Concolic 테스팅 기법을 구현한 KLEE 테스팅 도구의 사례 연구

김영주 ○ 김윤호 김문주

한국과학기술원

jerry88@cs.kaist.ac.kr kimyunho@kaist.ac.kr moonzoo@cs.kaist.ac.kr

Case Study on Testing with KLEE Concolic Testing Tool

Young-Joo Kim○ Yun-Ho Kim Moon-Zoo Kim

Korea Advanced Institute of Science and Technology

요 약

다양하고 많은 소프트웨어가 개발되고 있는 현 시장에서 소프트웨어의 질을 향상시키기 위해 적은 비용으로 효과적으로 신뢰성을 검증할 수 있는 다양한 테스팅 기법이 활발히 연구되고 있다. Concolic (Concrete+symbolic) 테스팅 또는 동적 심볼릭 수행(dynamic symbolic execution) 기법을 적용한 테스팅 도구가 많이 구현되고 있는데, 본 논문에서는 많은 Concolic 테스팅 도구 중 KLEE에 대한 특징을 살펴보고, 단위 시간당 분기 커버리지(branch coverage)를 살펴보는 사례 연구를 하였다. KLEE에서 제공하는 효과적으로 분기 커버리지를 높이는 역할을 하는 여러 탐색 방법(search strategy)들의 성능을 검증하기 위해 본 논문에서는 GNU Coreutils 버전 8.9를 대상 프로그램으로 하여 각 탐색 방법들의 분기 커버리지를 비교하였다.

1. 서 론

오늘날의 정보사회에서 소프트웨어의 중요성이 점차 커지고 있다. 로켓이나 전투기 등의 대형기기에서부터 메모리 칩이나 이동전화와 같은 소형기기까지 다양한 종류의 소프트웨어가 활용되고 있다. 이렇게 소프트웨어의 역할이 중요시됨에 따라 소프트웨어의 신뢰도를 향상시키기 위한 테스팅 기법에 대한 필요성이 증가하고 있다. 그러나 규모가 크고 복잡한 소프트웨어에서 개발자들이 여러 상황에서의 테스트 케이스를 수동적으로 도출하는 데 한계가 있다. 따라서 이와 같은 한계를 극복하기 위해서 자동으로 테스트 케이스를 적은 비용으로 도출해내는 기법에 대한 연구가 활발히 진행되고 있다.

Concolic 테스팅 기법 또는 동적 심볼릭 수행(dynamic symbolic execution) 기법은 동적 테스팅과 심볼릭 수행을 함께 적용시킨 기법이다[1]. Concolic 테스팅 기법은 모든 수행 경로를 커버하는 것을 목적으로 하는데, 이것은 현실적으로 불가능하므로 일정 분기 커버리지(branch coverage)에 도달하면 그 소프트웨어에 대한 신뢰성 검증이 되었다고 판단한다. 그러므로 주어진 시간 안에 높은 분기 커버리지를 달성하는 테스트 케이스를 도출하는 것이 중요하고, 이를 위해 다양한 탐색 방법(search strategy)이 많이 제시되었다. Concolic 테스팅 기법을 구현한 도구가 여러 가지가 있는데 그 중 KLEE는 LLVM(Low Level Virtual Machine)[2]을 수정하여 실시간으로 LLVM bitcode로 컴파일 된 대상 프로그램을 인터프리트하여 concolic 테스팅 기법을 수행하도록 하는 도구다[3]. KLEE는 오픈 소스로 제공되고 여러 가지 다양한 탐색 방법을 포함하고 있다는 장점이 있다.

이 논문에서는 KLEE 테스팅 도구의 각 탐색 방법의 성능 차이를 비교, 분석하기 위해 GNU Coreutils라는 대상 프로그램을 설정하여 각 탐색 방법 별로 대상 프로그램들을 테스트하여 결과를 도출하고 분석하였다.

2. 관련 연구

Concolic 테스팅 기법은 실제 수행(concrete execution)으로 부터 심볼릭 경로 공식(symbolic path formula)을 도출하는 방법에 따라 세 가지로 나뉜다. 첫 번째로, 대상 프로그램을 정적으로 instrument 하는 테스팅 도구다. 이 카테고리의 테스팅 도구는 probe 를 삽입하여 실시간으로 실제 수행을 하여 심볼릭 경로 공식을 도출해 낸다. CREST[4], CUTE[5], DART[6] 등이 이 카테고리에 해당한다. 두 번째로, 대상 프로그램을 동적으로 instrument 하는 테스팅 도구다. 이 카테고리의 concolic 테스팅 도구는 바이너리 대상 프로그램이 메모리에 로드되어 있을 때 instrument 하여 소스 코드 없이 자동 테스팅이 가능하다. SAGE[7] 등의 테스팅 도구가 이 카테고리에 해당한다. 세 번째로, 가상 머신에서 instrument 하는 테스팅 도구다. 이 카테고리의 concolic 테스팅 도구는 가상 머신을 수정하여 대상 프로그램 테스팅에 최적화 하도록 구현되어 있다. KLEE[3]는 LLVM 바이너리를 대상으로

구현되었고, PEX[8]는 Microsoft .Net 바이너리로 컴파일되는 C# 프로그램을 대상으로 한다.

Concolic 테스팅 기법은 점차 산업계에서 중요성이 부각되어 임베디드 시스템 소프트웨어에 적용하기 시작하고 있다. [9]에서는 플래시 메모리 플랫폼 소프트웨어 중 멀티 섹터 읽기 연산을 수행하는 함수에 concolic 테스팅 기법을 적용하여 성능 측정 및 장단점을 분석 하였다. [10]에서는 모바일 플랫폼의 일부 프로그램에 concolic 테스팅 기법을 적용하여 발견한 버그들을 분석하고, Concolic 테스팅 기법의 실제 소프트웨어 적용에 대한 어려움과 현재 제시된 테스팅 도구의 한계에 대해 분석하였다.

3. KLEE 테스팅 도구

KLEE 는 심볼릭(symbolic) 프로세스에 대한 운영체제와 인터프리터의 역할을 동시에 수행한다. 심볼릭 프로세스는 실제 concolic 테스팅의 실행 경로를 따라가며 경로 조건(path condition) 을 저장하는 역할을 하는 concolic 테스팅을 위한 프로세스로, 이것을 state 라고 표현한다. 각 프로세스는 스택, 힙, 프로그램 카운터, 경로 조건 등의 정보를 가지고 있다. 대상 프로그램은 LLVM assembly language 를 이용하여 컴파일된 bitcode 이다[3].

3.1. Concolic 테스팅 기법

```
1 int main(){
2    int i = 0,j = 0;
3    make_symbolic(&i);
4    make_symbolic(&j);
5       if(i == 0){
6          if(j > 0) return 1;
7          else if(j == 0) return 2;
8          else return 3;}
9       else if(i == 1){
10         if(i + j > 0) return 4;
11         else return 5;}
12      else return 6;}
```

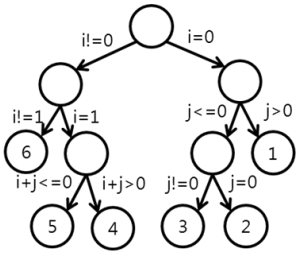

그림 1: 5개 분기 지점(branch point)을 가진 샘플 소스 코드 및 그 실행 경로 그래프.

Concolic 테스팅 기법은 임의의 입력 값으로 실제 수행을 하면서 실행한 경로에 대한 경로 조건들을 수집한다. 실행 경로가 끝나서 종료할 때 테스트 케이스를 생성하기 위해 이 경로 조건들 중 하나의 조건을 부정하여 새로운 경로를 실행할 다음 입력 값을 계산해 낼 수 있다. 예를 들어, 그림 1 의 실행 경로 그래프를 보자. 초기 입력 값을 $i=0, j=0$으로 주었을 때 5 번째 줄의 경로 조건은 $i=0$ 이 되고 이 if statement 를 통과한다. 6 번째 줄에 의해 경로 조건은 $j \le 0$ 이 되고 여기까지의 총 경로 조건은 $i=0 \wedge j \le 0$이 된다. 이렇게 코드가 종료할 때까지 수행하면 경로 조건은 $i=0 \wedge j \le 0 \wedge j=0$이 되고 이 때 마지막 조건, 즉 $j=0$을 부정한다. 그러면 $i=0 \wedge j \le 0 \wedge \neg(j=0)$ 이 새로운 경로 조건이 되고 이를 constraint solver 가 풀어 $i=0, j=-1$ 이라는 새로운 테스트 입력 값을 얻어낸다. 이렇게 새로 얻은 입력 값으로 프로그램을 다시 수행하여 새로운 경로들을 얻고 이런 과정은 모든 수행 경로들이 커버되거나 정해진 테스트 케이스 개수가 도출되거나 정한 시간에 도달할 때까지 반복된다.

3.2. KLEE 기본 구조

```
1    int main(){
2    int a, i, j = 3;
3    make_symbolic(&a);
4    if (a > 10) {  ... }
5    else {
6        for(i = 3; i < 10; i++){
7            if (a < i) {  ...  }
8        }
9        ...  }
10       return 0; }
```

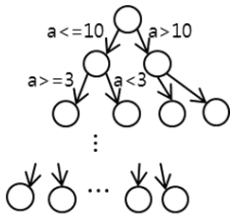

그림 2: for 루프 내에 심볼릭 조건을 가지는 분기를 포함하는 특성을 가지는 소스 코드 및 그 수행 경로 그래프.

KLEE 는 현재 수행하는 state 가 분기 지점에 도달하면 그 분기의 조건이 true 로 갈 수 있는 지와 false 로 갈 수 있는 지의 여부를 판단하기 위해 STP solver[11]에게 쿼리를 보낸다. true 와 false 모두 가능한 경우 현재 자기자신의 state 를 fork 하여 자식 state 를 생성하여 false 인 경로 조건을 추가하고 false 쪽의 경로를 수행하도록 하고 자기자신은 true 인 경로 조건을 추가하고 true 쪽의 경로를 수행하여 모든 방향의 경로를 커버한다.

위에서 설명한 바와 같이 KLEE 는 동시에 많은 state 를 생성할 수 있다. 그림 2 에서 for 루프 내에 심볼릭 변수가 포함된 조건을 가지는 분기를 포함하고 있는 코드의 경우 for 루프 내에서 생성되는 state 들이 많다. 이렇게 많은 state 들을 처리하기 위해 현재 실행 가능한 state 들 중에서 state 하나를 선택하고 그 state 의 프로그램 카운터가 가리키고 있는 한 instruction 을 수행하는 루프를 더 이상 수행할 state 가 남아있지 않을 때까지, 또는 임의로 정한 시간 제한까지 계속 반복한다[3].

3.3. State 스케줄링

주어진 시간 안에 높은 분기 커버리지를 달성하는 것이 중요하기 때문에 분기 커버리지를 높이는 데 효과적인 테스트 케이스를 먼저 도출하기 위한 탐색 방법이 중요시되고 있다. KLEE 는 state 스케줄링 방법으로서 다양한 탐색 방법을 제공하고 있다. 각 instruction 마다 여러 수행 가능한 state 들 중 하나를 선택하여 수행하도록 하는데, 몇 가지 스케줄링 방법을 제공한다. 현재 소개할 방법은 non-uniform random

search 라는 방법인데 하나의 특성에 대해 우선순위를 두어 state 를 임의로 선택한다. 총 6 가지 종류의 특성이 제공되고 세부적인 사항은 아래에 설명한다.

- depth: 분기를 적게 수행한 state 를 우선순위로 선택한다.
- icnt: 총 수행한 instruction 이 적은 state 를 우선순위로 선택한다.
- cpicnt: 현재 수행 중인 함수를 수행한지 얼마 안된 state 를 우선순위로 선택한다.
- query-cost: solver 가 경로 조건을 푸는 데 걸린 시간이 적은 state 를 우선순위로 선택한다.
- md2u: 가까운 미래에 새로운 코드를 커버할 것 같은 state 를 우선순위로 선택한다.
- covnew: 새로운 코드를 커버한 지 얼마 안된 state 를 우선순위로 선택한다.

또한 KLEE 는 이 스케줄링 방법들을 합쳐서 사용할 수 있도록 interleaving 하는 기법을 제공한다. 그리고 state 를 일정 시간, 혹은 instruction 개수만큼 계속 실행하도록 하는 라운드 로빈(round robin) 스케줄 방식과 비슷한 메커니즘을 가지는 batching 스케줄링 기법도 있어 위의 방법들과 함께 쓸 수 있다.

4. KLEE 적용 사례 연구

본 장에서는 KLEE 가 non-uniform random search 방법의 각 특성 별로 대상 프로그램에 대한 분기 커버리지를 측정하여 주어진 단위 시간 내에 어떤 차이를 보이는지 설명한다. 대상 프로그램은 GNU Coreutils 8.9[12]이고, 시간의 제약사항으로 인해 총 89 개의 유틸리티 중 기존 KLEE 논문에서 실험을 수행했던 15 개의 프로그램만 선택하여 진행하였는데, 선택한 프로그램들은 그림 3 에서 보이는 바와 같다. 각 프로그램 당 수행 시간은 30 분으로 제한하였고, 각 프로그램 당 6 가지의 non-uniform random search 방법을 한 번씩 테스팅 하도록 하였다. 실험은 64 bit Fedora Linux 9 시스템, CPU 는 Intel® Core™2 Duo E8400 @ 3.00 GHz, 메모리는 16GB 에서 수행하였다. 또한, LLVM 2.7[13], gcc 4.3.0 버전[14]을 이용하였다. 커버리지 측정은 gcov 4.3.0 버전[15]을 사용하였다.

4.1. 수행 옵션

모든 대상 프로그램은 입력 값으로 표준입력 또는 파일이 들어가므로 이것을 심볼릭 입력 값으로 만들어서 처리할 수 있는 심볼릭 POSIX 라이브러리를 사용하도록 옵션을 설정하였고, 대상 프로그램의 주요 외부 라이브러리(external library)를 지원하기 위해 uClibc 라이브러리를 포함시켰다. KLEE 는 LLVM bitcode 파일을 직접 수행하므로, uClibc 라이브러리는 KLEE 에서의 외부 함수(external function) 콜을 가능하게 하는 LLVM 버전으로 수정되어 있다. 또한 모든 실험에는 한 instruction 마다 state 스케줄링을 했을 때 생기는 오버헤드를 줄이기 위해 batching 탐색을 기본 옵션으로 하였고 10000 개 instruction 을 제한으로 두었다.

4.2. 가설 설정

실험에 대한 가설은 다음과 같다.

가설) 각 non-uniform random search 방법을 15 개의 대상 프로그램에 적용해서 각 non-uniform random search 별 평균 분기 커버리지를 구했을 때, 최대 커버리지를 가지는 방법과 최소 커버리지를 가지는 방법의 커버리지 차가 10%보다 크다.

가설에 대한 검정은 각 탐색 방법 별 모든 대상 프로그램에 대한 평균 분기 커버리지를 측정하는 것이다. 평균 최대 커버리지를 가지는 방법과 최소 커버리지를 가지는 방법의 커버리지 차가 10% 보다 큰 차이를 보인다면 위 가설을 채택할 것이고, 10% 이하의 커버리지 차이를 보인다면 위 가설을 기각한다.

4.3. 수행 결과

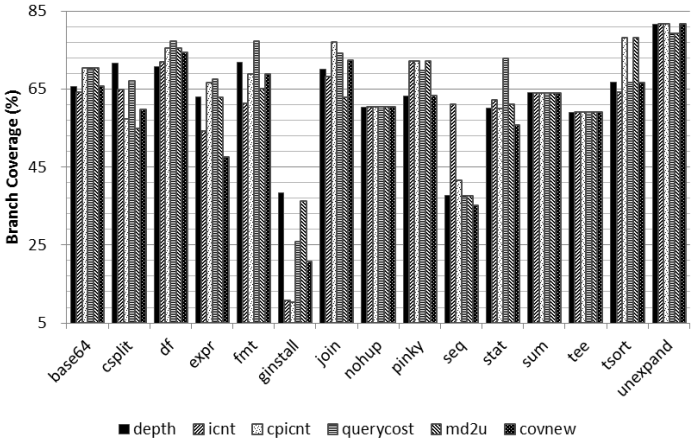

그림 3: 각 대상 프로그램에서의 6 가지 탐색 방법 별 분기 커버리지.

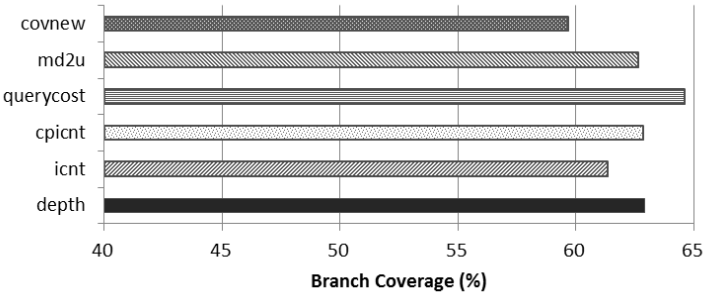

그림 4: 각 탐색 방법 별 모든 대상 프로그램에 대한 평균 분기 커버리지.

6 가지 특성의 non-uniform Random Search 방법을 적용해 본 결과는 그림 3 과 같다. 모든 방법이 확연하게 비슷한 분기 커버리지를 가지는 대상 프로그램들은 base64, df, nohup, sum, tee, unexpand 등 6 개이고, 특정 방법에서 높은 성능을 보이는 프로그램은 ginstall(depth, md2u), seq(icnt), stat(query-cost), tsort(cpicnt, md2u) 등 4 개의 프로그램이다. 그러나 6 가지 특성의 방법 중 특정 방법이 전체적으로 높은 분기 커버리지를 보이는 경향은 드러나지 않는다. 그림 4 의 그래프에서 보듯이 평균 분기 커버리지를 도출한 결과 대부분의 방법들이 모두 60%~65%의 분기 커버리지를 가져, 평균 최대 커버리지를 가지는 방법과 최소 커버리지를 가지는 방법의 커버리지 차는 약 5%의 비슷한 결과를 보였다. 그러므로 위에서 세운 실험 가설은 기각한다. 따라서 본 실험 결과로부터, 6 개의 서로 다른 탐색 방법 중 대상 프로그램에 최적화된 탐색 방법을 적용하려는 시도는 노력 대비 효과적이지 않음을 예상할 수 있다.

5. 결론

지금까지 KLEE 테스팅 도구의 주요 특징과 KLEE 에서 제공하는 여러 탐색 방법을 살펴보았고 Coreutils 를 적용하여 탐색 방법 별 분기 커버리지의 성능 차이를 살펴보았다. 결론적으로, 어떤 특성의 non-uniform random search 방법을 사용하든 비슷한 성능을 보이므로 6 가지의 non-uniform random search 는 효과가 크지 않다고 볼 수 있다.

현재 이 논문에서는 여러 탐색 방법들 중 non-uniform random search 의 6 가지 특성에 대한 설명 및 결과만 나타냈다. 이외에 KLEE 에서 제공하는 나머지 탐색 방법에 대한 적용 및 결과 분석도 수행할 예정이다. 또한 위의 수행 결과에서 covnew 특성의 non-uniform random search 방법에 대한 성능 저하에 대해서는 15 개 프로그램 이외의 다른 대상 프로그램들에도 적용시켜 보고 결과를 분석하여 원인을 검출할 계획이다.

참고 문헌

[1] C. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis,", in *Int. J. Softw. Tools Technol. Transfer*, 2009.

[2] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation." in *the international symposium on Code generation and optimization(CGO),* 2004.

[3] C. Cadar, D. Dunbar, and D. Engler. "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs." In *the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[4] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation.", in *Technical Report UCB/EECS-2008-123*, 2008.

[5] K. Sen, D. Marinov, G. Agha, "CUTE: A concolic unit testing engine for C.", in *5th joint meeting of the European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering(ESEC/FSE),* 2005.

[6] P. Godefroid, N. Klarlund, K. Sen, "DART: Directed automated random testing.", in *the Conference on Programming Language Design and Implementation (PLDI),* 2005.

[7] P. Godefroid, M.Y. Levin, and D. Molnar, "Automated whitebox fuzz testing.", in *Network and Distributed Systems Security*, 2008.

[8] N. Tillmann and W. Schulte, "Parameterized unit tests.", in *European Software Engineering Conference/Foundations of Software Engineering*, 2005.

[9] M. Kim, Y. Kim and Y. Choi, "Concolic Testing of the Multi-sector Read Operation for Flash Storage Platform Software", in *Formal Aspects of Computing (FACJ),* Vol 24, no 2, 2012. (to be published)

[10] Y. Kim, M. Kim and Y. Jang, "Concolic Testing on Embedded Software - Case Studies on Mobile Platform Programs", in *ACM SIGSOFT Foundation of Software Engineering (FSE) Industrial track*, Sep 2011.

[11] STP solver, http://sites.google.com/site/stpfastprover/

[12] Coreutils 8.9, www.gnu.org/software/coreutils

[13] LLVM 2.7, http://llvm.org/

[14] gcc4.3.0, http://gcc.gnu.org/

[15] gcov, http://gcc.gnu.org/onlinedocs/gcc/Gcov.html