Airac5 v1.0 User's Manual

RopasWork Inc.
Programming Research Laboratory
Seoul National University

December 2005

차 례

1 Airac5설치

1.1 시스템 요구 조건

Airac5는 Intel x86 프로세서를 가진 리눅스 서버에서 구동할 수 있으며 Airac5를 사용하기 위해서는 gcc가 설치되어 있어야 한다.

1.2 설치

Airac5는 컴파일된 실행파일만 배포 된다. Airac5홈페이지 http://ropas.snu.ac.kr/airac5 에서 내려 받아 설치할 수 있다. 내려 받은 파일이 airac5.tar.gz라고 할 때 다음의 방법으로 설치할 수 있다.

1단계 Airac5를 /usr/local에 설치하는 경우, 다음의 명령으로 설치가 가능하다:

```
$ tar xvzf airac5.tar.gz -C /usr/local
```

위 명령을 수행하고 나면 /usr/local/bin 디렉토리 밑에 Airac5 실행 프로그램인 airac5가 생기며, 실행에 필요한 부품들이 /usr/local/libexec/airac5/ 밑에 저장된다. 더불어, Airac5 문서 디렉토리 /usr/local/share/airac5/와 예제 C 프로그램 및 관련 데이타가 디렉토리 /usr/local/share/airac5/example/에 저장된다.

2단계 Airac5를 실행하기 위해서는 사용자 쉘의 PATH 환경 변수에 Airac5의 실행 파일이 있는 디렉토리를 지정해 줘야 한다.

- Bourne shell을 사용하는 경우

  ```
  $ PATH=$PATH:/usr/local/bin
  ```

- C shell을 사용하는 경우

  ```
  $ set path=($path /usr/local/bin)
  ```

위 명령을 각 쉘의 초기화 파일(Bourne shell은 .profile, C shell은 .cshrc)에 넣어 두면 편리하게 Airac5를 쓸 수 있다.

2 Airac5의 사용 예

Airac5는 ANSI C 언어의 문법을 따라 작성된 프로그램들만을 받아들이며 GNU C의 확장을 일부 지원한다. 주어진 프로그램이 제대로 분석이 되기 위해서는 gcc를 이용한 전처리가 가능해야 한다. 즉 C 프로그램의 전처리에 필요한 헤더 파일이 모두 있어야 함을 물론이고, C 프로그램 내에서 사용된 매크로 및 타입의 정의가 모두 알려져 있어야 한다. Airac5의 사용 방법을 다음의 사용예들을 통해 알아보자.

2.1 기본적인 실행 방법

Airac5는 이미 전처리가 끝난 소스 파일이나, 전처리를 할 수 있는 조건이 갖춰진 소스 파일만을 입력으로 받아들인다. 분석할 프로그램이 이런 조건을 만족한다면 다음의 예와 같이 분석을 할 수 있다.

옵션	설명
`-ud=`n	Unrolling depth로 n을 지정한다. 기본 값은 1 이다. 함수 호출을 호출될 함수의 바디로 바꿔치기하는 호출 단계의 수를 지정한다. 이렇게 함으로써 `-ud`로 지정된 단계의 동안의 함수 호출은 함수 펼치기(function inlining)의 효과를 얻는다.
`-ub=`n	Unrolling bound로 n을 지정한다. 기본 값은 0이다. 반복문의 바디를 `-ub`에 주어진 값만큼 순차적으로 펼친 코드로 변환한다. 따라서 반복문 펼치기(Loop unrolling) 효과를 얻는다.
`-ei=`n	Expected iteration number로 n을 지정한다. 기본값은 300이다. 정적 분석은 반복문의 반복 횟수를 정확하게 예측할 수가 없다. 따라서 Airac5는 반복문이 무한번 반복할 것이라고 가정하고 분석을 수행한다. `-ei`는 반복문이 최대 n 번 반복 후에 끝난다는 가정을 하도록 한다. 그러나 분석 중에 어떤 반복문이 n번 이상 수행될 수 있다고 판단되면 그 반복분은 무한번 반복될 거라는 가정을 한다.
`-cf`	`main()` 함수가 없는 경우 주어진 소스 코드에 정의된 모든 함수를 임의의 순서로 호출하는 `main()` 함수를 자동으로 생성한 후에 분석을 수행한다.
`-maxmem=`n	분석기가 사용할 수 있는 최대 메모리 사용량을 n MB로 지정. 기본 값은 시스템에 장착된 실제 메모리 크기이다. 지정된 크기 이상으로 메모리를 사용하는 경우 분석기는 그 때까지의 분석 결과를 바탕으로 경보를 출력한다.
`-h`	옵션 도움말 출력.

표 1: 명령줄 옵션

실행예 (기본적인 분석기 실행). Airac5의 실행은 옵션은 그림 1의 표에 나와 있으며 간단한 실행 예는 다음과 같다.

- `main()` 함수의 정의를 가지고 있는 `main.c` 파일 하나로 구성된 프로그램을 분석하는 경우에 다음의 명령으로 분석을 할 수 있다:

  ```
  $ airac5 main.c
  ```

  여기서 `main.c`는 "`gcc -o main.c`" 명령으로 컴파일이 가능한 소스 프로그램이어야 한다. 즉 특별한 헤더 디렉토리 지정이나 매크로 정의를 필요치 않는 소스 프로그램이어야 한다.

- C 프로그램 소스의 전처리를 위해 헤더 디렉토리나 매크로를 정의해야 경우에 `gcc`의 전처리 옵션인 `-I`, `-D`를 그대로 이용해 헤더 디렉토리나 매크로를 정의할 수 있다:

  ```
  $ airac5 -I /project/include -DTYPE=2 lib.c main.c
  ```

□

라이브러리나 리눅스 커널 같은 프로그램에는 `main()` 함수가 없다. 또 많은 수의 파일로 구성된 큰 C 프로그램은 시스템의 용량 상 전체를 한번에 분석할 수 없는 경우도 있다. 이런 경우 전체 프로그램을 연관성이 있는 몇 개의 파일들로 분할해서 분석할 필요가 있다.

실행예 (-cf 옵션 사용). 전체 10개 소스 파일로 구성된 프로그램을 10개 소스 파일 모두에 대한 분석을 하지 않고, 파일별로 따로 분석을 한다고 가정해 보자. 이 경우 1 개의 소스 파일에는 main() 함수가 있지만 나머지 9 개의 소스 파일에는 main() 함수가 있을 수 가 없다. Airac5는 main() 함수가 없는 소스 파일을 분석하는 경우를 위해 -cf 옵션을 제공한다. -cf 옵션을 사용하면 자동으로 파일들 내의 모든 함수를 호출하는 시나리오를 만들어 분석을 한다. 만일 분석할 파일 a.c, b.c에 세 함수 f(), g(), h()가 정의되어 있다고 하면 분석기는 각 함수를 호출하는 코드를 생성한 후 분석을 수행한다. 분석은 다음의 명령으로 실행한다:

```
$ airac5 -cf a.c b.c
```

□

main() 함수가 없을 때 분석할 프로그램이 어떻게 사용되는 지를 모방하는 시나리오 프로그램을 작성할 수도 있다. 시나리오 프로그램은 간단하게는 시작 함수의 호출만을 가지고 있을 수도 있고, 여러가지 초기 화에 이은 시작 함수 호출이 될 수도 있다.

실행예 (간단한 시나리오). program.c에 foo(), init() 함수가 정의되어 있고, global이라는 정수형 전역 변수가 있다고 하자. foo()가 호출 되기 전에 전역 변수 global을 100으로 초기화 하고, init()함수가 불려져야한다면, 시나리오 프로그램을 다음과 같이 만들 수 있다:

```
int main()
{
  global=100;
  init();
  foo();
  return 0;
}
```

위와 같은 시나리오 프로그램을 m.c에 저장했다면, 분석은 다음의 명령으로 실행한다:

```
$ airac5 program.c m.c
```

□

반복문은 분석의 정확도를 떨어뜨리기 쉽다. 왜냐하면 Airac5는 프로그램의 각 지점별로 프로그램의 실제 실행 중에 일어날 수 있는 상태를 모두 포괄하는 요약 상태를 계산해 내기 때문이다. 즉 어떤 지점이 여러번 실행될 수 있다면 그만큼 포괄해야 할 상태가 많아지고 그런 상태를 포괄하는 요약된 상태는 실제 상태에 대한 요약 정도가 커지기 때문이다.

실행예 (반복문 펼치기). 반복문을 가지고 있는 간단한 프로그램 loop.c를 분석한다고 하자. 정의는 다음과 같다:

```
1  int main()
2  {
3      int i, arr[10], *p;
```

```
4       p=arr;
5       for(i=0; i<10; i++) {
6           *p++ = 1;
7       }
8       return 0;
9   }
```

loop.c를 아래와 같은 명령으로 분석 하는 경우를 생각해 보자:

```
$ airac5 loop.c
```

Airac5는 프로그램의 각 지점에서 변수들이 가질 수 있는 값들을 포괄하는 값을 계산하는 분석을 한다. i는 for문의 진입부분의 i<10에 의해 $[0, 9]$를 값으로 가지는 것으로 분석된다. 그러나 포인터 변수 p는 for문의 진입부분에서 항상 이전보다 1 큰 값을 가지게 된다. Airac5은 p의 값을 안전하게 요약해 내기 위해 그 값이 배열 arr의 첫번째 원소 이후 모든 메모리를 가리킬 수 있다고 분석한다. 따라서 Airac5는 6번째 줄에서 버퍼 오버런 오류가 날 수 있다는 경보를 포함하고 있다:

```
loop.c:6:17: Offset: [0,+oo) Size: [10,10]
 6:          *p++ = 1;
```

그러나 loop.c의 코드를 실제로 실행해보면 6번째 줄에서 변수 p는 배열 arr의 10번째 이후를 가리키지 않는다. 그러므로 6번째 줄에 대한 버퍼 오버런 보고는 허위 경보(*false alarm*)이다. 이런 류의 허위 경보를 피하려면 프로그램의 상태를 기억할 위치를 늘려서 값들이 뭉쳐지지 않게 하면 된다. 이를 위해 Airac5는 반복문 펼치기(loop unrolling)를 지원한다. 반복문 펼치기는 반복문의 바디를 지정된 수만큼 복사 하여 매 반복에서 일어날 수 있는 상황을 별개로 분석한다. 반복문 펼치기는 다음과 같이 지정할 수 있다:

```
$ airac5 -ub=10 loop.c
```

위와 같이 -ub가 10으로 지정되면 6번째 줄에 대한 경보가 사라질 것이다. □

반복문과 같은 이유로 많이 사용되는 함수에 대한 분석의 정확도로 낮아지기 쉽다. 함수에 대한 분석 정확도를 향상 시키기 위해서는 함수 펼치기(function inlining)를 사용할 수 있다. Airac5는 함수 펼치기를 위한 옵션 -ud를 제공하고 있다.

2.2 정의가 없는 함수에 의미 주기

분석 대상 소스에서 사용된 함수의 정의가 모두 주어지지 않는 경우 분석의 정확도가 떨어질 수 있다. 프로그램을 분석할 때 그 프로그램에서 사용되는 라이브러리 함수 정의에 대한 소스 코드는 주어지지 않는 경우가 많다. 함수의 정의가 주어지지 않으면 함수 호출로 일어날 수 있는 상태의 변화를 제대로 포함하는 분석을 수행할 수가 없다. 분석 정확도가 정의가 없는 함수 때문에 저하된다고 의심이 되는 경우, 함수의 의미를 정의한 후 다시 분석을 시도할 수 있다.

실행예 (C 프로그램으로 함수 의미 정의하기). 그림 1의 (a)에 있는 foo.c를 분석하는 것을 생각해 보자. myalloc()이 파라미터의 값만큼의 메모리를 할당받고, 할당된 메모리의 시작 주소를 리턴하는 함수라고 하자. foo.c와 main()함수를 호출하는 코

```
int main()
{
    char *arr;
    arr=myalloc(10);
    arr[100]='a';
    return 0;
}
```
(a) `foo.c`

```
void * myalloc(int sz)
{
    return malloc(sz);
}
```
(b) `myalloc.c`

그림 1: `myalloc()` 함수의 정의

script	*script*	→	*library*$^+$
library	*library*	→	`lib` *id params overrun ret*
parameter	*params*	→	*params params*
		|	`param` *id*
alarm	*overrun*	→	*overrun overrun*
		|	`overrun` *e cond*
	cond	→	`always` | `when` *e*
return value	*ret*	→	`return` *e*
expression	*e*	→	*id* | *z* | `sizeof`(*e*) | `TOP`
		|	*e* `+` *e* | *e* `-` *e* | *e* `*` *e* | *e* `/` *e*
		|	*e* `<` *e* | *e* `>` *e* | *e* `<=` *e* | *e* `>=` *e* | *e* = *e*
		|	*e* `andalso` *e* | *e* `orelse` *e* | `not` *e*

그림 2: 스크립트 언어의 정의

드만 가진 `m.c`를 가지고 분석을 하면 분석 결과에는 버퍼 오버런에 대한 보고가 포함되지 않는다. 왜냐하면 `myalloc()`이 어떤 함수 인지를 Airac5는 알지 못하기 때문이다.

이 문제를 해결하기 위해 C 라이브러리 함수 `malloc()`을 이용하여 `myalloc()`의 의미를 Airac5가 알 수 있도록 정의해야한다. 그림 1의 (b)는 C 프로그램으로 `myalloc()`함수의 의미를 정의한 예를 보이고 있다. 원래의 `myalloc()`함수는 단순하게 메모리만 할당하고, 할당된 메모리의 주소를 리턴하는 것만으로 이뤄져 있지 않을 수 있다. 하지만 Airac5의 분석 정확도를 높이는데 `myalloc()`함수의 핵심적인 의미만 필요하다면 그림 1의 (b)에 있는 정의만으로도 분석 정확도 향상 효과는 충분할 것이다.

`myalloc()`함수의 정의를 이용한 `foo.c`의 분석은 다음과 같이 할 수 있다.

```
$ airac5 foo.c myalloc.c
```

□

Airac5는 함수의 의미를 정의하는 스크립트 언어를 지원한다. 스크립트 언어의 구문 정의는 그림 2와 같다. 이 언어를 사용하면 함수가 반환하는 값에 대한 간략한 의미와, 그 함수 내에서 버퍼오버런이 발생할 조건을 정의할 수 있다. 스크립트 언어로 함수를 정의하는 방법은 다음과 같다.

- 함수 선언부 : 함수의 이름과 인자를 정의한다.

  ```
  lib 함수명
  param 인자1
  param 인자2
  ...
  ```

  스크립트 언어에서는 각 함수를 함수의 이름과 인자의 수로 구분한다. 함수의 이름은 코드에서 직접 불려지는 이름을 사용하여야 하고, 불려질 때 전달될 인자의 수만큼 스크립트 함수 내부에서 사용될 인자를 정의해야 한다.

- 버퍼 오버런 발생조건 : 스크립트 함수 안에서 버퍼 오버런이 발생할 수 있는 경우를 정의한다.

  ```
  overrun 인자명 발생조건
  ```

  이는 발생조건을 만족하였을 때 인자로 전달된 해당 버퍼에서 오버런이 발생할 수 있다는 의미이다. 발생조건은 `when (수식)` 과 같이 정의하며, 어떠한 버퍼에서 항상 버퍼 오버런이 발생할 수 있는 경우에는 `always` 키워드를 사용한다. `always`는 `gets()`와 같이 이미 버퍼 오버런의 가능성이 알려진 위험한 함수를 표시하는데 유용하게 사용될 수 있다.

- 반환값 : 함수가 반환하는 값을 정의한다.

  ```
  return (수식)
  ```

  수식에 사용되는 변수는 이 함수에서 정의된 인자의 이름만을 사용할 수 있다. 즉, 수식 안에서 전역변수를 사용하거나 다른 함수를 호출할 수 없다.

메모리에 접근하는 기본적인 C 라이브러리 함수들에 대한 스크립트 함수들이 `example/cstd.sem`에 정의되어 있다. `sem` 확장자로 정의한 스크립트 함수는 아래와 같이 C 소스파일과 함께 입력하여 사용할 수 있다:

```
$ airac5 foo.c cstd.sem
```

실행예 (스크립트를 이용한 `memset()` 함수의 정의). C 라이브러리 함수 중 `memset()`의 실행 의미는 다음과 같이 정의할 수 있다:

```
lib memset
param buffer
param value
param n
overrun buffer when (sizeof(buffer) < n)
return buffer
```

이 스크립트의 의미는 다음과 같다. memset()함수는 세개의 인자(buffer, value, n)을 받는다. 만일 buffer의 크기보다 n의 값이 크면 버퍼 오버런이 발생할 수 있다. memset()은 buffer에 저장된 값을 리턴한다. □

2.3 분석 힌트 주기

분석의 안전성과 분석의 종료 두 가지 조건을 만족하기 위해 분석기는 안전하기는 하지만 실제와 크게 동떨어진 분석을 하는 경우가 있다. 예를 들어 반복문이 있는 경우 반복 횟수를 정확하게 예측하는 것은 불가능하므로 분석기는 반복문이 무한번 돌 수도 있다는 가정을 하면서 분석을 할 수 밖에 없다. 또 프로그램이 외부로부터 값을 읽어 들이는 경우에 어떤 값이 읽혀질 지는 실제 실행을 해 보기 전에는 알 수 없으므로 분석기는 모든 가능한 값이 될 수 있다는 가정을 할 수 밖에 없다. 이런 가정들로 인해 안전성은 유지를 하지만 정확도는 상당히 떨어지는 분석 결과가 나올 수 있다.

Airac5는 분석기가 과도하게 안전한 가정으로 인한 정확도 저하를 막기 위해 사용자가 분석기에게 분석의 힌트를 줄 수 있는 방법을 가지고 있다. 힌트 제공 방법은 C 의 assert()문과 비슷하다. Airac5는 airac_assert(*e*)를 제공한다 . airac_assert(*e*)의 실행 의미는 분석기가 요약한 메모리를 e가 참이되는 가장 정확한 메모리로 변경하는 것이다. 다음의 예를 살펴 보자.

실행예 (단정문 사용하기). 다음은 외부에서 정수 값을 읽어서 배열을 초기화 하는 프로그램이다:

```
1: int main()
2: {
3:     int a[10];
4:     int i, n = read_int();
5:
6:     for(i=0;i<n;i++)
7:         a[i] = 0;
8:     return 0;
9: }
```

분석기는 실제의 상황을 모두 포섭해야 하므로 read_int()가 $(-\infty, +\infty)$를 리턴한다고 가정하고 분석을 진행한다. 그 결과로 7번째 줄에서 버퍼 오버런이 발생할 수 있다는 경보를 출력하게 된다. 하지만 만일 read_int()가 항상 특정 범위의 값만을 리턴한다는 것이 보장된다면 이 사실을 분석기에게 알려줘서 보다 정확한 분석을 할 수 있도록 하면 좋을 것이다. 만일 read_int()가 0에서 10사이의 값만을 리턴한다면, 이 사실을 5번째 줄에 다음과 같은 코드를 넣음으로써 분석기에 전달할 수 있다:

```
5: airac_assert(0<=n && n<=10);
```

4번째 줄에서 n은 $(-\infty, +\infty)$의 값을 가지는 것으로 분석된다. 그러나 5번째 줄에 위의 코드를 넣으면 n은 0이상 10이하를 만족하는 최소 값, 즉 $[0, 10]$의 값을 가지는 것으로 변경된다. □

airac_assert(*e*)의 e에는 제한된 형태의 C 프로그램 식만이 올 수 있다. 우선 Airac5는 e에 '||'이 오는 것을 허용하지 않는다. "0>=n || n>= 10"과 같은 조건식은 n의 값을 한정하는 데

전혀 도움이 될 수 없다. 왜냐하면 이런 조건을 만족하는 n의 값은 $(-\infty, +\infty)$이기 때문이다. 또 "n=1 || n=2"는 n을 $[1, 2]$의 값을 갖도록 하며 이는 "n>=1 && n<=2"로 표현할 수 있다. 또 Airac5의 단정문은 변수에 관한 1차식만을 지원한다. 그러므로 airac_assert(x*y+z>3)과 같은 형태의 단정문은 분석의 정확도를 향상시키는데 기여할 수 없다.

2.4 분석기 강제로 끝내기

Airac5는 분석할 프로그램에 따라 많은 양의 메모리를 사용할 수도 있고 몇 시간씩 실행될 수도 있다. 시스템에 메모리가 부족하거나 분석 결과를 빨리 얻고 싶을 때는 분석기를 필요한 시점까지 실행 한 후 종료 시킬 필요가 있다. Airac5는 시스템의 용량이나 사용자의 필요에 따라 적절한 선에서 분석을 마치고 그 때까지 분석한 결과를 토대로 경보를 발생 시킬 수 있다.

-maxmem=n 옵션은 Airac5이 사용할 수 있는 최대 메모리 크기를 지정한다. 기본적으로는 시스템의 실제 메모리 크기로 n이 결정된다. Airac5는 사용 메모리 양이 n MB에 도달하면 분석을 중단한다.

Airac5의 분석을 끝내기 위해 시그널을 이용할 수도 있다. Airac5는 SIGQUIT 시그널을 받으면 분석을 중단하고 그 때까지의 분석을 바탕으로 경보를 출력한다. 리눅스에서 시그널은 Ctrl–\나 kill -QUIT 명령으로 발생시킬 수 있다.

3 경보 해석

분석결과는 표준 출력(stdout)으로 다음과 같이 출력된다. 경보는 함수별로 출력된다. 먼저 함수에 포함된 경보 수가 출력된다. 그 다음에는 Airac5가 찾안낸 오류 지점 정보로 파일이름, 위치정보(라인, 컬럼), 인덱스 정보(offset), 버퍼의 정보가 출력된다. 그 다음 줄에는 알람이 발생한 실제 소스코드상의 라인을 보여준다. 마지막에는 경보를 포함하고 있는 함수를 호출한 함수들을 2 단계까지 보여준다:

```
In procedure f():
 Number of alarms: 1
 sample.c:239:15: Offset: [0,+oo) Size: [32,32]
  239:          arr[i] = 1;
 Callers: f <- {g,h} <- {foo}
```

배열의 크기 및 인덱스는 모두 정수 범위 값으로 표현된다. 위의 예는 arr 변수가 가리키는 배열의 크기는 200인데, 이 배열 참조에 사용되는 인덱스 값은 0에서 ∞의 값을 가질 수 있으므로 버퍼 오버런이 발생할 수 있다는 의미이다. 안전한 분석을 위해 실제의 값을 요약하는 과정에서 위와 같이 $[0, \infty)$의 값을 가질 수 있는 것으로 요약되는 경우도 있다. 분석 결과 해석에서 주의할 점은 $[0, \infty)$ 값을 가진다고해서 항상 허위경보를 의미하는 것은 아니라는 것이다.

<끝>