스패로우: 소스 코드 분석기 [1]

오학주, 정영범, 진민식, 김덕환, 황의권, 박대준^{O}, 이희종, 공순호, 이광근
서울대학교 컴퓨터공학부 프로그래밍 연구실
{pronto, dreameye, msjin, dk, neo, pudrife, ihji, soon, kwang}@ropas.snu.ac.kr

Sparrow: The Source Code Analyzer

Hakjoo Oh, Youngbum Jung, Minsik Jin, Deokhwan Kim, Yikwon Hwang, Daejun Park^{O}, Heejong Lee, Soonho Kong, Kwangkeun Yi
Programming Research Laboratory, Seoul National University

Sparrow(스패로우)는 C/C++로 작성된 프로그램의 메모리 누수와 버퍼오버런 오류를 자동으로 프로그램 실행전에 찾아주는 도구이다. Sparrow는 실제 현장에서 쓰이는 상용 프로그램의 분석에 유용하게 쓰일 수 있도록 제작되었다. 이를 위해 분석기 엔진은 수십만 라인 이상의 소프트웨어를 적절한 시간과 정확도로 분석하기 할 수 있도록 많은 기술이 적용되었다. 또한 사용자의 편의성을 위해 대상 프로그램의 소스 구조를 자동으로 파악하여 분석을 준비하고, 거짓일 가능성이 높은 알람을 미리 제거하며, 발생한 알람의 원인을 설명해 준다. Sparrow로 httpd-2.2.2 를 비롯한 오픈 소프트웨어들을 분석한 결과 실제 오류들을 찾아낼 수 있었다.

1 서론

컴퓨터 프로그램은 휴대폰 단말기부터 의료기기, 자동차, 우주선에 이르기까지 거의 모든 기계에서 실행되어 기계들의 동작을 제어한다. 이러한 상황에서 컴퓨터 프로그램들이 오류없이 정확하게 동작하는 것을 보장하는 것은 무엇보다 중요하다.

현재 상용프로그램을 개발할 때 가장 많이 사용되는 언어로는 C/C++ 가 있다. 다양한 플랫폼의 컴파일러가 존재하고 개발자가 메모리를 관리할 수 있으며 하드웨어의 제어가 용이하기 때문이다. 그러나 C/C++에서는 프로그램 언어 차원에서 오류를 미리 방지해 주는 기능이 없기 때문에 프로그램의 규모가 커지면 오류가 없는 프로그램을 작성하기 어려워 진다.

C/C++로 작성된 프로그램의 치명적인 오류로는 메모리 누수와 버퍼오버런이 대표적이다.

- 메모리 누수 오류란 할당된 메모리를 반환하지 않아서 프로그램에서 사용할 수 있는 메모리가 점점 줄어드는 오류이다. 프로그램이 끝나지 않고 계속 수행되어야 하는 서버 프로그램에서는 갑자기 서버가 제공하는 서비스가 중지되는 상황이 발생하게 된다.
- 버퍼오버런 오류는 할당된 메모리의 영역 이외의 영역에 값을 쓰거나 읽는 것이다. 잘못 쓰여지거나 읽혀진 값에 의해서 프로그램은 비정상적인 상태가 되며 원하지 않는 동작을 하거나 프로그램이 종료되는 경우도 발생하게 된다

Sparrow는 C/C++ 로 작성된 프로그램의 메모리 누수와 버퍼오버런 오류를 프로그램을 실행하기 전에 찾아주는 도구이다. 이는 기존의 테스트를 통해서 오류를 찾는 방법에 비해서 다음과 같은 장점이 있다.

- 모든 경우를 다 포함한다. 테스트를 통한 방법은 모든 테스트 경우를 포함 못할 수도 있지만, Sparrow 는 프로그램이 실행되는 모든 경우를 다 고려한다.
- 테스트 환경이 갖추어 지지 않아도 되므로 오류를 일찍 발견할 수 있다. 테스트는 실행환경이 갖추어져야 가능하지만 Sparrow는 프로그램의 원시코드를 분석하므로 실행환경이 필요없다.

본 논문의 2 절에서는 Sparrow의 동작원리를 개괄적으로 설명하고, 3 절에서는 버퍼오버런 오류를 찾는 방법, 4 절에서는 메모리 누수를 찾는 방법을 설명하고 5 절에서는 Sparrow의 성능을 보여준다

2 Sparrow 개요

Sparrow는 분석기와 이를 도와주는 여러 가지 프로그램들로 이루어진 프로그램 패키지를 일컫는 말이다. Sparrow는 다음과 같은 프로그램들을 포함한다.

- 대상 프로그램의 Makefile을 참조하여 분석기의 입력이 되는 전처리된(preprocessed) 소스코드를 준비해 주는 환경구성 스크립트
- 버퍼 오버런과 메모리누수를 분석하는 분석기
- 분석기가 보고하는 알람 중 거짓이라 생각되는 것들을 제거하는 알람 필터
- 분석 후의 결과를 분류하고 웹페이지 형식으로 보여주는 UI 생성기

[1] 이 연구는 교육인적자원부 두뇌한국 21 사업의 지원을 받았음을 밝힙니다.

분석기가 작업을 시작하기에 앞서 전처리 된 소스코드를 만들어주는 프로그램을 smake라 부른다. smake는 make가 Makefile을 참조하여 컴파일과정을 수행 할 때 원래 컴파일러 대신 우리가 만들어 넣은 가짜 컴파일러를 부르게 한다. 이 가짜 컴파일러는 실제 컴파일과 프리프로세싱을 동시에 수행한다. 이런 방법을 사용하면 프리프로세싱 된 소스를 오브젝트/바이너리 파일 별로 따로 모을 수 있어 이후 대상프로그램의 일부분만 분석하고 싶을 때 유용하게 사용 할 수 있다. smake는 곧 오픈소스 프로그램으로 공개 될 예정이므로 누구나 사용 할 수 있게 될 것이다. 실제 프로그램을 대상으로 이를 분석하는 연구를 할 때, 프리프로세싱 된 코드를 입력으로 받아야 하는 경우가 많기 때문에 프로그램 분석 커뮤니티의 값진 자산이 되리라 기대한다.

분석기는 전처리 된 대상 소스코드를 입력으로 받아 분석을 수행하고 오류발생의 가능성이 높은 부분을 알려준다. 분석기가 하는 자세한 일은 다음 장에 소개하겠다.

분석이 끝난 후 분석기가 보고한 알람은 아무런 정제과정을 거치지 않을 경우 많은 거짓 알람을 포함한다. 이 알람은 Random Forest 필터를 거치며 거짓일 확률이 높은 알람과 참일 확률이 높은 알람으로 자동 분류된다. 필터는 우리 연구실에 지난 몇 년간 쌓인 오픈소스 프로그램들의 알람 참 거짓 분류 결과에 따라 훈련되었다.

최종적으로 참일 가능성이 높다고 판단된 알람들은 UI 생성기를 거쳐 잘 다듬어진 모습의 웹페이지로 만들어지게 된다. 웹페이지는 대상 프로그램의 소스코드를 마우스만으로 브라우징 하기 쉽도록 설계되었으며 ctag의 일부 기능과 문법 강조기능을 지원한다.

3 버퍼 오버런(buffer overrun)

Sparrow 의 버퍼오버런 분석은 요약해석(abstract interpretation)[1]의 틀을 바탕으로 하여 설계되었다. 먼저, 각 프로그램 지점에서 프로그램이 실행 중에 가질 수 있는 상태를 요약한 요약 메모리들을 계산하며 이 때 프로그램의 실행흐름과 함수호출영향을 고려한 분석(flow-sensitive and interprocedural analysis)을 수행한다. 실제 메모리에서의 정수와 실수 값들은 정수 구간(interval)으로, 배열은 요약된 시작 주소, 배열의 크기 구간, 현재 포인터가 가리키고 있는 배열의 오프셋 구간을 가지는 값으로 요약된다. 즉, 프로그램에 나타나는 배열과 인덱스들이 실행 중에 가질 수 있는 값들을 모두 모은 요약 메모리를 살펴보며 버퍼를 접근하는 프로그램 수식에서 버퍼 오버런이 발생할 수 있는 것들을 보고한다.

3.1 고정점 계산 알고리즘(fixpoint iteration algorithm)

각 프로그램 지점에 따른 요약메모리의 계산은 할 일부터 하기 알고리즘(worklist algorithm)을 이용한 고정점 계산을 통해 이루어진다. 이때 넓히기(widening)[1] 연산은 분석이 항상 끝남을 보장하며, 좁히기(narrowing)[1] 연산은 넓히기로 인해 부정확해진 분석의 정확도를 회복시킨다. 분석기는 프로그램식이 아닌 실행흐름그래프를 가지고 분석하므로 넓히기가 필요한 노드들만을 정확하게 찾을 뿐만 아니라 그래프상의 우열관계(dominator)를 이용하여 노드들 간의 분석 순서를 고려하여 효율적으로 고정점을 계산한다.

3.2 분석 정확도 향상을 위한 기술

버퍼오버런 분석기에는 적은 비용으로 분석의 정확도를 높이기 위한 기술들이 적용되었다. 먼저 문자열(string) 처리함수를 비롯한 C 표준 라이브러리의 버퍼 처리 관련 함수들의 동작 및 영향을 요약하여 분석 중에 이용한다. 일반적으로 구간(interval)을 도메인으로 가지는 분석기는 루프를 많이 사용하는 프로그램에 대하여 거짓알람을 많이 발생시킨다. 루프를 자세히 분석하려면 구간도메인보다 정확한 도메인을 사용해야 하는데 이는 분석비용을 증가시키는 단점이 있다. 우리는 구간 분석 후에 정확도 향상이 필요한 루프에 대해서만 더 자세한 분석방법으로 재분석하는 방법으로 루프내의 거짓알람을 효율적으로 줄였다. 또한 함수 호출을 하지 않는 간단한 함수의 경우 분석의 정확도를 위해서 함수 인라인(in-line) 을 한다

3.3 분석비용과 정확도 조절하기

분석기를 실제 현장에서 효율적으로 쓰기 위해 많은 엔지니어링이 적용되었다. 먼저 우리는 실제 프로그램에서 발생하는 버퍼 오버런 오류들이 함수의 인자로 전달되는 버퍼나 인덱스로 인해서는 많이 발생하지만 함수의 리턴값에 의해서는 자주 발생하지 않는다는 점을 발견했다. 또한 함수간 영향을 고려한 분석을 할 때 함수의 리턴값이 미치는 영향을 고려하지 않으면 분석비용을 크게 줄일 수 있다는 것도 알아냈다. 이 두 가지에 착안하여 Sparrow 는 정확도와 비용의 균형을 고려한 세가지 분석 방법을 지원한다. 첫번째 방법은 함수호출에 따른 모든 영향을 고려한다. 두번째 방법은 함수호출에 따른 인자전달을 고려하지만 리턴값을 고려하지 않으며, 마지막은 함수호출을 고려하지 않는 방법이다. 첫 번째 방법은 프로그램의 모든 실행의미를 포섭하므로 분석의 안전성을 보장할 수 있지만 두 번째와 세 번째는 그렇지 못하다. Sparrow 옵션을 주지 않았을 때 두 번째 방법을 이용하여 분석한다. 이 밖에도 분석도중 모르는 값이나 부정확한 값이 계산되었을 때 프로그램식의 타입을 이용하여 값을 알 수 있다면 그 값을 이용하여 분석을 진행한다.

3.4 통계적인 분석을 통한 알람 랭킹

정적분석기의 경우 거짓 알람(false alarm)은 피할 수 없다. 그러나 많은 거짓 알람은 분석결과를 확인해 봐야 하는 사용자에게 필요 이상의 노력을 기울이게 한다.

Sparrow는 Random Forest 통계적인 방법을 이용하여 사용자에게 진짜 오류일 것 같은 알람을 먼저 보여 주고 거짓일 가능성들이 아주 높은 알람들은 보여 주지 않는다.
Random Forest를 학습시키기 위해서 공개 소프트웨어에서 나온 알람으로부터 증후(symptom)을 추출하였다. 분석의 정확도에 영향을 줄 수 있는 요소들을 구분하여 증후들을 분류해보면 문법구조적인 증후(syntactic symptoms), 의미적인 증후(semantic symptoms), 알람 자체의 증후(symptoms of alarm themselves) 들로 구분될 수 있다.

3.5 오류의 원인 분석(Reason Chain)

Sparrow는 위와 같은 방법을 이용한 버퍼오버런 분석을 통해 얼마만큼의 크기를 가진 버퍼의 어느 부분(인덱스)을 접근하여 오버런이 발생했는지에 대한 정보(알람)를 제공하는데, 이 정보가 사실인지 확인하려면, 사용자는 알람이 발생한 위치로부터 프로그램을 거꾸로 실행시켜가면서 버퍼의 크기와 인덱스의 값이 왜 그렇게 되었는지 확인해야 한다. 즉, 프로그램을 거꾸로 실행시켜가면서 알람 발생과 관련이 없는 것들은 무시하고 관련이 있는 것들은 그 효과(메모리 상태 변화)를 기억해야 하며, 최종적으로 관련 있는 것들의 효과들을 모두 병합하여 알람의 참/거짓을 판단하게 된다. 만약 C 프로그램을 한 스텝씩 거꾸로 실행시켜주는 디버거(debugger)가 존재한다면 위와 같은 알람 검증 작업이 그리 어렵지 않겠지만, 안타깝게도 위 작업은 모두 사용자의 머리 속에서 이루어진다.

Sparrow는 알람 검증을 위한 사용자의 편의성을 고려하여 Reason Chain이라는 것을 제공한다. 직관적으로 이야기하면, 이것은 위에서 언급한 알람 검증 과정을 미리 자동으로 수행한 결과를 요약해 놓은 것이다. 정확히 이야기하면, Reason Chain이란 알람이 발생한 프로그램 식의 값에 영향을 줄 수 있는, 즉 알람이 발생한 프로그램 식에 종속적인, 모든 프로그램 식과 그에 해당하는 메모리 상태(Reason Point)를 모아놓은 후, 각각의 Reason Point를 프로그램의 실행 흐름의 역순으로 순서(order)를 정해놓은 것이다.

사용자는 Sparrow가 제공하는 Reason Chain의 시작인 알람 발생 위치로부터 체인을 순서대로 따라 올라가면서 각각의 Reason Point의 메모리 상태를 확인하는 것만으로도 알람 검증 작업을 완료할 수 있다. 이 때, 분석 대상의 모든 프로그램 식의 개수보다 Reason Chain의 Reason Point의 개수가 훨씬 작기 때문에, 알람 검증을 위해 확인해야 할 프로그램 소스 코드의 양이 훨씬 줄어들게 된다. 즉, 모든 프로그램 소스 코드를 한 라인씩 살펴볼 필요가 없이 좀 더 쉽고 빠르게 알람 검증을 할 수 있다.

4 메모리 누수(Memory Leak) 분석

메모리 누수 분석도 요약해석(abstract interpretation)을 기반으로 한다. 요약해석을 기반으로 하는 분석기는 문맥 고려 분석(context sensitive analysis: 함수가 호출될 때의 상태를 구분하는 분석)을 할 때 분석의 비용이 커지게 된다. 이는 함수가 호출되는 모든 곳에서 호출되는 함수들을 분석해야 하기 때문이다. 특정 함수가 여러 번 호출되면 매번 분석을 다시 해야 하는 부담이 생기고, 호출의 깊이가 깊어짐에 따라 분석의 비용은 지수적으로 증가하는 결과를 낳는다. 우리는 문맥 고려 분석을 하면서도 같은 함수가 재분석되는 것을 피하기 위해 함수가 하는 일을 간추려 저장한다. 이후 그 함수가 호출될 때 간추려 놓았던 정보를 사용하여 분석 한다.

4.1 정적 호출 그래프(Static Call Graph)

함수가 하는 일을 간추릴 때 어떤 함수부터 분석해야 할 지를 정해야 한다. 현재 분석 중인 함수가 호출할 함수에 대한 정보들이 있어야 그 정보를 사용해서 현재 분석하는 함수를 제대로 분석할 수 있기 때문이다. 당연히 자신이 부르는 함수가 없는 말단의 함수를 가장 먼저 분석해야 할 것이다. 말단의 함수는 다른 함수의 간추린 정보가 필요하지 않기 때문이다. 그 다음에는 이전에 간추린 정보가 있는 함수를 제외했을 때 가장 말단이 되는 함수를 간추려가면 된다.

이를 위해 분석하기 전에 프로그램에 명시적으로 나타나 있는 호출 구문을 보고, 정적 호출 그래프를 그린다. 이 정적 호출 그래프를 이용하여 위상적인 정렬의 역순(reversed topological sort)으로 함수들을 나열하면 우리가 함수 정보를 간추릴 분석 순서가 된다. 그림 1 은 하나의 예이다.

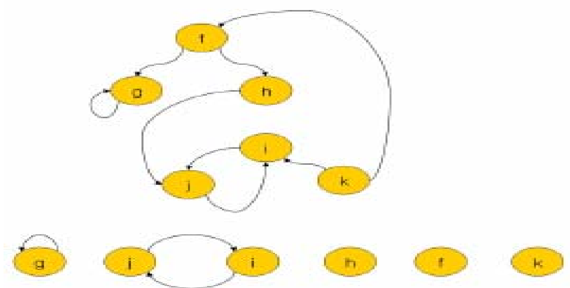

그림 1 정적 호출 그래프와 그의 위상적인 역순 정렬

함수 g는 자신 이외에는 호출하는 함수가 없으므로 가장 먼저 분석하고, 함수 i와 j는 서로 이외에는 호출하는 함수가 없으므로 같이 분석한다. 같이 분석하는 방법은 4.4 에서 설명하겠다. 함수 h는 함수 j를 호출하지만, 이미 함수 j의 정보는 간추려져 있기 때문에 그 정보를 이용하면 분석이 가능하다. 이런 식으로 나머지 함수들도 차례대로 분석이 가능하다.

4.2 함수 분석하기

함수를 간추리는 작업은 요약 해석 기반의 고정점 구하기(fixpoint iteration)로 함수내의 모든 베이직 블락(basic block)들에서의 상태를 구하는 것으로부터 시작된다. 이때 문제는 함수의 인자로 어떤 값이 들어

올 지 모르는 상태에서 분석을 시작해야 한다는 점이다. 요약 해석을 기반으로 하는 분석에서 모르는 값에 대한 처리를 어떻게 할 지 정하는 것은 상당히 중요한 문제이다. 요약 도메인(abstract domain)상에서 모르는 값은 흔히 가장 작은 값인 ⊥(bottom)값 혹은 가장 큰 값인 ⊤ (top)값으로 정하고 분석하는 경우가 많다. 모르는 값을 ⊥으로 정함의 의미는 정의가 안된 값을 나타낸다는 것으로 그 값으로부터 전이 되는 모든 값이 정의 되지 않음을 뜻한다. 따라서, 그 값을 사용하는 모든 연산이 의미가 없게 되어 버린다. 만약 ⊤으로 정하면 그 의미는 모든 값이 될 수 있다고 하여 분석의 정확도를 흐리게 된다. 예를 들어 myfree(int *x)라는 함수가 있어서 그 함수가 하는 일이 x가 가리키고 있는 주소를 해제하는 것이라고 하자. 이 함수를 분석할 때 x가 가리키는 주소는 모르는 값이라고 하여 x의 값을 ⊥으로 하면, 이 함수는 정의되지 않은 값을 해제하므로 아무 일도 하지 않는 함수와 같이 분석된다. 만약 ⊤이라고 해버리면 이 함수는 모든 주소를 해제할 수도 있다라는 의미 없는 분석이 된다. 둘 모두 우리가 원하는 것과는 거리가 멀다. 따라서, 우리는 모르는 값에 대해서 적절히 대처하는 방법이 필요하다.

우리는 모르는 값이 나오면 임의의 타입의 임의의 값을 가질 수 있다고 여기고 그 값이 사용될 때에의 타입을 보고 비로소 특정 값을 할당한다. 만약 그 값이 정수 타입으로 쓰인다면 모든 정수 값으로 하고, 포인터로 쓰인다면 어떤 주소를 가리키고 있는 포인터라고 분석한다. 예를 들어 어떤 프로그램 포인트 p에서 *x를 통해 x의 값을 참조하려고 할 때 변수 x가 메모리 도메인에 존재하지 않는다고 하자. 그러면, x가 실제로 어떤 값을 가지는지는 모르지만, p라는 곳에서 x가 가리키고 있는 어떤 주소가 기호적(symbolic) 존재한다고 생각함으로써 모르는 변수/주소에 대해서도 우리가 원하는 연산을 할 수 있다.

동적으로 할당되는 주소는 무한히 많이 생길 수 있기 때문에 요약할 필요가 있다. 같은 프로그램 포인트에서 할당되는 모든 주소를 하나의 주소로 요약한다. 언뜻 이러한 요약은 분석의 정확도를 현저히 낮출 수 있어 보인다. 하지만, 우리는 함수 간추림을 사용하는 곳, 즉 함수가 호출되는 지점 별로 문맥을 고려하기 때문에 서로 다른 주소가 같은 것으로 요약되는 경우는 극히 드물게 발생한다. 대부분의 경우는 문제가 되지 않고, 반복(loop)안에서 주소를 할당하는 경우에는 실제 실행 중에는 여러 개의 서로 다른 새로운 주소들이 하나로 요약되어 실제 오류를 찾지 못하는 경우도 있다.

4.3 함수 간추림(Procedural Summary)

함수가 하는 일을 정확하게 모두 저장할 수는 없기에 메모리 누수를 검증하는데 중요한 정보 만을 끄집어 내야 한다. 함수 호출을 통해 변화가 일어날 수 있는 환경은 우리가 함수에게 넘겨 주는 인자로부터 도달 가능한(reachable) 주소들과 전역 변수(global variable)들 뿐이다. 이 같은 사실로부터 우리는 메모리 누수를 검증하는데 꼭 필요한 정보가 다음의 9 가지라고 제안한다.

- 함수가 돌려 줄 수 있는 정수 값 (RetVal)
- 함수가 받은 인자로부터 도달 가능한 어떤 주소를 해제하는지 (ArgFree)
- 함수가 받은 인자로부터 도달 가능한 어떤 주소를 전역 변수에 붙이는 지 (Arg2Glob)
- 함수가 받은 인자가 어떻게 변해서 전역 변수로부터 도달 가능하게 되는지 (Glob2Arg)
- 함수가 받은 인자로부터 도달 가능한 어떤 주소를 할당하는지 (ArgAlloc)
- 함수가 돌려주는 주소로부터 도달 가능한 어떤 주소를 할당하는지 (RetAlloc)
- 함수가 돌려주는 주소로부터 도달 가능한 어떤 주소가 전역 변수로부터도 도달 가능한지 (Glob2Ret)
- 함수가 받은 인자로부터 도달 가능한 어떤 주소를 어떻게 돌려주는지 (Arg2Ret)
- 함수가 받은 어떤 인자로부터 도달 가능한 어떤 주소를 또 다른 인자에게 어떻게 넘겨주는지 (Arg2Arg)

4.4 함수 간추림 사용하기

어떤 함수가 호출 될 때 그 함수의 함수 간추림 정보가 있으면 그것을 이용하여 그 함수가 하는 일을 반영한다. 그런데, 만약 그 함수의 간추림 정보가 없다면 그 함수를 같이 분석해야 한다. 같이 분석하는 것은 요약 해석 기반의 분석기에서 행하는 문맥을 구분하지 않는 분석(context insensitive analysis)과 같다. 즉, 함수가 여러 번 호출 되면 호출 될 때 상태들이 모두 하나로 합쳐져서 분석된다.

5 실험결과

우리의 분석기를 상용 오픈 소프트웨어에 대해 적용한 결과는 표 1 과 같다. 실험은 Intel Pentium 4 CPU 3.20GHz 4GB memory의 리눅스(linux) 머신 위에서 이루어졌다. 표에서 true라고 표기한 것은 실제로 오류인 개수를 나타내는 것이고, false는 거짓 알람(false alarm)의 개수를 나타낸다. 거짓 알람은 실제 오류가 아닌데도 불구하고 분석기의 부 정확성 때문에 발생하는 알람이다.

이와 같은 오픈 소프트웨어는 오랜 시간에 걸쳐 여러 개발자들의 의해 검증을 받았을 것인데, 그럼에도 불구하고 실제 오류들을 찾아내었다는 것은 의미 있는 결과라고 생각한다. 실험결과 중 hanterm-3.1.6 의 경우에는 버퍼 오버런 오류가 다른 소프트웨어에 비해 유난히 많은데 이는 프로그래머가 반복적으로 동일한 유형의 실수를 하였기 때문으로 보인다. 표에 있는 소프트웨어 중에 fluxbox는 C++ 소프트웨어인데, 아직까지 우리의 분석기는 C++ 언어에 대한 지원은 미흡한 상태이다. C++의 코딩 스타일은

클래스 인스턴스(class instance)의 멤버 변수(member variable) 값에 대한 접근을 멤버 함수(member method)를 통해 이루어지는 데, 우리의 함수 간추림은 값의 흐름에 대해서는 표현하지 못하고 있다. 우리의 함수 간추림은 포인터 정보와 할당/해제에만 집중을 하고 있을 뿐이다.

표 1 오픈소프트웨어에 대한 분석 결과

Program	LOC	Memory Leak		Buffer Overrun		Time(s)
		Tru	Fals	True	False	
httpd-2.2.2	316,436	4	1	0	55	521
tcl-8.4.14	215,674	28	66	0	79	1,456
fftw-3.1.2	184,091	0	1	0	11	23
OpenSSH-4.3p2	77,329	7	7	0	12	306
fluxbox	56,127	1	8	0	4	3,577
tar-1.13	49,581	4	2	1	20	88
bison-2.3	42,855	5	4	0	17	362
grep-2.5.1a	31,160	1	3	0	1	52
sed-4.08	26,807	24	15	0	11	69
hanterm-3.1.6	25,518	1	0	34	26	67
GNUchess-5.07	13,768	4	0	4	6	100
gzip-1.2.4	9,076	0	0	0	45	13

표를 보면 코드의 크기와 분석 시간이 반드시 비례하지 않는 다는 사실을 알 수 있다. 우리의 분석기의 분석 시간은 코드의 크기와 실행흐름의 복잡도에 모두 관여하기 때문이다. C++ 소프트웨어인 fluxbox의 분석 시간이 유달리 오래 걸리는 이유는 우리가 파싱에 사용하고 있는 edg 파서가 C++ 언어를 C 언어로 변환하는 과정에서 코드가 폭발적으로 커지기 때문이다. C 소프트웨어인 tcl-8.4.14 의 분석 시간이 오래 걸리는 이유는 분석 대상 코드에 파싱을 하는 루틴이 포함되어 있으면 재귀적 호출(recursive call)이나 스위치 문의 잦은 사용으로 코드의 실행 흐름이 복잡하기 때문이다.

6 결론

우리는 실제 현장에서 유용하게 쓰일 수 있는 수준의 소스 코드 분석기 Sparrow를 개발하였다. Sparrow는 C/C++로 작성된 수십만 라인의 프로그램을 자동으로 분석하여 실제 오류일 가능성이 높은 알람들만을 사용자에게 보여준다. 상용 오픈 소프트웨어들을 분석한 결과 많은 수의 실제 오류들을 찾아낼 수 있었다.

큰 프로그램을 효율적으로 분석하기 위해 Sparrow의 분석기 엔진은 많은 엔지니어링이 적용되었다. 분석기는 효율적인 인라인을 비롯한 정확도 향상 기술, 함수호출 영향을 구분하여 정확도와 분석비용 조절 기술, 함수 간추림 정보의 활용 등의 기술을 이용하여 분석의 효율성을 높인다.

Sparrow는 사용자의 편의성을 위해 많은 기능을 제공한다. 먼저 대상 프로그램의 소스 구조를 자동으로 파악하여 전처리된 소스코드를 자동으로 준비하여 준다. 또한 분석기가 발생시킨 알람중 거짓일 가능성이 높은 알람을 자동으로 제거하며 실제 오류일 가능성이 높은 알람들의 발생원인을 설명해 주는 기능을 웹페이지 기반의 GUI로 제공한다.

7 참고 문헌

1 Patrick Cousot, Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238-252, January 1977.

2 Bruno Blanchet, Patric Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antonie Mine, David Monnizux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 196-207, June 2003

3 Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Mine, David Monniaux, and Xavier Rival. The astree analyzer. In M. Sagiv, editor, European Symposium on Programming (ESOP'05), volume 3444 of *Lecture Notes in Computer Science*, pages 21-30. Springer-Verlag, 2005.

4 Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In SAS 2005: 12th Annual International Static Analysis Symposium, volume 3672 of *Lecture Notes in Computer Science*, pages 203-217. Springer, 2005.

5 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*, pages 115-125, New York, NY, USA, 2005, ACM Press.

6 E. M. Nystrom, H. S. Kim, and W. M. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th Static Analysis Symposium*, August 2004.