함수 요약에 기반한 메모리 누수 정적 탐지기

(A Static Analyzer for Detecting Memory Leaks based on Procedural Summary)

정 영 범 † 이 광 근 ††

(Yungbum Jung) (Kwangkeun Yi)

요 약 C프로그램에서 발생할 수 있는 메모리 누수(memory leaks)를 실행 전에 찾아 주는 분석기를 제안한다. 이 분석기는 SPEC2000 벤치마크 프로그램과 여러 오픈 소스 프로그램들에 적용시킨 결과 다른 분석기에 비해 상대적으로 뛰어난 성능을 보여준다. 총 1,777 KLOC의 프로그램에서 332개의 메모리 누수 오류를 찾아냈으며 이 때 발생한 허위 경보(false positive)는 47개에 불과하다(12.4%의 허위 경보율). 이 분석기는 초당 720 LOC를 분석한다. 각각의 함수들이 하는 일을 요약하여 그 함수들이 불려지는 곳에서 사용함으로써 모든 함수에 대해 단 한번의 분석만을 실행한다. 각각의 함수 요약(procedural summary)은 잘 매개화 되어 함수가 불려질 때의 상황에 맞게 적용할 수 있다. 실제 프로그램들에 적용하고 피드백 받는 방법을 통해 함수가 하는 일중에 메모리 누수를 찾는데 효과적인 정보들만으로 추리는 과정을 거쳤다. 분석은 요약 해석(abstract interpretation)에 기반하였기 때문에 C의 여러 문법 구조와 순환 호출(recursive call), 루프(loop)등은 고정점 연산(fixpoint iteration)을 통해 자연스럽게 해결한다.

키워드 : 요약 해석, 메모리 누수, 정적 프로그램 분석, 함수 요약, 탈출 분석

Abstract We present a static analyzer that detects memory leaks in C programs. It achieves relatively high accuracy at a relatively low cost on SPEC2000 benchmarks and several open-source software packages, demonstrating its practicality and competitive edge against other reported analyzers: for a set of benchmarks totaling 1,777 KLOCs, it found 332 bugs with 47 additional false positives (a 12.4% false-positive ratio), and the average analysis speed was 720 LOC/sec. We separately analyze each procedure's memory behavior into a summary that is used in analyzing its call sites. Each procedural summary is parameterized by the procedure's call context so that it can be instantiated at different call sites. What information to capture in each procedural summary has been carefully tuned so that the summary should not lose any common memory-leak-related behaviors in real-world C program. Because each procedure is summarized by conventional fixpoint iteration over the abstract semantics ('a la abstract interpretation), the analyzer naturally handles arbitrary call cycles from direct or indirect recursive calls.

Key words : Abstract Interpretation, Memory Leaks, Static Program Analysis, Procedural Summary, Escape Analysis

· 본 연구는 교육인적자원부 두뇌한국21사업의 지원과 교육과학기술부/한국과학재단 우수연구센터 육성사업의 지원으로 수행하였다.(R11-2008-007-01002-0)
· 이 논문은 프로그래밍언어연구회 2008 추계학술대회에서 '함수 간추림을 이용해 메모리 누수를 찾아내는 분석기'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 서울대학교 컴퓨터공학부
dreameye@ropas.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
kwang@ropas.snu.ac.kr

논문접수 : 2009년 3월 17일
심사완료 : 2009년 5월 20일

정보과학회논문지: 소프트웨어 및 응용 제36권 제7호(2009.7)

1. 서 론

메모리 누수는 C 프로그램에서 치명적이다. 종료되지 않고 지속적으로 실행되는 프로그램에서는 아주 적은 양의 메모리 누수도 조용히 계속 메모리를 잠식하여 결국 프로그램이 비정상으로 종료되도록 한다. 이런 경우 프로그램이 어떤 이유로 종료하였는지 C 프로그램 코드만을 보고 알아내기란 여간 어려운 것이 아니다. 우리의 분석은 하나의 함수에 집중한다. 한 함수가 메모리 누수를 발생시킬 수 있는지를 분석을 통해 알아내고, 동시에 이 함수가 하는 일이 무엇인지를 요약하여 이 함수가 불려지는 곳에서 사용한다. 함수가 메모리 누수를 일으키는 경우는 다음과 같다. 메모리가 할당되고, 그 메모리가 해제되지 않고 함수 밖으로도 빠져나가지 못하는 경우다.

표 1 같은 프로그램에 대한 성능비교. 다른 분석기들의 결과는 참조한 논문들에서 가져왔다. Sparrow가 항상 더 많은 버그들을 찾아준다.

C program	Tool	Bug Count	False Alarm Count
SPEC2000 benchmark	SPARROW	81	15
	Fastcheck [1]	59	8
binutils · 2.13.1 & openssh · 3.5.p1	SPARROW	246	29
	Saturn [2]	165	5
	Clouseau [3]	84	269

이 논문에서 제안하는 분석기(Sparrow)는 자동으로 또 정적으로 C 프로그램을 분석해서 메모리 누수가 어느 지점에서 일어날 수 있는지를 찾아준다. 다른 기존의 메모리 누수 탐지하는 분석기들[1-4]과 비교한 결과 Sparrow가 같은 벤치마크 프로그램들에 대해서 항상 더 많은 버그를 찾아낸다.

Sparrow의 속도는 720LOC/sec으로 가장 빠른 분석기 FastCheck[1] 바로 다음이고, 허위 경보율은 유일하게 Saturn[2]보다만 높다. 하나의 버그를 찾기 위해 분석해야 할 C 프로그램의 크기는 Clouseau[3] 다음으로 작다.

표 2 전체적인 성능 비교. 다른 분석기에 대한 결과는 [4]에서 가져왔다. 하지만 이 모든 분석기가 서로 다른 프로그램들에 적용했다는 사실에 유의하기 바란다.

Tool	C Size KLOC	Speed LOC/s	Bug Count	False Alarm Ratio(%)
Saturn [2]	6,822	50	455	10%
Clouseau [3]	1,086	500	409	64%
FastCheck [1]	671	37,900	63	14%
Contradiction [4]	321	300	26	56%
SPARROW	1,777	720	332	12%

Sparrow는 각각의 함수들을 분석하여 메모리 누수를 찾고, 그 함수가 하는 일 중에 메모리 누수와 관련이 있는 정보들을 요약한다. 이 때 나중에 그 함수들이 불릴 때에만 알 수 있는 정보를 잘 매개화(parameterization)하여 호출되는 곳마다 다르게 실증화(instantiation)될 수 있게 한다. 요약해석에 기반하여 분석하기 때문에 루프나 임의의 순환 호출, 포인터를 사용한 앨리어싱(aliasing)같은 임의의 C 구조들을 자연스럽게 다룰 수 있다. 요약 카테고리(summary categories)들은 실제 C 프로그램에의 적용을 통해 재련되었다. 다양한 C 프로그램들에 Sparrow를 적용해보면서 어떤 정보들이 추가되어야 하고, 어떤 정보들은 배제되어도 되는지를 결정하였다. 그렇게 첨삭된 정보들은 우리가 실제 적용을 통해 알아낸 것들이지만, 실제로 모든 조합가능한 정보들 중에 효율적인 정보를 추렸다는 사실을 이 논문을 통해 보여준다. Sparrow는 어떤 프로그램이 모든 메모리 누수로부터 안전하다고 이야기 해주지는 못한다. 이는 참조된 다른 논문들도 마찬가지다. 다만 Sparrow가 많은 버그들을 잡으면서도 상대적으로 낮은 허위경보율을 보인다는 사실을 앞에 두 표를 통해 보여줄 수 있을 뿐이다.

1.1 분석의 개요

분석은 두가지 과정이 결합되어 진행한다. 첫째로 함수가 하는 일 중에 메모리 관련된 일을 요약하기, 둘째로 그렇게 요약한 정보들을 함수가 불릴 때마다 사용하기. 어떤 함수 요약은 그 함수를 부르는 함수를 요약하기 위해서 꼭 필요한 정보들을 제공한다. 고로, 한 함수를 제대로 분석하기 위해서는 그 함수가 부르는 모든 함수들은 이미 요약되어 있어야 한다. 따라서, 우리의 분석은 함수의 호출 그래프(static call graph)의 역방향 위상 순서(reverse topological order)로 진행한다. 만약 호출 순환이 있으면(정적으로든 동적으로든) 그 순환위에 있는 모든 함수들을 한꺼번에 분석하여 고정점 연산을 한다.

아래와 같은 함수 f,g를 분석하면 그 아래 그래프와 같이 함수 요약을 표현할 수 있다.

```
int *f(List *p) {
  if (n) return malloc(4);
  else {
    p->next = malloc(10);
    return 0;
  }
}
```

```
int *g(){
  List k;
  int *a = f(&k);
  return a;
}
```

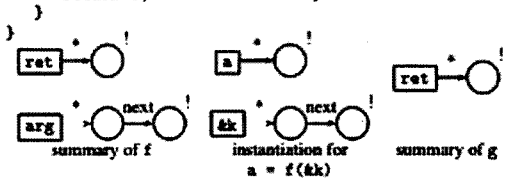


그림 1 예제 함수들의 요약과 실증

함수 요약의 그래프 표현은 2.1장에서 자세히 다루겠다. 여기서는 독자의 이해를 돕기 위해 간단하게 설명을 하겠다. 함수 f의 요약은 함수가 돌려주는 값이 할당된(allocated) 주소이고(느낌표!가 있는 노드의 의미는 할당된 주소라는 뜻) 인자가 가리키는 주소의 next 필드가 가리키는 주소가 할당되었다는 것을 나타낸다. 이때 실선으로 표현된 참조 관계는 이 함수가 만들어낸 관계이고, 점선은 이 함수를 부르기 전에 이미 존재하는 참조 관계를 나타낸다. 이제 함수 g안에 있는 함수 f를 호출하는 부분을 보자. 이때 실제 인자(actual argument)는 &k이고, 리턴값(return value)를 저장하는 실제 포인터는 a이다. 따라서, 함수 f를 이 호출 위치에서 실증화 하면 가운데 그래프와 같이 함수 f의 요약에서 arg, ret를 대체한 모습이 된다. 이제 함수 g가 끝나는 부분인 return a에서 이 함수의 외부에서 볼 수 있는 주소는 a를 통해서 참조 가능한 할당 주소 뿐이다. 변수 k를 통해서 참조 가능한 할당 주소는 외부에서 참조가 불가능하다. 따라서, 함수 g에 메모리 누수가 있다는 사실을 알 수 있고, 동시에 함수 g가 하는 역할은 맨 오른쪽 그래프와 같이 요약할 수 있다. 함수 g가 리턴하는 주소는 할당된 주소이다.

1.2 메모리 변화를 요약하기

함수를 요약하는 과정은 두 단계로 이루어진다. 먼저 함수의 메모리 변화를 계산하고, 계산된 메모리로부터 함수가 하는 일 중 메모리 누수와 관련이 있는 정보를 추출해낸다. 요약해석[5,6]의 프레임웍(framework)을 이용하여 메모리를 계산한다.

이때 계산하는 메모리는 세가지 정보를 담고 있다. 할당된 주소들이 무엇인지, 해제된 주소들이 무엇인지, 그리고 메모리 상태(어떤 주소들이 어떤 값들을 가리키고 있는지에 대한 정보)이다. 이 메모리들을 함수가 끝나는 지점에서 관찰하여 함수가 하는 일을 요약한다. 우리가 모으는 정보는 함수가 끝나고 났을때 외부 환경을 통해 접근이 가능한 모든 주소들에 대한 정보다. 어떤 함수가 접근하는 주소들 중 외부에서 볼 수 있는 주소들은 모두 전역 변수(global variables)들이나 함수의 인자들 혹은 리턴값을 통해서 접근 가능한 모든 주소들이다. 따라서, 그러한 주소들을 계산하여 그 중 할당된 주소가 있는지 해제된 주소가 있는지 아니면 앨리어스된 주소들이 있는지 계산하여 그 정보를 함수 요약으로 만든다.

접근 경로를 통해 메모리 효과를 매개화 하기 함수들의 메모리 상태를 계산하는데 처음으로 부딪히는 난관은 분석을 시작할 때 도대체 어떤 메모리 상태(호출 상태 call context)로부터 시작해야 하는지 모른다는 사실이다. 그러므로, 지금은 모르지만 호출될 때 알 수 있는 사실들은 모두 매개화해서 저장하고 있어야한다. 어떻게 매개화 하는 지 살펴보자.

우선 다음과 같은 사실을 알 수 있다. 우리는 입력 메모리 상태의 모든 것을 알 필요는 없다. 오직 어떤 함수에서 접근되는 주소들만을 알면 된다. 이 함수가 사용하지 않는 주소들은 절대로 변하지 않는다. 물론 병렬화된 프로그램(parallelized program)의 경우는 다르다. 우리의 분석은 병렬화된 프로그램에는 적합하지 않다. 다음은 분리 로직(separation logic)[7,8]에서 자주 등장하는 frame rule이다.

$$\frac{\{p\}c\{q\}}{\{p^*r\}c\{q^*r\}}\ \text{frame rule}$$

여기서 r에 있는 자유 변수가 함수 c에 의해서 변경되지만 않는다면 함수 c가 수정하는 메모리 p만 q로 변하게 된다. 우리 분석에서 r은 함수 c가 변화시키지 않는 전역 메모리 상태이다. 이 전역 메모리는 함수 c를 분석할때는 굳이 알 필요가 없다(사실 알 수도 없다). 물론 변경되지 않고, 사용만 되는 전역 변수들도 있을 수 있으나 그런 것들은 모든 가능한 경우를 고려하며 분석하는 것이 가능하다. 두 번째로 알 수 있는 사실은 C 프로그램에 있는 함수들은 인자나 전역 변수들을 통해서만 외부 환경을 접근할 수 있다는 사실이다. 마지막으로 우리는 이 함수를 통해 접근되는 주소의 값을 알 수는 없지만 그 것들이 어떤 경로를 통해 접근되는지는 알 수 있다. 그리고, 그 경로는 프로그램 코드에 드러나 있다.

앞의 예를 통해 구체적으로 살펴보자. 함수 f의 끝나는 지점에서의 메모리 상태를 계산해 보면 다음과 같다.

n	$[-\infty,-1][1,\infty]$
ret	ℓ_1

n	[0,0]
p	α
α.next	ℓ_2
ret	null

우선 오른쪽 메모리 상태를 살펴보자. 분기점에서 조건문의 값이 거짓일때의 마지막 지점에서의 메모리 상태이다. 오직 접근된 주소들에 대해서만 메모리 상태가 그려진다. 여기서 사용되는 주소들은 변수 n(조건문에서 사용되니까), 변수 p(p->를 통해), α.next (p->next를 통해), 그리고 리턴 값을 저장하기 위한 메타 변수(meta variable) ret가 있다.

여기서 우리가 주목해야 하는 주소는 α다. 이 주소는 프로그램에 드러나 있지 않은 주소이다. 하지만, 변수 p의 값을 꺼내올 때 그 값을 현재 메모리 상태에서는 알 수가 없는 경우 심볼릭 주소(symbolic address)를 도입하여 그런 주소들을 표현한다. 그리고, α.next도 역시 심볼릭 주소 α로부터 파생되는 심볼릭 주소이다. 이런 주소들은 만들어질 때 자신들이 어떤 주소로부터 생성되었는지 그 접근 경로(access path)를 기억한다. α는

변수 p로부터 생성되었고, α.next는 p->next로부터 생성되었다. 이 정보는 실증화를 통해 구체적인 값으로 치환이 될 때 정확히 어떤 주소들이 대체 되어야 하는지를 결정하는 데에 쓰인다.

동적으로 할당된 주소 ℓ_2은 malloc(10)을 통해 생성되는 모든 주소를 표현하는 요약 주소(abstract address)이다. 이 주소 ℓ_2도 함수가 호출되는 곳마다 저마다 다른 주소들로 실증화되어 다른 호출 지점에서 생성되는 주소들은 서로 다르게 표현되도록 한다.

왼쪽의 메모리 상태는 분기점에서 조건문의 값이 참일때를 나타낸다. 여기서 사용되는 주소들은 변수 n, 메타 변수 ret뿐이다. 이 때 ret는 malloc(4)을 통해 할당된 주소 ℓ_1을 가리키고 있다. 그리고, n은 0이 아닌 값을 갖고 있는데 우리가 모르는 전역 변수에 대해서 주소가 아닌 정수 값을 꺼낼 때는 모든 값을 가질 수 있다고 가정하고 분석을 진행한다. 여기서는 조건문의 결과를 이용하여 0이 아닌 모든 정수라고 표현된다. 이렇게 우리는 정수값에 대해서는 잘 알려진 구간 분석을 사용한다. C 프로그램에서 0이 아닌 값은 중요한 의미를 갖는다. 때문에 우리는 분석에서 0이 아닌 값을 표현하기 위해 두개의 구간쌍을 도메인으로 사용하고 있다.

함수 요약하기 함수가 하는 일을 여러 개의 카테고리로 쪼개어 저장한다. 각각의 카테고리는 함수 외부에서 접근 가능한 주소들에 대한 정보들과 변화를 담고 있다. 모든 접근 가능한 주소들은 접근 경로로 매개화하여 표현한다. 접근 가능한 주소들은 전역 변수, 함수 인자들, 그리고 리턴값으로부터 접근 가능한 주소들이다. 이 주소들과 할당/해제된 주소들에 대한 정보로부터 함수가 하는 일을 요약한다. 앞의 예에서, 함수가 끝나는 두 지점의 메모리 상태로부터 외부에서 접근 가능한 주소들 중에 할당된 주소들이 있다. 따라서, 함수 f는 그림 1과 같이 간단하게 요약할 수 있다.

1.3 요약 실증화

함수를 분석하는 중 다른 함수를 호출하는 지점을 만나면 호출되는 함수의 요약을 실증화하여 현재의 메모리 상태를 변화시킨다. 실증화는 함수 요약에 비어있는 부분을 현재 메모리를 보면서 치환하고, 그 결과를 다시 현재 메모리에 반영하는 방식으로 이루어진다. 이때 각각의 호출 지점마다 다르게 실증화가 되기 때문에 C 프로그램 분석에서 어렵다고 일컬어지는 앨리어스(alias)도 자연스럽게 해결된다. 아래의 예를 통해 앨리어스 된 주소가 어떻게 실증화에 영향을 끼치는지 살펴보자.

함수 f의 요약을 보면 이 함수가 첫번째 인자가 가리키는 주소를 해제(↻로 표시된 주소)하는 것을 알 수 있다. 또, 두 번째 인자로부터 *next를 통해 접근 가능한 주소도 해제한다는 사실을 알 수 있다. 함수 g안에 함

```
f(List *x, List *y) {
   free(y->next);
   free(x);
}
```

```
g() {
   List *a = malloc();
   List *b = a;
   a->next = malloc();
   f(a,b);
}
```

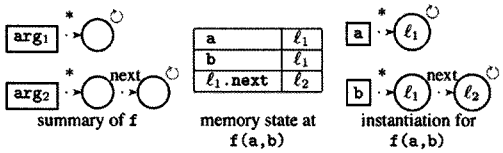


그림 2 호출 환경을 고려한 감추림 실증화

수 f를 호출하는 부분을 보면 두 개의 실제 인자 a,b가 모두 같은 주소 ℓ_1으로 가리키고 있는 메모리 상태를 갖고 있다. 이 상태를 이용해 실증화를 하면 맨 오른쪽에 나타나 있는 그래프와 같이 할당되었던 두 주소 ℓ_1, ℓ_2가 모두 안전하게 해제된다는 사실을 알 수 있다. 그런데 여기서 한가지 우리 분석기가 놓치고 있는 정보가 있다. 만일 함수 f의 두 해제 함수가 호출되는 순서가 바뀐다면 어떻게 될까? 그렇다면 x가 가리키는 주소가 먼저 해제되고, 그 다음에 그 주소로부터 next 필드를 통해 접근가능한 주소를 해제하려 할 것이다. 이때는 해제된 주소를 참조하는 오류가 발생한다. 하지만, 우리 분석기는 메모리 누수 이외에는 다른 오류는 존재하지 않는다는 가정하에 분석을 하기 때문에 그러한 경우에도 함수 f는 두 주소를 안전하게 해제한다고 잘 못된 분석을 한다.

1.4 이 논문의 기여

- 우리의 접근 방법은 현존하는 다른 실용적인 분석기에 비해 보다 빠르고 정확하며 많은 버그를 찾는 편이다. 다른 분석기들[1-4]과의 똑같은 프로그램에 대한 성능 비교를 통해 살펴보면 모든 경우에 Sparrow가 더 많은 버그를 찾아내고 있다. 분석 속도 측면에서 살펴보면 FastCheck[1]에 이어 두번째로 빠르고 (720 LOC/sec), 허위경보율(12.4%)은 Saturn[2]에 이어 두번째로 낮다.
- 메모리 누수를 찾는데 효과적인 함수의 정보를 제안한다. 이 정보들은 경로를 고려하지 않는(path-insensitive) 분석에서 달성할 수 있는 효율적인 정보를 포함하고 있다. 이 정보를 8가지의 카테고리(category)로 분류했다(2장).
- C 프로그램에서 메모리 누수를 찾는 분석기를 고안할 때 사용할 수 있는 효율적인 디자인 결정(design decision)들을 보고한다(3장). 여기 제시되어 있는 결정들은 분석의 안전성(soundness)를 깨뜨리기도 하지만 비용 대비 정확도 향상이 큰 방법들이다.
- 함수들을 다른 함수들과 격리해서 분석할 수 있는 방

법[9,10]을 제시한다. 각각의 함수들을 분석하고 그들이 하는 일을 잘 요약함으로써 함수들을 다시 분석하는 일이 없게 하고, 실증화의 비용이 크지 않은 상태로 전체 프로그램을 분석할 수 있다. 함수에서 사용하는 주소들을 접근 경로로 표현함으로써 함수를 매개화하여 요약하는 방법을 제시한다.

2. 함수 요약

함수들의 메모리 누수 관련 효과들을 분석하고, 외부로부터 접근 가능한 주소들에 초점을 맞춘다. 다시 한번 언급하지만, 외부로부터 접근 가능한 주소들은 모두 전역 변수, 함수 인자, 리턴 값들을 통해서만 접근이 가능하다. 그리고, 이 주소들간의 앨리어스 정보, 할당된 주소들, 해제된 주소들로부터 함수의 메모리 관련 효과들을 이해할 수 있다. 다시말해서, 우리가 알아야하는 정보는 할당된 주소가 외부에 어떻게 노출될 것인지, 외부로부터 볼 수 있었던 어떤 주소가 함수 내에서 해제되는지, 외부로부터 접근 가능한 주소들간의 앨리어스 관계가 어떤 가이다.

표 3 함수 요약 카테고리. 모든 조합 가능한 카테고리들 중에 우리가 사용하는 것은 이 8가지이다. 외부로부터 접근 가능한 주소들간의 앨리어스 정보, 할당된 주소들, 해제된 주소들로부터 위와 같은 정보를 추출해 낸다.

	free	global	argument	return
allocation	·	·	Alloc2Arg	Alloc2Ret
global	·	·	Glob2Arg	Glob2Ret
argument	Arg2Free	Arg2Glob	Arg2Arg	Arg2Ret

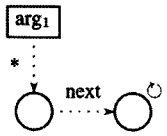
(1) 함수가 새롭게 할당한 주소가 외부에 어떻게 노출될 것인가? 할당된 소는 그 주소가 리턴되거나 호출하는 함수의 환경에서 보이는 주소에 저장되는 방법으로만 노출될 수 있다. 그리고, C 프로그램에서 그 환경에는 전역 변수와 포인터 인자만이 속한다. 따라서, 우리는 할당된 주소가 리턴되는지(Alloc2Ret), 혹은 인자에 저장되는지(Alloc2Arg)를 기억한다. 여기서 할당된 주소가 전역변수에 저장되는 경우(Alloc2Glob)는 기록하지 않는데, 그 이유는 전역 변수에 저장되는 주소는 모든 환경에서 접근 가능하기 때문이다. 모든 환경에서 접근 가능한 주소는 언제든지 쓰일 수 있기 때문에 메모리 누수로 보지 않는다. 이로 인해 함수간의 호출에서 발생할 수 있는 특정 유형의 메모리 누수는 찾지 못할 수 있다. 그 유형은 함수 호출을 통해 똑같은 전역 변수에 할당된 주소를 저장하고, 나중에 함수 호출을 통해 다른 할당된 주소로 덮어 쓰는 경우이다.

(2) 함수가 어떤 주소를 해제하는지에 대한 정보를 기록한다. 함수가 호출되기 전에 할당된 주소들이 이 함수에서 보이는 경우는 두 가지 경우 뿐이다. 함수 인자 혹은 전역 변수를 통해서. 인자를 통해서 보이는 주소가 해제되는지(Arg2Free)를 기록한다. 이때 전역 변수를 통해 접근 가능한 주소가 해제되는지(Glob2Free)에 대한 정보는 기록하지 않는데 이는 위에서 설명했듯이 전역 변수에서 접근 가능한 주소는 모든 환경에서 접근 가능하기 때문에 메모리 누수로 보기 어렵기 때문이다.

(3) 외부에서 보이는 주소들의 앨리어스 정보를 기록한다. 이는 함수간의 호출로 인해 발생하는 앨리어스를 이해함으로써 할당된 주소가 어디로 흘러가는지 정확하게 따라가기 위해 사용된다. 전역 변수, 함수 인자, 리턴 값들간에 발생할 수 있는 모든 앨리어스의 조합은 9가지이다. 이중에 다음의 5가지 경우만이 의미가 있다: 1) 전역변수를 인자에 저장하는 경우(Glob2Arg) 2) 혹은 리턴하는 경우(Glob2Ret)가 있다. 또, 3) 함수 인자를 전역 변수에 저장하는 경우(Arg2Glob) 4) 다른 인자에 넘겨주는 경우(Arg2Arg) 5) 리턴하는 경우 (Arg2Ret)가 있다. 나머지 중 세가지 카테고리(Ret2Glob, Ret2-Arg, Ret2Ret)는 불가능하다. 왜냐하면 리턴 값이라는 것은 함수가 끝나는 지점에서나 결정되는 것이기 때문이다. 다시 말해 함수가 어떤 값을 리턴하게 되면 그 함수가 하는 일은 끝나는 것이고 따라서 그 값을 다른 것에 저장하는 일 따위는 할 수 없다. 이제 마지막으로 남은 단 하나의 카테고리(Glob2Glob)은 우리의 분석에서는 고려하지 않는다. 분석의 효율성을 위해 전역변수들의 이름은 함수 내에서만 구분하고, 함수 호출을 통해 전역변수에 대한 메모리 변화가 일어날 때는 모든 전역 변수를 하나로 묶는다.

2.1 함수 요약 정보의 예

이 장에서는 8가지의 함수 카테고리를 예를 통해 보여주겠다. 함수 요약을 표현하기 위해 앞장에서 사용했던 그래프를 정확히 정의하고 사용하겠다. 함수 요약은 방향성 그래프(directed graph)로 표현된다. 노드(node)는 하나의 요약 주소를 표현한다. 원으로 표현된 노드는 힙 메모리에 있는 주소를 나타낸다. 사각형으로 표현된 노드는 스택에 있는 주소를 표현하거나 메타 변수들을 나타내는데 쓰인다. 이 함수에서 새롭게 할당된 주소는 !가 표시되어 있고, 해제된 주소는 ○가 표시되어 있다. 노드 a로부터 노드 b로의 간선(edge)은 주소 a가 주소 b를 가리키고 있음을 나타낸다. 간선위에 있는 레이블은 어떠한 방법으로 참조하고 있는지를 보여준다. 레이블 * 는 포인터 참조(dereference)를 통해 가리키고 있음을 뜻하고, 필드 이름이 있는 경우는 그 필드가 가리키고 있음을 뜻한다. 점선으로 표시된 간선은 이 함수가 불리




```
struct List{
   struct List * next;
   int *val;
};

freeNext(List *lst){
   free(lst->next);
}
```

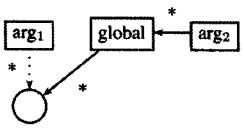
그림 3 Arg2Free: 이 함수는 인자로부터 접근 가능한 주소를 해제한다.

기 전에 이어져 있던 참조 관계를 나타낸다. 이 간선으로 연결된 주소들은 이 함수가 호출될 때의 메모리 상태에 따라 실증화되는 주소들이다. 실선은 이 함수에 의해 발생하는 참조 관계를 표현한다.

- Arg2Free(그림 3): 함수 freeNext는 함수의 인자로부터 접근 가능한 주소를 해제한다. 그 정보가 함수 요약으로 표현되어있다. 우리 분석에서 할당된 메모리를 해제하는 유일한 방법이다.
- Arg2Glob, Glob2Arg(그림 4): 이 예제에서 첫번째 인자의 노드는 점선으로 연결되어 있고, 두번째 인자의 노드는 실선으로 연결되어 있다. 점선으로 연결된 부분이 Arg2Glob 경우를 나타낸다. 함수 attachGlob 안에서 첫번째 인자가 점선을 통해 가리키고 있던 노드를 전역변수가 가리키게 된다. 두번째 인자에 있는 실선은 Glob2Arg를 나타낸다. 이 두가지 카테고리의 차이는 다음과 같은 코드를 통해 확연히 드러난다.

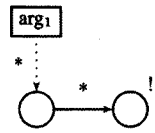
  ```
  int *p1 = malloc(1);
  int **p2;
  attachGlob(p1,&p2);
  *p2 = malloc(2);
  ```

  결과적으로 위의 코드에 나오는 두개의 할당된 주소들은 모두 전역 변수로부터 접근이 가능하게 되어 안전하다(메모리 누수가 일어나지 않는다). 함수 attachGlob은 우선 변수 p1이 가리키는 할당된 주소를 전역변수로부터 접근 가능하게 만든다. 거기에 더해 포인터 *p2를 전역변수와 앨리어스 상태로 만든다. 그러므로, 이후에 *p2에 붙는 주소는 결과적으로 전역변수로부터 접근가능하게 된다. 이와 같이 Arg2Glob은 인자가 가리키고 있던 주소를 전역변수에 붙이는 것이고, Glob2Arg은




```
attachGlob(int *p1,
           int ***p2){
   gInt = p1;
   *p2 = &(gLst.val);
}
```

그림 4 Arg2Glob, Glob2Arg: 이 함수는 첫번째 인자를 전역변수에 붙이고, 두번째 인자는 전역변수와 앨리어스 시킨다.

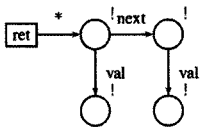



```
makeArray(int ** p){
   *p = malloc();
}
```

그림 5 Alloc2Arg: 이 함수는 할당된 주소를 인자에 붙인다. 할당된 노드는 !로 표시된다.

인자가 전역변수와 앨리어스가 되어 이후에 전역변수인 것처럼 변하는 것이다.

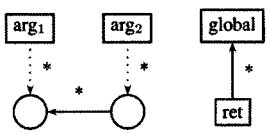
- Alloc2Arg(그림 5): 실제 C 프로그램에서 인자에 할당된 주소를 붙이는 경우가 많이 발생하였다. 이런 상황을 이해함으로써 더 많은 메모리 누수를 탐지할 수 있다.
- Alloc2Ret(그림 6): 실제 C 프로그램에서 많은 메모리 할당이 함수 호출을 통해서 이루어진다. 할당된 구조체를 리턴하는 함수를 불러서 사용하는 것이 가장 흔하게 사용되는 C 프로그래밍 방법이다. Sparrow는 할당된 구조체의 모양도 이해한다.




```
List * make2List(){
   List * lst = malloc();
   lst->val = malloc();
   lst->next = malloc();
   (lst->next)->val = malloc();
   return lst;
}
```

그림 6 Alloc2Ret: 이 함수는 길이가 2개짜리 리스트를 할당해 리턴한다.

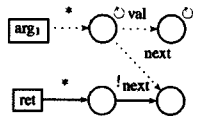
- Glob2Ret, Arg2Arg(그림 7): 리눅스 커널(Linux kernel)에 있는 어떤 함수들은 어떤 주소를 전역 테이블에서 가져와 리턴을 한다. 만약 할당된 주소가 이런 주소에 붙는다면 그것은 메모리 누수가 아니다. 오픈 소스 프로그램 중 하나인 프로그램 tar에 있는 어떤 함수는 인자로 리스트와 할당된 주소를 받아 그 리스트에 할당된 주소를 붙인다. 이런 경우에 리스트에서 그 할당된 주소가 접근 가능하다는 사실을 알지 못하면 메모리 누수가 있다고 허위 경보를 내게 된다. 이 예제에서는 첫번째 인자를 두번째 인자에 붙인다. 이 카테고리는 일반적으로 인자들 사이에 생기는 앨리어스 정보를 기억한다.




```
List * argPassing
(List *lst1,
 List **lst2){
   *lst2 = lst1;
   return &gLst;
}
```

그림 7 Glob2Ret, Arg2Arg: 이 함수는 첫번째 인자를 두번째 인자에 붙이고 전역 변수의 주소를 리턴한다.

- Arg2Ret(그림 8): 몇몇 라이브러리 함수(e.g. memcpy나 strcpy 등등)들은 인자가 가리키는 주소를 리턴한다. 이런 관계를 이해하지 못하면 할당된 주소가 위와 같은 함수들에 전달되었다 리턴되었을 때 그 주소를 추적하지 못한다. 따라서, 허위경보가 발생하게 된다.




```
List * renewList(List * lst){
  List * ret = malloc();
  ret->next = lst->next;
  free(lst->val);
  free(lst);
  return ret;
}
```

그림 8 Arg2Ret:이 함수는 인자로부터 접근 가능한 주소를 리턴한다.

2.2 함수 요약 정보 실증화

함수 요약 정보를 어떻게 실증화하는지 간단한 C 프로그램을 통해 보여주겠다(그림 9). 함수 leak은 변수 lst2가 가리키는 할당된 주소를 함수가 끝날 때 누수한다. 세번째 줄에서 함수 make2List가 호출된다. 이 함수는 이미 분석이 되었을테고 이 함수의 함수 요약(그림 6)을 이용할 수 있다. 변수 lst1이 리턴값을 가리키게 되고 따라서, 길이가 2인 할당된 리스트를 가리키게 된다. 네번째 줄에서 함수 renewList(그림 8)이 호출된다. 첫번째 인자는 lst1이 리턴값은 lst2가 실증화된다. 함수 renewList는 인자로부터 접근 가능한 두개의 주소를 해제한다.

```
    void leak( ) {
1:    List *lst1,*lst2;
2:    int **ptr;
3:    lst1 = make2List();
4:    lst2 = renewList(lst1);
5:    attachGlob((lst2->next)->val, &ptr);
6:    makeArray(ptr);
7:    freeNext(lst2);
    }
```

그림 9 이 함수는 위에서 소개된 함수들을 호출한다.

어떤 주소가 해제되는지는 접근 경로를 보고 좇아갈 수 있다. 여기서 해제되는 주소는 3번째 줄에서 할당되었었던 *lst1과 *(*lst1).val이다. 해제된 주소들은 할당되어 있는 주소들의 집합에서 제거되고, 해제된 주소들의 집합에 들어간다.

4번째 줄까지 실행하고 난 현재까지의 메모리 상태를 그림 10에 그려 두었다. 독자의 이해를 돕기 위해 메모리 상태를 테이블로 나타내지 않고 함수 요약을 표현하는데 사용했던 그래프로 표현하였다.

다섯번째 줄에서 (lst2->next)->val이 가리키는 할당된 주소가 attachGlob함수의 호출을 통해 전역 변수에 매달리게 된다. 또, 포인터 ptr은 전역 변수와 앨리어스 된다. 여섯번째 줄에서 makeArray함수는 할당된 주소를 ptr이 가리키게 한다. 일곱번째 줄에서 함수 free-Next를 호출함으로써 lst2->next가 가리키는 할당된 주소를 해제한다. 함수 leak이 끝나는 지점에서 메모리 상태를 그림 11에 나타내었다.

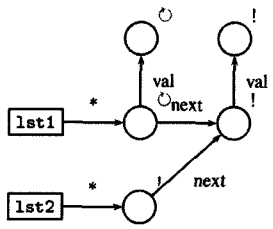


그림 10 그림 9에서 4번째 줄까지 실행시킨후의 메모리 상태. 어떤 주소들은 해제되었고, 어떤 주소들은 lst2로부터 접근이 가능하다는 사실을 볼 수 있다.

할당되었었던 주소들 중에 단 하나를 제외하고는 모든 주소가 해제되거나 전역 변수로부터 접근 가능하다. 따라서, 변수 lst2가 가리키고 있던 메모리만이 재활용이 불가능한 상태로 새고 있음을 알 수 있다.

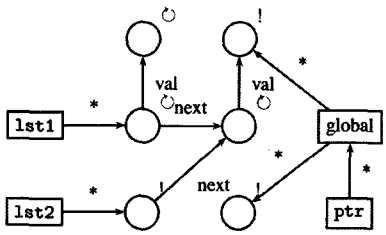


그림 11 함수 leak의 끝나는 지점에서의 메모리 상태. 변수 lst2가 가리키고 있는 주소 하나만이 전역 변수로부터 접근이 불가능하다.

3. 성능 향상을 위한 여러 가지 기술들

분석기를 개발하고 적용하는 과정 중에 여러 가지 선택이 가능한 상황이 있었다. 그때마다 정확도와 분석 비용사이에서 줄타기를 하며 실용적인 방향으로 성능이 개선되도록 선택을 했다. 그 과정에서 효과적이라고 생각하는 선택지들을 이 장에서 소개하도록 하겠다.

- 전역변수의 과감한 요약

함수를 분석 중에는 전역변수들을 구분하지만, 함수요약을 할 때 모든 전역변수들은 하나로 요약을 한다. 다시말해, 전역변수의 이름을 잊어버린다. 이는 불필요하게 전역변수들을 구분함으로 인해 메모리에 너무 많은

변수들이 생겨 성능이 크게 저하되는 것을 막기 위함이다. 물론 이로 인해 놓치는 메모리 누수의 유형이 있다. 함수 호출을 통해 똑같은 전역변수에 주소를 붙이는 경우 덮어씌여짐으로써 발생하는 메모리 누수는 Sparrow가 찾을 수 없다. 예를 들어 다음과 같은 코드는 명백히 메모리 누수가 발생함에도 불구하고 놓치는 경우이다.

```
int *gp;
f(int *p){ gp = p; }
g(){
  int *q = malloc();
  f(p);
  p = malloc();
  f(p);                    // overwritten memory leak!
}
```

- 경로를 고려하지 않는 분석에서 허위 경보 최소화하기

경로 고려 분석(path sensitive analysis)은 메모리와 시간의 부담이 크기 때문에 효율성을 위해 포기하였다. 이로 인해 허위경보가 많이 발생할 수 있는데 이를 막는 방향으로 선택 하였다. 카테고리 Arg2Free는 경로에 상관없이 해제되는 주소들을 모두 모은다. 이로 인해 어떤 할당되었던 주소가 한 쪽 경로를 따라 프로그램이 실행되면 안전하게 해제되고, 다른 경로를 따라 실행되면 해제되지 않고 그대로 남아있는 경우 실제로는 누수가 있지만 Sparrow는 찾지 못한다. 아래의 코드에서 함수 f는 인자 n이 0보다 큰 경우에만 인자 p를 해제하지만 요약될 때는 항상 해제하는 것으로 기억한다. 마찬가지로, 함수 g도 인자 n이 0보다 큰 경우에만 전역변수에 인자 p를 붙이지만 요약할 때 항상 붙인다고 기억한다. 우리 분석에서 해제하는 함수 free는 인자가 가리킬 수 있는 모든 주소들이 해제된다고 여긴다. 실제로는 그 중에 하나의 주소만이 해제되겠지만 해제하는 함수에 대해서 하는 일을 더 많게 봄으로써(over-approximation) 허위 경보를 줄이는 방향으로 선택을 한다. 예를 들어 아래의 코드에서 함수 h는 인자 x와 y 모두를 해제한다고 요약한다.

```
f(int n, int *p){        int *gp;          h(int *x, int *y){
   if(n>0) free(p);      g(int n, int *p){     int *p = x;
}      if(n>0)           gp = p; if(n>0)           p = y;
   }                      free(p);
      }
```

- 우리가 관심있는 정보에 대해서 약간의 경로 고려 분석의 느낌을 첨가하기

대부분의 허위 경보는 경로를 고려하지 않는 분석 때문에 발생한다. 예를 들어 아래와 같은 코드는 실제로 누수가 없는데도 우리 분석기는 허위경보를 내게 된다.

```
1: int f(int n){
2:   int *p = 0;
3:   if(n>0) p = malloc();
4:
5:   if(n>0) free(p);
6:
```

분석중에 가지고 있는 정보는 주소들 간에 참조 관계의 맵, 할당된 주소들의 집합, 그리고 해제된 주소들의 집합이다. 이때 위와 같은 코드가 어떻게 해서 허위 경보를 내게 되는지 그 경로를 간단하게 살펴보자. 네번째 줄에서는 세번째줄에서 분기된 두 가지 경로가 합쳐지게 된다. 여섯번째 줄에서는 다섯번째줄에서 분기된 두 경로가 합쳐지게 된다. 여섯번째 줄에서 최종 상태를 보면 할당된 주소들의 집합에 주소가 남아있어서 메모리 누수라고 판단할 수 있다.

$4: \langle\{p \mapsto \{0\}, n \mapsto [-\infty, 0]\}, \varnothing, \varnothing\rangle \cup \langle\{p \mapsto \{\ell\}, n \mapsto [1, \infty]\}, \{\ell\}, \varnothing\rangle$
$= \langle\{p \mapsto \{0, \ell\}, n \mapsto \top\}, \{\ell\}, \varnothing\rangle$

$6: \langle\{p \mapsto \{0, \ell\}, n \mapsto \top\}, \{\ell\}, \varnothing\rangle \cup \langle\{p \mapsto \{0, \ell\}, n \mapsto \top\}, \varnothing, \{\ell\}\rangle$
$= \langle\{p \mapsto \{0, \ell\}, n \mapsto \top\}, \{\ell\}, \{\ell\}\rangle$

이때 합하기 연산(join operator, $\cup$)을 우리가 관심있는 정보에 민감하도록 바꾸면 경로 고려 분석의 느낌을 살릴 수 있다. 새로운 합하기 연산 $\Cup$의 정의는 다음과 같다.

$$\langle m1, a1, f1\rangle \Cup \langle m2, a2, f2\rangle = \begin{cases} \langle m1, a1, f1\rangle & \text{if } a2 \subseteq a1 \wedge f2 \subseteq f1 \\ \langle m2, a2, f2\rangle & \text{if } a1 \subseteq a2 \wedge f1 \subseteq f2 \\ \langle m1, a1, f1\rangle \cup \langle m2, a2, f2\rangle & \text{otherwise} \end{cases}$$

이 연산은 만약 한 경로에서 우리가 관심있는 메모리 할당/해제가 확실히 더 많이 일어났다면 그 경로만을 택하겠다는 의도를 가지고 있다. 이 방법은 경로 고려 분석을 하지 않는 어떤 분석에든 일반적으로 사용할 수 있는 방법이다. 분석의 목적에 초점을 두고 목적에 부합하는 정보가 많이 일어나는 경로가 전체 경로를 지배하도록 할 수 있다.

- 심볼릭 변수의 도입을 제한하기

함수를 분석하는 도중에 모르는 변수를 만나는 경우 심볼릭 변수를 도입하게 된다. 이때 무제한으로 도입을 하면 함수내에 루프가 있는 경우 계속해서 새로운 심볼릭 변수들을 만들어내게 된다. 그러면 당연히 루프의 고정점을 계산할 수 없게 되고, 분석이 끝이 나지 않는다. 이를 막기 위해 한 프로그램 포인트(program point)에서 도입할 수 있는 심볼릭 변수의 개수를 특정 상수 k로 제한을 한다. 이것의 의미는 어떤 함수가 루프를 돌면서 인자로 주어진 회수만큼 루프를 돌면서 리스트를 할당하는 함수라면 Sparrow는 항상 길이가 k개인 리스트를 만들어 낸다. 그렇게 생성된 리스트를 루프를 돌면서 해제하면 모두 k번을 돌면서 해제하게 된다. 분석중에 루프를 통해 일어나는 일을 더 작게 봄으로써(under-approximation) 분석의 비용이 너무 커지지 않도록 한

다. Sparrow에서 사용하는 상수 k는 5이다.

- 함수의 부가 정보 이용하기

분석의 전반적인 정확도 증가를 위해 함수가 하는 일에 관해 8가지의 함수 요약 카테고리 이외의 정보를 이용한다. 현재 Sparrow에서 사용하는 정보들은 다음과 같다. 1) 함수가 리턴하는 정수 값에 대한 정보: 보다 정확한 값 분석을 위해 2) 함수가 프로그램을 종료시키는 함수인지에 대한 정보: 프로그램을 종료시킨 후에는 모든 힙 메모리가 해제되기 때문에 3) 가변 인자를 받아 들이는 함수인지에 대한 정보: 가변인자에 대해서는 함수 요약을 제대로 만들 수 없고, 따라서, 이 함수를 사용하는 함수도 제대로 분석할 수 없다. 이런 함수가 쓰이면 그 함수의 인자로부터 접근 가능한 모든 주소들이 안전하다고 여기고 분석을 진행한다. 4) 할당된 주소가 널 주소(null address)와의 비교가 끝난 주소인지: 보다 정확한 값 분석을 위해.

- 할당된 주소들의 태생을 함수 요약에서도 구분하기

함수 요약 그래프에 있는 할당된 주소를 보면 모두 서로 다른 주소인 것처럼 분리되어 있다. 그림 6을 보면 모든 할당된 노드들(!으로 표시된 노드들)은 각각이 서로 다른 프로그램 포인트에서 생성되어 있다. 하지만 어떤 함수(그림 12)는 할당된 주소를 리턴하기도 하고, 그 똑같은 주소를 인자에 붙이기도 한다. 그런 경우에 접근 경로만으로 구분을 하면 함수 요약으로 표현했을 때 서로 다른 할당 주소인 것처럼 보인다. 만약 그렇다면 실제로 할당되지도 않은 주소를 할당되었다고 여기는 말도 안되는 허위경보가 발생한다. 따라서 할당된 주소에 대해서는 그 주소가 어느 프로그램 포인트에서 생성된 주소인지를 함수 요약 상에서 표현이 되어야 한다.

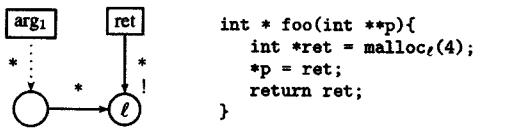


그림 12 인자가 가리키는 주소나 리턴 되는 주소나 모두 같은 곳에서 할당된 동일한 주소이다.

- 고정점 계산과 분석정보 분리하기

프로그램에 있는 루프는 프로그램 분석에 가장 큰 적이다. 루프가 몇번 실행될지 돌려보지 않고서는 미리 알 수 없기 때문이다. 따라서, 실제 실행에서 생길 수 있는 무수히 많은 경로들을 루프안에서 하나의 실행으로 요약한다. 이 과정에서 전통적으로 루프의 불변하는 성질(loop invariant)을 고정점 계산으로 구하게 된다. 이 과정에서 분석의 정확도가 어쩔 수 없이 떨어지게 된다. 메모리 상태중에 할당/해제된 주소들의 집합은 고정점

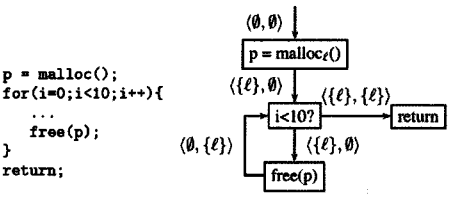


그림 13 간선에 붙어 있는 정보는 할당/해제된 주소들 집합의 쌍이다. 빠져나갈 때 할당된 주소들의 집합에 주소 ℓ이 들어가 있기 때문에 허위경보가 발생한다.

계산을 할 때 처다보지 않음으로써 루프의 실행흐름을 요약할 때 루프의 가장 마지막 실행의 정보를 가지도록 한다. 예를 보면 쉽게 이해할 수 있다. 그림 13에 있는 코드는 명백히 할당된 주소를 해제함에도 불구하고, 루프의 시작점(loop head)에서 루프를 실행하지 않을 때의 상태와 루프를 실행하고 나온 상태를 합치기 때문에 허위경보가 발생한다.

우리는 고정점 계산을 할 때 할당/해제된 주소들의 집합은 처다보지 않기 때문에 루프가 한번이라도 실행될 수 있다면 루프의 시작에서 루프가 실행되지 않을 때의 상태를 합치지 않는다. 오직 참조관계의 맵만을 합친다. 이는 프로그램에 있는 루프는 확실히 실행이 되지 않는 것들은 컴파일 시간(compile time)에 알 수 있고, 잘 알 수 없는 루프는 대부분 한 번 이상 실행될 거라는 믿음의 기초한 선택이다.

4. 메모리 변화로부터 함수 요약기

함수가 하는 일을 요약하기 위해 우선 함수를 분석해야한다. 이 때 입력 메모리에 대해 모르는 값을 참조하는 경우에는 심볼릭 주소를 도입하여 그 주소를 가리키고 있었을 것이라 가정하고 분석을 진행한다. 이 심볼릭 변수들을 입력 메모리에 대한 이미지를 만들어 낸다. 이 이미지로 부터 우리는 함수가 하는 일을 끌어낼 수 있다. 그 일 중에 메모리 누수를 찾는데 의미있는 일들을 함수 요약으로 저장한다.

4.1 요약 도메인

요약 해석 기반 분석을 할 때 사용하는 요약 도메인(abstract domain)을 표 4에 나타내었다. 이 중에 $\widehat{Explore}$가 우리가 사용하는 심볼릭 변수를 나타낸다. 모르는 메모리 상태를 탐색하는 주소라는 의미에서 explore라는 이름을 붙였다. 앞에서 줄곧 사용했던 α가 이 심볼릭 변수이다. 분석 중에 생성되는 심볼릭 주소들은 모두 다르다고 가정한다. 하지만 이런 주소들은 호출 환경에 따라 같은 주소로도 실증화 될 수 있다(그림 2). 심볼릭 주소는 두 개의 정보를 가지고 있다. 이 심볼릭

주소를 값으로 가리키고 있다고 가정하는 주소를 기억하고, 이 심볼릭 주소가 생성된 프로그램 포인트를 기억한다. 이를 이용해 이 심볼릭 주소가 어느 접근 경로를 통해 어디서 생성되었는지를 관리할 수 있다. 프로그램 포인트는 심볼릭 변수의 무제한 도입을 막는데 사용한다.

표 4 고정점 계산에 쓰이는 분석의 요약 도메인

$$
\begin{array}{rcrl}
\widehat{T} & \in & \widehat{Table} & = Block \xrightarrow{\text{fin}} \widehat{Memory} \\
\widehat{m} & \in & \widehat{Memory} & = \widehat{Map} \times \widehat{AllocFree} \\
\widehat{M} & \in & \widehat{Map} & = \widehat{Addr} \xrightarrow{\text{fin}} \widehat{Value} \\
\widehat{L} & \in & \widehat{Address} & = 2^{\widehat{Addr}} \\
\ell & \in & \widehat{Region} & = AllocSite \\
i & \in & PgmPoint & \\
\langle \hat{a}, i \rangle & \in & \widehat{Explore} & = \widehat{Addr} \times PgmPoint \\
\hat{a} & \in & \widehat{Addr} & = GVar + ProcId \times Var \\
 & & & + \widehat{Region} + \widehat{Addr} \times FieldId \\
 & & & + \widehat{Explore} + Null \\
\widehat{V} & \in & \widehat{Value} & = \hat{\mathbb{Z}} + \widehat{Address} \\
 & & \widehat{AllocFree} & = \widehat{Alloc} \times \widehat{Free} \\
\widehat{AL} & \in & \widehat{Alloc} & = 2^{\widehat{Region}} \\
\widehat{FR} & \in & \widehat{Free} & = 2^{\widehat{Region} + \widehat{Explore}}
\end{array}
$$

Sparrow가 사용하는 요약은 분석이 유한 시간내에 항상 종료함을 보장한다. 심볼릭 변수들은 모든 프로그램 포인트마다 k개 이상 생성될 수 없도록 제한이 된다. 만약 그 이상 생성하려고 하면 마지막으로 생성된 주소를 사용한다. 무한히 생성될 수 있는 동적으로 할당되는 주소들은 정적 호출 위치(static call site)가 같으면 모두 같은 주소로 본다. 이 주소들은 상위 레벨에서 함수 호출이 일어날 경우 호출 환경 민감하게(context sensitive) 새로운 이름이 붙는다. 정수값은 구간의 쌍으로 요약하고, 루프에 의해 터지는 값은 축지법(widening)을 사용하여 수렴시킨다.

Block은 대상 프로그램의 모든 기본 블럭(basic block)들이다. GVar과 ProcId X Var은 각각 전역변수와 지역변수들을 나타낸다. FieldId은 모든 구조물(structure)들의 필드 이름들이다. 정수를 요약하는데 쓰이는 $\hat{\mathbb{Z}}$은 구간의 쌍이다. 할당된 주소들의 집합 $\widehat{Alloc}$과 해제된 주소들의 집합 $\widehat{Free}$은 할당/해제된 주소들을 기억한다. 프로그램에서 나타나는 메모리 할당(e.g. malloc)의 의미는 할당된 주소를 에 추가하는 것이고, 메모리 해제(e.g. free)의 의미는 인자가 가리키고 있는 모든 주소를 $\widehat{Alloc}$에서 제거하고, $\widehat{Free}$에 넣는 것이다. 분석의 결과는 모든 기본 블럭에서 그 지점의 메모리 상태로 가는 맵이다. 그 중에 우리는 함수가 끝나는 지점에서의 메모리 상태에 관심이 있다.

그림 14는 분석 결과로 나오는 메모리 상태와 그 것으로 부터 계산되는 함수 요약 카테고리들, 그리고 그것의 그래프 표현을 보여주고 있다. 첫번째 리턴문(return statement)에서의 메모리 상태를 보면 변수 n은 조건문에 의해 가지치기(pruning)되어 그 값은 0이 될 수 없고, 이때 리턴값은 할당된 주소 ℓ이다. 두번째 리턴문에서의 메모리 상태를 보면 변수 n은 0이다. 또, 변수 p를 참조할때 메모리에 그 값이 존재하지 않기 때문에 심볼릭 주소가 도입되었음을 볼 수 있다. 여기서 우리는 p가 심볼릭 주소 <p, i>을 가리키고 있었다고 가정한다. 여기서 i는 두번째 리턴문의 프로그램 포인트이다. 다시 <p, i>·val을 통해 모르는 값이 참조되고, 심볼릭 주소의 새로운 심볼릭 주소 <<p, i>·val, i><<p, i>·val, i>가 생긴다. 그리고, 이 주소가 리턴된다.

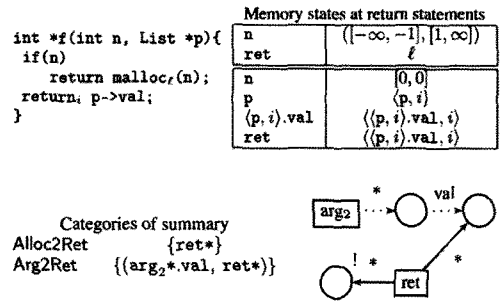


그림 14 예제 함수 f, 함수가 끝나는 지점에서의 메모리 상태들(할당된 주소의 집합은 $\{\ell\}$이다), 함수의 카테고리들, 그리고 함수 요약의 그래프 표현

4.2 함수가 끝나는 지점에서 함수 요약 만들기와 메모리 누수 찾기

함수가 끝나는 지점에서의 메모리 상태로부터 함수를 요약한다. 앞에서 소개된 여덟가지 카테고리를 계산하기 위해서 특정 주소로부터 접근 가능한 모든 주소들을 계산할 필요가 있다. 이때 그 접근 경로를 기억하기 위해 앵커(anchor)[1]라는 개념을 도입하겠다.

$$\psi \in Anchor = (\texttt{ret} \mid \texttt{arg}_i \mid \texttt{global})(* \mid .\texttt{f})^*$$

앵커는 리스트로 표현되는데 첫번째 원소는 시작 주소이다. 심볼릭 주소 argi는 형식 인자(formal parameter)를 나타내고, i는 i번째 인자임을 나타낸다. 심볼 *는 포인터 참조를, .f는 구조물의 필드를 통한 참조를 표현한다. 그림 14에 함수 f의 요약을 나타내었다. 카테고리 Alloc2Ret는 앵커들의 집합이다. 각각의 앵커는 리턴값으로부터 그 앵커를 통해 접근 가능한 주소가 새로 할당된 주소임을 의미한다. 카테고리 Arg2Ret는 두개

1) 끝에 무거운 추가 달린 닻을 연상해보자. 여기서 쓰이는 앵커는 그 추가 닿은 지점을 줄을 따라 좇아가는 것을 표현한다.

앵커 쌍들의 집합이다. 첫번째 앵커는 인자의, 두번째 앵커는 리턴값의 접근 경로를 나타낸다. 인자로부터 첫 번째 앵커를 통해 접근 가능한 주소가 리턴값으로부터 두번째 앵커를 통해 접근 가능한 주소와 앨리어스된다는 것을 의미한다.

이제 실제로 메모리로부터 어떻게 함수 요약을 만들어 내는지 자세히 살펴보자. 여덟가지 카테고리 중에 반은 함수가 끝나는 시점에서의 메모리를 통해 알아낼 수 있지만, 나머지는 함수가 시작될 때의 메모리 상태로 부터 알아낼 수 있다.

- 함수가 끝 지점 메모리 상태로부터 함수 요약으로 우선 몇가지 계산에 필요한 함수들을 정의하자. 주어진 맵 $\hat{M}$으로부터 $\mathrm{reach}_{\hat{M}}\hat{L}$은 $\hat{L}$로부터 접근 가능한 모든 주소들과 그 주소로 도달하기까지의 앵커들의 집합을 계산한다.

$$\begin{array}{rcl}\mathrm{reach}_{\widehat{M}} & : & 2^{\widehat{Addr}} \to 2^{\widehat{Addr}\times Anchor} \\ X, S & \in & 2^{\widehat{Addr}\times Anchor} \\ \mathrm{reach}_{\widehat{M}}\widehat{L} & = & \mathrm{lfp}^{\trianglelefteq}\ \lambda S.X \cup (F\ S)\end{array}$$

위에서 쓰인 X와 F S는 아래와 같다.

$$\begin{array}{l} X = \{(\hat{a},\hat{a}) \mid \hat{a} \in \widehat{L}\} \\ F\ S = \bigcup\{(\hat{a}',\psi*) \mid \hat{a}' \in \widehat{M}(\hat{a}), (\hat{a},\psi) \in S\} \\ \quad\quad \bigcup\{((\hat{a},\mathrm{f}),\psi.\mathrm{f}) \mid (\hat{a},\mathrm{f}) \in dom(\widehat{M}), (\hat{a},\psi) \in S\}\end{array}$$

X는 처음 시작의 주소와 앵커인데, 앵커에서의 시작 주소를 의미한다. F S는 S로부터 바로 접근 가능한 모든 주소들을 찾아준다. 참조하는 방법은 포인터 참조와 필드 참조 두 가지 방법이 있다. reach는 모든 접근 가능한 주소와 그 것의 앵커를 구해서 변하지 않을 때까지 집합을 키워나간다.

집합이 고정점에 도달했는지 검사할 때는 오직 주소만을 쳐다본다. 만약 앵커까지 보면 앵커가 무한히 붙으면서 수렴하지 않고 무한히 커질 수 있다. 함수 addr는 집합 S에서 모든 주소들만을 꺼낸다.

$$\begin{array}{l} S \trianglelefteq S' = \mathbf{addr}\ S \subseteq \mathbf{addr}\ S' \\ \mathbf{addr}\ S = \{\hat{a} \mid (\hat{a},\_) \in S\}\end{array}$$

이제 함수가 끝날 때 우리가 가지고 있는 정보를 정리하자. 메모리 상태는 $(\hat{M}, (\widehat{AL}, \widehat{FR}))$이고, $\widehat{GL}$은 전역변수로부터 접근 가능한 모든 주소, $\widehat{RL}$은 리턴값으로부터 접근가능한 모든 주소라고 하자.

그러면 다음 4가지의 카테고리들을 간단하게 구할 수 있다.

$$\begin{array}{rcl}\mathsf{Glob2Arg} & = & \bigcup_i \mathrm{reach}_{\widehat{M}} \\ \mathsf{Alloc2Arg} & = & \bigcup_i \mathrm{reach}_{\widehat{M}}\end{array}$$

이때 $S \Cap \widehat{L}$는 S에 있는 주소가 $\hat{L}$에도 있을 때 그 주소의 앵커를 꺼낸다.

$$S \Cap \widehat{L} = \{\psi \mid (\hat{a},\psi) \in S, \hat{a} \in \widehat{L}\}$$

카테고리 Glob2Arg은 인자로 부터 도달 가능한 주소들 중에 전역변수로부터 온 주소가 있는지 검사한다. 카테고리 Alloc2Arg는 인자로부터 접근 가능한 모든 주소들 중에 할당된 주소가 있는지 검사한다. 이때 전역변수로부터도 접근 가능한 주소는 생략한다. 카테고리 Alloc2-Ret, Glob2Ret는 Alloc2Arg, Glob2Arg와 각각 비슷한 방법으로 계산 가능하다.

- 입력 메모리 상태로부터 함수 요약으로

다른 네가지 카테고리는 모두 입력 상태로부터만 계산이 가능하다. 모든 카테고리가 Arg2로 시작한다. 따라서, 이 함수가 호출될 때 인자로부터 접근 가능한 주소들이 어떤 구조를 가지고 있었는지를 알아내야한다. 입력 메모리 상태를 정확히 알아내는 것은 불가능하지만 이 함수가 끝날때의 메모리 상태를 보고, 유추하는 것이 가능하다. 분석중에 입력 메모리를 탐색할 상황이 될 때 사용했던 심볼릭 주소들에 의해 유추가 된다.

새로운 함수 Sreach는 심볼릭 주소를 통해 접근 가능한 모든 주소들과 그 것의 앵커를 계산한다. 이 함수는 F S의 포인터 참조대신에 아래의 심볼릭 참조를 사용한다는 사실을 빼고는 앞에서 등장한 reach와 동일하다.

$$F\ S = \bigcup\{(\hat{a}',\psi*) \mid \langle\hat{a},i\rangle \in dom(\widehat{M}), (\hat{a},\psi) \in S\}$$

주어진 주소 $\hat{a}$에 대해 익스플로어 주소 $\langle\hat{a},i\rangle$를 꺼낸다. 이 함수는 입력 메모리 상태를 그려주고, 이로 부터 아래의 카테고리들의 계산이 가능하다.

$$\begin{array}{rcl}\mathsf{Arg2Free} & = & \bigcup_i \mathit{S}\mathrm{reach}_{\widehat{M}}\{\mathbf{arg}_i\} \Cap \widehat{FR} \\ \mathsf{Arg2Glob} & = & \bigcup_i \mathit{S}\mathrm{reach}_{\widehat{M}}\{\mathbf{arg}_i\} \Cap \widehat{GL} \\ \mathsf{Arg2Arg} & = & \bigcup_{ij} \mathbf{common}\ (\mathrm{reach}_{\widehat{M}}\{\mathbf{arg}_j\})\ (\mathit{S}\mathrm{reach}_{\widehat{M}}\{\mathbf{arg}_i\}) \\ \mathsf{Arg2Ret} & = & \mathbf{common}\ (\mathrm{reach}_{\widehat{M}}\{\mathbf{ret}\})\ (\mathit{S}\mathrm{reach}_{\widehat{M}}\{\mathbf{arg}_i\})\end{array}$$

함수 common $S\,S'$은 S와 S' 모두에 속하는 주소의 앵커쌍들을 구한다.

$$\mathbf{common}\ S\ S' = \{(\psi,\psi') \mid (\hat{a},\psi) \in S, (\hat{a},\psi') \in S'\}$$

카테고리 Arg2Free와 Arg2Glob은 각각 인자로부터 접근 가능했던 모든 주소들과 해제된 주소들, 전역변수로부터 접근 가능한 주소들과의 교집합으로부터 계산된다. 나머지 Arg2Arg와 Arg2Ret은 각각 역시 인자로부터 도달가능했던 주소들과 다른 인자들로부터 도달가능해진 주소들, 리턴값으로부터 도달가능한 주소들과의 공통으로 계산된다.

만약 함수가 끝나는 지점이 여러 곳인 경우에는 모든 가능한 카테고리들을 합한다. 예를 들어, 그림 14에서 함수 f는 한 경로에서는 할당된 주소를 리턴하고, 다른

경로에서는 인자로부터 접근 가능한 주소를 리턴한다. 이 때에는 이 함수가 두가지 일을 모두 다한다고 요약한다.

이제 메모리누수를 찾아보자. 할당된 주소들 중에 외부로부터 접근가능하지 않은 주소가 새고 있는 주소다. 외부로부터 접근 가능하려면 전역변수, 인자, 혹은 리턴값으로부터 접근 가능해야 한다.

$$\text{LeakedAddress} = \widehat{AL} - \widehat{GL} - (\text{addr reach}_{\widehat{M}}(\bigcup_i \{\text{arg}_i\} \cup \{\text{ret}\}))$$

누수되는 주소들 LeakedAddress에 속하는 각각의 주소들에 대해 주소가 어디서 할당되었는지와 누수가 발생하는 리턴문이 어디있는지를 보고한다. 이때 함수 요약의 정보를 이용하여 함수 호출 깊이(function call depth)가 아무리 깊어도 어떤 경로로 할당이 되었는지 그 함수 호출 경로를 모두 보여준다. 이로 인해 사용자는 메모리 누수 경보에 대해 참/거짓을 쉽게 판단할 수 있다.

4.3 함수 요약 실증화 하기

함수가 호출되면 호출되는 함수의 요약을 보고 실증화를 하여 메모리 상태를 변화시킨다. 실증화에 필요한 정보는 현재 메모리 상태 $(\widehat{M}, (\widehat{AL}, \widehat{FR}))$와 실제 인자들의 값과 리턴값을 저장할 주소이다. 지면의 한계로 모든 카테고리의 실증화 함수를 설명하지는 못하고, Arg2Free에 대해서만 설명하겠다. Arg2Free에 포함되어 있는 모든 앵커들을 가지고 현재 메모리를 탐색해서 실제로 해제되는 주소들이 무엇인지를 알아내야한다. 이렇게 알아낸 주소들은 할당된 주소들의 집합에서 제거하고, 해제된 주소들의 집합에 추가한다.

$$\widehat{AL}' = \widehat{AL} - \hat{L} \quad \widehat{FR}' = \widehat{FR} \cup \hat{L}$$

$$\text{where } \hat{L} = \bigcup_{\psi \in \text{Arg2Free}} \Psi(\hat{L}_i, \psi, \widehat{M})$$

이때 모든 앵커들의 첫 원소는 arg_i이다 $\hat{L}_i$는 i번째 실제 인자가 가리키는 주소이다. 함수 Ψ는 앵커를 가지고 주어진 주소로부터 메모리를 탐색하는 역할을 한다.

$$\Psi \ : \ \widehat{Address} \times Anchor \times \widehat{Map} \to \widehat{Address}$$

$$\begin{aligned}
\Psi(\hat{L}, \hat{a} :: t, \widehat{M}) &= \Psi(\hat{L}, t, \widehat{M}) \\
\Psi(\hat{L}, * :: t, \widehat{M}) &= \Psi(\{\hat{a}' \mid \hat{a}' \in M(\hat{a}), \hat{a} \in \hat{L}\}, t, \widehat{M}) \\
\Psi(\hat{L}, .\text{f} :: t, \widehat{M}) &= \Psi(\{\langle \hat{a}, \text{f} \rangle \mid \hat{a} \in \hat{L}\}, t, \widehat{M}) \\
\Psi(\hat{L}, \text{nil}, \widehat{M}) &= \hat{L}
\end{aligned}$$

함수 Ψ는 다른 카테고리들을 적용할 때도 쓰인다. 카테고리 Alloc2Arg와 Alloc2Ret는 할당된 주소들의 집합에 새롭게 할당된 주소들을 추가하고, 할당된 주소들을 앵커가 가리키는 곳에 끼워 맞추어 메모리 맵도 변화시킨다. 카테고리 Arg2Arg와 Arg2Ret은 앵커 쌍을 보고 앨리어스되야 하는 주소를 찾아 앨리어스시킨다. 카테고리 Glob2Arg, Arg2Glob과 Glob2Ret은 전역변수와 관계되야 하는 주소들을 찾아 메모리 맵을 변경한다.

5. 실험 결과

이 논문에 있는 분석은 Sparrow로 구현이 되었다. Sparrow는 크게 두가지 엔진이 있어 하나는 버퍼오버런(buffer overrun)을 하나는 이 논문에 나와 있는 방식으로 메모리누수(memory leak)을 찾는다. 여러 오픈 소스 프로그램에 대한 실험결과가 표 5에 나와 있다. 실험은 3.2GHz 펜티엄 4에 4GB 메모리의 리눅스 기계에서 진행하였다.

표 5 SPEC2000 벤치마크와 몇몇 오픈 소스 프로그램에 대한 Sparrow의 성능

Programs	Size KLOC	Time (sec)	Bug Count	False Alarm
ammp	13.2	9.68	20	0
art	1.2	0.68	1	0
bzip2	4.6	1.52	1	0
crafty	19.4	84.32	0	0
equake	1.5	1.03	0	0
gap	59.4	31.03	0	0
gcc	205.8	1330.33	44	1
gzip	7.7	1.56	1	4
mcf	1.9	2.77	0	0
mesa	50.2	43.15	9	0
parser	10.9	15.93	0	0
twolf	19.7	68.80	5	0
vortex	52.6	34.79	0	1
vpr	16.9	7.85	0	9
binutils-2.13.1	909.4	712.09	228	25
openssh-3.5p1	36.7	10.75	18	4
httpd-2.2.2	316.4	74.87	0	0
tar-1.13	49.5	11.73	5	3

- Saturn과의 비교

네가지 오픈 소스 프로그램들(binutils, openssh, httpd, 그리고 tar)을 분석하였다. 처음 두개에 대해서는 다른 분석기들[2,3]과의 비교를 위해 예전 버젼을 사용하였다. 사용된 오픈 소스 프로그램들은 여러가지 방식으로 컴파일되어 바이너리(binary)나 라이브러리를 만들 수 있는데, 표 5에서는 그 중 가장 많은 경보가 발생한 타겟을 선정했다.

Saturn은 openssh 프로그램에서 29개의 버그를 찾았지만 Sparrow는 18개 밖에 찾지 못했다. 위에서 언급했듯이(3장), Sparrow는 경로를 고려하지 않음으로써 몇몇 버그들을 놓치고 있다. 경로를 고려하는 것이 openssh를 분석하는데에는 중요한 역할을 한다. 할당된 주소를 어떤 경로에서는 전역변수에 붙이고, 다른 경로에서는 붙이지 않는 경우 Sparrow는 안전하다고 판단하기 때문에 누수를 찾지 못한다.

반면 Sparrow는 binutils 프로그램에서 228개의 버그

를 찾고 세턴은 136개만을 찾는다. 이는 Saturn이 사용하는 함수의 요약 정보는 인자에 할당된 주소가 붙는 정보를 표현하지 못하기 때문이다. binutils 프로그램에서는 많은 메모리 할당이 인자를 통해 이루어진다(Alloc2Arg). 인자에 할당하는 함수의 개수는 491인 반면 할당된 주소를 리턴하는 함수는 160개에 불과했다. 또, Saturn의 함수요약은 리턴값 자체가 할당되는 주소라는 사실만을 알뿐이고, 할당되어 리턴되는 구조물의 형태가 어떠한지는 알지 못한다(Alloc2Ret). 따라서, 그림 16과 같은 메모리 누수는 찾아내지 못한다.

- FastCheck과의 비교

```
261: osmesa = (OSMesaContext) calloc( 1, sizeof( ...
262: if (osmesa) {
263:   osmesa->gl_visual = gl_create_visual( rgbmode, ..
272:   if (!osmesa->gl_visual) {
273:     return NULL;
274:   }
276:   osmesa->gl_ctx = gl_create_context( ...
279:   if (!osmesa->gl_ctx) {
280:     gl_destroy_visual( osmesa->gl_visual );
281:     free(osmesa);
282:     return NULL;
283:   }
284:   osmesa->gl_buffer = gl_create_framebuffer( ...
285:   if (!osmesa->gl_buffer) {
286:     gl_destroy_visual( osmesa->gl_visual );
287:     gl_destroy_context( osmesa->gl_ctx );
288:     free(osmesa);
289:     return NULL;
290:   }
```

그림 15 SPEC2000 벤취마크 프로그램 중에 하나인 mesa 프로그램에서 발견된 메모리 누수 버그

FastCheck[1]과의 비교를 위해 같은 SPEC2000 벤취마크에 대해 Sparrow를 적용하였다. Sparrow는 96개의 경보 중에 81개의 버그를 찾았고, FastCheck은 67개의 경보 중에 59개의 버그를 찾았다. Sparrow와 FastCheck의 경보들을 일일이 대조한 결과 Sparrow는 gcc 프로그램에서 나온 2개의 버그를 제외하고는 FastCheck이 찾은 모든 버그들을 찾는다.

그림 15는 mesa 프로그램에서 Sparrow가 찾아낸 두 개의 메모리 누수를 보여준다. 코드의 261~263줄을 분석하면 포인터 osmesa는 할당된 구조체를 가리키고, osmesa->gl_visual은 다른 할당된 구조체를 가리킨다. Sparrow는 함수 gl_create_visual이 할당된 구조체를 리턴한다는 사실을 함수 요약으로 가지고 있다. 그런데, 만약 그 함수가 널 포인터를 리턴한다면 현재 함수도 273줄에서 널 포인터를 리턴한다. 이때 261줄에서 calloc을 통해 할당되어 포인터 osmesa가 가리키고 있는 주소는 해제 되지 않는다. 따라서, 메모리 누수라고 Sparrow가 경보를 낸다. 한편 282줄에서는 Sparrow가 경보를 내지 않는데, 이는 261, 263줄에서 할당된 구조체가 280, 281줄에서 안전하게 해제되기 때문이다. 289줄에서는 함수 gl_create_context를 통해 276줄에서 할당된 주소들 중 몇개가 누수되고 있다고 Sparrow가 경보를 낸다. 일견에 이 경보는 함수 gl_destroy_context가 287줄에서 불리기 때문에 허위경보로 보인다. 하지만 실제로 메모리 누수이다. 우리가 분석한 함수 요약에 의하면 함수 gl_create_context에 의해 할당된 주소들 중에 함수 gl_destroy_context에 의해 해제되지 않는 주소가 존재한다. FastCheck은 이 메모리 누수를 찾지 못한다.

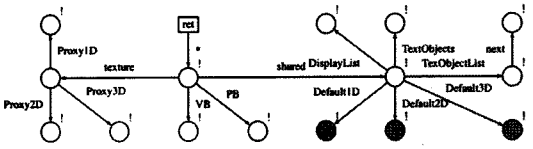


그림 16 함수 gl_create_context의 함수 요약의 그래프 표현. 함수 gl_destroy_context에 의해 해제되지 않는 노드들만이 어둡게 칠해져있다.

그림 16에 할당하는 함수의 함수 요약을 표현하고, 그 중에 해제하는 함수에 의해 해제되지 않는 주소들을 표현했다. 우리의 함수 요약은 이와 같이 구조체의 모양을 표현할 수 있다. SPEC2000 벤취마크에서 Sparrow의 허위경보들은 다음과 같은 이유로 발생한다. 1) 분석의 조건문 가지치기 함수의 한계(gcc) 2) 경로를 고려하지 않는 분석(gzip) 3) 루프의 인해 발생하는 분석의 부정확성(vortex) 4) 2차원 배열을 과도하게 요약해서(vpr).

6. 관련 연구

Whaley와 Rinard는 자바(Java)를 위한 컴포지셔날 포인터와 탈출 분석(compositional pointer and escape analysis)를 제안하였다[10]. 이 분석도 매개화된 탈출 그래프를 통해 어떤 메소드(method)에서 어떤 메모리 블럭이 안전하게 탈출하고 있는지를 저장한다. 이 정보는 우리의 함수 요약과 비슷하지만, 메모리 누수를 찾기 위해서는 더 많은 정보들이 필요하다.

Heine와 Lam[3][11]은 흐름과 호출 환경에 민감한 C와 C++에서 메모리 누수를 찾을 수 있는 분석기를 고안했다. 그들은 소유 관계 모델(ownership model)을 이용하여 메모리를 관리하는 시스템을 만들고, 그 시스템을 검증하는 일종에 타입 시스템을 이용하여 메모리 누

수가 없음을 증명하려 한다. 소유 관계 모델은 모든 객체들은 자신을 소유하는 포인터가 유일하게 하나 있다고 가정하는 모델이다. 소유하는 포인터는 항상 자신이 소유하는 객체를 해제하거나 다른 포인터에게 소유권을 넘겨야만 하는 제약을 따른다. 이런 제약 조건으로부터 입력 프로그램을 검사하여 그 제약 조건들을 모두 만족할 수가 없다면 그 프로그램에는 메모리 누수나 중복 해제(double deletion)가 존재한다고 판단한다. 그들의 분석은 우리 분석보다 훨씬 많은 허위경보를 발생시킨다. 또, 제약 조건의 모순이 어째서 생기는지 정확한 위치를 찾아주지 못하기 때문에 사용자에게 정확한 메모리 누수 위치와 원인을 제공하기 어렵다.

Xie와 Aiken[2,12]은 Saturn에 기반한 메모리 누수 탐지기를 만들었다. Saturn은 입력 프로그램을 불리안 식(boolean formulas)으로 변환하여 경로를 고려하는 분석을 한다. 메모리 누수 탐지 문제는 이 불리안 식들을 만족시킬 수 있는지 여부의 문제가 된다. 그들의 분석은 호출 환경과 경로를 고려하지만 루프와 순환 호출을 제대로 분석하지 못한다. 그리고, 경로 고려 분석을 하다보니 우리의 분석기보다 분석 속도가 느리다.

*Orlovich*와 *Rugina*[4]는 메모리 누수가 있다는 가정으로부터 거꾸로 프로그램을 분석하여 그 가정이 잘못되었다는 증명을 시도함으로써 프로그램에 누수가 없음을 보이려했다. 이 과정에서 거꾸로 프로그램을 분석해서 올라가는 과정이 끝나지 않을 수 있기 때문에 과정의 길이에 제한을 두게 된다. 이로 인해 SPEC2000 벤치마크 프로그램에서 Sparrow가 찾는 버그보다 적은 버그를 보다 높은 허위경보로 찾게 된다.

최근에 Rugina등은 FastCheck[1]이라는 조건이 마크된 값의 흐름(guarded value-flow) 분석을 이용하여 메모리 누수를 찾는 분석기를 제안했다. 메모리 누수 찾는 문제를 발생점과 소멸점(source-sink)관계로 모델링한다. 그 후 프로그램을 도달 정의(reaching definition)과 분기의 조건문을 이용하여 조건이 마크된 값의 흐름으로 단순화 한다. 그들의 분석은 매우 빠르지만 부가적인 영역 분석(region analysis)가 필요하다. 그들의 분석은 허위경보율이 작은 편이다. 하지만 Sparrow가 SPEC2000 벤치마크에서 비슷한 허위경보율(2~3% 차이)로 더 많은 버그를 찾아낸다.

7. 경로고려(Path-sensitive) 분석기로의 확장

Sparrow가 더욱 정확한 분석을 하기 위해서는 경로 고려를 분석에 추가하는 것이 가장 효과적[13-15]이다. 따라서 이 장에서는 어떻게 Sparrow를 경로 고려 분석이 가능하도록 할지 그 설계 방법에 대해 설명하겠다.

7.1 분석을 위한 언어

$$
\begin{array}{rcrcl}
cmd & \in & \mathbf{Cmd} & ::= & \mathtt{assert}\ grd \mid le := e \mid \mathtt{call}(le,e,e) \\
 & & & \mid & \mathtt{return}\ e \mid \mathtt{Alloc}\ le \mid \mathtt{Free}\ e \\
le & \in & \mathbf{LExp} & ::= & \mathtt{x} \mid *e \mid e.\mathtt{f} \\
e & \in & \mathbf{Exp} & ::= & c \mid le \mid e \oplus e \mid \neg e \mid \&le \\
grd & \in & \mathbf{Guard} & ::= & e \sim e \mid \neg grd \\
\diamond & \in & \mathbf{LOpeartor} & ::= & \wedge \mid \vee \\
\sim & \in & \mathbf{Relation} & ::= & = \mid \neq \mid > \mid < \mid \geq \mid \leq \mid \diamond \\
\oplus & \in & \mathbf{BOperator} & ::= & + \mid - \mid * \mid \div \mid \% \mid \sim
\end{array}
$$

프로그램 언어 C에는 다양한 구조물들이 있다. 이는 사용자의 편의를 위해 사용되는 것들이다. 이런 다양한 구조물들은 위와 같이 정제된 언어로 모두 표현할 수 있다. Sparrow는 복잡한 C 프로그램을 받아 들여 프로그램을 분석하기 용이한 형태로 변환한다. 우리는 C 프로그램의 식들을 메모리 변화를 일으키는 Cmd와 값을 계산하기만 하는 Exp로 나누어 분석 디자인을 간결하게 하였다. 또, 프로그램에 존재하는 분기문에 있는 조건문을 특별하게 취급함으로써 경로 분석에 이용할 수 있도록 하였다.

7.2 경로 고려 요약 도메인

분석이 하는 일은 함수 내의 모든 프로그램 포인트에서 프로그램이 실행 중에 가질 수 있는 상태의 요약인 State를 구하는 것이다. State는 다시 현재 프로그램 포인트로 도달하기 위해 만족해야 하는 식 Assert와 현재 메모리의 상태 Memory, 그리고 할당된 주소와 해제된 주소를 기억하는 Minfo로 구성된다. 경로를 고려하기 위해 Memory내에 있는 값들은 모두 그 값들을 갖기 위한 조건 Guard를 포함하고 있다. 또, 메모리가 할당될 때의 조건과 해제될 때의 조건을 기억함으로써 보다 정교한 정보를 표현할 수 있다.

$$
\begin{array}{rcrcl}
\mathcal{M} & \in & \mathit{Memory} & = & \mathit{Addr} \xrightarrow{\mathrm{fin}} \mathit{GuardedValues} \\
gv & \in & \mathit{GuardedValues} & = & \mathit{Value} \xrightarrow{\mathrm{fin}} \mathit{Guard} \\
g & \in & \mathit{Guard} & = & (\mathit{Value} \times \mathit{Relation} \times \mathit{Value}) + (\mathit{Guard} \times \mathbf{LOperator} \times \mathit{Guard}) \\
 & & & & + \neg \mathit{Guard} + \mathtt{true} + \mathtt{false} \\
v & \in & \mathit{Value} & = & \mathcal{N} + (\mathit{Value} \times \mathbf{BOperator} \times \mathit{Value}) + \mathit{Addr} + \top \\
l & \in & \mathit{Addr} & = & \mathit{Var} + \mathit{Explore} + \mathit{AllocSite} + (\mathit{Addr} \times \mathit{Field}) + \mathit{Ret} + \mathtt{Null} \\
 & & \mathit{Explore} & = & \mathit{Addr} \\
 & & \mathit{Ret} & = & \mathit{FunctionName} \\
\mathcal{S} & \in & \mathit{State} & = & \mathit{Assert} \times \mathit{Memory} \times \mathit{Minfo} \\
ast & \in & \mathit{Assert} & = & \mathit{Guard} \\
 & & \mathit{Minfo} & = & \mathit{Alloc} \times \mathit{Free} \\
al & \in & \mathit{Alloc} & = & \mathit{Addr} \xrightarrow{\mathrm{fin}} \mathit{Guard} \\
fr & \in & \mathit{Free} & = & \mathit{Addr} \xrightarrow{\mathrm{fin}} \mathit{Guard}
\end{array}
$$

Guard는 값들 간의 관계나 다른 Guard와의 논리 관계로 표현된다. Assert는 현재 상태에 도달하기 위한 조건을 기억하는데 이는 경로가 갈라졌다가 만나는 지점에서 여러 경로에서 오는 값들의 Guard를 결정하는 역할을 한다. 게다가 현재 경로가 실제로 실행가능한지 검사할 수 있도록 해준다. 만약 Assert가 어떠한 경우에도 거짓이라면 전체 상태를 bottom(아무 것도 아닌 상태)로 간주할 수 있다.

7.3 프로그램의 실행 의미

각 프로그램 식들의 의미는 State를 변화시키는 함수로 표현된다. State는 그 위치에서 우리가 프로그램으로부터 필요로 하는 모든 정보를 담고 있다. 다음은 각 프

로그램 식들의 의미를 정의이다. State의 구성요소들 중에 변하지 않는 것들은 간결함을 유지하기 위해 의도적으로 표기하지 않았다.

$\boxed{S \vdash cmd : S'}$

$$\frac{\mathcal{M} \vdash grd : g', \mathcal{M}'}{g, \mathcal{M} \vdash \texttt{assert}\ \ grd : g \wedge g', \mathcal{M}'}$$

$$\frac{\mathcal{M} \vdash_l le : gv_l, \mathcal{M}_1 \quad \mathcal{M}_1 \vdash e : gv, \mathcal{M}_2 \quad \mathcal{M}_2 \vdash \texttt{bind}\ gv_l\ gv : \mathcal{M}'}{\mathcal{M} \vdash le := e : \mathcal{M}'}$$

$$\frac{\begin{array}{c}\mathcal{M} \vdash_l le : gv_l, \mathcal{M}_1 \quad \mathcal{M}_1 \vdash e_1 : gv_1, \mathcal{M}_2 \quad \mathcal{M}_2 \vdash e_2 : gv_2, \mathcal{M}_3 \\ g, \mathcal{M}_3, al, fr \vdash \texttt{inst}\ gv_l\ gv_1\ gv_2 : g', \mathcal{M}', al', fr'\end{array}}{g, \mathcal{M}, al, fr \vdash \texttt{call}(le, e_1, e_2) : g', \mathcal{M}', al', fr'}$$

$$\frac{\mathcal{M} \vdash_l le : gv_l, \mathcal{M}' \quad \mathcal{M}' \vdash \texttt{bind}\ gv_l\ \{\ell \mapsto \texttt{Success}_\ell,\ \texttt{Null} \mapsto \texttt{Fail}_\ell\} : \mathcal{M}''}{\mathcal{M}, al, fr \vdash \texttt{alloc}_\ell\ le : \mathcal{M}'', al\{\ell \mapsto \texttt{Success}_\ell \vee al(\ell)\}, fr\{\ell \mapsto \texttt{false}\}}$$

$$\frac{\mathcal{M} \vdash e : gv, \mathcal{M}' \quad gv = \{l_i \mapsto g_i\}_i}{\mathcal{M}, al, fr \vdash \texttt{free}\ e : \mathcal{M}', al\{l_i \mapsto al(l_i) \wedge \neg g_i\}_i, fr\{l_i \mapsto g_i \vee fr(l_i)\}_i}$$

$$\frac{\mathcal{M} \vdash e : gv, \mathcal{M}'}{\mathcal{M} \vdash \texttt{return}\ e : \mathcal{M}'\{\texttt{ret}_f \mapsto gv\}}\ f \text{ is current function}$$

assert문은 현재 위치에 와야 할 조건에 인자의 조건을 추가한다. 함수의 호출은 불리는 함수, 리턴값을 저장할 주소, 그리고 인자들을 계산한 후 함수 요약에 따라 실증화를 한다. alloc함수는 프로그램 실행 중에 상황에 따라 새로운 메모리를 할당하는 데에 실패하거나 성공할 수 있으므로 이 두가지 상황을 모두 포괄할 수 있는 GuardedValue를 만들어서 표현하게 된다. 이때 할당되는 주소가 해제된 주소로 이미 표기 되어 있었다면 그 조건을 거짓으로 만든다. free함수는 현재 해제되는 주소를 가리킬 조건을 이용하여 그 주소가 여전히 할당된 상태로 남아있을 조건을 강화하고, 해제될 조건을 더욱 넓혀준다. return문은 경로 고려를 하지 않는 분석과 마찬가지로 이 함수의 리턴값을 저장할 주소에 리턴값을 저장한다.

$\boxed{\mathcal{M} \vdash \texttt{bind}\ gv_l\ gv : \mathcal{M}'}$

$$\frac{gv_l = \{l_i \mapsto g_i\}_i \quad gv = \{v_j \mapsto g_j\}_j}{\texttt{bind}\ \mathcal{M}\ gv_l\ gv : \mathcal{M}\{l_i \mapsto \{v_j \mapsto g_i \wedge g_j\}_j\}_i}\ \texttt{strong\_update}$$

값을 저장하는데 사용된 bind 함수는 위와 같이 정의된다. 이는 강한 변경(strong update)이 가능한 조건일 때의 정의이다. 어떤 주소를 가리킬 수 있을 조건과 어떤 값을 가질 수 있을 조건을 결합하여 새로운 메모리를 만들어 낸다.

$\boxed{\mathcal{M} \vdash e : gv, \mathcal{M}'}$

$$\frac{}{\mathcal{M} \vdash c : \{c \mapsto \texttt{true}\}, \mathcal{M}}$$

$$\frac{\mathcal{M}(\mathbf{x}) = gv}{\mathcal{M} \vdash \mathbf{x} : gv, \mathcal{M}}$$

$$\frac{\mathcal{M} \vdash e_1 : gv_1, \mathcal{M}' \quad \mathcal{M}' \vdash e_2 : gv_2, \mathcal{M}''}{\mathcal{M} \vdash e_1 \oplus e_2 : gv_1 \hat{\oplus} gv_2, \mathcal{M}''}$$

$$\frac{\mathcal{M} \vdash_l le : gv, \mathcal{M}'}{\mathcal{M} \vdash \&le : gv, \mathcal{M}'}$$

$$\frac{\mathcal{M} \vdash_l le : gv_l, \mathcal{M}' \quad \mathcal{M}' \vdash \texttt{lookup}\ gv_l : gv, \mathcal{M}''}{\mathcal{M} \vdash le : gv, \mathcal{M}''}$$

$\boxed{\mathcal{M} \vdash e : gv, \mathcal{M}'}$

$$\frac{}{\mathcal{M} \vdash c : \{c \mapsto \texttt{true}\}, \mathcal{M}}$$

$$\frac{\mathcal{M}(\mathbf{x}) = gv}{\mathcal{M} \vdash \mathbf{x} : gv, \mathcal{M}}$$

$$\frac{\mathcal{M} \vdash e_1 : gv_1, \mathcal{M}' \quad \mathcal{M}' \vdash e_2 : gv_2, \mathcal{M}''}{\mathcal{M} \vdash e_1 \oplus e_2 : gv_1 \hat{\oplus} gv_2, \mathcal{M}''}$$

$$\frac{\mathcal{M} \vdash_l le : gv, \mathcal{M}'}{\mathcal{M} \vdash \&le : gv, \mathcal{M}'}$$

$$\frac{\mathcal{M} \vdash_l le : gv_l, \mathcal{M}' \quad \mathcal{M}' \vdash \texttt{lookup}\ gv_l : gv, \mathcal{M}''}{\mathcal{M} \vdash le : gv, \mathcal{M}''}$$

값을 만들어 내는 식은 위와 같이 계산된다.

$\boxed{\mathcal{M} \vdash \texttt{lookup}\ gv : gv', \mathcal{M}'}$

$$\frac{gv = \{l_i \mapsto g_i\}_i \quad \mathcal{M}(l_i) = \{v_j \mapsto g_j\}_j \quad gv' = \{v_j \mapsto g_i \wedge g_j\}_j}{\mathcal{M} \vdash \texttt{lookup}\ gv : gv', \mathcal{M}}\ l_i \in dom(\mathcal{M})$$

$$\frac{gv = \{l_i \mapsto g_i\}_i \quad gv' = \{\texttt{explore}\ l_i \mapsto g_i\}_i}{\mathcal{M} \vdash \texttt{lookup}\ gv : gv', \mathcal{M}\{l_i \mapsto gv'\}_i}\ l_i \notin dom(\mathcal{M})$$

위와 같이 어떤 주소를 메모리에서 꺼내려 하는데 만약 그 주소가 메모리에 존재하지 않는 경우에는 symbolic 주소를 만들어서 메모리의 상태를 변화시키고 그 값을 돌려준다. 만약 존재한다면 그 값을 가질 조건을 결합하여 그 값을 돌려준다. 주소를 가리킬 조건과 그 주소가 어떤 값을 가질 조건을 결합한다.

$\boxed{\mathcal{M} \vdash_l le : gv, \mathcal{M}'}$

$$\frac{}{\mathcal{M} \vdash_l \{\mathbf{x} \mapsto \texttt{true}\}, \mathcal{M}}$$

$$\frac{\mathcal{M} \vdash e : gv, \mathcal{M}'}{\mathcal{M} \vdash_l *e : gv, \mathcal{M}'}$$

$$\frac{\mathcal{M} \vdash e : gv, \mathcal{M}'}{\mathcal{M} \vdash_l e.\mathbf{f} : \texttt{field}\ gv\ \mathbf{f}, \mathcal{M}'}$$

주소값은 위와 같이 계산한다. 이때 field함수는 gv에 있는 모든 값들과 조건들에 대해 field를 access한 값들과 조건들을 결합하여 돌려준다.

$\boxed{\mathcal{M} \vdash grd : g, \mathcal{M}'}$

$$\frac{\mathcal{M} \vdash grd : g}{\mathcal{M} \vdash \neg grd : \neg g}$$

$$\frac{\mathcal{M} \vdash e_1 : gv_1, \mathcal{M}' \quad \mathcal{M}' \vdash e_2 : gv_2, \mathcal{M}''}{\mathcal{M} \vdash e_1 \sim e_2 : gv_1 \hat{\sim} gv_2, \mathcal{M}''}$$

조건문을 계산하면 나오는 GuardedValue들의 관계로부터 새로운 guard를 만들어 낸다.

$\boxed{\bigsqcup_i S_i = S}$

$$\bigsqcup_i \langle g_i, \mathcal{M}_i, al_i, fr_i \rangle = \langle \vee_i g_i, \sqcup_i \mathcal{M}_i, al', fr' \rangle$$

where $\mathcal{M}_1 \sqcup \mathcal{M}_2 = \forall l \in dom(\mathcal{M}_1) \cup dom(\mathcal{M}_2).\ \{l \mapsto \texttt{addgrd}(g_1, \mathcal{M}_1(l)) \sqcup_{gv} \texttt{addgrd}(g_2, \mathcal{M}_2(l))\}$

where $\{v_i \mapsto g_i\}_i \sqcup_{gv} gv = gv\{v_i \mapsto g_i \vee gv(v_i)\}_i$

$\boxed{\texttt{addgrd}\ g\ gv = gv'}$

$$\texttt{addgrd}\ g\ \{v_i \mapsto g_i\} = \{v_i \mapsto g \wedge g_i\}$$

여러 경로들이 합쳐지는 곳에서는 각 경로에 있는 State들을 모두 결합한다. 이때 State에 있는 assert 조건을 각 메모리에 있는 값들의 조건들을 변경하는데 사용한다. 이를 통해 모든 값들이 어떤 경로를 통해 흘러 들어 모이게 되었는지 그 역사를 계속 기록해 나갈 수 있다.

7.4 확장된 함수 요약

경로를 고려해서 분석을 해나갈 수 있다면 보다 정교한 메모리 누수를 찾을 수 있고, 더 나아가 널 참조(null dereference)나 해제된 메모리 사용(use after free)같이 허위경보를 피하기 위해 경로 고려 분석이 필수적인 오류들도 높은 정확도로 찾아 낼 수 있다. 자연히 이런 분석을 지원하기 위해서는 함수 요약도 확장을 해야한다. 예를 들어 어떤 함수가 널 주소를 리턴할 수 있는지, 혹은 인자나 전역 변수에 붙이는지를 아는 것은 보다 많은 널 참조 버그를 찾아내는데 필수적이다. 또, 함수가 인자나 전역변수를 접근한다면 그 것을 미리 기록해 놓아야 나중에 널 참조나 해제된 메모리 사용오류를 찾아내는 데에 쓸 수 있다.

표 6 확장된 함수 요약 카테고리들

categories	free	global	argument	return
allocation		Alloc2Glob	Alloc2Arg	Alloc2Ret
global	Glob2Free	Glob2Glob	Glob2Arg	Glob2Ret
argument	Arg2Free	Arg2Glob	Arg2Arg	Arg2Ret
null		Null2Glob	Null2Arg	Null2Ret
dereferences		GlobDeref	ArgDeref	

이 함수 요약들은 모두 조건을 포함하고 있어야 한다. 예를 들어 아래와 같은 프로그램을 분석하면 함수 요약도 조건을 표현하고 있는 형태가 된다. 예를 들어 아래와 같은 함수를 분석하면 어떤 결과가 나오는지 보자.

```
int * foo(int n, int *p){
   if(n > 0) free(p);
   if(n < 10) return 0;
   return alloc();
}
```

위의 함수는 첫번째 인자 n이 0보다 크면 두번째 인자 p를 해제한다. 또, n이 10보다 작으면 널을 리턴하고, 아니면 할당된 주소를 리턴한다. 따라서, 이 함수가 하는 일을 인자에 따라 다르게 동작하는 함수 요약으로 표현할 수 있다.

```
arg1 > 0 ∧ arg1 < 10   free arg2*, null ret*
arg1 ≥ 10              free arg2*, alloc ret*
arg1 ≤ 0               null ret*
```

이런 식으로 함수를 요약하면 프로그램 전체의 실행 흐름을 모두 좇아가며 분석하는 것이 가능하다. 여기서 가장 큰 문제점은 얼마나 적은 비용으로 분석이 가능할 것이라는 점이다. 경로를 모두 고려하기 때문에 비용이 커지는 대신 우리의 분석 방법은 함수를 한번만 분석하고, 메모리 누수에 관련된 정보만을 뽑아내기 때문에 비용이 줄어드는 측면이 크다. 하지만, 여전히 분석 중에 고려해야 하는 프로그램의 경로의 개수가 지수 스케일로 커지는 것을 막아야 하는 문제는 있다. 이는 구현을 통해 많은 엔지니어링으로 해결해야 하는 문제이다. 경로 분석에서 가장 큰 걸림돌이 되는 것이 루프(loop)이다. 값들을 심볼들로 계산하고 있는 상황에서 다음과 같은 프로그램이 가지는 값은 자연히 고정점에 도달할 수 없다.

while(k < n){ k++; }

위에서 k, n 모두 함수에 인자라고 가정해보자. 그렇다면 분석을 할 때는 (함수가 호출되는 상황을 알지 못한다) 당연히 루프가 몇번이 실행되는지 알 방법이 없다. 따라서, k의 값은 k, k + 1, k + 2, …와 같이 루프 내에서 반복해서 변하게 된다. 이때 n보다 작아야 한다는 조건은 아무런 도움도 주지 못한다. 우리는 손쉽게 요약 해석에서 사용하는 넓히기(widening)기법을 사용하여 k가 루프 안에서 감당할 수 없이 변하면 모든 값을 가질 수 있다는 top 값을 갖도록 강제할 수 있다. 하지만, 이는 분석의 정확도를 회복할 수 없다는 한계를 갖는다.

경로 고려 분석기를 설계하는 일을 비교적 쉽게 할 수 있으나 이를 또 다시 쓸모있는 것으로 만드는 일은 결코 쉽지 않다.

8. 결 론

실용적인 메모리 누수 탐지기인 Sparrow의 분석 방법을 제안하고 구현하였다. 다른 메모리 누수 탐지 도구[1-4]와의 비교를 보면 Sparrow가 같은 프로그램에 대해 더 많은 버그를 찾으면서 속도나 정확도도 실용적인 수준임을 알 수 있다. Sparrow는 함수들을 분석하여 하는 일을 요약하여 함수가 호출되는 곳에서 사용하기에 함수를 한번씩만 분석한다. 함수 요약은 잘 매개화 되어 함수 호출 환경에 맞게 실증화 된다. 메모리 누수 탐지를 위해 정리한 여덟가지 카테고리는 효율적인 분석을 가능하게 한다. 분석기를 만드는 과정에서 발생하는 선택지중 효과적인 선택사항들을 보고하였다. 이후에 경로를 고려하는 분석에 대한 설계를 제시하였다.

참 고 문 헌

[1] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina, Practical Memory Leak Detection Using Guarded Value-flow Analysis, SIGPLAN 2007.

[2] Yichen Xie and Alex Aiken, Context- and Path-sensitive Memory Leak Detection, In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pp.115-125, New York, NY, USA, 2005. ACM.

[3] David, L. Heine and Monica, S. Lam, A Practical Flow-sensitive and Context-sensitive C and C++ Memory Leak Detector, In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Lan-*

guage Design and Implementation, pp.168-181, 2003.

[4] M. Orlovich and R. Rugina, Memory Leak Analysis by Contradiction, In *SAS 2006: 13th Annual International Static Analysis Symposium*, Lecture Notes in Computer Science, Springer, 2006.

[5] Patrick Cousot and Radhia Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, In *Proceddings of ACM Symposium on Principles of Programming Languages*, pp.238-252, January 1977.

[6] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Mine, David Monniaux, and Xavier Rival, A static analyzer for large safety-critical software, In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pp.196-207, New York, NY, USA, 2003. ACM Press.

[7] J. Reynolds, Separation Logic: A Logic for Shared Mutable Data Structures, In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[8] Cristiano Calcagno, Dino Distefano, Peter O'hearn, and Hongseok Yang, Footprint Analysis: A Shape Analysis That Discovers Preconditions, In *SAS 2007: 14th Annual International Static Analysis Symposium, Lecture Notes in Computer Science*, Springer, 2007.

[9] Erick M. Nystrom, H.-S. Kim, and Wen mei W. Hwu, Bottom-up and Top-down Context-sensitive Summary-based Pointer Analysis, In the *proceeding of the 11th Annual International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer, 2006.

[10] John Whaley and Martin Rinard, Compositional Pointer and Escape Analysis for Java Programs, In *Proceddings of ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp.187-206, 1999.

[11] David L. Heine and Monica S. Lam, Static Detection of Leaks in Polymorphic Containers. In *ICSE'06: Proceeding of the 28th international conference on Software Engineering*, pp.252-261, New York, ACM Press.

[12] Yichen Xie and Alex Aiken, Scalable Error Detection Using Boolean Satisfiability, In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.351-363, New York, NY, USA, 2005. ACM.

[13] M. Das, S. Lerner, and M. Seigle, ESP: Path-sensitive program verification in polynomial time, In *Proc. Conference on Programming Language Design and Implementation*, pp.57-68, 2002.

[14] Manuvir Das, Sorin Lerner, and Mark Seigle, ESP: Path-sensitive Program Verification in Polynomial Time, In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp.57-68, June 2002.

[15] David Evans, Static Detection of Dynamic Memory Errors, In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pp.44-53, May 1996.

정 영 범

1998년~2004년 서울대학교 학사. 2004년~서울대학교 석박사 통합과정. 관심분야는 프로그램 분석 및 검증

이 광 근

1983년~1987년 서울대학교 학사. 1988년~1990년 Univ. of Illinois at Urbana-Champaign 석사. 1990년~1993년 Univ. of Illinois at Urbana-Champaign 박사. 1993년~1995년 Bell Labs., Murray Hill 연구원. 1995년~2003년 한국과학기술원 교수. 2003년~현재 서울대학교 교수. 관심분야는 프로그래밍 언어, 프로그램 분석 및 검증