KLEE

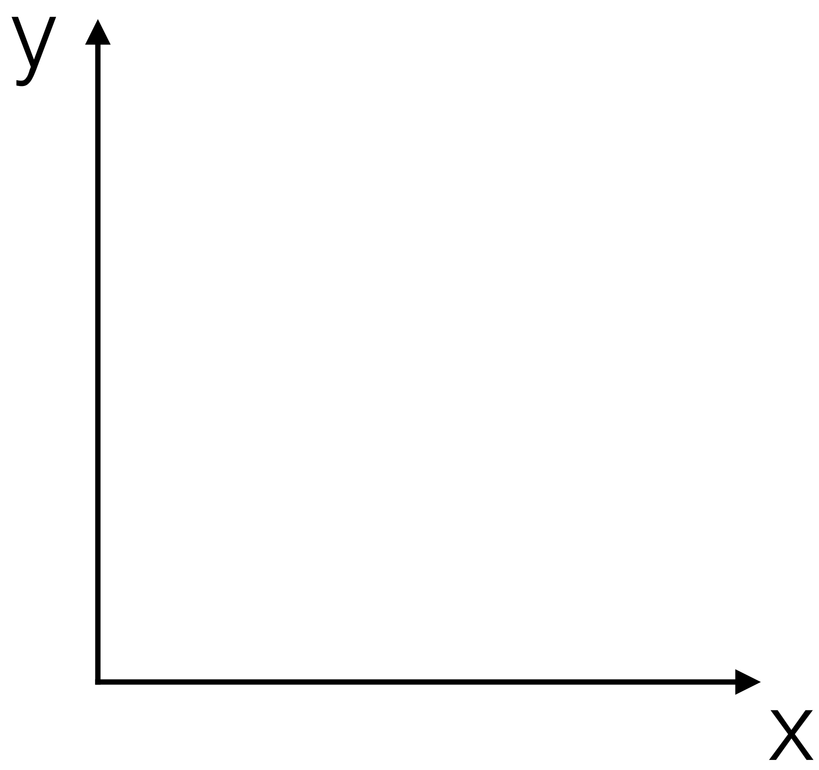
Symbolic Execution

How would you test this program?

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

Symbolic Execution

How would you test this program?

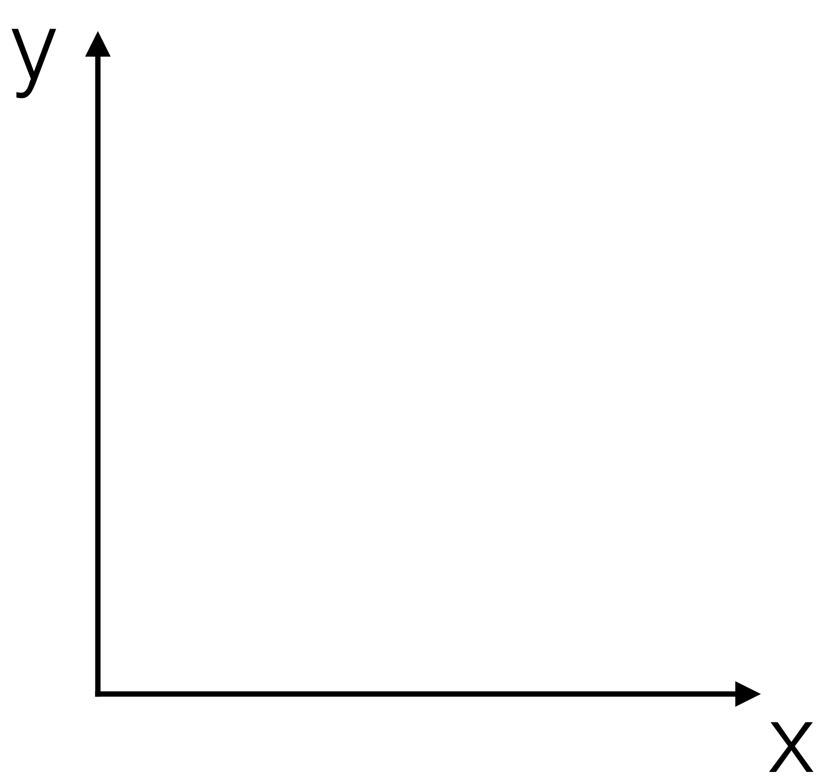



```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

Exhaustively! 2^{64} inputs to try…

Symbolic Execution

How would you test this program?

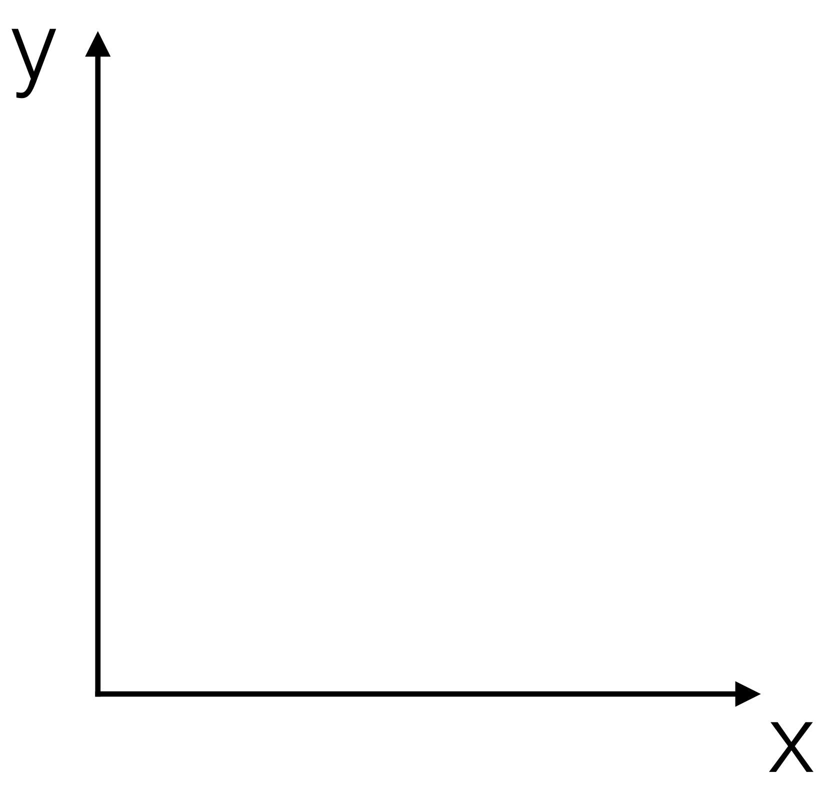



```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

Exhaustively! 2^{64} inputs to try…

Symbolic Execution

How would you test this program?

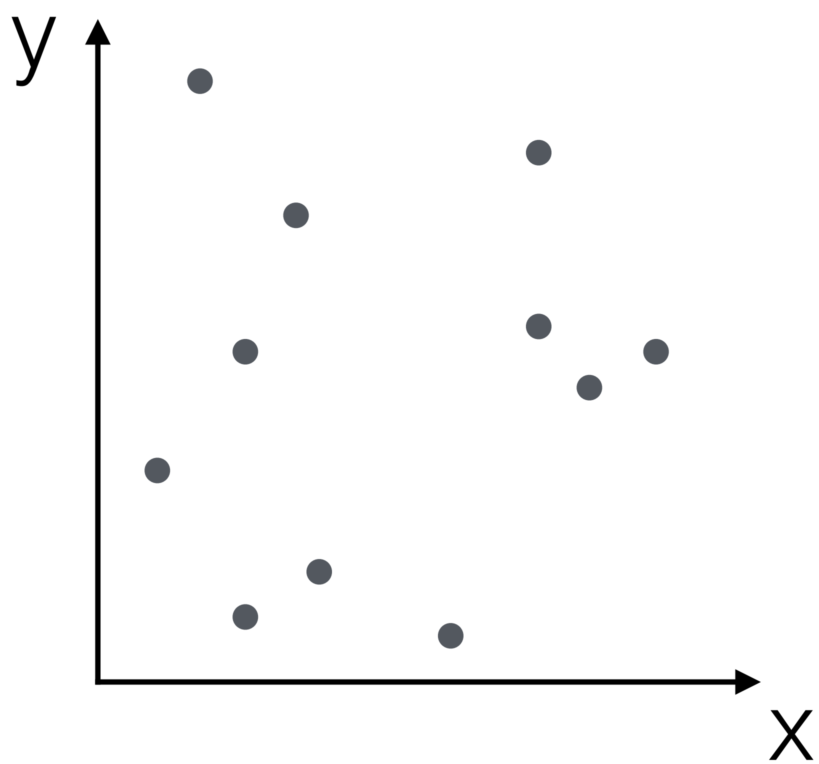



```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

Randomly? $1/2^{64}$ chance that a random input crashes.

Symbolic Execution

How would you test this program?




```c
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

Randomly? $1/2^{64}$ chance that a random input crashes.

Symbolic Execution

How would you test this program?

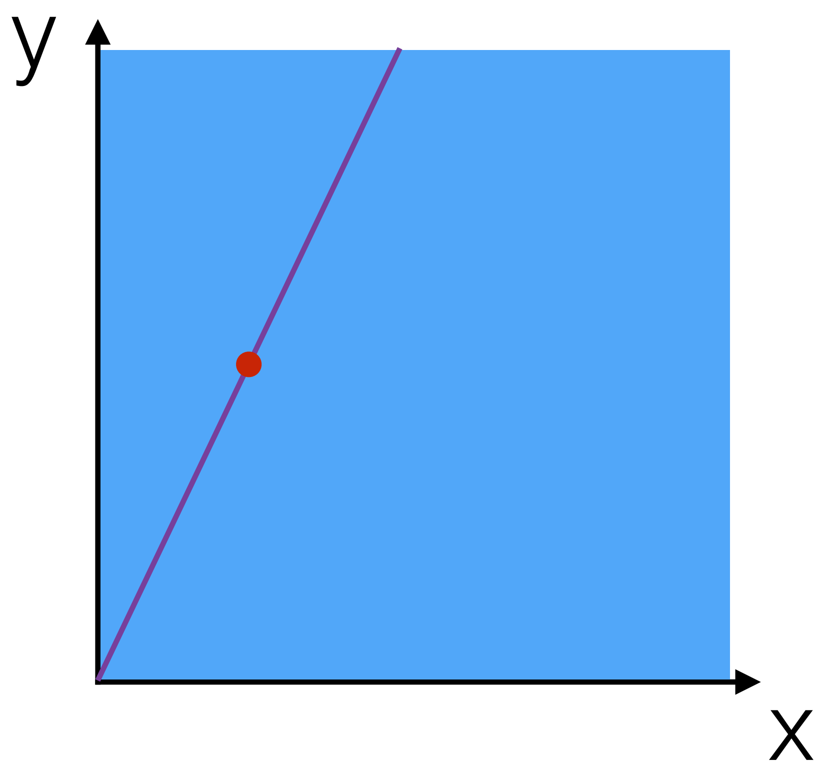
z == 2x
z == y
y == x+10
…
y = 2x and 2x = x+10
…
y = 20, x = 10

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

Logically!

Symbolic Execution

How would you test this program?




```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

Logically!

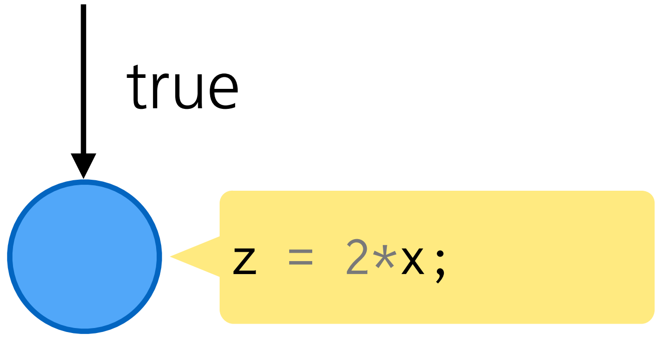
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

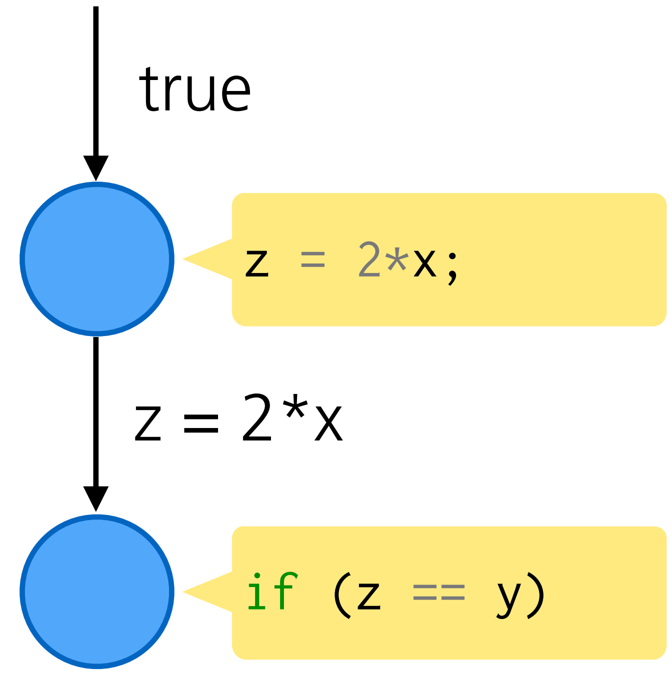
Symbolic Execution




```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```

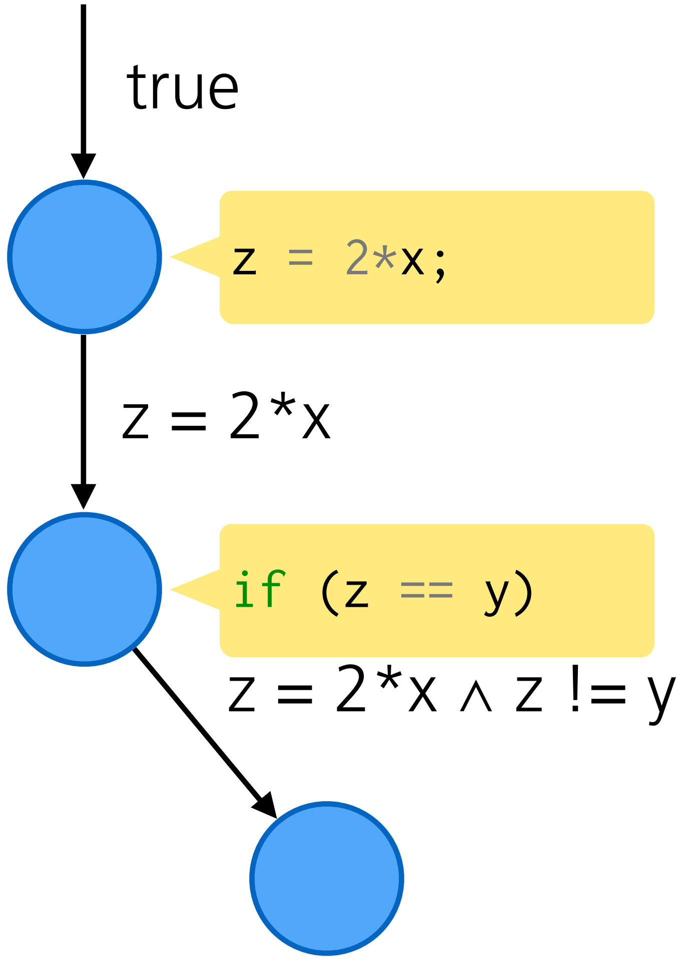
Symbolic Execution

```c
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



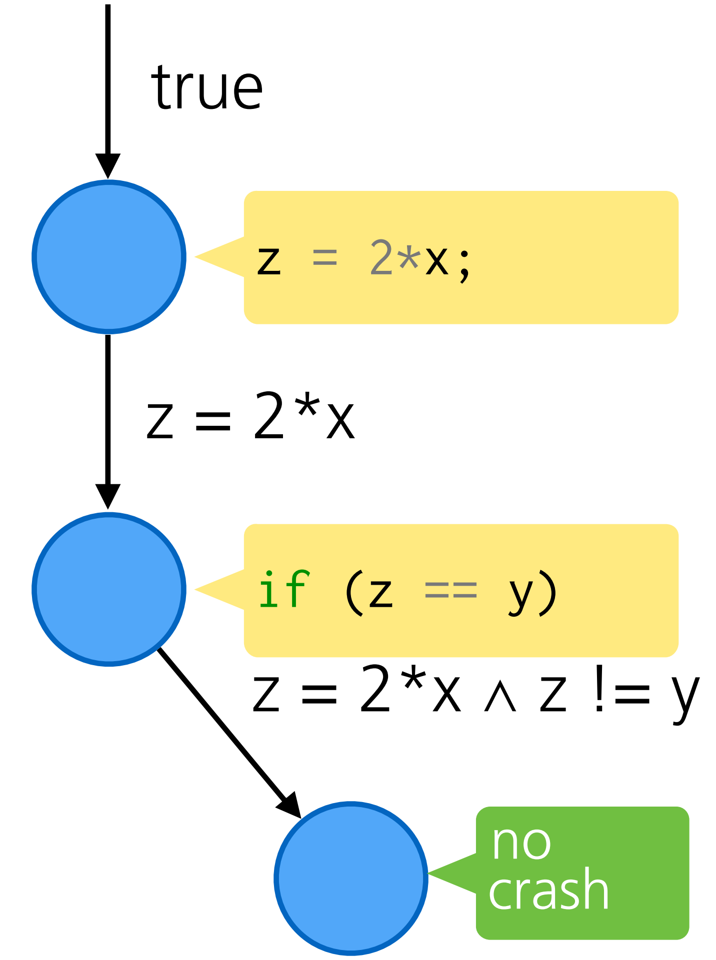
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



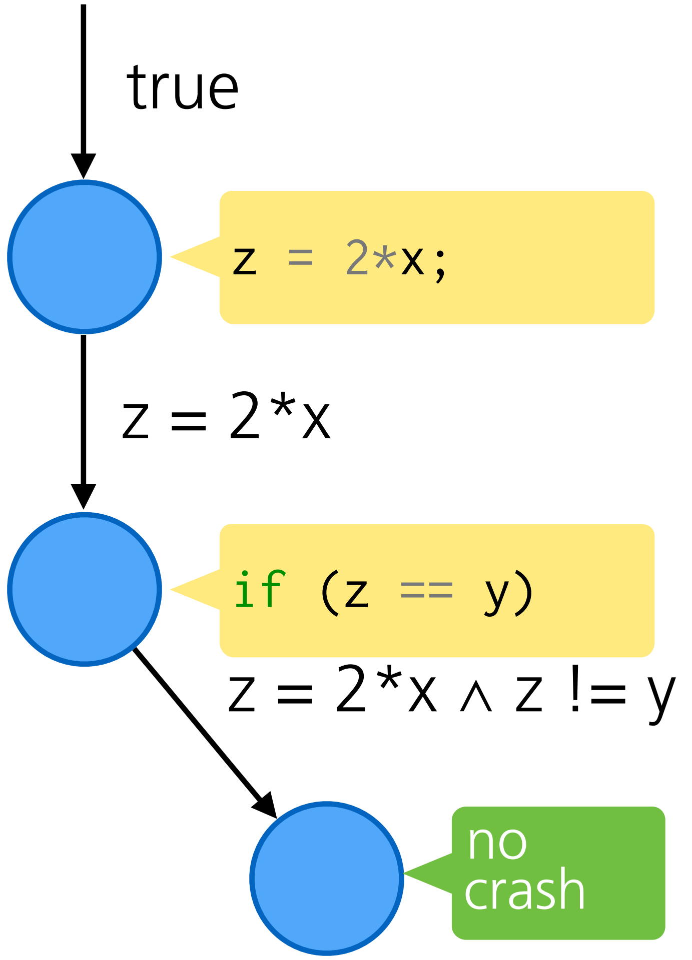
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



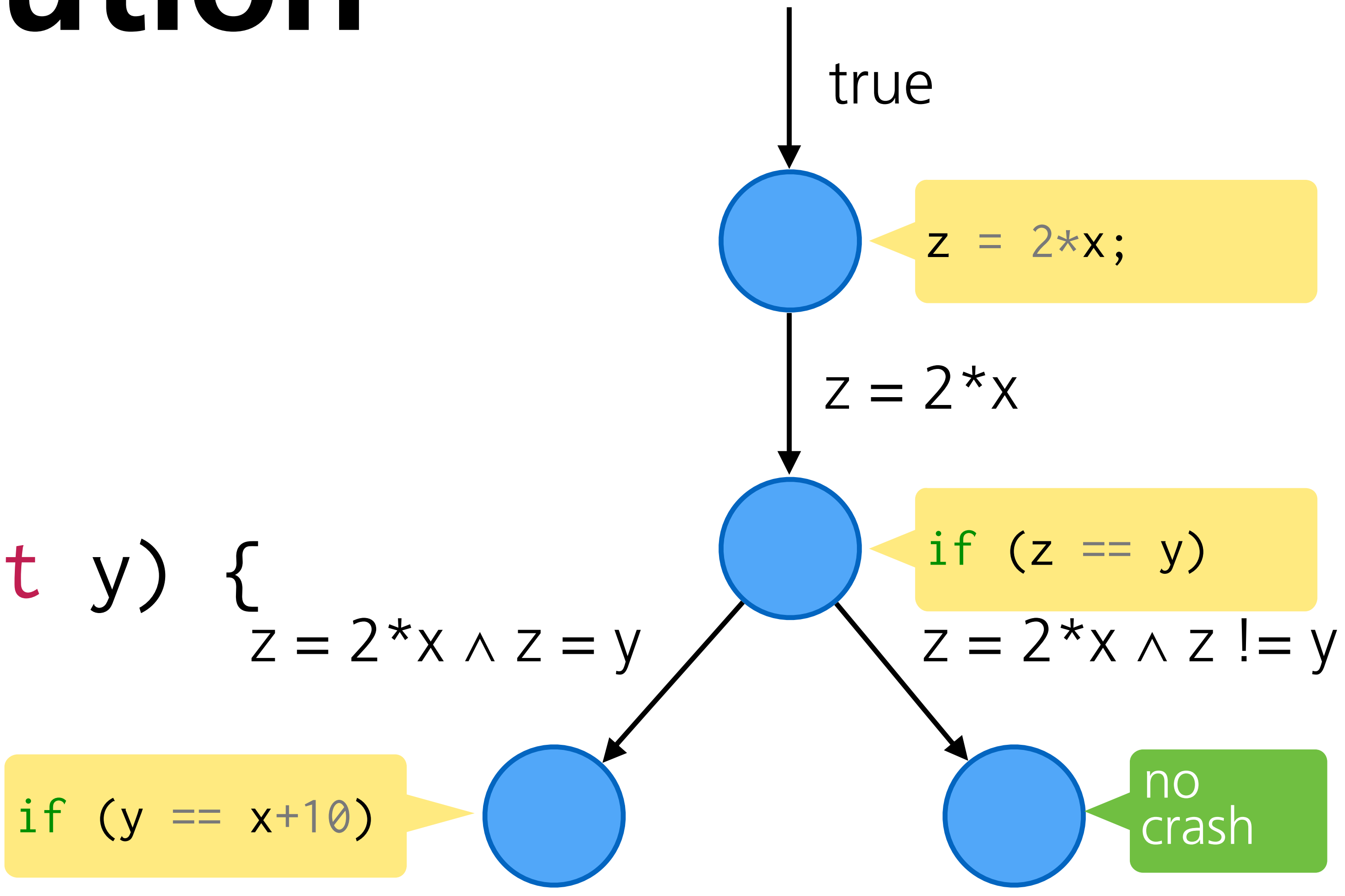
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



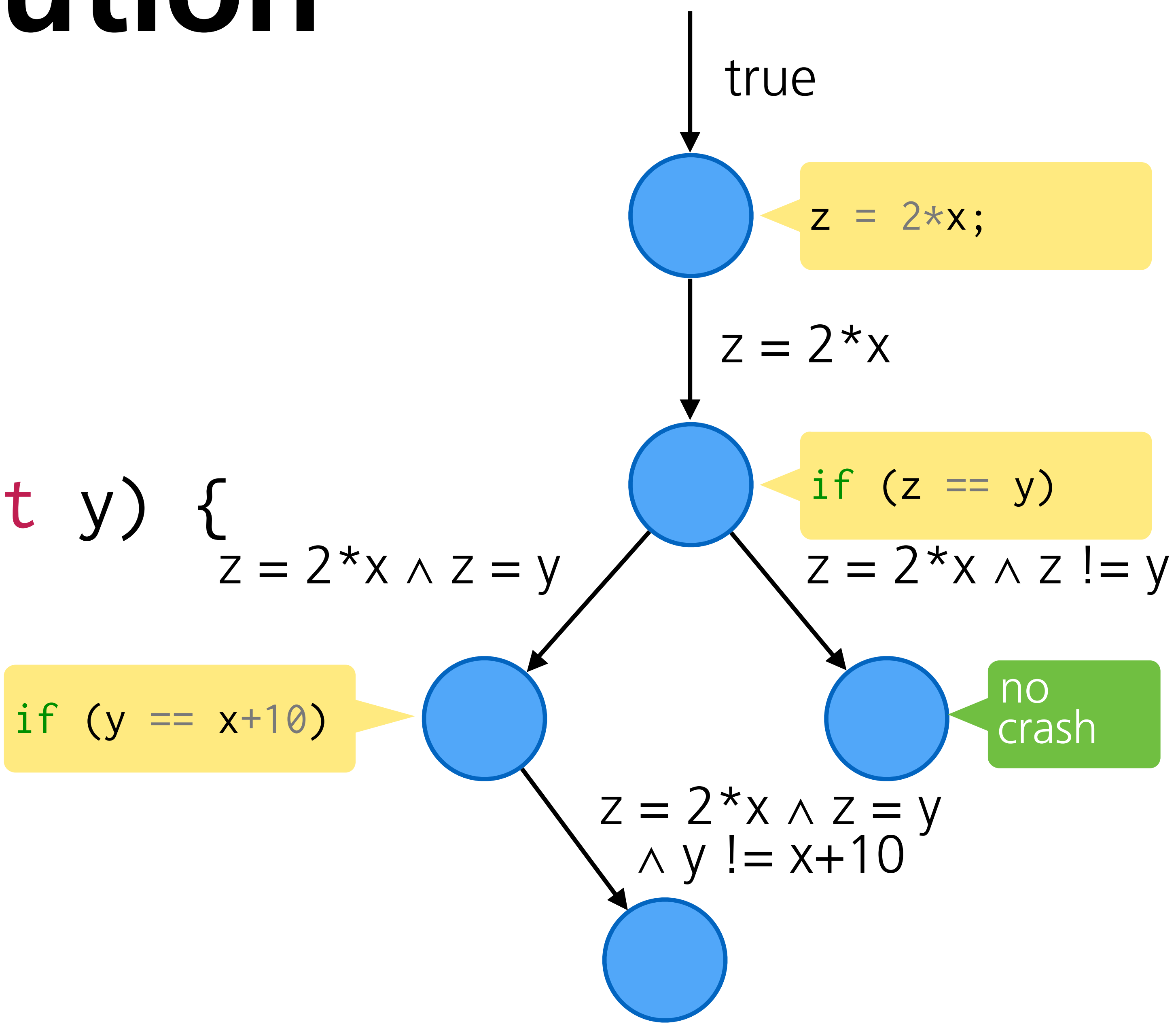
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



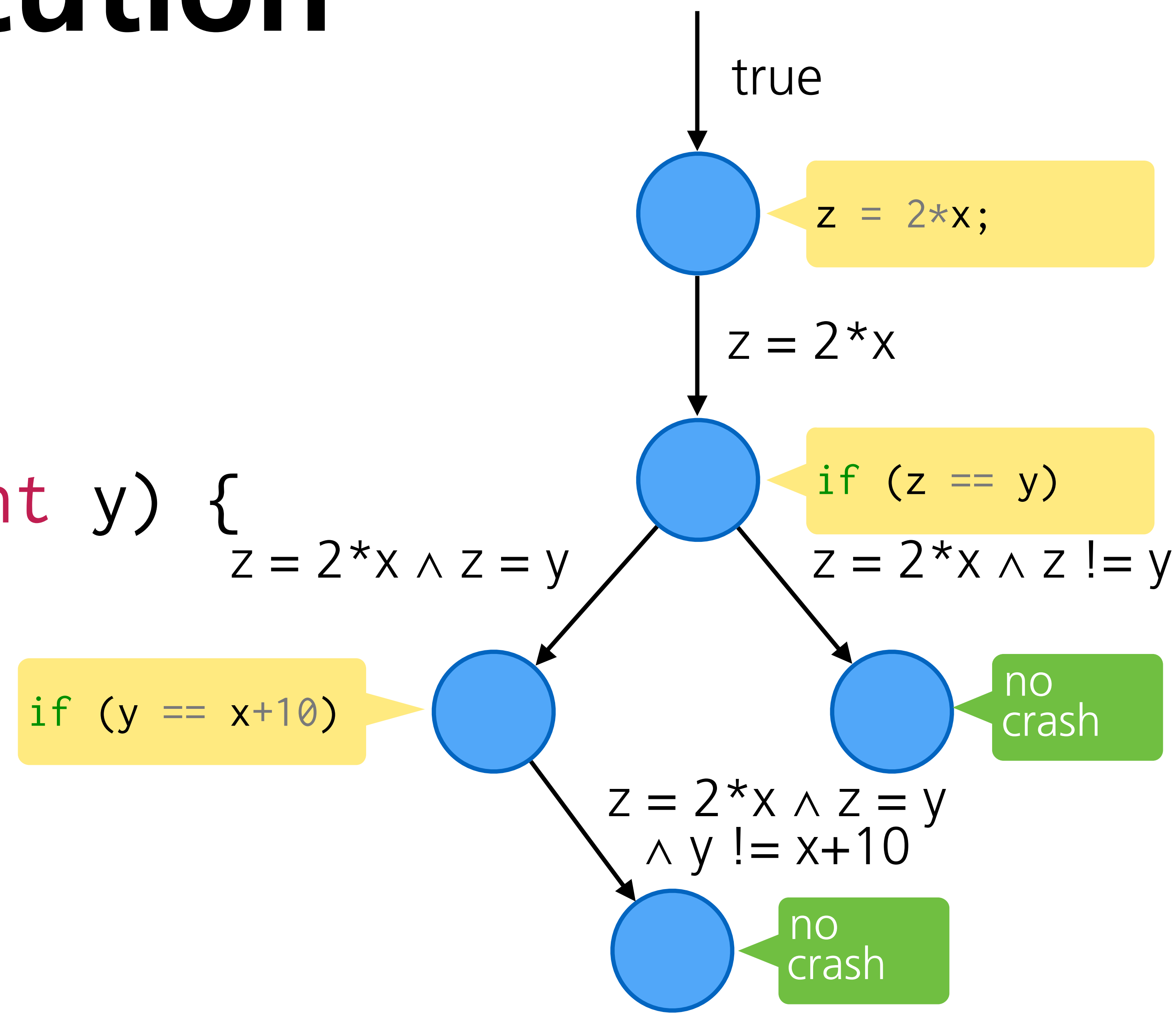
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



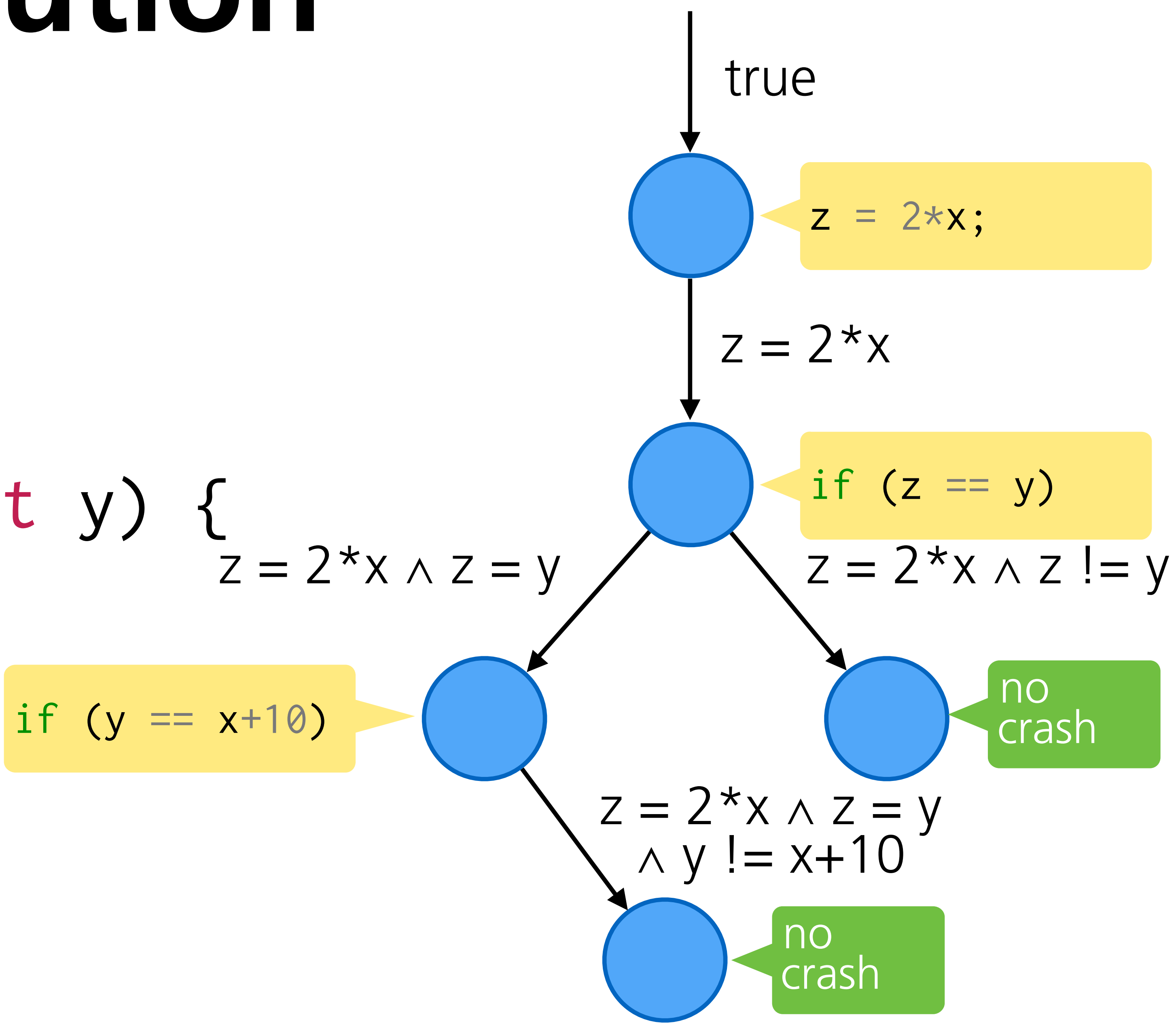
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



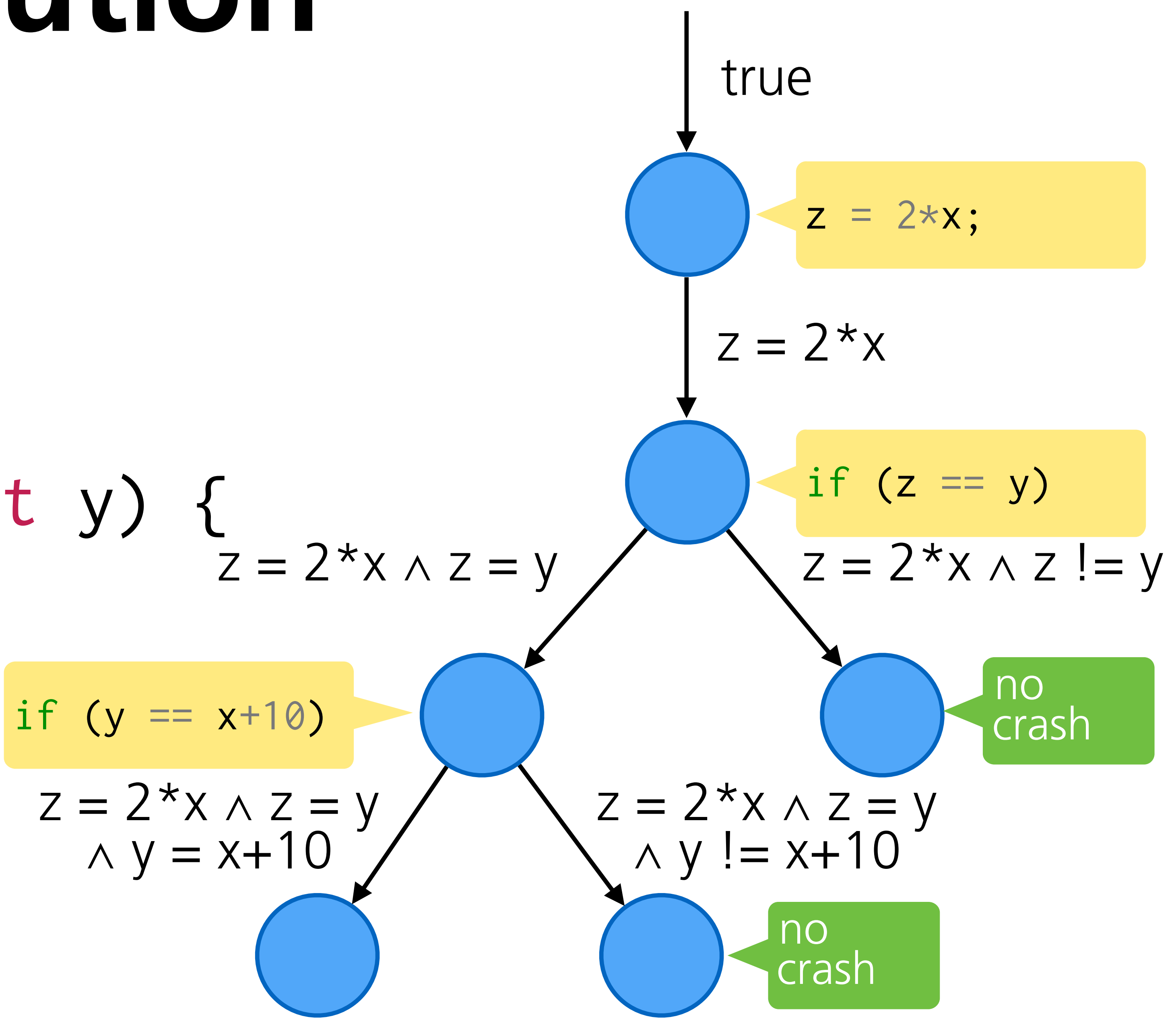
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



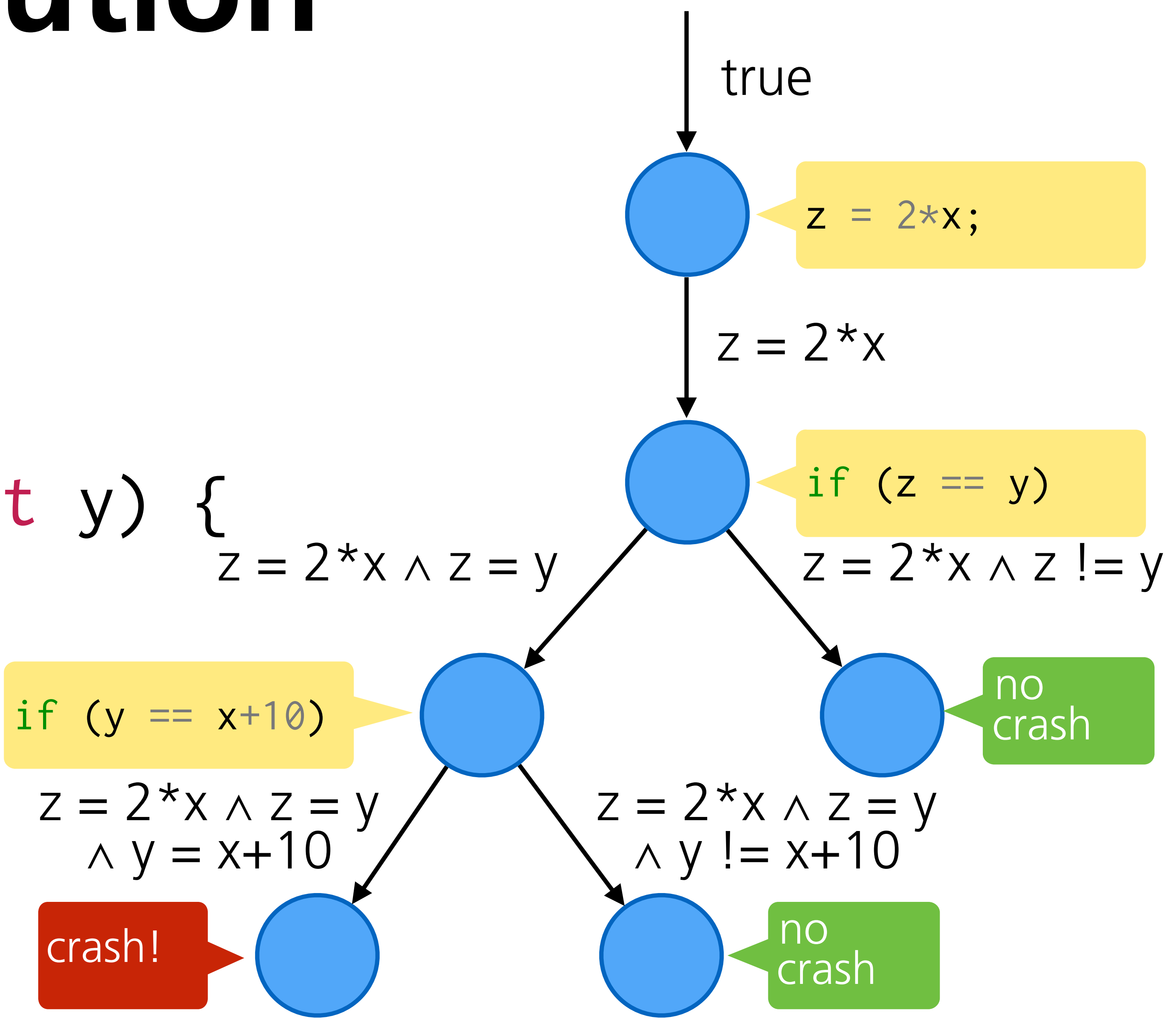
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



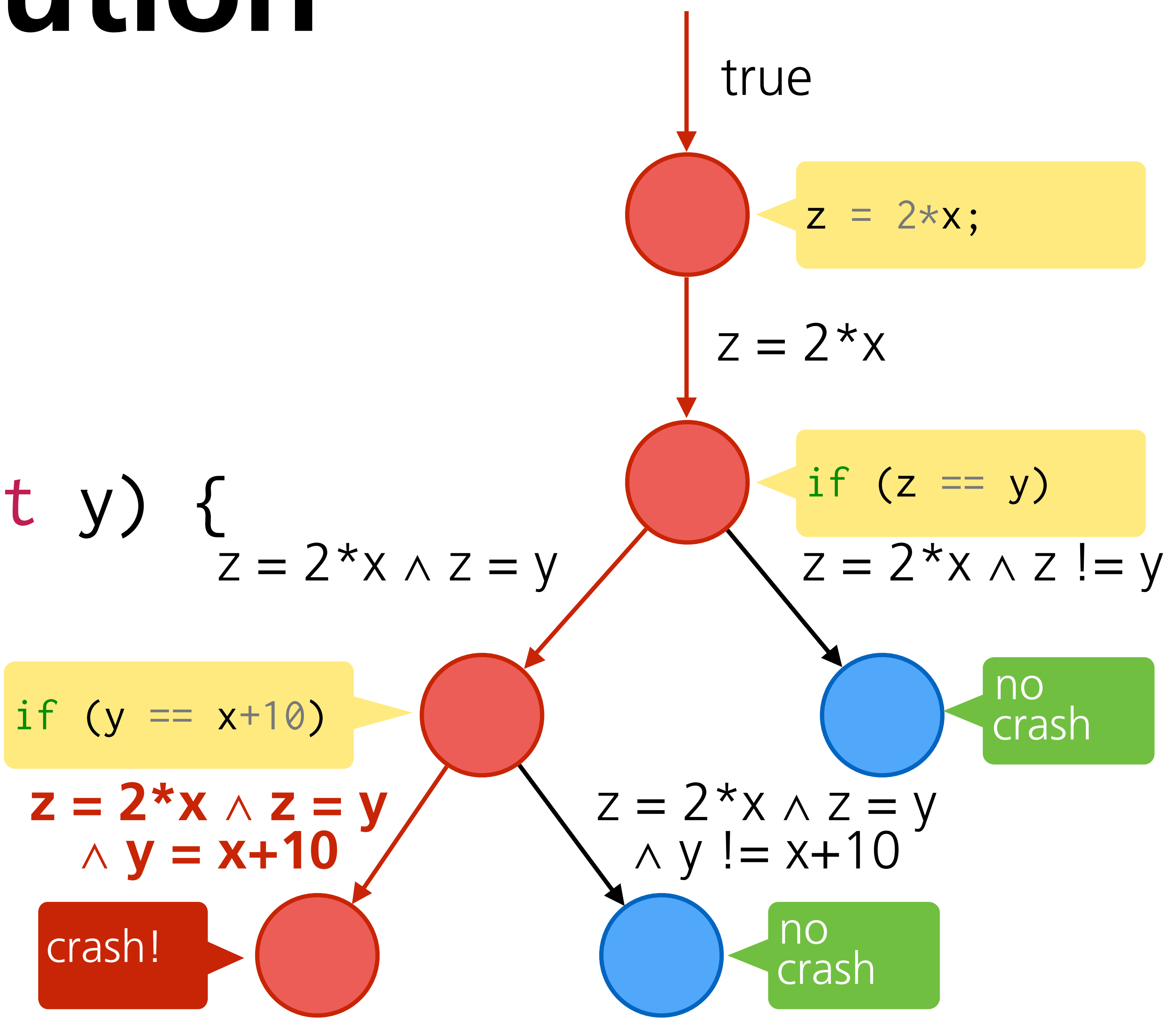
Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



Symbolic Execution

```
void test_me(int x, int y) {
  int z = 2*x;
  if (z == y)
    if (y == x+10)
      abort();
}
```



Is the path feasible? Or is the path condition contradictory?
Ask a friendly SMT solver!

Symbolic Execution

Is the path satisfiable?

$z = 2*x \wedge z = y \wedge y = x+10$

Symbolic Execution

Is the path satisfiable?

$z = 2*x \wedge z = y \wedge y = x+10$

```
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)

(assert (= z (* x 2)))
(assert (= z y))
(assert (= y (+ x 10)))

(check-sat)
(get-model)
```

Symbolic Execution

Is the path satisfiable?

$z = 2 * x \wedge z = y \wedge y = x + 10$

```
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)

(assert (= z (* x 2)))
(assert (= z y))
(assert (= y (+ x 10)))

(check-sat)
(get-model)
```

Z3

```
sat
(model
  (define-fun z () Int
    20)
  (define-fun x () Int
    10)
  (define-fun y () Int
    20)
)
```

Symbolic Execution

That’s great! But what about…

- Vagaries of a real language (C)
- Interaction with libraries
- Input/output (files, command line)

And then, make it fast enough to use.

Symbolic Execution

That's great! But what about…

- Vagaries of a real language (C)
- Interaction with libraries
- Input/output (files, command line)

And then, make it fast enough to use.

Symbolic Execution

That's great! But what about…

- Vagaries of a real language (C)
- Interaction with libraries
- Input/output (files, command line)

And then, make it fast enough to use.

Symbolic Execution

That's great! But what about…

- Vagaries of a real language (C)
- Interaction with libraries
- Input/output (files, command line)

And then, make it fast enough to use.

KLEE is symbolic execution that actually works.

KLEE Architecture

- An “operating system” for “symbolic processes”
- Actually: symbolic execution for LLVM bitcode
- “Forks” on every branch, to evaluate both sides

This all sounds like a *terrible* idea.

KLEE Architecture

To make it fast:

- Concretize instructions wherever possible
- Don’t fork for infeasible paths (ask an SMT solver)
- Don’t keep executing a path once it reaches an error
- Don’t model memory as a single flat array
  - Bad for SMT solvers — instead, model each object as a distinct array
  - Model each possibility in a points-to set as a different state

Compact State Representation

- Lots and lots of states! (up to 100k, **1GB** RAM)
- Each state needs to track all memory objects in that state — but most memory objects are rarely changed
- Copy-on-write at object granularity
- Heap is an immutable map for sharing between states
  - And can be cloned in constant time when forking

Query Optimization

- Execution time dominated by constraint solving — so do as little constraint solving as possible
- Constraint Independence
  - Only include constraints from the current state if they affect the query being evaluated
  - $\{i < j, j < 20, k > 0\}$ and query $i=20$

Query Optimization

- Counter-example Cache
  - KLEE makes many redundant queries
  - Naive cache: just map each query to its result
  - Fancier cache: can index subsets and supersets of a query
    - If A is unsatisfiable, then $A \wedge X$ is unsatisfiable
    - If $A \wedge X$ is satisfiable, then A is satisfiable
    - If A is satisfiable, its solution might also be a solution to $A \wedge X$ (and this is cheap to check)

Query Optimization

Optimizations	Queries	Time (s)	STP Time (s)
None	13717	300	281
Independence	13717	166	148
Cex. Cache	8174	177	156
All	699	20	10

Number of queries reduced by 95%
Runtime reduced by 10x

State Scheduling

- The core of KLEE is a loop that chooses the next symbolic state to evaluate
- Random Path Selection
  - State is a binary tree (nodes are forks)
  - Randomly select a path through the tree, and execute the node at the leaf
  - Why? Favors nodes high in the tree (more freedom), and avoids fork bombs from loops

State Scheduling

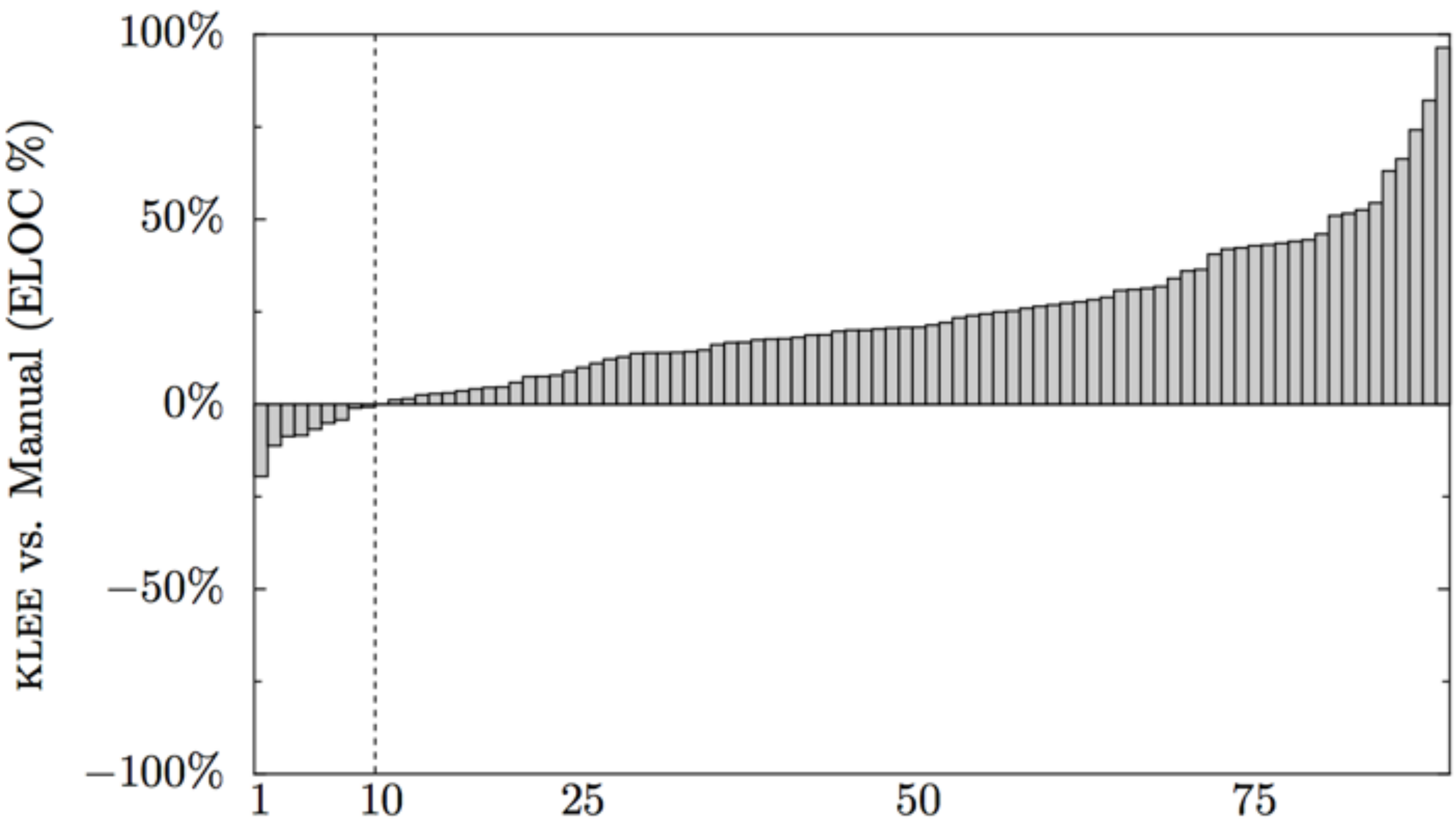
- Coverage-Optimized Search
  - There is a heuristic.
  - Guides the search towards uncovered instructions
- These two strategies are applied in round-robin style

Environment Modeling

- Real programs run on real operating systems and use real files and stuff
- Files and other inputs could be symbolic
  - Need to model all system calls for symbolic inputs (`read`, `write`, `stat`, …)
  - Modeling system calls rather than `libc` makes implementation easier — can just compile some `libc` using our system call implementations
- Can model failing system calls, and provide replay for failing test cases (via ptrace)

Evaluation

- Ran KLEE over all GNU coreutils



coverage versus coreutils test suite

Evaluation

- Can use symbolic execution to check equivalence of two implementations
- Compared Coreutils to Busybox

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		

Evaluation

- Tested the HiStar kernel, executing a single process that executes up to three system calls

Test	Random	KLEE	ELOC
With Disk	50.1%	67.1%	4617
No Disk	48.0%	76.4%	2662

Discussion

- Is coverage a good metric for measuring the quality of tests?
- KLEE's not easy to use — where is the trade-off between wrangling KLEE and just writing tests?
  - SAGE — as an x86 symbolic execution engine — is more usable?
- Handling environment is hard — KLEE shoots for 100% accuracy, SAGE doesn't. How important is it?
- Is it web scale? Is it Google scale?