

# KLEE: Effective Testing of Systems Programs

**Cristian Cadar**

Joint work with Daniel Dunbar and Dawson Engler



STANFORD  
UNIVERSITY

---

# Writing Systems Code Is Hard

---

- Code complexity
  - Tricky control flow
  - Complex dependencies
  - Abusive use of pointer operations
- Environmental dependencies
  - Code has to anticipate all possible interactions
  - Including malicious ones

# KLEE

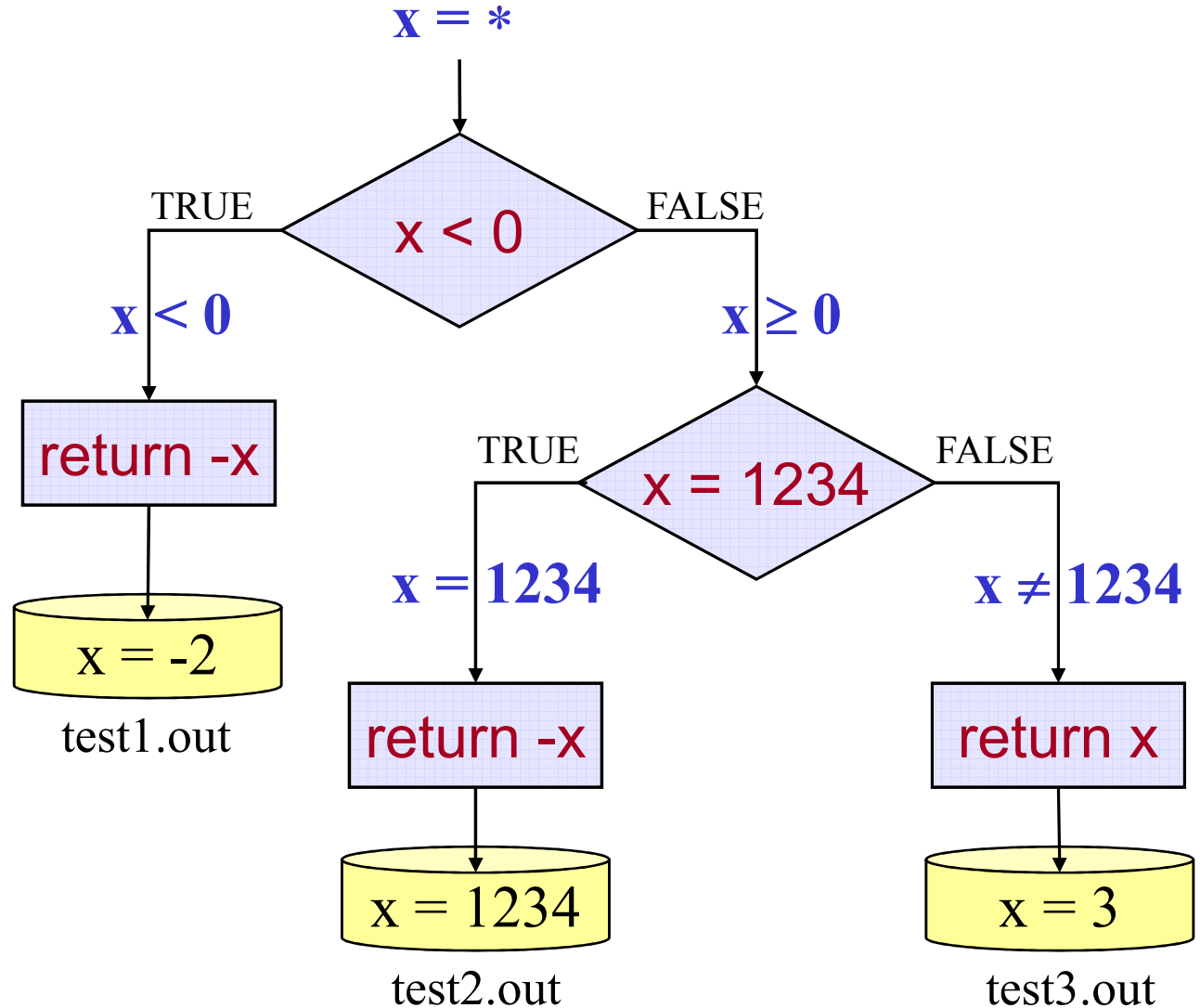
[OSDI 2008, Best Paper Award]

---

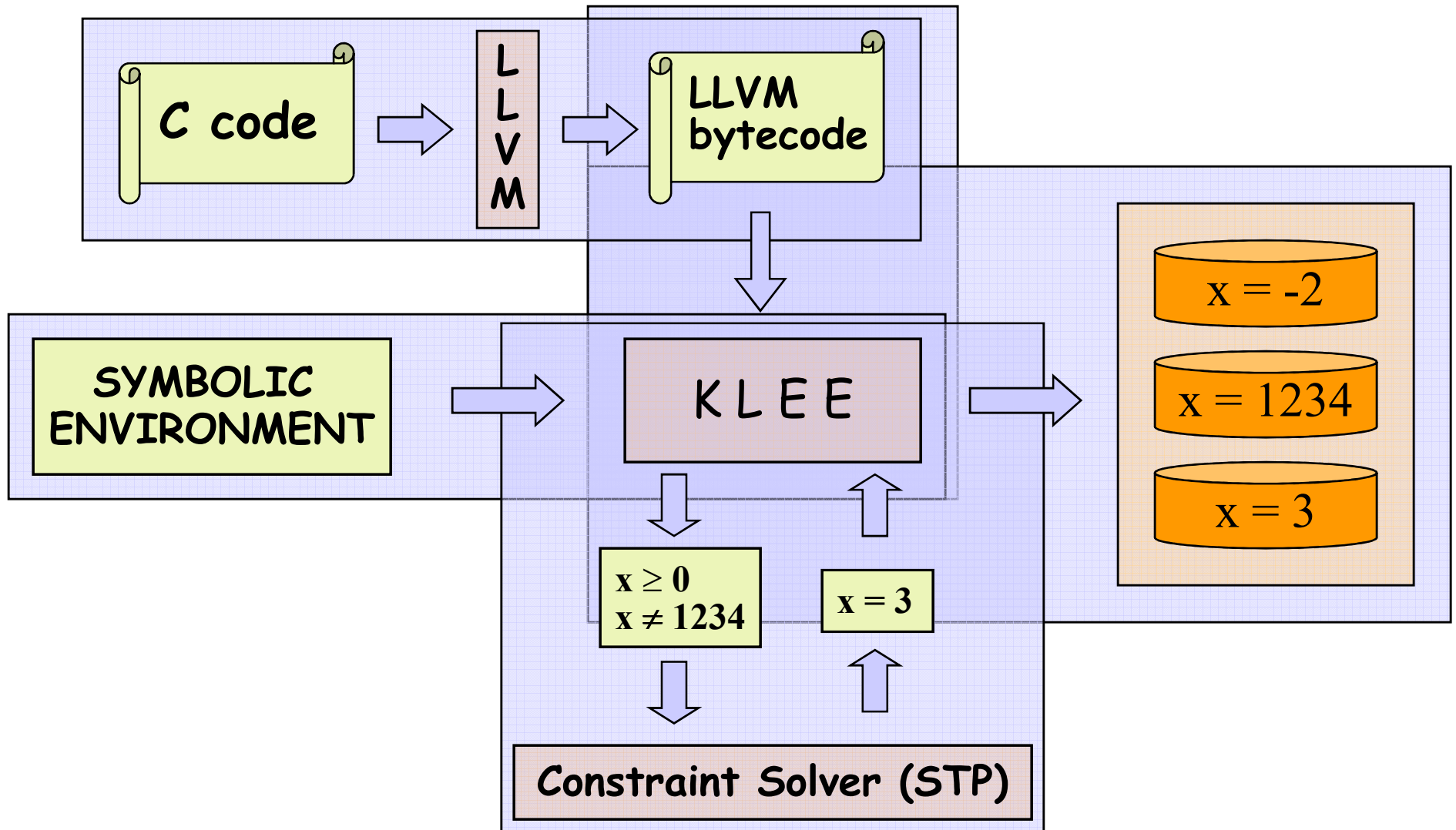
- Based on symbolic execution and constraint solving techniques
- 
- Automatically generates high coverage test suites
    - Over 90% on average on ~160 user-level apps
  - Finds deep bugs in complex systems programs
    - Including higher-level correctness ones

# Toy Example

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



# KLEE Architecture



# Outline

---

- Motivation
- Example and Basic Architecture
- • Scalability Challenges
- Experimental Evaluation

# Three Big Challenges

---

- Motivation
- Example and Basic Architecture
- ➔ • Scalability Challenges
  - Exponential number of paths
  - Expensive constraint solving
  - Interaction with environment
- Experimental Evaluation

# Exponential Search Space

---

Naïve exploration can easily get “stuck”

Use search heuristics:

- **Coverage-optimized search**
  - Select path closest to an uncovered instruction
  - Favor paths that recently hit new code
- **Random path search**
  - See [KLEE – OSDI’08]



# Three Big Challenges

---

- Motivation
- Example and Basic Architecture
- Scalability Challenges
  - Exponential number of paths
  - ➔ – Expensive constraint solving
  - Interaction with environment
- Experimental Evaluation

# Constraint Solving

---

- Dominates runtime
  - Inherently expensive (NP-complete)
  - Invoked at every branch
- Two simple and effective optimizations
  - Eliminating irrelevant constraints
  - Caching solutions
    - Dramatic speedup on our benchmarks

# Eliminating Irrelevant Constraints

---

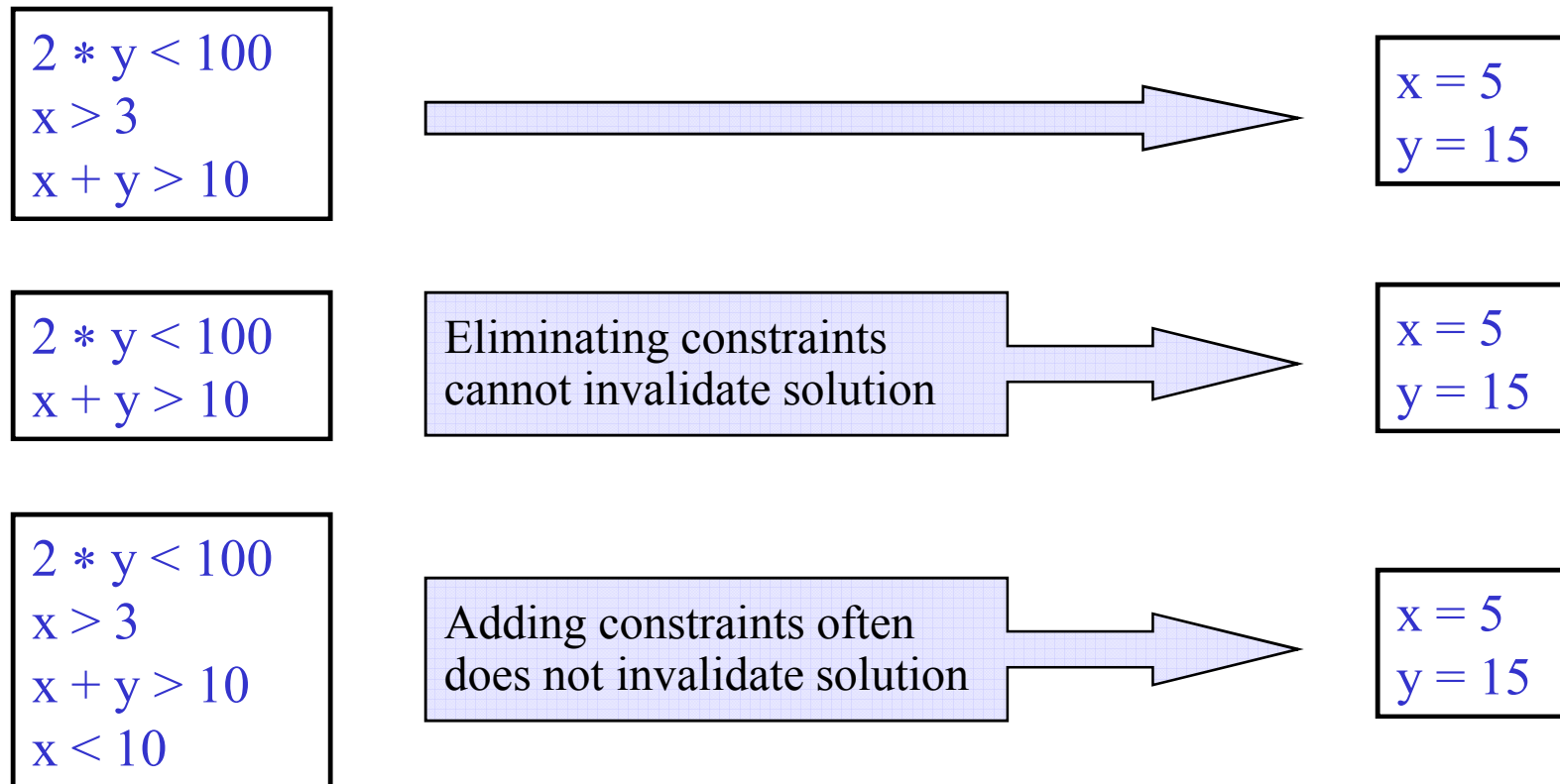
- In practice, each branch usually depends on a small number of variables

...		$x + y > 10$
...		<del><math>z \&amp; -z = z</math></del>
if (x < 10) {	→	$x < 10 ?$
...		
}		

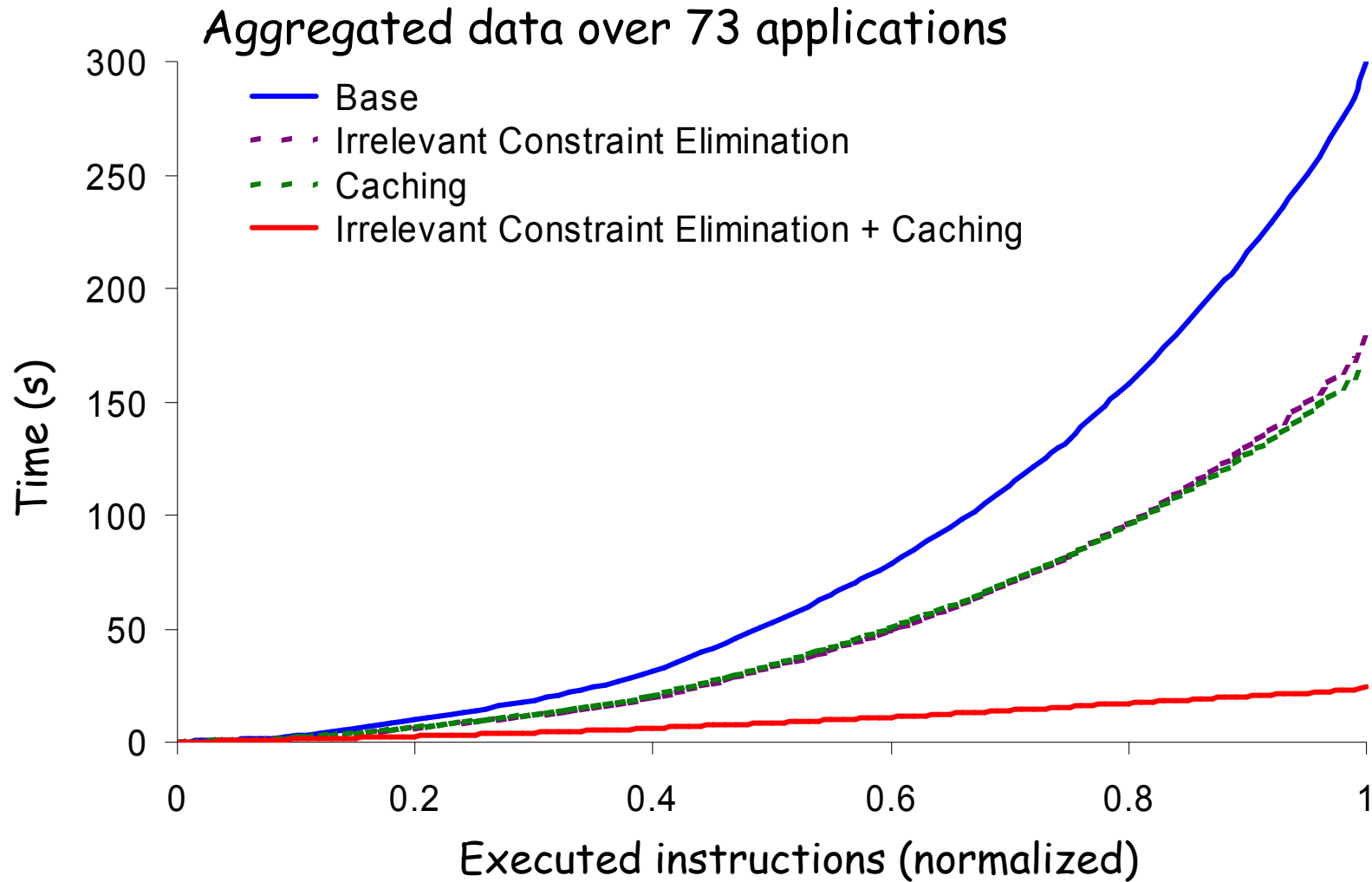
# Caching Solutions

---

- Static set of branches: lots of similar constraint sets



# Dramatic Speedup



# Three Big Challenges

---

- Motivation
- Example and Basic Architecture
- **Scalability Challenges**
  - Exponential number of paths
  - Expensive constraint solving
  - ➔ – Interaction with environment
- **Experimental Evaluation**

# Environment: Calling Out Into OS

---

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


- If all arguments are concrete, forward to OS

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

- Otherwise, provide *models* that can handle symbolic files
  - Goal is to explore all possible *legal* interactions with the environment

# Environmental Modeling

---

```
// actual implementation: ~50 LOC
ssize_t read(int fd, void *buf, size_t count) {
    exe_file_t *f = get_file(fd);
    ...
     memcpy(buf, f->contents + f->off, count)
    f->off += count;
    ...
}
```

- Plain C code run by KLEE
  - Users can extend/replace environment w/o any knowledge of KLEE internals
- Currently: effective support for symbolic command line arguments, files, links, pipes, ttys, environment vars



# Does KLEE work?

---

- Motivation
- Example and Basic Architecture
- Scalability Challenges
- ➔ • **Evaluation**
  - Coverage results
  - Bug finding
  - Crosschecking

# GNU Coreutils Suite

---

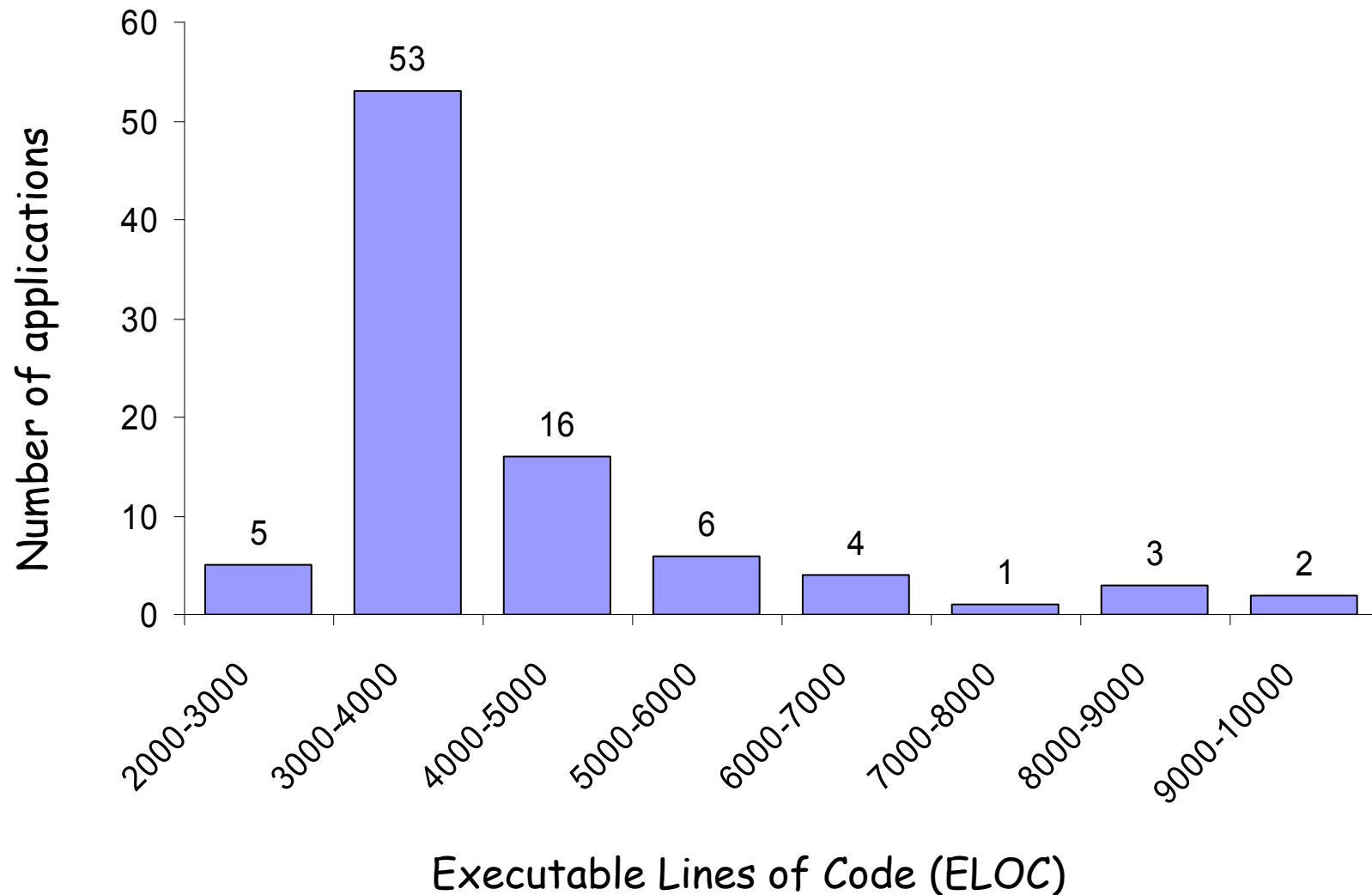
- Core user-level apps installed on many UNIX systems
- 89 stand-alone (i.e. excluding wrappers) apps (v6.10)
  - File system management: `ls`, `mkdir`, `chmod`, etc.
  - Management of system properties: `hostname`, `printenv`, etc.
  - Text file processing : `sort`, `wc`, `od`, etc.
  - ...

**Variety of functions, different authors,  
intensive interaction with environment**

**Heavily tested, mature code**

# Coreutils ELOC (incl. called lib)

---



# Methodology

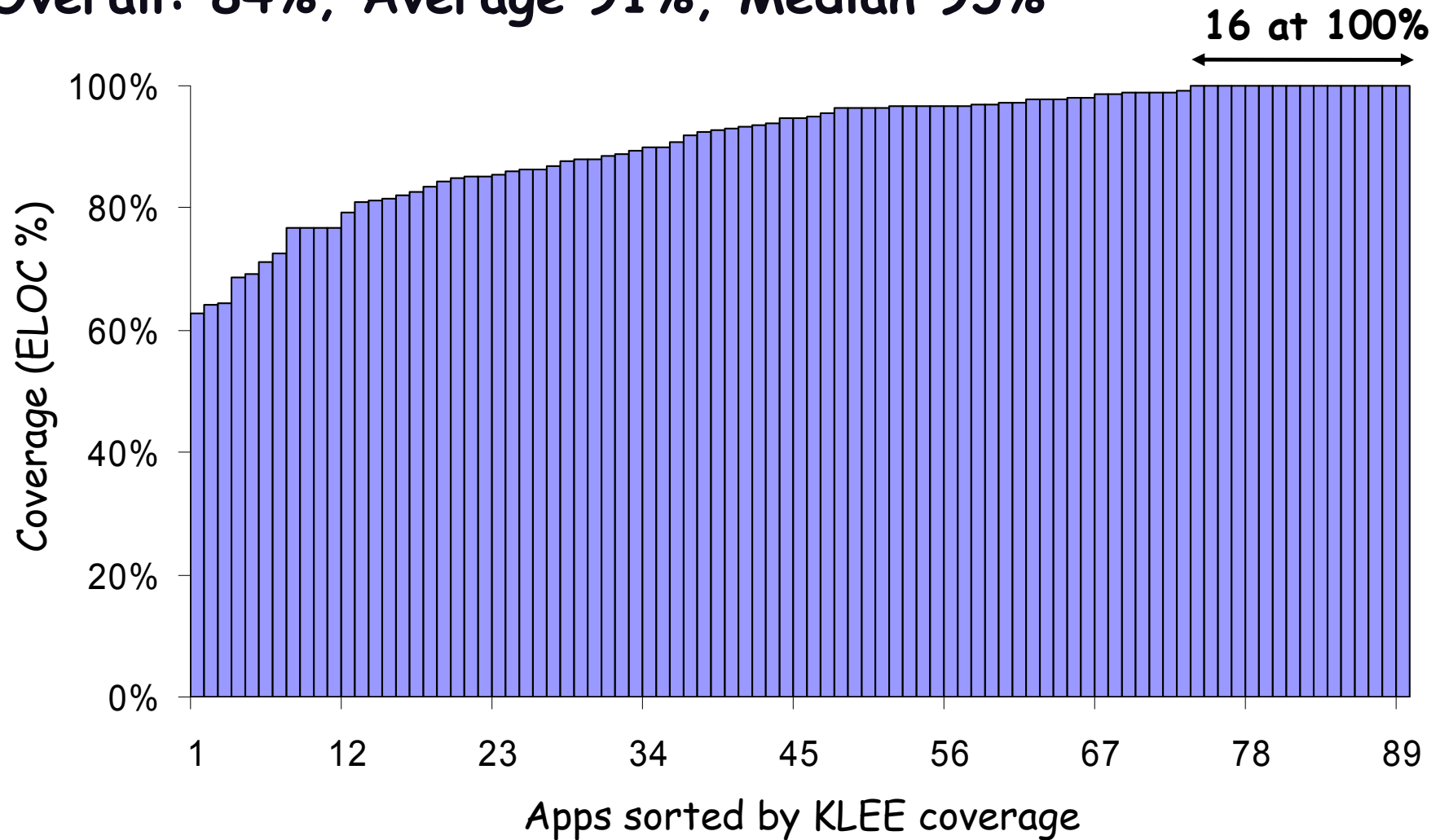
---

- Fully automatic runs
- Run KLEE one hour per utility, generate test cases
- Run test cases on *uninstrumented* version of utility
- Measure line coverage using *gcov*
  - Coverage measurements not inflated by potential bugs in our tool

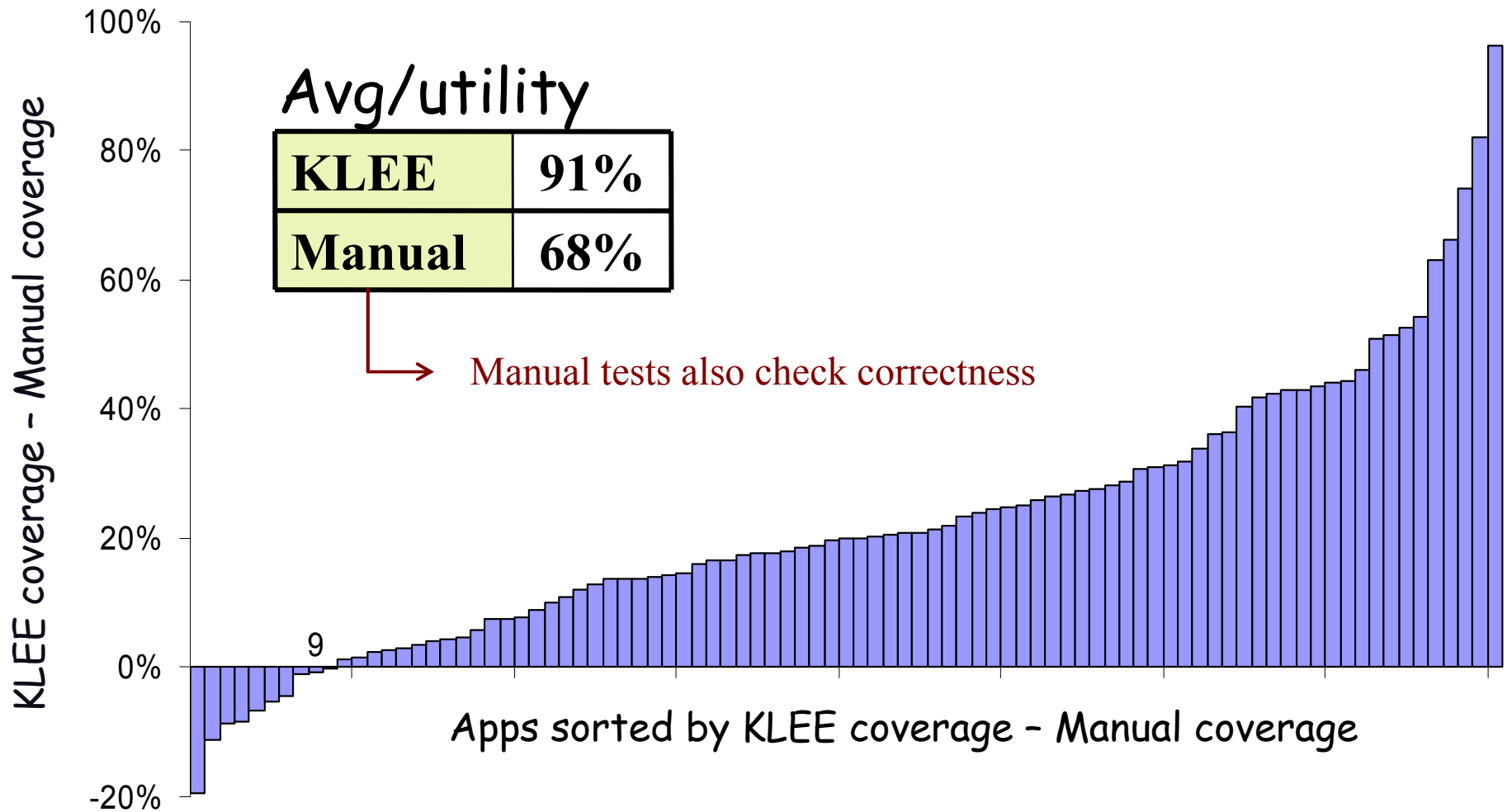
# High Line Coverage

(Coreutils, non-lib, 1h/utility = 89 h)

Overall: 84%, Average 91%, Median 95%

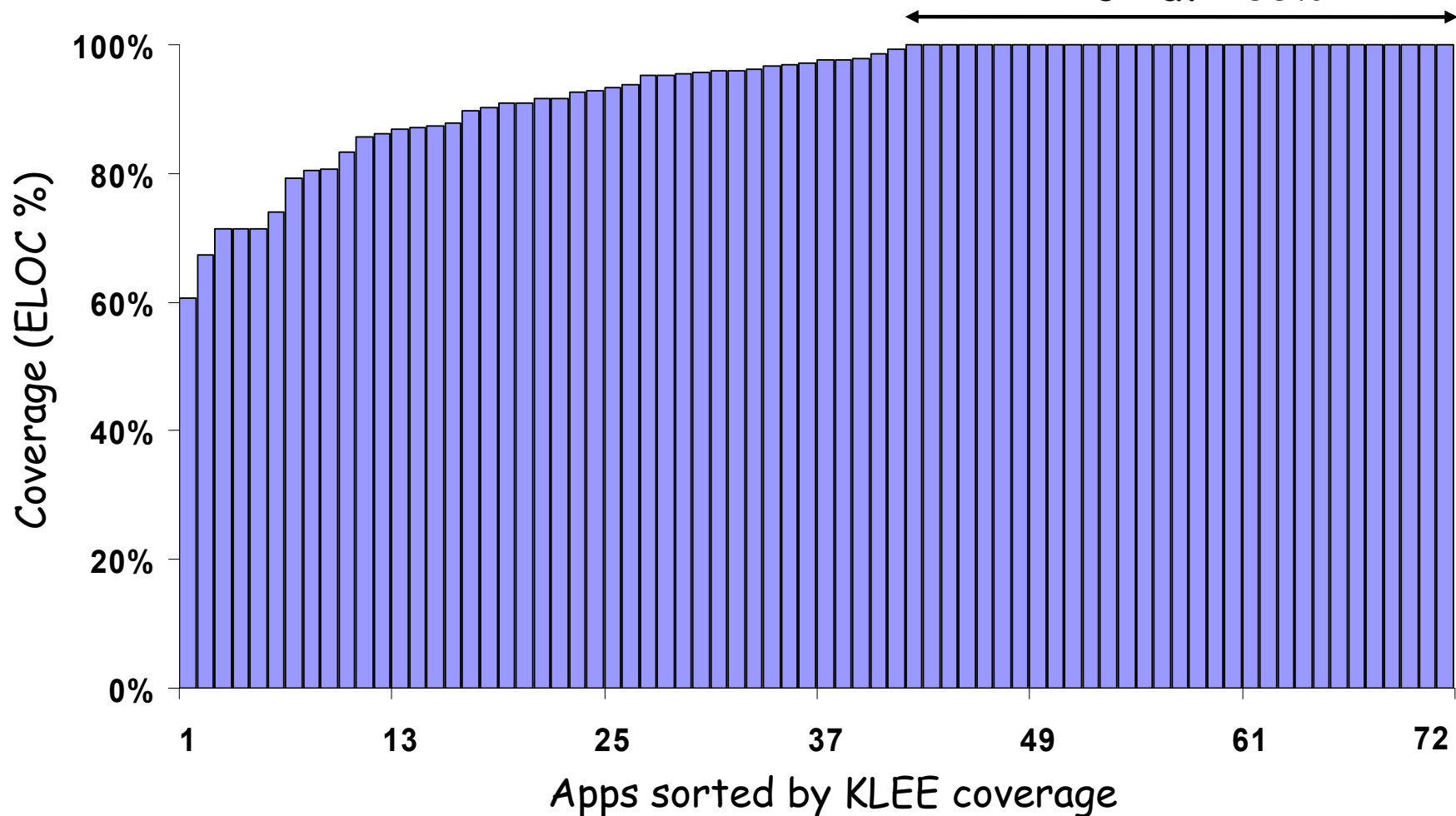


# Beats 15 Years of Manual Testing

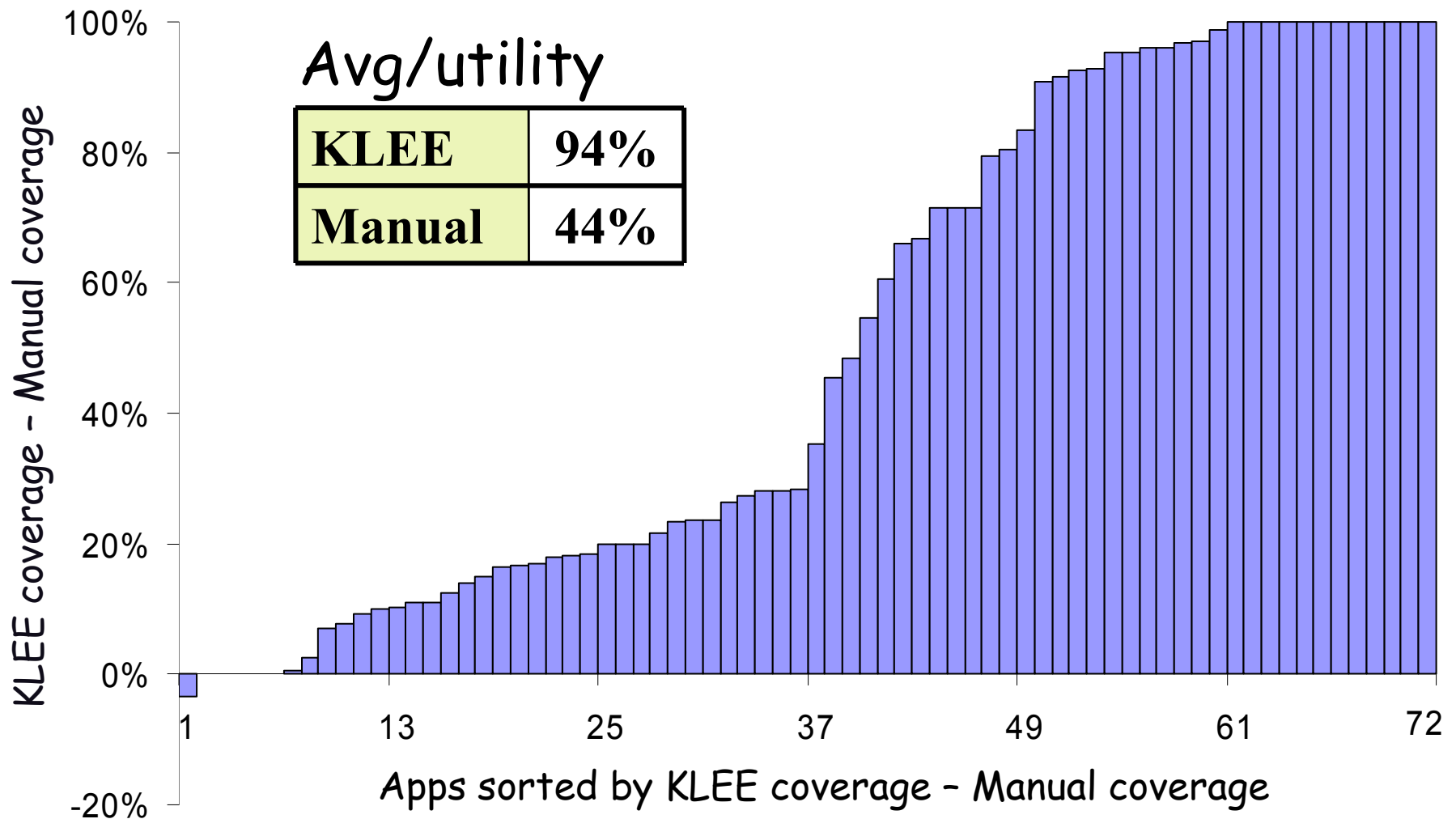


# Busybox Suite for Embedded Devices

Overall: 91%, Average 94%, Median 98% 31 at 100%



# Busybox – KLEE vs. Manual





# Does KLEE work?

---

- Motivation
- Example and Basic Architecture
- Scalability Challenges
- **Evaluation**
  - Coverage results
  - – **Bug finding**
  - **Crosschecking**

# GNU Coreutils Bugs

---

- Ten crash bugs
  - More crash bugs than approx last three years combined
  - KLEE generates actual command lines exposing crashes

# Ten command lines of death

---

<pre>md5sum -c t1.txt mkdir -Z a b mkfifo -Z a b mknod -Z a b p seq -f %0 1</pre>	<pre>pr -e t2.txt tac -r t3.txt t3.txt paste -d\ abcdefghijklmnopqrstuvwxyz ptx -F\ abcdefghijklmnopqrstuvwxyz ptx x t4.txt</pre>
	<pre>t1.txt:  t  tMD5( t2.txt:  b b b b b b t t3.txt:  n t4.txt: A</pre>

# Does KLEE work?

---

- Motivation
- Example and Basic Architecture
- Scalability Challenges
- **Evaluation**
  - Coverage results
  - Bug finding
  - **– Crosschecking**

# Finding Correctness Bugs

---

- KLEE can prove asserts on a per path basis
  - Constraints have no approximations
  - An assert is just a branch, and KLEE proves feasibility/infeasibility of each branch it reaches
  - If KLEE determines infeasibility of false side of assert, the assert was proven on the current path

# Crosschecking

---

Assume  $f(x)$  and  $f'(x)$  implement the same interface

1. Make input  $x$  symbolic
2. Run KLEE on `assert(f(x) == f'(x))`
3. For each explored path:
  - a) KLEE terminates w/o error: paths are equivalent
  - b) KLEE terminates w/ error: mismatch found

Coreutils vs. Busybox:

1. UNIX utilities should conform to *IEEE Std.1003.1*
2. Crosschecked pairs of Coreutils and Busybox apps
3. Verified paths, found mismatches

# Mismatches Found

Input	Busybox	Coreutils
<code>tee "" &lt;t1.txt</code>	[infinite loop]	[terminates]
<code>tee -</code>	[copies once to stdout]	[copies twice]
<code>comm t1.txt t2.txt</code>	[doesn't show diff]	[shows diff]
<code>cksum /</code>	"4294967295 0 /"	"/: Is a directory"
<code>split /</code>	"/: Is a directory"	
<code>tr</code>	[duplicates input]	"missing operand"
<code>[ 0 "&lt;" 1 ]</code>		"binary op. expected"
<code>tail -2l</code>	[rejects]	[accepts]
<code>unexpand -f</code>	[accepts]	[rejects]
<code>split -</code>	[rejects]	[accepts]
t1.txt: a t2.txt: b (no newlines!)		

# Related Work

---

Very active area of research. E.g.:

- EGT / EXE / KLEE [Stanford]
- DART [Bell Labs]
- CUTE [UIUC]
- SAGE, Pex [MSR Redmond]
- Vigilante [MSR Cambridge]
- BitScope [Berkeley/CMU]
- CatchConv [Berkeley]
- JPF [NASA Ames]

## KLEE

- Hundred distinct benchmarks
- Extensive coverage numbers
- Symbolic crosschecking
- Environment support



# KLEE

## Effective Testing of Systems Programs

---

- KLEE can effectively:
  - Generate high coverage test suites
    - Over 90% on average on ~160 user-level applications
  - Find deep bugs in complex software
    - Including higher-level correctness bugs, via crosschecking