라이브러리의 malloc/free함수 호출 시 그림 10과 같은 순서로 호출된다.

그림 10의 테스트 대상 프로그램에서 ①,③,⑤에 나타난 메모리 할당/해제 함수의 호출에 따라 표준 C 라이브러리 함수의 동작이 수행되며, 표준 C 라이브러리 함수는 그림 10의 ②,④,⑥과 같이 콜백 형태로 래퍼 함수를 호출한다. 이 때, 메모리 블록의 할당 및 해제는 표준 C 라이브러리 함수에 의해 수행되며, 래퍼 함수는 단지 변경된 동적 메모리 할당 정보를 내부 자료구조인 인덱스-비트맵에 기록한다. 즉, 본 연구에서는 시스템이 제공하는 동적 메모리 할당함수의 내부 메커니즘을 변경하지 않는다.

소스코드 분석에 기반한 기존의 연구에서는 테스트 대상 프로그램에 할당되는 각 동적 메모리 블록의 구성을 변경한다[4-7]. 따라서 기존의 방법들을 다양한 플랫폼에 적용하기 위해서는 반드시 각 플랫폼의 특성에 따라 별도의 동적 메모리 할당함수를 구현해야 한다. 반면, 본 연구에서는 앞서 밝힌 바와 같이 동적 메모리 할당의 내부 메커니즘을 변경하지 않기 때문에 플랫폼 간 이식성이 기존의 연구에 비해 뛰어난 장점을 갖고 있다.

또한, 본 연구에서는 래퍼 함수가 표준 C 라이브러리 함수에 의해 콜백 형태로 호출되기 때문에 그림 10과 같이 공유 라이브러리 내에서 일어나는 모든 동적 메모리의 할당 정보를 유지할 수 있다. 따라서 공유 라이브러리로부터 테스트 대상 프로그램에 반환된 동적 메모리 블록에 대한 접근 연산을 모두 검사할 수 있다.

그림 11은 동적 메모리 영역의 구성 정보를 기록하기 위한 인덱스-비트맵의 구성이다. 동적 메모리 블록의 주소 값에서 상위 10비트는 인덱스-비트맵의 상위 인덱스 오프셋으로 사용된다. 상위 인덱스의 각 항목은 연관된 중간인덱스 블록을 가리킨다. 주소 값의 중간 10비트는 중간 인덱스의 오프셋으로 사용된다. 또한 중간인덱스의 각 항목은 연관된 비트맵 블록의 시작주소를 가리킨다. 비트맵 블록은 하나의 메모리 페이지에 대응되는 비트맵 필드들을 포함한다. 메모리 페이지의 크기가 4KB인

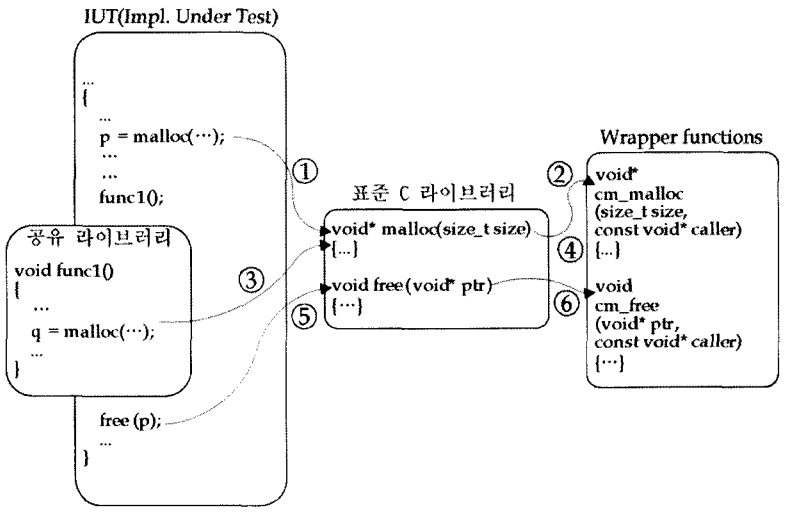


그림 10 메모리 할당/해제 함수의 래핑(wrapping)

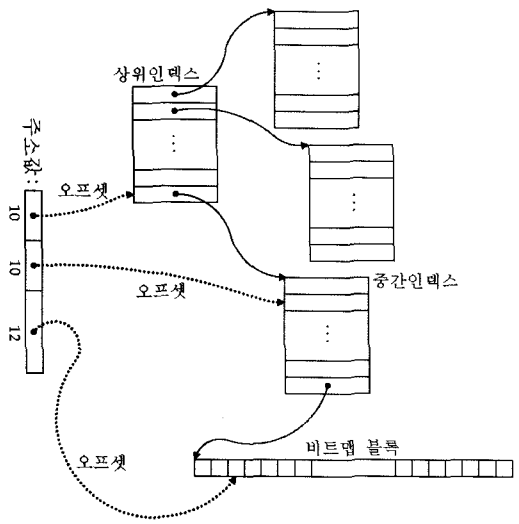


그림 11 인덱스 비트맵의 구성

시스템에서 각 페이지에 대응되는 비트맵 블록의 크기는 최대 1KB이다. 비트맵 블록의 크기는 메모리 할당함수로부터 반환되는 동적 메모리 블록의 최소 크기에 따라 달라진다. 표 5는 동적 메모리 블록의 최소 크기에 따라 비트맵 블록에 소요되는 공간 오버헤드(overhead)를 보이고 있다.

표 5 비트맵 블록의 공간 오버헤드

최소메모리블록의 크기	비트맵블록의 크기
1Byte	1KB/Page(4KB)
Half Word(2Byte)	512Byte /Page
Full word(4Byte)	256Byte/Page
Double word(8Byte)	128Byte/Page

메모리 할당 함수로부터 새로운 메모리 블록이 할당되면 생성된 메모리 블록의 시작주소에 대응되는 비트맵 블록을 인덱스-비트맵으로부터 검색한다. 또한 생성된 메모리 블록의 시작주소에서 하위 12비트에 해당하는 값을 검색된 비트맵 블록에서 해당 메모리 블록의 할당 정보를 기록할 비트맵 필드의 오프셋으로 사용한다. 생성된 메모리 블록의 주소에 대응되는 비트맵 필드를 검색한 후 해당 비트맵 필드로부터 메모리 블록의 크기에 해당되는 연속된 비트맵 필드들은 인접한 비트맵 필드의 값과 구분되도록 컬러링된다. 각 비트맵 필드에 기록되는 2비트 이진코드 값은 표 6과 같다.

동적 메모리 블록을 하나도 할당 받지 않은 테스트 대상 프로그램의 비트맵 블록은 그림 12(a)와 같이 모두 W값으로 초기화되어 있다. 이 때 테스트 대상 프로그램이 요청한 블록크기 5인 메모리 블록이 생성되면

표 6 비트맵 필드 값

할당여부	1st bit	2nd bit	Color
X	0	0	W(white)
O	0	1	R(red)
O	1	0	G(green)
O	1	1	B(blue)

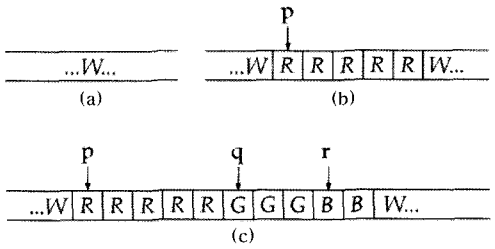


그림 12 비트맵 컬러링

그림 12(b)와 같이 할당된 영역의 주소범위에 대응되는 비트맵 블록은 인접 블록과 구분되도록 R값으로 기록된다. 그림 12(c)는 테스트 대상 프로그램이 각각 블록크기 5, 3, 2에 해당하는 동적 메모리 블록을 생성한 경우 각 메모리 블록에 대응되는 비트맵 필드의 값을 인접 블록과 구분되도록 컬러링 값을 기록한 모습이다.

3.3 오류검사기

소스코드 확장단계에서 삽입된 오류검사 함수는 테스트 대상프로그램의 실행시간에 메모리 접근 및 해제연산에 대한 유효성을 검사한다. 그림 13은 _check_memory() 함수에 의한 메모리 접근오류의 검사 단계를 나타낸다.

동적 메모리에 대한 접근오류 검사는 그림 13에 나타난 바와 같이 다음의 여섯 단계로 수행된다.

① 두 번째 인자 값으로 전달된 접근연산의 대상 메모리 주소 값이 힙 영역의 주소범위에 포함되는지 검사하고, 그렇지 않은 경우 오류검사를 중단한다.

② 접근연산의 대상 메모리 주소 값이 널인 경우 널 포인터에 의한 연산오류를 보고하고 오류검사를 중단한다.

③ o_p값이 널인 경우 잘못된 포인터 값에 의한 접근오류를 보고하고 오류검사를 중단한다.

④ 인덱스-비트맵으로부터 o_p와 p의 주소 값에 대응되는 컬러링 값을 검색하고 c1과 c2에 각각 저장한다.

⑤ 두 개의 컬러링 값인 c1과 c2가 서로 다른 경우 경계를 벗어난 접근오류를 보고하고 오류검사를 중단한다.

⑥ c1과 c2가 동일하지만 컬러링 값이 W인 경우 할당되지 않은 메모리 블록에 대한 접근오류를 보고하고 오류검사를 중단한다.

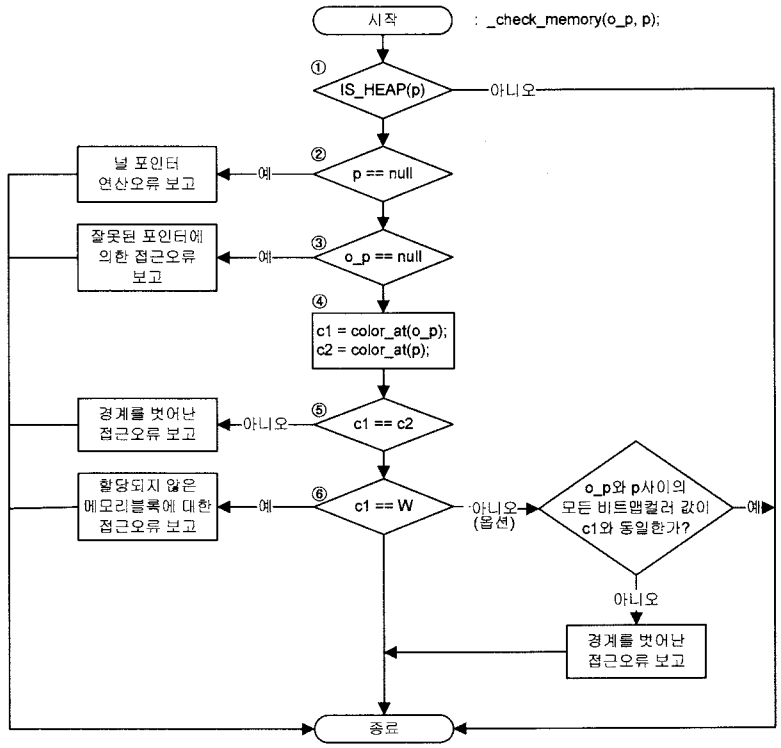


그림 13 접근오류검사 순서도

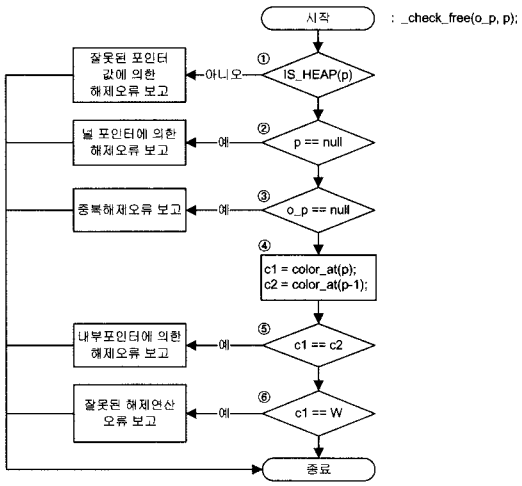


그림 14 해제오류검사 순서도

(옵션) 단계 ⑥에서 c1이 W가 아닌 경우 테스터의 선택에 따라 o_p와 p사이의 주소 값에 해당하는 모든 비트맵 블록이 c1과 동일한 컬러링 값을 갖는지 검사한다.

그림 14는 _check_free()함수에 의한 메모리 해제오류의 검사 단계를 나타낸다.

동적 메모리에 대한 해제오류 검사는 그림 14에 나타난 바와 같이 다음의 여섯 단계로 수행된다.

① 두 번째 인자 값으로 전달된 해제연산의 대상 메모리 주소 값이 힙 영역의 주소범위에 포함되는지 검사하고, 그렇지 않은 경우 잘못된 포인터 값에 의한 해제오류를 보고한 후 오류검사를 중단한다.

② 해제연산의 대상 메모리 주소 값이 널인 경우 널 포인터에 의한 해제오류를 보고하고 오류검사를 중단한다.

③ o_p값이 널인 경우 중복해제오류를 보고하고 오류검사를 중단한다.

④ 인덱스-비트맵으로부터 p와 (p-1)의 주소 값에 대응되는 컬러링 값을 검색하고 c1과 c2에 각각 저장한다.

⑤ 두 개의 컬러링 값인 c1과 c2가 서로 동일한 경우 p 값이 가리키는 메모리 주소가 연속된 메모리 블록의 시작주소가 아니므로 내부포인터에 의한 해제오류를 보고하고 오류검사를 중단한다.

⑥ c1에 저장된 컬러링 값이 W인 경우 잘못된 해제연산 오류를 보고하고 오류검사를 중단한다.

3.3.1 경계를 벗어난 접근 오류

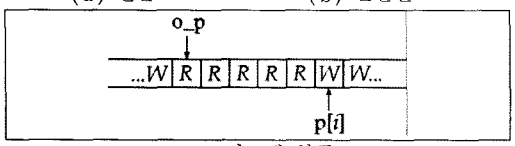
3.3.1절에서 3.3.3절까지는 본 논문에서 제안하는 방법의 능력을 보이기 위해 구체적인 소스 코드를 사례로

```
int i;
int *p =
(int*)malloc(sizeof(int)*5);

for (i=0; i<5; i++);
p[i] = 1;
```

(a) 원본코드

```
int i;
int *p =
(int*)malloc(sizeof(int)*5);
int *o_p = p;

for (i=0; i<5; i++);
_check_memory(o_p, p+i);
p[i] = 1;
```

(b) 변환된 코드



(c) 비트맵 블록

그림 15 경계를 벗어난 접근 오류 검사

설명한다. 그림 15(a)의 C언어 프로그램은 경계를 벗어난 동적 메모리 접근오류를 일으킨다.

전처리 과정을 통해 변환된 그림 15(b)의 코드에서 배열 p의 i번째 블록에 대한 쓰기 연산은 _check_memory() 함수에 의해 메모리 접근 오류로 보고된다. _check_memory() 함수는 그림 15(c)의 비트맵 블록으로부터 첫 번째 인자 값으로 전달된 주소 값에 대응되는 비트맵 블록의 값(R)과 두 번째 인자 값으로 전달된 주소 값에 대응되는 비트맵 블록의 값(W)이 서로 다르기 때문에 p+i번지의 메모리 블록에 대한 접근이 경계를 벗어난 접근 오류라고 판단한다.

그림 16(a)의 C언어 프로그램은 강제 형 변환에 의한 메모리 오손 오류를 일으킨다.

그림 16(a)의 C언어 프로그램은 구조체 형의 포인터 변수 r에 의한 메모리 접근 연산에서 인접한 메모리 블록 q가 가리키는 메모리 블록의 데이타를 손상시킬 수 있다. 그림 16(b)의 변환된 코드는 포인터 변수 r에 의한 메모리 접근 연산 이전에 그림 16(c)의 비트맵 컬러링 정보로부터 메모리 오류를 검사한다. 이 때 _check_memory() 함수는 인자 값으로 전달된 o_r에 대응되는 비트맵 블록의 값(R)과 연산의 대상 주소인 &(r->arr[7])에 대응되는 비트맵 블록의 값(G)이 서로 다르기 때문에 포인터 변수 r에 의한 메모리 접근에 대해 오류를 보고한다.

3.3.2 잘못된 포인터 값에 의한 메모리 접근 오류

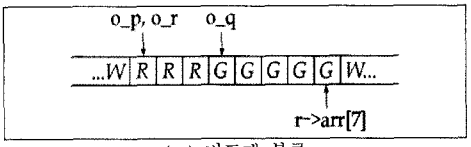
그림 17(a)의 C언어 프로그램에서 포인터 변수 buf1은 동적 메모리 블록이 해제된 이후에 값이 초기화되지 않았다. 이 때 buf2에 할당된 동적 메모리 블록이 우연히 이전에 buf1이 가리키던 주소 값에 해당하는 영역에 생성되었다면 buf1에 의한 메모리 접근연산은 buf2의 동적 메모리 블록에 저장된 내부 데이타 손실을 일으킨다.

```
typedef struct _complex_t { int arr[8]; } complex_t;
...
int *p, *q;
...
{
  p = (int*)malloc(sizeof(int)*3);
  q = (int*)malloc(sizeof(int)*5);
  ...
  complex_t *r = (complex_t*)p;
  r->arr[7] = 1;
      ...
}
```

(a) 원본코드

```
typedef struct _complex_t { int arr[8]; } complex_t;
...
int *p, *q;
int *o_p, *o_q;
...
{
  p = (int*)malloc(sizeof(int)*3);
  o_p = p;
  q = (int*)malloc(sizeof(int)*5);
  o_q = q;
  ...
  complex_t *r = (complex_t*)p;
  complex_t *o_r = (complex_t*)o_p;

  _check_memory(o_r, &(r->arr[7]));
  r->arr[7] = 1;
  ...
}
```

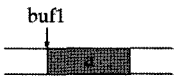
(b) 변환된 코드



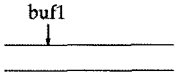
(c) 비트맵 블록

그림 16 형 변환에 의한 메모리 오손 오류 검출

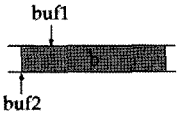
```
int *buf1 = malloc(...);
int *buf2;
```




```
...
free(buf1);
...
```



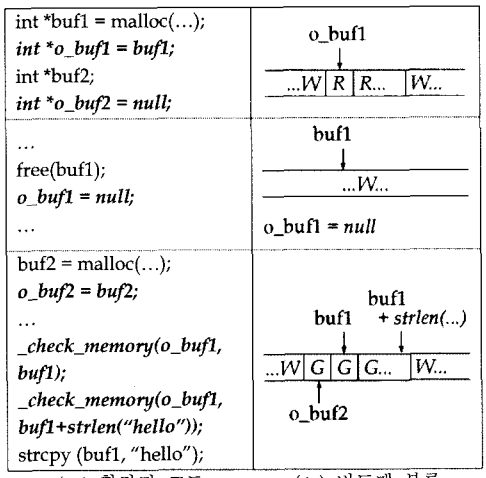

```
buf2 = malloc(...);
...
strcpy (buf1, "hello");
```



(a) 원본코드 (b) 동적 메모리 블록

그림 17 잘못된 포인터 값에 의한 메모리 접근

그림 17(a)의 소스코드는 전처리 과정을 통해 그림 18(a)와 같이 확장된다.



(a) 확장된 코드 (b) 비트맵 블록

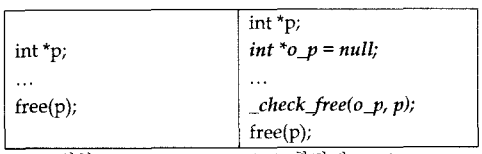
그림 18 잘못된 포인터 값에 의한 접근 검사

확장된 코드에서는 buf1이 가리키는 메모리 블록에 문자열을 복사하기 전에 메모리 오류를 검사한다. 이 때, 두 번째 인자 값으로 전달된 buf1의 주소 값은 유효한 메모리 블록을 가리키고 있다. 그러나 오류 검사 API함수의 첫 번째 인자 값으로 전달된 o_buf1가 널 값을 갖고 있기 때문에 잘못된 포인터 변수에 의한 메모리 오류를 보고할 수 있다.

위에서 살펴본 메모리 오류 이외에 할당되지 않은 메모리 블록에 대한 접근 연산 및 널 포인터에 의한 메모리 접근 연산을 검출할 수 있다. 할당되지 않은 메모리 블록에 대한 접근 연산은 연산의 대상 주소에 대응되는 비트맵 필드의 값이 W인 경우 해당 오류를 보고한다. 또한 연산의 대상 주소 값이 널인 경우 널 포인터에 의한 메모리 접근 연산 오류를 보고한다.

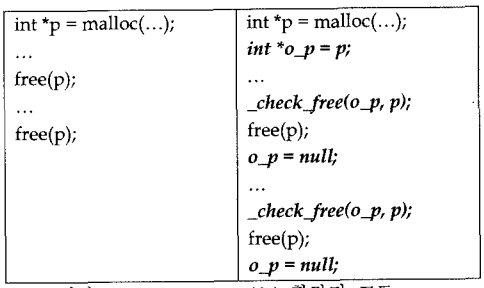
3.3.3 부적절한 동적 메모리 해제 시도

그림 19(a)와 그림 20(a)의 C언어 프로그램은 각각 메모리 블록이 할당되지 않은 포인터 변수 값에 의한 메모리 해제 연산 및 동적 메모리 블록의 중복해제를 시도하고 있다.



(a) 원본코드 (b) 확장된 코드

그림 19 잘못된 해제연산 시도

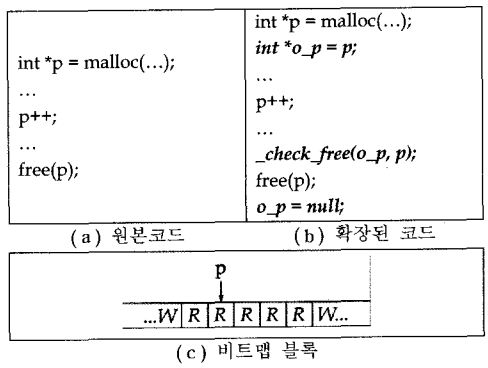


(a) 원본코드 (b) 확장된 코드

그림 20 동적 메모리 블록의 중복해제 시도

위의 테스트 대상 프로그램으로부터 확장된 그림 19(b)의 _check_free()함수와 그림 20(b)의 두 번째 _check_free()함수는 첫 번째 인자로 전달된 값이 널 값이므로 모두 잘못된 포인터 값에 의한 메모리 해제시도 오류를 보고한다.

그림 21(a)의 C언어 프로그램은 동적 메모리 블록의 내부 주소를 통해 메모리 블록의 해제를 시도하고 있다.



(a) 원본코드 (b) 확장된 코드

(c) 비트맵 블록

그림 21 내부 포인터에 의한 해제

그림 21(b)의 _check_free() 함수는 해제할 블록의 주소에 대응되는 비트맵 필드의 값(R)이 연속된 비트맵 필드 영역의 시작이 아니기 때문에 내부 포인터에 의한 메모리 블록 해제오류를 보고한다.

3.4 테스트 환경

본 연구에서 설계한 메모리 테스트 자동화 도구는 테스트 대상 프로그램의 실행 과정에서 발생하는 동적 메모리 접근 오류를 검사한다. 따라서 테스트 대상 프로그램의 수행경로 분석 및 적절한 테스트 데이타의 선택이 매우 중요하다.

그림 22에서 '*(p+i) = 1' 연산이 동적 메모리 접근 오류를 일으키는 경우, 프로그램의 수행경로에 따라 오류가 발견되지 않을 수 있다. 본 연구에서 설계한 메모리 테스트 자동화 도구는 테스트 대상 프로그램의 분기

```
if (condition)
{
    *(p+i) = 1;
}

return 0;
```

그림 22 수행경로분석을 통한 오류발견

및 문장 커버리지 검사를 수행한다. 커버리지 검사 기능을 통해 테스터(tester)는 테스트 대상 프로그램의 수행경로 분석을 자동으로 수행할 수 있으며, 적절한 테스트 데이타의 선택을 위해 수행경로 분석결과를 활용할 수 있다.

4. 실험 및 평가

본 연구에서는 실험을 통해 테스트 대상 프로그램에 가해지는 코드삽입에 따른 실행파일 크기증가와 실행시간의 공간 및 성능오버헤드를 측정하였다. 또한 본 논문의 2.2절에서 살펴본 오류검출방법(이하, JKRM[6])을 동일한 테스트 대상 프로그램에 대해 적용하고 실험결과를 비교하였다.

표 7은 본 연구에서 테스트 대상 프로그램으로 사용한 GNU 오픈소스 프로젝트의 목록이다.

표 7 실험대상 오픈소스 코드

IUT	설명	LOC
grep-2.5.1	패턴검색기	24,734L
enscript-1.6.1	Postscript변환기	23,778L
tar-1.16	Archiving 유틸리티	73,446L
gnupg-1.4.7	암호키 생성도구	117,781L

표 7의 LOC항목은 실험을 위해 사용된 테스트 대상 프로그램의 규모를 나타내기 위해 SLOCCount[10] 유틸리티를 통해 측정하였다.

표 8은 테스트 대상 프로그램의 원본 소스코드로부터 생성된 실행파일과 코드삽입 후 생성된 실행파일의 크기를 비교한 결과이다.

표 8 원본코드와의 실행파일 크기비교

IUT	원본코드	본 연구	증가율(%)
grep-2.5.1	73.2K	211.5K	289%
enscript-1.6.1	184.5K	295K	160%
tar-1.16	259.7K	623.1K	211%
gnupg-1.4.7	817.8K	2.5M	310%

또한, JKRM과 실행파일 크기비교를 수행한 결과는 표 9와 같다.

표 9 JKRM과의 실행파일 크기비교

IUT	JKRM	본 연구
grep-2.5.1	374.2K	211.5K
enscript-1.6.1	580.2K	295K
tar-1.16	1.0M	623.1K
gnupg-1.4.7	2.7M	2.5M

성능 및 공간상의 실행시간 오버헤드를 측정하기 위해 본 연구에서는 표 7의 각 테스트 대상 프로그램을 표 10과 같이 실행하였다.

표 10 IUT 실행구문

IUT	실행구문	설명
grep-2.5.1	dmesg | grep mode	문자열패턴 검색
enscript-1.6.1	enscript -p test.ps test	아스키파일을 Postscript파일로 변환
tar-1.16	tar xvf glibc-2.5.tar	파일묶음해제
gnupg-1.4.7	gpg --gen-key	암호키 생성

각 테스트 대상 프로그램에 대한 실행시간의 공간 오버헤드를 측정하기 위해 원본 소스코드로부터 생성된 실행파일과 코드삽입을 통해 생성된 실행파일에 대해 각각의 전체 실행시간 동안 동적 메모리 사용량을 조사한 결과는 표 11과 같다.

표 11 원본코드와의 동적 메모리 사용량 비교

IUT	원본코드	본 연구	증가율(%)
grep-2.5.1	260K	264K	101.5%
enscript-1.6.1	400K	556K	139%
tar-1.16	132K	264K	200%
gnupg-1.4.7	256K	264K	103.1%

표 12는 JKRM과 본 연구의 동적 메모리 사용량을 비교한 결과이다.

표 12 JKRM과의 동적 메모리 사용량 비교

IUT	JKRM	본 연구
grep-2.5.1	342.7K	264K
enscript-1.6.1	772.9K	556K
tar-1.16	405.1K	264K
gnupg-1.4.7	706.6K	264K

실행시간에 테스트 대상 프로그램에 가해지는 성능오버헤드를 측정하기 위해 각 테스트 대상 프로그램의 평균 실행시간을 측정하였으며 결과는 표 13과 같다.

표 14는 JKRM과 본 연구의 수행시간을 비교한 결과이다.

표 13 원본코드와의 수행시간 비교

IUT	원본코드	본 연구	증가율(%)
grep-2.5.1	0.03s	0.12s	400%
enscript-1.6.1	0.02s	0.06s	300%
tar-1.16	6.92s	7.93s	114.6%
gnupg-1.4.7	24.61s	29.33s	119.2%

표 14 JKRM과의 수행시간 비교

IUT	JKRM	본 연구
grep-2.5.1	0.19s	0.12s
enscript-1.6.1	0.18s	0.06s
tar-1.16	12.19s	7.93s
gnupg-1.4.7	74.64s	29.33s

한편, 본 연구에서는 이진코드 분석에 기반한 메모리 오류검사도구인 GNU Valgrind에 대해서 각 테스트 대상 프로그램의 수행시간을 측정하고 본 연구결과와 [표 15]와 같이 비교를 수행하였다.

표 15 Valgrind와의 수행시간 비교

IUT	Valgrind	본 연구
grep-2.5.1	0.66s	0.12s
enscript-1.6.1	1.46s	0.06s
tar-1.16	15.18s	7.93s
gnupg-1.4.7	299.8s	29.33s

실험결과를 통해 살펴 본 바와 같이 본 연구결과는 기존의 대표적인 소스코드 기반 오류검출방법인 JKRM에 비해 테스트 대상 프로그램의 실행파일 크기, 실행시간의 성능 및 공간 오버헤드가 상대적으로 적다.

본 연구에서 설계한 도구의 동적 메모리 오류검출성능을 JKRM, Purify, Valgrind와 비교하면 표 16과 같다.

표 16 각 도구별 오류검출성능

(O:모두 찾을 수 있음, △:일부 못 찾음, X:전혀 찾지 못함)

	JKRM	Purify/ Valgrind	본 연구
경계를 벗어난 접근	O	△	O
초기화되지 않은 포인터에 의한 접근	△	△	O
널 포인터에 의한 접근	O	O	O
내부 포인터에 의한 해제	O	O	O
중복해제	O	O	O
할당되지 않은 메모리 블록 해제	O	O	O

표 16에서 나타난 바와 같이 JKRM은 초기화되지 않은 포인터 변수가 포인터 산술연산에 의해 유효메모리 블록의 주소 값을 가리키는 경우, 해당 포인터 변수에 의한 메모리 접근연산을 검출할 수 없기 때문에 "초기화되지 않은 포인터에 의한 접근" 오류를 일부 검출할 수 없다. 또한 표 16에 나타난 바와 같이 이진코드 분석에 기반한 Purify와 Valgrind는 "경계를 벗어난 접근"과 "초기화되지 않은 포인터에 의한 접근" 이 유효메모리 블록의 주소범위 내에서 일어나는 경우 오류를 검출할 수 없다.

한편, 표 16에 나타난 바와 같이 본 연구에서는 포인터 변수가 유효한 동적 메모리 블록의 시작주소로 초기화되지 않은 경우 대응되는 o_p 값에 의해 해당 포인터 변수의 상태를 알 수 있다. 따라서 JKRM이 발견할 수 없는 "초기화되지 않은 포인터에 의한 접근"오류를 모두 검출할 수 있다. 또한 본 논문의 3.3.1절과 3.3.2절에서 살펴본 바와 같이 Purify나 Valgrind에서 검출할 수 없는 "경계를 벗어난 접근"과 "초기화되지 않은 포인터에 의한 접근"이 유효메모리 블록의 주소범위 내에서 일어나더라도 정확히 오류를 보고할 수 있다.

테스트 대상 프로그램의 소스코드 분석에 기반한 또 다른 메모리 오류검사도구인 CCured 및 Cyclone은 C 언어로 작성된 프로그램의 형(type) 시스템을 확장함으로써 메모리 오류를 검사한다[11,12]. 따라서 테스트 대상 프로그램의 원본코드를 유지하면서 추가적인 코드삽입을 통해 메모리 오류를 검사하는 본 연구와는 직접적으로 비교할 수 없다. 또한 이러한 오류검사도구에서는 공유라이브러리와 링크된 테스트 대상 프로그램을 비롯해 복잡한 C언어 프로그램에 대해 테스터가 수동으로 코드확장 및 추가를 수행해야 한다[6,7,11,12]. 따라서 공유 라이브러리와의 호환성을 유지하면서 복잡한 C언어 프로그램에 대해 자동으로 오류검사를 수행하는 본 연구결과와 비교할 수 없다.

5. 결론 및 향후연구

본 연구에서는 소스코드 분석에 기반한 C언어 기반 프로그램의 동적 메모리 접근오류 테스트 자동화 도구를 설계하였다. 또한 오픈소스 프로젝트에 대한 실험을 통해 테스트 대상 프로그램에 가해지는 부가적인 오버헤드를 측정하고 기존연구와의 비교를 수행하였다.

기존연구에 비해 본 연구결과가 갖는 장점은 다음과 같다.

- 소스코드 분석을 통해 메모리 접근오류를 검사하기 때문에 기존의 이진코드 분석 방법에 비해 오류검출 성능이 뛰어나다. 특히 기존의 상용화 도구인 Rational Purify 및 GNU Valgrind에서 검출할 수 없는 "형 변환에 의한 메모리 오손(memory corruption)"과 "의도하지 않은 포인터 변수에 의한 유효 메모리 접근(abuse of pointer)" 오류를 검출할 수 있다.

- 테스트 대상 프로그램의 동적 메모리 할당정보를 인덱스-비트맵 형태로 유지하고 포인터 산술연산에 따른 부가적인 연산을 수행하지 않는다. 따라서 기존의 소스코드 분석에 기반한 방법에 비해 소스코드의 변경과 실행시간의 공간 및 성능 오버헤드를 최소화할 수 있다.
- 비트맵 블록에 기록된 컬러링 정보를 비트(bitwise)연산을 통해 비교함으로써 경계를 벗어난 접근 및 할당되지 않은 영역에 대한 접근 연산을 검사한다. 따라서 기존의 소스코드 분석 기법에 비해 오류검사를 수행하기 위한 성능 오버헤드가 적다.
- 공유 라이브러리 내에서 생성되거나 해제되는 동적 메모리 블록의 할당정보를 추적할 수 있다. 따라서 공유 라이브러리와의 호환성 문제를 안고 있는 기존의 연구(e.g., CCured, Cyclone, Safe C 등)에 비해 사용성(usability)과 오류검출 성능이 뛰어나다.
- 메모리 할당 및 해제함수의 내부구현에 독립적이기 때문에 별도의 메모리 관련 함수를 구현해야 하는 기존의 방법에 비해 플랫폼 간 이식성(portability)이 뛰어나다.

본 연구에서는 테스트 대상 프로그램에 나타나는 간접포인터연산에 의한 메모리 접근, 구조체 혹은 공용체 내부의 필드에서 발생하는 경계를 벗어난 접근오류 등을 고려하지 않았다. 또한 스택 및 정적영역의 메모리 블록에 대한 오류검출은 본 논문의 연구범위에서 제외되었다. 따라서 향후 본 연구결과를 바탕으로 C언어로 작성된 프로그램에 나타나는 다양한 메모리 접근연산의 형태를 고려하여 소스코드 분석 및 확장기법을 보완하고, 스택 및 정적 메모리영역에 대한 오류검출을 수행할 수 있도록 메모리 접근오류 검사기능을 추가할 것이다. 또한 본 연구에서 설계한 내용에 대한 구현을 완료하여 커버리지 분석 및 테스트 데이타 자동생성 기능 등이 포함된 메모리 테스트 자동화 도구의 공간 및 성능 오버헤드를 측정하고 기존의 상용화 도구 등에 대해 사용성(usability) 측면에서 비교를 수행할 것이다.

참 고 문 헌

[1] Barton P. Miller, Lars Fredriksen, and Bryan So. "An empirical study of the reliability of Unix utilities," Communications of the ACM, 33(12): 32-44, December 1990.

[2] Mark Sullivan and Ram Chillarege, "Software defects and their impact on system availability - a study of field failures in operation systems," Digest of the 21st International Symposium on Fault Tolerant Computing, pp. 2-9, June 1991.

[3] W. Xu, D. C. DuVarney, and R.Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," Proc. of the 12th ACM SIGSOFT Symp. on the FSE, Oct. 2004, pp. 117-126.

[4] T. Austin, S. Breach, and G. Sohi, "Efficient detection of all pointer and array access errors," In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, June 1994.

[5] R. Jones and P. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," In Proceeding of Third International Workshop On Automatic Debugging, May 1997.

[6] O. Ruwase and M. Lam, "A practical dynamic buffer overflow detector," In Proceedings of the Network and Distributed System Security (NDSS) Symposium, pp. 159-169, Feb. 2004.

[7] D. Dhurjati, V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," In Proceeding of the 28th international conference on software engineering(ICSE), Shanhai, China, May 2006.

[8] R. Hastings and B. Joyce, "Purify: fast detection of memory leaks and access errors," In Proceedings of the Winter USENIX Conference, pp. 125-136, 1992.

[9] Valgrind Online Manual. http://valgrind.org/docs/manual/manual.htm

[10] D. A. Wheeler. Sloccount http://www.dwheeler.com/sloccount

[11] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," ACM Trans. Program. Lang. Syst., 27(3):477-526, 2005.

[12] T. Jim, G. Morriset, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," In Proceedings of the USENIX Annual Technical Conference, pp. 275-288, June 2002.

조 대 완

2006년 충남대학교 컴퓨터공학과 학사 2006년~현재 충남대학교 공과대학 컴퓨터공학과 석사과정. 관심분야는 소프트웨어 테스팅, 소프트웨어 품질관리, 시험자동화

오 승 욱

1996년 한국과학기술원 전산학과 학사 1998년 한국과학기술원 전산학과 석사 1998년~현재 한국과학기술원 전산학과 박사과정. 2001년~현재 슈어소프트테크㈜ 연구소. 관심분야는 소프트웨어 테스팅, 요구사항 검증

김 현 수

1988년 서울대학교 계산통계학과 학사 1991년 한국과학기술원 전산학과 공학석사. 1995년 한국과학기술원 전산학과 공학박사. 1995년~1995년 한국전자통신연구원 Post Doc. 1996년~2001년 금오공과대학교 조교수. 1999년~2000년 Colorado State University 방문교수. 2001년~현재 충남대학교 전기정보통신공학부 부교수. 관심분야는 소프트웨어 테스팅, 소프트웨어 재공학, 컴포넌트 마이닝, 컴포넌트 테스팅, SOA