Guest
Article

C/C++ 언어의 정적 분석 기술에 대한 이해

정적 분석 기술은 C/C++ 프로그램에서 컴파일 시에 테스트 케이스가 없어도 심각한 오류를 찾을 수 있게 한다. 전체 프로그램에 대한 프로시저간 분석을 통해 예를 들면 버퍼 오버런(overrun), 널 포인터 역참조(dereference), 초기화를 하지 않은 변수의 사용과 같은 주요한 문제들을 찾아내는데 초점을 맞추고 있다. 이 기술을 지원하는 제품들은 빌드 환경에 변화를 주지 않고 사용할 수 있어서 매우 쉽게 셋업 할 수 있다. 또한 사용자가 별도의 입력을 할 필요도 없다. 커다란 프로젝트도 분석할 수 있는데 리눅스 커널에도 적용되었고, 수 천만 라인의 C/C++ 프로그램에도 적용 가능하다.

이 기술을 사용하면 소프트웨어의 오작동 감소, 개발 지연의 감소, 안정성의 향상과 같은 이점이 있다. 이는 프로그램이 이유 없이 죽는 버그나 또는 죽지는 않더라도 문제를 야기하는 골치 아픈 버그들을 찾을 수 있게 해준다. 이를 통해 소프트웨어의 품질을 향상하고 비용을 줄일 수 있게 해 준다.

자료제공: MDS테크놀로지㈜
www.mdstec.com

정적 분석 기술의 배경

정적 분석 기술은 소프트웨어의 정적 표현(representation)에 근거해 소프트웨어의 동작에 대한 정보를 추론한다. 이것은 소프트웨어가 동작할 때 이를 관찰하여 정보를 수집하는 동적 분석 기술과 대비된다. 코드가 실행되지 않은 상태에서 분석하므로 테스트 케이스가 필요 없다.

정적 분석은 두 단계 과정을 거친다. 첫 번째 단계에서는 소프트웨어 소스 코드로부터 의미 정보(semantic information)를 추출한다. 두 번째 단계에서는 이 정보를 이용해 결함이나 원하는 다른 특성을 찾아 낸다.

최근까지 정적 분석 도구는 실용적이지 않았다. 예를 들면 린트(Lint)와 같은 단순한 도구들은 코드의 피상적인 구조를 파악하는 정도였다. 이는 코딩 표준을 따랐는지 확인하고 몇 가지 피상적인 검사를 할 수는 있었지만 심각한 문제를 찾아 낼 수 있을 만큼 강력하지 않았다. 또한 수 많은 거짓 양성(false positive, 실제로는 문제가 없음에도 불구하고 문제로 인식되는) 경고를 발생해 출력을 검토하기가 어려웠다.

반면에 매우 정교한 모델 분석 도구들도 있었는데 이들은 시스템의 순서 특성을 확인할 수 있지만 소스코드에 직접 적용하기가 어렵고 시스템을 상당히 난해한 언어로 기술한 추상적 모델에서 동작한다. 모델 확인 도구는 아직까지는 확장성이 떨어진다. 최신 정적 분석 도구들은 이러한 장애를 극복하고 커다란 프로그램에 대해 매우 정교한 분석을 할 수 있게 되었다.

정적 분석 기술의 투자 대비 효과(ROI)

정적 분석의 효과는 기존의 테스팅 보다 더욱 많은 실행 경로를 검사하며 개발 초기에 적용할 수 있다. 전통적인 테스팅은 코드가 실행될 때에만 확인이 가능해

서 테스트 케이스를 얼마나 잘 만들었는가에 의존했다. 실제 시스템에서는 테스팅에서 실행되는 경우의 수보다 훨씬 더욱 많은 경로로 실행될 수 있다. 테스트 스위트(suite)는 100%의 서술 범위(statement coverage)를 달성할 수 있으나 대부분의 경로가 실행되지 않을 수 있다. 반면에 정적 분석 도구들은 테스트 케이스가 작성되지 않은 경로에 대해서도 분석을 한다.

이 때문에 정적 분석 도구들은 이미 충분히 테스트를 거친 소프트웨어에 대해서도 많은 버그를 발견할 수 있게 해준다. 또한 소프트웨어의 보안성을 향상시켜 주는데 이는 많은 소프트웨어의 취약성이 코딩 결함에서 비롯되기 때문이다. 해커들은 종종 코너 케이스 프로그램 동작(corner-case program behavior)을 악용하는데 이는 테스트 스위트가 놓치기 십상이다. 정적 분석 도구는 이러한 취약점을 잘 찾아낸다.

정적 분석은 개발 초기에 버그를 찾아낼 수 있어서 문제를 고치기가 쉽기 때문에 시간과 비용을 절감하게 해 준다. 미국 국립 표준 기술원이 2002년에 소프트웨어에 대한 광범위한 연구를 한 바에 따르면 코딩 결함은 출시된 제품에서 발견된 결함을 고치기 위해서는 코딩 과정에서 발견된 결함을 고치는 것보다 10배에서 30배 정도의 노력이 든다고 한다.

따라서 나중에 발견될수록 고치는데 비용이 들어가므로 코딩 직후, 시스템 테스팅 전에 가능한 빨리 버그를 찾는 것은 매우 중요하다.

IBM의 Hailpern과 Santhanam이 연구한 바에 따르면 전체 소프트웨어 개발 비용의 50%에서 75%가 디버깅, 테스팅, 확인 작업에 소요되기 때문에 이러한 비용 절감 효과는 대단히 크다.

정적 분석을 이용하면 수작업으로 버그를 찾을 경우 오랜 시간이 걸리는 문제들을 빠르게 찾아낼 수 있다. 개발자들은 수작업에 의한 디버깅 블랙홀에 빠지지 않고 좀 더 중요하고 즐거운 일에 몰두할 수 있다. 또한 자연스럽게 제품 개발의 위험이 감소한다.

GrammaTech의 CodeSonar

GrammaTech은 저명한 Wisconsin 대학과 Cornell 대학의 컴퓨터 공학과 교수들에 의해서 설립되었고 직원들은 정적 분석 분야의 박사 학위를 보유하고 있다. Lockheed Martin, Northrop Grumman, BAE Systems, GE Aviation, LG 전자, 삼성전자, 퀄컴, 파나소닉, Kawasaki, 미국 식품의약청(FDA), 나사(NASA) 등이 CodeSonar를 사용하여 소프트웨어 품질을 향상시키고 비용을 절약하고 있다. CodeSonar는 GrammaTech에 의해 개발된 대표적인 정적 분석 도구이다.

GrammaTech CodeSonar는 컴파일 과정에서 분석을 하며 사용자가 프로그램 일부에 대해 적용할 수도 있지만 대개 전체 소프트웨어 프로젝트에 적용된다. CodeSonar는 빌드 과정에서 함께 동작하며 별도의 빌드 스크립트를 필요로 하지 않는다.

CodeSonar는 컴파일러처럼 코드를 파싱(parsing)하는 방식으로 동작하지만 컴파일러처럼 오브젝트 코드를 만들지는 않고 대신에 프로그램에 대한 추상적인 표현을 만든다. 개별적인 파일이 빌드되면 합성 단계에서 그 결과를 전체 프로그램 모델로 합성 한다. 이 모델은 프로시져 간 분석의 집합으로 탐색되며 결함이 발견되면 경고를 발생시킨다.

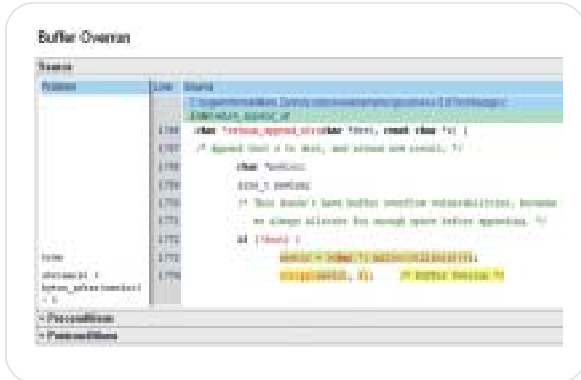
주 분석은 GrammaTech의 Smash Proof™ 분석엔진에 의해 이루어진다. CodeSonar는 기존 개발 과정과 빌드 환경 그대로 동작한다. 결과는 이해하기 쉽고 거짓 양성(false positive) 발생률이 낮다. 사용자 인터페이스는 웹기반으로 되어 있고 결과 데이터베이스는 팀원들이 함께 검토할 수 있으며 경고 리포트를 통합적으로 관리할 수 있다.

이렇게 쉽게 사용할 수 있으며 다양한 프로그램이 죽는 결함과 죽지 않는 결함 모두를 찾아내고 새로운 검사기를 정의하여 분석 항목을 확장할 수도 있다.

각각의 경고는 그 결함이 발생하는 경로를 보여주며 추적 내용을 포함한다. 따라서 사용자는 경고의 내용을 이해하고

문제를 재빨리 해결할 수 있다.

아래는 버퍼 오버런의 예제로 오픈 소스 프로젝트를 분석한 것이다.

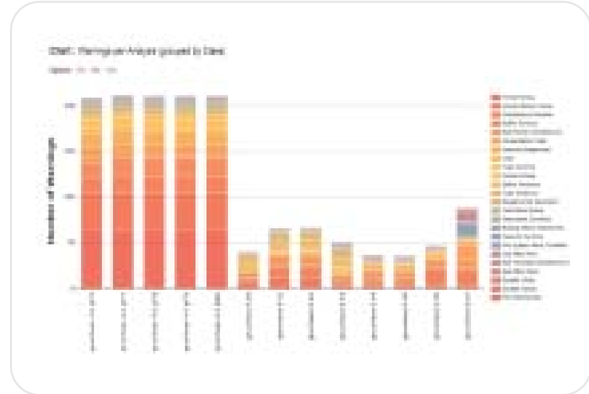


이 버퍼 오버런은 1860번째 줄에서 발생한다. 1859번째 줄이 잘못되어 있는데 "+1"이 잘못된 위치에 있어 메모리 할당량이 잘못 계산되었다. 두 번째 오류는 malloc()의 리턴 값이 확인되지 않은 것이다.

팀간 협업을 용이하게 하기 위해 경고는 데이터베이스에 저장되어 웹 서버에 의해 관리된다. 분석, 데이터베이스, 웹 컴포넌트 등은 여러 컴퓨터에 분산될 수 있다. 검색 인터페이스가 제공되어 사용자는 특정한 기준에 부합되는 결과의 위치를 찾을 수 있다. 경고의 검토자를 지정할 수도 있다. 문장 정보와 코멘트는 경고에 첨부되어 분석을 여러 번 하는 동안 코드가 변경되더라도 유지될 수 있다.

전체 데이터는 차트로 만들어 팀원들이 품질이 어떻게 변하는지에 대해 이해하는데 도움을 줄 수 있다. 아래의 샘플 차트는 소프트웨어 프로젝트에서 각 버전 별로 발견된 경고를 종류별로 나누어 개수를 보여준다.

다섯 번째와 여섯 번째 분석 사이에서 경고의 숫자가 현저히 감소하였는데 이는 코드의 전체적으로 수정한 것과 일치한다. CodeSonar는 미리 만들어진 차트를 제공하며 아울러 차트 위저드가 있어 개인적으로 원하는 차트를 원하는 형태로 만들 수 있다.



CodeSonar® 의 동작 원리

어떤 정적 분석 도구들은 코드를 퍼지 파싱하는데 이 경우 프로그램 동작에 대한 모델이 불완전하며 결과의 정확성이 떨어진다. 반면에 CodeSonar는 코드를 컴파일러가 분석하는 것처럼 정확하게 분석하여 정밀한 프로그램 모델을 생성한다. CodeSonar는 컴파일 단위 별로 전처리기 지시자(preprocessor directive), 헤더 파일, 매크로 등을 정확하게 파싱한다.

이 단계에서 추상적인 문법 트리, 심볼 테이블 정보, 호출 그래프(call graph), 제어 흐름 그래프 등을 만들어 낸다. 개별 컴파일 단위에 대한 파싱이 완료되면 링크가 이루어진다. 이는 오브젝트 파일들을 링크하는 것과 유사한데 단지 CodeSonar는 오브젝트 코드 대신에 프로그램의 요소들에 대한 추상적인 표현을 링크하는 것이 다르다. 예를 들면 각 프로시저는 제어 흐름 그래프(CFG, control-flow graph)를 가진다. 링크에서 이 CFG들은 전체 프로그램에 대한 커다란 CFG로 통합된다.

다음 단계에서 CodeSonar는 프로그램 경로에 대한 프로시저간 탐색을 시행한다. 이것이 분석에서 핵심적인 부분으로 모든 가능한 경로, 프로그램 변수, 어떻게 이것들이 서로 관련되는지에 대한 추론을 하게 된다. CodeSonar는 함수, 데이터 형(type), 데이터 등에 대한 정보를 추출하기 위해 전

체 프로그램에 대해 선형 경로 탐색을 실시 한다.

이후, 각각의 프로시저에 대해 CodeSonar는 프로시저의 동작과 부작용에 대한 모델을 만든다. 그 후, 프로시저간 경로 탐색이 시행된다. 탐색은 경로와, 문맥(context), 그리고 오브젝트에 따라 이루어진다. 경로 탐색에서 이상이 발견되면 경고가 발생된다. CodeSonar에 의해 유지되는 추상적인 표현은 상당이 커질 수 있어서 여러 가지 다양한 전략을 사용해서 확장성을 확보한다.

예를 들면, 분석과정에서 프로시저 집합을 정제하고 축약하고, 페이징을 최소화 할 수 있도록 경로를 탐색 한다. 성능을 향상시킬 수 있는 다른 기술은 탐색에서 가능성이 희박한 프로그램 경로를 제거하기 위해 발전된 데이터 흐름 분석 정보를 사용하는 것이다. 이것은 거짓 양성(false positive)을 감소시키므로 정확성을 향상시키는 데에도 도움을 준다.

검사

CodeSonar에 의해 발견되는 결함은 아래와 같이 세 가지로 구분된다.

* 언어의 잘못 사용: 이 오류는 C/C++ 언어 자체의 특성을 잘못 사용하는데 따른 것이다. 예를 들면 버퍼 오버런, 널 포인터 역참조, new에 의해 생성된 오브젝트를 free 하는 것 등이다.

* 라이브러리 잘못 사용: 이 에러는 표준 라이브러리 API를 잘못 사용하는데 따른 것이다. 예를 들면 메모리 외의 리소스 누설(leak), 잘못된 상태에서 socket에 대해 accept()를 호출하는 것 등이다.

* 사용자 정의 오류: 사용자가 CodeSonar의 분석을 확장해서 특별한 검사를 시행할 수 있다. 예를 들면 “인터럽트 핸들러에서 foo()를 호출하지 않는다” 와 같은 것이 있다. Foo()는 특정 시스템에서 인터럽트 핸들러에서는 호출이 금지된 함수와 같은 것이 될 수 있다.

이 글의 맨 마지막에는 CodeSonar에서 검사하는 항목들의 리스트를 확인할 수 있다.

정적 분석 도구를 평가할 때에는 한 가지 형태의 결함이 다양한 방법으로 발생할 수 있다는 것을 인식하는 것이 중요하다. 정적 분석 도구의 품질은 어떤 버그 한 가지가 여러 가지 다양한 형태로 나타나더라도 이를 찾을 수 있는가에 달려 있다. 예를 들면, 아래 두 가지는 모두 버퍼 오버런에 해당하는 결함으로 사실 한 바이트를 벗어나는 오류이다. 하지만 오른쪽 버그가 훨씬 더 복잡하며 발견하기 위해서는 훨씬 더 깊이 있는 분석을 필요로 한다.

```
char buf[10];
buf[10] = 'c';
```

```
int foo(int size) {
    char * p;
    p = malloc(size);
    if(p==NULL)
        return ERROR;
    bar(p, size);
    free(p);
        return 0;
    }
    void bar(char *p, int size) {
        int i;
        for (i=0; i<=size; i++)
            p[i] = 'c' ;
    }
```

실제 코드에서 버퍼 오버런은 종종 위의 예제 보다 훨씬 더 복잡하다. 어떤 도구들은 많은 수의 피상적인 버그 검사기를 가지고 있다. 반면에 CodeSonar는 깊은 분석을 통해 낮은 거짓 양성(false-positive)율을 유지하면서도 복잡한 버그들은 검출할 수 있다.

예를 들면 Maryland 대학의 William Pugh 교수와 York 대학의 David Hovemeyer 교수는 여러 가지 정적 분석 도구들이 널 포인터 역참조를 얼마나 잘 찾아내는지에 대해 연구를 했다(사실 널 포인터 역참조를 찾아내는 것은 통계적으로

매우 어려운 것으로 알려져 있다). Pug 교수와 Hovemeyer 교수는 연구에서 구조체를 사용하는 매우 복잡한 버그를 포함시켰다. 그런데 CodeSonar는 연구 대상에 포함된 모든 널 포인터 역참조를 찾아낸 유일한 도구였다.

또한 CodeSonar 만이 거짓 양성(false positive) 경고를 하나도 만들지 않았다(Hovemeyer 교수와 Pugh 교수의 2007년 연구).

위에서 언급한 바와 같이 CodeSonar는 사용자가 검사기의 종류를 확장할 수 있는 풍부한 API를 포함하고 있다. 이 API들은 가능한 다양한 검사를 지원할 수 있도록 구현 되어 있는데 사실 CodeSonar에 포함된 검사기들도 이 API를 이용해 구현되었다. 예를 들면 아래와 같다.

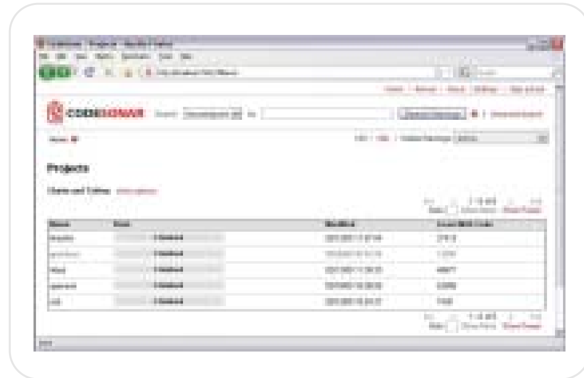
· (커스텀 자원 할당기를 위한 custom resource allocators) 누설(leak) 검사기
· 허용되지 않은 매개변수 값을 찾는 검사기
· 시간상 성질을 검토하는 검사기
· 복잡한 프로그래밍 관용구에 충실하게 코딩 되었는지를 확인하는 검사기

커스텀 검사기는 코드 베이스를 변경하지 않고도 구현할 수 있다.

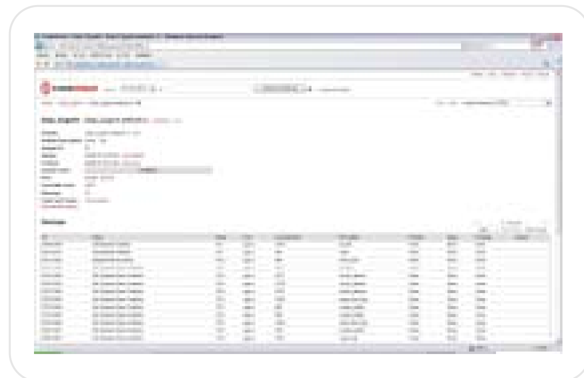
결과 검토 및 관리

CodeSonar는 CodeSonar 웹서버와 데이터베이스로 구성된 hub에 의해 관리된다. 분석 결과는 hub에 저장되고 실제 분석은 다른 컴퓨터에서 실행될 수도 있다. 빌드를 하기 위해 사용되는 컴퓨터에서 동작하는 CodeSonar 분석 데몬(daemon)은 결과를 hub에 업로드 한다. 사용자는 웹브라우저를 사용해서 경고 보고를 볼 수 있다. 사용자는 CodeSonar에 로깅 한 후에 아래와 같은 화면을 보게 된다.

이 화면은 분석중인 프로젝트를 보여주고 있다. 각각의 프로젝트는 관련된 몇 개의 분석을 동시에 진행할 수 있는데 이는 사용자가 대개 코드를 바꾸고 새로운 결함을 찾기 위해 분석을 다시 실행하기 때문이다.



일단 사용자가 프로젝트와 검토할 분석 결과를 선택하면 아래와 비슷한 화면이 표시된다.



"분석 페이지"라고 불리는 이 화면은 분석 결과에서 만들어진 각각의 경고를 한 줄에 하나씩 보여 준다. 각 열은 아래와 같다.

· ID: 경고 번호로 hub에 의해 유일한 값이 주어진다.
· Class: 경고의 클래스로 "누설", "버퍼 오버런", "널 포인터 역참조", "초기화 되지 않은 변수" 등이 있다.
· Rank: 검토 등급
· Priority: 경고에 할당된 우선 순위. 이는 사용자에 의해

서 정해 지는데 "High", "Medium", "Low", "Suppress", 또는 사용자가 정한 다른 값이다.

· State: 경고의 상태. 이는 사용자에 의해 정해 지며 검토를 위해 개발자에게 할당이 되었다는 의미의 "Assigned" 또는 다른 값이다.

· Finding: 경고에 대해 사용자가 확인한 결과. 사용자에 의해 정해 지며 각 경고에 대해 확인 결과 "진짜 양성(true positive)" 이었는지 아니면 "거짓 양성(false positive)" 이었는지를 표시해 놓기 위해 사용될 수 있다.

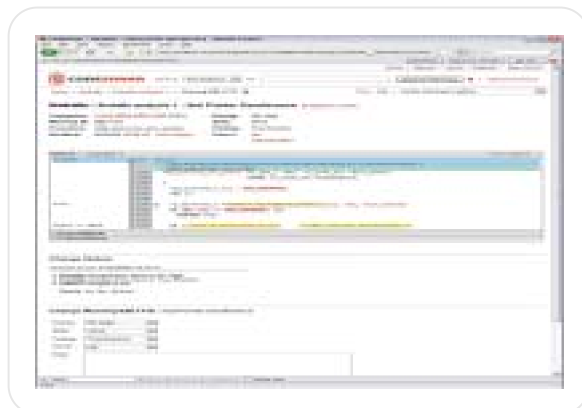
경고와 관련된 정보는 코드가 변경되고 실행을 반복해도 남아 있다.

분석 페이지의 줄을 클릭하면 해당 버그에 대한 경고 보고서 페이지로 연결된다. 예를 들면 널 포인터 역참조에 대한 경고 보고서 페이지는 아래와 같다.

경고의 중간 부분은 버그가 발생하는 실행 경로를 보여 준다. 다양한 색깔을 사용하여 경로와 경고 내용을 쉽게 이해할 수 있도록 한다.

1809번째 줄에 있는 re_acquire_state 호출은 NULL을 반환하고 변수 state를 NULL로 셋팅하게 된다.

이것이 경우 1813번째 줄의 state의 역참조는 널 포인터 역참조이다. 1810번째 줄에서 오류 검사를 하고 있지만 NULL 포인터가 반환되는 경우는 오류 코드가 re_acquire_state에 의해 셋팅 되지 않는다. 1809번째 줄의 + 심볼을 클릭하면 사용자는 re_acquire_state의 내부 코드로 연결되고 경로를 볼 수 있게 된다. 이것은 함수의 코드를 볼 수 있게 해준다.



경고 보고 페이지의 아래 부분은 사용자에 의해 설정된다. 이 경고는 사용자 Joe에게 검토가 의뢰 되었다는 것을 보여준다. Joe는 이 경고 보고를 높은 우선 순위로 설정했고 코멘트를 추가했다.

다른 사용자들은 그 경고와 관련된 정보를 볼 수 있다. 그 정보는 분석을 반복해도 남아 있다.

CodeSonar는 사용자들이 특정 범주에 부합되는 경고들을 찾아 내고 관리하는 것을 도와 주기 위해 검색 기능을 제공한다.

예를 들면, 사용자는 어떤 종류 경고와 혹시 그것이 어떤 특정 파일들에서 발생하는 가에만 관심이 있을 수 있다. 검색 결과는 더 많은 범주를 추가하여 더 상세하게 걸러내고 저장할 수 있다.

단계적 분석 능력

프로젝트의 첫 번째 분석은 많은 프로그램 경로를 탐색해야 하므로 다소 시간이 걸린다. 하지만 기본 분석이 끝난 후의 후속 분석은 훨씬 빠르다. 후속 분석에서 CodeSonar는 새로운 프로그램 경로와 기존 경로 중에서는 변경된 코드에 의해서 영향을 받을 가능성이 있는 것들 만을 탐색한다.

거짓 양성(false positive)의 억제

일반적인 크기의 프로그램을 분석하는 경우에 모든 정적 분석 도구들은 확장성을 보장하기 위해 적절히 타협점을 찾아야 한다. 예를 들면 프로그램의 가능한 데이터 상태에 대해 추정을 해야 한다. 추정을 하게 되면 결과의 정확성에 악영향을 끼치게 된다. 이 때 분석에 대해 어떤 가정을 하느냐에 따라 거짓 음성(false negative)이나 거짓 양성(false positive)을 만들어 낼 수 있다. 거짓 양성(false positive)은 그 경로가 실제

로 발생할 가능성이 없거나 그 버그가 발생하기 위해서 필요한 값을 실제로는 가질 수 없기 때문에 실제 버그나 아닌데도 경고를 하는 것이다. 거짓 음성(false negative)은 버그임에도 보고가 되지 않는 것이다.

거짓 양성(false positive)을 너무 많이 만드는 정적 분석 도구는 개발자가 결과를 검토하는데 너무 많은 시간을 필요로 하므로 채택하기가 어렵다. CodeSonar는 매우 낮은 거짓 양성(false positive) 비율을 가지고 있는데 이를 위해 아래와 같은 기법들을 사용한다.

· 발전된 데이터 흐름 분석을 사용해서 불가능한 경로를 찾아내어 이것들은 탐색하지 않음

· 공통적인 프로그램 관용구와 프로그래머의 의도를 이해

· 사용자가 CodeSonar에 아주 중요한 함수의 행동을 알려 줄 수 있는 메커니즘을 제공(예를 들면, 프로그램 실행이 my_exit()을 호출하는 부분에서 종료된다). 이 메커니즘을 사용하는데 코드를 변경할 필요는 없다.

그럼에도 불구하고 거짓 양성(false positive)이 나타나는 사용자가 코드를 변경하거나 주석을 붙이지 않고 거짓 양성(false positive)으로 태그를 붙인다. CodeSonar는 이 후 실행에서부터는 그런 것들은 거짓 양성(false positive)으로 보고하지 않는다. CodeSonar의 거짓 양성(false positive) 비율은 버그 클래스에 따라 다르고 소프트웨어 프로젝트에 따라 다르다. 또한, 어떤 경고가 진짜인지 여부는 사용자의 관점에 따라 다를 수도 있다. 아래 예를 보자.

· Malloc()에 의해 반환된 포인터의 값이 사용되기 전에 NULL인지 검사되지 않는다. 사용자는 이러한 경우에 대해 염려할 수 있다(왜냐하면 이것은 가용한 메모리가 남아 있지 않다는 것을 나타내기 때문이다). 그리고 이것을 정말 문제라고 생각한다. 반면에 사용자는 시스템에서 malloc()이 메모리 할당을 못하는 경우가 발생한다면 어떤 다른 심각한 문제가 있는 것이고 프로그램은 어차피 죽을 것이기 때문에 시스템은 그러한 상태에 빠지면 절대 안 된다는 관점을 가질 수도 있다. 사용자는 이러한 경고를 하거나 또는 하지 않도록 CodeSonar를 설정할 수 있다.

· 버퍼 오버런은 너무 많은 데이터를 수신하게 되면 발생한다. 수신된 데이터의 양은 외부 환경에 의해 결정된다. 어떤 사용자에게 이에 대한 경고는 진짜 문제일 수 있지만 다른 사용자에게는 환경이 제어 가능해서 버퍼 오버런이 발생하는 시나리오를 예방할 수 있다.

· 어떤 코드는 도달 불가능하다. 어떤 사용자는 이러한 코드는 제거 되어야 한다고 생각할 수도 있지만 다른 사용자는 그것이 별 문제 아니라고 생각할 수 있다.

위와 같이 경계 영역에 속하는 것들은 종종 발생한다. 이러한 형태의 경고들 대다수는 사용자가 CodeSonar를 설정해서 사용자가 원하는 대로 보이게 하거나 안보이게 할 수 있다.

거짓 음성(false negative)을 피하는 방법

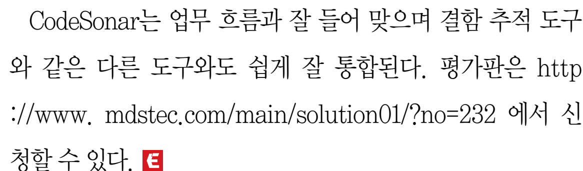
거짓 양성(false positive)을 최소화 하는 것이 중요하지만 또한 거짓 음성(false negative, 보고 되지 않는 진짜 버그)을 최소화 하는 것도 똑같이 중요하다. 결국 버그를 찾아내는 것이 정적 분석의 요점이다. 잘못하면 거짓 양성(false positive)을 줄이는 데에 너무 몰두한 나머지 거짓 음성(false negative)이 대폭 늘어나는 상황을 초래할 수 있다. CodeSonar는 이러한 일이 발생하지 않도록 매우 조심스럽게 설계되어 있어 관리 가능한 수준에서 거짓 양성(false positive) 비율을 유지하면서 매우 중요한 결함들을 찾아낼 수 있다.

참고: 대표적인 검사 항목

버퍼 오버런/언더런, 타입 오버런/언더런, 널포인터 역참

조, 0으로 나눔, 이중 free(), free() 후 사용, 힙 변수 아닌 것을 free(), 변수를 초기화 하지 않음, 누설, 위험한 함수 형 변환, malloc() 또는 new 로 생성된 오브젝트를 Delete[] 하려고 함, new[] 또는 new 로 생성된 오브젝트를 free하려고 함, Return 문을 빠뜨림, 어떤 조건문이 항상 만족되거나 절대 만족될 수 없음, 지역 변수에 대한 포인터 또는 free()한 포인터를 반환, 사용되지 않는 값, 무의미한 할당, 서로 다른 파라미터나 반환 값을 갖는 함수 포인터를 Varargs 함수 포인터로 형전환 시도, Return 값 무시, 널 포인터를 free(), 도달 불가능한 코드, 역참조 후 널 테스트, 특정 함수 인자로 포맷 스트링을 가져야 하는 함수가 포맷 스트링이 아니거나 신뢰할 수 없는 스트링을 전달 받음(보안 취약성 발생 가능), 이중 close(), TOCTTOU 취약성, 이중 lock ()/unlock(), 성공할 수 없음에도 mutex에 대해 lock()을 시도, 잘못 사용된 메모리 할당/복사, 라이브러리의 잘못 사용, 무한 루프 등이다.

CodeSonar는 밤에 빌드하는 과정에서 실행하여 새로운 결함을 초기에 발견하고 적은 비용으로 수정하는 방식으로 운용될 수 있다.



CodeSonar는 업무 흐름과 잘 들어 맞으며 결함 추적 도구와 같은 다른 도구와도 쉽게 잘 통합된다. 평가판은 http://www. mdstec.com/main/solution01/?no=232 에서 신청할 수 있다. E

TECHWORLD, INC.

21여 년 간 쌓아온 테크월드의 광대하고 풍부한 콘텐츠는 지금까지 경험해보지 못했던 다양한 정보의 스펙트럼을 누리게 해줍니다.

대한민국 전자산업의 역사와 함께 해온
(주)테크월드는 항상 여러분 가까이에 있습니다.
테크월드는 잡지 뿐만 아니라
온라인 웹서비스, 전시회, 컨퍼 런스를 통해
언제나 기술 한국을 응원하는 전문 미디어 중심회사입니다.

Tel (02)2026-5700(代) Fax (02)2026-5701
www.techworld.co.kr