Concolic Testing 도구 KLEE의 다양한 탐색 방법 비교

(Comparison of Search Strategies of KLEE Concolic Testing Tool)

김 영 주 † 김 문 주 ††
(YoungJoo Kim) (Moonzoo Kim)
김 윤 호 ††† 정 의 준 †††
(Yunho Kim) (Uijune Jung)

요 약 테스트 케이스 자동생성기법인 Concolic (concrete+symbolic) 테스팅 기법을 사용하는 테스팅 도구인 KLEE의 심볼릭 state를 스케줄링하는 탐색 방법(search strategy)들의 분기 커버리지 성능을 비교 분석했다. 또한 분기 커버리지를 보다 빠르게 높일 수 있는 breadth first search (BFS)를 새롭게 구현해 기존 방법들과 비교했다. 본 실험에서는 GNU Coreutils 버전 8.9를 대상으로, 주어진 시간에 각 탐색 방법들이 달성하는 분기 커버리지(branch coverage)를 비교하고 결과를 분석했다.

키워드 : Concolic 테스팅, 동적 심볼릭 수행, KLEE, 탐색 방법, Breadth First Search

Abstract Concolic (Concrete+symbolic) testing is an automated test case generation technique that works on target source code. This paper analyzes the branch coverage performances of the search strategies of a Concolic testing tool KLEE. In addition, we implemented the breadth first search (BFS) strategy to get higher branch coverage in KLEE. To compare the effectiveness of the search strategies, we applied KLEE to GNU Coreutils version 8.9 and compared the branch coverage of these search strategies and analyze their results.

Key words : Concolic Testing, Dynamic Symbolic Execution, KLEE, Search Strategy, Breadth First Search

· 본 연구는 교육과학기술부/한국연구재단 우수연구센터 육성사업의 지원으로 수행되었음(과제번호 2011-0000978)
· 이 논문은 제38회 추계학술발표회에서 'Concolic 테스팅 기법을 구현한 KLEE 테스팅 도구의 사례 연구'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 한국과학기술원 전산학과
jerry88@cs.kaist.ac.kr
(Corresponding author임)
†† 종신회원 : 한국과학기술원 전산학과 교수
moonzoo@cs.kaist.ac.kr
††† 비 회 원 : 한국과학기술원 전산학과
kimyunho@kaist.ac.kr
uijune.jeong@gmail.com
논문접수 : 2011년 12월 29일
심사완료 : 2012년 2월 6일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제18권 제4호(2012.4)

1. 서 론

동적 심볼릭 기법으로도 알려진 Concolic 테스팅 기법은 동적 테스팅과 심볼릭 수행을 함께 적용시킨 기법이다[1]. Concolic 테스팅 기법은 모든 수행 경로를 커버하는 테스트 케이스를 생성하는 것을 목표로 하지만, 주어진 시간 안에 높은 분기 커버리지를 달성하는 테스트 케이스 생성이 중요하고, 이를 위해 심볼릭 state를 스케줄링 하는 다양한 탐색 방법이 제시됐다.

본 논문에서는 Concolic 테스팅 도구인 KLEE에서 제공하는 여러 탐색 방법들을 비교하고, 또한 새로운 탐색 방법으로 breadth first search(BFS) 방법을 KLEE에 새롭게 구현했다. 기존 탐색 방법과 BFS 방법과의 효과 및 효율성 차이의 비교, 분석을 위해 GNU Coreutils을 대상으로 각 탐색 방법 별로 대상 프로그램들을 테스트해 분기 커버리지 결과를 비교 분석했다.

2. 관련 연구

Concolic 테스팅 기법은 실제 수행(concrete execution)으로부터 심볼릭 경로 수식(symbolic path formula) 추출 방법에 따라 두 가지로 나뉜다. 첫째, 대상 프로그램을 정적으로 instrument하는 테스팅 도구다. 이 카테고리의 테스팅 도구는 probe를 삽입해 실제 수행으로부터 심볼릭 경로 수식을 추출한다. CREST[2], CUTE[3], DART[4] 등이 이 카테고리에 해당한다. 둘째, 심볼릭 경로 수식을 추출하기 위해 수정된 가상 머신을 활용하는 방법이다. KLEE[5]는 LLVM[6] 바이너리를 대상으로 구현됐고, PEX[7]는 Microsoft .Net 바이너리로 컴파일되는 C# 프로그램을 대상으로 한다.

Concolic 테스팅 기법은 다양한 소프트웨어에 적용되고 있다. [8]에서는 플래시 메모리 플랫폼 소프트웨어의 멀티 섹터 읽기 연산 수행 함수에 concolic 테스팅 기법을 적용해 테스팅 성능 측정 및 장단점을 분석했다. 또한 [9]에서는 오픈소스 라이브러리 libexif에 concolic 테스팅 툴인 CREST와 KLEE를 적용해 concolic 테스팅 기법의 효과를 분석하고, 두 툴의 장단점을 분석해 비교했다.

3. KLEE 테스팅 도구

KLEE는 LLVM bitcode로 컴파일 된 프로그램을 인터프리트해 concolic 테스팅 기법을 수행하는 도구다[5]. KLEE는 오픈 소스로 제공되며 다양한 탐색 방법을 포함한다는 장점이 있다. KLEE는 심볼릭 프로세스의 운영체제와 인터프리터 역할을 동시에 수행한다. 심볼릭 프로세스는 실제 concolic 테스팅 실행 경로를 따라가며 분기 조건(branch condition)을 저장하는 concolic 테스팅을 위한 프로세스로, state라 표현한다. 각 state는 프로세스와 동일하게 스택, 힙, 프로그램 카운터 등의 정보를 갖고 있고, 자신이 수행한 경로 조건 정보를 축적해 나중에 경로 종료 시 테스트 케이스 도출에 쓴다.

3.1 KLEE 기본 구조

KLEE는 현재 수행하는 state가 분기 지점에 도달하면 그 분기의 조건이 true로 갈 수 있는 지, false로 갈 수 있는 지의 여부를 판단하기 위해 STP solver[10]에게 현재까지의 심볼릭 경로 수식에 현재 분기 조건을 더해서 쿼리로 보낸다. true와 false 모두 가능한 경우 state를 fork하여 자식 state를 생성해 그 state가 false인 경로 조건을 수행하도록 false 경로 조건을 추가하고, 자기자신은 true인 경로 조건을 추가한다. 그리고 현재 모든 실행 가능한 state들 중 하나를 택하고, 그 state의 프로그램 카운터가 가리키는 instruction 수행을 계속해 모든 가능한 state를 다 분석할 때까지 또는 주어진 시간 제한까지 수행한다[5].

예를 들어 그림 1 예제 코드로부터 KLEE는 각 노드가 분기 지점이고, 총 노드 7개인 심볼릭 수행경로 트리를 생성한다(그림 2참조). 3-4번째 줄은 i와 j변수를 심볼릭 변수로 선언하고, 심볼릭 변수가 포함된 분기 조건

```
1 int main(){                 8       else ret=2;
2   int i, j, ret;            9   } else {
3   make_symbolic(&i);        10      if(j == 1)
4   make_symbolic(&j);        11        ret=3;
5   if(i == 0){               12      else ret=4;
6     if(j == 0)              13  }
7       ret=1;                14  return ret; }
```

그림 1 여러 개의 분기문을 갖는 예제 코드

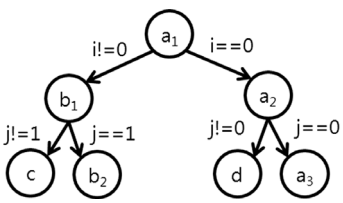


그림 2 그림 1에서 생성한 심볼릭 수행경로 트리
(알파벳은 각 노드를 수행한 state)

문만 고려해 심볼릭 수행 경로 트리를 생성한다. 제일 처음 main에 해당되는 a_1 state 생성 후, i 값이 아직 정해지지 않았으므로 5번째 줄에서 i==0을 만족하지 않는, 즉 i!=0 조건의 b_1 state를 생성하고, 자기자신은 i==0조건을 저장한다. 이 true쪽 경로 조건이 추가된 a_1 state를 a_2 state 라 한다. 그 후, a_2와 b_1 state 중 어떤 state를 먼저 선택해 수행할지는 탐색방법이 결정한다(3.2절 참조). b_1 state가 먼저 수행한다 가정할 때, 10번째 줄의 분기문을 실행하고 마찬가지 방법으로 j!=1 조건의 c state를 생성한다. 이 때 b_1 state는 j==1 조건을 저장하고 이를 b_2 state라 한다. 이렇게 해서 b_2와 c state는 심볼릭 수행을 마치게 된다. 이 때, b_2와 c state는 자신들이 축적한 경로 조건들을 solver에게 보내 각 경로의 테스트케이스를 도출하며 종료한다. 그 다음에 실행 가능한 state는 a_2 state 뿐이므로, a_2 state에서 d state를 생성해 분기 조건을 추가해 a_3 state로 업데이트 하고 모든 state가 종료하면서, 더 이상 수행 가능한 state가 없으므로 KLEE는 종료하게 된다.

3.2 State 스케줄링(또는 state 탐색방법)

KLEE는 각 instruction 마다 수행 가능한 state들 중 하나를 택해 수행토록 하는 state 스케줄링 방법을 사용한다. KLEE는 총 6개의 non-uniform random search (NURS) 방법을 제공하며[5] 상응하는 조건을 만족하는 state를 확률적으로 우선 선택한다.

- depth: 총 수행한 분기 개수가 적은 state.
- icnt: 총 수행한 instruction이 적은 state.
- cpicnt: 수행 중인 함수를 수행한 지 얼마 안된 state. 즉, 현재 함수를 수행한 instruction이 적은 state.
- query-cost: STP solver가 분기 조건을 푸는 데 걸린 시간이 적은 state. 즉, state가 분기 지점을 수행할 때마다 그 분기 조건을 STP solver가 푸는 데 걸린 시간을 축적해 기록한 값이 작은 state.
- md2u: 아직 커버되지 않았지만 곧 커버할 코드와 거리가 작아서 새로운 코드를 곧 커버할 것 같은 state.
- covnew: 새 코드를 커버한 지 얼마 안된 state. 즉, 커버되지 않았던 코드를 커버한 가장 최근 시점부터 지금까지 수행한 instruction이 적은 state.

그리고 round robin 방식으로 일정 시간, 혹은 일정 instruction 개수 수행 후 다음 수행 state를 선택하는 배칭(batching) 기법을 위 스케줄링 기법과 함께 쓸 수 있다.

3.3 Breadth First Search (BFS) 방법

본 연구에서는 KLEE에서 제공하는 기존 탐색방법 외에 새롭게 BFS 방법을 구현했다. BFS 방법은 루트 노드에서 시작해 모든 이웃한 노드들을 수행하며 진행한다. 이 방법은 규모가 큰 대상 프로그램의 경우 동시에 너무 많은 state 생성으로 인한 메모리 부족 문제가

있으나, 크기가 작은 Coreutils와 같은 프로그램은 효과적으로 분기 커버리지를 높일 수 있다. BFS를 depth first search(DFS) 방법(수행 경로 트리 가장 깊은 부분까지 먼저 수행하며 진행)과 비교해 설명하기로 한다.

BFS 방법과 DFS 방법을 그림 1에 적용했을 때 수행 순서는 그림 3과 같다. 우선 BFS 방법의 수행 순서는 그림 1의 a_1 state(그림 3 BFS 트리 1번 노드)가 먼저 선택돼 수행되고, a_1 state 수행 중 false 경로 쪽 분기 조건을 추가한 b_1 state가 생성된다. 이 때 a_1 state는 true 쪽 분기 조건을 추가해 a_2 state로 업데이트된다. 그 후 스케줄링을 통해 a_2와 b_1 state 중 b_1 state(2번 노드)를 선택해 수행한다. b_1 state 수행 중 c state가 생성된 시점에 b_1 state는 true쪽 경로 조건을 추가해 b_2 state가 되고, 스케줄링을 통해 a_2, b_2, c state 중 a_2 state(3번 노드)를 선택해 수행한다. a_2 state 수행 중 d state가 생성된 시점에 a_2 state는 true쪽 경로 조건을 추가해 a_3 state가 되고, 스케줄링을 통해 a_3, b_2, c, d state 중 c state(4번 노드)를 선택해 수행한다. c state가 수행을 종료하고 a_3, b_2, d state만 남은 시점에 b_2 state(5번 노드)가 선택되고, d state(6번 노드), a_3 state(7번 노드) 순으로 선택돼 수행한다.

DFS 방법의 수행 순서는 a_1 state(DFS 트리 1번 노드)가 먼저 선택돼 수행되고, a_1 state 수행 중 b_1 state가 생성된 시점에 a_1 state는 true쪽 분기 조건을 추가해 a_2 state가 된다. 그 후 스케줄링을 통해 a_2와 b_1 state 중 b_1 state(2번 노드)를 선택해 수행한다. b_1 state 수행 중 c state가 생성된 시점에 b_1 state는 true쪽 분기 조건을 추가해 b_2 state가 되고, 스케줄링을 통해 a_2, b_2, c state 중 c state(3번 노드)를 선택해 수행하고, c state가 수행을 마치고 종료하면 a_2와 b_2 state 중 c state의 마지막 조건에 반대되는 쪽 경로의 state인 b_2 state(4번 노드)를 선택해 수행한다. b_2 state가 수행을 종료하면 a_2 state밖에 남지 않아 a_2 state(5번 노드)를 수행토록 한다. a_2 state 수행 중 d state가 생성된 시점에 a_2 state는 분기 조건을 추가해 a_3 state가 되고, 아까와 같은 방식으로 d state(6번 노드)를 먼저 선택하고, d state 종료 후 a_3 state(7번 노드)를 선택해 수행한다.

BFS 방법을 KLEE에 구현한 알고리즘은 그림 4와 같다. 그림 4 알고리즘에서 StateQueue는 BFS 순서대로 수행하기 위한, state를 각 요소로 하는 큐 구조체고, 프로그램 첫 번째 state를 initState라 가정하고 시작한다(1-2번째 줄). 우선, 첫 번째 state인 initState가 StateQueue 안에 들어간다(3번째 줄). StateQueue에 state가 존재하므로 while 반복문 내부로 들어간다(4번째 줄). 현재 StateQueue에 initState 밖에 없으므로 수행할 state인 curState는 initState를 가리킨다(5번째 줄). 그 후 Execute 함수에서 인자로 들어온 curState를 한 instruction 수행하고, 분기를 만나 fork해 자식 state를 생성한 경우에만 childStates 배열에 자식 state부터 자기자신 순으로 저장하고 리턴한다(6번째 줄). childStates 배열의 state들은 반복문을 돌며 StateQueue에 삽입된다(7-9번째 줄). 그 후 수행 가능한 state가 존재하지 않을 때까지 while 반복문 내에서 위의 수행 단계를 반복한다.

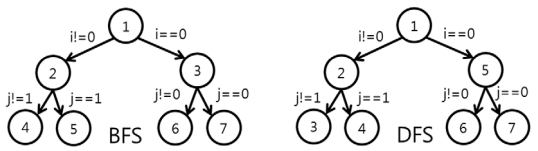


그림 3 그림 1의 소스코드에서 BFS와 DFS적용시 수행 경로 그래프(각 노드 번호는 state 선택순서)

```
Function: runProgram_BFS
1  StateQueue = new empty queue
2  initState = initial state of the program
3  Enqueue initState into StateQueue
4  while StateQueue is not empty do
5    curState = Dequeue from StateQueue
6    childStates = Execute(curState)
//Execute: curState를 한 instruction 수행하는 함수
//리턴 값은 curState의 자식 state들의 배열이다.
7    for each state s in childStates
8      Enqueue s into StateQueue
9    end for
10 end
```

그림 4 BFS 방법의 프로그램 수행 알고리즘

4. KLEE 적용 비교 분석

본 장에서는 NURS 방법, DFS 방법 및 새로이 구현한 BFS 방법을 적용해 달성한 분기 커버리지를 측정해 분석한다. 대상 프로그램은 GNU Coreutils 8.9[11]이고, 시간 제약으로 인해 총 89개의 유틸리티 중 기존 KLEE논문[5]에서 실험했던 15개 프로그램을 선택했다. 선택한 프로그램 각각의 분기 개수 및 코드 크기(행 수)는 표 1과 같다. 각 프로그램 당 수행 시간은 30분으로

표 1 Coreutils 15개 유틸리티의 크기

유틸리티	분기 수	LOC	유틸리티	분기 수	LOC
base64	128	322	pinky	158	616
csplit	316	1493	seq	154	443
df	343	958	stat	375	1473
expr	221	976	sum	89	276
fmt	263	1013	tee	83	223
ginstall	1178	3533	tsort	156	563
join	352	1139	unexpand	174	535
nohup	96	241			

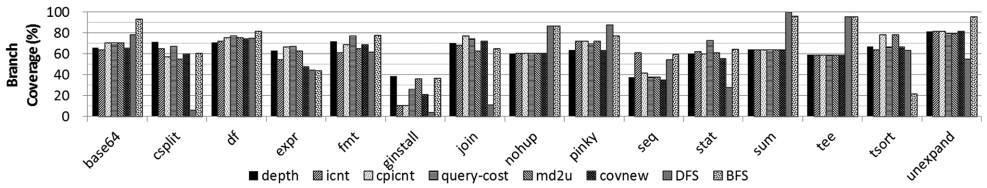


그림 5 각 대상 프로그램에서의 6 가지 NURS 방법과 BFS 방법, DFS 방법 별 분기 커버리지

제한했고, 각 프로그램 당 6가지 NURS 방법, DFS, BFS를 적용해 concolic 테스팅을 수행했다. 실험은 64 bit Fedora Linux 9, intel CoreTM2 Duo 3 GHz, 16GB 메모리를 사용했다. 또한, KLEE 리비전 136605, LLVM 2.7[12], gcc 4.3.0, 그리고 커버리지 측정은 gcov 4.3.0을 사용했다.

4.1 수행 옵션

모든 대상 프로그램은 입력 값으로 표준입력 또는 파일이 들어가므로 이를 심볼릭으로 처리해주는 KLEE의 심볼릭 POSIX 라이브러리를 사용하도록 옵션을 설정했고, 주요 C라이브러리를 심볼릭하게 지원하는 uClibc 라이브러리를 포함했다. 또한 모든 실험은 state 스케줄링 비용을 줄이기 위해 10000개 instruction마다 스케줄링이 생기도록 배칭 기법을 적용했다.

4.2 가설 설정

실험에 대한 가설은 다음과 같다.

가설) 각 탐색 방법을 15개 대상 프로그램에 적용해 각 탐색 방법 별 평균 분기 커버리지를 구했을 때, 최대 커버리지를 가지는 방법과 최소 커버리지를 가지는 방법의 커버리지 차가 10%보다 크다.

이 때 일반적으로 커버리지 차가 적어도 10% 이상 돼야 커버리지 차이가 실제적 의미를 가진다고 받아들여지므로 그 기준을 10%로 뒀다.

가설검정을 위해 각 탐색 방법 별 모든 대상 프로그램의 평균 분기 커버리지 (%)를 측정했다. 평균 분기 커버리지 측정은 아래 식과 같은 산술 평균 방식으로 구했다. 여기서 $Util_{total}$은 총 유틸리티 개수고, Cov(i)는 i 번째 유틸리티의 분기 커버리지(%)다.

$$\frac{\sum_{i-1}^{Util_{total}} Cov(i)}{Util_{total}}$$

평균 최대 커버리지를 갖는 탐색 방법과 최소 커버리지를 갖는 방법의 커버리지 차가 10% 보다 크면 위 가설을 채택하고, 10% 이하면 위 가설을 기각한다.

4.3 수행 결과

6 가지 특성의 NURS 방법과 DFS방법, BFS 방법을 적용한 결과는 그림 5와 같다. df의 경우 모든 탐색 방법의 커버리지 차가 약 10% 이내로 비슷한 반면, base64, nohup, sum, tee는 특정 탐색 방법이 그 외 탐색 방법의 평균보다 약 20% 이상의 차로 높은 분기 커버리지를 달성했다. 대체로 BFS가 다른 방법들보다 높은 커버리지를 보이는 경향을 보인다.

그림 6의 그래프에서 보듯이 평균 분기 커버리지를 도출한 결과 DFS 방법이 약 57%로 가장 낮았고, BFS 방법이 약 70%로 가장 높았다. 6 가지 NURS 방법들은 60-65%였다. 평균 최대 커버리지를 가지는 방법과 최소 커버리지를 가지는 방법의 커버리지 차는 약 13%로, 위 가설의 기준인 10%보다 크므로, 가설을 채택한다. 이는 이전 논문인 [13]과 다른 결과로, Coreutils의 유틸리티들이 갖고 있는 특성을 가진 도메인에 대해 평균적으로 봤을 때 우리가 구현한 BFS 방법이 KLEE에서 제공하는 기존 탐색 방법과 비교해 보다 효과적이며, 탐색 방법의 선택이 중요하다는 것을 알 수 있다.

4.4 결과 분석

앞의 실험 결과 분석을 위해 최고의 성능을 보인 BFS 방법과 최저의 성능을 보인 DFS 방법을 비교했다. NURS 방법은 랜덤 특성을 가져 BFS와 비교가 힘들어 직접 비교는 하지 않았지만, NURS방법은 BFS와 DFS의 중간 특성 및 성능을 보여 BFS와 DFS의 비교분석 결과의 간접 적용이 가능하리라 생각한다. csplit, ginstall, join, stat, unexpand의 경우 BFS 방법이 DFS 방법보다 약 30-50% 높은 분기 커버리지를 보였다. tsort는 예외적으로 BFS가 DFS보다 약 40% 낮은 분기 커버리지를 보였다.

각 유틸리티 소스코드 분석 결과, 두 경우 대상 소스코드의 구조 특성이 다름을 파악했다. BFS 방법이 효과적인 5개 유틸리티는 공통적으로 그림 7 unexpand의 CFG 예 (a)와 같이 수행 트리 윗부분에서 커맨드라인에서 받는 다양한 옵션 처리에 대한 switch문이 존재하는, 다중 분기 정도가 높은 CFG 구조였다. BFS에서는

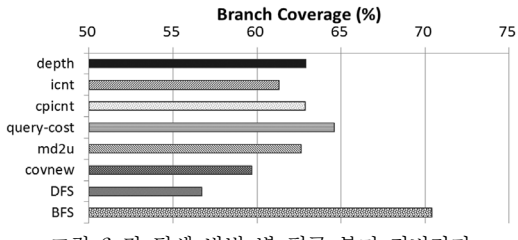
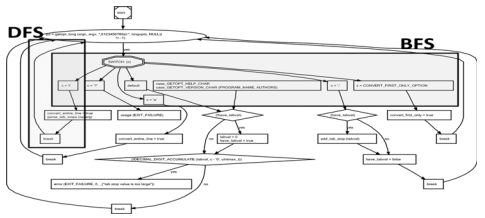
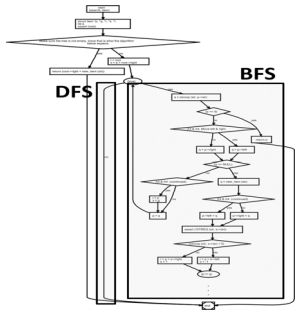


그림 6 각 탐색 방법 별 평균 분기 커버리지



(a)



(b)

그림 7 unexpand의 control flow graph (CFG) 예(a), tsort의 search_item 함수의 CFG 일부분(b)

모든 가능한 분기를 커버하면서 내려가므로, 그림 7(a)의 BFS 상자와 같이 한번 반복문 수행에 switch 문의 모든 case를 커버했다. 반면, DFS는 한 분기 끝까지 커버하며 반복문을 수행하므로 그림 7(b)의 DFS 상자와 같이 반복문을 돌며 같은 분기만 계속 수행해, 분기 커버리지를 높이지 못하고 결국 switch 문의 한 가지 case만 커버했다. 따라서 unexpand 유틸리티에서 BFS가 DFS 보다 분기 커버리지가 약 40% 높았다.

반면, tsort유틸리티는 커맨드라인에서 받는 옵션이 없어 다중 분기가 없었고, CFG가 좁고 깊은 구조였다. BFS는 tsort 시작 부분에서 불리는 search _item함수(그림 7(b))에서 이미 첫 반복문 수행 시 커버한 분기들임에도 계속 반복문을 돌며 true와 false 경로 모두를 커버하며 많은 state를 생성하느라 30분을 다 소요해, 제한 시간 내에 tsort 메인 알고리즘의 깊은 부분에 접근하지 못했다. 반면, DFS는 search_item 함수의 한쪽 분기로만 계속 수행해 반복문 내로 진입하지 않고 바로 빠져 나와, tsort 메인 알고리즘의 깊은 부분까지 접근해 보다 많은 분기를 커버했다.

이처럼 대상 프로그램의 코드 구조 특성에 따라 효과적인 탐색 방법이 존재할 수 있다. 그런 면에서 Coreutils의 대부분 유틸리티가 갖고 있는, 커맨드라인 옵션 처리 특성 구조가 BFS 이용 시 커버리지 측면에서 유리하도록 만든다고 할 수 있다.

5. 결론 및 향후 연구

KLEE의 여러 탐색 방법과 새로이 구현한 BFS 방법을 Coreutils의 15개 유틸리티를 적용해 분기 커버리지의 성능 차이를 비교했다. 어떤 탐색 방법이든지 약 57-70%의 높은 분기 커버리지를 도출했으므로 concolic 테스팅이 효과적이라는 것을 알 수 있다. 또한 Coreutils 유틸리티에 대해 BFS 방법이 KLEE의 기존 탐색 방법들보다 분기 커버리지 측면에서 우수했다. 앞으로 보다 큰 규모 대상 프로그램으로 Concolic 테스팅의 실제적 효과에 대해 좀 더 분석하고 연구할 것이다.

참 고 문 헌

[1] C. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," in *ICSE*, 2009.
[2] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Technical Report UCB/EECS-2008-123*, 2008.
[3] K. Sen, D. Marinov, G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE*, 2005.
[4] P. Godefroid, N. Klarlund, K. Sen, "DART: Directed automated random testing," in *PLDI*, 2005.
[5] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on OSDI*, 2008.
[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.
[7] N. Tillmann and W. Schulte, "Parameterized unit tests," in *ESEC/FSE*, 2005.
[8] M. Kim, Y. Kim and Y. Choi, "Concolic Testing of the Multi-sector Read Operation for Flash Storage Platform Software," in *FACJ*, Vol.24, No.2, 2012.
[9] Y. Kim, M. Kim, Y. Kim, and Y. Jang, "Industrial Application of Concolic Testing Approach: A Case Study on libexif by Using CREST and KLEE," in *ICSE SEIP track*, Jun.2-9th 2012.
[10] STP solver, http://sites.google.com/site/stpfastprover/
[11] Coreutils 8.9, http://www.gnu.org/software/coreutils
[12] LLVM 2.7, http://llvm.org/
[13] Y. Kim, Y. Kim and M. Kim, "Case Study on Testing with KLEE Concolic Testing Tool," in *KCC*, 2011.