사례연구를 통한 정적 프로그램 분석 기법을 사용하는 도구의 비교

(A Comparative Case Study on Static Program Analysis Tools)

김 윤 호 † 박 용 배 †
(Yunho Kim) (Yongbae Park)

김 문 주 ††
(Moonzoo Kim)

요 약 프로그램 신뢰성 향상을 위해서 정적 프로그램 분석 도구가 많이 사용되고 있다. 하지만, 정적 분석 결과에서 거짓 경보를 걸러내기 위한 추가적인 분석이 필요하고 실제 오류를 찾지 못할 수 있는 문제점이 있다. 본 논문에서는 상용 정적 프로그램 분석 도구인 Coverity와 Sparrow 및 오픈 소스 도구인 Clang analyzer를 buffer overflow 벤치마크와 libexif 0.6.20 소스코드에 적용하여 실험 대상 도구가 오류를 찾아내는데 얼마나 효과적인지 사례 연구를 수행하였다. 실험 결과 buffer overflow 벤치마크에서는 Coverity, Sparrow, Clang analyzer 각각 6.02%, 3.61%, 0%의 버그 탐지율과 2.41%, 1.20%, 0%의 오탐율을 보였다. libexif 0.6.20 적용에서 Coverity, Sparrow, Clang analyzer는 총 745개의 오류를 찾아내었으나 알려진 7개의 오류 중 6개를 찾아내지 못하였다. 이와 같이 정적 분석 도구가 놓칠 수 있는 오류가 많으므로 높은 소프트웨어 신뢰성 보장을 위해 다른 기법과 병행하여 적용하는 것이 필요하다.

키워드: 정적 분석, 사례 연구, Coverity, Sparrow, Clang analyzer

Abstract Static analysis tools have been widely used to improve software reliability. However, static analysis tools have limitations that can generate false alarms and can miss real bugs. This paper presents a comparative study of three static analysis tools, Coverity, Sparrow, and Clang analyzer, through a case study on a set of buffer overflow benchmark programs and libexif 0.6.20 to evaluate the effectiveness of the three static analysis tools in terms of a bug finding capability. Experiment results on buffer overflow benchmark show that bug detection capabilities of Coverity, Sparrow, and Clang analyzer are 6.02%, 3.61%, and 0%, respectively and false alarm ratios of Coverity, Sparrow, and Clang analyzer are 2.41%, 1.20% and 0%, respectively. Experiment results on libexif 0.6.20 show that Coverity, Sparrow, and Clang analyzer detect 745 bugs and miss 6 out of 7 real bugs. Thus, it is necessary to use other bug finding techniques with static analysis techniques for achieving high software reliability, because static analysis tools can miss real bugs in target programs.

Keywords: static analysis, case study, coverity, sparrow, clang analyzer

1. 서 론

소프트웨어의 활용 범위가 넓어짐에 따라 소프트웨어의 신뢰성이 더욱 중요해지고 있다. 특히 의료장비, 우주탐사 장비 등의 다양한 종류의 기기가 소프트웨어로 구동되면서 소프트웨어의 오류로 심각한 사회, 경제적 피해 및 인명피해가 발생한다. 예를 들어 아리안 5호 로켓은 64bit 부동소수점을 16bit 정수형으로 변환하던 중 발생한 오류로 폭발하여 5억 달러의 피해를 남겼다[1]. 또 다른 예로 Therac-25라는 방사선 의료기기의 소프트웨어 오류로 인해 환자가 필요 이상의 많은 방사선에 노출되어 사망하는 사고가 있었다[2]. 이러한 피해를 막기 위해서 소프트웨어의 신뢰성을 높이기 위한 노력이 필요하다.

소프트웨어 신뢰성을 효율적으로 높이기 위한 방법으로 정적 프로그램 분석 기법이 주목받고 있다. 기존에 소프트웨어의 신뢰성 향상을 위해 표준적으로 사용되는 소프트웨어 테스트는 개발자가 테스트 케이스를 수동으로 생성하고 그 결과를 분석해야 하기 때문에 비효율적이다. 정적 프로그램 분석 기법은 자동화된 방식으로 프로그램을 분석하여 프로그램이 비정상적으로 종료되는 오류와 overflow오류 등을 찾을 수 있다.

· 본 연구는 미래부가 지원한 2013년 정보통신·방송(ICT) 연구개발사업, 지식경제부/한국산업기술평가관리원 IT R&D 프로그램[10041752, 초소형·고신뢰 OS와 고성능 멀티코어 OS를 동시 실행하는 듀얼 운영체제 원천 기술 개발], 그리고 미래부/한국연구재단의 중견연구자지원사업-핵심연구(2012046172)의 연구비 지원으로 수행되었음
· 이 논문은 제39회 추계학술발표회에서 '정적 프로그램 분석 기법을 사용하는 도구의 비교:Coverity와 Sparrow를 사용한 libexif 사례 연구'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : KAIST 전산학과
kimyunho@kaist.ac.kr
yongbae2@gmail.com
†† 종신회원 : KAIST 전산학과 교수
moonzoo@cs.kaist.ac.kr
(Corresponding author임)

논문접수 : 2013년 1월 21일
심사완료 : 2013년 5월 10일

본 논문에서는 상용 정적 분석 도구 Coverity[3]와 Sparrow[4] 및 오픈 소스 도구 Clang analyzer[5]를 사용하여 사례 연구를 통해 버그 찾는 능력 및 오탐율을 비교하였다. 사례 연구는 크게 두가지로 진행하였다. 먼저 기존 프로그램의 buffer overflow 버그를 기반으로 생성된 소규모(100~1000LOC) 벤치마크 프로그램을 대상으로 버그 찾는 능력과 오탐율을 비교하고 규모 있는(9KLOC) 오픈소스 라이브러리인 libexif 0.6.20을 대상으로 적용하여 발생한 경보 및 발견한 오류를 비교하였다. 실험 결과 buffer overflow 벤치마크에서는 Coverity, Sparrow, Clang analyzer 각각 6.02%, 3.61%, 0%의 버그 탐지율과 2.41%, 1.20%, 0%의 오탐율을 보였다. libexif 0.6.20 적용에서 Coverity, Sparrow, Clang analyzer는 총 745개의 오류를 찾아내었으나 알려진 7개의 오류 중 6개를 찾아내지 못했다.

2. 정적 프로그램 분석

정적 프로그램 분석[6,7]은 프로그램을 실제로 실행시키지 않고 분석하여 프로그램 소스코드를 실행하였을 때 나타나는 오류를 찾아내는 방법이다. 분석 방법은 프로그램이 실행 가능한 경로, 프로그램의 변수가 가질 수 있는 값의 범위 등을 추상화하여 분석하고 가능한 실행 경로나 값의 범위가 프로그램 오류가 발생하는 조건을 만족하는 지 검사하여 오류가 발생하는 부분을 찾아내는 것이다. 정적 프로그램 분석 방법을 사용하여 동적으로 할당한 메모리 자원, 0으로 나누는 연산, 잘못된 메모리 접근과 같은 오류를 효과적으로 찾아낼 수 있다.

정적 프로그램 분석은 프로그램을 실행시키지 않고 분석하기 때문에 실제 프로그램이 수행될 만큼 개발이 진행되지 않아도 개발 도중에 적용할 수 있고, 프로그램을 실제 실행하여 분석하는 방법인 동적 프로그램 분석에 비해서 짧은 시간에 넓은 범위를 분석할 수 있는 장점이 있다. 또한 분석 과정이 자동화되어 있어서 개발자가 큰 노력 없이 적용할 수 있다. 이와 같은 장점에 힘입어 Coverity, Sparrow, Clang analyzer와 같은 다양한 도구들이 개발되어 사용되고 있다. 하지만, 정적 프로그램 분석 기법은 프로그램이 가질 수 있는 무한한 크기의 실행 경로와 프로그램 상태를 추상화하여 유한한 크기 내에서 분석하기 때문에 이 과정에서 정확도가 떨어져 거짓 경보를 발생할 수 있다. 거짓 경보가 발생할 경우 개발자가 직접 경보를 분석하여 실제 버그인지 아닌지 판별해야 하기 때문에 거짓 경보가 많이 발생할 경우 정적 분석을 통한 개발 비용 절감보다 거짓 경보 분석에 따르는 개발 비용 증가가 더 커질 수 있다. 따라서 거짓 경보를 줄이기 위한 다양한 관련 연구가 진행되고 있다[8]. 또한, 여러 정적 프로그램 분석 도구가 효율적으로 buffer overflow 오류를 찾아내는지를 비교하는 연구도 있다[9]. 본 논문에서는 buffer overflow 오류 외에도 두 도구가 찾아낼 수 있는 모든 오류를 분석하여 분류하였다.

3. 실험 환경

3.1 도구 설정

Coverity 5.4와 Sparrow 4.7 도구는 경보가 발생하는 수준을 조절할 수 있으며 Coverity의 경우 3단계가, Sparrow에는 5단계가 있다. 낮은 단계에서는 거짓 경보를 줄이기 위해 최소의 경보를 발생시키고, 높은 단계에서는 오류를 놓치지 않기 위해 최대한 많은 경보를 발생한다. 하지만 두 도구에서 검사할 수 있는 오류의 종류와 분석 강도가 서로 다르기 때문에 완전히 동일한 설정에서의 비교는 어렵다. 따라서 본 논문에서는 경보 분석을 위한 시간을 고려하여 두 도구가 수백 개 수준의 경보를 발생할 수 있는 경보 발생 단계를 적용하여 비교하였다. Clang analyzer 3.3 도구는 경보 발생 수준을 조절하는 옵션이 없어 기본 설정으로 수행하였다. 각 도구는 Intel Core i7 3.4GHz 프로세서와 8GB의 메모리가 장착된 Debian 서버에서 실행하였으며 모든 분석은 5분 내에 종료되었다.[1)]

3.2 Buffer overflow 벤치마크 실험 설정

각 도구의 오류 찾는 능력과 오탐율을 비교하기 위해 buffer overflow 벤치마크[9]를 대상으로 적용하여 결과를 비교하였다. buffer overflow는 C 프로그램에서 자주 발생하는 버그로 발생 시 프로그램 비정상 종료 및 심각한 보안 버그를 야기할 수 있는 치명적인 버그이다. 벤치마크 프로그램은 오픈 소스 프로그램 BIND, Sendmail, wu-ftpd에서 기존에 발견된 버그를 재현하기 위한 코드로 작성되어 있으며 크기는 90~800LOC이다. 총 14쌍의 테스트 케이스로 구성되어 있으며 각 테스트 케이스는 buffer overflow 버그가 있는 BAD 버전과 해당 버그를 수정한 OK 버전 한 쌍으로 구성되어 있다. 각 BAD 버전은 최소 1개에서 최대 28개, 평균 5.9개의 buffer overflow 버그를 발생시킬 수 있는 소스 코드 라인을 포함하고 있다.

Buffer overflow 벤치마크 실험에서는 미리 지정된 테스트 케이스의 오류와 관련되지 않은 경보는 무시하였다.

3.3 libexif 0.6.20 실험 설정

분석 대상인 libexif는 Exchangeable Image File Format(EXIF)[10] 형식의 이미지 파일에 포함된 정보를 읽고 쓰기 위해 사용한다. libexif 0.6.20은 C 언어소스코드 파일 24개와 30개의 헤더 파일로 구성되어 있

1) 본 논문은 2012 한국컴퓨터종합학술대회 추계학술발표회에 발표된 "정적 프로그램 분석 기법을 사용하는 도구의 비교: Coverity와 Sparrow를 사용한 libexif 사례 연구" 논문을 확장한 것으로 기존 논문에서 적용한 Sparrow의 성능을 개선한 새 버전의 Sparrow를 사용하여 실험을 수행하고 결과를 분석하였다.

표 1 Buffer overflow 벤치마크 실험 결과
Table 1 Experiment results of buffer overflow benchmark

	Bug detection capability	False alarm ratio
Coverity	6.02%	2.41%
Sparrow	3.61%	1.20%
Clang analyzer	0.00%	0.00%

표 2 libexif 0.6.20에서 발생한 경보
Table 2 The number of generated alarms for libexif 0.6.20

	Coverity	Sparrow	Clang -analyzer
Real bugs	147 (84%)	599 (95%)	3 (100%)
False alarms	28 (16%)	33 (5%)	0 (0%)
Total alarms	175 (100%)	632 (100%)	3 (100%)

으며, 8,922줄이 들어있으며 함수의 개수는 229개이다. libexif를 사용한 이유는 실제 널리 사용되고 있는 규모 있는 오픈 소스 라이브러리이며 알려진 오류와 찾지 못한 오류가 모두 있어서 각 정적 분석 도구가 얼마나 효과적으로 오류를 찾아내는지 확인할 수 있기 때문이다. 공개된 버그 리포트를 확인하고 libexif의 최신 버전인 0.6.21 버전에서 오류를 고친 부분과 비교하면 0.6.20에서 오류가 발생하는 부분을 찾을 수 있다.

4. 실험 결과

4.1 Buffer overflow 벤치마크 실험 결과

표 1은 buffer overflow 벤치마크 실험 결과를 보여준다. Bug detection capability는 BAD 버전의 전체 83개의 buffer overflow를 야기할 수 있는 소스 코드 라인 중에서 각 도구가 실제 오류로 찾아낸 비율을 나타내고 false alarm ratio는 OK 버전에서 수정된 83개의 소스 코드 라인 중에서 오류가 있다고 경보가 발생한 수의 비율을 나타낸다. Coverity의 경우 6.02% (=5/83)의 bug detection capability와 2.41%의 false alarm ratio(=2/83)을 보였으며 Sparrow의 경우 bug detection capability는 3.61%(=3/83)로 Coverity보다 낮았으나 false alarm ratio가 1.20%(=1/83)로 Coverity보다 낮아 오탐이 적게 발생하였다. Clang analyer의 경우 모든 테스트 케이스에서 경보를 발생시키지 않았다.

Buffer overflow 벤치마크 실험 결과 Coverity와 Sparrow 의 buffer overflow bug detection capability는 7% 미만으로 낮은 것으로 나타났다. 상용 정적 분석 도구들은 수 십만~수 백만 줄 규모의 프로그램을 분석하면서 유용한 결과를 주기 위해 오탐율을 낮추는 다양한 휴리스틱 알고리즘을 적용한다. 이 과정에서 실제 오류인지 확실하지 않은 것으로 보일 경우 오류가 아닌 것으로 간주하여 bug detection capability를 낮추더라도 오탐을 줄이기 위해 최대한 노력한다. 따라서 정적 분석 도구는 프로그램이 올바르게 작성된 것을 증명하는 도구가 아니기 때문에 정적 분석 도구가 경보를 발생시키지 않더라도 숨어있는 버그가 있을 수 있으며 높은 신뢰성을 달성하기 위해 정적 분석 도구에만 의존하지 말고 concolic testing과 같은 동적 분석 기법과 상호 보완하여 적용해야 한다.

4.2 libexif 0.6.20 실험 결과

표 2는 Coverity, Sparrow, Clang analyzer로 libexif를 분석하여 발생한 경보의 수를 나타낸다. Coverity, Sparrow, Clang analyzer는 각각 175, 632, 3개의 경보를 발생시켰고 세 도구가 모두 공통적으로 발견한 경보가 1개 있었다. Coverity와 Sparrow는 세 도구가 모두 찾은 경보 외에도 2개의 동일한 경보를 발생시켰으며 Coverity와 Clang analyzer 역시 모두 찾은 경보 외에 2개의 동일한 경보를 발생시켰다. Sparrow와 Clang analyzer는 모두 찾은 경보 외에 동일한 경보를 발생시키지 않았다.

그림 1은 경보의 종류별로 각 도구 별 각각 발생시킨 경보 및 공통적으로 발생시킨 경보를 나누어 비교한 결과이다. 경보는 원인에 따라서 8가지 분류의 17종류로 구분하였다.

그림 2, 3, 4는 각각 Coverity, Sparrow, Clang analyzer가 찾은 실제 오류와 거짓 경보를 비교한 것이다. 각 도구가 찾아낸 오류와 찾지 못한 오류가 있고 경보의 종류에 따른 실제 오류와 거짓 경보 비율을 알 수 있다.

4.2.1 찾아낸 실제 오류

세 도구는 총 745개의 실제 오류를 찾아내었으며, 그 중 세 도구가 공통적으로 찾은 오류는 1개이다. 찾아낸 오류 중 0.6.21에서 고친 오류도 존재하며 고쳐지지 않은 오류도 존재한다.

그림 5는 Coverity, Sparrow, Clang analyzer가 공통으로 발견한 null pointer dereference 오류이다. exif-loader.c 파일의 `exif_loader_get_buf()` 함수 413번째 줄에서 포인터 `loader`가 `NULL`일 경우 조건이 참이 되어 414 번째 줄을 실행한다. 하지만 414번째 줄 에서 `exif_log()` 함수의 첫 번째 인자에서 `loader`를 참조하므로 null pointer dereference 오류가 발생한다. 해당 오류는 0.6.21 버전에서 아직 수정되지 않았다.

이처럼 정적 프로그램 분석 기법을 사용하면 개발자가 찾기 힘든 오류까지 찾을 수 있으므로 정적 프로그램 분석으로 소프트웨어의 신뢰성을 향상시킬 수 있다.

4.2.2 발견한 경보 종류의 비교

그림 1을 봤을 때 Coverity의 경우 null pointer dereference를 포함하여 총 12종류의 경보를 발생시켰으며 그 중 null pointer dereference를 제외한 11종류의 경보는 Sparrow가 발생시키지 않은 경보이다. 반면 Sparrow의 경우 null pointer dereference, bad integer constant

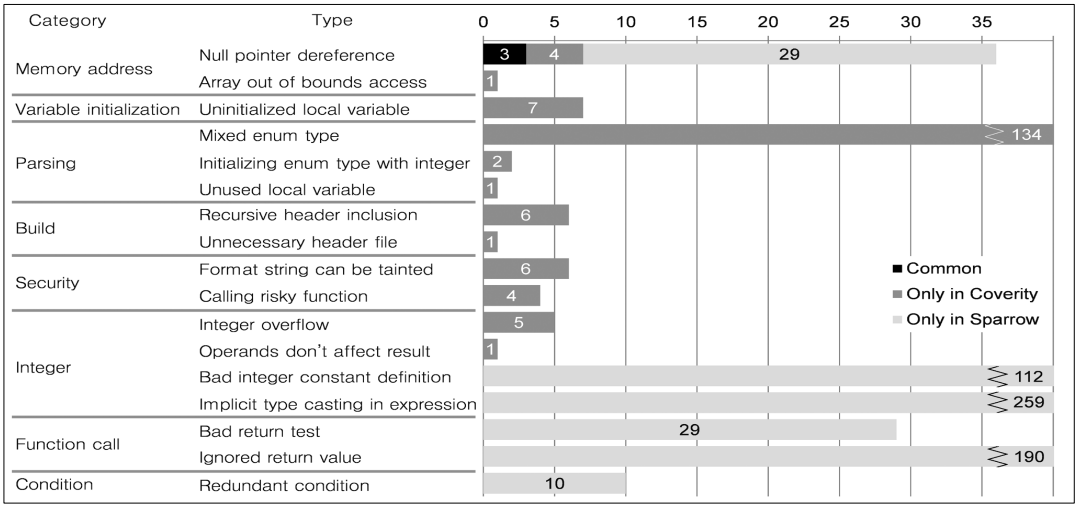

그림 1 각 도구가 생성한 경보 비교
Fig. 1 Alarms generated by Coverity, Sparrow, and Clang analyzer

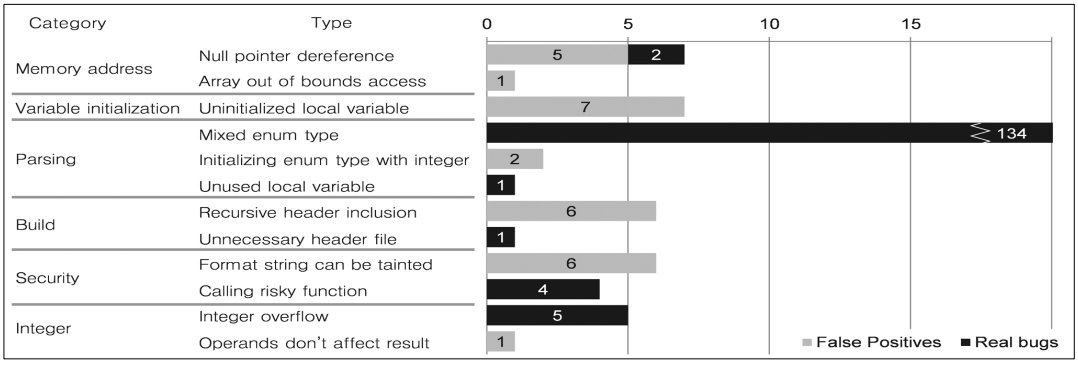

그림 2 Coverity가 찾은 실제 오류와 발생된 거짓 경보
Fig. 2 Real bugs and false alarms generated by Coverity

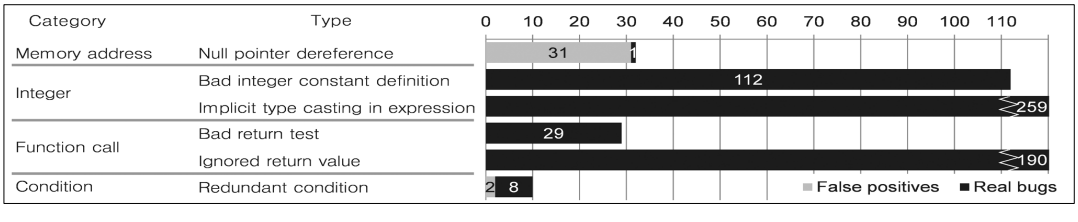

그림 3 Sparrow가 찾은 실제 오류와 발생된 거짓 경보
Fig. 3 Real bugs and false alarms generated by Sparrow

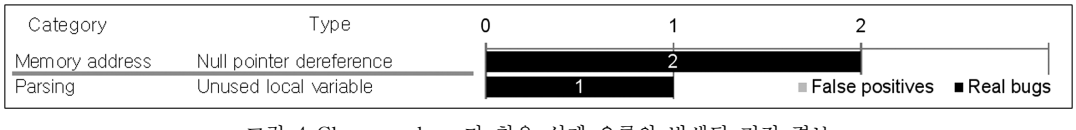

그림 4 Clang analyzer가 찾은 실제 오류와 발생된 거짓 경보
Fig. 4 Real bugs and false alarms generated by Clang analyzer

```
406 void
407 exif_loader_get_buf (ExifLoader *loader, const
408 unsigned char **buf, unsigned int *buf_size)
409 {
    ...
413    if (!loader || (loader->data_format ==
    EL_DATA_FORMAT_UNKNOWN)) {
414      exif_log (loader->log, EXIF_LOG_CODE_DEBUG,
    "ExifLoader","Loader format unknown");
```

그림 5 exif-loader.c에서 발생하는 null pointer dereference 오류

Fig. 5 Null pointer dereference error in exif-data.c

```
371 vr = exif_get_rational (entry->data,
    entry->order);
372 r = (double)vr.numerator / vr.denominator;
```

```
 80 #define R2D(n) ((n).denominator ?
    (double)(n).numerator/(n).denominator : 0.0)
    ....
377 vr = exif_get_rational (entry->data,
    entry->order);
378 r = R2D(vr);
```

그림 6 mnote-olympus-entry.c에서 0.6.20(위)과 0.6.21(아래)의 차이점

Fig. 6 Difference of mnote-olympus-entry.c between 0.6.20(above) and 0.6.21(below)

definition 등 총 6종류의 경보를 발생시켰으며 그 중 null pointer dereference를 제외한 5종류의 경보는 Coverity가 발견하지 못한 경보이다. Clang analyzer는 null pointer dereference와 unused local variable 두 종류의 경보를 발생시켰다. 이와 같이 각 도구는 오류를 찾기 위한 경보를 발생시키는 방식이 달라서 한 도구가 찾아낸 오류를 다른 도구에서 찾아내지 못할 수 있으며, 최대한 많은 오류를 찾아야 한다면 여러 가지 도구를 함께 사용하는 것이 효과적이다.

4.2.3 찾아내지 못한 오류

0.6.21에서 수정된 7개 오류 중 1개를 찾아내었으나 나머지 6개는 찾지 못하였다. 그 중 하나는 libexif 내부의 동작과 관련된 오류로, 정적 프로그램 분석으로 찾을 수 없으며, 나머지 오류는 배열 범위를 벗어난 접근 오류 3개, buffer overflow 오류 1개, 0으로 나누는 오류 1개이다.

Coverity와 Sparrow는 0으로 나누는 오류를 찾아내는 기능이 있지만 libexif에 있는 오류는 찾아내지 못하였다. 그림 6은 0으로 나누는 오류가 존재하는 0.6.20과 그 오류가 수정된 0.6.21을 비교한 것이다. `vr.denominator`라는 변수는 `exif_get_rational`에서 파일의 특정 부분의 값이 대입되며 0이 대입될 가능성이 존재한다. 따라서 변수가 0인지 확인해야 하며, 0.6.21 버전의 80번 줄에 `vr.denominator`가 0인지 확인한 후에 `vr.denominator`로 나누는 매크로가 추가되어 378번 줄에서 이를 사용하도록 소스코드가 변경되었다. 이 오류는 concolic execution 기법을 사용하는 도구인 CREST-BV는 찾아내는 오류이며 0.6.21에서 수정된 7개 오류 가운데 2개 오류를 CREST-BV를 사용해서 발견하였다[11]. 이처럼 정적 프로그램 분석 기법만으로는 모든 오류를 찾아낼 수 없으므로, concolic execution과 같은 다른 기법을 함께 사용하는 것이 필요하다.

5. 결론 및 향후 연구

사례 연구를 통해 정적 프로그램 분석 도구를 비교하였다. 각 도구 모두 libexif 0.6.20의 실제 오류를 발견하였으며 따라서 정적 프로그램 분석 도구는 소프트웨어 신뢰성을 향상시키는데 유용하다. 하지만 buffer overflow 벤치마크 결과가 보여주듯이 오탐율을 줄이기 위해 낮은 버그 탐지율을 보일 수 있고 각 도구가 찾을 수 있는 오류 종류가 서로 다를 수 있기 때문에 소프트웨어 신뢰성을 높이기 위해서는 여러 정적 분석 도구를 같이 활용하고 concolic execution과 같은 다른 분석 기법을 사용하여 오류를 찾아내는 것도 필요하다.

References

[1] J. Jézéquel, B. Meyer, "Design by contract: The lessons of ariane," *IEEE Computer*, vol.30, no.1, pp.129-130, 1997.

[2] N. Leveson, C. Turner, "An investigation of the therac-25 accidents," *IEEE Computer*, vol.26, no.7, pp.18-41, 1993.

[3] Coverity [Online]. Available: http://www.coverity.com (downloaded 2013, Jul. 28)

[4] Sparrow [Online]. Available: http://www.fasoo.com/product/sparrow_overview.asp (downloaded 2013, Jul. 28)

[5] Clang analyzer [Online]. Available: http://clang-analyzer.llvm.org (downloaded 2013, Jul. 28)

[6] P. Emanuelsson, U. Nilsson, "A comparative study of industrial static analysis tools," *Proc. of the 3rd International Workshop on Systems Software Verification (SSV 2008)*, 2008.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," CACM, vol.53, no.2, pp.66-75, 2010.

[8] K. Yi, J. Kim, Y. Jung, "Design and Implementation of Static Program Analyzer Finding All Buffer Overrun Errors in C Programs," *Journal of KIISE : Software and Applications*, vol.33, no.5, pp.508-524, 2006.

[9] M. Zitser, R. Lippmann, T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," *Proc. of the SIGSOFT '04/FSE-12*, 2004.

[10] JEITA. Exif.org [Online]. Available: http://www.exif.org/ (downloaded 2013, Jul. 28)

[11] Y. Kim, M. Kim, Y. Kim, Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE," *Proc. of the Software Engineering (ICSE), 2012 34th International Conference,* SEIP track, 2012.