C언어 기반 프로그램의 소스코드 분석을 이용한 메모리 접근오류 자동검출 기법

조 대 완† · 오 승 욱†† · 김 현 수†††

요 약

잘못된 메모리 접근으로부터 발생되는 오류는 C언어로 작성된 프로그램에서 가장 빈번하게 발생하는 오류이다. 이러한 오류를 자동으로 검출하기 위한 기존의 상용화 도구 및 연구결과는 수행시간에 테스트 대상 프로그램에 가해지는 부가적인 오버헤드가 매우 크거나 검출할 수 있는 메모리 접근오류의 종류가 제한적이다. 본 논문에서는 기존연구의 한계점을 개선한 새로운 메모리 접근오류 검출기법을 제안하고 실험을 통해 기존연구와의 비교분석을 수행하였다. 본 논문은 C언어 기반 프로그램의 소스코드 분석기법에 기반하고 있으며, 테스트 대상 프로그램에 할당된 동적 메모리 블록의 주소 범위에 대해 컬러링 기법을 적용한다. 본 논문에서 제안하는 오류검출기법은 기존의 바이너리 코드 분석기법에 비해 다양한 형태의 메모리 접근오류를 검출할 수 있으며, 테스트 대상 프로그램의 수행시간에 요구되는 메타데이터의 유지 및 갱신연산에 따른 공간 및 성능오버헤드가 기존의 소스코드 분석기법에 비해 개선되었다. 또한 본 논문에서 제안하는 기법은 테스트 대상 프로그램과 공유 라이브러리간의 호환성 문제를 일으키지 않으며, 메모리 할당함수의 내부 메커니즘을 변경하지 않는 특징을 갖고 있다.

키워드 : 메모리 접근 오류, 메모리 테스트, 소스코드 분석, 테스트 자동화

An automated memory error detection technique using source code analysis in C programs

Cho Dae Wan† · Oh Seung Uk†† · Kim Hyeon Soo†††

ABSTRACT

Memory access errors are frequently occurred in C programs. A number of tools and research works have been trying to detect the errors automatically. However, they have one or more of the following problems: inability to detect all memory errors, changing the memory allocation mechanism, incompatibility with libraries, and excessive performance overhead. In this paper, we suggest a new method to solve these problems, and then present a result of comparison to the previous research works through the experiments. Our approach consists of two phases. First is to transform source code at compile time through inserting instrumentation into the source code. And second is to detect memory errors at run time with a bitmap that maintains information about memory allocation. Our approach has improved the error detection abilities against the binary code analysis based ones by using the source code analysis technique, and enhanced performance in terms of both space and time, too. In addition, our approach has no problem with respect to compatibility with shared libraries as well as does not need to modify memory allocation mechanism.

Key Words : Memory Access Error, Memory Test, Source Code Analysis, Test Automation

1. 서 론

C언어는 시스템 자원에 대한 세밀한 조작을 지원할 뿐만 아니라 일반적으로 다른 프로그래밍 언어에 비해 높은 성능을 보장하기 때문에 시스템 프로그래밍 분야에서 가장 널리 사용되는 대표적인 구현 언어이다[1]. 한편, C언어는 강한형 검사(strong type checking)를 지원하는 다른 언어에 비해 오류의 내포 가능성이 매우 높기 때문에 개발과정에서 디버깅 및 테스트에 소요되는 비용이 매우 크다. 특히, 포인터 변수를 통한 잘못된 메모리 접근 연산으로부터 발생되는 메모리 접근 오류는 C언어로 작성된 테스트 대상 프로그램에서 가장 빈번하게 발생하는 오류이다[2, 3].

메모리 접근 오류는 C언어로 작성된 테스트 대상 프로그램의 전체 메모리 영역(스택영역, 정적영역, 힙 영역)에 걸쳐 나타날 수 있다. 한편, 명시적인 요청에 의해 생성 및 해제되는 동적 메모리 블록에서 발생하는 오류는 다른 영역의 메모리 블록에서 발생하는 오류에 비해 종류가 다양하고 디버깅 또는 일반적인 테스트 과정에서 오류를 발견하기가 상

† 정 회 원 : 충남대학교 컴퓨터공학과 석사과정
†† 정 회 원 : 슈어소프트테크(주) 연구원
††† 종신회원 : 충남대학교 전기정보통신공학부 교수
논문접수 : 2007년 3월 31일, 심사완료 : 2007년 7월 7일

대적으로 어렵다[4].

기존의 연구를 통해 동적 메모리 블록에서 발생하는 메모리 오류를 자동으로 검출하기 위한 다양한 방법이 제안되었다. 하지만 기존의 연구결과는 수행시간에 테스트 대상 프로그램에 가해지는 부가적인 오버헤드가 매우 크거나[5, 6, 7] 검출할 수 있는 메모리 오류의 종류가 제한적이다[10, 11].

기존 연구의 문제점을 해결하기 위해 본 논문에서는 C언어로 작성된 테스트 대상 프로그램에 내재된 동적 메모리 접근 오류를 검출하기 위한 새로운 방법을 제안한다. 본 논문에서 제안하는 비트맵 컬러링 기법은 기존 연구결과에 비해 다음과 같은 장점을 갖는다.

- 소스코드 분석을 통해 메모리 접근오류를 검사하기 때문에 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 유지한다. 따라서 기존의 소스코드 분석에 기반한 방법에 비해 소스코드의 변경을 최소화할 수 있으며 비트맵 연산을 통해 오류검사를 수행하기 때문에 수행시간 오버헤드가 적다.
- 공유 라이브러리 내에서 생성되거나 해제되는 동적 메모리 블록의 할당정보를 추적할 수 있다. 따라서 공유 라이브러리와의 호환성 문제를 해결하지 못한 기존의 연구에 비해 사용성(usability)과 오류검출 성능이 뛰어나다.
- 메모리 할당 및 해제함수의 내부구현에 독립적이기 때문에 별도의 메모리 관련 함수를 구현해야 하는 기존의 방법에 비해 이식성(portability)이 뛰어나다.

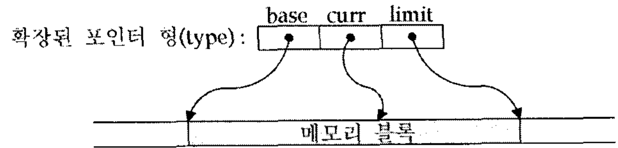
본 논문의 구성은 다음과 같다. 2장에서는 기존의 연구결과를 살펴본다. 3장에서는 C언어로 작성된 프로그램에서 발생할 수 있는 동적 메모리 접근오류에 대해 살펴본다. 4장에서는 본 논문에서 제안하는 비트맵 컬러링 기법을 살펴본다. 5장에서는 본 논문의 연구결과를 기존의 연구결과와 비교한다. 마지막으로 6장에서는 결론 및 향후 연구과제에 대해 기술한다.

2. 관련연구

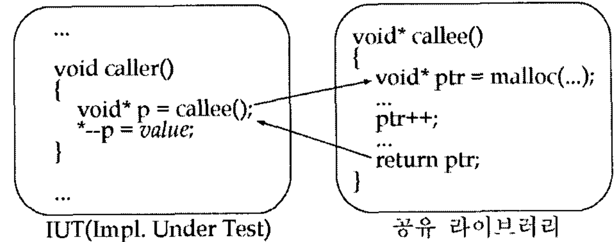
2.1 소스코드 분석을 이용한 오류검출 기법

2.1.1 Safe C Compiler

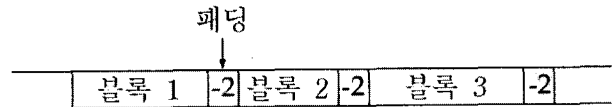
소스코드의 확장과 수행시간 오류검사를 통해 메모리 접근 오류를 검출하기 위한 방법으로써 T.M. Austin, S.E. Breach, G.S.Sohi가 제안한 Safe C Compiler는 테스트 대상 프로그램의 모든 포인터 변수를 메모리 블록에 대한 메타데이터를 포함하도록 (그림 1)과 같이 확장한다[5].



(그림 1) 메타데이터를 포함하는 포인터 형 확장



(그림 2) 함수 호출 간 메타데이터 유지 문제



(그림 3) 패딩이 삽입된 메모리 영역

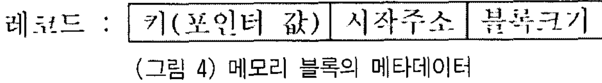
위와 같은 코드 확장 기법은 포인터 자료 형의 크기증가로 인해 테스트 대상 프로그램의 코드 길이가 증가한다. 특히 포인터 변수의 앨리어싱(aliasing)이 많이 일어나는 테스트 대상 프로그램의 경우 코드 길이는 기하급수적으로 증가한다. 또한 위의 방법은 함수 호출 간 전달되는 포인터 변수의 메타데이터 값을 유지하기 어렵기 때문에 (그림 2)와 같은 테스트 대상 프로그램에서 오류가 아닌 메모리 접근연산을 오류로 보고하는 문제(false negative problem)를 일으킨다.

(그림 2)에서 테스트 대상 프로그램의 caller()함수는 공유라이브러리에 정의된 callee()함수로부터 반환된 동적 메모리 블록을 포인터 변수 p에 저장한다. 이 때 공유라이브러리는 이진파일로 제공되기 때문에 callee()함수에서 포인터 형의 확장을 수행할 수 없기 때문에 반환된 메모리 블록의 시작주소와 끝 주소를 caller()함수에서 결정할 수 없다.

2.1.2 확장된 Safe C Compiler

R. Jones와 P. Kelly가 제안한 확장된 Safe C Compiler는 소스코드의 분석 및 확장을 통해 C언어로 작성된 테스트 대상 프로그램의 메모리 접근 오류를 검사한다[6]. 이 도구에서는 테스트 대상 프로그램의 동적 메모리 할당요청에 대해 (그림 3)과 같은 형태로 동적 메모리 영역을 관리하도록 메모리 할당 함수를 구현하였다.

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 공간오버헤드를 감수해야 한다. 확장된 Safe C Compiler에서는 각각의 동적 메모리 블록에 대해 (그림 4)와 같은 레코드로 구성된 메타데이



(그림 4) 메모리 블록의 메타데이터

```
int *p = (int*)malloc(...);
...
free(p);
*p = 1;
```

(그림 5) 해제된 블록에 대한 쓰기연산

터를 스플레이(splay) 트리에 저장한다.

(그림 4)에서 레코드의 키 값은 해당 메모리 블록을 가리키는 포인터 변수의 최근 값이다. 또한 시작주소와 블록크기는 메모리 할당함수를 통해 생성된 동적 메모리 블록의 주소영역을 나타낸다. (그림 4)의 레코드는 테스트 대상 프로그램의 실행시간에 메모리 블록 당 최소 12바이트의 부가적인 메모리 공간이 필요하다. 또한 산술연산에 의한 포인터 값 변경(pointer arithmetic)이 일어나는 경우 메타데이터의 키 값을 유지하기 위해 레코드 값 갱신을 수행해야 한다. 만약 테스트 대상 프로그램에서 포인터 변수 간의 앨리어싱이 일어나는 빈도가 매우 높은 경우 각 메모리 블록에 대한 메타데이터는 해당 블록을 가리키는 포인터 변수마다 모두 생성되어야 하기 때문에 테스트 대상 프로그램에 가해지는 성능 및 공간 오버헤드는 더욱 증가한다.

초기의 Safe C Compiler는 다양한 연구결과를 통해 성능이 개선되었다. 먼저, O. Ruwase와 M. Lam은 포인터 산술연산에 의해 메모리블록의 유효범위를 벗어난 포인터 변수 값이 다른 산술연산에 사용되는 문제를 해결하였다[7]. 또한, D. Dhurjati와 V. Adve은 메타데이터 유지를 위해 사용되는 스플레이 트리를 작은 크기로 분할함으로써 테스트 대상 프로그램에 가해지는 수행시간 오버헤드를 획기적으로 개선하였다[8]. 하지만 이러한 모든 방법들은 시스템이 제공하는 메모리 할당함수의 내부 메커니즘을 변경해야 하기 때문에 플랫폼 간 이식성이 떨어지는 문제점을 여전히 갖고 있다.

2.1.3 CCured

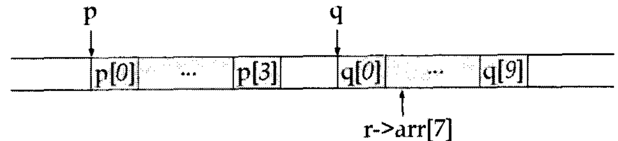
GC Necula, S McPeak, W Weimer가 제안한 CCured는 테스트 대상 프로그램의 원본코드와 의미상으로 동등한(semantically equivalent) 중간코드를 생성하며 생성된 중간코드로부터 의미정보 분석을 통해 컴파일 시간과 수행시간에 걸쳐 메모리 오류를 검사한다[9].

이 방법에서는 공유 라이브러리로부터 참조되는 모든 함수에 대해 사용자가 직접 래퍼함수를 작성해야 하기 때문에 도구의 사용성이 현저히 떨어진다. 또한 메모리 해제연산을 명시적으로 수행하지 않고 Boehm-Weiser Conservative 가비지 콜렉터(garbage collector)에 의한 메모리 블록 해제를 수행하기 때문에 수행시간에 소요되는 가비지 콜렉션에 의한 성능 오버헤드가 매우 크다.

또한, 가비지 콜렉터에 의한 동적 메모리 블록 해제는 CCured의 메모리 오류검사 성능을 저하시키는 주요원인이다. 예를 들어, (그림 5)와 같이 명시적으로 해제된 동적 메모리 블록에 대한 쓰기연산에서 CCured는 동적 메모리 블록 p가 free()함수의 호출에 의해 곧바로 삭제되지 않기 때문에 '*p = 1'구문에 의한 접근오류를 검출할 수 없다.

```
typedef struct _complex_t { int arr[7]; } complex_t;
...
{
complex_t *r = (complex_t*)p;
r->arr[7] = 1;
}
```

(그림 6) 명시적인 형 변환에 의한 메모리 오손 오류



(그림 7) 메모리 오손에 의한 내부데이터 손실

2.2 바이너리 코드 분석을 이용한 오류검출 기법

2.2.1 Rational Purify

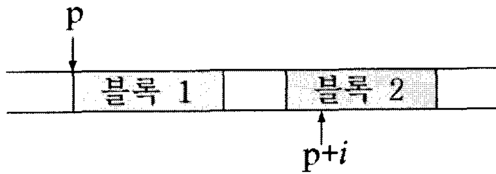
IBM사의 Rational Purify는 C언어로 작성된 테스트 대상 프로그램으로부터 다양한 동적 메모리 오류를 자동으로 검사할 수 있는 대표적인 상용화 도구이다. 이 도구에서는 테스트 대상 프로그램의 목적코드에 대한 분석을 통해 메모리 영역에 대한 로드/스토어 명령에 대해 바이트 단위 접근 오류를 검사한다[10].

Rational Purify는 각각의 바이트 단위 메모리 블록마다 2비트 코드 값으로 해당 블록의 상태를 저장한다. 테스트 대상 프로그램에 할당된 메모리 블록의 바이트 단위 상태를 저장하기 위해 Rational Purify는 별도의 메모리 할당 함수를 구현하였다.

한편 Rational Purify는 소스코드 상에 나타나는 메모리 접근 연산의 의미정보를 알 수 없기 때문에 검출할 수 있는 동적 메모리 접근 오류가 제한적이다. (그림 6)은 명시적인 형 변환에 의한 메모리 오손 오류를 일으키는 C언어 코드이다.

(그림 6)에서 구조체 형의 포인터 변수 r은 명시적인 형 변환에 의해 포인터 변수 p와 동일한 메모리 블록을 가리킨다. 이 때 포인터 변수 p에 할당된 동적 메모리 블록의 길이가 구조체 형의 길이에 비해 짧은 경우 (그림 7)에서와 같이 포인터 변수 r을 이용한 메모리 접근 연산이 메모리 블록 p의 범위를 벗어난 인접 메모리 블록 q의 데이터를 변경시키는 메모리 오손 문제를 일으킨다.

(그림 8)은 의도하지 않은 포인터 변수에 의한 메모리 접근오류를 나타낸다. (그림 8)에서 '블록 1'의 경계를 벗어난 'p+i' 번지에 대한 쓰기 연산은 인접한 '블록 2'의 내부 데이터 손실을 일으킨다. 하지만 'p+i' 번지에 대한 쓰기 연산은 목적코드 상에서 유효한 메모리 블록에 대한 스토어 명령어로 나타나기 때문에 소스코드의 의미정보를 알 수 없는 Rational Purify는 위와 같은 연산에서 오류를 보고할 수 없다.



(그림 8) 의도하지 않은 포인터 변수에 의한 메모리 접근오류

2.2.2 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[11]. 따라서 Valgrind 역시 소스코드의 의미 분석을 통해서만 알 수 있는 포인터 변수의 명시적인 형 변환에 의한 메모리 오손과 의도하지 않은 포인터 변수에 의한 메모리 접근오류를 검출할 수 없으며 별도로 구현된 메모리 할당 함수를 사용해야 하는 단점을 극복하지 못했다.

3. 동적 메모리 접근 오류

메모리 오류검출에 관련된 기존의 연구를 살펴보면 C언어로 작성된 테스트 대상 프로그램에서 발생할 수 있는 전체 메모리 오류는 다음과 같이 분류될 수 있다[1, 3, 5-11].

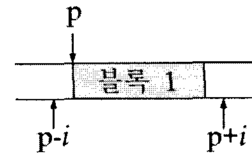
- 경계를 벗어난 메모리 접근
- 널 포인터에 의한 메모리 접근
- 초기화되지 않은 포인터에 의한 메모리 접근
- 잘못된(invalid) 포인터에 의한 메모리 접근
- 동적 메모리 중복해제
- 할당되지 않은 메모리블록 해제
- 메모리 누수
- 스택 오버플로우
- 지역변수에 대한 포인터 반환
- 잘못된 함수포인터에 의한 접근

한편, 동적 메모리 영역에서 발생할 수 있는 메모리 접근오류는 스택 영역이나 함수 포인터에 의한 메모리 오류 및 메모리 누수 오류에 비해 발생빈도가 매우 높다. 이러한 특징은 Bush 등이 Mozilla, Apache, GDI 라이브러리에 대해 수행한 오류유형 분석을 통해 밝혀진 바 있으며, 이 연구에 따르면 본 논문에서 다루고자 하는 동적 메모리 접근오류가 전체 메모리 관련오류의 90%에 달하는 것으로 나타났다[4].

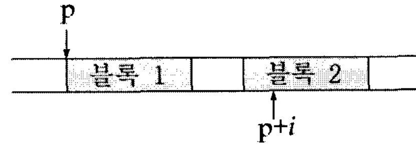
따라서 본 논문에서는 우선 아래와 같은 동적 메모리 접근오류에 대한 검출만을 대상으로 하며, 스택 영역이나 함수 포인터에 의한 메모리 오류 및 메모리 누수에 의한 오류 검출은 향후 연구를 통해 다루도록 한다.

3.1 경계를 벗어난 접근

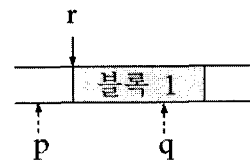
경계를 벗어난 접근 오류는 정상적인 메모리 블록의 유효 범위를 벗어난 영역에 대한 읽기/쓰기 연산에 의해 발생한다. (그림 9)에서 '블록 1'의 주소범위를 벗어난 'p-i'와 'p+i' 번지에 대한 읽기/쓰기 연산은 경계를 벗어난 접근 오류를 일으킨다.



(그림 9) 경계를 벗어난 접근



(그림 10) 영역 침범에 의한 데이터 손실



(그림 11) 초기화되지 않은 포인터

경계를 벗어난 접근오류는 인접한 메모리 블록의 데이터 손실을 일으킬 수 있다. (그림 10)에서 '블록 1'의 경계를 벗어난 'p+i' 번지에 대한 쓰기 연산은 인접한 '블록 2'의 내부 데이터 손실을 일으킨다.

C언어로 작성된 테스트 대상 프로그램에서 경계를 벗어난 접근오류는 배열인덱스를 이용한 읽기/쓰기 연산, 재귀함수에 의한 메모리 블록 읽기/쓰기 연산, 문자열 관련 함수(strcpy, strcat 등), 메모리 연산 함수(memcpy, memset 등), 명시적인 형 변환(type casting)에 의한 포인터 연산 등에 의해 발생된다.

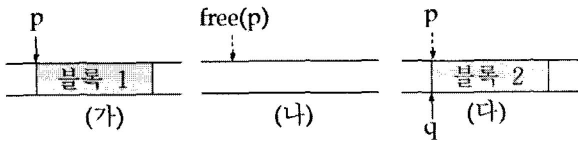
3.2 잘못된 포인터 값에 의한 접근

3.2.1 초기화되지 않은 포인터

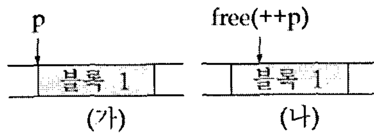
명시적인 값이 지정되지 않은 임의의 포인터 변수는 실행시간에 다른 포인터 변수에 의해 지시되는 정상적인 메모리 블록의 주소 또는 할당되지 않은 메모리 영역의 주소를 갖게 된다. (그림 11)에서 초기화되지 않은 포인터 변수 p와 q는 각각 할당되지 않은 메모리 영역과 다른 포인터 변수 r에 할당된 메모리 블록의 주소영역을 가리킨다.

(그림 11)에서 포인터 변수 p가 가리키는 메모리 영역에 대한 읽기/쓰기 연산은 의도하지 않은 프로그램의 동작을 일으킬 수 있다. 또한 포인터 변수 q가 가리키는 메모리 영역에 대한 읽기/쓰기 연산은 의도하지 않은 프로그램의 동작 또는 내부 데이터의 손실을 일으킨다.

잘못된 포인터 값에 의한 접근 오류는 메모리 블록의 해



(그림 12) 해제 후 초기화되지 않은 포인터



(그림 13) 내부포인터에 의한 메모리 블록 해제

제 이후에 값이 초기화되지 않은 포인터 변수에 의해서도 발생한다. (그림 12 (나))에서 포인터 변수 p는 '블록1'의 해제 이후에 값이 변경되지 않았다. 따라서 (그림 12 (다))에서 포인터 변수 q에 새로 할당된 '블록2'의 시작주소는 의도하지 않은 포인터 변수 p에 의해 지시되고 있다. 이 때 포인터 변수 p가 가리키는 메모리 영역에 대한 읽기/쓰기 연산은 포인터 변수 q에 할당된 메모리 영역의 내부 데이터 손실 또는 의도하지 않은 프로그램의 수행결과를 일으킨다.

3.2.2 널 포인터

널 포인터에 의한 메모리 접근 연산시도는 주로 메모리 할당 함수로부터 널 값이 반환되는 예외상황을 고려하지 않은 프로그램에서 발생한다. 이 때 널 포인터를 이용한 메모리 접근 시도는 프로그램의 강제종료 또는 의도하지 않은 프로그램의 동작을 일으킨다.

3.3 부적절한 메모리 해제

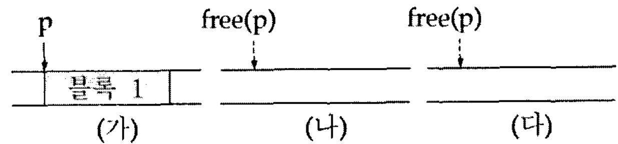
3.3.1 내부 포인터(internal pointer)에 의한 해제

내부 포인터는 포인터 산술연산에 의해 값이 변경되어 메모리 블록의 내부를 가리키는 포인터 변수이다. (그림 13 (나))에서 포인터 변수 p는 산술연산을 통해 '블록1'의 내부 주소를 가리키는 내부 포인터로 변경되었다. 이 때 변경된 포인터 변수 p의 값을 이용한 메모리 해제 시도는 프로그램의 강제종료 또는 의도하지 않은 수행결과를 일으킨다.

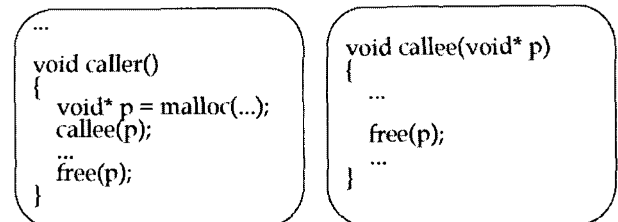
3.3.2 중복 해제

중복 해제에 의한 오류는 이미 해제된 메모리 블록을 다시 해제하려는 시도에 의해 발생한다. (그림 14 (가))에서 포인터 변수 p에 할당된 '블록1'은 (그림 14 (나))에서 해제되었다. 이 때 (그림 14 (다))에서 동일한 영역에 대한 중복 해제 시도는 프로그램의 강제종료 또는 의도하지 않은 수행결과를 일으킨다. 특히 라이브러리 함수에서 해제된 메모리 블록을 다시 해제하는 (그림 15)와 같은 코드의 경우 중복 해제에 의한 오류가 쉽게 드러나지 않을 수 있기 때문에 개발자에 의한 오류 검출 및 수정이 더욱 어려울 수 있다.

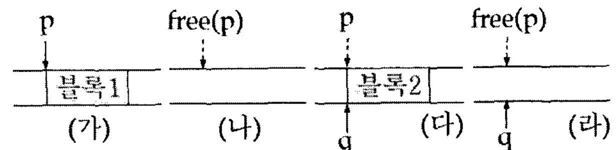
메모리 블록의 중복 해제는 다른 메모리 블록의 내부 데이터를 손상시킬 수 있다. (그림 16 (나))에서 '블록1'이 해



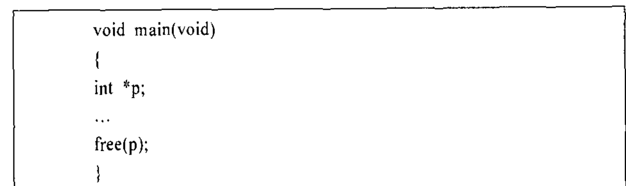
(그림 14) 중복 해제



(그림 15) 함수 간 호출에 의한 중복해제



(그림 16) 중복 해제로 인한 데이터 손실



(그림 17) 할당되지 않은 메모리 블록 해제

제된 이후 (그림 16 (다))에서 포인터 변수 q에 새로 할당된 '블록2'는 우연히 이전에 해제된 '블록1'과 동일한 주소영역에 생성되었다. 이 때, (그림 16 (라))와 같이 초기화되지 않은 포인터 변수 p에 의한 메모리 중복 해제는 포인터 변수 q에 할당된 '블록 2'의 내부데이터 손실을 일으킨다.

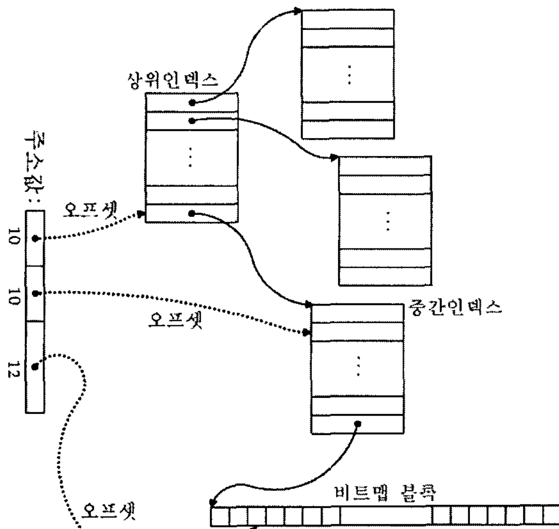
3.3.3 할당되지 않은 메모리 블록 해제

명시적인 값이 지정되지 않은 임의의 포인터 변수는 실행시간에 다른 포인터 변수에 의해 지시되는 정상적인 메모리 블록의 주소 또는 할당되지 않은 메모리 영역의 주소를 갖게 된다. (그림 17)에서 초기화되지 않은 포인터 변수 p의 주소 값에 대한 해제 시도는 다른 메모리 블록의 내부 데이터를 손실시키거나 프로그램의 강제종료를 일으킬 수 있다.

4. 비트맵 컬러링 기법

4.1 구성

본 논문에서는 테스트 대상 프로그램의 동적 메모리 접근 연산에 대해 3장에서 살펴본 다양한 동적 메모리 오류를 검



(그림 18) 인덱스-비트맵의 구성

〈표 1〉 비트맵 블록의 공간 오버헤드

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

사하는 비트맵 컬러링 기법을 제안한다.

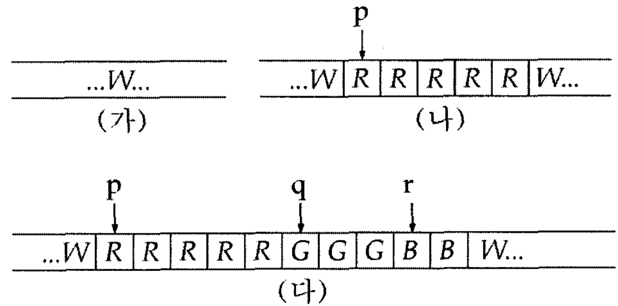
비트맵 컬러링 기법은 2.1절에서 살펴본 기존의 소스코드 분석에 기반한 방법에 비해 테스트 대상 프로그램에 포인터 변수의 앨리어싱이나 함수 간 주소 값 전달이 많이 일어나는 경우 동적 메모리 블록의 할당정보를 유지하기 위한 공간 오버헤드가 적다. 또한 포인터 산술연산에 대해 메타데이터의 갱신을 수행하지 않을 뿐만 아니라 오류검사 과정에서는 비트연산(bitwise operation)을 통해 블록의 할당여부와 오류 발생여부를 빠르게 알 수 있기 때문에 테스트 대상 프로그램에 가해지는 성능 오버헤드가 매우 적다.

테스트 대상 프로그램이 할당 받은 동적 메모리 블록은 각 메모리 블록의 주소범위에 대응되는 연속된 비트맵 필드에 기록된다. (그림 18)은 동적 메모리 블록의 할당 정보를 기록하기 위한 인덱스-비트맵(indexed bitmap)의 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 시스템에서 각 페이지에 대응되는 비트맵 블록의 크기는 최대 1KB이다. 비트맵 블록의 크기는 메모리 할당함수로부터 반환되는 동적 메모리 블록의 최소 크기에 따라 달라진다.

〈표 2〉 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 19) 비트맵 필드

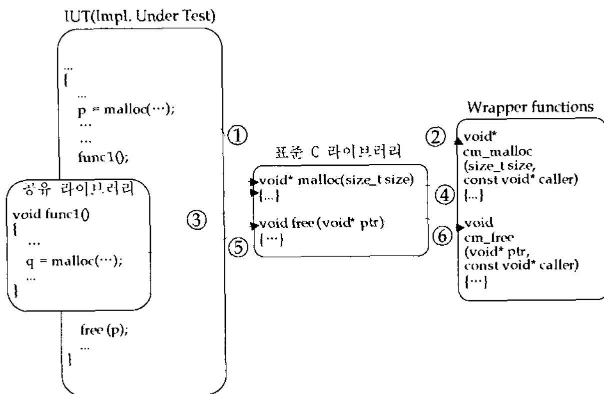
<표 1>은 동적 메모리 블록의 최소 크기에 따라 비트맵 블록에 소요되는 공간 오버헤드를 보이고 있다.

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 값은 비트맵 블록에서 해당 메모리 블록의 할당 정보를 기록할 비트맵 필드의 오프셋으로 사용된다. 생성된 메모리 블록의 주소에 대응되는 비트맵 필드를 검색한 후 해당 비트맵 필드로부터 메모리 블록의 크기에 해당되는 연속된 비트맵 필드들은 인접한 비트맵 필드의 값과 구분되도록 <표 2>의 2비트 이진코드 값으로 컬러링된다.

동적 메모리 블록을 하나도 할당 받지 않은 테스트 대상 프로그램의 비트맵 블록은 (그림 19 (가))와 같이 모두 W값으로 초기화되어 있다. 이 때 테스트 대상 프로그램이 요청한 블록크기 5인 메모리 블록이 생성되면 (그림 19(나))와 같이 할당된 영역의 주소범위에 대응되는 비트맵 블록은 인접 블록과 구분되도록 R값으로 기록된다. (그림 19 (다))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.

비트맵 필드에 컬러링 값은 메모리 할당 및 해제함수에 대한 래퍼(wrapper)함수에 의해 기록된다. (그림 20)은 표준 C 라이브러리의 메모리 할당 및 해제 함수와 래퍼함수 간의 호출관계를 나타낸다.

래퍼함수는 테스트 대상 프로그램 및 링크된 라이브러리에서 malloc/free함수 호출 시 (그림 20)과 같이 콜백(callback) 형태로 호출되기 때문에 소스코드가 제공되지 않는 공유 라이브러리 등의 함수 내부에서 생성된 동적 메모리 블록에 대해 할당 정보를 모두 유지할 수 있다. 따라서 비트맵 컬러링 기법은 본 논문의 2.1.1절과 2.1.3절에서 각각 살펴본



(그림 20) 메모리 할당/해제 요청에 대한 래퍼함수 구성

<표 3> 포인터 변수 p의 상태정의

상태	의미
$P_{uninitialized}$	초기화되지 않은 상태
$P_{initialized}$	명시적으로 값이 지정된 상태
$P_{allocated}$	메모리 할당함수의 반환 값으로 설정된 상태
$P_{modified}$	포인터 산술연산에 의해 값이 변경된 상태

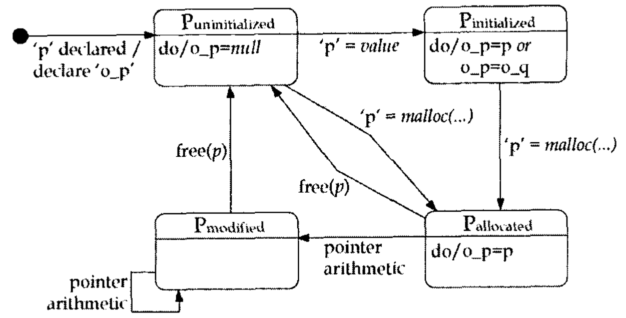
Safe C Compiler와 CCured에서 나타나는 공유 라이브러리와의 호환성 문제를 갖지 않는다는 점을 알 수 있다.

또한, 메모리 오류검사 도구는 래퍼 함수를 통해 인덱스 비트맵을 유지하기 때문에 시스템이 제공하는 메모리 할당/해제 함수의 내부구현에 독립적이다. 따라서, 메모리 주소의 길이와 동적 메모리 영역의 주소범위를 변경함으로써 주소지정방식이 서로 다른 다양한 플랫폼에 쉽게 적용시킬 수 있기 때문에 각 플랫폼에 따라 별도로 메모리 할당/해제 함수를 구현해야 하는 기존의 연구결과에 비해 이식성이 뛰어나다는 점을 알 수 있다.

4.2 소스코드 분석 및 확장

비트맵 컬러링 기법에서는 테스트 대상 프로그램의 소스코드에 대한 분석 및 확장을 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. 소스코드 분석 단계에서는 포인터 변수 o_p의 선언 및 정의 구문을 삽입하기 위해 포인터 변수 p의 상태를 추적한다. 이 때 각 포인터 변수 p의 상태는 <표 3>과 같이 정의한다.

소스코드 확장단계에서는 포인터 변수 p의 상태변화에 따라 (그림 21)에 나타난 상태 다이어그램의 액션(action) 및 액티비티(activity)부분에 해당하는 구문을 테스트 대상 프로



(그림 21) 'o_p' 선언 및 정의 규칙

(가) 원본코드	(나) 변환된 코드

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;

(그림 22) 'o_p'의 선언 및 정의

(가) 원본코드	(나) 변환된 코드

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;

(그림 23) 앨리어싱에 따른 코드 변환

그램의 소스코드에 삽입한다. (그림 21)의 구문 삽입 규칙에 따라 테스트 대상 프로그램에 코드삽입을 수행한 결과는 (그림 22)와 같다.

(그림 23)은 포인터 변수의 앨리어싱이 일어난 경우의 코드변환 방법을 보이고 있다. (그림 23 (나))에서 포인터 변수 q의 값이 p에 할당될 때 포인터 변수 p의 상태는 $P_{initialized}$ 상태로 전이되면서 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 소스코드 확장단계에서 삽입되는 또다른 구문은 메모리 접근 연산의 직전에 삽입되는 오류검사 함수의 호출구문이다. 실행시간에 수행되는 오류검사 함

〈표 4〉 오류검사 함수목록

함수명	인자(argument) 목록	
	인자값	의미
_check_memory	o_p	포인터 변수 p에 할당된 메모리 블록의 시작주소
	p	접근 연산이 일어나는 메모리 주소
_check_free	o_p	포인터 변수 p에 할당된 메모리 블록의 시작주소
	p	해제 연산이 일어나는 메모리 주소

〈표 5〉 __check__memory( ) 함수 호출구문의 삽입예

메모리 접근연산	오류검사함수 호출
array[i]	_check_memory(o_array, array+i);
*(++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);
	}

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

```
void *p = malloc(...);
void *o_p = p;
_check_free(o_p, p);
free(p);
```

(가) 원본코드 (나) 확장된 코드

(그림 24) __check__free( ) 함수 호출구문의 삽입예

수는 <표 4>와 같다.

<표 4>의 _check_memory()함수는 포인터 변수를 이용한 접근 연산의 직전에 삽입되며 해당 연산의 유효성을 검사한다. 테스트 대상 프로그램의 소스코드에 대한 분석 및 확장 과정에서 <표 5>와 같이 _check_memory()함수 호출구문을 메모리 접근 연산의 직전에 삽입한다.

<표 4>의 _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 24).

함수 간 호출에서 포인터 변수의 값이 전달되는 경우 o_p의 선언 및 유지 구문과 오류검사함수 호출구문은 (그림 25)와 같이 삽입된다. (그림 25)에서 함수 foo()는 메모리 블록의 주소 값을 인자 값으로 전달받는다. 이 때 오류검사 함수의 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.

4.3 오류검사

본 논문에서 제안하는 동적 메모리 접근오류 검출기법은

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

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

```
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);
...
}
```

(가) 원본코드 (나) 변환된 코드

(그림 25) 함수간 포인터변수 전달

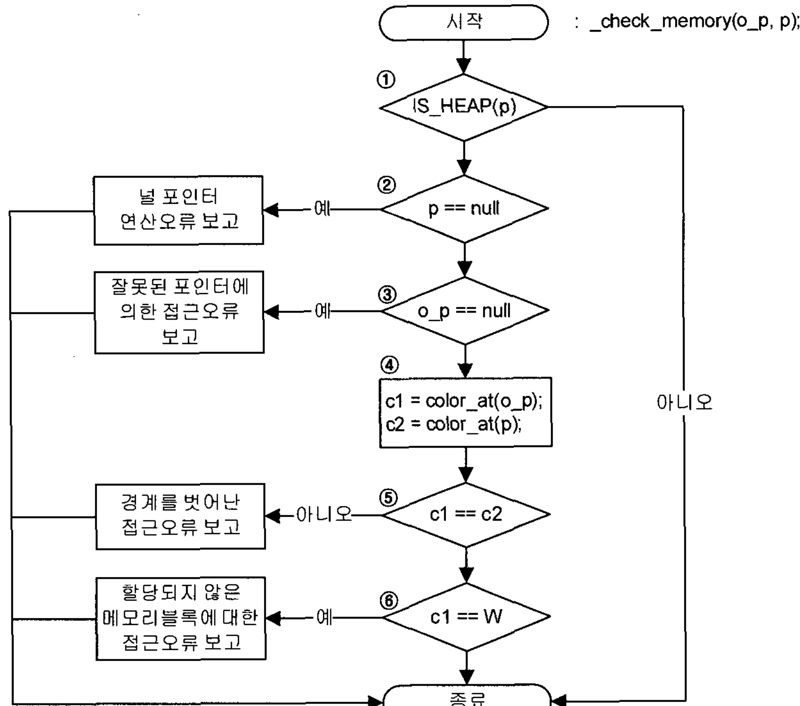
본 논문의 3장에 나타난 다양한 형태의 동적 메모리 접근오류에 대해 다음과 같이 오류검사를 수행한다. 소스코드 확장단계에서 삽입된 오류검사 함수는 테스트 대상프로그램의 수행시간에 메모리 접근 및 해제연산에 대한 유효성을 검사한다. (그림 26)은 _check_memory() 함수에 의한 메모리 접근오류의 검사 단계를 나타낸다.

동적 메모리에 대한 접근오류 검사는 (그림 26)에 나타난 바와 같이 다음의 여섯 단계로 수행된다.

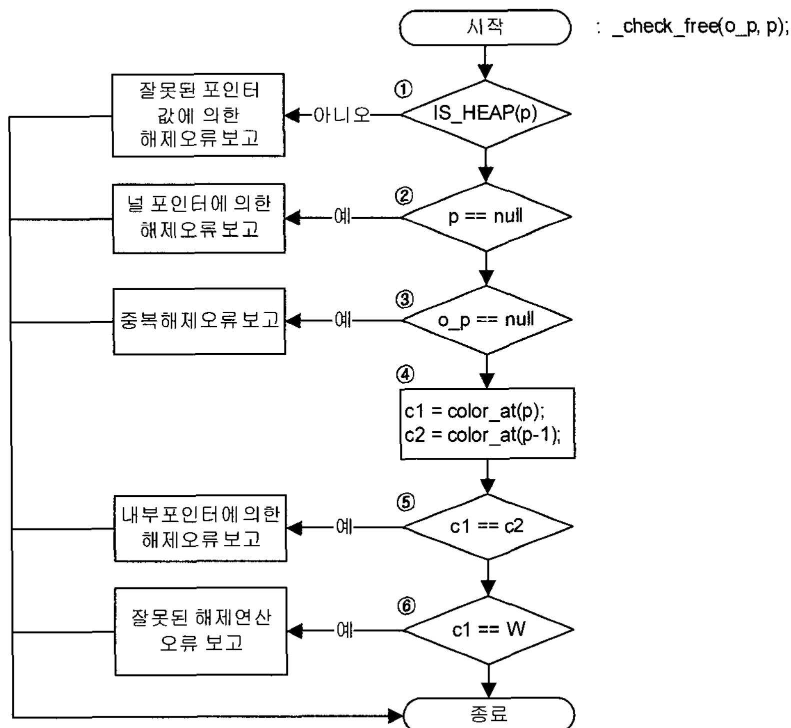
① 두 번째 인자 값으로 전달된 접근연산의 대상 메모리 주소 값이 힙 영역의 주소범위에 포함되는지 검사하고, 그렇지 않은 경우 오류검사를 중단한다.
② 접근연산의 대상 메모리 주소 값이 널인 경우 널 포인터에 의한 연산오류를 보고하고 오류검사를 중단한다.
③ o_p값이 널인 경우 잘못된 포인터 값에 의한 접근오류를 보고하고 오류검사를 중단한다.
④ 인덱스-비트맵으로부터 o_p와 p의 주소 값에 대응되는 컬러링 값을 검색하고 c1과 c2에 각각 저장한다.
⑤ 두 개의 컬러링 값인 c1과 c2가 서로 다른 경우 경계를 벗어난 접근오류를 보고하고 오류검사를 중단한다.
⑥ c1과 c2가 동일하지만 컬러링 값이 W인 경우 할당되지 않은 메모리 블록에 대한 접근오류를 보고하고 오류검사를 중단한다.

(그림 27)은 _check_free()함수에 의한 메모리 해제오류의 검사 단계를 나타낸다. 동적 메모리에 대한 해제오류 검사는 (그림 27)에 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 해제



(그림 26) 접근오류검사 순서도



(그림 27) 해제오류검사 순서도

오류를 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 p값

이 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.

4.3.1 경계를 벗어난 접근 오류

(그림 28 (가))의 C언어 프로그램은 경계를 벗어난 동적 메모리 접근오류를 일으킨다. 소스코드 확장을 통해 변환된 (그림 28 (나))의 코드에서 배열 p의 i번째 블록에 대한 쓰기 연산은 _check_memory() 함수에 의해 메모리 접근 오류로 보고된다. _check_memory() 함수는 (그림 28 (다))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+*i*번지의 메모리 블록에 대한 접근이 경계를 벗어난 접근 오류라고 판단한다.

(그림 29 (가))의 C언어 프로그램은 구조체 형의 포인터 변수 r에 의한 메모리 접근 연산에서 인접한 메모리 블록 q가 가리키는 메모리 블록의 데이터를 손상시킬 수 있다. (그림 29 (나))의 변환된 코드는 포인터 변수 r에 의한 메모리 접근 연산 이전에 (그림 29 (다))의 비트맵 컬러링 정보로부터 메모리 오류를 검사한다. 이때 _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.

4.3.2 잘못된 포인터 값에 의한 메모리 접근 오류

(그림 30 (가))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 데이터 손실을 일으킨다.

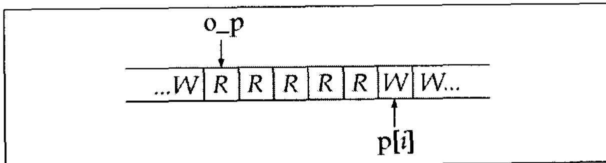
(그림 30 (가))의 소스코드는 확장단계를 통해 (그림 31 (가))와 같이 확장된다. 확장된 코드에서는 buf1이 가리키는 메모리 블록에 문자열을 복사하기 전에 메모리 오류를 검사

```
int i
int *p =
(int*)malloc(sizeof(int)*5);
for (i=0; i<5; i++);
p[i] = 1;
```

(가) 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;
```

(나) 변환된 코드



(다) 비트맵 블록

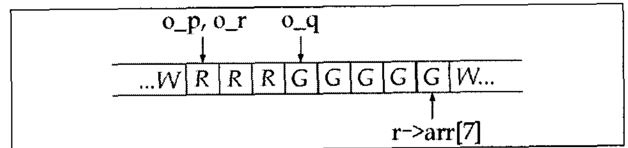
(그림 28) 경계를 벗어난 접근 오류 검사

```
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;
...
}
```

(가) 원본코드

```
typedef struct _complex_t { int arr[8]; } complex_t;
...
int *p, *q;
int *o_p=null, *o_q=null;
...
{
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;
...
}
```

(나) 변환된 코드



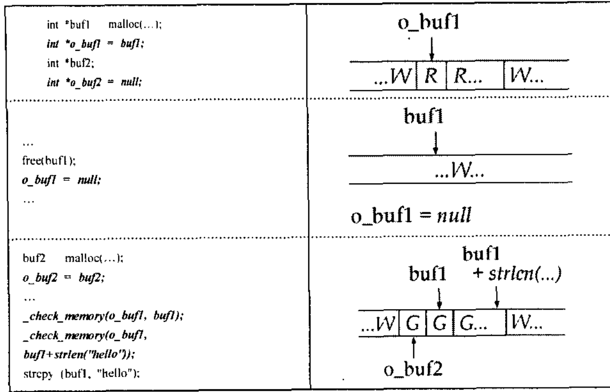
(다) 비트맵 블록

(그림 29) 형 변환에 의한 메모리 오손 오류 검출

(가) 원본코드	(나) 동적 메모리 블록
int *buf1 = malloc(...); int *buf2;	buf1 a
... free(buf1); ...	buf1
buf2 = malloc(...); ... strcpy (buf1, "hello");	buf1 b buf2

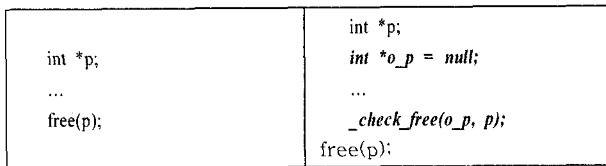
(가) 원본코드 (나) 동적 메모리 블록

(그림 30) 잘못된 포인터 값에 의한 메모리 접근



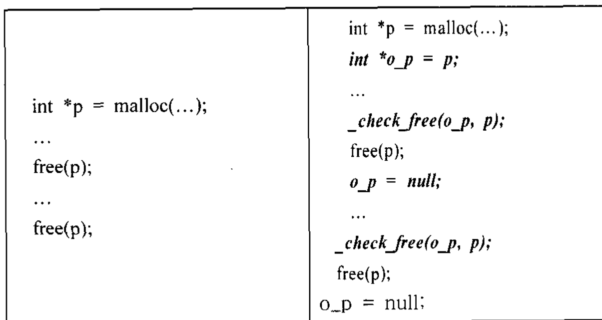
(가) 확장된 코드 (나) 비트맵 블록

(그림 31) 잘못된 포인터 값에 의한 접근 검사



(가) 원본코드 (나) 확장된 코드

(그림 32) 잘못된 해제연산 시도



(가) 원본코드 (나) 확장된 코드

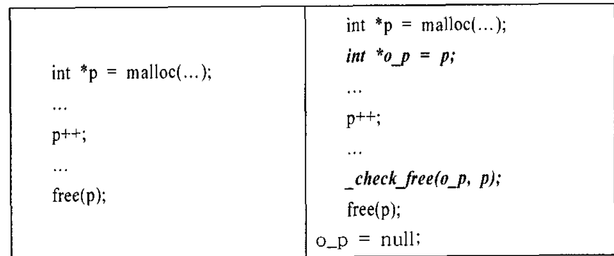
(그림 33) 동적 메모리 블록의 중복해제 시도

한다. 이 때, 두 번째 인자 값으로 전달된 buf1의 주소 값은 유효한 메모리 블록을 가리키고 있다. 그러나 오류검사 함수의 첫 번째 인자 값으로 전달된 o_buf1가 널 값을 갖고 있기 때문에 잘못된 포인터 변수에 의한 메모리 오류를 보고할 수 있다.

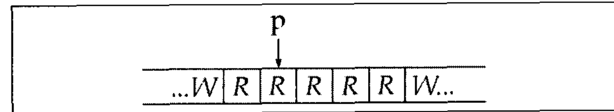
위에서 살펴본 메모리 오류 이외에 할당되지 않은 메모리 블록에 대한 접근 연산 및 널 포인터에 의한 메모리 접근 연산을 검출할 수 있다. 할당되지 않은 메모리 블록에 대한 접근 연산은 연산의 대상 주소에 대응되는 비트맵 필드의 값이 W인 경우 해당 오류를 보고한다. 또한 연산의 대상 주소 값이 널인 경우 널 포인터에 의한 메모리 접근 연산 오류를 보고한다.

4.3.3 부적절한 동적 메모리 해제 시도

(그림 32 (가))와 (그림 33 (가))의 C언어 프로그램은 각각



(가) 원본코드 (나) 확장된 코드



(다) 비트맵 블록

(그림 34) 내부 포인터에 의한 해제

<표 6> 메모리 할당정보 유지를 위한 시간복잡도

연산종류	Safe C	Purify/Valgrind	본 논문
삽입	O(1)	O(1)	O(1)
갱신	O(1)	O(1)	없음
삭제	O(1)	O(1)	O(1)

메모리 블록이 할당되지 않은 포인터 변수 값에 의한 메모리 해제 연산 및 동적 메모리 블록의 중복해제를 시도하고 있다. 테스트 대상 프로그램으로부터 확장된 (그림 32 (나))의 _check_free()함수와 (그림 33 (나))의 두 번째_check_free() 함수는 첫 번째 인자로 전달된 값이 널 값이므로 모두 잘못된 포인터 값에 의한 메모리 해제시도 오류를 보고한다.

(그림 34 (가))의 C언어 프로그램은 동적 메모리 블록의 내부 주소를 통해 메모리 블록의 해제를 시도하고 있다. (그림 34 (나))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.

5. 기존 연구와의 비교

본 연구결과는 테스트 대상 프로그램에 가해지는 오버헤드 및 오류검출성능을 바탕으로 기존의 연구결과와 비교할 수 있다. 기존 연구와의 비교를 위해 2.1.2절에서 살펴본 확장된 Safe C Compiler(이하, Safe C)와 2.2절에서 살펴본 Rational Purify 및 GNU Valgrind가 사용되었다. 2.1.3절의 CCured는 테스트 대상 프로그램의 변환과정과 수행시간 동작방법이 본 논문에서 제안한 기법과 전혀 다르기 때문에 직접적으로 비교할 수 없다. 따라서 CCured는 오류검출성능에 대해서만 비교하도록 한다.

<표 6>은 테스트 대상 프로그램의 동적 메모리 할당정보를 유지하기 위한 시간복잡도 비교이다. 메모리 할당정보의 삽입/삭제연산은 <표 6>에 나타난 바와 같이 세 가지 방법 모두 상수시간 내에 수행된다. 한편, Safe C와 Purify/Valgrind는 각각 포인터 산술연산과 메모리 접근연산에 대해 할당정보의

〈표 7〉 코드삽입을 위한 시간복잡도

(m/f/a : malloc()호출/free()호출/포인터 산술연산)

연산종류	Safe C	Purify/Valgrind	본 논문
코드삽입	O(m+f+a)	O(m+f)	O(m+f)

〈표 8〉 메모리 할당정보 유지를 위한 공간오버헤드

	공간오버헤드
Safe C	포인터변수 당 최소 12바이트
Purify/Valgrind	전체 힙 사용량의 25%
비트맵 컬러링	전체 힙 사용량의 25% 이하

〈표 9〉 오류검출성능 비교(O:모두 찾을 수 있음, △:일부 못 찾음, X:전혀 찾지 못함)

	Safe C	Purify/ Valgrind	CCured	본 논문
경계를 벗어난 접근	△	△	△	○
초기화되지 않은 포인터에 의한 접근	△	△	△	○
널 포인터에 의한 접근	○	○	○	○
내부 포인터에 의한 해제	○	○	○	○
중복해제	△	△	△	△
할당되지 않은 메모리 블록 해제	○	○	X	○

갱신연산을 상수시간에 수행한다. 반면, 본 논문에서 제안한 기법은 메모리 할당정보의 갱신연산을 수행하지 않기 때문에 다른 방법들에 비해 메모리 할당정보 유지를 위한 성능 오버헤드가 적다.

<표 7>은 코드삽입연산에 대한 시간복잡도 비교이다. Safe C는 테스트 대상 프로그램에 대한 전처리 과정에서 소스코드에 나타나는 메모리 할당/해제 함수호출 구문과 포인터 산술연산 구문에 대해 추가적인 코드를 삽입한다. 반면, Purify/Valgrind와 본 논문에서 제안한 기법은 포인터 산술연산 구문에 대한 추가적인 코드를 삽입하지 않기 때문에 Safe C에 비해 코드삽입을 위한 오버헤드가 적다.

<표 8>은 테스트 대상 프로그램의 메모리 할당정보를 저장하기 위한 공간오버헤드 비교이다. Safe C는 테스트 대상 프로그램에 나타나는 모든 포인터 변수에 대해 메타데이터를 최소 12바이트 크기의 레코드로 유지해야 한다. 한편, Purify/Valgrind와 본 논문에서 제안한 기법은 테스트 대상 프로그램이 사용 중인 전체 힙 메모리 크기의 최대 25%를 메모리 할당정보를 유지하기 위해 사용한다.

본 논문에서 제안한 기법은 <표 9>와 같이 기존 연구결과에 비해 오류검출성능이 향상되었다.

본 논문에서 제안한 메모리 접근오류 검출기법은 구현을 통해 GNU 오픈소스 프로그램을 대상으로 본 논문의 2.2절에서 살펴본 소스코드 분석기법에 기반한 오류검출기법과 공간 및 성능 오버헤드를 비교하였으며 실험결과는 <표 10>과 같다. 실험결과표에서 JK&RM은 R. Jones와 P. Kelly가 제안한 초기의 Safe C Compiler에 O. Ruwase와 M. Lam이 제안한 기법이 적용된 향상된 메모리 접근오류 검출도구이다[6, 7].

<표 10>의 실험결과에서 JK&RM은 동적 메모리 사용량이 테스트 대상 프로그램에 따라 가변적인데 반해 본 논문의 수행결과는 모든 프로그램에 대해 264K로 동일한 것을 알 수 있다. 이는 JK&RM이 테스트 대상 프로그램의 수행시간에 생성되는 각 메모리 블록에 대한 메타데이터를 유지하는 반면, 본 논문에서는 4.1절에 언급된 바와 같이 테스트 대상 프로그램이 사용하는 페이지 단위 주소영역에 대한 메타데이터를 유지하기 때문이다.

바이너리 코드분석기법에 기반한 메모리 오류검사 도구인 GNU Valgrind는 오류검사환경에서 테스트 대상 프로그램이 단독으로 수행되지 않기 때문에 본 연구결과와 동적 메모리 사용량에 대한 비교를 수행할 수 없다. 따라서 본 논문에서는 수행시간 비교를 통해 테스트 대상 프로그램에 가해지는 성능 오버헤드만을 GNU Valgrind와 비교했으며 결과는 <표 11>과 같다.

〈표 10〉 JK&RM과의 공간 및 성능오버헤드 비교실험 결과

IUT		LOC	JK&RM		본 논문	
			동적메모리 사용량	수행 시간	동적메모리 사용량	수행 시간
grep-2.5.1	패턴검색 유틸리티	24,734L	342.7K	0.19s	264K	0.12s
tar-1.16	Archiving 유틸리티	73,446L	405.1K	12.19s	264K	7.93s
gnupg-1.4.7	암호키 생성도구	117,781L	706.6K	74.64s	264K	29.33s

〈표 11〉 GNU Valgrind와의 성능오버헤드 비교실험 결과

IUT		LOC	GNU Valgrind	본 논문
			수행시간	수행시간
grep-2.5.1	패턴검색 유틸리티	24,734L	0.66s	0.12s
tar-1.16	Archiving 유틸리티	73,446L	15.18s	7.93s
gnupg-1.4.7	암호키 생성도구	117,781L	299.8s	29.33s

6. 결론 및 향후 연구

본 논문에서는 C언어로 작성된 테스트 대상 프로그램으로부터 동적 메모리 접근오류를 검출하기 위한 방법으로 소스코드 분석기법을 기반으로 한 비트맵 컬러링 기법을 제안하였으며 실험을 통해 기존연구와의 비교분석을 수행하였다.

본 논문의 연구결과는 소스코드 분석에 기반하고 있기 때문에 바이너리 코드 분석에 기반한 상용화 도구인 Rational Purify나 GNU Valgrind에서 검출할 수 없는 "형 변환에 의한 메모리 오손"과 "의도하지 않은 포인터 변수에 의한 유효 메모리 접근" 오류를 검출할 수 있으며 상대적으로 오류검출 성능이 뛰어나다. 또한 5장에서 살펴본 바와 같이 동적 메모리 할당정보를 유지하기 위한 공간 오버헤드가 Rational Purify나 GNU Valgrind에 비해 동일하거나 더 적다. 또한, 본 논문의 연구결과는 포인터 산술연산에 대해 추가적인 연산을 수행하지 않기 때문에 기존의 소스코드 분석에 기반한 방법에 비해 테스트 대상 프로그램에 가해지는 성능오버헤드가 적다. 비록 소스코드 분석에 기반한 최근의 연구결과인 CCured는 포인터 산술연산에 따른 추가적인 연산을 수행하지 않지만 2.1.3절에서 살펴본 바와 같이 소스코드가 제공되지 않는 공유 라이브러리와의 호환성 및 메모리 해제연산 이후에 발생하는 메모리 접근에 대한 오류검출 성능이 떨어지는 문제를 해결하지 못하고 있다. 따라서 비트맵 컬러링 기법은 테스트 대상 프로그램에 가해지는 공간 오버헤드를 최소화하면서 오류검출성능을 최대화하고 있다는 점을 알 수 있다.

이외에도 비트맵 컬러링 기법은 메모리 할당함수의 내부 구현에 독립적이기 때문에 주소지정방식이나 주소공간의 구성이 상이한 다양한 플랫폼에 쉽게 적용할 수 있다. 이러한 특징은 별도의 메모리 할당함수를 구현해야 하는 기존의 방법에 비해 비트맵 컬러링 기법이 갖는 또다른 장점이다.

향후 본 논문의 연구결과를 바탕으로 테스트 대상 프로그램에 나타나는 간접포인터에 의한 메모리 접근, 구조체 및 공용체 필드 내부에서 발생하는 경계를 벗어난 접근, 스택 영역에서의 메모리 접근 등으로부터 발생하는 메모리 오류를 검출할 수 있도록 확장할 것이다.

참 고 문 헌

[1] 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.

[2] 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.

[3] 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, page 2-9, June 1991

[4] William R. Bush, Jonathan D. Pincus and David J. Sielaff. "A static analyzer for finding dynamic programming errors," SP&E, 30(7):775-802, June 2000.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

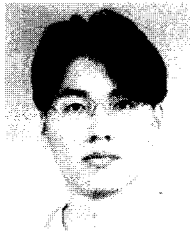
조 대 완

e-mail : oopmania@cnu.ac.kr

2006년 충남대학교 컴퓨터공학과(학사)

2006년~현 재 충남대학교 컴퓨터공학과 석사과정

관심분야 : 소프트웨어 테스팅, 소프트웨어 품질관리, 시험 자동화



오 승 욱

e-mail : suoh@suresofttech.com

1996년 한국과학기술원 전산학과(학사)

1998년 한국과학기술원 전산학과 (공학석사)

1998년~현 재 한국과학기술원 전산학과 박사과정

2001년~현 재 슈어소프트테크㈜ 연구소

관심분야 : 소프트웨어 테스팅, 요구사항 검증

김 현 수

e-mail: hskim401@cnu.ac.kr

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~2007년 9월 충남대학교 전기정보통신공학부 부교수

2007년 8월~현 재 Purdue University 방문교수

2007년 10월~현 재 충남대학교 전기정보통신공학부 교수

관심분야 : 소프트웨어 테스팅, 소프트웨어 재공학, 컴포넌트 마이닝, 컴포넌트 테스팅, SOA