Testing

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (2008)

Cristian Cadar, Daniel Dunbar, Dawson Engler

EXE: Automatically Generating Inputs of Death (2006)

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler

Presented by Oren Kishon
9/3/2014

Agenda

- Testing: Introduction
- KLEE + STP: Technical details
- Evaluation
- Related work
- Summary
- Discussion

Agenda

- Testing: Introduction
- KLEE + STP: Technical details
- Evaluation
- Related work
- Summary
- Discussion

Testing

- Purpose :
  - Verifying functional correctness (vs. spec)
  - Verifying software completeness - no crashes, memory leaks, assert violations…

Testing

- Purpose :
  - Verifying functional correctness (vs. spec)
  - Verifying software completeness - no crashes, memory leaks, assert violations…

Testing: example

Example [edit]

Consider the program below, which reads in a value and fails if the input is 6.

```
y = read()
y = 2 * y
if (y == 12)
   fail()
print("OK")
```

- Manual test creation: build test with input 6
- Large number of fail paths?
  - QA person works long hours…
  - Test auto-generation

Random input test generation

- ✔ Much more tests generated than manually
- ✘ Error path distribution is not uniform: Boundary values, zero-division…

Back to example: y being a 32 bit int

```
y = read()
y = 2 * y
if (y == 12)
    fail()
print("OK")
```

Symbolic execution

Example [edit]

Consider the program below, which reads in a value and fails if the input is 6.

```
y = read()
y = 2 * y
if (y == 12)
    fail()
print("OK")
```

y is symbolic: y = s

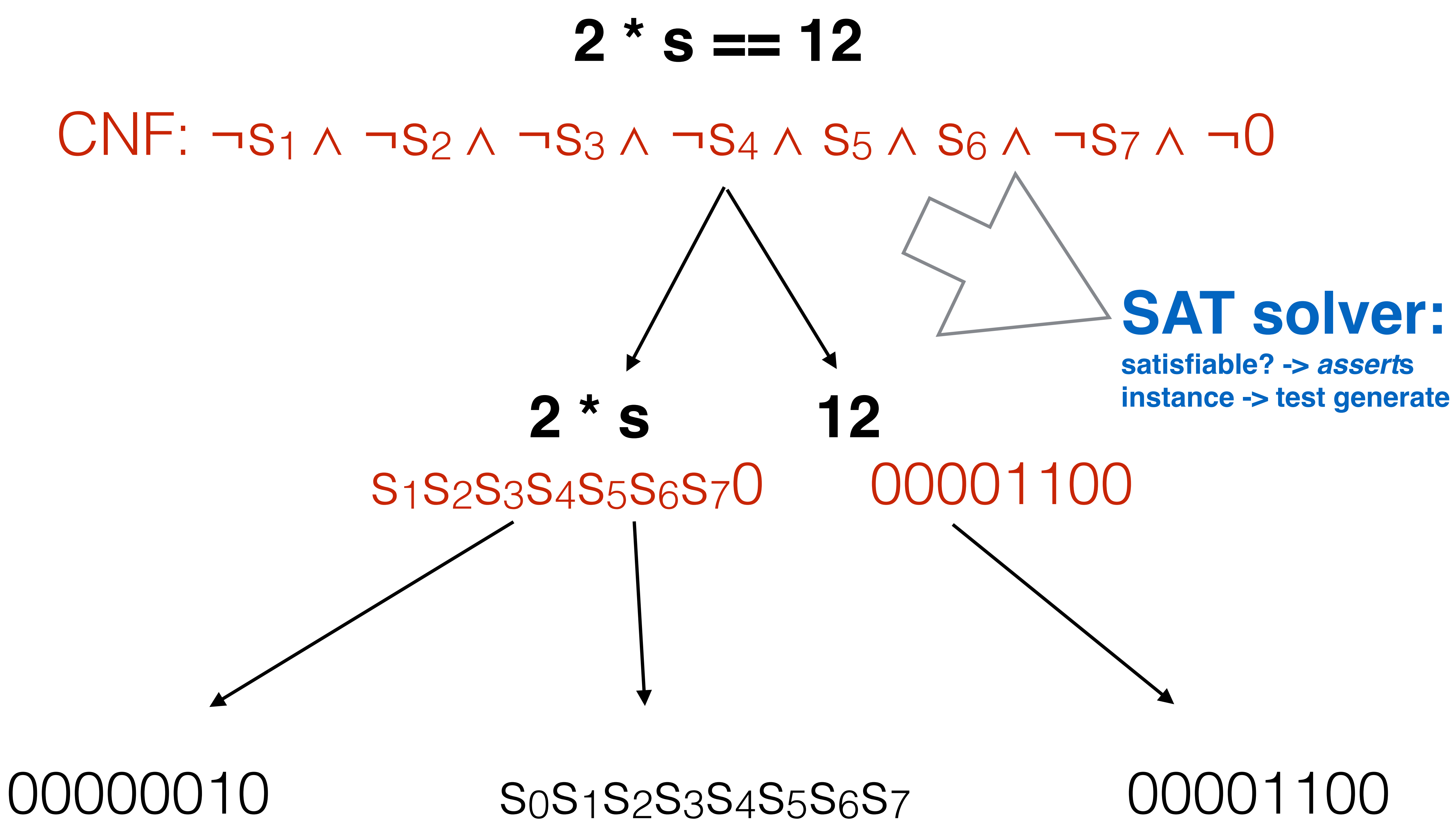
y = 2 * s // still symbolic

Fork execution, add constraints to each path

***true* path constraint: 2*s==12**

Need constraint solver

Constraint solver



KLEE: symbolic executer

- Architecture: compiles C code to LLVM byte code. Executes a symbolic interpreter.
- Map LLVM instructions to constraints. Constraint solver: STP.
- generates executable tests, independent of KLEE.
- Used to check all GNU Coreutils and covered 90% lines: more than 15 year on-going manual test suite - in 89 hours.

Introduction

- Before technical details - any questions?

Agenda

- Testing: Introduction
- KLEE + STP: Technical details
- Evaluation
- Related work
- Summary
- Discussion

Symbolic execution - a deeper look

- Definition: execution state
  - Line number
  - values of variables (symbolic/concrete): $x=s_1$, $y=s_2+3*s_4$
  - Path Condition (PC): conjunction of constraints (boolean formulas) over symbols: $s_1>0 \land \alpha_1+2*s_2>0 \land \neg(s_3>0)$

Symbolic Execution and Program Testing
JC King - 1976

Symbolic execution - a deeper look

- Execute assignment: evaluate RHS symbolically, assign to LHS as part of the the state.

- Execute IF (r) / then / else: fork

    - then: $PC \longleftarrow PC \wedge r$

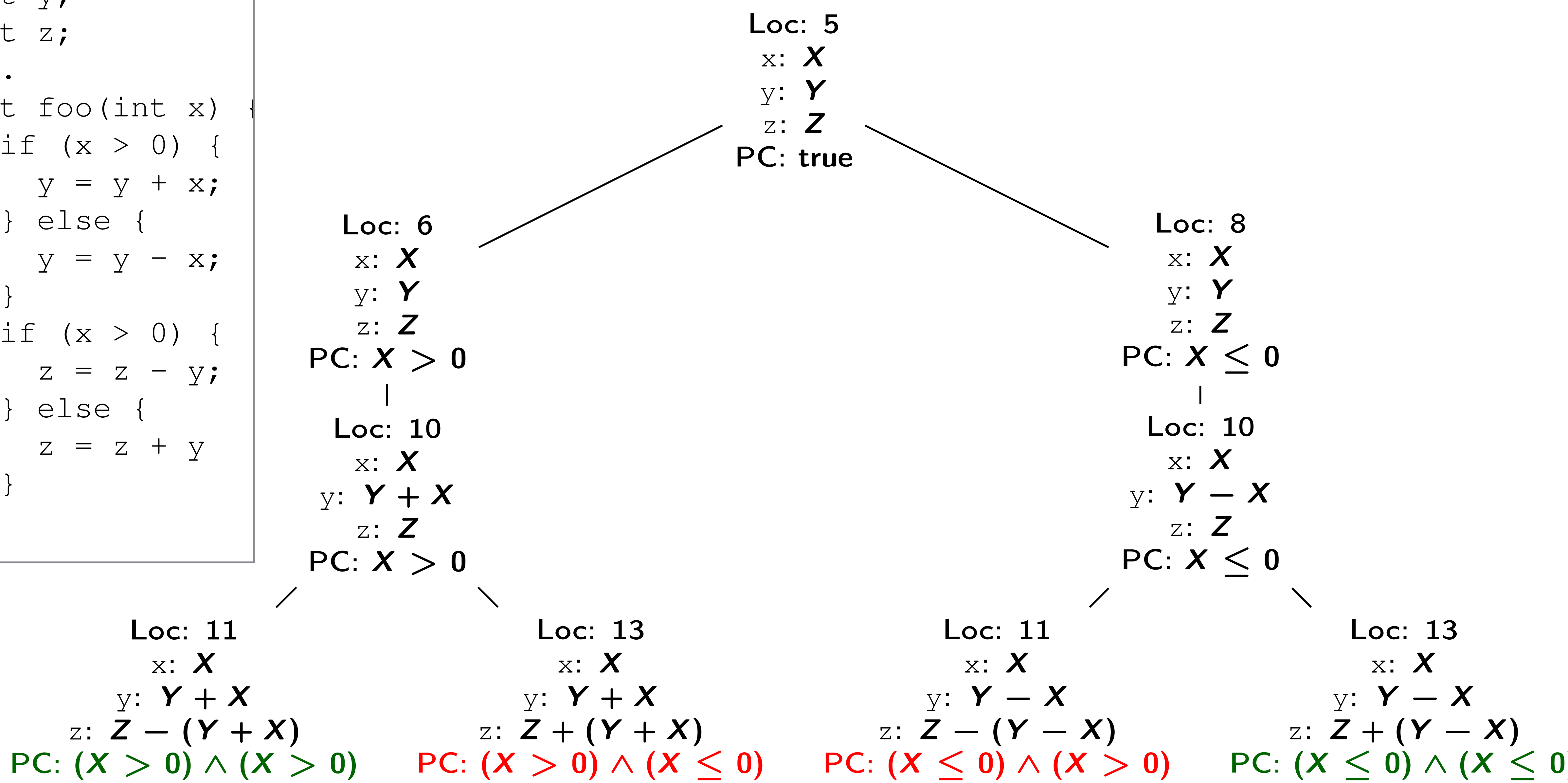
    - else: $PC \longleftarrow PC \wedge \neg r$

- Termination: solve constraint (supply values for symbols, for test generation)

Symbolic Execution and Program Testing
JC King - 1976

Execution tree

```
1   int y;
2   int z;
3   ...
4   int foo(int x) {
5     if (x > 0) {
6       y = y + x;
7     } else {
8       y = y - x;
9     }
10    if (x > 0) {
11      z = z - y;
12    } else {
13      z = z + y
14    }
15  }
```



Execution tree properties

- For each satisfiable leaf exists a concrete input for which the real program will reach same leaf ⇒

  can generate test

- PC's associated with any two **satisfiable** leaves are distinct ⇒ code coverage.

Symbolic Execution and Program Testing
JC King - 1976

KLEE - usage

Compile C programs to LLVM byte code and run KLEE interpreter with wanted parameters:

```
$ llvm-gcc --emit-llvm -c tr.c -o tr.bc

$ klee --max-time 2 --sym-args 1 10 10
   --sym-files 2 2000 --max-fail 1 tr.bc
```

KLEE - symbolic execution: tr (Minix)

```
void expand(char *arg, unsigned char *buffer) {          8
  int i, ac;                                              9
  while (*arg) {                                          10*
    if (*arg == '\\') {                                   11*
      arg++;
      i = ac = 0;
      if (*arg >= '0' && *arg <= '7') {
        do {
          ac = (ac << 3) + *arg++ − '0';
          i++;
        } while (i<4 && *arg>='0' && *arg<='7');
        *buffer++ = ac;
      } else if (*arg != '\0')
        *buffer++ = *arg++;
    } else if (*arg == '[') {                             12*
      arg++;                                              13
      i = *arg++;                                         14
      if (*arg++ != '-') {                                15!
        *buffer++ = '[';
        arg −= 2;
        continue;
      }
      ac = *arg++;
      while (i <= ac) *buffer++ = i++;
      arg++;        /* Skip ']' */
    } else
      *buffer++ = *arg++;
  }
}
. . .
```

3 symbolic arguments

```
int main(int argc, char* argv[]) {                        1
  int index = 1;                                          2
  if (argc > 1 && argv[index][0] == '-') {                3*
    . . .                                                 4
  }                                                       5
  . . .                                                   6
  expand(argv[index++], index);                           7
  . . .
}
```

Fork execution

Fork, constraint arg[0]==‘[‘

Detect bug (implicit array bounds checking) and generate test: input={“[“, “”, “”}

all 37 paths in 2 minutes

KLEE architecture

- Execution state:
  - Instruction pointer
  - Path condition
  - Registers, heap and stack objects
  - Above objects refer to trees of symbolic expressions.
  - Expressions are of C language: arithmetic, shift, dereference, assignment…
  - checks inserted at dangerous operations: division, dereferencing

STP - constraint solver

- A *Decision Procedure* for Bit-Vectors and Arrays
- "Decision procedures are programs which determine the satisfiability of logical formulas that can express constraints relevant to software and hardware"
- STP uses new efficient SAT solvers.

STP - constraint solver

- Treat everything as bit vectors - no types.
- Expressions on bit vectors: arithmetic (incl. non linear), bitwise operations, relational operations.
- All formulas are converted to DAGs of single bit operations (node for every bit!)

STP

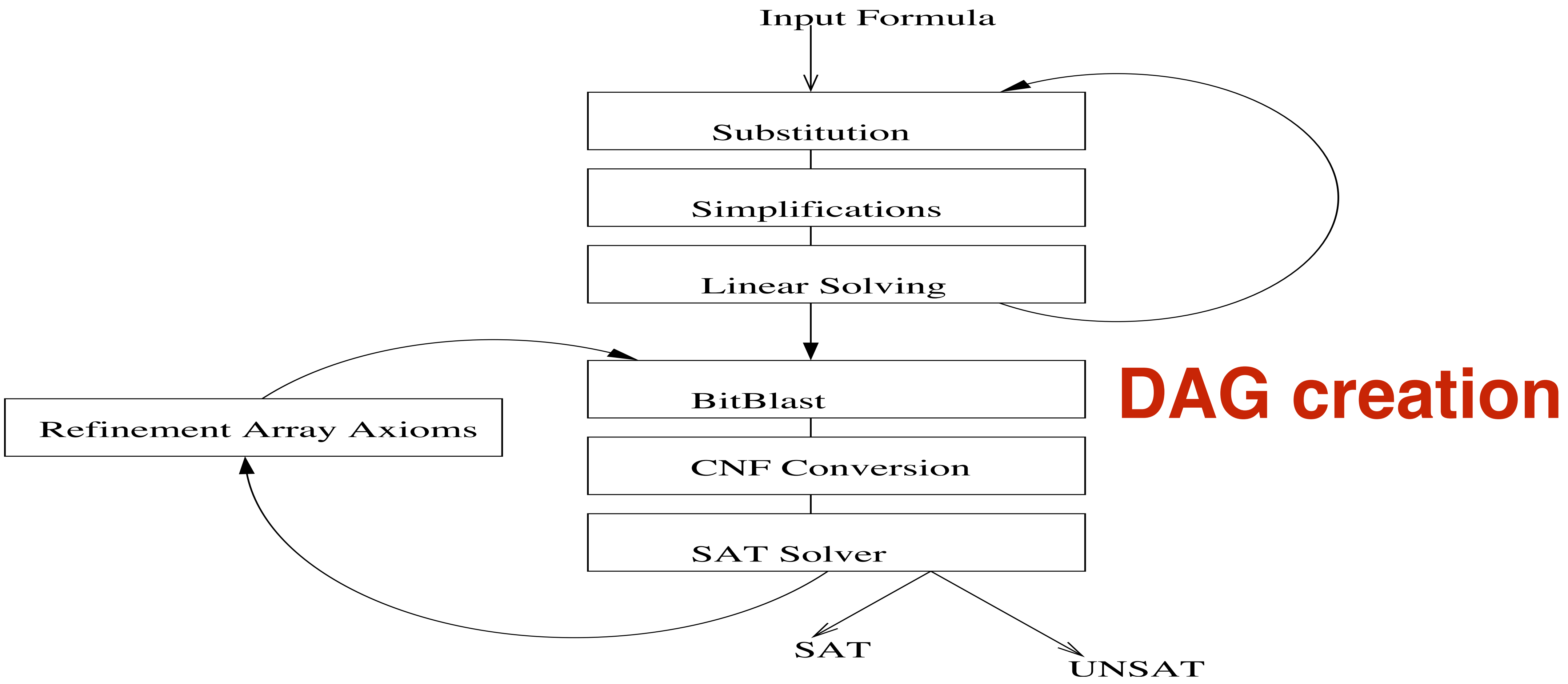


Fig. 1. STP Architecture

Query optimizations

- Constraint solver dominates run time (NP-complete problem in general…)
- Can pre-process calls to solver to make query easier
- Two complicated optimizations (presented next) and other basic ones (later on)

Query optimizations

Constraint independence

- Partition constraint set according to symbols
- Call solver with relevant subset only
- Example: $\{i < j, j < 20, k > 0\}$. a query of whether $i = 20$ just requires the first two constraints

Query optimizations

Counter example cache

- Cache results of previous constraint solver results
- If constraint set C has no solution and C ⊆ C’, then neither does C’
- If constraint set C has solution s and C’ ⊆ C, then C’ has solution s
- If constraint set C has solution s and C ⊆ C’, then C’ likely has solution s

State choosing heuristics:

- A big challenge of symbolic executing: path explosion
- Can’t cover all paths: need to choose wisely
- Use different choosing heuristic at each selection (using round robin)

State choosing heuristics:

Random Path Selection

- Maintain binary tree of paths
- When branch reached, traverse randomly from root to select state to execute
- Done to prevent starvation caused by large subtrees (i.e loops with symbolic condition)

State choosing heuristics:
Coverage-optimize search

- Compute state weight using:
  - Minimum distance to an uncovered instruction
  - Call stack of the state
  - Whether the state recently covered new code

Environment modeling

- Another big challenge of symbolic executing: symbolizing file systems, env. variables, network packets, etc.
- KLEE's solution: model as much as you can. modeling means to costumize code of system calls (e.g. open, read, write, stat, lseek, ftruncate, ioctl): 2500 lines of modeling code.

Environment modeling

- File system examples
  - Read concrete file with symbolic offset: read() is wrapped with pread()
  - Open symbolic file-name:
    - Program was initiated with a symbolic file system with up to N files (user defined).
    - Open all N files + one open() failure

Environment modeling

- How to generate tests after using symbolic env:
  - Except of supplying input args, supply an description of symbolic env for each test path.
  - A special driver creates real OS objects from the description

Other optimizations

- Copy On Write for forking - object level, not page level
- Pointer to many possible objects - branch all
- Query optimizations
  - Constraint set simplification: $\{x<10\}$, $x==5 \Rightarrow \{x==5\}$
  - Implied Value Concretization: $\{x+1==10\} \Rightarrow x = 9$

KLEE

- Questions?

Agenda

- Testing: Introduction
- KLEE + STP: Technical details
- Evaluation
- Related work
- Summary
- Discussion

Evaluation - Metrics

- Line coverage, only executable: ELOC percentage
- Doesn’t measure actual conditional paths used
- Used also because the gcov profiler outputs it and its a common tool among testing tools.

Coreutils

- All 89 Coreutils programs ran with command:
```
./run <tool-name> --max-time 60
                  --sym-args 10 2 2
                  --sym-files 2 8
                  [--max-fail 1]
```

Coreutils

76.9% line coverage of all 89 Coreutils programs

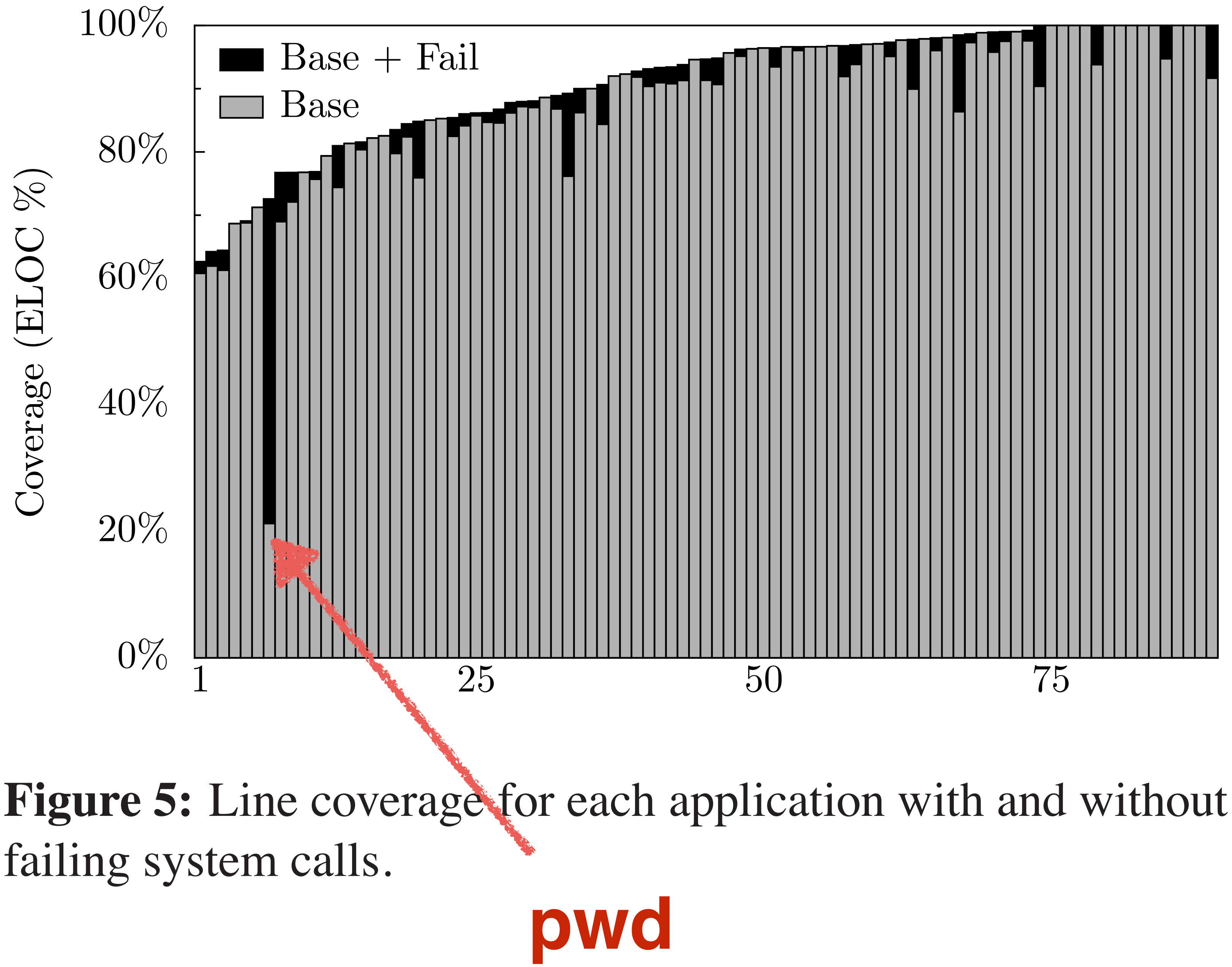
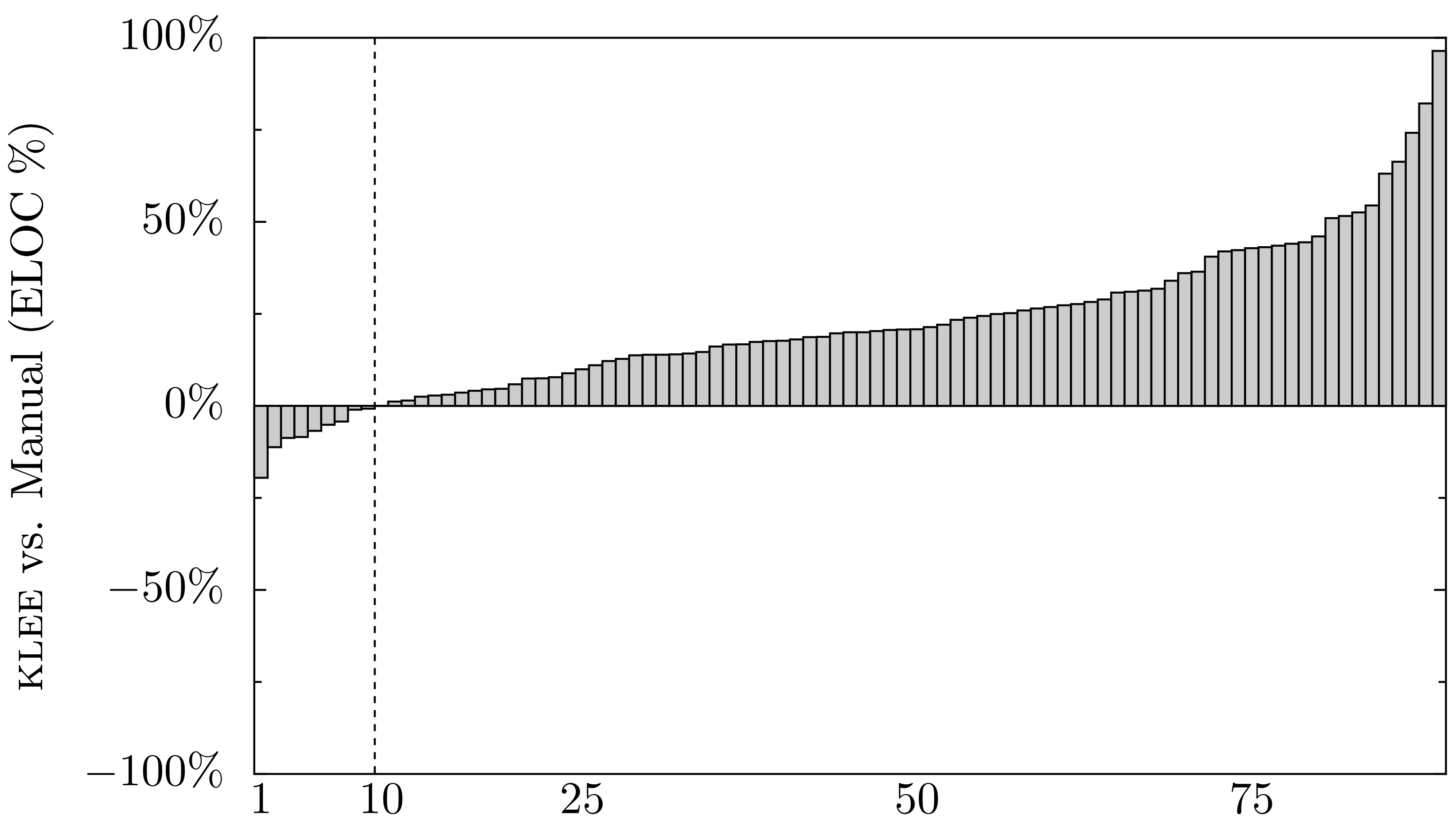


Figure 5: Line coverage for each application with and without failing system calls.

KLEE vs. manual suite

	COREUTILS		BUSYBOX	
Coverage (w/o lib)	**KLEE tests**	**Devel. tests**	**KLEE tests**	**Devel. tests**
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%



$(L_{KLEE} - L_{man}) / L_{total}$

output tests of bugs

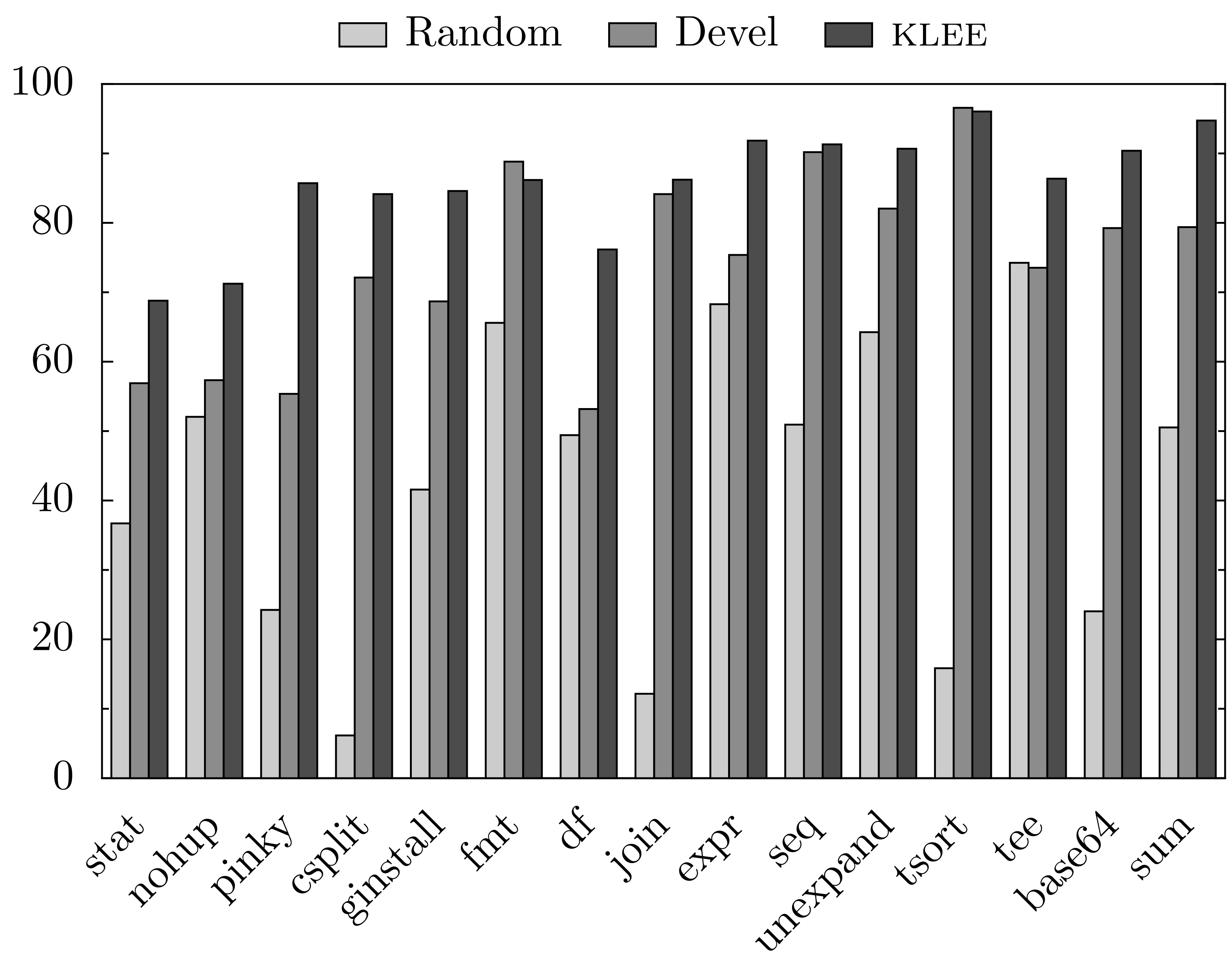
```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

Since 1992

Cause: modulus negative

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

KLEE vs. random



Observation: random quickly gets the cases it can, and then revisits them over and over

Program equivalence

- Needed in:
  - standard implementation
  - New version testing

Program equivalence

Need to manually wrap programs:

```
unsigned mod_opt(unsigned x, unsigned y) {
    if((y & −y) == y) // power of two?
        return x & (y−1);
    else
        return x % y;
}
unsigned mod(unsigned x, unsigned y) {
    return x % y;
}
int main() {
    unsigned x,y;
    make_symbolic(&x, sizeof(x));
    make_symbolic(&y, sizeof(y));
    assert(mod(x,y) == mod_opt(x,y));
    return 0;
}
```

Program equivalence

Coreutils vs. Busybox

Interesting mismatches:

Input	BUSYBOX	COREUTILS
`comm t1.txt t2.txt`	[does not show difference]	[shows difference]
`tee -`	[does not copy twice to stdout]	[does]
`tee "" <t1.txt`	[infinite loop]	[terminates]
`cksum /`	`"4294967295 0 /"`	`"/:  Is a directory"`
`split /`	`"/:  Is a directory"`	
`tr`	[duplicates input on stdout]	`"missing operand"`
`[ 0 ‘‘<’’ 1 ]`		`"binary operator expected"`
`sum -s <t1.txt`	`"97 1 -"`	`"97 1"`
`tail -2l`	[rejects]	[accepts]
`unexpand -f`	[accepts]	[rejects]
`split -`	[rejects]	[accepts]
`ls --color-blah`	[accepts]	[rejects]
t1.txt: a *t2.txt:* b		

Agenda

- Symbolic testing: Introduction
- KLEE + STP
- Metrics, experimental methods, results
- Related work
- Discussion

Related work

- Similar to KLEE path choose heuristic: generational search (Godefroid, P., Levin, M. Y., And Molnar, D. Automated whitebox fuzz testing)
- Give score to states according to line coverage they done.
- But uses random values when symbolic execution is hard (environment interfacing)

Related work

- Concolic (concrete/symbolic) testing:
  Run on concrete random inputs. In parallel, execute symbolically and solve constraints. Generate inputs to other paths than the concrete one along the way.

  - Godefroid, Patrice; Nils Klarlund, Koushik Sen (2005). "DART: Directed Automated Random Testing”

  - Sen, Koushik; Darko Marinov, Gul Agha (2005). "CUTE: a concolic unit testing engine for C"

Agenda

- Symbolic testing: Introduction
- KLEE + STP
- Metrics, experimental methods, results
- Related work
- Discussion

Discussion

- Code coverage is not good enough as a metric. Path coverage is preferred (admitted in the paper)
- Symbolic environment interaction - how reliable can the costume modeling really be? think about concurrent programs, inter-process programs, etc.
- What is more commonly needed - functional testing or security/completeness/crash testing?

Added subject

KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment

Raimondas Sasnauskas*, Olaf Landsiedel*, Muhammad Hamad Alizai*,

Carsten Weise‡, Stefan Kowalewski‡, Klaus Wehrle*

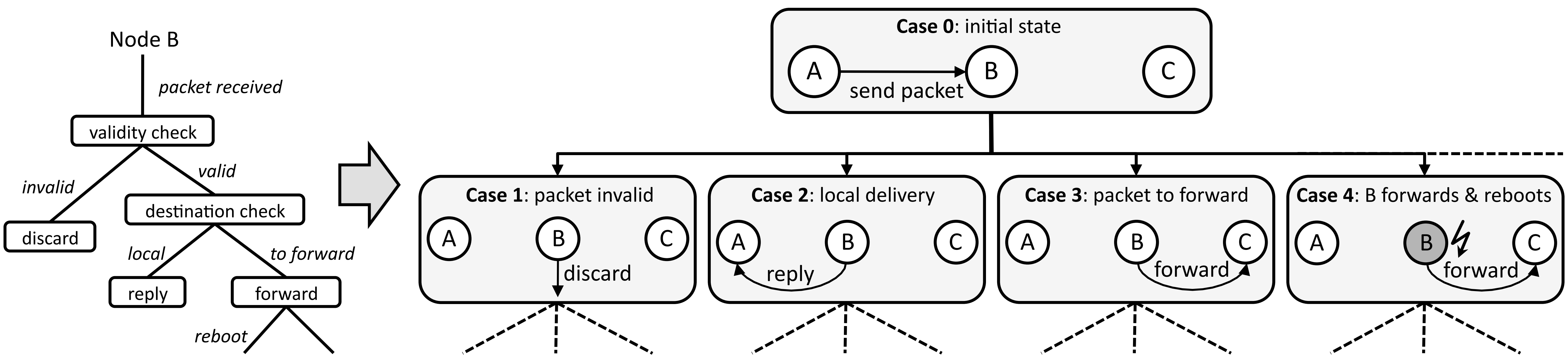
*Distributed Systems Group, ‡Embedded Software Laboratory RWTH Aachen University, Germany

- Sensor networks: network of nodes with unreliable, resource-constrained devices
- On comm loss: hard to find/fix
- Packet loss/corruption, often reboots

KleeNet

- Node model - same as Klee's environment model. Focuses on TCP failures (invalid packets, etc)
- **Network model**: Holds status of network and packet passing. Injects network wide failures.
- Essentially its a testing tool for distributed systems

KleeNet



Symbolic protocol execution

Injected node reboot - creates new node!

KleeNet

- Insight - after all, complicated systems need customizing tests…