아이락: C 프로그램의 메모리 오류 정적 분석기

정영범O, 김재황, 신재호, 이광근
서울대학교 프로그래밍 연구실
{dreameyeO, jaehwang, netj, kwang}@ropas.snu.ac.kr

Airac: Static Analyzer for Automatic Verification of Array Index Ranges in C Programs

Yungbum JungO , Jaehwang Kim, Jaeho Shin, Kwangkeun Yi
Programming Research Lab, Seoul National University

요 약

아이락(Airac)은 C 프로그램의 버퍼오버런(buffer overrun)오류를 찾아주는 정적 프로그램 분석기(static program analyzer)이다. 아이락은 요약해석(abstract interpretation)의 틀 속에서 디자인되었다. 설계 및 구현 과정에서 프로그램 분석 분야에서 축적되어온 다양한 기술들을 적용하여 분석의 성능 및 정확도 향상을 이룩하였다. 아이락은 리눅스 커널(linux kernel), GNU 소프트웨어, 상용 소프트웨어등에 적용되어 오류를 찾아냈다.

1. 서 론

안전하고 견고한 C 프로그램을 작성하는 것은 어려운 일이다. 버퍼 오버런(buffer overrun) 오류는 C 프로그래머를 가장 곤란하게 하는 문제 중의 하나다. 버퍼 오버런은 할당된 메모리 영역 밖의 주소가 참조되는 경우에 발생한다. 버퍼 오버런은 프로그램에 특별한 입력이 주어졌을 때만 발생하기 때문에 문제가 되는 입력 값을 알기 전에는 디버깅하기가 어렵다. 일반적으로 프로그램 입력 값의 범위는 무한하기 때문에 모든 경우를 테스트해볼 수 없다. 따라서 테스트로는 버퍼 오버런 문제를 완벽하게 해결할 수가 없다. Java처럼 실행 중에 배열의 참조가 배열의 범위 내에서 이루어지는 지를 검사하는 것은 실행의 비용을 증가시킨다.

아이락(Airac)은 정적 프로그램 분석(static program analysis) 기술을 이용하여 C 프로그램에 잠재된 모든 버퍼 오버런 에러들을 실행 전에 찾아준다. 아이락은 요약해석틀(abstract interpretation framework)[1]을 따라 설계되었다. 아이락은 프로그램의 분석의 안전성(soundness)을 100% 보장하기 위해서 프로그램이 실행 중에 가질 수 있는 모든 상태를 고려한다. 아이락은 프로그램이 실행 중에 가질 수 있는 모든 상태를 정적으로 안전하게 요약(sound approximation)하고, 요약된 프로그램의 상태 정보를 바탕으로 발생 가능한 모든 버퍼 오버런을 찾아 준다.

안전한 요약은 실재에는 일어나지 않는 상황까지 포함하기 때문에 허위 경보(false alarm)가 발생할 수 있다. 즉 실제로는 버퍼 오버런이 일어나지 않는데 일어난다고 알려줄 수 있다. 아이락은 모든 버퍼 오버런을 찾아주면서 허위 경보를 최소화하기 위해 프로그램 분석의 다양한 기술을 사용하였다.

2. 분석기 디자인

2.1 프로그램의 실행 의미

아이락은 C 프로그램을 같은 의미를 가지면서 보다 간결한 프로그램 구성자(program construct)로 이루어진 C'프로그램으로 변환 한다. 아이락은 ANSI C를 지원하며 리눅스 커널을 분석할 수 있을 정도의 GNU C 확장을 지원한다.

요약해석(abstract interpretation)은 프로그램의 실제 의미를 요약하여 실행하는 프로그램 분석의 틀이다. 그 틀에 따라 프로그램의 실제 의미 공간(concrete semantic domain)을 정의하고 실제 의미 공간과 갈로아 연결(Galois connection)을 가지는 요약 의미 공간(abstract semantic domain)을 정의한다. 분석은 요약 의미 공간에서 정의된 요약 실행 의미(abstract semantics)를 따라 프로그램을 실행함으로써 프로그램의 실제 실행에서 가질 수 있는 가능한 상태들을 포섭하는 프로그램 상태를 계산한다.

C' 프로그램의 실제 실행은 기계 상태의 전이로 정의된다. 기계 상태는 수행해야 할 명령문의 위치($PgmPnt$)와 그 위치에서 가지는 상태($State$)의 짝으로 이루어진다:

$$Machine = State \times PgmPnt$$

$$F : 2^{Machine^{\omega}} \rightarrow 2^{Machine^{\omega}}$$

$$F(X) = \{m_0\} \cup$$

$$\{t \rightarrow m_{n+1} \mid t = \mathrm{K} \rightarrow m_n \in X, m_n \rightarrow m_{n+1}\}$$

그러므로 실제 프로그램의 의미는 위와 같이 정의된 함수 F 의 최소 고정점으로 정의된다.

버퍼 오버런을 분석하기 위한 요약 공간에서 정수값들은 구간(interval) 도메인의 값으로 요약된다. 구간 도메인 전체의 최소상계(least upper bound) T은 [-∞,+∞] 이고, "모든 정수 값이 될 수 있음" 을 뜻한다. 요약 공간에서 메모리 주소는 변수명과 메모리가 할당된 프로그램 위치로 요약된다. 분석의 정확도를 향상시키기 위해 모든 변수의 식별자를 고유한 이름을 갖도록 변환하여 각 변수들이 다른 주소를 갖도록 한다. 배열은 메모리 주소, 배열의 크기, 오프셑의 데카르트 곱으로 정의된다. 배열의 크기, 오프셑은 모두 정수 구간의 값이다.

아이락은 프로그램 실행 궤적(trace)을 프로그램 포인트 별로 다시 요약하여 각 프로그램 포인트에서 프로그램이 실행 중에 가질 수 있는 상태를 포섭하는 요약 상태를 계산한다.

2.2 고정점 알고리즘 (fixpoint algorithm)

고정점은 할 일만 하기 방식(working set)을 사용하여 구한다. 다음에 할 일은 스택에 저장된다. 스택은 요약 실행 과정에서 다음에 계산해야 하는 프로그램 포인트의 집합을 가진다. 어떤 프로그램 포인트에서 계산한 프로그램 상태가 변했다면 다음에 다시 계산해야 하는 프로그램 포인트를 스택에 추가한다.. 따라서, 할 일만 하기 방식은 깊이 우선 탐색(depth-first traverse)으로서 프로그램의 플로우 그래프를 탐색하면서 고정점을 계산해 낸다. 만약 다음에 계산해야 할 프로그램 포인트가 여러 개이면(if문이나 while문에서 조건문의 값이 참인지 거짓인지 알 수 없는 경우), if문은 넓이 우선 탐색(breadth-first traverse)을 while문에서는 루프의 바디를 먼저 실행한다. 고정점을 구하는 알고리즘은 두 개의 부분으로 구성되어 있다. 우선 넓히기(widening)를 쓰고 난 후에 좁히기(narrowing)를 한다. 넓히기는 정수 구간 도메인의 높이가 무한하기 때문에 유한한 시간 내에 분석을 끝내기 위해 필수적이다. 하지만 넓히기를 사용하면 최소 고정점 보다 큰 고정점을 계산하므로 분석 정확도는 떨어지게 된다. 이를 보완하기 위해 넓히기 후에 좁히기를 적용한다.

3. 성능

3.1 정확도 향상을 위한 기술

아이락은 분석의 정확도 향상을 위하여 프로그램 분석 분야에서 오랫동안 축적한 전통적인 기술들을 사용한다. 각 기술들은 서로 독립적이기 때문에, 모든 기술들을 조합하여 사용하면 각각을 사용하였을 때에 비해 더 정확한 분석을 얻을 수 있다. 1) **새롭게 이름짓기(unique renaming)** 요약 공간의 기계 상태는 환경을 갖지 않는다. 따라서 모든 식별자들을 고유한 이름으로 바꿔준다. 2) **넓히기 후에 좁히기 적용(narrowing after widening)**[2] 정수 구간 공간의 높이는 무한하다. 넓히기(widening) 연산자는 유한한 시간 안에 분석이 끝나게끔 하는 핵심요소이나, 이 연산자는 분석결과 의 정확도를 떨어뜨린다. 좁히기(narrowing) 연산은 떨어진 정확도를 복구하기 위해 쓰인다. 배열 인덱스의 정확도를 위해 특화된 넓히기와 좁히기 연산을 사용한다. 3) **수행 순서 고려 분석(flow sensitive analysis)** 메모리에 덮어쓰기(destructive update)는 대상 주소의 실제 의미가 여러 개를 갖지 않는 경우에는 항상 허용된다. 4) **가지치기 (context pruning)** 분기문이나 반복문의 조건식(conditional expression)을 만족하는 최소의 기계 상태를 구한다. 아이락은 선형수식(linear expression)에 대해서는 이러한 가지치기연산을 지원한다. 가지치기는 소위 역방향 분석(backward analysis)라고도 한다. 5) **함수 펼치기(function inlining)** 호출 위치에 따라 함수의 분석을 구분하여 진행하는 것을 의미한다. 이 방법을 통해 같은 함수가 여러 프로그램 포인트에서 호출 될 때 함수 내부의 프로그램 포인트에서 이전에 호출 되어 남아 있던 상태들의 의해 현재 값이 오염되는 것을 막을 수 있다. 그런데, 만약 무한히 재귀 호출되는 함수에서도 펼치기를 계속하게 되면 분석기가 끝나지 않는다. 이를 막기 위해 사용자로부터 펼치기 회수를 받아 제한한다. 6) **루프 펼치기(loop unrolling)** 함수 펼치기과 마찬가지로 루프의 바디 안에서도 루프의 반복 회수에 따라 프로그램 분석을 구분하여 정확도를 높인다.

역시 무한한 루프를 도는 프로그램의 분석을 유한한 시간 안에 끝내기 위해 사용자가 정해준 회수 이상으로 루프를 풀지는 않는다. 5), 6) 번에서 사용자로부터 받는 패러미터는 그 값이 클수록 분석의 정확도는 올라가지만 분석의 시간도 따라서 오래 걸리게 된다

3.2 분석 속도 향상을 위한 기술

다음은 아이락의 분석 속도를 높이기 위해 고안한 기술들이다. 1) **효율적인 합치기/비교 연산(selective join/ordering)** 메모리의 원소 중에 어떤 것이 변화했는지에 대한 정보를 가지고 접합점에서 변화된 원소들에 대해서만 조인/비교 연산을 취한다. 2) **분석 지연시키기(wait-at-join)** 분석이 여러 실행흐름이 모이는 곳에서는 모든 실행의 분석이 끝날 때까지 기다린 후에 분석을 실행한 다. 이 방법을 통해 분석이 중복될 수 있는

종류	이름	줄 수	시간 (초)	경보 개수		실제 오류
				배열 수	접근 수	
GNU SW	tar-1.13	20258	576.79s	24	66	1
	bison-1.875	25907	809.35s	28	50	0
Linux kernel 2.6.4	vmax302.c	246	0.28s	1	1	1
	xfrm_user.c	1201	45.07s	2	2	1
	usb-midi.c	2206	91.32s	2	10	4
	atkbd.c	811	1.99s	2	2	2
	keyboard.c	1256	3.36s	2	2	1
상용 SW	T*	109878	4525.02s	16	64	0
	DS*	138305	38328.88s	34	147	47
	Es*	233536	4285.13s	28	162	6
	El*	47268	2458.03s	25	273	1

표 1 아이락의 성능

경우를 제거한다. 특히 C 프로그램의 스위치(switch)문에 대해 아주 큰 효과를 발휘할 수 있다. 3) **스택 제거 (stack obviation)** 아이락에서 사용한 실행 의미(semantics)는 프로그램 식 수준의 기계 상태 전이로 정의되어 있다. 이를 위해 프로그램 상태는 프로그램 식의 값을 저장하기 위한 스택을 포함한다. 그런데, 스택이 프로그램 상태의 다른 구성자에 비해 상당히 크기 때문에, 분석 시간의 대부분은 스택을 합치고, 비교하는데 소요된다. 아이락은 스택의 변화가 모두 메모리에 반영되도록 프로그램을 변환하여 스택을 합치고, 비교하는 대신 메모리에 대해서만 합치기, 비교를 할 수 있게 한다. 이를 통해 프로그램 분석 속도를 경감시킬 수 있다.

4. 실험

nML(http://ropas.snu.ac.kr/n) 언어로 아이락을 직접 구현하여 다양한 소프트웨어들을 분석해 보았다. 아이락은 GNU 소프트웨어, 리눅스 커널 프로그램과 상업용 s 내장형 소프트웨어(embedded software)에서 여러 가지 버퍼오버런 오류들을 찾아내었다. 상업용 붙박이 소프트웨어 중의 일부는 아직 개발 중인 단계였고, 어떤 것들은 이미 테스트 단계를 거친 것들이었다. [표 1]은 아이락의 분석 결과를 보여준다. 모든 분석은 펜티엄4 3.2GHZ CPU와 4GB 메모리를 가진 리눅스 2.6 시스템에서 이뤄졌다.

성능을 향상시키기 위해 사용한 기술 세가지가 어떤 분석 속도에 어떤 영향을 미치는 지에 대한 실험 결과

Version	Time[a](s)	Speed-Up	Time[b](s)	Speed-Up
none	18317.55	0 %	16253.18	0%
selective join	16055.58	12.35%	14286.72	12.1%
wait-at-join	19317.67	-5.45%	13153.43	19.98%
stack obviation	3717.06	79.71%	3247.79	81.02%
all	3461.57	81.11%	2320.58	85.73%

표 2 아이락 분석 속도 향상을 위한 기술 세그

를 [표 2]에 나타내었다. 이 [표 2]에서 Time[a]는 43개의 리눅스 프로그램에 대해 적용했을 때의 결과를 나타내고, Time[b]는 그 중에서 분석 지연시키기(wait-at-join)가 오히려 악영향을 미치는 하나의 프로그램을 제외한 시간을 보여준다. 이 결과로부터 스택 제거 방법이 프로그램 분석 속도를 올리는 획기적인 방법임을 확인 할 수 있다.

5. 결론 및 토의

정적 프로그램분석 기술은 실제 프로그램들에 적용할 수 있을 만큼 성숙하였다. 우리는 아이락을 구현하여 이를 확인했다. 이를 통해 분석기 디자인의 실제와 여러 가지 성능 향상을 위한 방법을 제시했다. 아이락에 대한 보다 자세한 정보는 [3]에서 얻을 수 있다.

참고문헌

[1] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints
Proceedings of ACM Symposium on Principles of Programming Languages. pp. 238--252. January 1977

[2] Patrick Cousot and Radhia Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation
Proceedings of the International Workshop Programming Language Implementation and Logic Programming. pp. 269--295. 1992

[3] 아이락 홈페이지 (http://ropas.snu.ac.kr/airac)