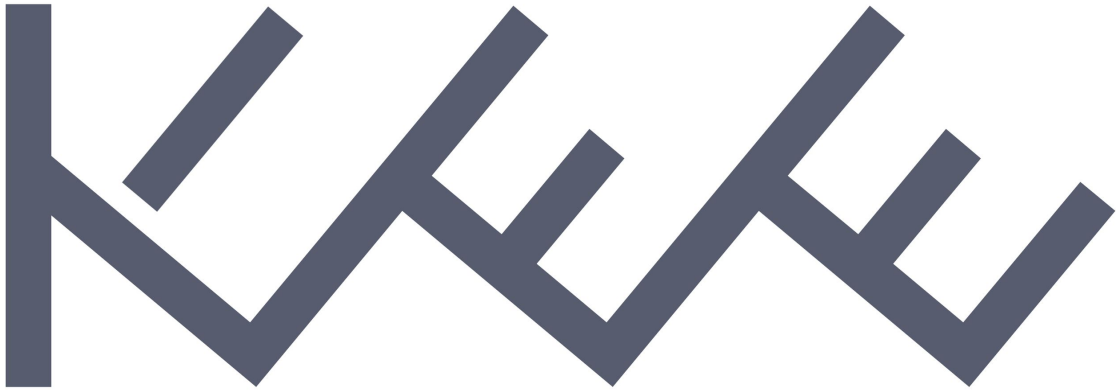


KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler
Stanford University

Ben Lowman

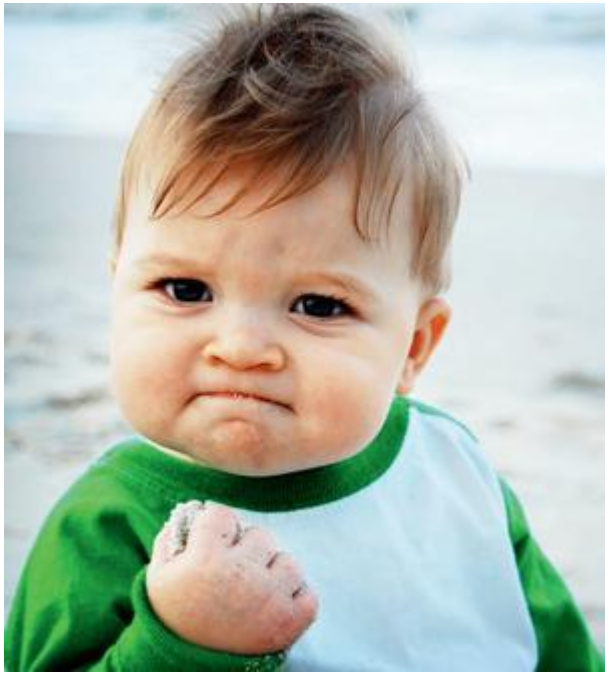
Select graphs borrowed from conference presentation.

Systems Code is Hard!

- Complex control flow
- C “type system”
- Pointers!
- Environmental dependencies
- Resiliency requirements
- Resource constraints

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

KLEE to the rescue!

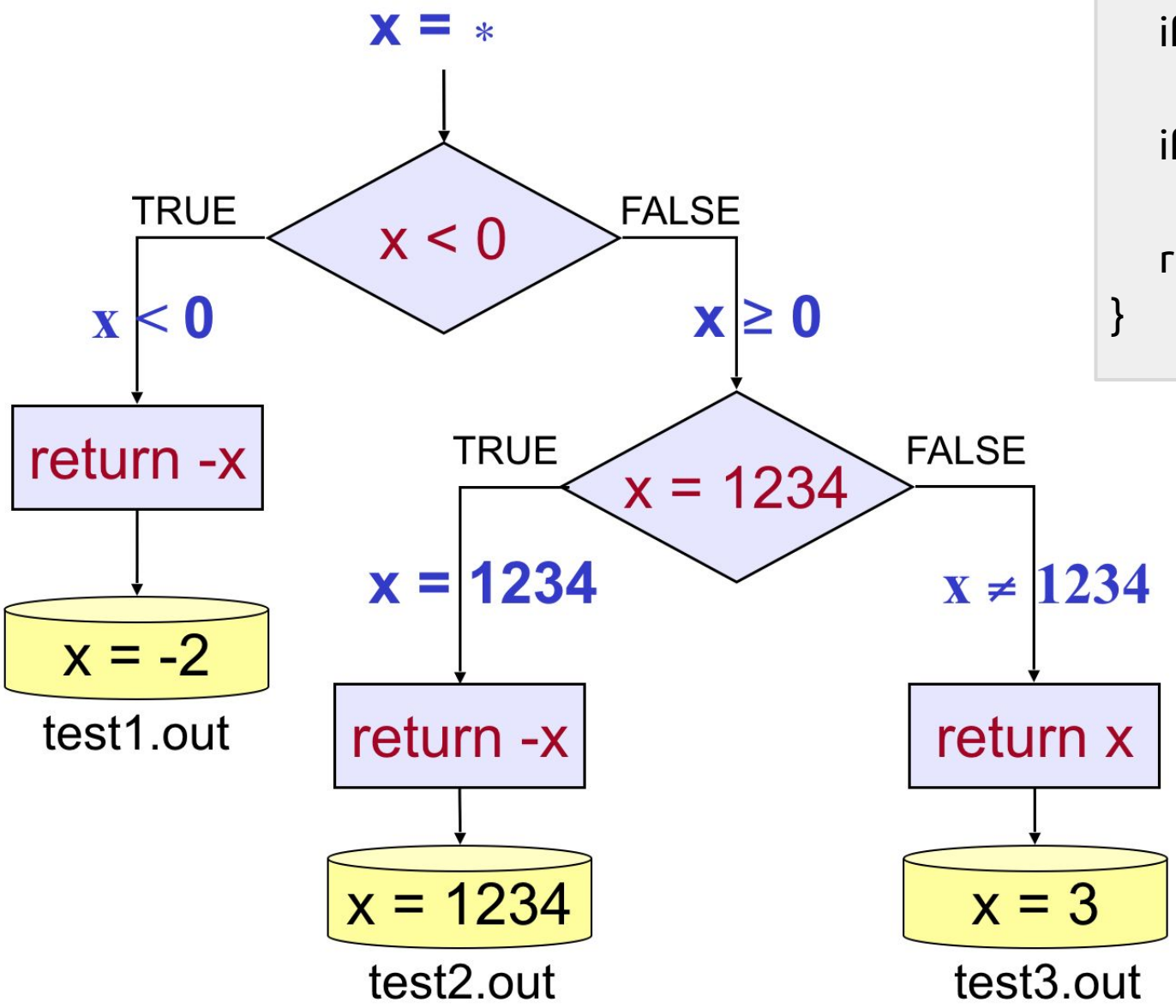


OSDI 2008, Best Paper Award

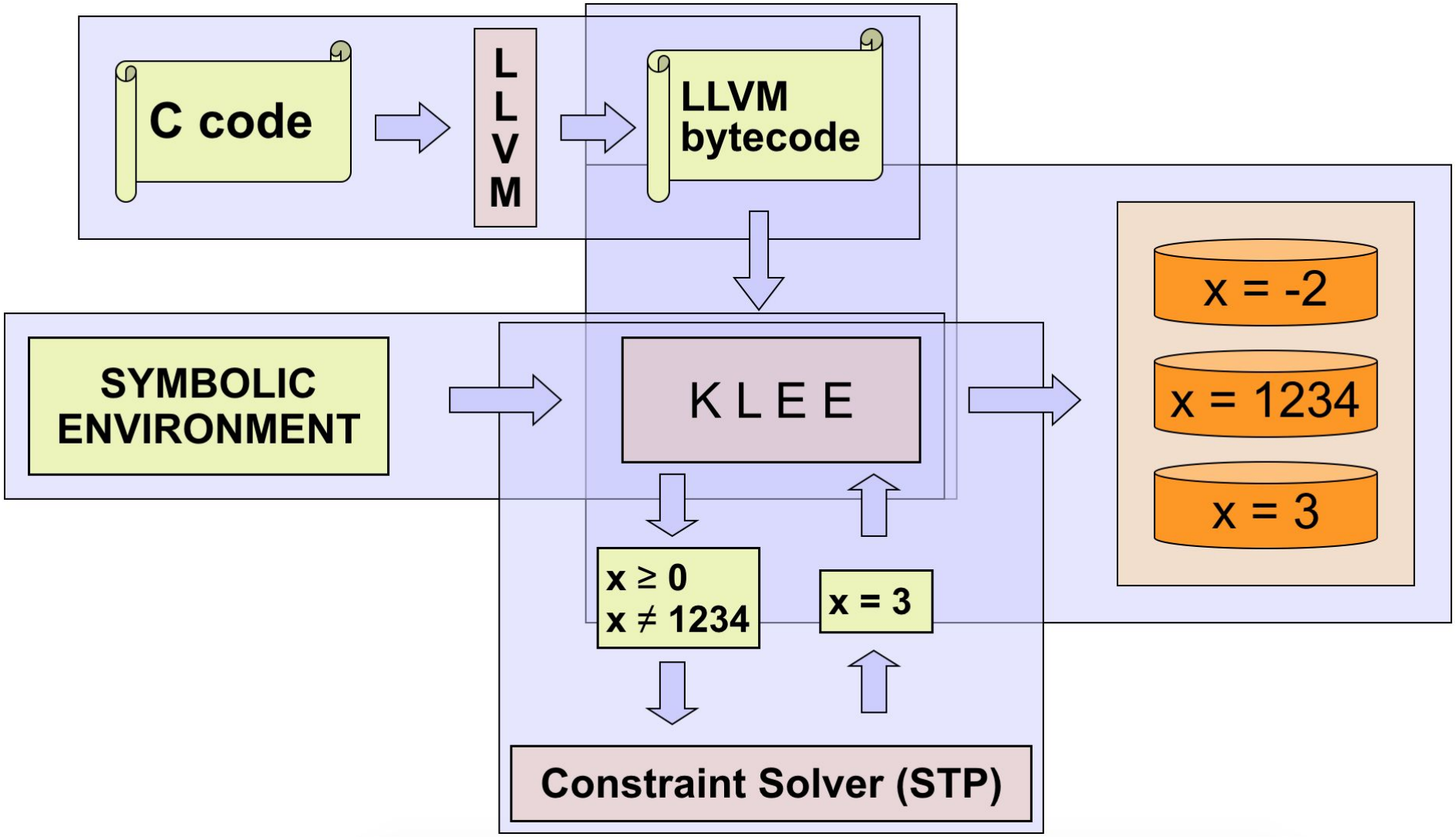
- Leverages symbolic execution, constraint solving
- Automatically generates high coverage test suites
  - 90% on average in approximately 160 applications
- Finds deep bugs in complex systems programs
  - Including “hard”, “high level” bugs

Symbolic Execution

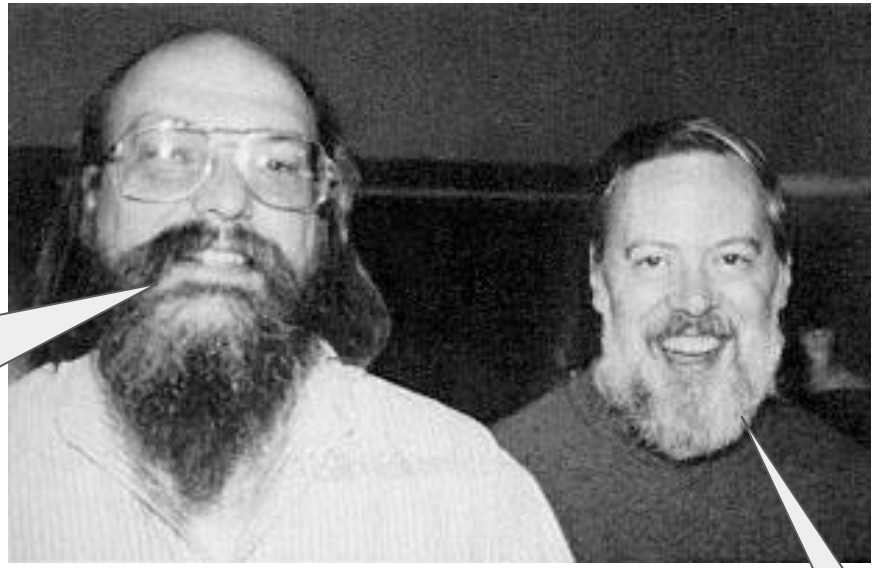
```
int bad_abs(int x)
{
    if (x < 0)
        return –x;
    if (x == 1234)
        return –x;
    return x;
}
```



KLEE Architecture



Does is scale?



Environment?

lol k

Scalability Challenges

- Exponential number of paths (path explosion)
- Constraint solving is NP-Complete
- Environment is arbitrarily complex

jmp <input>

Exponential Search Space

State representation takes up a lot of space

- Copy on write for memory objects
- Common heap structures are shared among states

Exploration can easily get “stuck” -- use search heuristics

- Coverage optimized search
- Random path search

Expensive Constraint Solving

- Dominates Runtime -- 92 percent!
  - NP-Complete
  - Invoked at every branch/assert
  - Can’t really be avoided
- Several optimizations
  - Expression rewriting / simplification / concretization
  - Constraint independence
  - Predicate caching

Name	License type	Web service	Library	Standalone
ACL2	3-clause BSD	No	No	Yes
Otter	Public Domain	Via System on TPTP	Yes	No
j'Imp	?	No	No	Yes
Metis	?	No	Yes	No
MetiTarski	MIT	Via System on TPTP	Yes	Yes
Jape	?	Yes	Yes	No
PVS	?	No	Yes	No
Leo II	?	Via System on TPTP	Yes	Yes
EQP	?	No	Yes	No
SAD	?	Yes	Yes	No
PhoX	?	No	Yes	No
KeYmaera	GPL	Via Java Webstart	Yes	Yes
Gandalf	?	No	Yes	No
Tau	?	No	Yes	No
E	GPL	Via System on TPTP	No	Yes
SNARK	Mozilla Public License	No	Yes	No
Vampire	?	Via System on TPTP	Yes	Yes
Waldmeister	?	Yes	Yes	No
Saturate	?	No	Yes	No
Theorem Proving System (TPS)	?	No	Yes	No
SPASS	FreeBSD license	Yes	Yes	Yes
IsaPlanner	GPL	No	Yes	Yes
KeY	GPL	Yes	Yes	Yes
Meta Theorem	?	No	No	Yes
Princess	GPL	Via Java Webstart and System on TPTP	Yes	Yes

Z3 Example

Constraint Independence

Branches usually only reference a few program variables

```
…
    …
    if (x < 4)
    {
            …
    }
…
```

$x + z < 120$

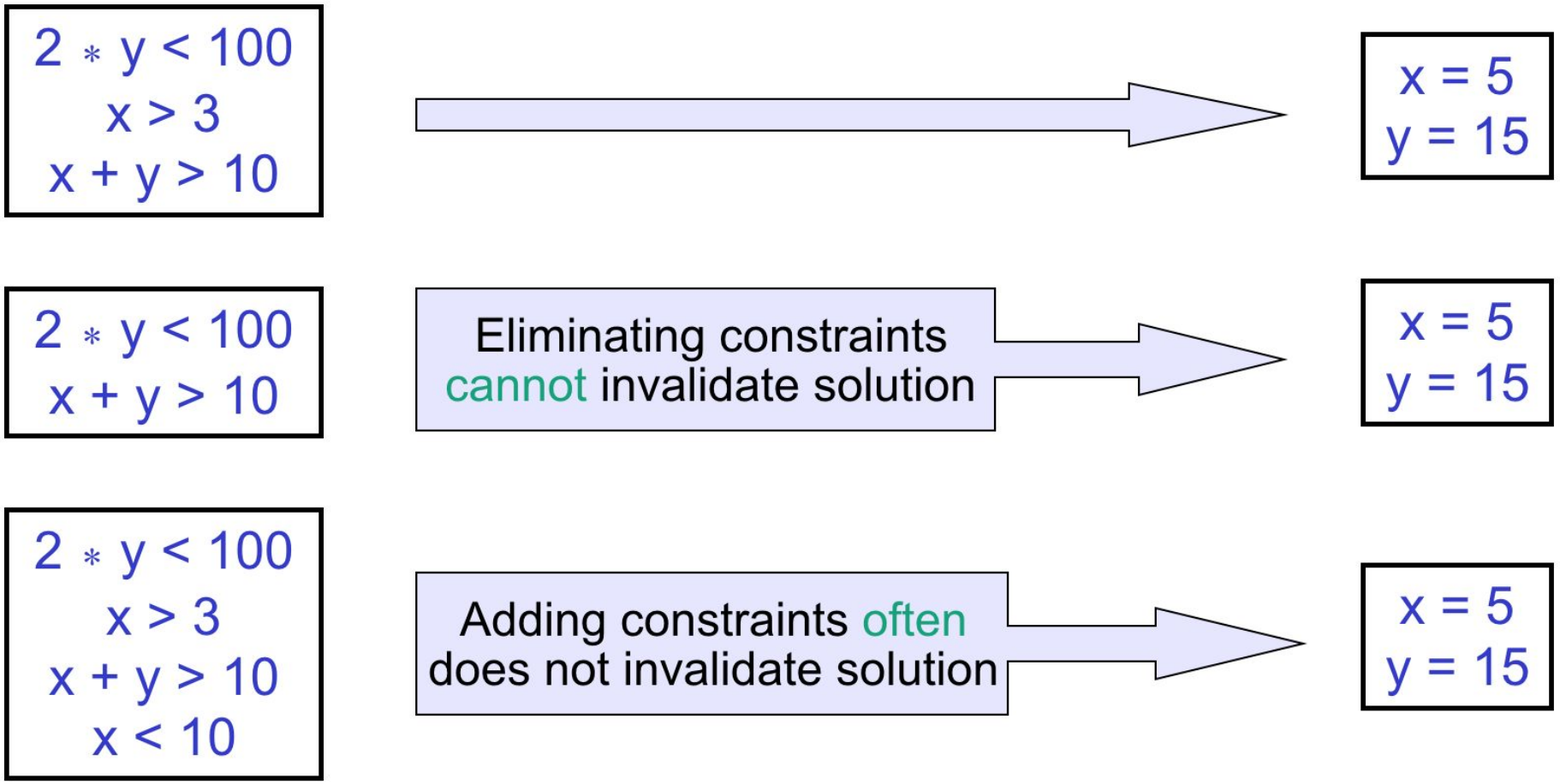
$a \& b = a$

$argc > 1$

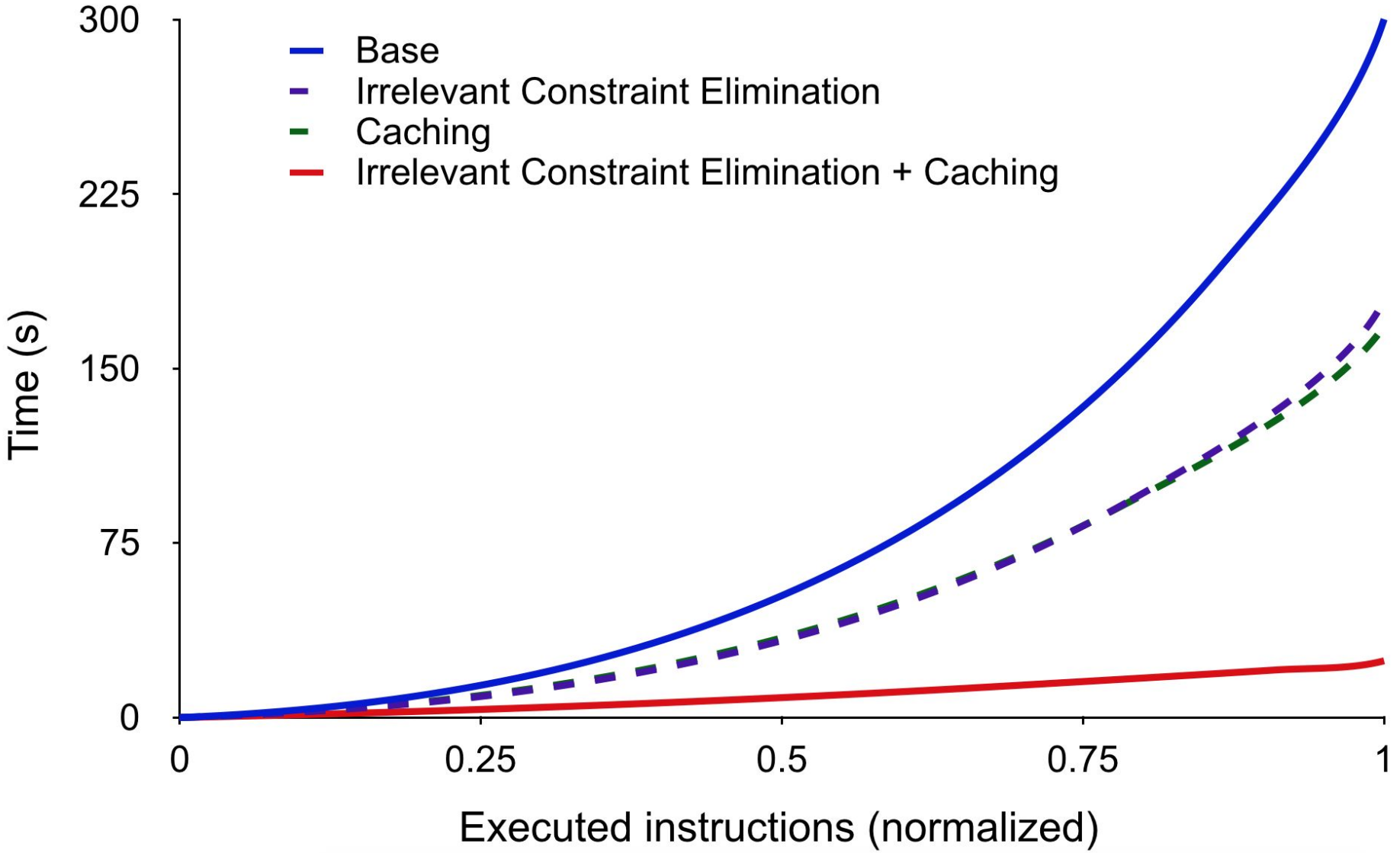
$x < 4$

Predicate Caching

Saving the results of previous predicates can speed up future queries to theorem prover



Huge Speedup



Handling the Environment

- Environment often exposes edge cases
  - *What we really care about*
- Extremely important if tool is to be useful in “real world”

```
void foo(int fd, char input)
{
    fwrite(fd, input, 1);
    char c;
    fread(fd, &c, 1);
    If (c == input)
        *0;
}
```

Environment Models

```
int fd  = open("file.txt", O_RDONLY);
```

If all arguments are concrete, forward to the host operating system and proceed with a concrete value

```
int fd  = open(my_file, O_RDONLY);
```

Otherwise, a user-programmed model is created to handle abstract interactions with the environment.

Sample File System Model

abstract

concrete

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :   if (is_invalid(fd)) {
3 :     errno = EBADF;
4 :     return -1;
5 :   }
6 :   struct klee_fd *f = &fds[fd];
7 :   if (is_concrete_file(f)) {
8 :     int r = pread(f->real_fd, buf, count, f->off);
9 :     if (r != -1)
10:       f->off += r;
11:     return r;
12:   } else {
13:     /* sym files are fixed size: don't read beyond the end. */
14:     if (f->off >= f->size)
15:       return 0;
16:     count = min(count, f->size - f->off);
17:     memcpy(buf, f->file_data + f->off, count);
18:     f->off += count;
19:     return count;
20:   }
21: }
```

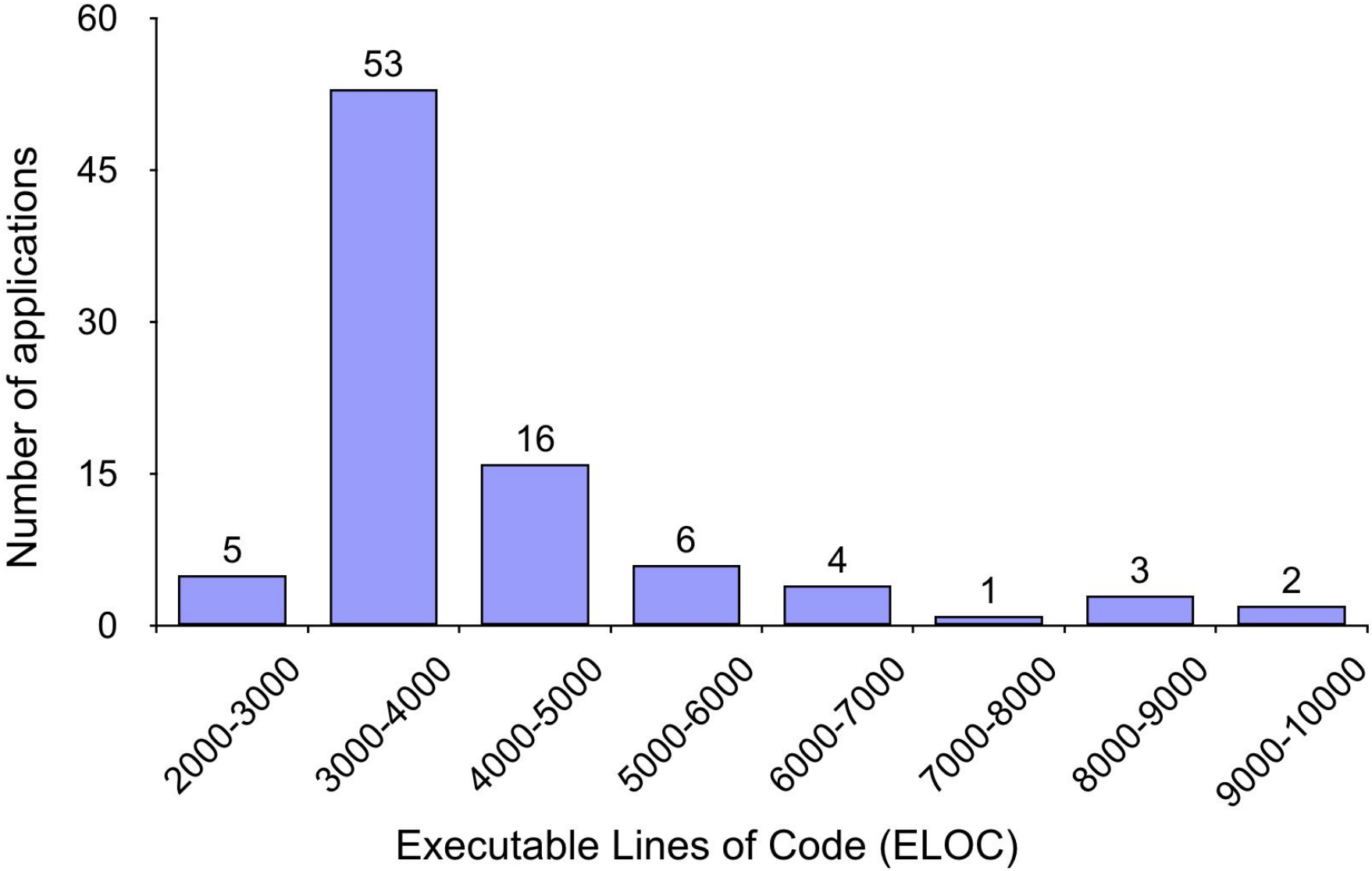
"Out of the box" models for input, output, pipes, links, ttys and over 40 system calls (2500 LOC)

Test Suite

- GNU Coreutils:
  - 89 apps installed on almost all UNIX systems
  - Variety of functions, authors, env. interaction
  - Heavily tested, mature code
- Busybox:
  - 75 “coreutils”
  - lightweight clone of GNU coreutils
  - Lots of overlapping functionality

1992-10-31	*Jim Meyering*	**Add parentheses to expressions like (c = *p++) as per...**
1992-10-31	*Jim Meyering*	**Add parentheses to expressions like (c = *p++) as per...**
1992-10-31	*Jim Meyering*	**(adjust_blocks): Convert to a macro. The static**
1992-10-31	*Jim Meyering*	**Initial revision**

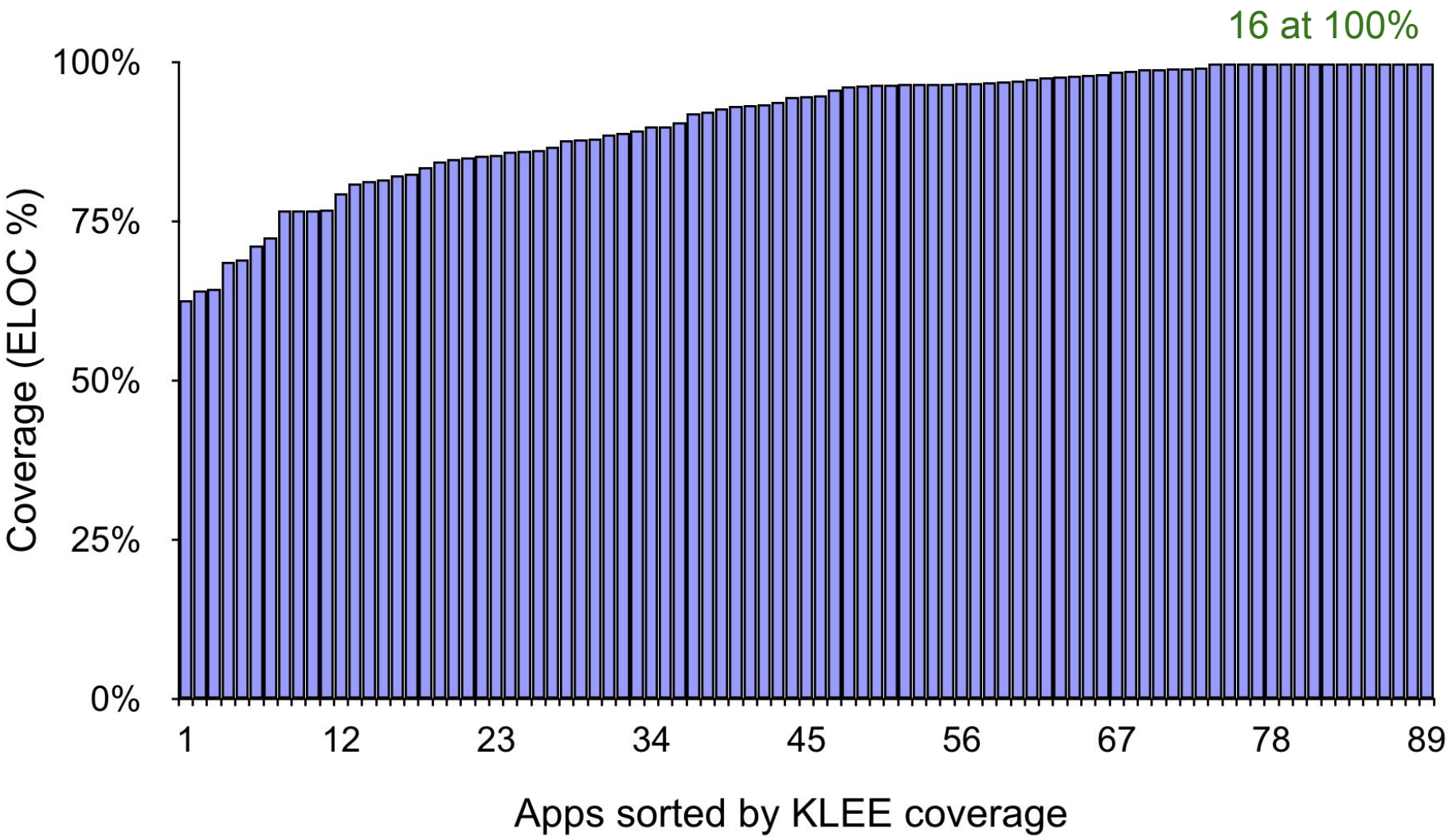
Test Suite (cont.)



Evaluation

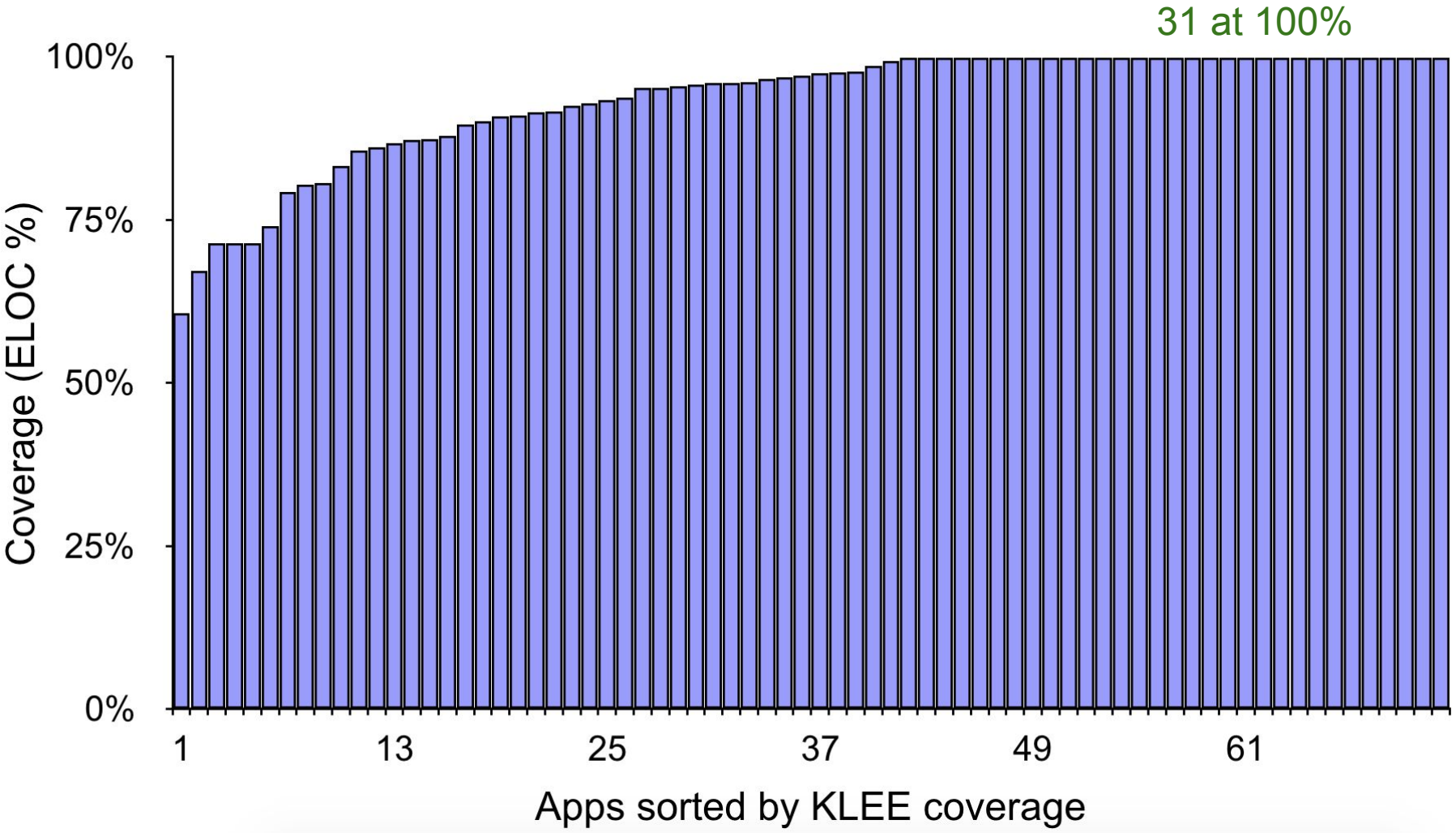
- Fully automatic runs
- Run KLEE for one hour on each program
- Run resulting test cases on *uninstrumented* program
- Measure line coverage using gcov
  - Eliminates unintended KLEE bias

Coreutils Coverage

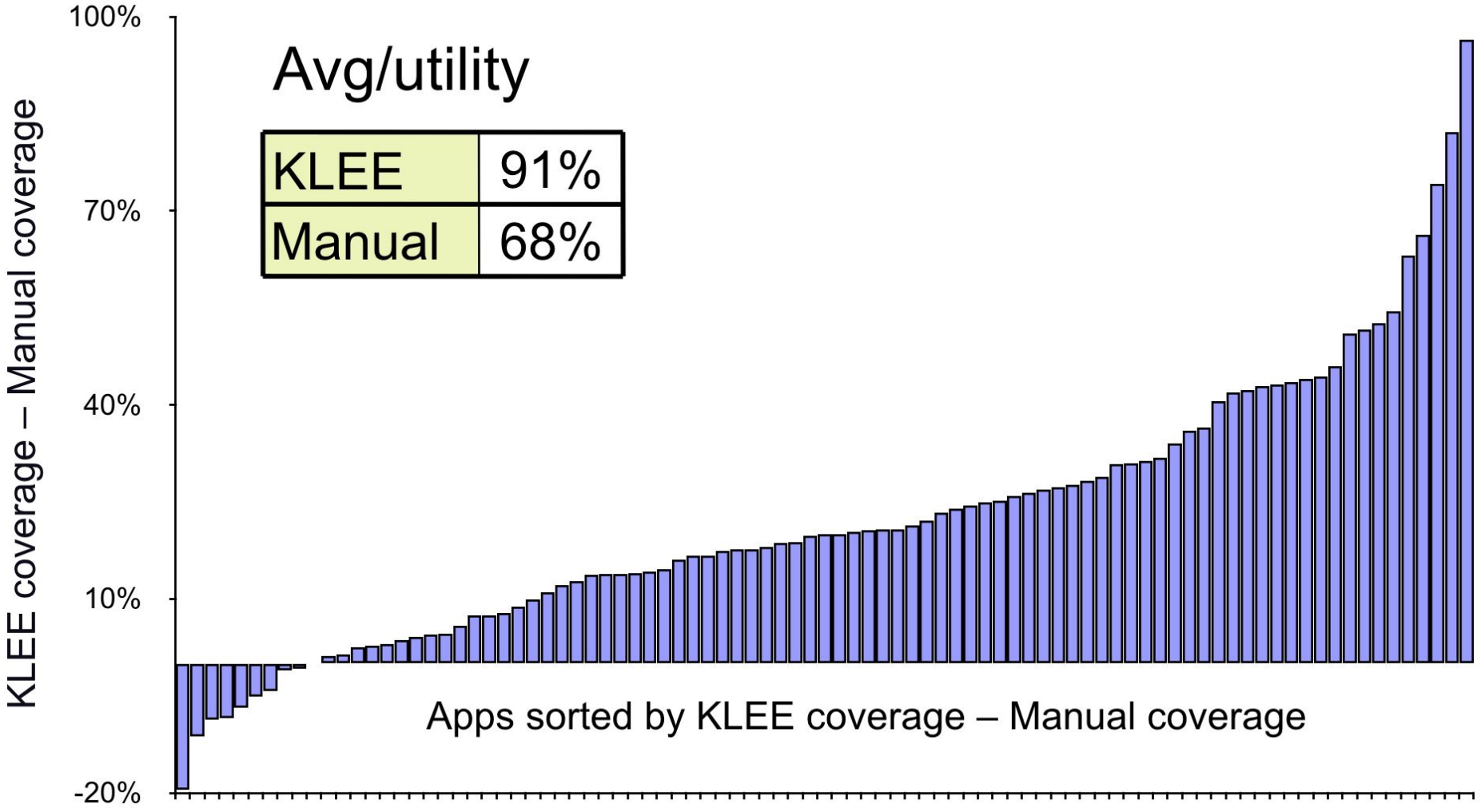


BusyBox Coverage

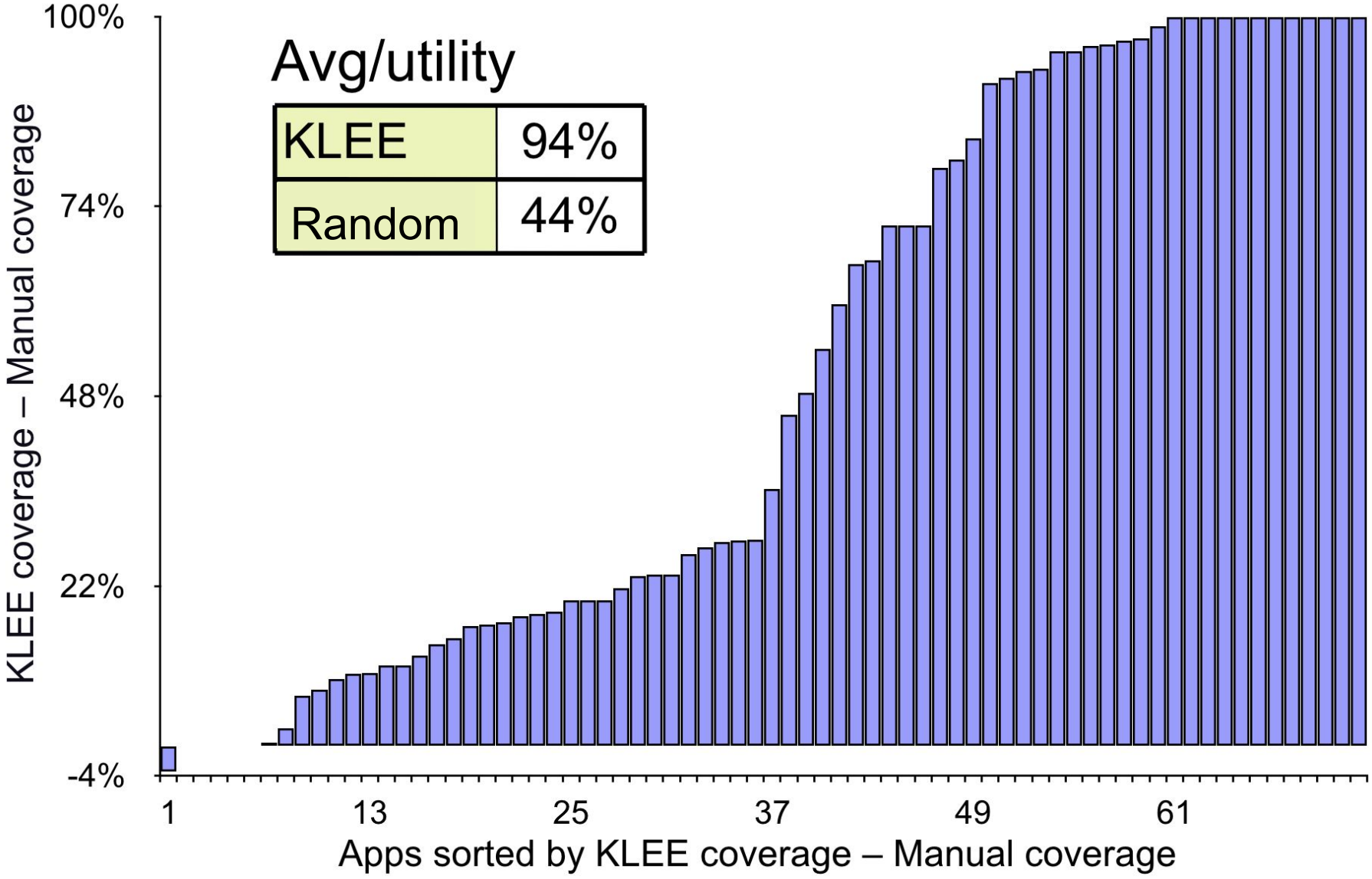
Overall: 91%, Average: 94%, Median: 98%



KLEE vs. Manual Testing



KLEE vs. Random Testing



Correctness Bugs

“One way to look at KLEE is that it automatically translates a path through a C program into a form that a theorem prover can reason about.”

```
func() { … }
…
if ( func() matches spec. )
{
      …
      assert_true:
      …
}
```

```
func() { … }
…
assert(func() matches spec.)
```

Crosschecking

If f(x) and g(x) implement the same interface:

1. Make input x symbolic
2. Run KLEE on assert(f(x) == g(x))
3. If KLEE terminates without errors, then f(x) are g(x) semantically equivalent

Mismatches found between coreutils and busybox.

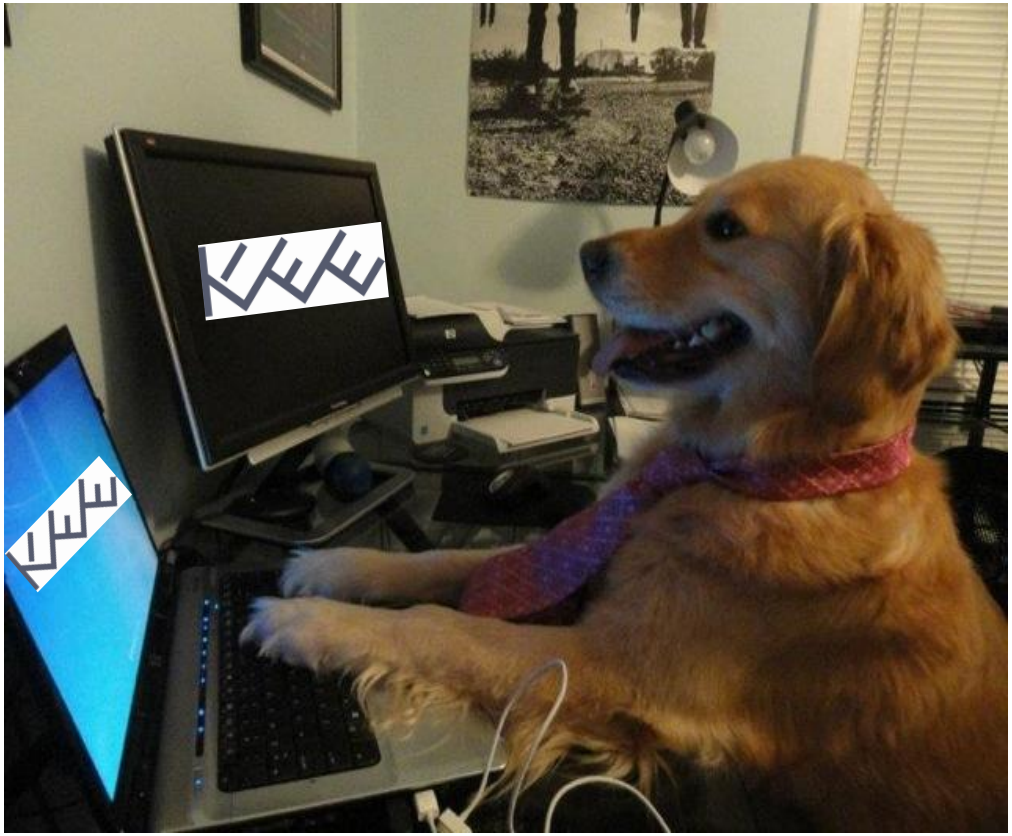
```
int main() {
    unsigned x,y;
    make_symbolic(&x, sizeof(x));
    make_symbolic(&y, sizeof(y));
    assert(mod(x,y) == mod_opt(x,y));
    return 0;
}
```

Limitations

- Still can’t solve the halting problem
- Finding bugs is completely dependent on the precision of models
- S L O W
- C-specific
- Limited by theorem prover

> “The functions in STP’s input language include concatenation, extraction, left/right shift, sign-extension, unary minus, addition, multiplication, (signed) modulo/division, bitwise Boolean operations, if-then-else terms, and array reads and writes. The predicates in the language include equality and (signed) comparators between bitvector terms.”

Questions?



Demo