C언어 기반 프로그램의 메모리 누수 검출기법

배기곤[O] 이숙희 권용래
한국과학기술원 전자전산학과
{ggbae[O], shlee, kwon}@salmosa.kaist.ac.kr

Memory Leak Detection in C

Gigon Bae[0], Sukhee Lee, Yongrae Kwon
Dept. of Electronic Engineering and Computer Science, KAIST

요 약

더 이상 사용되지 않는 메모리가 계속해서 유지되는 것을 메모리 누수라고 한다. 메모리 누수가 발생하면 메모리 낭비가 누적되기 때문에 시스템의 성능이 저하되고 궁극적으로 시스템 크래시(crash)가 발생할 수 있다. 본 논문에서는 이러한 메모리 누수를 검출하기 위하여 참조 계수 기법을 이용한다. 참조계수 기법을 이용하면 메모리 누수의 발생 여부뿐만 아니라 메모리 누수 발생시점에 대한 정보까지 제공할 수 있어 디버깅이 용이해진다. 그리고 본 논문에서 제안한 기법을 구현한 도구를 이용하여 사례연구를 수행한다. 사례 연구 분석을 통하여 본 연구에서 제안한 기법이 정확하게 메모리 누수를 검출하고 디버깅에 유용한 정보를 제공할 수 있다는 것을 보인다.

1. 연구 배경

Garbage 수집기가 더 이상 사용되지 않는 메모리를 자동으로 회수하는 Java와 달리, C에서는 개발자가 직접 메모리를 관리하기 때문에 메모리 에러가 자주 발생한다. 그 중, 메모리 누수(memory leak)란 할당된 메모리가 더 이상 사용되지 않지만 계속 유지되는 것으로, 의도하지 않은 메모리 낭비가 발생한다. 메모리 누수는 누적적이므로 시스템 수행속도를 저하하고 결국 시스템 고장을 초래할 수 있다. 특히 임베디드(embedded) 시스템과 같이 시스템의 자원이 한정되어 있고 계속적으로 돌아가는 프로그램의 경우 메모리 누수가 미치는 영향은 더욱 치명적이다.

지금까지 메모리 누수를 검출하기 위한 많은 방법들이 제안되었다. 하지만 기존의 도구들은 시스템의 종료 후 메모리 누수 발생 여부에 대해서만 보고할 뿐 메모리 누수 발생 시점이나 발생 원인 등 메모리 누수를 제거하기 위한 디버깅 정보는 제공하지 않는다. 간단한 프로그램이라도 메모리 누수 발생원인을 찾기 어렵고 시스템이 복잡해질수록 메모리 누수를 제거하기 위해 많은 시간과 노력이 요구된다. 따라서 기존의 도구들로 메모리 누수를 제거하기는 어렵다.

본 논문에서는 이러한 문제를 해결하기 위하여 참조 계수 기법을 이용한다. 참조 계수 기법은 원래 Java에서 garbage를 수거하기 위해 사용하는 기법으로서 메모리 누수 검출에 사용되면 프로그램 수행 중 메모리 누수가 발생한 순간에 누수의 발생 시점뿐만 아니라 디버깅에 유용한 정보를 제공할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 C언어로 작성된 프로그램의 메모리 누수를 검출해주는 기존의 도구들을 살펴본다. 3장에서는 본 연구에서 메모리 누수 검출을 위해 이용하는 참조 계수 기법에 대해서 설명한다. 4장에서는 메모리 누수 검출 기법을 제안하고 이어서 5장에서는 본 논문에서 제안한 기법을 지원하는 도구 구현을 살펴본다. 6장에서 사례연구를 통해 제안된 기법의 효용성을 알아보고 마지막으로 7장에서 결론을 맺는다.

2. 관련 연구

C언어에서 메모리 누수를 검출하기 위한 방법은 크게 정적 분석 기법과 동적 분석 기법으로 나눌 수 있다. 정적 분석 기법은 소스코드를 분석하여 메모리 누수를 검출하는 방법으로 프로그램 실행에 따른 오버헤드가 발생하지 않는다. 하지만 실행 정보의 부족으로 인해 모든 누수를 찾지 못하고 거짓 경보(false alarm)의 발생이 많다[1]. 동적 분석 기법을 이용한 메모리 누수 검출 도구 중에서 주로 사용되는 오픈 소스 도구들을 소개하면 아래와 같다.

- MemCheckDeluxe

  MemCheckDeluxe[2]는 실행중인 애플리케이션의 메모리 할당 상태를 사용자에게 보여주는 API를 제공한다. 메모리 누수를 검출하려면, 개발자가 애플리케이션 소스 코드에 API를 삽입해야 한다. 그리고 API가 보여주는 메모리 사용 현황 정보를 이용하여 개발자가 직접 메모리 누수 여부와 누수 위치를 판단해야 한다

- MemWatch

  MemWatch[3]는 라이브러리 형태로, 메모리 누수를 검출할 프로그램과 함께 컴파일 후 사용할 수 있다. 대상 프로그램이 실행되는 동안 메모리관련 함수가 호출되면 MemWatch가 이를 대신 관리하고 프로그램이 종료 되면 해지되지 않은 메모리 블록의 정보를 기록한다. MemCheckDeluxe에 비해 사용방법은 간단하지만 여전히 해지되지 않은 메모리에 대해서 메모리 할당 시점에 대한 소스 코드상의 위치 정보만을 제공해 주기 때문에 누수의 근본 원인을 알기가 어렵다.

- Valgrind

  Valgrind[4]는 오픈 소스로 구현된 대표적인 메모리 오류

검출 도구로서 가상 머신(virtual machine)을 사용하여 이진 코드를 에뮬레이션(emulation)함으로써 각종 메모리 오류를 검출한다. 소스 코드가 없어도 메모리 누수 검출이 가능하며 누수가 일어난 메모리 블록에 대해서 소스 코드 상의 할당 위치, 할당 시점의 호출 스택(call stack) 등의 정보를 보여준다. 그러나 메모리 누수를 제거하기에는 제공되는 정보가 여전히 부족하며 Valgrind를 실행할 수 있는 플랫폼과 이진 코드가 제한적이다.

- DIOTA

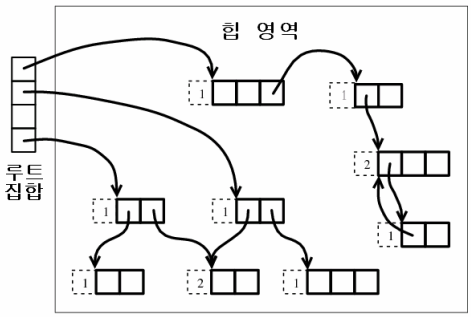
DIOTA[5]는 이진 코드를 대상으로 하여 프로그램 실행 중에 코드 삽입과 garbage 컬렉션 기법을 통해 메모리 누수를 검출하는 도구다. 앞에서 설명한 기존의 다른 도구들에 비해서 디버깅에 유용한 정보를 가장 많이 제공한다. 하지만 실행 오버헤드가 DIOTA를 사용 하지 않았을 경우에 비해 200~300배로 매우 크며 리눅스/X86 시스템 만을 지원하기 때문에 플랫폼에 독립적이지 못하다.

기존에 존재하는 대부분의 도구들은 공통적으로 다음과 같은 문제점을 가진다. 첫째, 프로그램의 수행이 끝난 후에 메모리 누수를 검출하기 때문에, 서버나 임베디드 시스템의 애플리케이션처럼 지속적으로 실행되는 경우 적용하기가 어렵다. 둘째, 대부분의 도구들은 누수된 메모리 블락의 시작 주소, 크기, 그리고 소스 코드 상의 할당 위치 등과 같은 정보만 제공한다. 하지만 검출된 메모리 누수를 제거하기 위해서는 누수된 메모리에 대한 정보뿐만 아니라 메모리 누수가 발생한 지점에 대한 정보가 필요하다.

3. 배경 지식

3.1 참조 계수(Reference counting) 기법

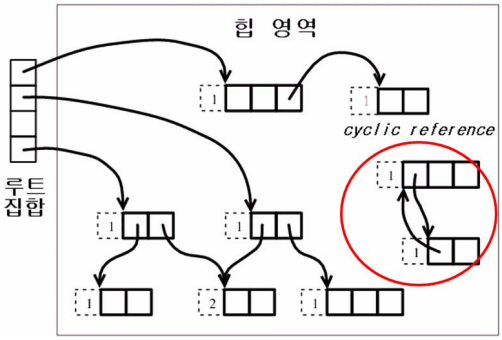
참조 계수(reference counting) 기법은 garbage 컬렉션(garbage collection, 이하 GC)에 사용되는 기법 중 하나로 각 객체마다 그 객체를 가리키는 참조수(reference count)를 저장한다. 어떤 포인터가 객체를 가리킬 때 마다 그 객체의 참조수는 증가하고 객체를 가리키고 있던 포인터가 사라지게 되면 참조수가 줄어든다(그림 2).



<그림 2. 참조 계수 기법[6]>

따라서 객체의 참조수가 0이 되면 그 객체를 가리키는 포인터가 없기 때문에 더 이상 사용되지 않는 객체로 판단하고 해당하는 메모리를 회수한다. 메모리를 회수할 때에는 회수되는 객체 내에 다른 객체를 가리키는 포인터가 있는지 살펴보고 만약 그런 포인터가 존재 한다면 내부 포인터가 가리키는 객체에 대한 참조수를 감소시킴으로써 정확한 참조수가 유지되도록 한다.

하지만 일반적인 참조 계수 기법만으로는 접근 불가능한(unreachable) 모든 객체를 회수할 수 없다. 할당된 객체들이 순환 구조(circular structures)를 가지고 있는 경우에는 메모리 누수가 일어나도 참조 계수 기법으로 회수하지 못할 수 있다[7].



<그림 3. 참조 계수 기법의 문제점[6]>

[그림 3]에서 타원 안의 객체들이 사이클을 형성하고 있다. 이 객체들은 루트 집합으로부터 도달할 수 있는 경로가 없지만 두 객체가 서로를 가리키고 있으므로 참조수가 1이 되어 회수되지 않는다. 이런 객체들을 사이클 메모리 누수라고 한다. 일반적으로 참조 계수 기법만으로는 사이클 메모리 누수를 검출할 수 없기 때문에 다른 garbage 컬렉션 기법들과 함께 사용된다. 본 연구에서는 참조 계수 기법을 보완하기 위하여 Mark-Sweep 기법을 사용한다.

3.2 Mark-Sweep 기법

Mark-Sweep 기법[8] 역시 GC에 사용되는 기법 중의 하나로 다음의 두 가지 과정을 통해 garbage를 회수한다.

[1단계] 사용 되지 않는 객체 구분

- 루트집합에서부터 포인터를 따라 접근할 수 있는 객체들을 마크(mark)한다. 이렇게 마크된 객체들은 프로그램 실행 동안 계속해서 사용되는 객체들이다.

[2단계] 가비지 회수

- 마크되지 않은 객체들의 메모리(garbage)를 회수한다

4. 메모리 누수 검출 기법

참조 계수 기법을 이용하여 메모리 누수를 검출하는 방법을 제안하기에 앞서, 그래프를 이용하여 메모리 상태와 메모리 누수를 모델링 한다. 그리고 메모리 누수 검출에 필요한 메타데이터와 메타데이터 변환 법칙을 정의하였다. 이를 통해 참조 계수 기법을 이용한 메모리 누수 검출 방법을 설명한다.

4.1 메모리 모델링

메모리 참조 구조(할당된 메모리 블록과 포인터 변수간의 관계)를 유향 그래프(directed graph)로 나타내면 다음과 같이 정의할 수 있다.

$$G(N, E) = (M \cup P, Ex \cup In)$$

- M = {m| m은 Heap에 할당된 메모리 블록}
- P = {p| p는 메모리 블록 m을 가리키는 포인터. m∈M}
- Ex = {(p, m)| 포인터 p가 메모리 블록 m을 가리키고 있다. p∈P, m∈M}
- In = {(m, p)| 메모리 블록 m이 내부 포인터 p를 포함하고 있다. p∈P, m∈M. }

따라서 메모리 누수는 아래와 같이 나타낼 수 있다.

(메모리 누수 블록의 집합) ={m | m∈M ∧ SAP(m)=ϕ }

이 때,

- ✓ P_In = { p | p∈P ∧ (∃m∈M ((m, p)∈In)) }
- ✓ SAP(m) = {(s, m_1, p_1, m_2, p_2, … , m_n, p_n, m)| s∈P−P_In ∧ m∈M ∧ (s, m_1),(p_i, m_{i+1}), (p_n, m)∈Ex ∧ (m_i, p_i)∈In. 단, n은 자연수 ∧ i는 0≤i≤n인 자연수}

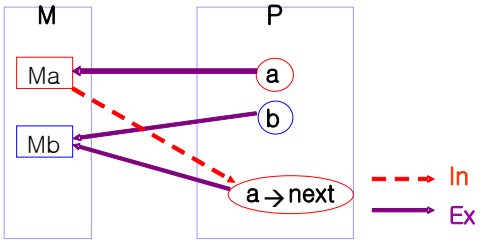
예를 들어, [그림 4]과 같은 C언어 코드가 있다고 하면 (3) 위치까지 수행된 시점에서의 그래프는 [그림 5]와 같다.

```
struct link
{
    struct link *next;
};

int main(){
    struct link *a, *b;
    a=(struct link*)malloc(sizeof(struct link)); /*(1)*/
    b=(struct link*)malloc(sizeof(struct link)); /*(2)*/
    a->next = b;                                 /*(3)*/
    a = NULL;                                    /*(4)*/
    …
}
```

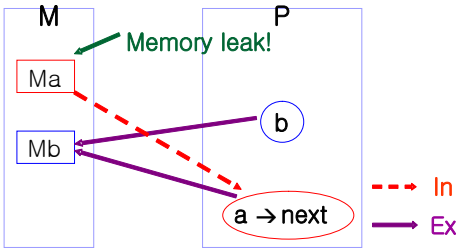
<그림 4. 메모리 모델링 예시 코드>

[그림 5]에서 사각형은 할당된 메모리 블록의 시작주소를 나타내고, 동그라미는 포인터 변수들의 주소 값을 나타낸다.



<그림 5. 메모리 모델링>

[그림 4]의 코드에서 (1)과 (2)를 수행하게 되면 포인터 a, b가 각각 Heap에 할당된 메모리 Ma, Mb를 가리키게 된다. (3)을 수행하게 되면 a→next가 메모리 Mb를 참조하는 것을 나타내는 실선과 a→next가 Ma 내부에 존재하는 포인터라는 것을 표현하는 점선이 추가된다. (4)가 수행되면 [그림 6]과 같은 상황이 된다. 이 때 메모리 블록 Ma에 도달 가능한 경로가 존재하지 않으므로 Ma는 메모리 누수가 된다.



<그림 6. 메모리 모델링에서의 메모리 누수>

4.2 메타데이터 정의

4.1절에서 설명한 모델링을 통해 메모리 누수를 검출하는 도구를 구현하기 위해서는 테스트할 프로그램의 실행 상태를 표현하는 메타데이터가 필요하다. 메타데이터는 메모리 메타데이터, 포인터 메타데이터, 그리고 호출 스택 메타데이터로 나눌 수 있다.

1)메모리(memory) 메타데이터

- Memory Information Element (MIE)
  Heap에 메모리블록이 할당될 때마다 생성되는 메타데이터로써 할당된 메모리 블록의 시작주소, 블록의 크기, 참조수, IPL의 참조 등의 정보를 담고 있으며 디버깅 정보 제공을 위해 추가적으로 할당 시점의 소스 코드 상의 파일 이름과 줄 번호, 호출 스택 정보를 가지고 있다.
- Memory Information Set (MIS)
  MIE들을 관리하기 위하여, 생성된 모든 MIE들의 정보를 가진다. 메모리 모델링에서 집합 M에 해당한다.
- Internal Pointer List Element (IPLE)
  Heap에 할당된 메모리 블록 내에 다른 메모리 블록을 가리키는 포인터가 존재할 경우에 생성되는 메타데이터로써 PE를 통해 참조관계 정보를 저장한다. 모델링에서 In 에지(edge)에 해당한다.
- Internal Pointer List (IPL)
  IPLE들을 관리하기 위한 메타데이터다. 메모리 모델링에서 집합 In에 해당한다.

2)포인터(pointer) 메타데이터

- Pointer Element (PE)
  어떤 포인터가 Heap에 할당된 메모리 블록을 가리킬 때 생기며 포인터의 메모리 주소와 포인터가 가리키는 메모리 블록의 메타데이터 MIE에 대한 참조를 가지고 있다. 메모리 모델링에서 집합 P, Ex에 해당한다.
- Stack Point Address Set (SPAS)
  로컬 포인터 변수들을 관리하기 위한 메타데이터. 로컬 포인터 변수들의 PE 정보를 가지고 있다.
- Pointer Address Set (PAS)
  전역 변수 혹은 Heap에 할당된 메모리 블록 내부에 존재하는 포인터들에 대한 정보를 가지고 있는 메타데이터. 해당하는 포인터들의 PE들을 참조할 수 있다.

3)호출 스택(call stack) 메타데이터

- Call Stack Element (CSE)

함수가 호출될 때마다 생성되고 함수가 종료될 때 사라지는 메타데이터로써 호출된 함수의 이름과 해당 함수 내에 로컬 변수로 존재하는 포인터들의 정보(SPAS), RCL 정보를 담고 있다.

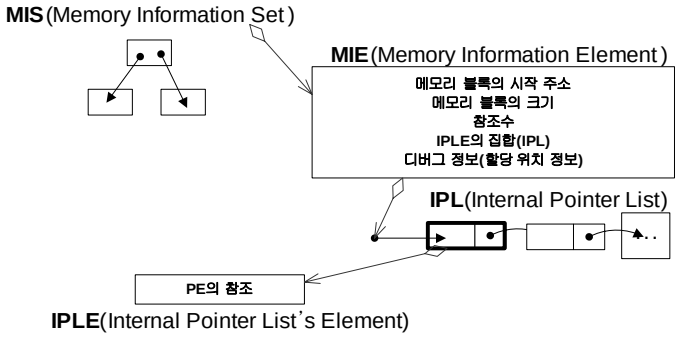
- Call Stack (CS)

  프로그램의 함수 호출 상황을 나타내기 위한 메타데이터로써 CSE의 참조를 가지고 있다. CS를 통해 특정 시점의 함수 호출 순서를 알 수 있다. 검출된 메모리 누수를 보고할 때 메모리 할당 시점과 메모리 누수 시점의 함수 호출 정보를 제공하기 위해서 필요하다.

- Return Check List (RCL)

  메모리 누수의 가능성이 있는 메모리 블록에 대해서 메모리 블록의 메타데이터 MIE의 참조 정보를 가지고 있는 메타데이터이다.

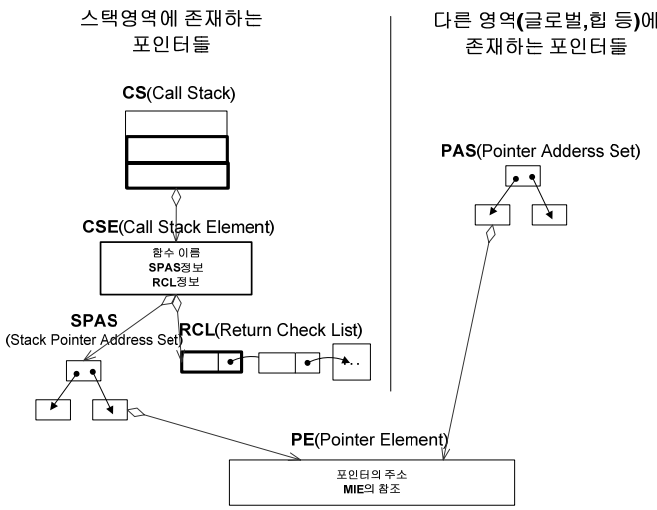
메모리 메타데이터들의 관계는 [그림 7]과 같이 다이어그램으로 나타낼 수 있다.



<그림 7. 메모리 관련 메타데이터의 구조>

[그림 7]과 [그림 8]은 각각 메모리 메타데이터들의 관계, 포인터와 호출 스택 메타데이터들의 관계를 다이어그램으로 나타낸 것이다. [그림 8]에서 포인터 메타데이터 PE(Pointer Element)의 경우 포인터가 메모리의 어떤 영역에 존재하는지에 따라 PAS(Pointer Address Set) 혹은 SPAS(Stack Pointer Address Set)에 의해 참조된다. 이렇게 메모리의 스택 영역에 존재하는 포인터의 메타데이터를 따로 분리하였을 때에 얻는 이점은 다음과 같다.

프로그램 수행 중 어떤 함수의 수행이 종료되었을 때 그에 따른 메타데이터를 유지하기 위해서는 함수 내의 로컬 변수에 의해 생성된 포인터의 메타데이터들을 모두 제거해 주어야 한다. 이 때, 종료되는 함수에 해당하는 CSE의 SPAS를 제거함으로써 함수의 로컬 포인터와 관계된 모든 PE를 손쉽게 제거할 수 있다. 또한 스택 영역에 존재하는 포인터의 메타데이터 PE는 상대적으로 PAS가 참조하는 PE에 비해 빈번하게 사용되므로 도구 구현 시 특정 포인터 주소에 해당하는 PE를 찾을 때 속도 면에서 이점을 지니게 된다.



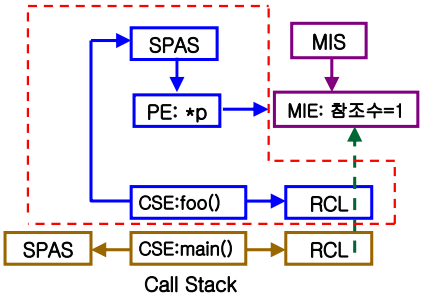
<그림 8. 포인터와 호출 스택 관련 메타데이터의 구조>

RCL은 메모리 누수 여부를 확인해야 하는 MIE의 참조 정보를 가지고 있는 메타데이터이다. [그림 9]의 코드를 살펴보면 foo함수는 malloc함수를 통해 할당된 메모리의 시작주소를 반환한다. 만약 (가)코드를 수행하는 경우라면 foo함수에 할당된 메모리가 free함수에 의해 해지될 때까지 메모리를 참조하는 포인터 변수가 존재하므로 메모리 누수가 발생하지 않는다. 그러나 (나)코드를 수행할 경우 foo함수가 반환하는 값을 받는 포인터 변수가 없으므로 참조수가 0이 되어 메모리 누수가 발생한다. 예제에서처럼 어떤 함수가 포인터를 반환하는 경우에는 메모리의 누수 여부를 포인터 반환 시점에 알 수 없다. 하지만 누수 가능성이 있는 메모리의 메타데이터 MIE의 참조를 RCL에 저장함으로써 함수가 수행된 이후에 메모리 누수 여부의 확인이 가능하다.

```
1:char *foo(){
2:    char *p=(char*)malloc(10);    /*(1)*/
3:    return p;                     /*(2)*/
4:}
5:int main(){
   (가)
     6:char *q=foo();               /*(3)*/
     7:free(q);
   (나)
     6:foo();                       /*(4)*/
8:   return 0;
9:}
```

<그림 9. 포인터를 리턴 하는 함수의 예시 코드>

[그림 9]의 코드에서 (2)가 수행되기 직전의 메타데이터 정보는 [그림 10]과 같다. (2)가 수행되면 foo함수가 포인터 반환과 함께 종료되기 때문에 [그림 10]에서 점선으로 둘러싸인 영역의 메타데이터가 사라지고 MIE의 참조수가 0이 되지만 MIE를 가리키는 포인터를 반환하고 있으므로 메모리 누수를 보고 하지 않고 main함수의 RCL에 MIE의 참조를 저장한다. 그리고 (3) 혹은 (4)가 수행된 이후에 RCL을 검사하여 메모리 누수 여부를 확인한다.



<그림 10. 포인터를 리턴 하는 함수에서의 누수 확인>

4.3 메타데이터 변환

프로그램 실행에 따라 메모리 상태가 변한다. 그에 따라 메타데이터도 업데이트 하여야 정확하게 메모리 누수를 검출할 수 있다. 메타데이터를 업데이트 해야 할 시점은 [표 1]과 같이 정리할 수 있다.

<표 1. 메타데이터 변환 시점>

번호	위치	설명
1	메모리 할당 함수	malloc, memalign 등의 함수
2	포인터 할당 (pointer assignment)	char *p, *q; … p = q;
3	구조체 할당 (structure assignment)	struct st a, b; … a = b;
4	메모리 재할당 함수	realloc 함수
5	메모리 해지 함수	free 함수

6	함수의 시작	char* foo(char *p)
		{
		←
		…
		}

7	블록의 끝	{
		…
		{
		char *p=(char)malloc(10);
		…
		←
		}
		}

8	함수의 끝	char * foo(char *p)
		{
		…
		if(…) return q;
		…
		return p;
		}

9	포인터를 반환하는 함수를 수행한 후	… char *p = foo(); ←

프로그램 수행 중 메모리가 할당/재할당, 해제 될 때에는 메모리 관련 메타데이터를 갱신한다. 메타데이터 MIE의 경우, 메모리 할당 함수로 Heap에 메모리 블록을 할당할 때 생성되어 초기화 된다. 초기화 작업에는 메모리 블록의 시작 주소와 크기를 메타데이터에 기록하고 참조수 정보를 0으로 만드는 일 등이 포함된다. realloc과 같은 메모리 재할당 함수가 불릴 때에는 메모리 블록의 시작 주소와 크기 변경에 따른 메타데이터들을 갱신한다. 메모리 해지 함수가 불리게 되면 해지되는 메모리블록의 메타데이터 MIE의 IPL에 존재하는 각각의 IPLE에 대해서, IPLE에 의해 참조되는 MIE의 참조수를 1만큼 줄여줌으로써 정확한 참조수를 유지한다.

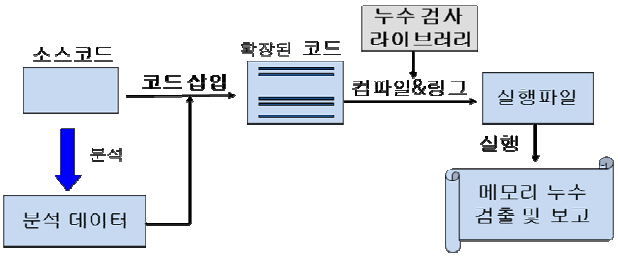
포인터나 구조체 할당이 일어날 때에는 포인터 관련 메타데이터가 변경된다. 포인터나 구조체 할당으로 인해 포인터가 가리키는 값이 달라지는 경우 포인터가 메모리의 어느 영역에 존재하느냐에 따라 PAS 혹은 SPAS의 PE정보를 갱신해 준다. PE정보가 바뀔 때 PE가 가리키는 MIE의 참조수도 변화시킨다.

함수가 호출되면 메타데이터 CSE가 생성되어 CS에 추가된다. 함수 수행 중 함수의 인자(argument)이나 로컬변수로 인해 생성되는 포인터들의 메타데이터 PE는 CSE의 SPAS에 추가가 되고 블록의 끝이나 함수의 끝에서 삭제가 된다. 이 때에도 마찬가지로 PE가 가리키는 MIE의 참조수를 변화시켜 준다.

포인터를 반환하는 함수가 수행된 이후에는 메타데이터 RCL이 가리키는 MIE들의 참조수가 0인지를 확인함으로써 메모리 누수 여부를 확인한다

5. 구현

제안한 기법을 지원하는 도구 구현을 위해 메타데이터들을 초기화 및 관리하는 API를 구현하고 [표 1]에 따라 소스 코드 내 API 호출 구문의 삽입 위치를 정의하였다.



<그림 11. 메모리 누수 검출 자동화 도구의 구조>

메모리 누수의 검출 과정은 [그림 11]과 같다. 테스트 대상 프로그램의 소스코드를 메타데이터 변환 규칙에 따라 누수 검사 라이브러리의 API 구문을 삽입하여 확장하고, 확장된 코드를 메모리 누수검사 라이브러리와 링크(link)시켜 실행 파일을 생성한다. 생성된 실행 파일을 테스트 환경에서 수행 중에 특정 메모리의 참조수가 0이 되어 누수가 발생하면 누수가 일어난 메모리 블록의 시작주소, 크기, 메모리 할당과 누수 시점의 호출 스택과 소스코드 상의 위치 정보를 실시간으로 혹은 프로그램 종료 시점에 보여 준다. 순환구조를 가진 메모리 누수는 Mark-Sweep 기법을 통해 주기적으로 혹은 프로그램의 종료 시점에 보고한다. 이 때에는 누수 시점의 호출 스택과 소스코드 상의 위치 정보는 제공되지 않는다. 마지막으로 프로그램의 종료 시점에는 메모리 누수는 아니지만 종료 시까지 메모리를 해지하지 않은 블록에 대한 정보를 출력한다. [그림 9]의 코드에 대해서 검출된 누수의 출력 결과는 [그림 12]와 같다. 결과를 살펴보면 소스 코드의 2번째 줄에서 할당된 10bytes의 메모리 블록이 main함수의 6번째 줄에서 누수 되었음을 알 수 있다.

```
==== Memory Leak Detected by Reference Count ====

 [ 1 ] == Leaked at ==
        at main (test.c:6 block address:0X80521F8 size:10)
       == Allocated at ==
        at foo (test.c:2 block address:0X80521F8 size:10)
          main

==== Leaked Memory Block(Cyclic Reference Group) ====

====Unfreed Memory Block List(No Memory Leak)====
```

<그림 12. [그림 9] 코드의 메모리 누수 출력 결과>

6. 사례 연구

본 논문에서는 사례연구를 통하여 제안하는 메모리 누수 검출 기법의 효용성을 확인해 보았다.

실험에는 오픈 소스 프로그램 중 메모리 누수가 발생하는 것으로 알려진 Html Tidy 프로그램을 사용하였다. Tidy는 HTML/XHTML 문서의 문법상의 오류를 검사하고 보정해주는 기능을 제공하며 약 160000 LOC로 이루어져 있다.

메모리 누수를 발생시키는 것으로 알려진 3개의 테스트 케이스를 이용하여 본 논문에서 구현한 도구와 Valgrind를 통해 메모리 누수 검출 결과를 비교해 보았다. 사용한 테스트 케이스에 대한 정보는 [표2]와 같다

<표 2. 테스트 케이스 정보>

테스트 케이스	알려진 누수블록의 개수 (개)	누수 블록의 총 바이트 수 (bytes)
1	2	65
2	3	37
3	6	193

```
==== Memory Leak Detected by Reference Count ====

 [ 1 ] == Leaked at ==
        at ParseInline (parser.c:1083 block address:0X80981C0 size:60)
           ParseTag
           ParseBody
           ParseTag
           ParseHTML
           ParseDocument
           main
       == Allocated at ==
        at MemAlloc (tidy.c:154 block address:0X80981C0 size:60)
           NewNode
           TagToken
           ...
```

<그림 13. 테스트 케이스 1에 대한 누수 검사 결과의 일부>

실험 결과 우리가 제안한 기법과 Valgrind는 알려진 메모리 누수 블록을 모두 찾아내었고 거짓 경보가 발생하지 않았다. 하지만 Valgrind의 경우 검출된 누수에 대해서 메모리 블록의 할당 정보만을 제공해 주기 때문에 Valgrind가 알려주는 정보만으로는 누수의 근본 원인을 알아내어 고치기가 어려웠다. [그림 13]과 같이 우리가 제안한 기법은 프로그램의 종료 시점이 아닌 메모리 누수가 발생한 순간에 누수가 일어난 위치와 메모리의 할당 정보를 제공해 주었다. 이 정보를 통하여 소스코드 상의 어느 위치에서 메모리 누수가 발생하였는지 알 수 있었고, 누수가 일어난 근본 원인을 파악하여 고칠 수 있었다.

7. 결론

본 논문에서는 C언어로 작성된 프로그램의 메모리 누수를 검출하는 기법을 제안하였다. 제안하는 누수 검출 기법은 참조 계수 기법을 이용하여 메모리 사용 상태를 실시간으로 모델링 하기 때문에 메모리 누수가 일어나는 시점에 검출이 가능하고, 기존의 도구들이 제공하지 못하는 디버깅에 유용한 정보들을 제공하므로 디버깅을 용이하게 할 수 있다. 또한 코드 삽입을 통해 누수를 검출하므로 플랫폼에 덜 의존적이다.

그리고 본 연구에서는 제안한 기법을 지원하는 도구를 구현하여 사례 연구를 수행하였다. 사례 연구 결과 기존의 도구들과는 달리 메모리 누수가 발생하는 시점에서 메모리 누수 검출을 보고할 수 있고 디버깅에 유용한 정보를 제공할 수 있다는 것을 확인할 수 있었다.

하지만 본 연구에서 제안한 방법은 소스 코드를 필요로 하기 때문에 소스 코드가 존재하지 않는 라이브러리를 사용하는 경우에 메타데이터를 정확하게 유지하기 어렵다. 그리고 다중 쓰레드 환경은 지원하지 않는다. 이러한 문제들을 해결하고 보다 많은 디버깅 정보를 제공할 수 있는 방안에 대해서 계속해서 연구 중이다.

참고 문헌

[1] F. Qin, S. Lu, and Y. Zhou, “SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption during Production Runs”, 8th International Symposium on High Performance Computer Architecture, February 2005.

[2] MemCheckDeluxe http://prj.softpixel.com/mcd/

[3] MemWatch http://www.linkdata.se/sourcecode.html

[4] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision", Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005.

[5] J. Maebe, M. Ronsse, and K.D Bosschere. “Precise detection of memory leaks”, In Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004), Edinburgh, Scotland, May 2004.

[6] P.R. Wilson, “Uniprocessor Garbage Collection Techniques”, International Workshop on Memory Management, St. Malo, France, September 1992.

[7] J.H. McBeth, "On the Reference Counter Method", Communications of the ACM, 6(9):575, September 1963.

[8] J. Cohen, “Garbage Collection of Linked Data Structures”, ACM Computing Surveys, 13(3):341-367, September 1981.