논·문 19-2-?

요약해석기 AiracV [1]

Abstract Interpreter AiracV

신재호 · 김재황 · 오학주 · 정영범 · 이광근
서울대학교 컴퓨터공학부 프로그래밍 연구실
{netj; jaehwang; pronto; dreameye; kwang}@ropas.snu.ac.kr

요 약

AiracV는 요약해석틀에 기반하여 C 프로그램의 버퍼오버런을 안전하게 찾는 정적 프로그램 분석기이다. 우리가 Airac으로부터 얻은 경험을 바탕으로 AiracV는 한 층 개선된 설계를 바탕으로 구현하고 있다. AiracV가 기존의 Airac에 비해서 달라진 점을 살펴보고, 현재 우리가 설계의 일부를 구현한 AiracV 첫 판의 성능을 Airac과 비교하여 본다.

1 요약해석기 AiracV

Airac[8]은 요약해석틀(abstract interpretation framework)[5, 6, 4]에 기반하여 C 프로그램[9]에서 발생할 수 있는 모든 버퍼 오버런 오류를 안전하게 찾아주는 정적 프로그램 분석기이다. Airac은 프로그램의 각 지점 별로 도달가능한 상태를 안전하게 추정한다. 그리고 이 상태들을 검사하여 버퍼 오버런이 발생할 수 있는 지점들을 알려준다. Airac은 리눅스 커널, GNU 소프트웨어 및 상용 임베디드 소프트웨어 등 실제 C 프로그램에서 오류를 적절한 비용과 정확도로 찾아내는데 성공하였다. 정적 프로그램 분석 기술을 실제 현장에 적용해도 무리가 없을 정도로 무르익었음을 확인시켜주었다.

AiracV[아이락 파이브]는 Airac을 설계하고 만들면서 얻은 경험을 바탕으로 밑바닥부터 완전히 새로 구현하고 있는 요약해석기이다. 우리는 Airac에 사용한 기술들을 명료하게 정리하고 그 안전성을 엄밀하게 검증하고 싶었으며, Airac에서 얻은 성과를 버퍼오버런 외의 다른 종류의 분석으로도 쉽게 확장하고 싶었다. 이 과정에서 기존의 설계나 구현으로는 쉽지 않은 일임을 확인하였고, 설계에서부터 구현까지 빈틈이 없도록 명료하게 해나가는 작업을 진행해왔다. 그 결과 AiracV는 똑같이 구간 도메인(interval domain)에서 요약해석을 하는 분석기임에도 불구하고 Airac과 매우 다른 모습을 갖게 되었다.

2 AiracV가 Airac과 다른 점

2.1 깔끔한 중간언어

Airac의 C'에서는 모두 뒤섞여있었던 메모리 반응을 일으키거나 실행흐름을 바꾸는 *명령*(statement; command)과 값을 표현하기 위한 *계산 식*(expression)을, AiracV에서는 서로 완전히 분리하여 매우 명료한 중간언어로 만들고 분석에 사용한다.

[1]본 연구는 정보통신부 선도기반기술개발사업과 교육인적자원부 두뇌한국21사업의 지원으로 수행하였다.

2.2 실행흐름 그래프

AiracV는 프로그램 식이 아닌 실행흐름 그래프(control flow graph)[1, §9.4]를 가지고 분석한다. 명령과 식으로 분리한 새 중간언어를 사용하기 때문에 실행흐름 그래프를 간단한 방법으로 그릴 수 있다. 분석 전 단계에서 이 그래프를 미리 그려두고 분석에 들어간다. 가능한 실행 흐름을 미리 파악할 수 있기 때문에 축지법 지점(widening point)을 정교하게 찾는 일 등을 자연스럽게 할 수 있다.

Airac은 프로그램 식을 분석에서 직접 다루면서 드러나지 않게 실행흐름 그래프를 그리면서 분석을 진행했다. C 프로그램에서는 goto문을 자유롭게 사용할 수 있기 때문에 실행흐름 그래프를 만들지 아니할 수 없기 때문이다. 프로그램의 모든 부분에 대해서 그래프를 그리는 작업을 분석과 함께 하는 것은 구현을 불필요하게 복잡하게 만들었으며, 한 번 그려둔 부분의 그래프를 반복해서 그리게 되는 비효율도 있었다. 따라서 우리는 설계에서부터 일관되게 그래프를 대상으로 분석하는 방향을 택한 것이다.

2.3 블럭 단위의 분석

AiracV에서는 분석 단위를 블럭(basic block)[1, §9.4]으로 크게 늘렸다. 기존의 Airac은 프로그램의 실행흐름을 매우 조밀하게 나누어 각 명령 및 계산 식들까지 모든 지점의 상태를 보관하고 있었다. 그러나 이는 분석에 필요한 메모리를 지나치게 늘리는 문제가 있어 우리는 흐름이 바뀌지 않는 블럭 단위로 상태를 기억하기로 하였다. 마지막에 분석 결과로부터 경보를 생성할 때에는 각 블럭의 입력 상태를 가지고 한 번 더 각 명령들의 의미를 따라가야 하는 단점이 있지만 우리는 분석에 필요한 메모리를 절약할 수 있다는 큰 장점을 택했다.

2.4 체계적인 구성

AiracV는 고정점 알고리즘과 프로그램 의미 정의, 그리고 분석 도메인의 정의들을 구현 수준에서부터 별도의 계층으로 완전히 격리하였다. 이로써 다른 도메인을 사용하는 분석으로 확장하는 일이 쉬워졌으며, 구현과 설계를 거의 일치 시킨 것이므로 구현의 검증이 설계의 검증 보다 별로 어렵지 않은 수준으로 끌어올렸다. 대부분의 비용 절감 방법들도 층을 넘나들지 않고 구현하기 때문에 그 방법이 왜 올바른가를 부분만 보면서도 명확하게 확인할 수 있으며 어떠한 원리로 비용이 줄어드는가도 분명히 드러난다.

2.5 줄어든 축지법 지점

축지법 연산[7]은 높이가 무한한 요약공간에서 분석을 유한 시간 안에 끝내기 위해서 반드시 해야만 한다. 그러나 정확도를 크게 낮추는 요인이 되기도 하며, 이 정확도를 회복하기 위한 좁히기(narrowing) 연산을 유발하여 분석을 지연시키기도 한다. C 프로그램의 경우, 어떤 지점이 변화시킨 상태가 다시 자기 자신에게 들어오는 경우, 즉 고리(loop)에 속한 지점들은 상태가 무한히 바뀌면서 고정점을 찾지 못할 수 있다. 따라서 바깥에서 고리로 들어가는 길과 고리에서 되돌아오는 길(back edge)이 만나는 고리시작점(loop head)에서 축지법 연산을 적용해주어야 한다. 기존의 Airac은 고리시작점들을 포함하여 두 길 이상이 만나는 모든 합류점(join point)에서 축지법을 적용하고 있었다.

AiracV는 그 구조적 이점을 이용하여 축지법(widening) 연산을 효율적으로 적용하려고 한다. AiracV는 함수 내의 그래프는 미리 그려두기 때문에 반복문이나 goto문으로 된 고리들을 미리 찾아둘 수 있고 재귀적인 함수 호출에 의해서 생기는 고리들은 분석 중에 함수 호출 관계로부터 알 수 있다. 나아가 각 고리시작점의 되돌아오는 길의 상태를 검사하여 실제로 고리가 형성될 수 없는 경우까지도 배제하여 가짜 고리도 식별해낼 수 있다. AiracV는

이렇게 찾은 고리의 시작점들에만 축지법 연산을 적용시켜 분석을 더 효율적으로 끝낼 수 있다.

2.6 똑똑한 계산 순서

AiracV는 실행흐름 그래프를 가지고 분석하기 때문에 고정점 알고리즘에서 계산 순서를 더 똑똑하게 할 수 있다. Airac이 사용하는 합류점 지연(wait-at-join)[8, §2.4] 기술을 개선하여 비용을 더 줄일 수 있다. 기다리는 합류점이 여럿 있을 때, Airac은 임의로 선택을 하는 반면, AiracV는 그래프에서의 지점들의 우열 관계(dominator)[1, §10.4]를 이용하여 불필요한 계산을 줄일 수 있다. 나아가 고정점 계산 과정에서 드러나는 함수 호출 관계까지 고려하면 재개할 합류점을 잘못 골라 발생하는 계산량을 더 크게 줄일 수 있다.

2.7 개선된 골라 합치기

AiracV는 분석 전반에 걸쳐서 효과가 지속되는 골라 합치기(selective join) 기술을 사용하려고 한다. 골라 합치기 기술은 [2, §6.2] 에서도 간략히 그 개념을 언급하고 있다. Airac의 골라 합치기 구현[8, §2.4]은 노폐물이 누적되면서 그 효과가 특정 부분에서만 나타나는 결함이 있었다. AiracV는 이러한 노폐물들을 꾸준히 격리하여 쌓아가는 방법으로 골라 합치기의 완성도를 높이고 확실한 비용 절감을 달성하였다.

2.8 가벼운 호출경로별 분석

함수들의 호출경로(context sensitivity)를 고려한 분석에 드는 비용을 단순히 함수의 몸뚱이(procedure body)를 가져와 펼치는(inlining) 것보다 대폭 낮추려고 한다. Airac이 취하는 방식은 함수 호출 지점을 함수의 몸뚱이를 새로운 프로그램 지점들로 만들어 복사해넣어 분석을 진행하는 것이다. 그러나 함수를 어떤 경로로 불렸느냐에 따라 함수의 몸뚱이가 달라지지는 않으므로 바로 이 부분에 상당량의 개선의 여지가 있음을 알 수 있다. AiracV는 함수 몸뚱이의 그래프는 복사하지 않고 꼭 기억할 필요가 있는 함수간의 호출 관계와 각 지점의 호출경로별 상태만을 기억하여 비용을 대폭 절감할 것이다.

2.9 일반적 실행과정 분할

분석의 정확도를 높이기 위해서 실행과정 분할(trace partitioning) 기술[11]을 사용할 수 있다. Airac이 사용하는 반복문 펼치기(loop unrolling)나 재귀호출 펼치기 역시 이 기술의 한 예로 이해할 수 있다. Airac은 goto문으로 만들어진 고리는 펼치지 못하며, 함수 펼치기와 마찬가지로 프로그램 코드를 복사하여 반복문 및 재귀호출을 펼친다. 프로그램의 생긴 모양이 실행과정에 따라서 변하는 것이 역시 아니므로 코드를 복사하여 구현하는 것은 불필요한 메모리를 사용하는 방법이다. 따라서 함수 펼치기와 비슷하게 실행과정별 상태만을 더 기억하는 방식으로 구현하면 비용을 줄일 수 있다. AiracV는 반복문 펼치기 및 재귀호출 펼치기를 포함하는 일반적인 실행과정 분할 기술을 효율적으로 할 수 있도록 구현하고 있다. 나아가 임의의 실행과정을 나누어 분석시킬 수 있는 지시법을 고안하여 사람이 수동으로 또는 별도의 분석기가 분석의 정확도를 조절할 수 있게 하려고 한다.

3 성능 맛보기

앞서 설명한 내용들이 현재 AiracV에 모두 구현된 것은 아니나, Airac에서 쓰인 성능 개선 기술들을 비슷한 수준에서 사용하고 있다. 표 I, II에 나타낸 값들은 현재의 AiracV를 가지고 측정한 결과이다. 실험은 모두 인텔 펜티엄4 3.2GHz 프로세서와 4GB 메모리를 갖춘 리

눅스 2.6 시스템에서 수행하였다. AiracV과 Airac은 재귀호출이 없는 함수 호출을 몇 번까지 펼칠(inlining) 것인지를 지정 할 수 있는데, 모두 한 번까지 펼친 후에 분석하도록 하였다. AiracV와 Airac의 분석시간은 전처리와 변환에 걸리는 시간을 제외한 온전히 분석에만 걸린 CPU시간이며, 알람 수는 모두 버퍼의 크기나 인덱스 범위를 전혀 알 수 없는 경우는 제외한 개수이다. 표 I의 GNU 소프트웨어의 경우에는 모든 함수를 차례대로 분석하는 방식으로 실험한 결과이며, 표 II는 분석 시나리오를 만들어서 실험한 결과임을 밝혀둔다.

프로그램 (LOC)	AiracV			Airac		
	시간(sec)	알람수	실제버그	시간(sec)	알람수	실제버그
sed-4.0.8 (6053)	256	5	0	1089	18	0
gzip-1.2.4a (7327)	330	50	0	2363	102	0
grep-2.5.1 (9297)	1314	28	0	864	16	0
tar-1.13 (20258)	2003	76	1	6524	138	1
bison-1.875 ()	7050	30	0	5089	91	0

[표 I]AiracV와 Airac의 성능 비교 – GNU 소프트웨어

프로그램 (LOC)	AiracV			Airac		
	시간	알람수	실제버그	시간	알람수	실제버그
vmax301.c (246)	0.01	1	1	0.38	1	1
cdc_acm.c (849)	0.14	2	2	7.14	3	2
atkbd.c (944)	0.27	3	2	2.80	5	2
eata_pio.c (984)	12.3	10	1	7.12	5	1
ip6_output.c (1110)	1.07	0	0	2.09	0	0
xfrm_user.c (1201)	93.63	3	1	53.45	5	1
keyboard.c(1256)	0.38	14	0	4.21	6	0
af_inet.c (1273)	0.87	3	2	4.03	1	1
usb_midi.c (2206)	14.45	11	4	167.66	10	4
aty128fb.c (2466)	0.03	1	1	0.64	1	1
mptbase.c (6158)	0.06	1	1	1.83	1	1

[표 II]AiracV와 Airac의 성능 비교 – Linux Kernel 2.6.4의 일부

몇몇 경우를 제외하고 리눅스 커널(Linux Kernel)과 GNU 프로그램 모두에서 AiracV의 분석시간 및 정확도가 Airac에 비해 매우 향상되었음을 확인할 수 있다. AiracV처럼 호출경로를 무시하고 함수의 몸뚱이를 하나로 요약하는 경우, 만약 함수 호출을 두 번 이상 한다면 함수를 호출한 새 입력환경에 따라 더 큰(느슨해진) 출력환경을 가지고 그 함수를 호출한 모든 지점 이후를 다시 분석해야만 한다. 따라서, 함수를 하나로 보는 분석(AiracV)이 함수를 모두 펼치는 경우(Airac)보다 반드시 계산량이 적은 것은 아니며, 함수를 모두 펼치는 경우보다 더 많은 계산을 하는 상황에 종종 놓인다. 물론, 함수를 호출 경로별로 분석하면 각각의 상태를 기억하기 위해 훨씬 많은 공간을 필요로 하는 것은 분명하다. 하지만 공간만을 더 필요로 할 뿐 시간까지 더 필요로 하지는 않을 것이라고 예상한다. 함수들의 상태를 도를 지나쳐 요약(over approximation)하기 위해 불필요한 계산을 하고 있음에도 불구하고 Airac보다 빠른 분석 시간을 보여주고 있으므로 AiracV에 함수호출별(context sensitive) 분석을 장착하여 재귀 호출을 제외한 모든 함수 호출을 펼친다고 하더라도 크게 느려지지 않는 성능을 보여줄 수 있을 것으로 기대한다.

표 III은 Raccoon[10]과의 비교를 보여준다. AiracV와 Raccoon과의 비교에서 Raccoon의 분석시간은 프로그램의 CFG 생성 및 포인터 분석 시간, 파일 I/O 시간을 모두 더한 값임

프로그램 (LOC)	AiracV					Raccoon		
	변환	그래프생성	분석	알람수	실제버그	시간	알람수	실제버그
vmax301.c (246)	0.64s	0.14s	0.01s	1	1	0.04s	1	1
cdc_acm.c (849)	1.04s	0.17s	0.14s	2	2	0.73s	2	2
atkbd.c (944)	0.92s	0.16s	0.27s	3	2	0.68s	2	2
xfrm_user.c (1201)	2.36s	0.45s	93.63s	3	1	8.77s	1	1
af_inet.c (1273)	3.07s	0.55s	0.87s	3	2	1.17s	3	1
usb_midi.c (2206)	1.32s	0.3s	14.45s	11	4	1.04s	4	4
aty128fb.c (2466)	1.34s	0.18s	0.03s	1	1	1.06s	1	1
mptbase.c (6158)	2.27s	0.4s	0.06s	1	1	3.34s	1	1

[표 III]AiracV와 Raccoon의 성능 비교 – Linux Kernel 2.6.4의 일부

을 밝혀둔다. 총 8개의 프로그램 중에서 vmax301.c를 포함한 6개 프로그램의 분석에서는 AiracV가 더 빠른지만 xfrm_user.c와 usb_midi.c에 대하여는 Raccoon이 앞섰다.

4 결론 및 앞으로

AiracV는 우리가 Airac을 만들면서 얻은 경험을 토대로하여 현재 만들고 있는 더 가볍고, 더 빠르고, 더 정확하며, 더 유연한, C 언어를 위한 버퍼 오버런 분석기이다. Airac을 만들면서 겪은 여러 설계상의 문제점들을 고려하여 설계 및 구현을 신중하게 진행하고 있다. Airac의 구현에서 취했던 암묵적인 가정들을 드러내어 사소한 부분들까지도 명확히 하고 있다. 그 과정에서 분석의 신뢰도나 성능을 대폭 높일 수 있는 숨겨진 부분들을 Airac에서 발견하고 있으며 새 AiracV의 설계와 구현에 반영중이다. 새 설계 중 일부를 구현한 현재의 AiracV로도 이미 Airac의 성능을 큰 폭으로 개선할 수 있음을 확인할 수 있다. 우리는 현존하는 정적 프로그램분석 기술들을 조화하여 앞으로 AiracV를 C 프로그램들과 같이 지점별 상태가 중요한 프로그램을 요약해석하기 위한 하나의 모범적인 틀로 발전시키려고 한다.

참고문헌

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, chapter Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, pages 85–108. Lecture Notes in Computer Science 2566. Springer-Verlag, 2002.

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, 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*, pages 196–207, New York, NY, USA, 2003. ACM Press.

[4] Patrick Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM*, 28(2):324–328, June 1996.

[5] Patrick Cousot and 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.

[6] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[7] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, London, UK, 1992. Springer-Verlag.

[8] 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 Igor Siveroni Chris Hankin, editor, *Static Analysis: 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005. Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 203–217. Springer-Verlag, September 2005.

[9] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[10] Youil Kim. 프로그래밍언어연구회 추계 학술대회 논문 응모. email to Kwangkeun Yi, 22 October 2005.

[11] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, Lecture Notes in Computer Science, pages 5–20. Springer-Verlag, 2005.

신 재 호

– 2000-2004 한국과학기술원 학사
– 2004-현재 서울대학교 석사과정

<관심분야> 프로그램 분석 및 검증, 프로그램 일반화 및 최적화

김 재 황

– 1992-1997 서울대학교 학사
– 1997-2003 (주)누리텔레콤
– 2004-현재 서울대학교 석사과정

<관심분야> 프로그램 분석 및 검증

오 학 주

– 2001-2005 한국과학기술원 학사
– 2005-현재 서울대학교 석사과정

<관심분야> 프로그램 분석 및 검증

정 영 범

– 1998-2004 서울대학교 학사
– 2004-현재 서울대학교 석사과정

<관심분야> 프로그램 분석 및 검증

이 광 근

– 1983-1987 서울대학교 학사
– 1988-1990 Univ. of Illinois at Urbana-Champaign 석사
– 1990-1993 Univ. of Illinois at Urbana-Champaign 박사
– 1993-1995 Bell Labs., Murray Hill 연구원
– 1995-2003 한국과학기술원 교수
– 2003-현재 서울대학교 교수

<관심분야> 프로그램 분석 및 검증, 프로그래밍 언어