Towards making it a surprise when embedded code breaks

Dawson Engler
(Anthony Romano & David Ramos & others)
Stanford

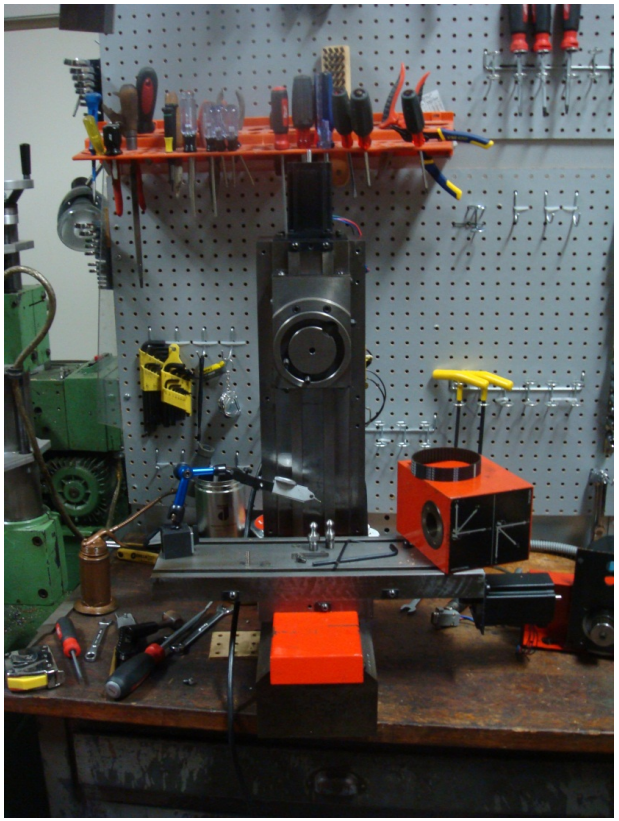
Secure Internet of Things Project Workshop
Stanford University
August 11, 2014

Dawson Engler

- Academic Lineage
  - MIT PhD (exokernel operating system)
  - Stanford Associate professor CS/EE
  - Co-founded Coverity
  - 6th Weiser award, Grace Hopper Award 2008
- Focus: finding bad bugs in real code.
  - Main religion = results.
  - Static bug finding
  - Model checking
  - Symbolic execution (Current)

Also: A bit of embedded moonlighting

- Pulled many allnighters guiltily rebuilding CNC machines
- Will be great to combine embedded and research.



Context

- A short history of 0 and 1:
  - Everything is software (even hardware)
  - All software is broken.
  - Everything is broken.
  - Gives many stories a tedious narrative
- Some intensifiers for embedded:
  - Bugs generally costly. And monetized.
  - Effects of bad people controlling of real things can be unpleasant.
  - Environment ugly: concurrency, weird space hacks, weirdo devices
  - Checking bare metal code is hard. In practice often “can’t.”

Current research = reaction to disappointment

- Used to do a lot of static bug finding
  - Easy. Worked.
  - Push button = 1000s of bugs. Kernels, embedded, whatever.
  - Check a large system? Always find lots of bugs.
    - Otherwise something must be misconfigured
    - Reasonably successful tools company.

- But, static bug-finding not going to solve the problem.
  - Bugs visible in code? Great.
  - Bugs implied by code? Not so great.

  - What we want: check code so deeply, you are surprised if it crashes.
  - Really need to be able to run code. Hard.

How to run code?

- Manual Testing
  - Historical record not encouraging.
  - Enormous undertaking to hit all corner cases
- Random Testing
  - Cheap, but gets stuck easily.
- Trick: use code to construct its own inputs!
  - Run on constraints rather than constants
  - Fork execution at if-statements
  - Solve constraints, get input to rerun same path

Klee

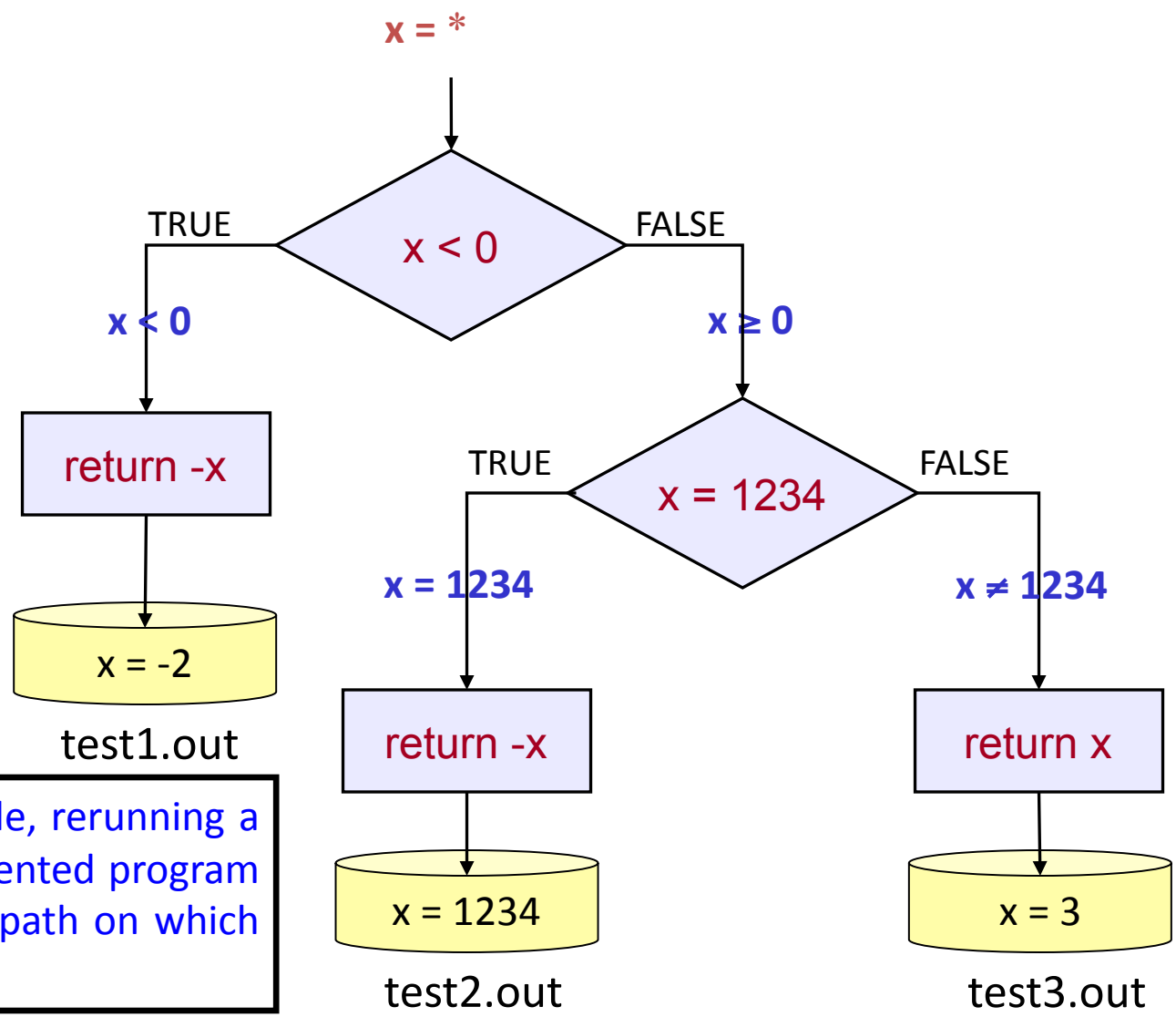
- Based on symbolic execution and constraint solving.
  - Constraints accurate down to the level of a single bit.
- Some results.
  - Automatically execute almost all statements in real code.
  - Generate inputs of death that crash running servers
  - In many cases makes it easier to (finitely) verify code / patch correctness than it is to write a single test case.

  - Key for embedded: developing techniques to jump to arbitrary code and run it. In practice often not possible to check otherwise.

- Long term vision
  - Want: tool that checks code so deeply and thoroughly that if you fix all bugs you are surprised if crashes.
  - How: Run code on all interesting paths on all possible values.

Toy Example

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

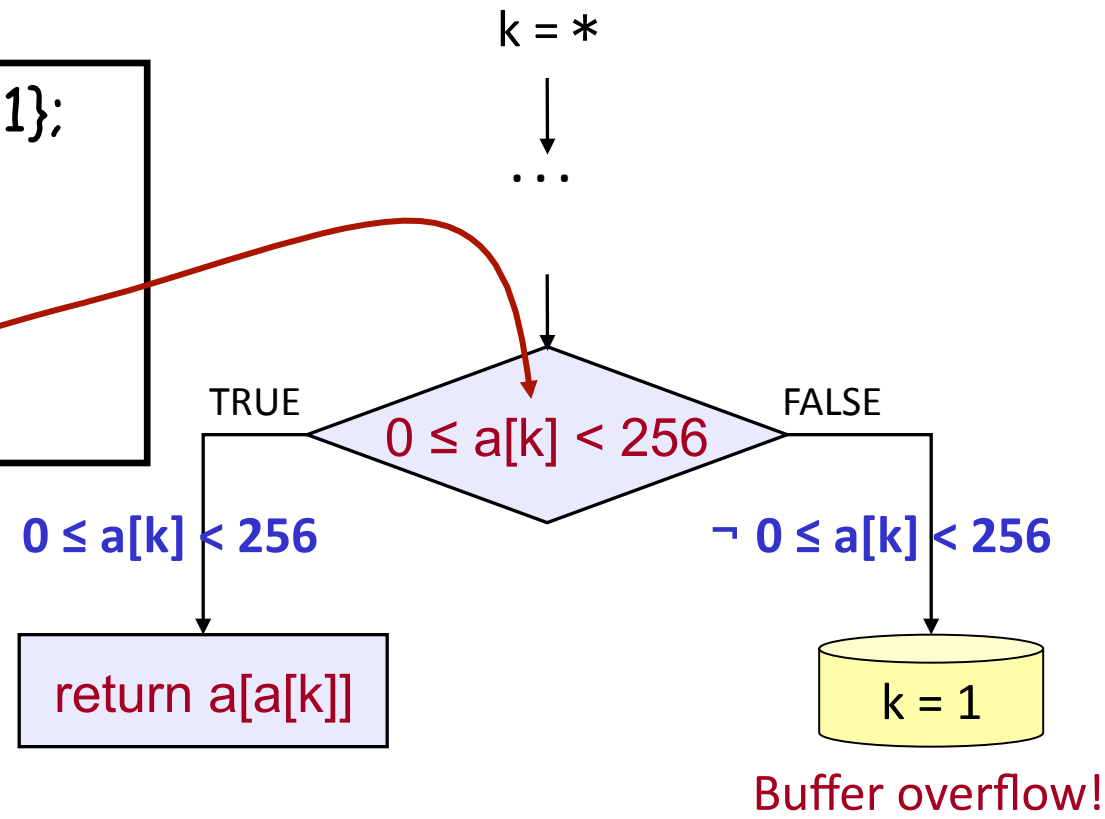


Assuming deterministic code, rerunning a test case on the uninstrumented program will follow <u>the exact same</u> path on which it was generated.

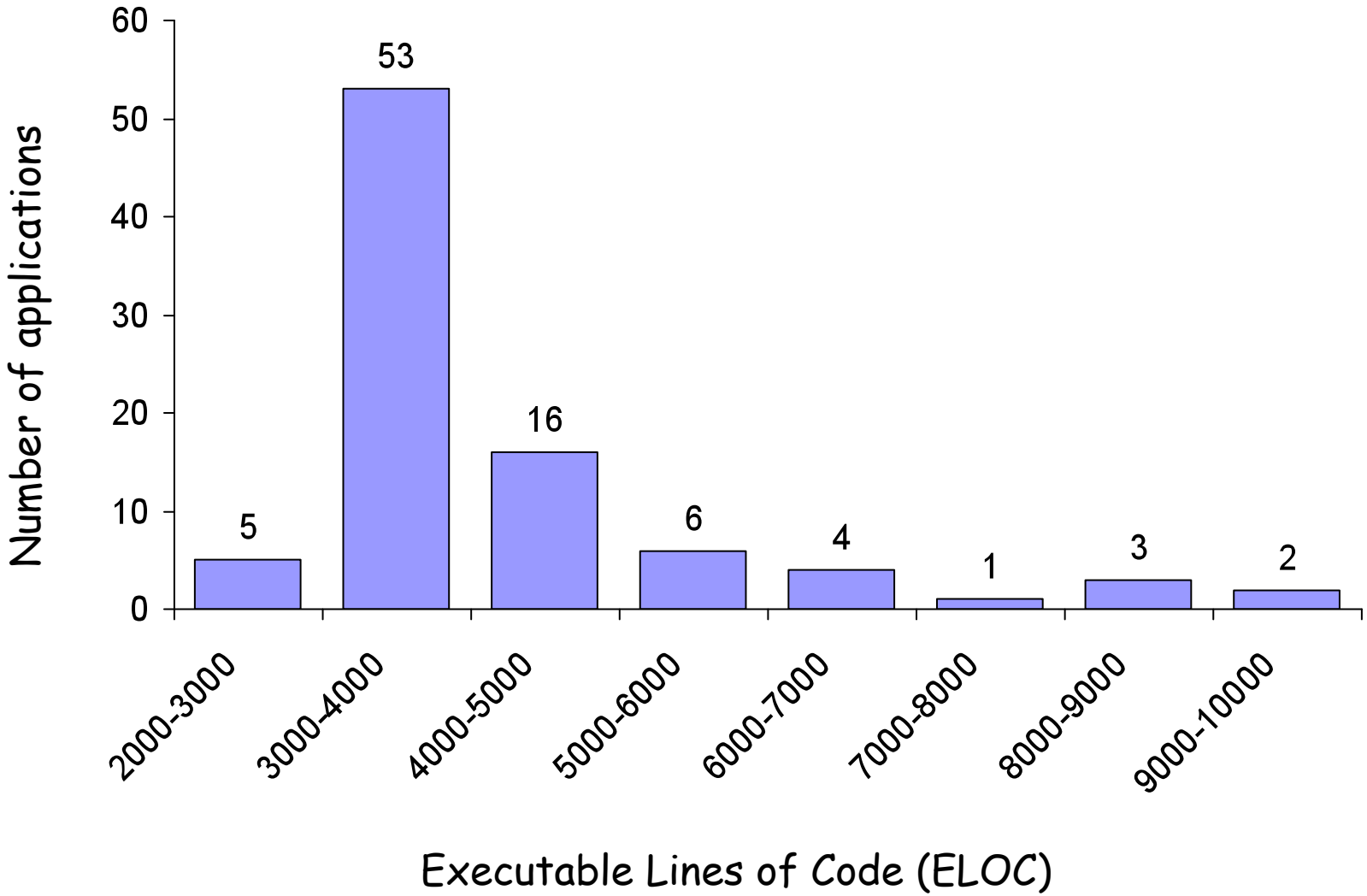
Universal Checks

- KLEE will find the bug if any buggy values exists on that path!
- Unlike traditional dynamic execution tools, which can reason only about a single set of values at a time

```
int a[256] = {1, -1, 1, 1, ..., 1};
...
int foo(unsigned char k) {
    return a[a[k]];
}
```



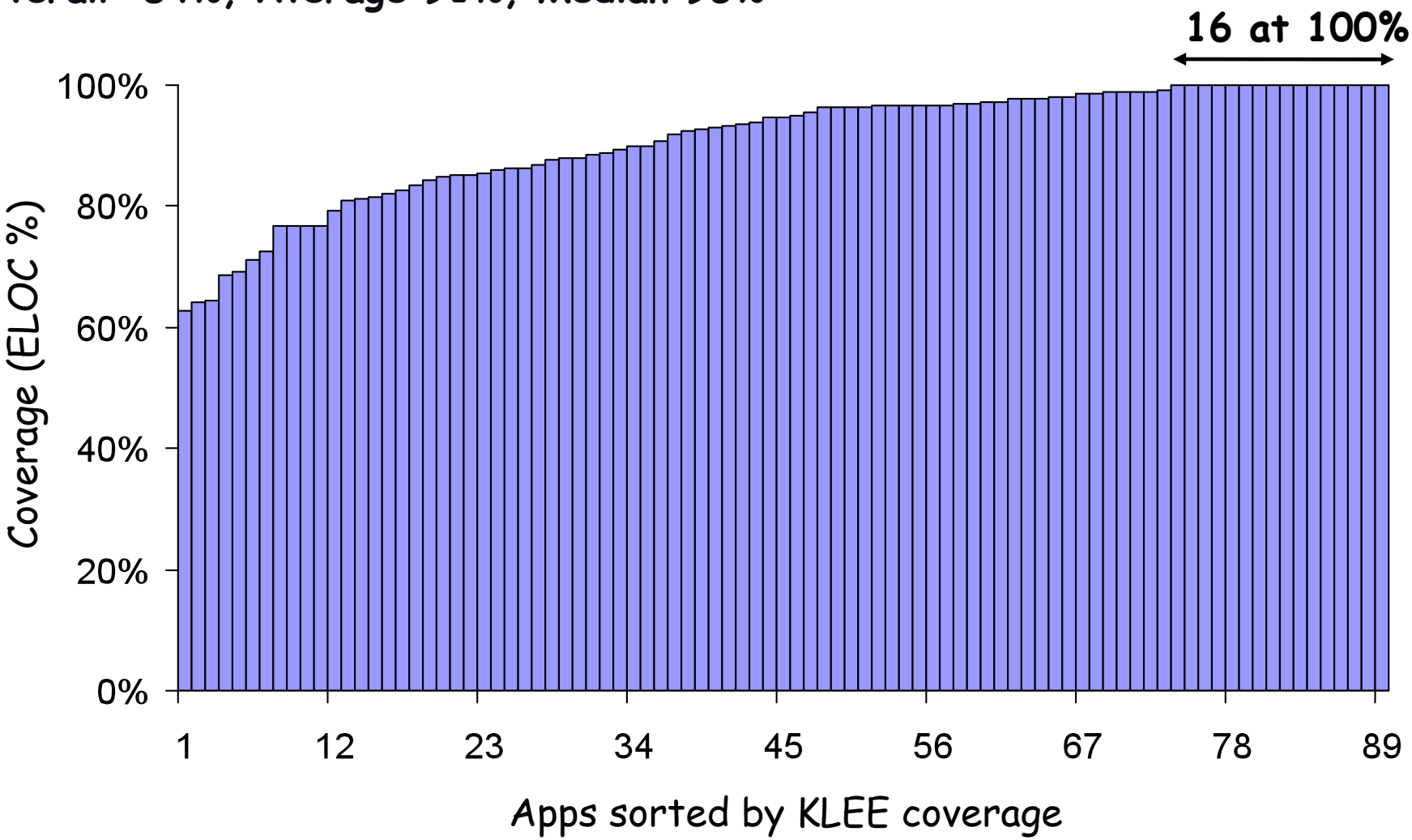
Coreutils ELOC (incl. called lib)



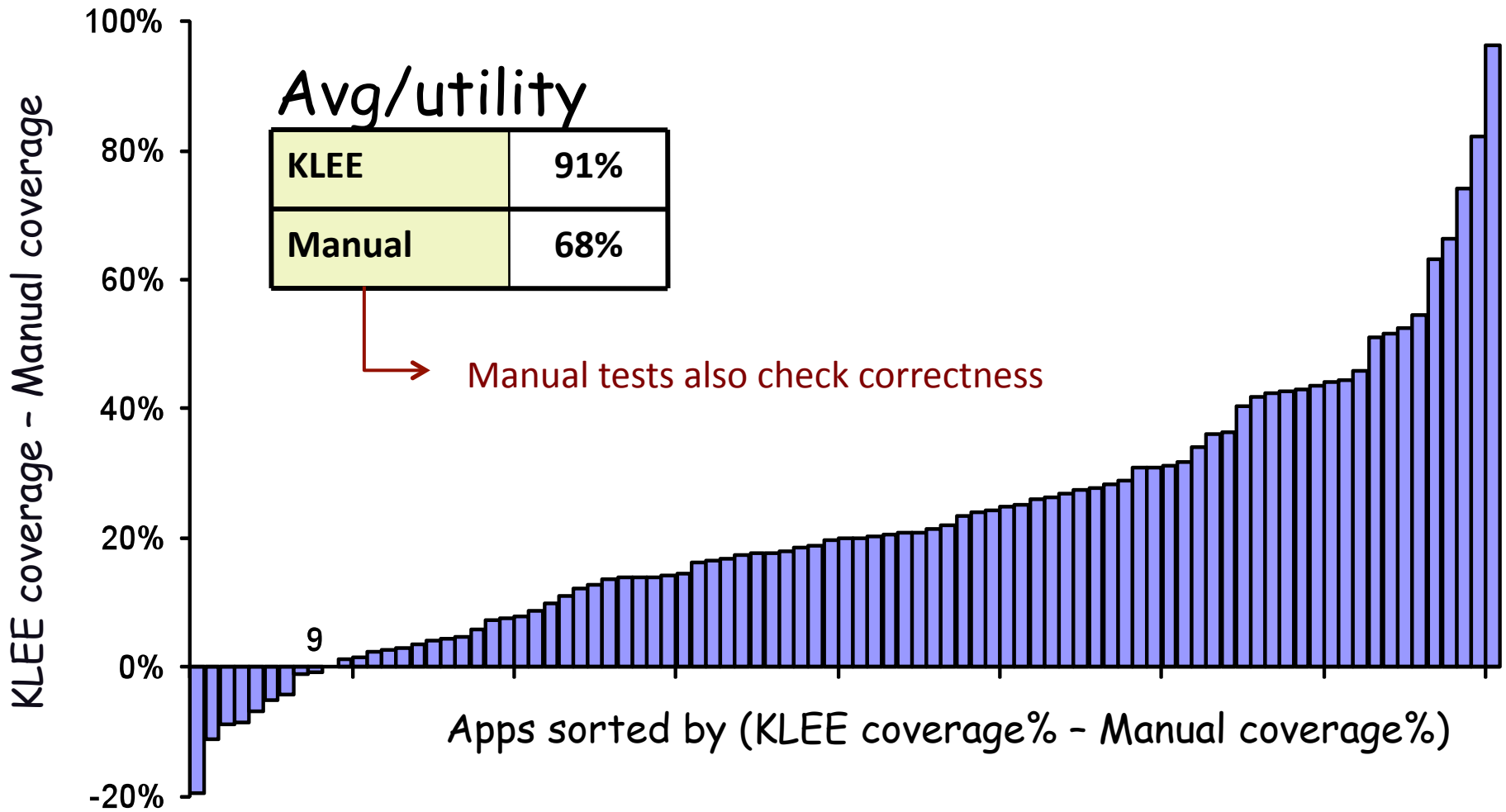
High Line Coverage

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

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



Beats 15 Years of Manual Testing

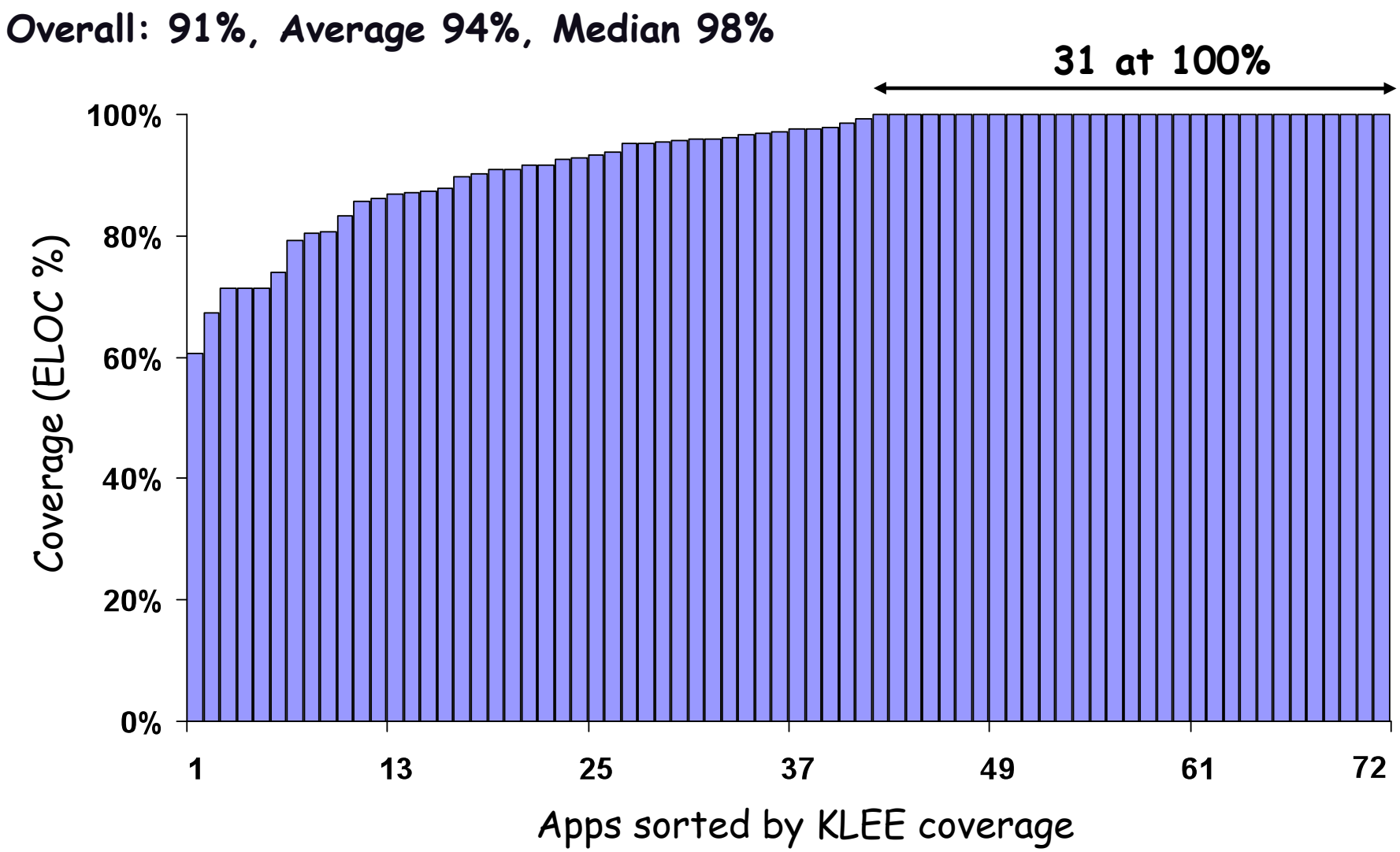


Avg/utility

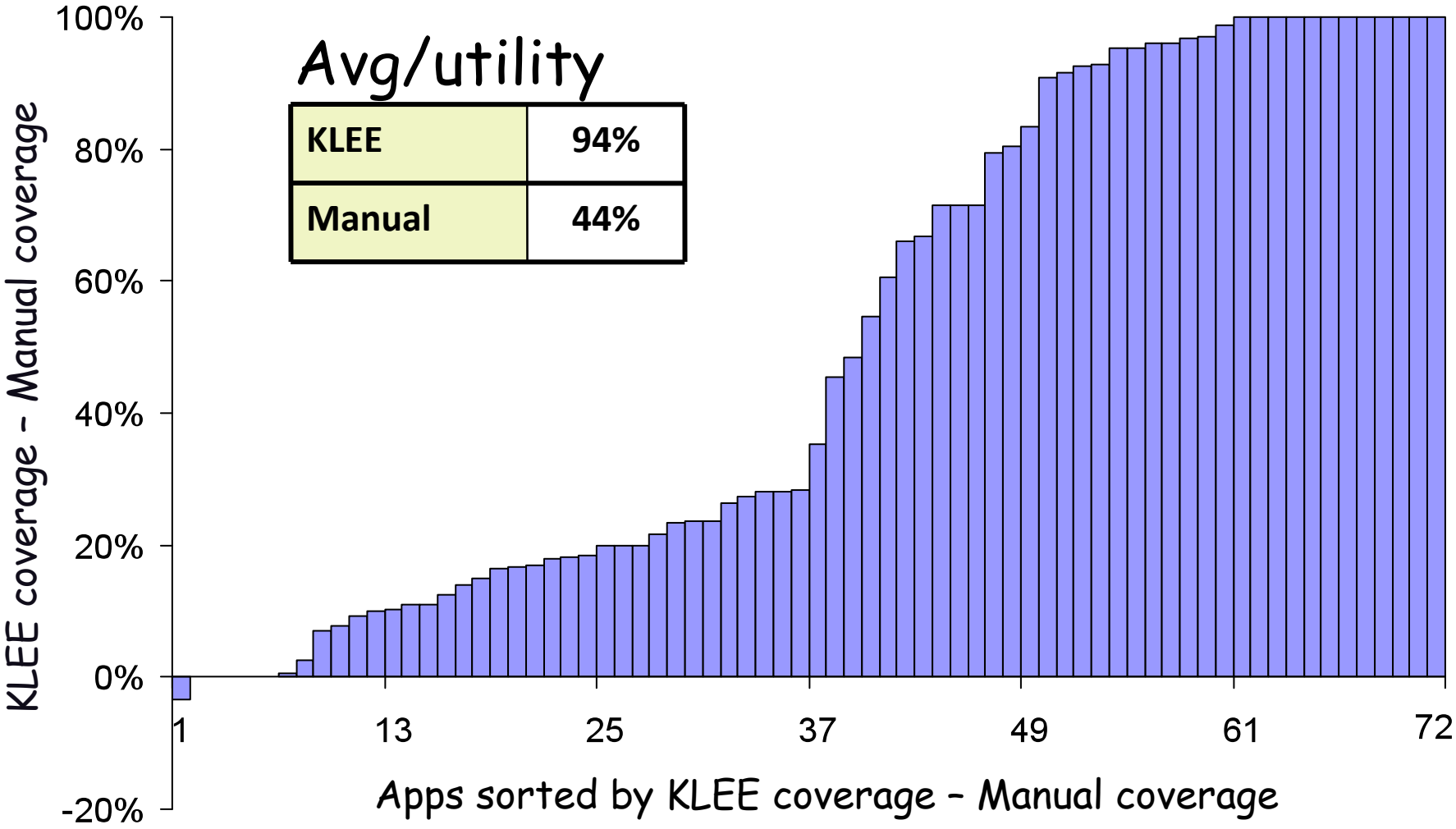
KLEE	91%
Manual	68%

Manual tests also check correctness

Busybox Suite for Embedded Devices



Busybox – Klee vs. Manual



Find bug? Spit out command line! Ten command lines of death

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

t1.txt: `\t \tMD5(`
t2.txt: `\b\b\b\b\b\b\b\t`
t3.txt: `\n`
t4.txt: **A**

- More crash bugs than last three years combined
- All fixed promptly. Test case added to official suite.

Klee with a machine-code frontend

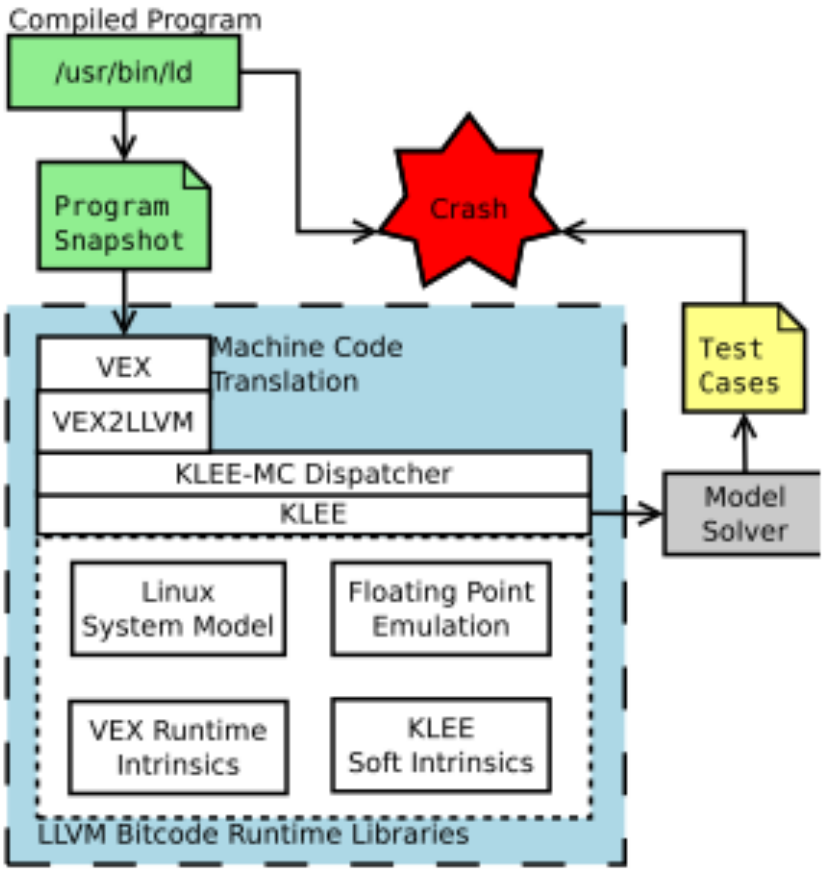
KLEE-MC extends KLEE to support program binaries (a lot of programs).

Symbolically Executes Binary Programs

- LIBVEX machine code front-end
- System calls return symbolic data
- Test cases as system call traces

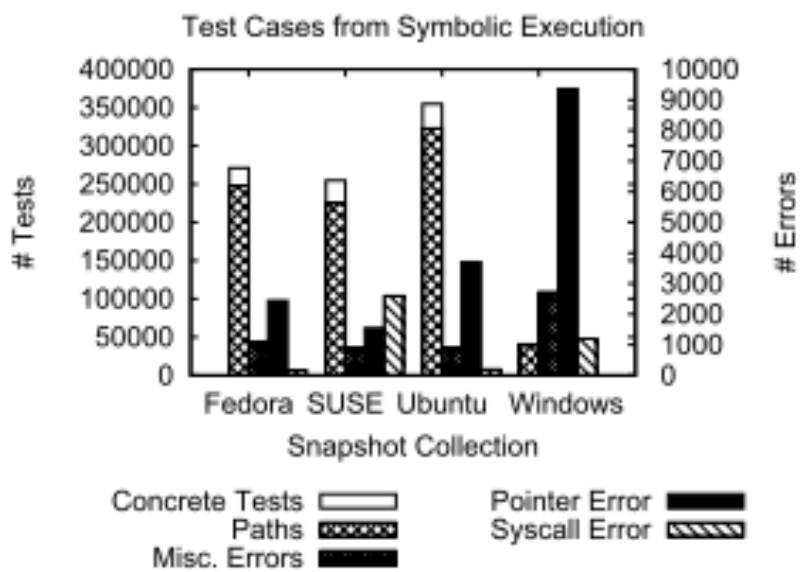
Architectures and platforms

- amd64/linux
- i386/linux
- ARM/linux, android-linux
- i386/windows



Checking tens of thousands of binary programs

Collection	Packages	Programs	Unique
Fedora64	7800	11830	4124
OpenSUSE32	5206	7863	1946
Ubuntu64	12775	15375	9586
Windows32	19312	1905	1579
Total	45093	**36973**	25728



ARM	CM v10.1.2	RPI v1.3
Total Binaries	147	3026
Tests	8148	125398
Flagged Tests	125	903
Bad Memory Accesses	84	462

Wide-scale symbolic execution

- Search for Program Faults
- Five minutes per program
- Hardware-validated tests

Selected faulting inputs:

Program	Arguments
cclambda	`--ran .`
dc	`--e=[`
h5stat	`-AA --ob`
pmake	`!=\xb\x1$\x0\n\x1\x1`
tao_loadmonitor	`'\n "' '\t'`
transqt2	`-f\x1\x1 .`
ttfdump	`ttc`
vgrep	`x \'-/`
xrootd	`--bbbbc \"`

Replay to eliminate false reports

Mass checking:

Type	Tests	Code Sites	Programs
Double Free	56	43	33
Dangling Access	405	147	46
Uninitialized Read	1195	94	230
Total	1656	240	267

Example bug (cpio, simplified):

```
tape_buffered_read (
  ((char *) short_hdr) + 6,
  in_des, sizeof *short_hdr - 6);
file_hdr->c_namesize = short_hdr->c_namesize;
file_hdr->c_name = (char *)xmalloc (
  file_hdr->c_namesize);
cpio_safer_name_suffix (file_hdr.c_name, ...);
char *p = safer_name_suffix (name, ...);
size_t prefix_len=FILE_SYSTEM_PREFIX_LEN(
  file_name);
```

Mass results commentary

- 1919 binaries from host system
- Confirmed with host valgrind
- Valgrind replay is lossy; conservative

cpio bug

- Specially crafted input
- Buffer length defined by `c_namesize`
- Accesses 0-byte buffer
- Spread over three source files

Cross-checking for integrity

Binary symbolic execution and replay with integrity mechanisms.

Machine Checked:

Feature	**Machine Check**
Decoder	Hardware execution
Interpreter $\times$ JIT	Logging replay
JIT $\times$ HW	Shadow process replay
Expression optimization	SMT solver
Soft floating-point	Test pooling, Hardware FPU
Symbolic heap violations	valgrind replay
Unconstrained symbolic pointers	Compiled to C source; core files

Finds Bugs:

- Third party binary programs
- Third party floating-point solvers
- Decoder (vex fp, etc)
- Interpreter and JIT
- Expression builder

Limitations

1. Cannot reason about paths it doesn't execute

2. May not terminate on some paths (constraint solving is NP complete)

3. Cannot reason about objects of symbolic size (chooses one possible concrete value) and cannot reason about symbolic pointer referring to multiple objects (chooses one object or tries each object)

4. Does nothing special about loops with symbolic bounds (running them explores all possible bounds, one at a time)

Running embedded code = hard

- Trick: run code in a vacuum
  - Mark inputs as symbolic, but missing preconditions (“under-constrained”)
  - Jump to it.
  - Lazily allocate memory on dereference
  - Reason carefully to avoid false positives.
- Under-constrained rules:
  - Normal symbolic: flag error if *any* feasible value causes error
  - Under-constrained: flag error if *all* feasible values cause error
  - Key: after dangerous operation assert precondition holds

Under-constrained execution

```
void wildly_contrived(list *l, int x, int y) {
    int scale = 10 / x;          // POST: x != 0
    lock(l->lock);               // POST: l != null
                                 // POST: l->lock = 1
    if(!alloc(…))
        goto error;
    ….
error:
        lock(l->lock);           // ERROR: "double lock!"
         ….
```

Using UC execution to make verification easy

- The Grail: Verification of real code.
  - Unfortunately: tedious, hard, impossible. Etc.
  - Real coders rarely write comments.
  - No one writes formal specs.
- Trick: use klee + cross-checking to eliminate need for specification
  - Combine with UC to do finite verification of code fragments
  - Case 1: Cross checking multiple library implementations
  - Case 2: Cross checking that patches only remove errors.
  - Result: sometimes verification easier than writing one test case!

Context: Proving path equivalence

- 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 = equivalent
     b. Klee terminates w/ error: mismatch found

If terminates, then has proven equivalence up to that input size.

If one path correct, verified the other is correct.

Often have > 1 impl of interface

f	? =	f'
Different implementation of standard interfaces (e.g., libraries, servers, compilers)		
Reference implementation		Real-world one
Yesterday's code		Today's
cc –O0 kernel.c		cc –O3 kernel.c
klee foo.c		klee foo.c (!)
$f' = f^{-1}(x)$ assert(f(f'(x)) = x)		

UC-Klee: Easy (finite) verification.

- Input: two library routines that purport to be equal.
  - (Don't need spec: any difference on legal inputs = error.)

- Uses sound symbolic execution to automatically explore (ideally all) paths
  - Automatically makes up inputs on demand
  - Automatically walks over all outputs, verifying equivalence.
  - User does *zero* work in common case

- Result: easier to show equivalence than to write one test!
  - Verified equivalance of 100s of ugly C routines.
  - Found many bugs.
  - Caveat: finite input size, finite number of paths (*), illegal inputs may differ

Verification Example (6.8s)

Newlib

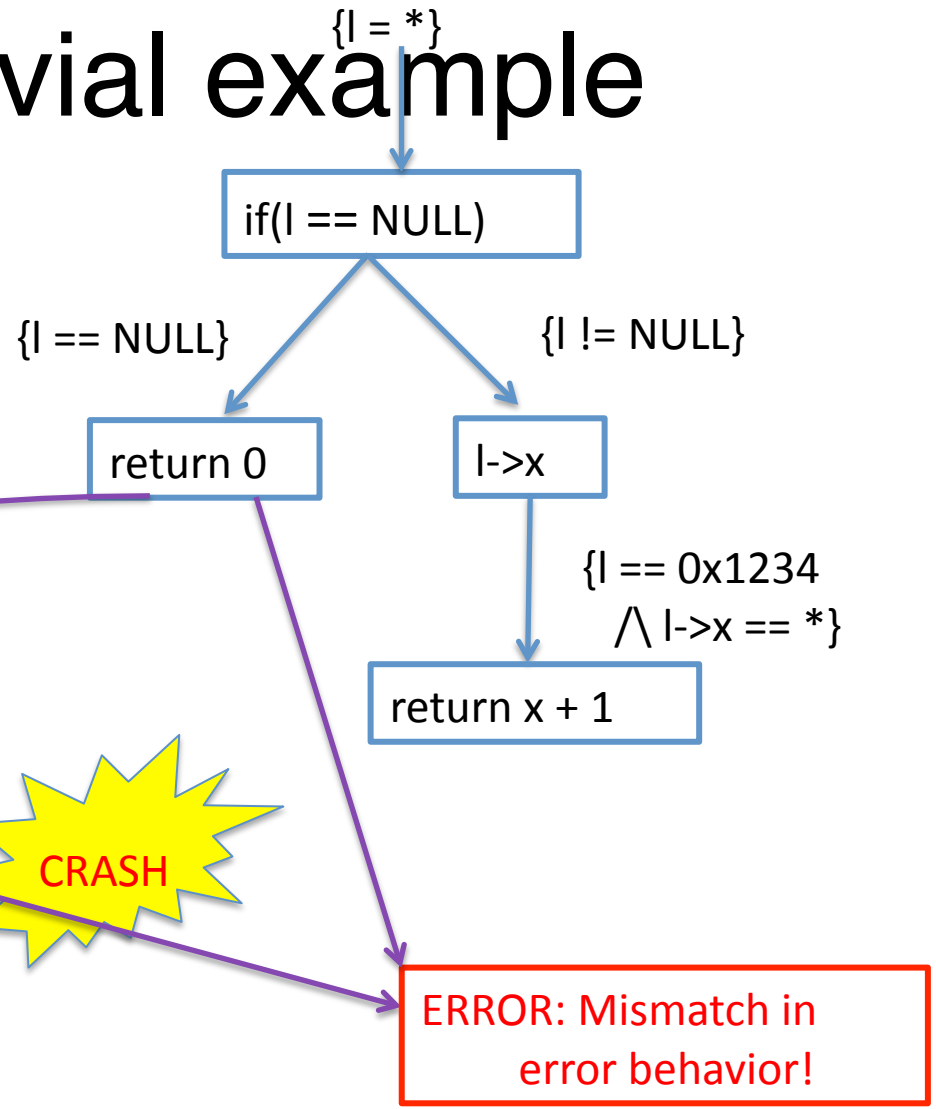
```
int ffs(int i) {
  char n = 1;
  if (!(i & 0xffff)) { n += 16; i >>= 16; }
  if (!(i & 0xff))   { n += 8; i >>= 8; }
  if (!(i & 0x0f))   { n += 4; i >>= 4; }
  if (!(i & 0x03))   { n += 2; i >>= 2; }
  return (i) ? (n+((i+1) & 0x01)) : 0;
}
```

uClibc

```
int ffs (int word) {
  int i=0;
  if (!word)
    return 0;
  for (;;)
    if (((1 << i++)&word) != 0)
      return i;
}
```

Trivial example

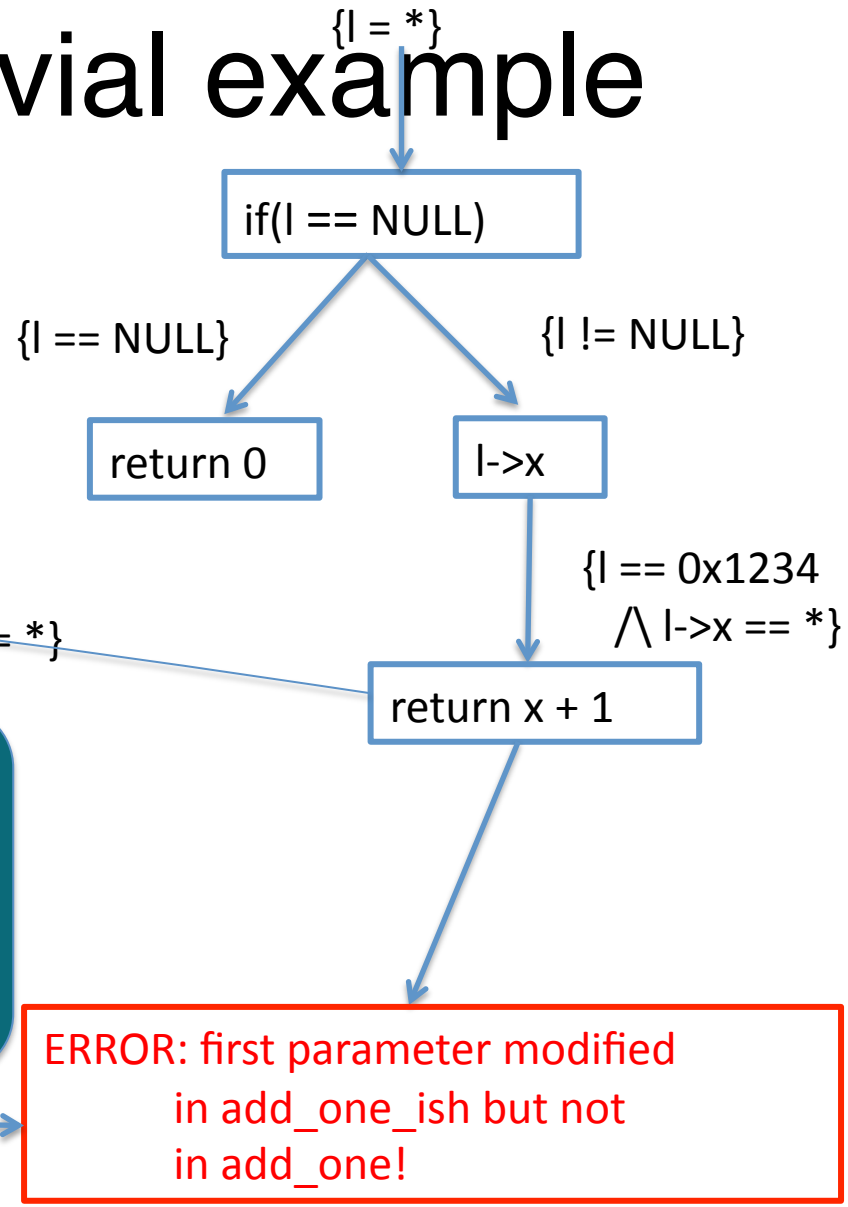
```
int add_one(list
*l) {
   if(l == NULL)
      return 0;
   int x = l->x;
   return x + 1;
}
```




```
int
add_one_ish(list
*l) {
  int x = l->x++;
  return x+1;
}
```

Trivial example

```
int add_one(list
*l) {
   if(l == NULL)
      return 0;
   int x = l->x;
   return x + 1;
}
```




```
int
add_one_ish(list *l) {
  int x = l->x++;
  return x+1;
}
```

Error Equivalence

- Real code exhibits similar behavior on illegal inputs
- Uninteresting differences can be filtered with simple C code:

```
if (c < EOF || c > 255)
  return 0;
else
  return isdigit(c) != 0;
```

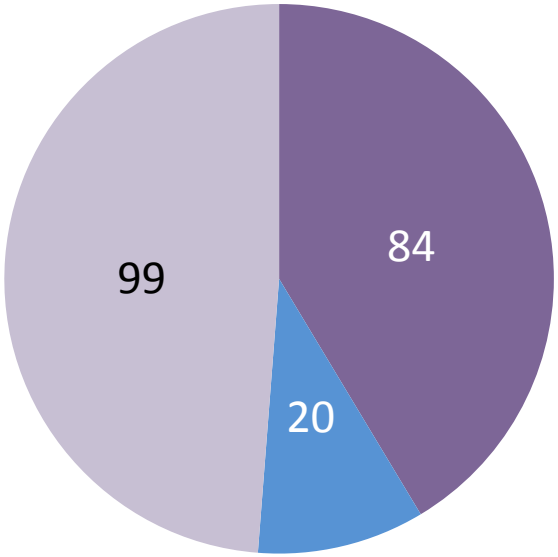
- Filters only required **after** UC-KLEE finds differences

Experiments

1. Two versions of uClibc (203 routines)
2. RedHat's Newlib vs. uClibc (143 routines)
3. uClibc compiled with high optimization (-O3) vs. no optimization (622 routines)
4. uClibc against itself (622 routines)

Results

uClibc Versions

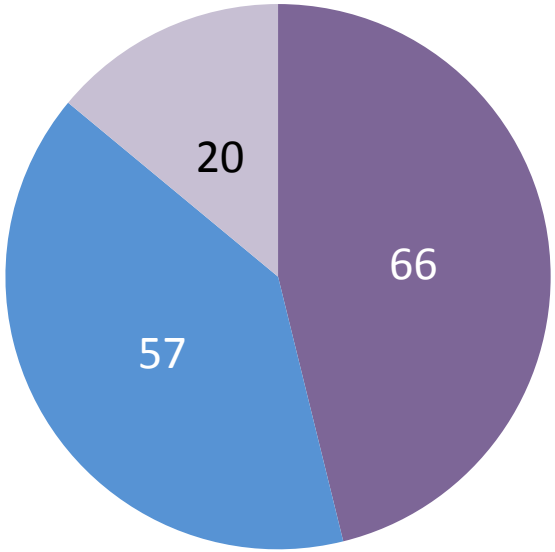


Verified

Different

Unverified

Newlib/uClibc



203 Routines

Median Coverage: 100%

143 Routines

Median Coverage: 90.1%

Max. Runtime: 10 minutes per routine

Results

	Newlib/ uClibc	uClibc Versions	LLVM Optimizer	UC-KLEE Self Check
Procedures Checked	143	203	622	622
Procedures Verified	66	84	335	335
Differences Detected	57	20	70	12
No Diff. (timeout)	15	30	85	91
KLEE Limitations	4	56	94	147
UC-KLEE Limitations	1	13	38	37
100% Coverage	59	105	367	375
Mean Coverage	72.2%	80.7%	85.6%	85.6%
Median Coverage	90.1%	100.0%	100.0%	100.0%

Newlib:

```
int _remove_r(struct _reent *ptr,
              const char *filename) {
  if (_unlink_r (ptr, path) == -1)
    return -1;
  return 0;
}
int remove(const char *filename) {
  return _remove_r(_REENT,
                   filename);
}
```

uClibc:

```
int remove(const char *filename) {
  int saved_errno = errno;
  int rv;
  rv = rmdir(filename);
  if ((rv < 0) && (errno == ENOTDIR))
{
    __set_errno(saved_errno);
    rv = unlink(filename);
  }
  return rv;
}
```

Bugs Found

uClibc:

`ctime`
`ether_aton`
`ether_aton_r`
`ether_line`
`vsnprintf`

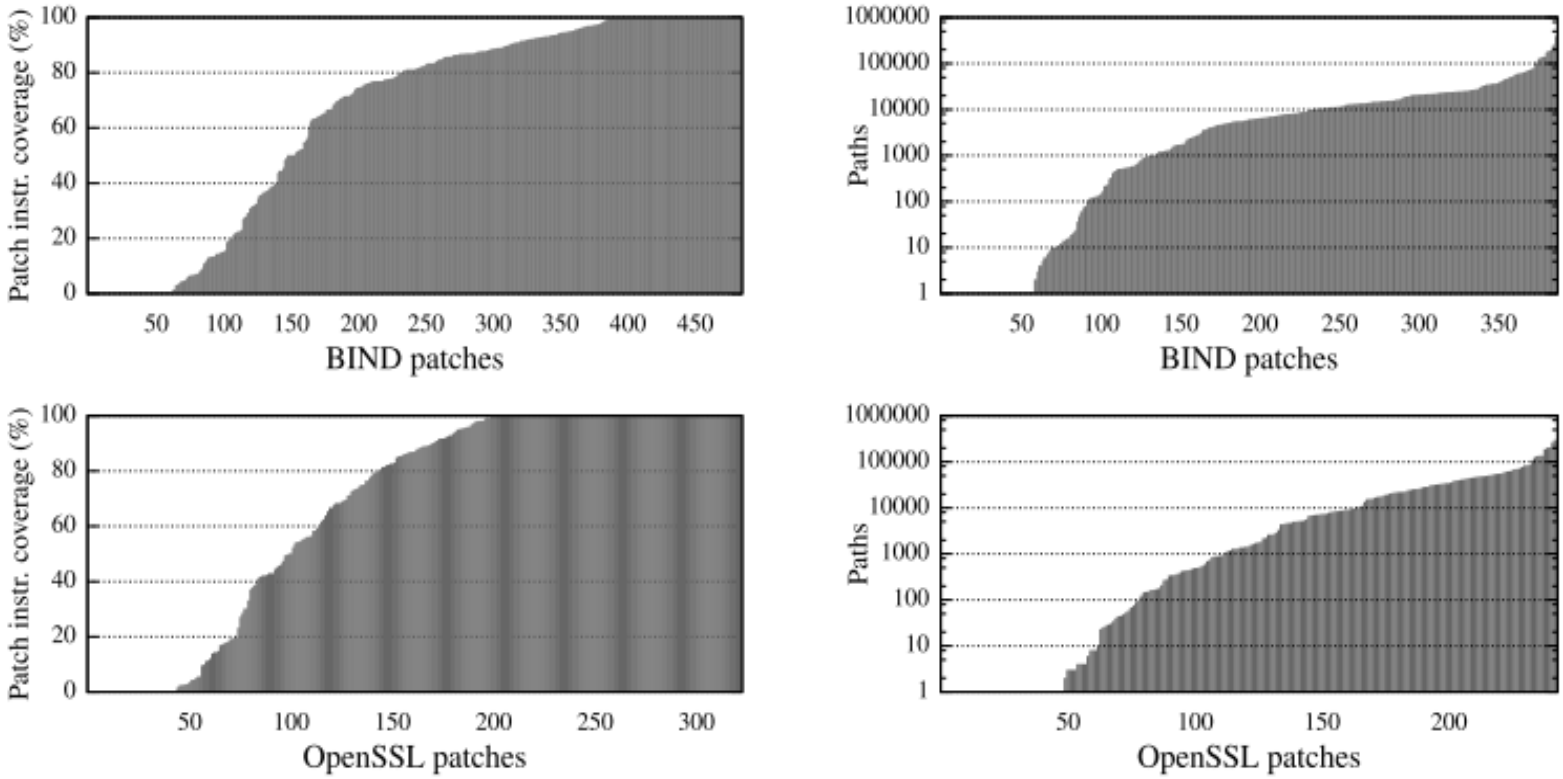
Newlib:

`memccpy`
`memcpy`
`memmove`
`mempcpy`
`remove`

Validating bug-fix patches

- Many patches just remove security holes or crashes
  - Patched code should have same functionality as unpatched, just fewer errors.
- Use cross-checking to verify patched function has no new errors
  - Checked 487 BIND patches and 324 OpenSSL patches
  - High median coverage.
  - Serious bugs. One CVE in OpenSSL

High code / path coverage

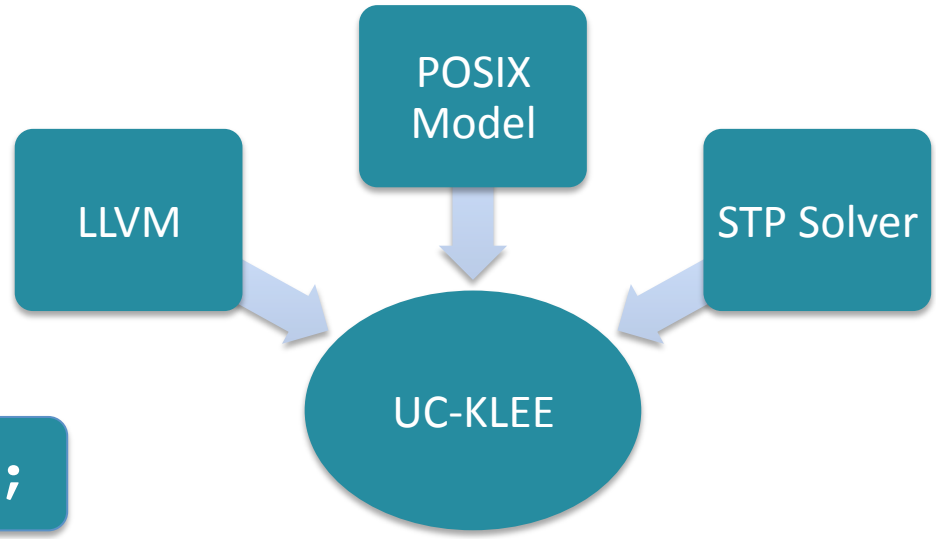


- Median coverage: 81% BIND, 87% OpenSSL.
- Median paths checked: 5,828 BIND, 1,412 OpenSSL

OpenSSL CVE

```
/* do_ssl3_write */
/* If we have an alert to send, lets send it */
if (s->s3->alert_dispatch) {
  /* [call sets wb->buf to NULL] */
  i=s->method->ssl_dispatch_alert(s);
  if (i <= 0)
    return(i);
  /* if it went, fall through and send more stuff */
}
...
unsigned char *p = wb->buf; /* <-- p = NULL */
*(p++)=type&0xff; /* NULL deref */
```

The joys of assert(f(x)==f(x))




```
assert(f(x) == f(x));
```

- Detected severe bug in LLVM 2.6 optimizer
- Found UC-KLEE bugs during development

Where this is headed

- Effective under-constrained execution
  - Jump to arbitrary routine in embedded code, start checking, good bugs, low false positives
  - Requires careful reasoning about concrete, fully symbolic, and symbolic data where we are missing constraints. The latter can taint all of the former.
- Co-design of tool and code.
  - Simple methods of writing code so that it's orders of magnitude easier to check?
- Symbolic execution is currently way to hard.
  - Static bug finding is much much easier.
  - Finds many bugs that sym ex misses
  - A static lesson: adding more analysis often makes bugs worse.
  - There must be a way to get the best of both.

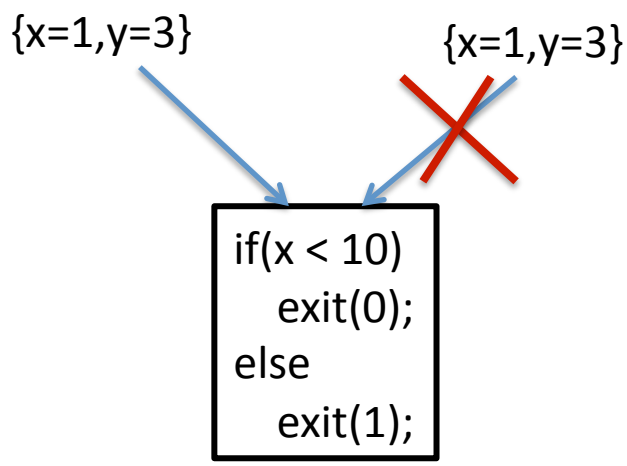
Conclusion

- Long term goal:
  - Take arbitrary program, routinely crush it under tests that exercise all interesting paths with all values.
  - Success = you are surprised when your program breaks.

- Not there, but approach has promise:
  - High coverage on many real, unaltered applications
    - Handily beats good manual test suites
  - Good bugs in very heavily tested code.
    - Can often generate inputs of death to crash uninstrumented code
  - Low-effort path verification from assert(f(x) == f'(x));
    - In interesting cases, easier to verify code than to write a single test case.

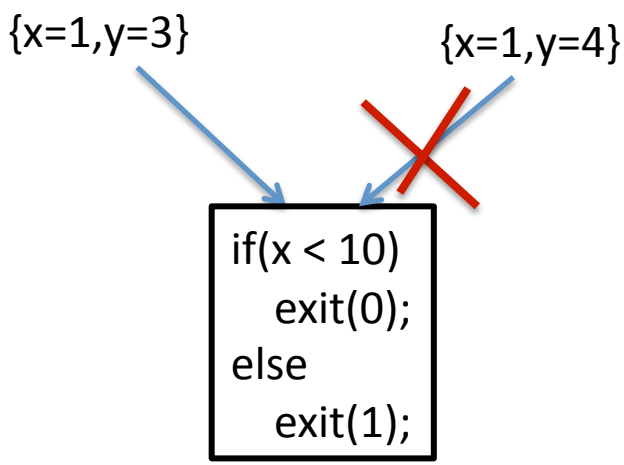
Scaling klee: pruning paths

- Exponential paths, but many are the same
  - If execution reaches same program point with same values, will produce same result, so stop.



Scaling klee: pruning paths

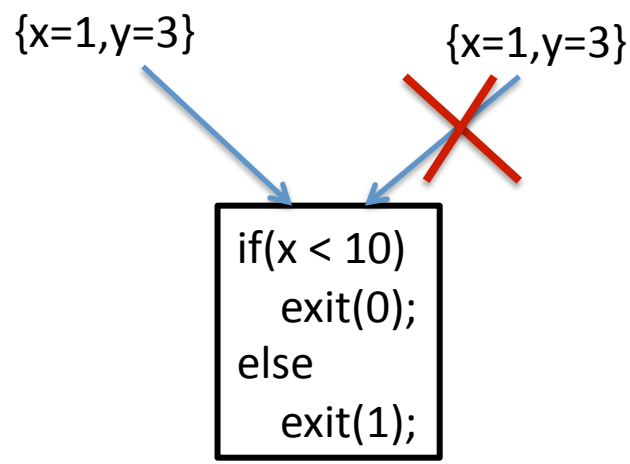
- Never the same b/c so precise.
  - Key: discard any values that can't affect execution



  - This gets very tricky, but will spare you the details.
  - Result: Huge speedup on real applications.

Scaling klee: pruning paths

- Exponential paths, but many are the same
  - If execution reaches same program point with same values, will produce same result, so stop.



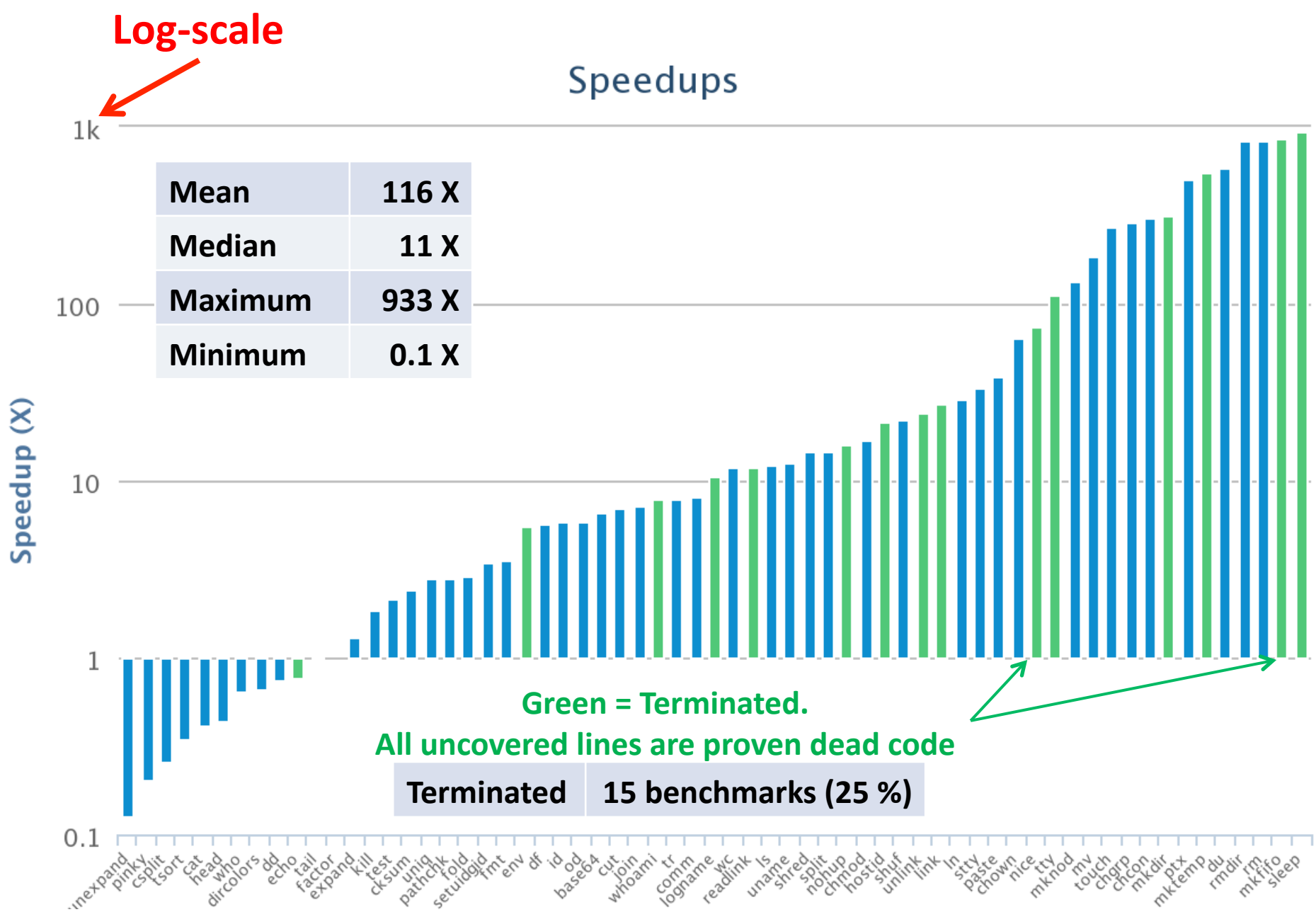
  - Complicated. But works: last time ~50x ave, 10x median speedup.

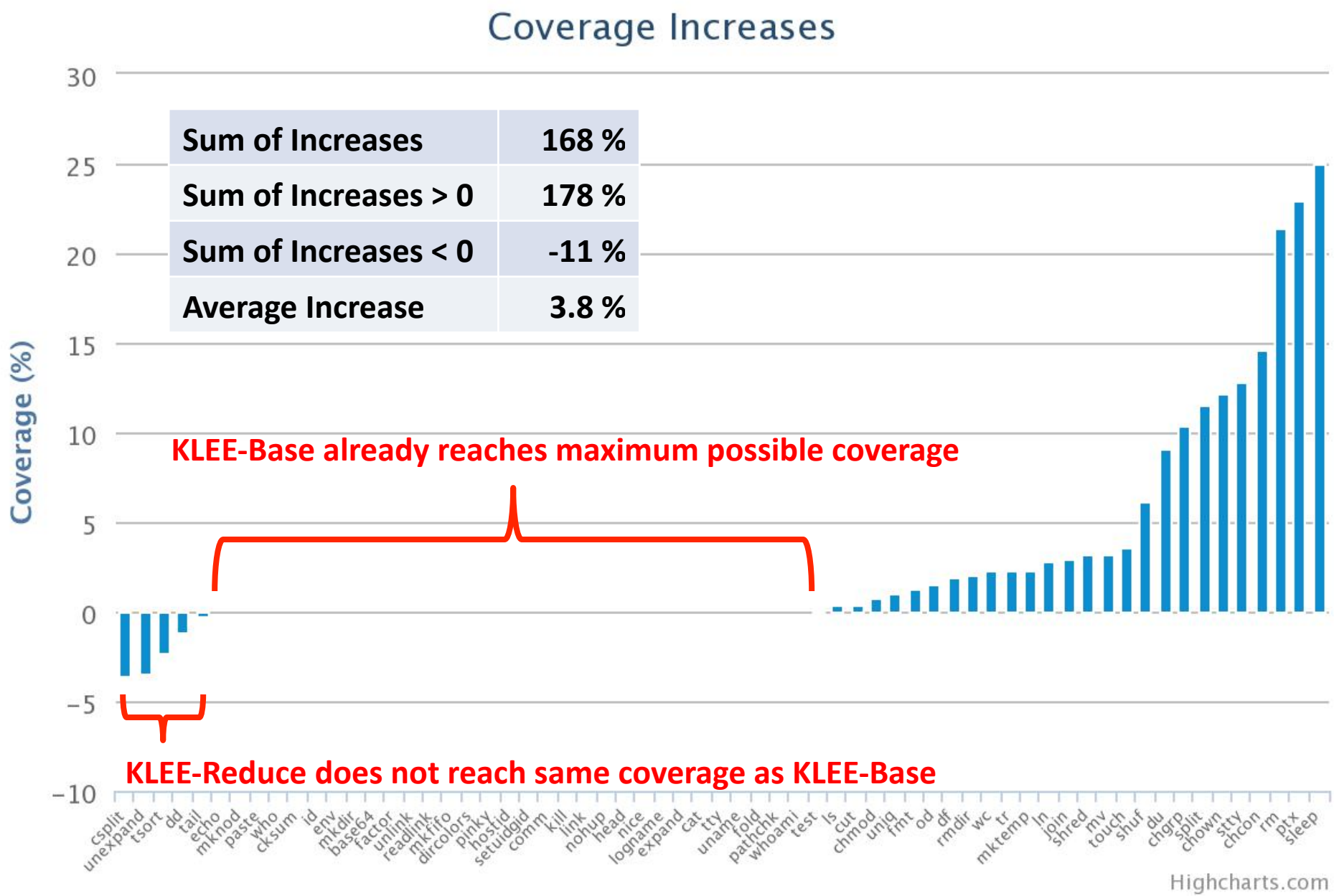
Speedup Calculation

- Ran each program for one hour with KLEE-Base and one hour with KLEE-Reduce

1. If $C_{Base} = C_{Reduce,}$
   T_{Base} = time it took KLEE-Base to reach C_{Base}
   T_{Reduce} = time it took KLEE-Reduce to reach C_{reduce}
   speedup = T_{base} / T_{reduce}

2. If $C_{base} > C_{reduce}$
   T_{base} = time it took KLEE-Base to exceed C_{Reduce}
   T_{reduce} = 1 hour
   speedup = T_{base} / T_{reduce}
3. If $C_{base} < C_{reduce}$
   T_{base} = 1 hour
   T_{reduce} = time it took KLEE-Base to exceed C_{base}
   speedup = T_{base} / T_{reduce}


Log-scale
Speedups
Speedup (X)
1k
100
10
1
0.1
Mean 116 X
Median 11 X
Maximum 933 X
Minimum 0.1 X
Green = Terminated.
All uncovered lines are proven dead code
Terminated 15 benchmarks (25 %)
unexpand pinky csplit tsort cat head who dircolors dd echo tail factor expand kill test cksum uniq pathchk fold setuidgid fmt env df id od base64 cut join whoami tr comm logname wc readlink ls uname shred split nohup chmod hostid shuf unlink link ln stty paste chown nice tty mknod mv touch chgrp chcon mkdir ptx mktemp du rmdir rm mkfifo sleep



Sum of Increases	168 %
Sum of Increases > 0	178 %
Sum of Increases < 0	-11 %
Average Increase	3.8 %