How is Dynamic Symbolic Execution Different from Manual Testing?

An Experience Report on KLEE

Xiaoyin Wang, Lingming Zhang, Philip Tanofsky
University of Texas at San Antonio
University of Texas at Dallas

Outline

- Background
- Research Goal
- Study Setup
- Quantitative Analysis
- Qualitative Analysis
- Summary and Future Work

Background

- Dynamic Symbolic Execution (DSE)
  - A promising approach to automated test generation
  - Aims to explore all/specific paths in a program
  - Generates and solves path constraints at runtime
- KLEE
  - A state-of-the-art DSE tool for C programs
  - Specially tuned for Linux Coreutils
  - Reported higher coverage than manual testing

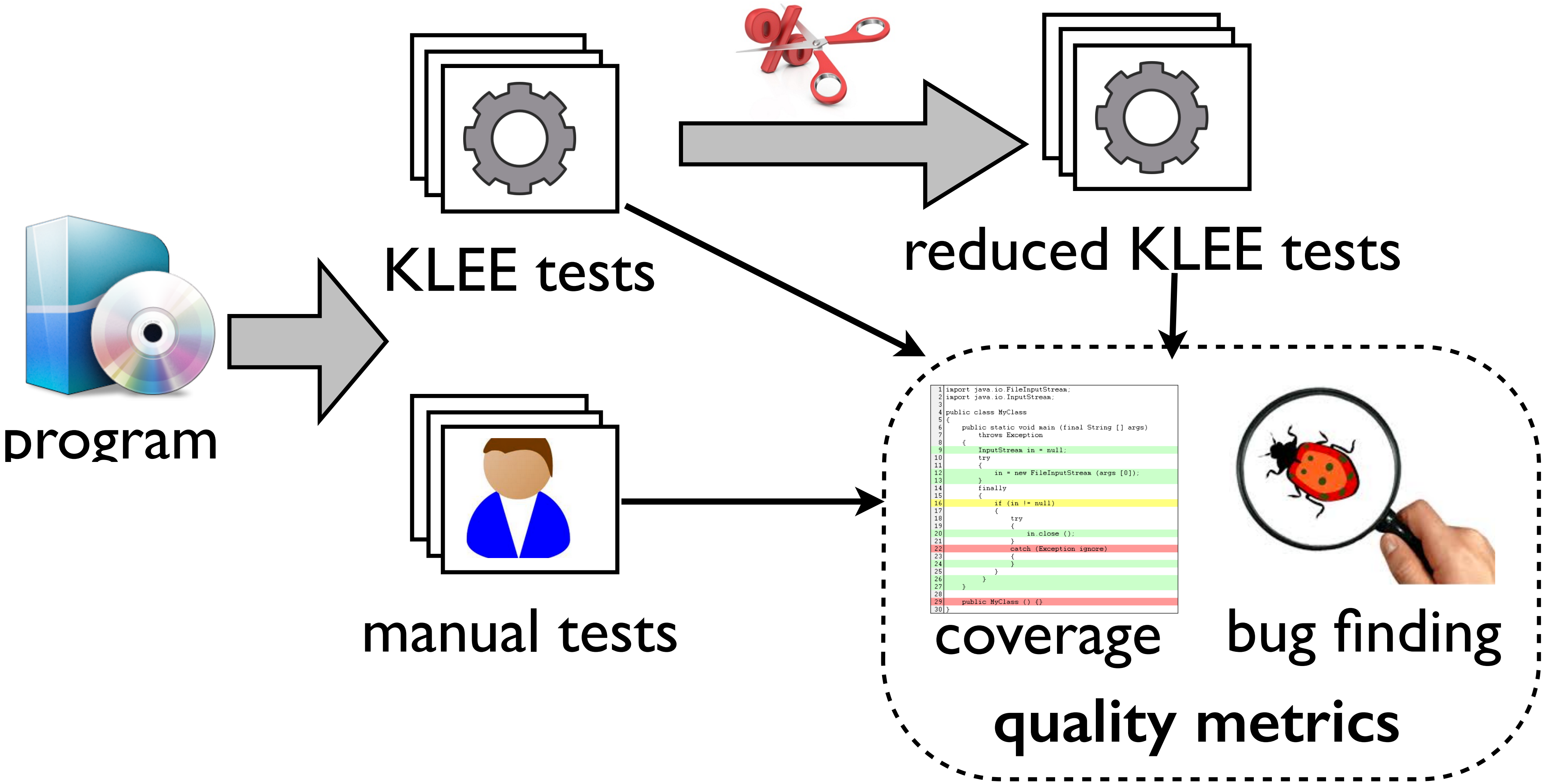
Research Goal

- Understand the ability of state-of-art DSE tools
- Identify proper scenarios to apply DSE tools
- Discover potential opportunities for enhancement

Research Questions

- Are KLEE-based test suites comparable with manually developed test suites on test sufficiency?
- How do KLEE-based test suites compare with manually test suites on harder testing problems?
- How much extra value can KLEE-based test suites provide to manually test suites?
- What are the characteristics of the code/mutants covered/killed by one type of test suites, but not by the other?

Study Process



Study Setup: Tools

- KLEE
  - Default setting for test generation
  - Execute each program for 20 minutes

- GCOV
  - Statement coverage collection

- MutGen
  - Generates 100 mutation faults for each program
  - 4 mutation operators

Study Setup: Subjects

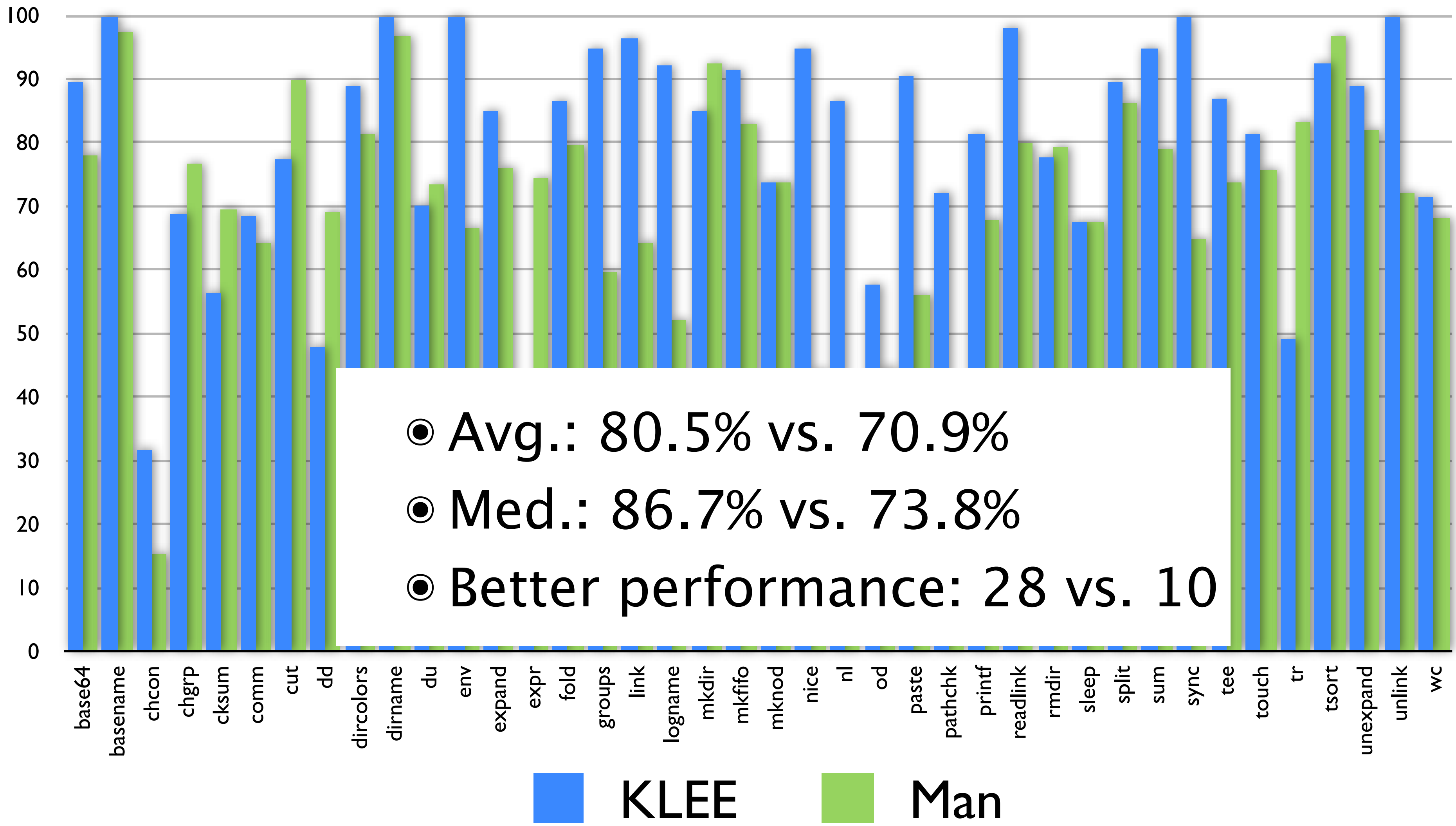
- CoreUtils Programs
  - Linux utilities programs
  - KLEE includes API modeling and turning of them
  - Used by KLEE in its evaluation

  - We did not include CoreUtils programs:
    - Do not generate any output
    - Output is not deterministic

Study Setup: Measurements

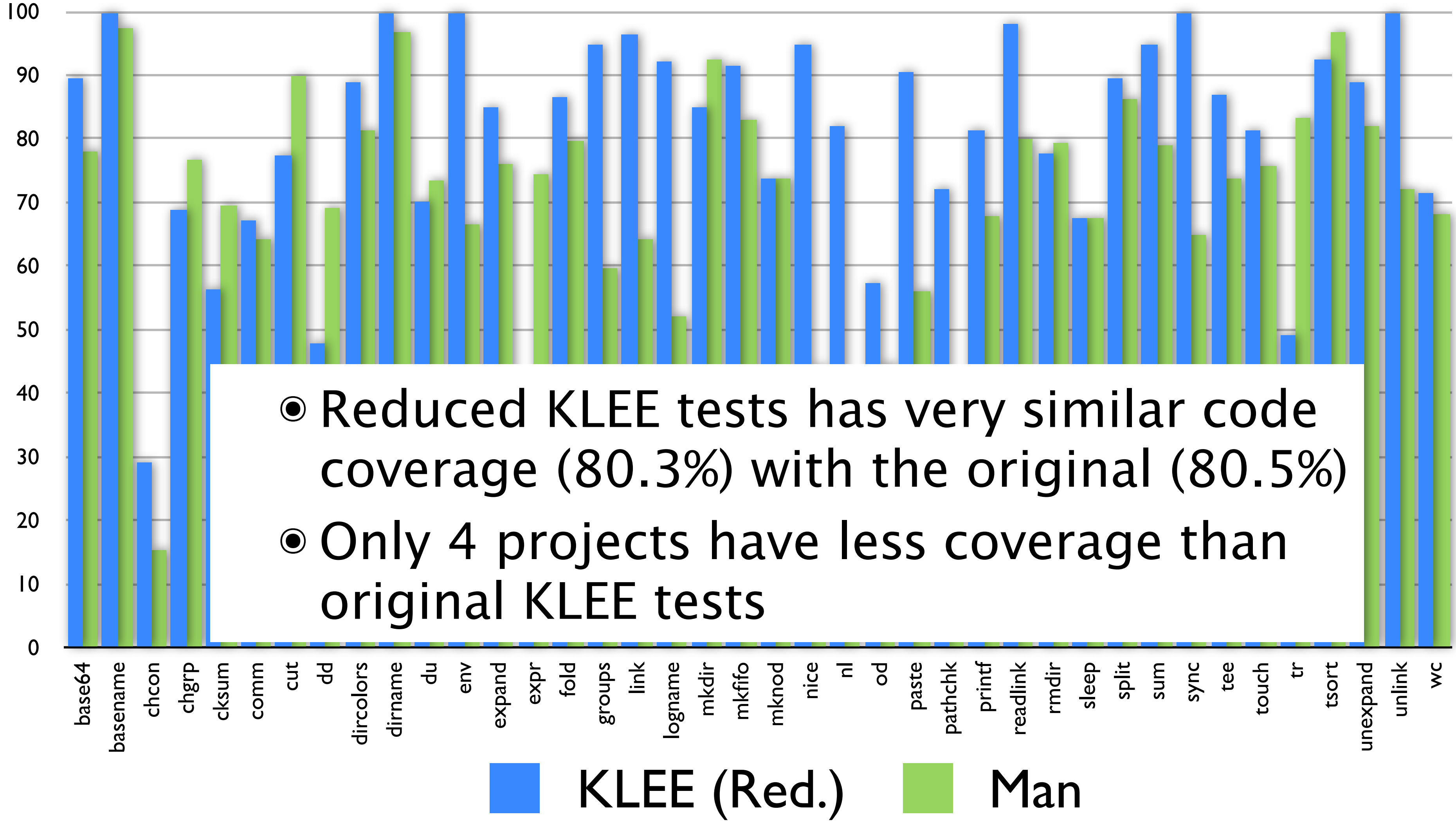
- Code coverage
  - Statement coverage
- Fault detection rate
  - Compare the command-line output of the original program and mutated programs to check if the mutation faults can be detected

Quantitative Analysis: Coverage



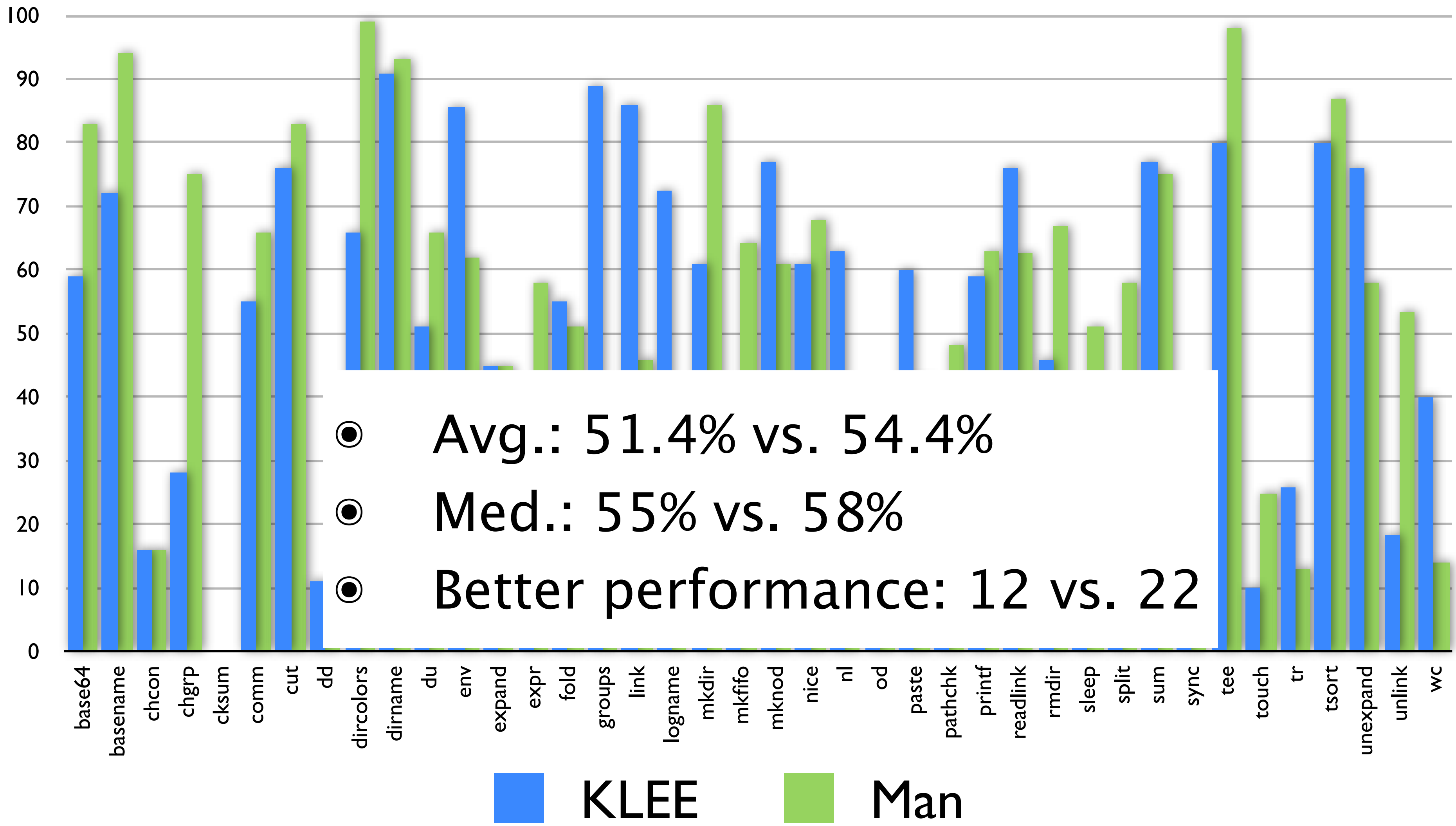
- Avg.: 80.5% vs. 70.9%
- Med.: 86.7% vs. 73.8%
- Better performance: 28 vs. 10

Quantitative Analysis: Coverage



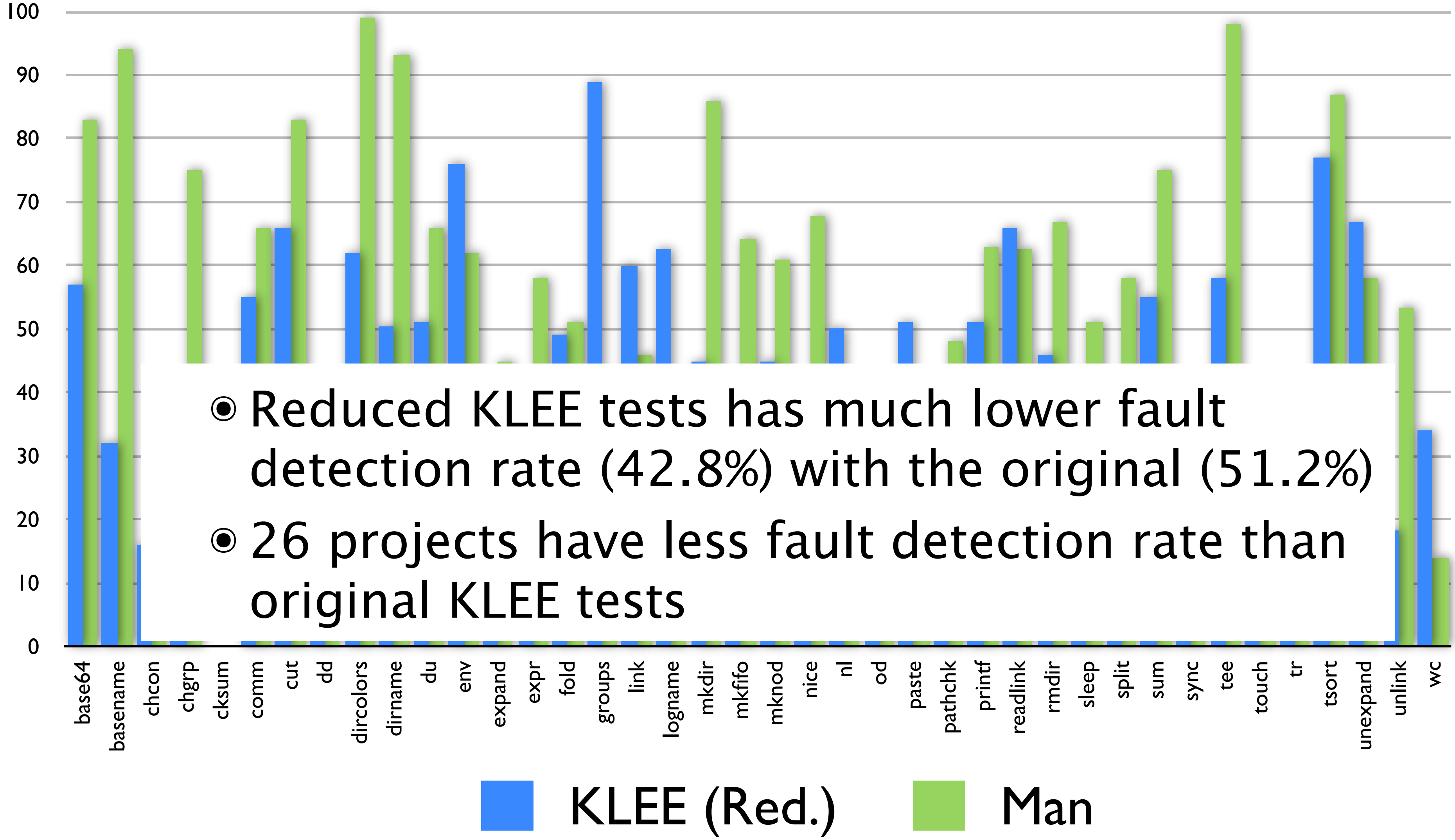
- Reduced KLEE tests has very similar code coverage (80.3%) with the original (80.5%)
- Only 4 projects have less coverage than original KLEE tests

Quantitative Analysis: Fault Detection



- Avg.: 51.4% vs. 54.4%
- Med.: 55% vs. 58%
- Better performance: 12 vs. 22

Quantitative Analysis: Fault Detection



- Reduced KLEE tests has much lower fault detection rate (42.8%) with the original (51.2%)
- 26 projects have less fault detection rate than original KLEE tests

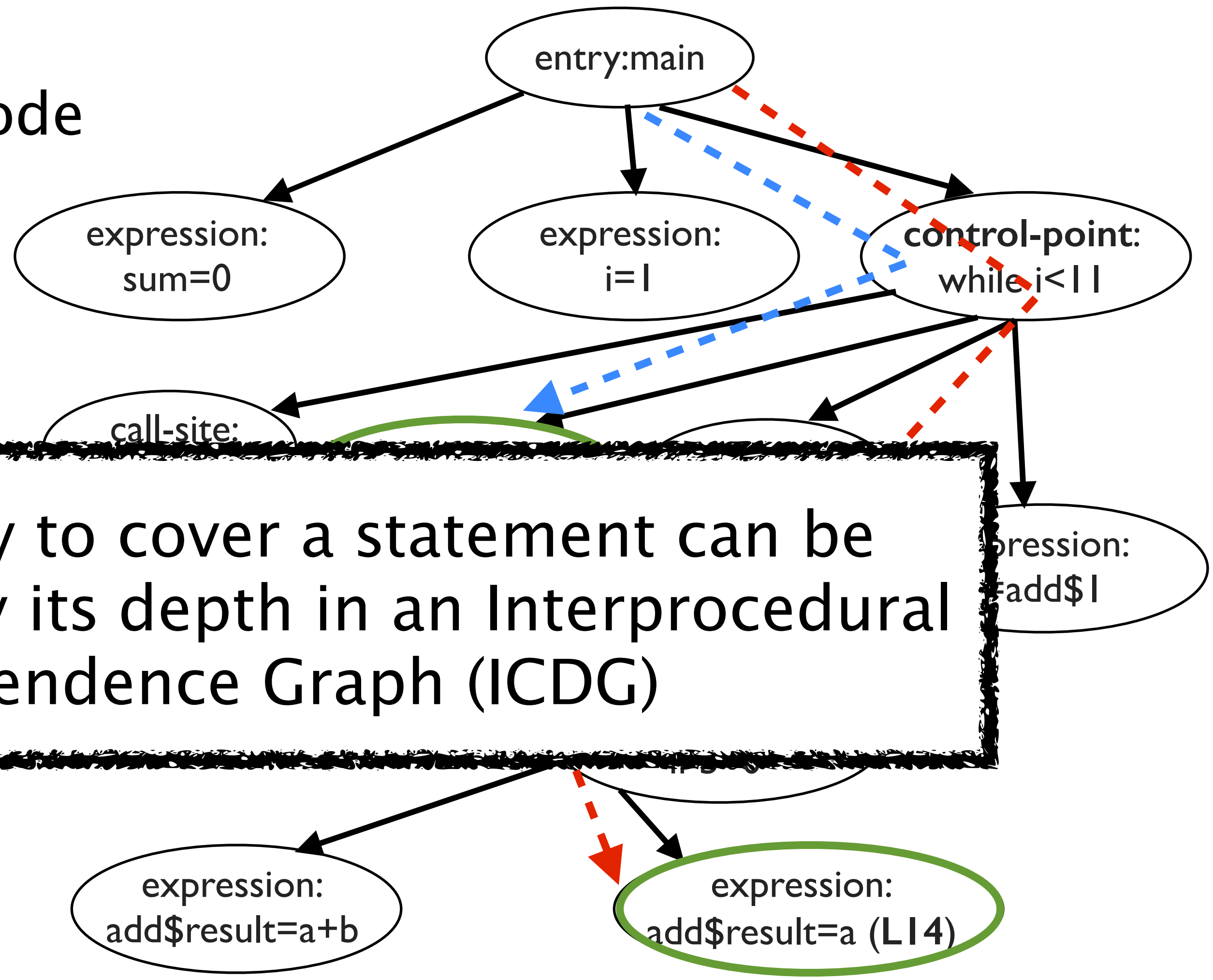
Quantitative Analysis: Harder Tasks (Code)

- Hard-to-cover code

```
void main() {
        int sum, i;
  sum = 0;
  i = 1;
  while ( i<11 ) {
    s
    i
  }
}
int a
  if
    r
  els
    return a;
}
```

Example

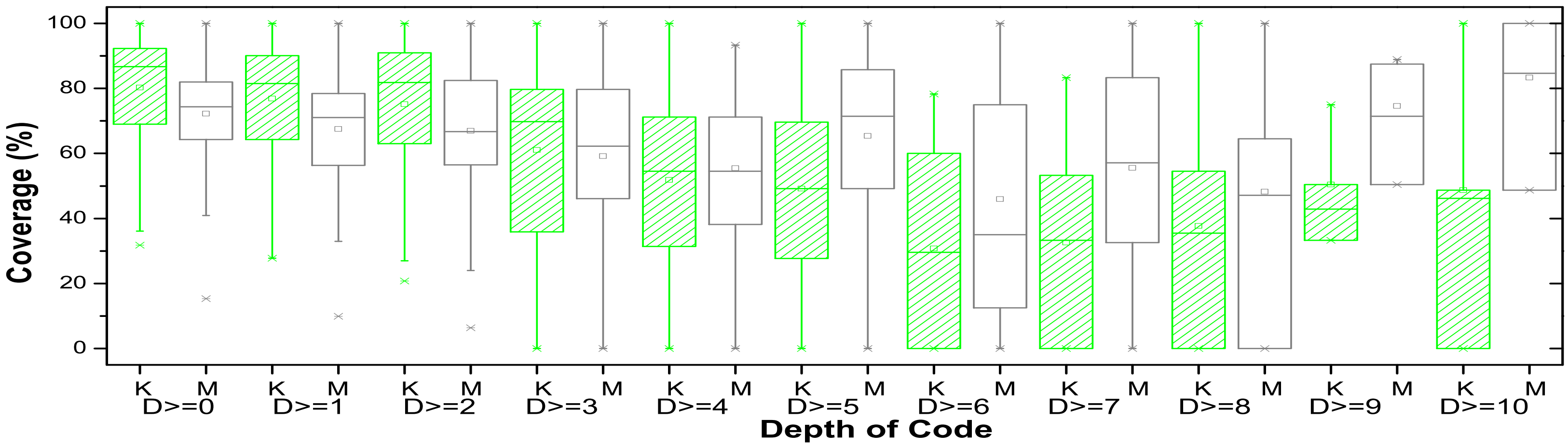
The difficulty to cover a statement can be measured by its depth in an Interprocedural Control Dependence Graph (ICDG)



Interprocedural Control Dependence Graph (ICDG)

Quantitative Analysis: Harder Tasks (Code)

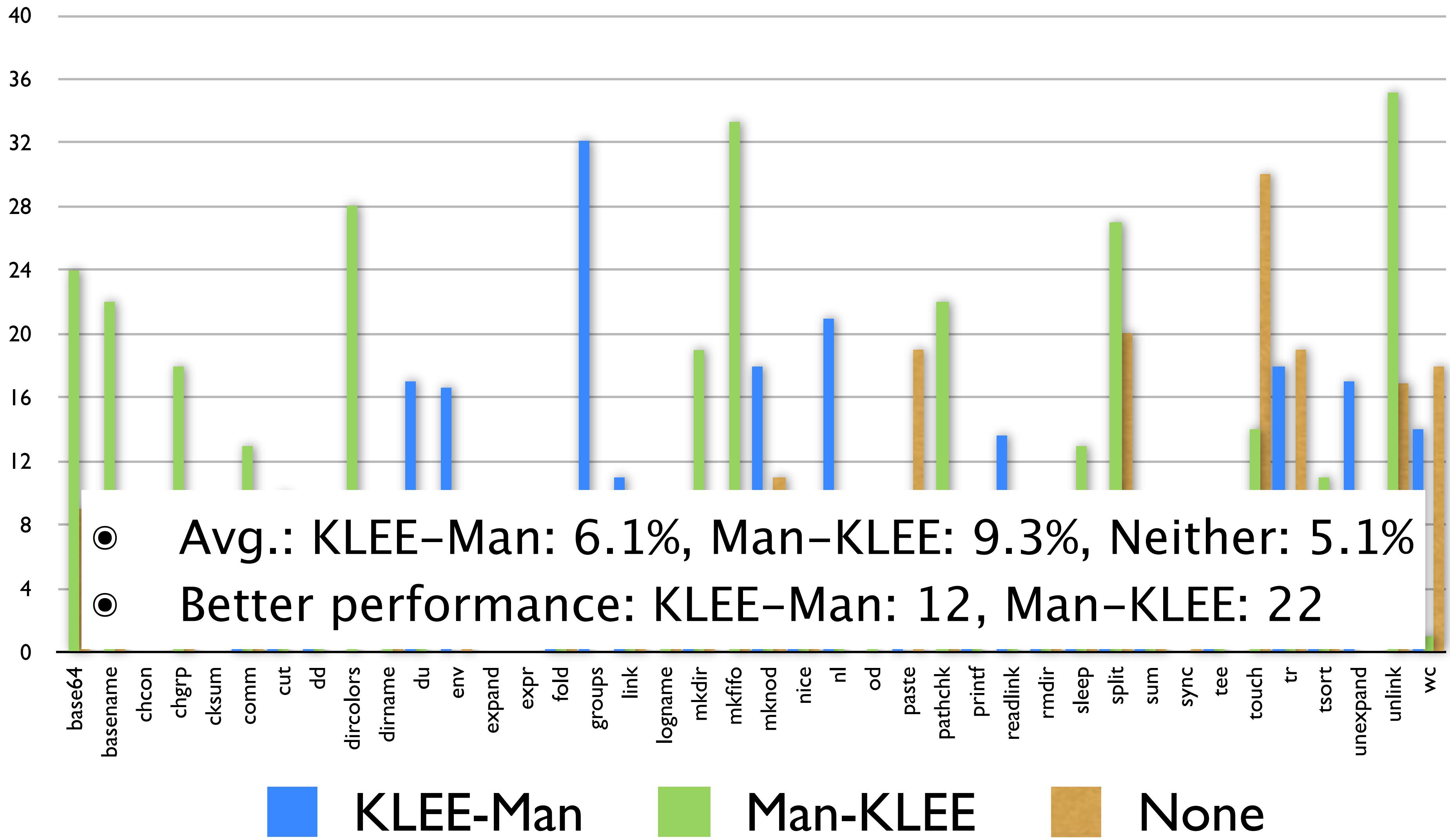
- Hard-to-cover code:



- Coverage of KLEE tests drops dramatically as depth goes lar… the same cover…

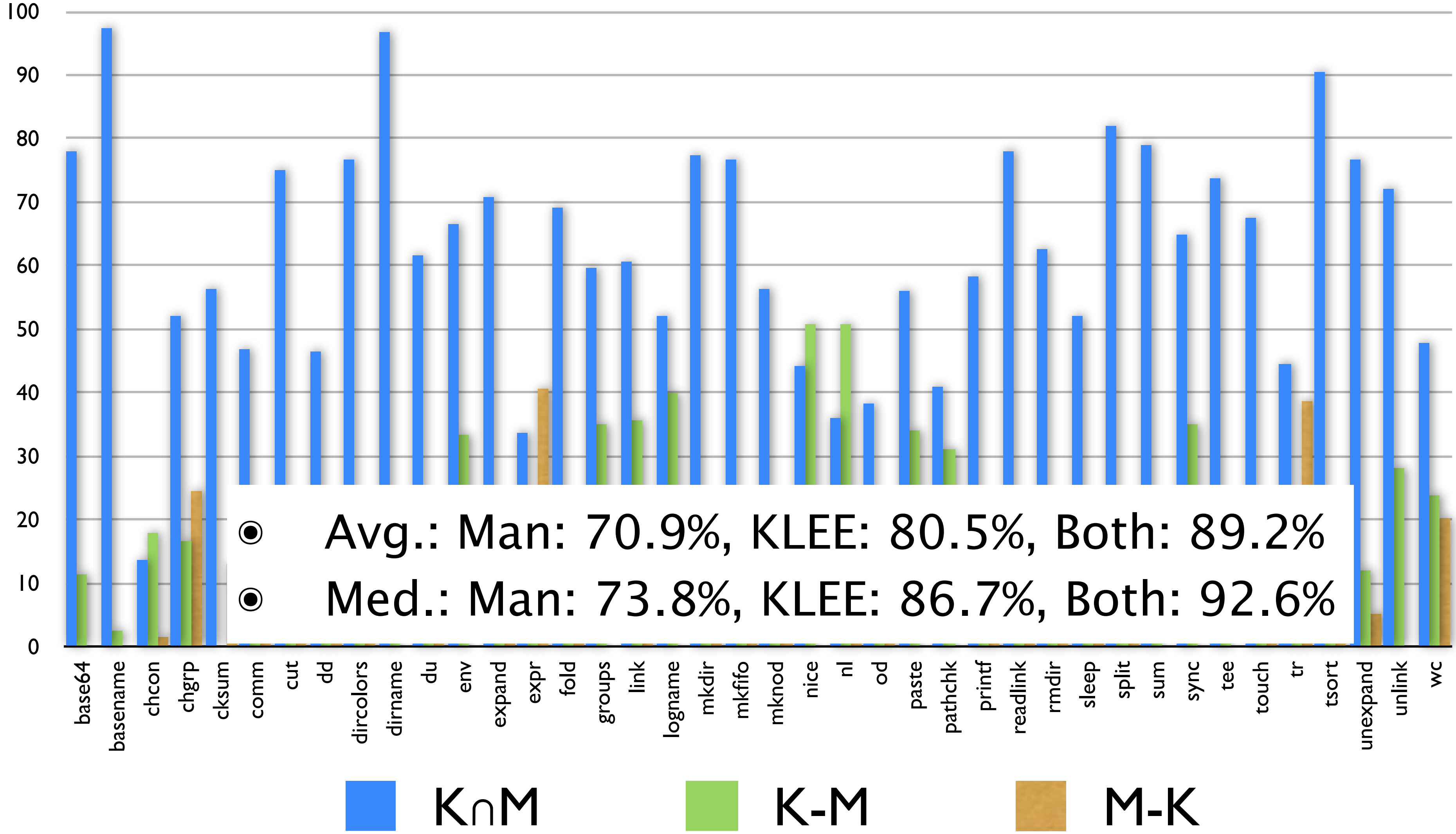
A hint of where human developers should help

Quantitative Analysis: Harder Tasks (Faults)



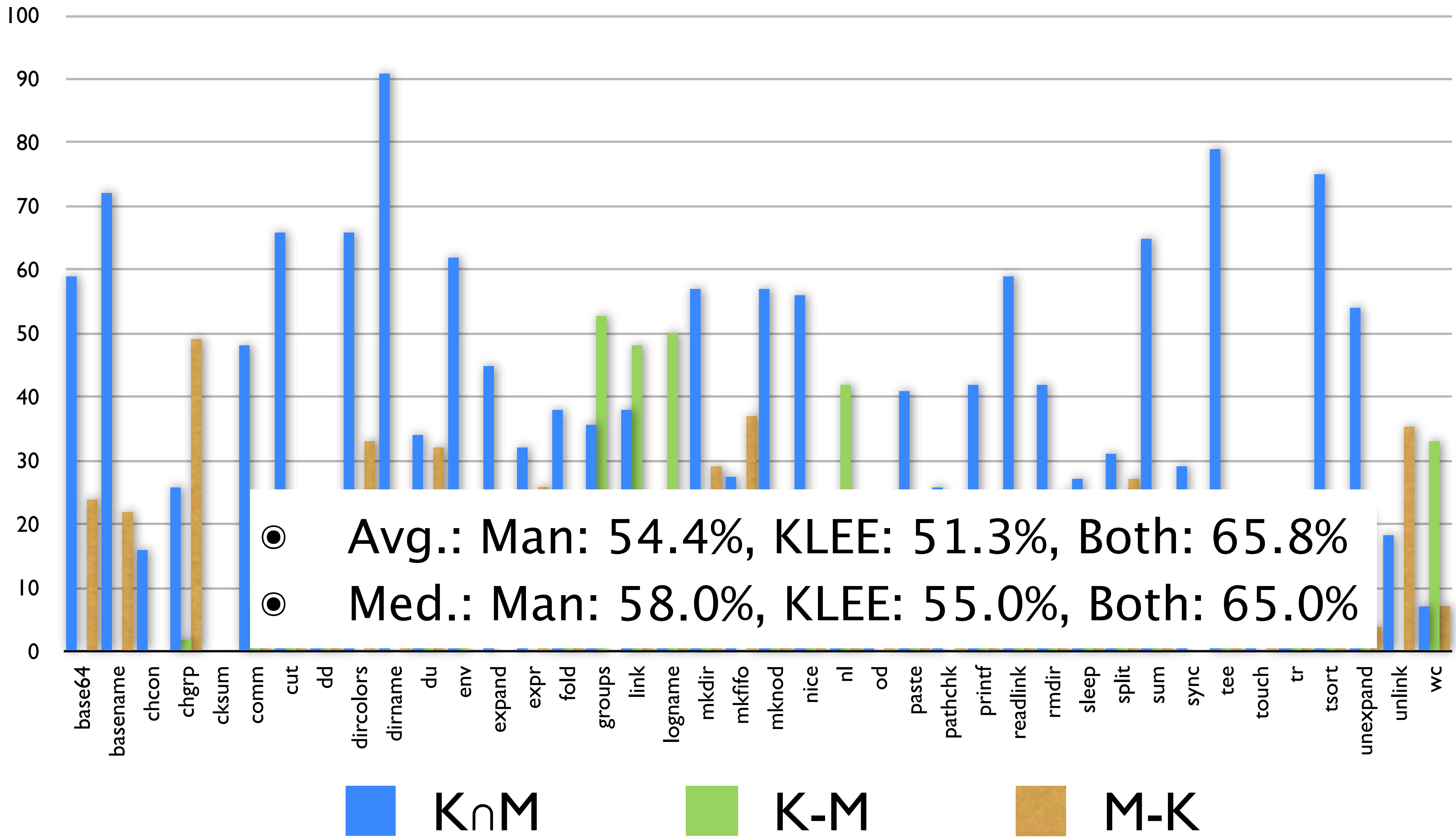
- Avg.: KLEE–Man: 6.1%, Man–KLEE: 9.3%, Neither: 5.1%
- Better performance: KLEE–Man: 12, Man–KLEE: 22

Quantitative Analysis: KLEE's Extra Value (Coverage)



- Avg.: Man: 70.9%, KLEE: 80.5%, Both: 89.2%
- Med.: Man: 73.8%, KLEE: 86.7%, Both: 92.6%

Quantitative Analysis: KLEE's Extra Value (Fault Detection)



- Avg.: Man: 54.4%, KLEE: 51.3%, Both: 65.8%
- Med.: Man: 58.0%, KLEE: 55.0%, Both: 65.0%

Qualitative Analysis

- Selection of code portion and mutation faults
  - KLEE-Man code:
    - 5 subjects with highest KLEE-Man code proportion
    - 5 longest code chunks
  - Man-KLEE code
    - 10 longest Man-KLEE code chunks
  - KLEE-Man / Man-KLEE mutation faults:
    - 10 Randomly selected mutants (at most 1 in each subject project)
    - Covered by both test suites

KLEE-Man Code

- Error Handling Code
  - Examples
    - **expr**: Manual tests fail to generate a bracket mismatch
    - **paste**: Manual tests fail to generate a file read error
- Exhausting all options
  - Example:
    - **nl**: Manual tests cover only 8 of 11 command options
    - **printf**: Manual tests fail to cover most escape characters

Man–KLEE Code

- Complex input structures:
  - Example:
    - **expr**: KLEE tests fail to include an expression containing a ":" operation and parsed correctly
    - **rmdir**: KLEE tests fail to generate a valid path
    - **tsort**: KLEE tests fail to include a tree structure requiring double rotation in balancing

KLEE-Man Mutants

- Why not detected by manual tests?
  - Major Reason: mutation affects only uncovered code
  - Example:

covered

```
if(optind + 1 < argc){     //mutate to “optind + 2”
    error (0, 0, ("extra operand \%s"), quote (…));
}
```

Fault detection condition:

$$(optind+1<argc) \mathrel{!=} (optind+2<argc)$$

$$optind+2 == argc$$

$$optind+1 < argc$$

Error Condition Not Covered by Manual Tests

Man-KLEE Mutants

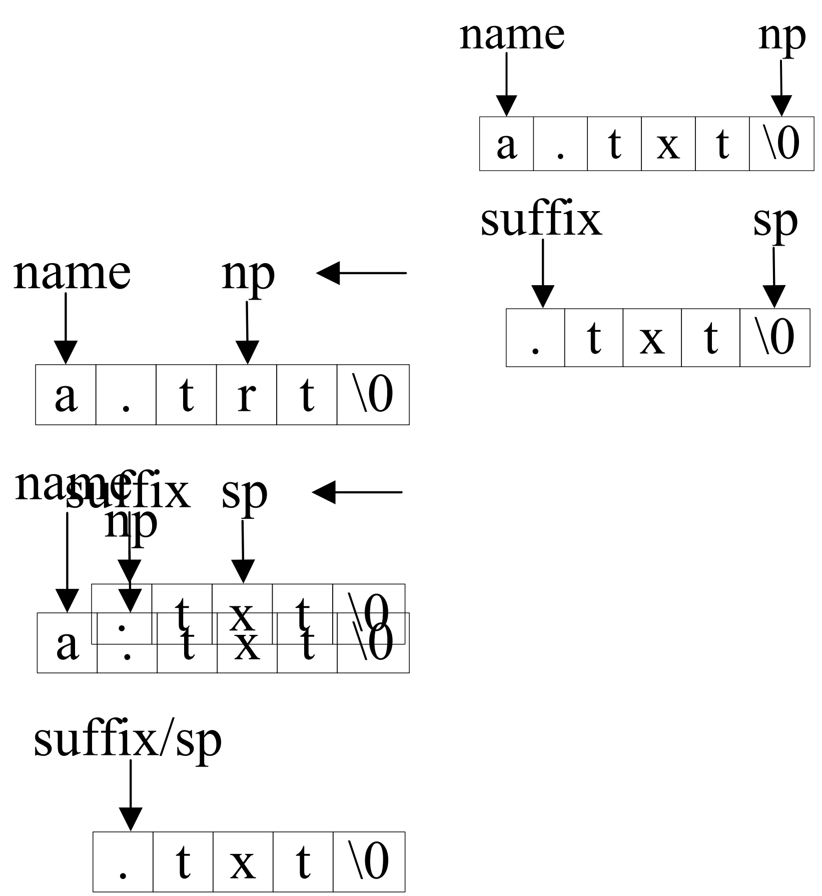
- Why not detected by KLEE tests?
  - Major reason: meaningful path not covered
  - Example: **basename**, try to remove suffix of a file name

```
char *np;
const char *sp;
```

```
np = name + strlen (name);
sp = suffix + strlen (suffix);
```

```
while (np > name && sp > suffix)
    if (*--np != *--sp)
        return;
```

```
if (np > name)
    *np = '$\slash$0';
```



Man-KLEE Mutants

- Why not detected by KLEE tests?
  - E.g., meaningful path not covered
  - Example: **basename**, try to remove suffix of a file name

```
char *np;
const char *sp;

np = name + strlen (name);
sp = suffix + strlen (suffix);

while (np > name && sp > suffix)
    if (*--np != *--sp)
        return;

if (np > name)
    *np = '$\slash$0';
```

KLEE Tests

np="a.txt" sp=""	np="a.txt" sp="bc"

KLEE tests: Although covering all statements, cannot execute the valid path

DANGER

Take-Home Message (Summary)

- While KLEE tests provide competitive coverage, their fault detection rates are lower
- Manual tests are better in covering hard-to-cover code and detecting hard-to-detect faults
- KLEE tests can provide non-trivial extra supports to manual tests in both coverage and fault detection
- KLEE is better at covering error handling code and exhausting a large number of options
- KLEE is worse at handling input with complicated structures, and may miss meaningful paths

Future Work

- Larger-scale quantitative and qualitative study
  - Larger and more subject programs
  - More test termination criteria
  - More measurements of code-coverage difficulty
  - Real-world faults
- More studies on other DSE tools
- Improving state-of-the-art DSE techniques
  - Knowledge of input formats
  - Integration of string constraint solvers
  - Guiding test-generation towards meaningful paths
  - …

Thanks! Questions?