Case Study on Testing with KLEE Concolic Testing Tool

Young Joo Kim Uijune Jung Yunho Kim Moonzoo Kim

Provable **S**oft**w**are **Lab**, KAIST

Contents

Our Motivation

- Concolic testing is not effective in practice
  - Limited time
  - Impossible to get full path coverage
- Effective search strategy is needed
- KLEE : concolic testing tool
  - The most widely used tool
  - Various search strategies
  - But, no criteria for search strategies
    - No documentation
    - No performance evaluation

- **Goal**

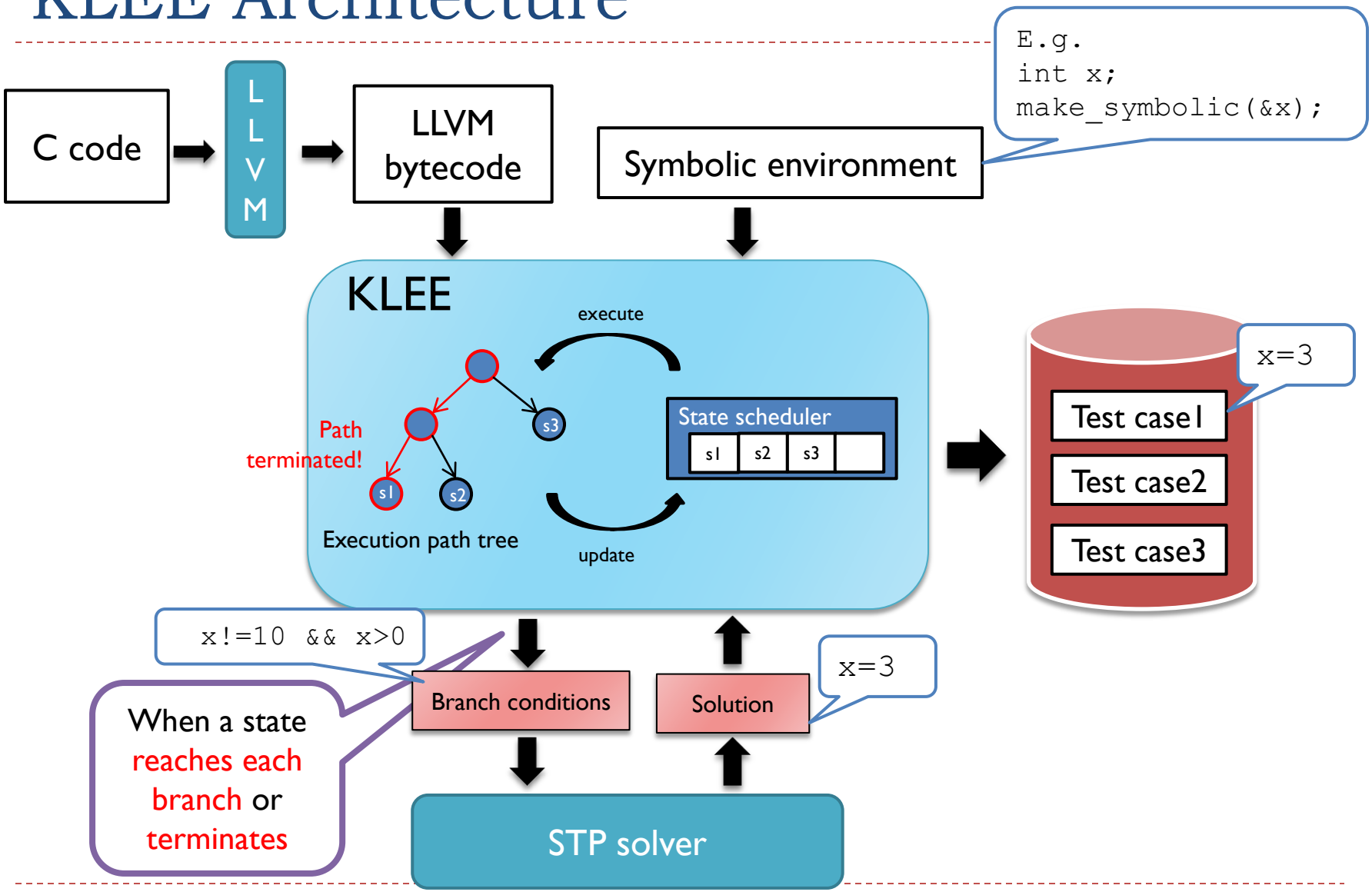
What: See power of each concolic testing search strategy
How: Evaluate (coverage)/(fixed time) of concolic testing search strategy

Concolic Testing

- Combine concrete and symbolic execution
  - Concrete + Symbolic = Concolic

- Automated testing for real-world C programs
  - Generate test cases automatically
  - Explore all possible paths
  - Achieve higher branch coverage than randomized testing

KLEE Architecture

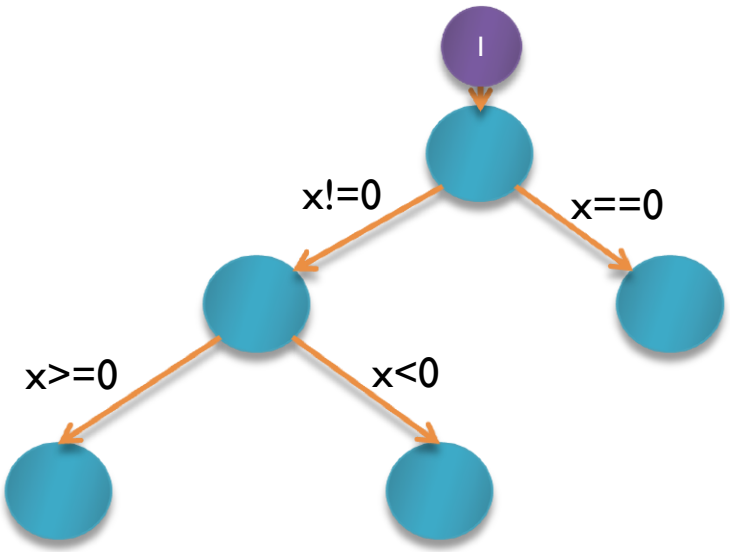


KLEE State

- Running through the target program execution paths.
- When a state reaches a branch condition, the state forks the branch.

```
int get_sign(int x) {
    if (x == 0)
        return 0;

    if (x < 0)
        return -1;
    else
        return 1;
}
```



<Execution Path>

STP solver

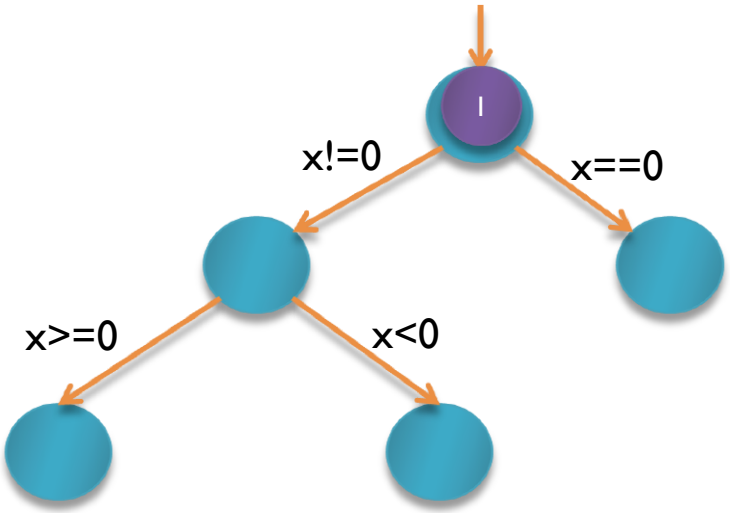
KLEE State

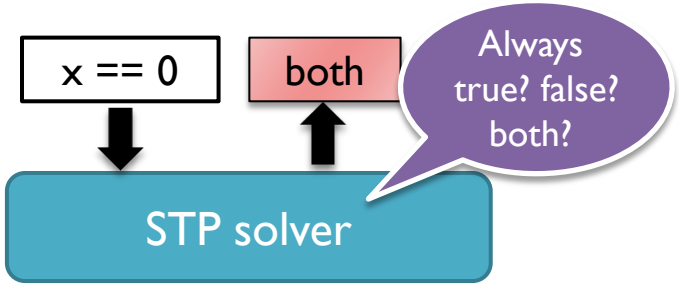
- Running through the target program execution paths.
- When a state reaches a branch condition, the state forks the branch.

```
int get_sign(int
    if (x == 0)
        return 0;

    if (x < 0)
        return -1;
    else
        return 1;
}
```

Branch point





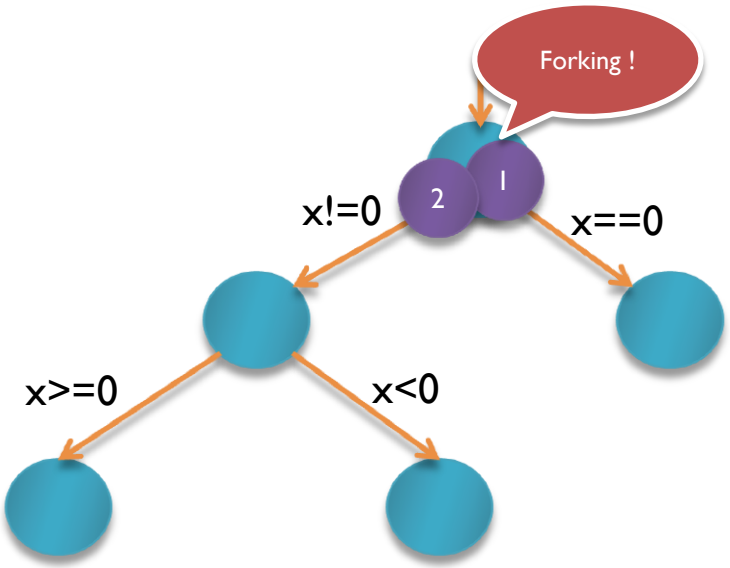
<Execution Path>

KLEE State

- Running through the target program execution paths.
- When a state reaches a branch condition, the state forks the branch.

```
1 int get_sign(int
2     if (x == 0)
3         return 0;
4
5     if (x < 0)
6         return -1;
7     else
8         return 1;
9 }
```

Branch point



<Execution Path>

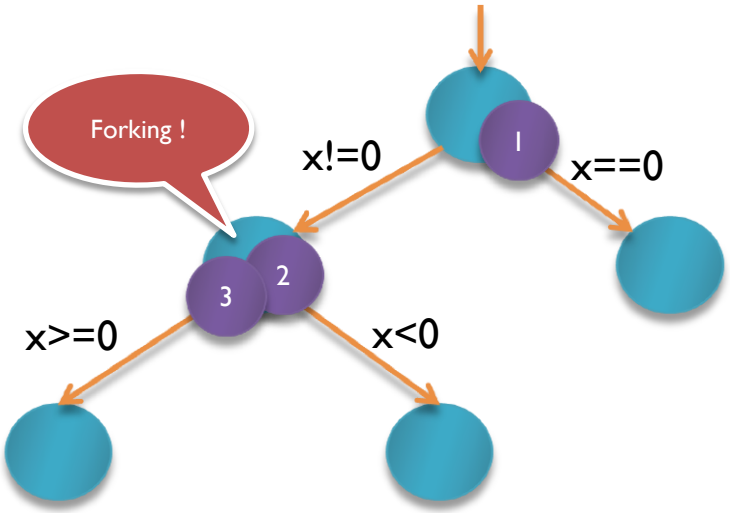
STP solver

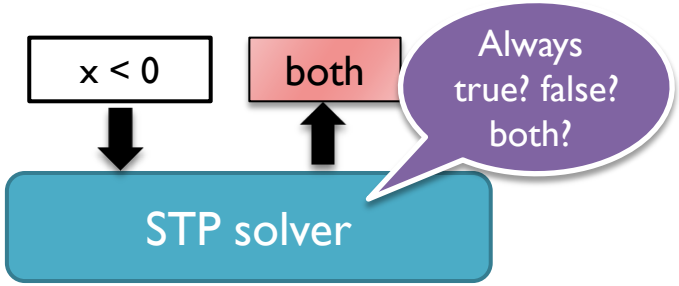
KLEE State

- Running through the target program execution paths.
- When a state reaches a branch condition, the state forks the branch.

```
1 int get_sign(int x) {
2     if (x == 0)
3         return 0;
4
5     if (x < 0)
6         return -1;
7     else
8         return 1;
9 }
```

Branch point



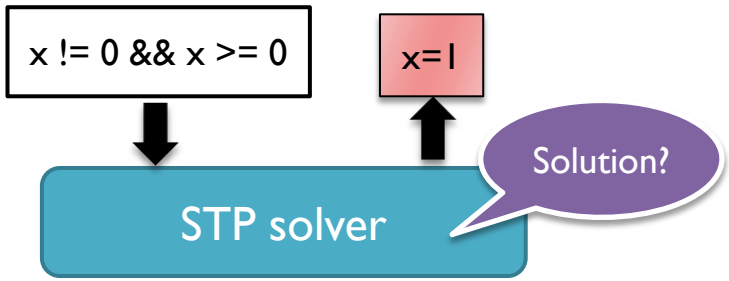


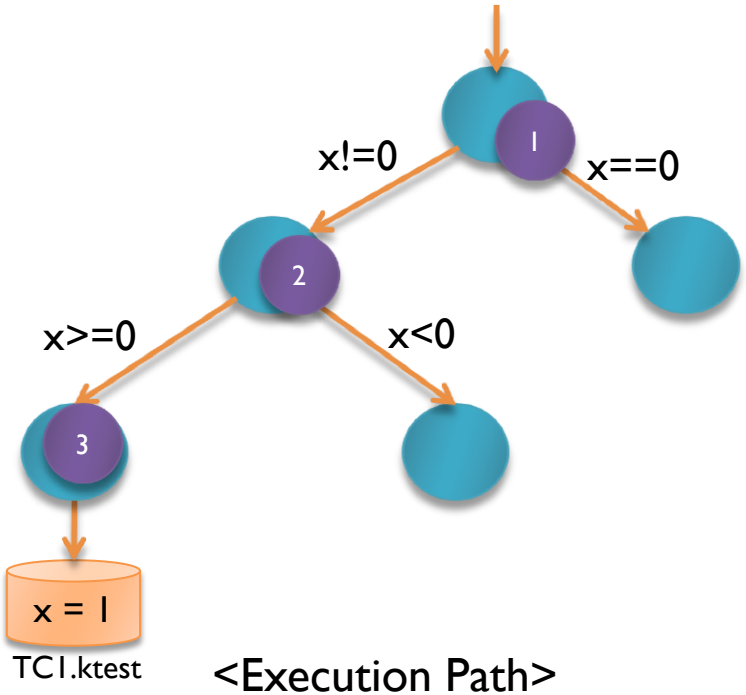
<Execution Path>

KLEE State

- Running through the target program execution paths.
- When a state reaches a branch condition, the state forks the branch.

```
1 int get_sign(int x) {
2     if (x == 0)
3         return 0;
4
5     if (x < 0)
6         return -1;
7     else
8         return 1;
9 }
```





<Execution Path>

Search Strategy of KLEE

- Search Strategy: heuristic of selecting states to execute paths

- Basic Search Heuristics
  - Depth First Search(DFS)
  - Random Path
  - Random Search
  - Six Non-uniform Random Search
    - depth
    - icnt
    - cpicnt
    - query-cost
    - md2u
    - covnew
  - Breadth First Search(BFS)
    - Implemented by us
- Interleaving approach
  - Combination of basic search heuristics
  - E.g. DFS + Random Search
- Round robin-like approach
  - Batching search
    - Base searcher
      - basic search heuristics
    - Quantum limit
      - time
      - # of instructions
  - E.g. DFS with Batching search

Case Study

- Target program
  - GNU Coreutils ver. 8.9
- Experimental environments

Machine	Specification
CPU	Intel(R) Core(TM)2 Duo E8400 @ 3.00GHz
Memory	16GB
Linux	Fedora 9 64-bit

- Test Strategy
  - Measurement
    - Branch coverage
      - To see effectiveness for each search strategy
  - Time limit
    - 30 minutes for each run
  - Basic search strategy
    - The 6 non-uniform random search heuristics

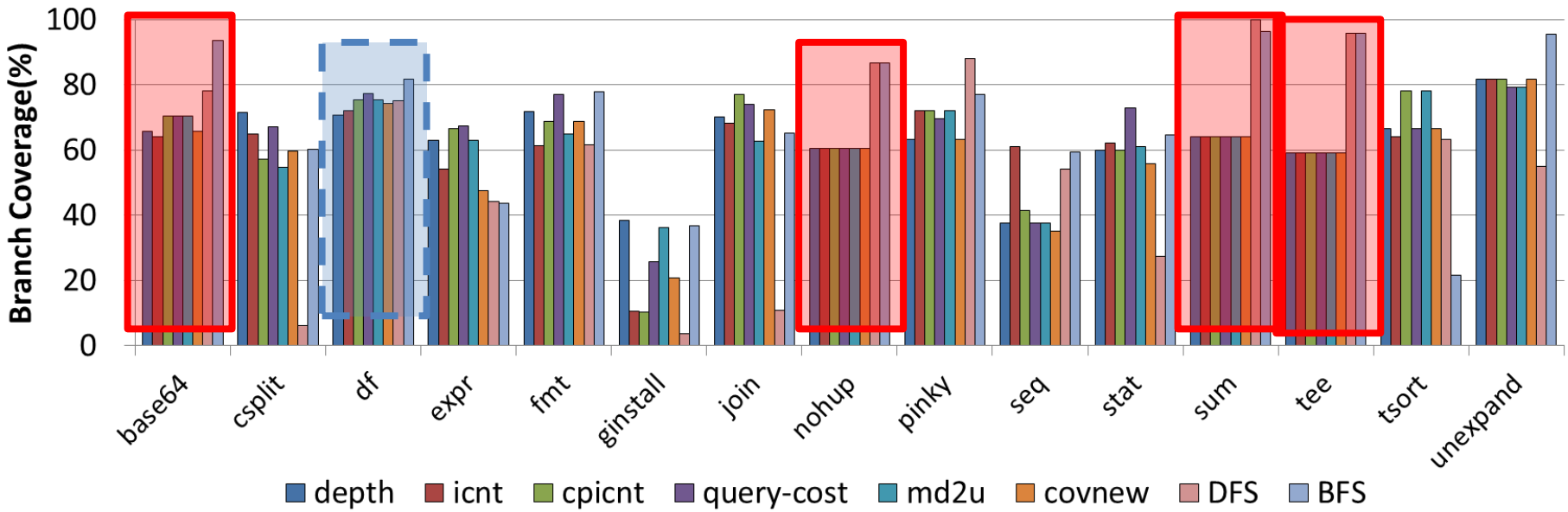
Target Program: Coreutils

- Basic command-line utilities of the GNU operating system such as Linux

- We selected following 15 utilities among total 89.
  - **base64, csplit, df, expr, fmt, ginstall, join, nohup, pinky, seq, stat, sum, tee, tsort, unexpand**
  - The utilities are chosen from KLEE paper "*KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*".

Result

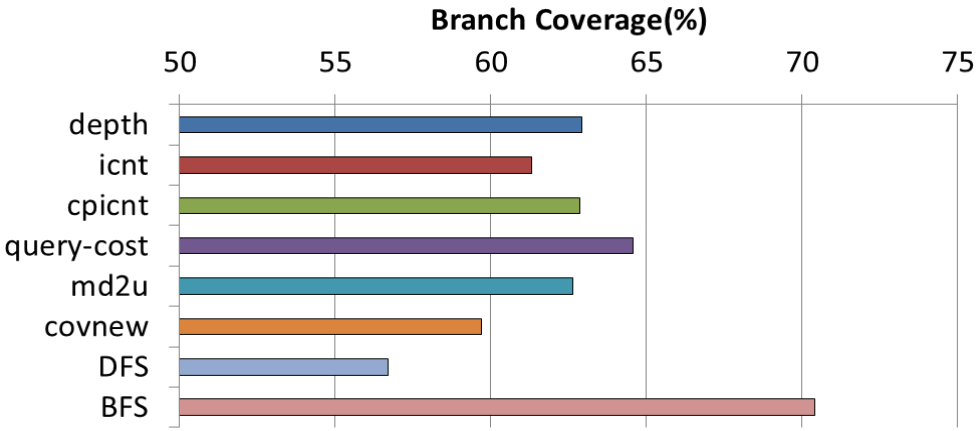
- Branch Coverage for Each Utility



- Specific strategy works best for others. (more than 20% higher)
  - base64, nohup, sum, tee
- None of the search strategies work best for others. (less than 10% difference)
  - df

Result

- Branch Coverage

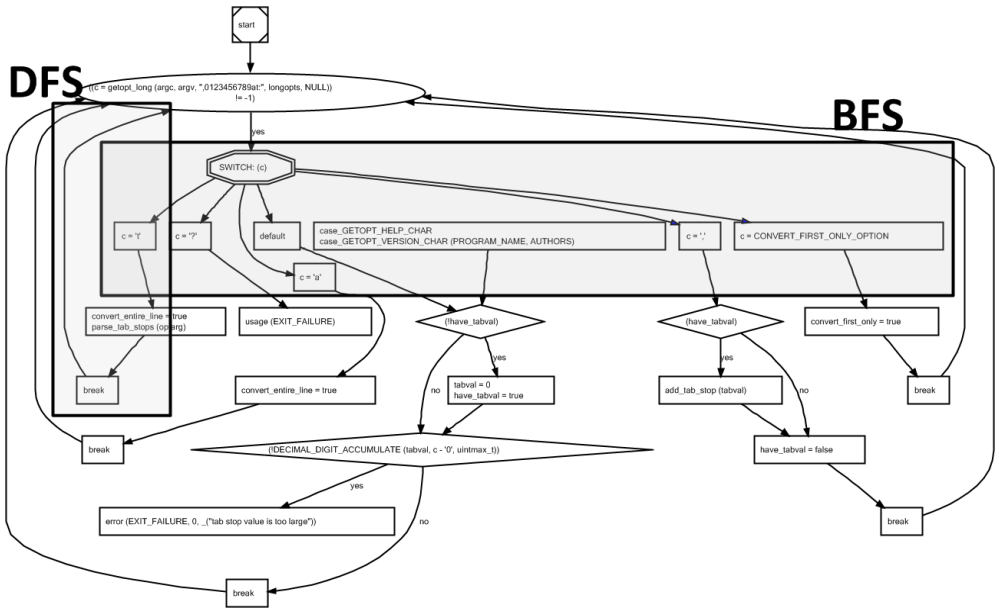


Util	Branch coverage (%)
depth	62.93
icnt	61.32
cpicnt	62.85
query-cost	64.58
md2u	62.63
covnew	59.68
DFS	56.70
BFS	70.41
|max-min|	**13.71**

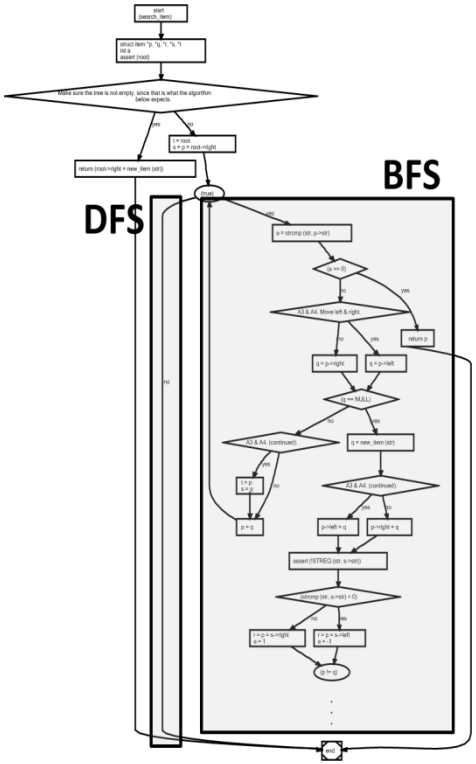
- Maximum: BFS
- Minimum: DFS
- About 60%~70% branch coverage reached
- The difference of these two branch coverage: 13.71%

Analysis

- Comparison of BFS and DFS



Case 1. Control Flow Graph(CFG) of unexpand



Case 2. CFG of search_item function in tsort

- In case 1, BFS covers all option cases, but DFS covers only one option case.
- In case 2, DFS covers deeper branch of tsort's algorithm than BFS.

Conclusion

- We demonstrated the effectiveness of concolic testing search strategies in KLEE by measuring branch coverage of each heuristic.

- Concolic testing is effective for getting high branch coverage.

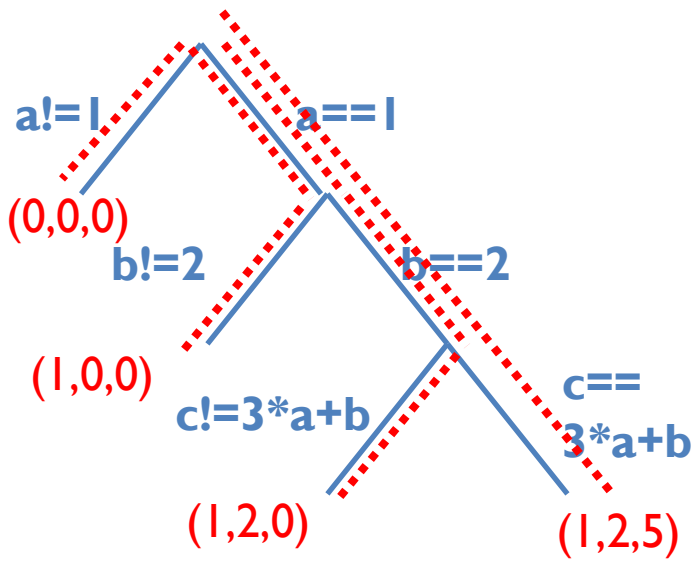
- **BFS works better than other search strategies in KLEE for branch coverage.**

Future Work

- We will propose other effective search strategies for getting higher branch coverage in concolic testing.
- We will experiment remaining search strategies and combination of search strategies.
- We will apply bigger target programs for all search strategies.

Concolic Testing Example

```
// Test input a, b, c
void f(int a,int b,int c){
 if(a == 1){
  if(b == 2){
   if(c == 3*a + b){
    Error();
}}}}
```



- Concolic testing generates the following 4 test cases
  - (0,0,0): initial random input
    - Obtained symbolic path formula (SPF) ϕ: a!=1
    - Next SPF ψ generated from ϕ: !(a!=1)
  - (1,0,0): a solution of ψ (i.e. (a==1))
    - SPF ϕ: a==1 && b!=2
    - Next SPF ψ: a==1 && !(b!=2)
  - (1,2,0)
    - SPF ϕ: a==1 && (b==2) && (c!=3*a +b)
    - Next SPF ψ: a==1 && (b==2) && !(c!=3*a +b)
  - (1,2,5)
    - Covered all paths and Error() reached