Symbolic execution

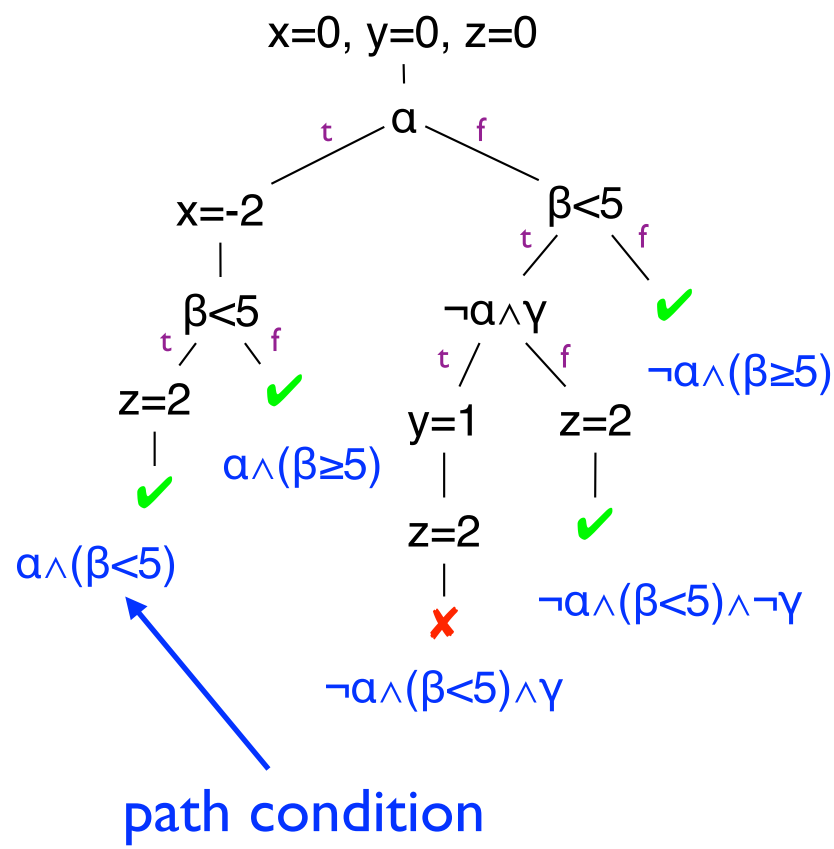
A middle ground

- Testing works: reported bugs are real bugs
  - But, **each test** only explores **one possible execution**
    - `assert(f(3) == 5)`
    - In short, **complete**, but **not sound**
  - We *hope* test cases generalize, but no guarantees

- **Symbolic execution generalizes testing**
  - "More sound" than testing
  - Allows unknown symbolic variables α in evaluation
    - `y = α; assert(f(y) == 2*y-1);`
  - If execution path depends on unknown, conceptually fork symbolic executor
    - `int f(int x) { if (x > 0) then return 2*x - 1; else return 10; }`

Symbolic execution example

```
int a = α, b = β, c = γ;
                // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z != 3)
```



Insight

- Each **symbolic execution path** stands for ***many*** actual **program runs**
  - In fact, exactly the set of runs whose concrete values satisfy the path condition
- Thus, we can **cover a lot more of the program's execution space than testing**
- Viewed **as a static analysis, symbolic execution** is
  - **Complete**, but not sound (usually doesn't terminate)
  - **Path, flow, and context sensitive**

A Little History

The idea is an old one

- Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. **SELECT–a formal system for testing and debugging programs by symbolic execution**. In ICRS, pages 234–245, **1975**.

- James C. King. **Symbolic execution and program testing**. CACM, 19(7):385–394, **1976**. **(most cited)**

- Leon J. Osterweil and Lloyd D. Fosdick. **Program testing techniques using simulated execution**. In ANSS, pages 171–177, **1976**.

- William E. Howden. **Symbolic testing and the DISSECT symbolic evaluation system**. IEEE Transactions on Software Engineering, 3(4):266–278, **1977**.

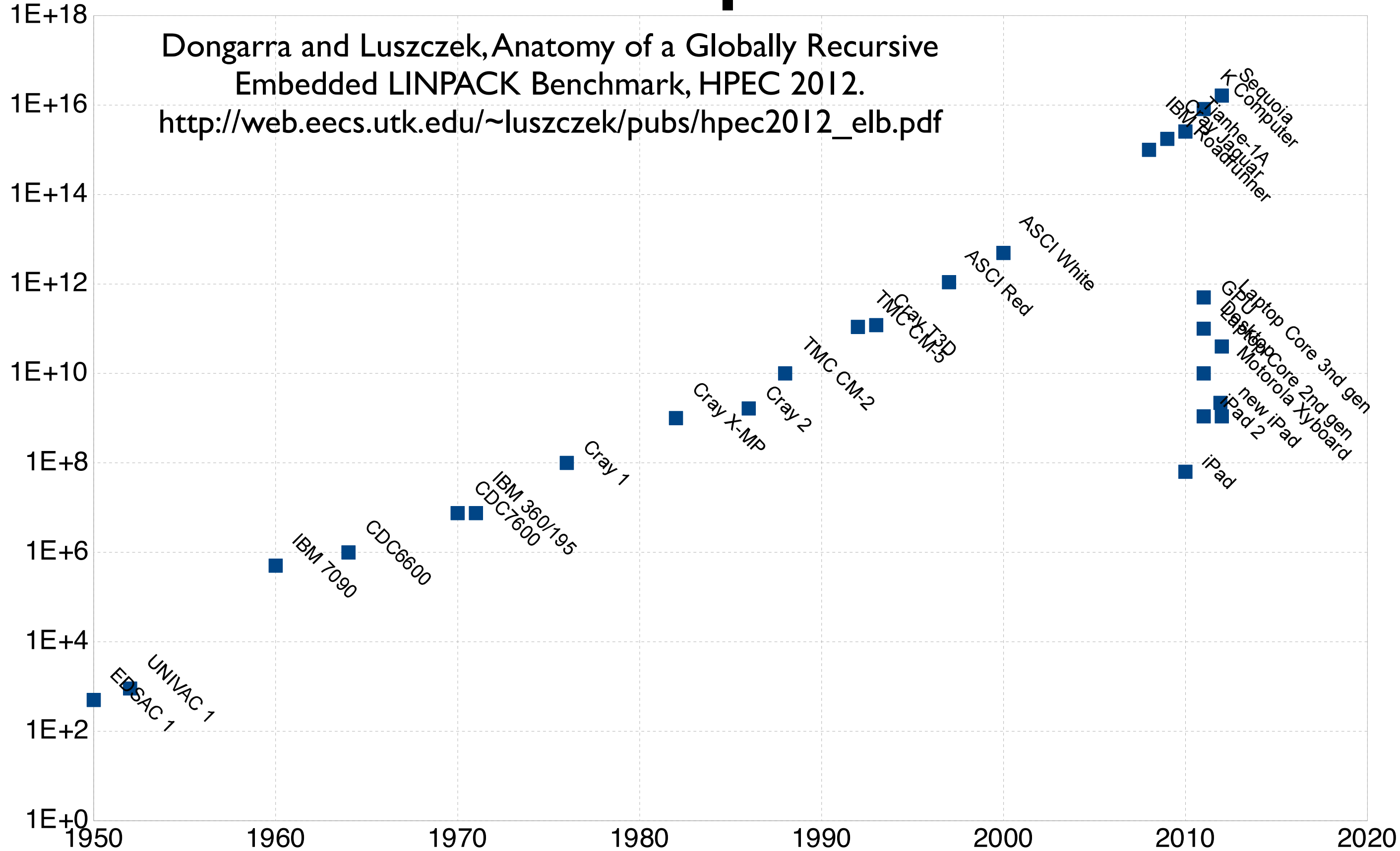
Why didn't it take off?

- **Symbolic execution can be compute-intensive**
  - Lots of possible program paths
  - Need to query solver a lot to decide which paths are feasible, which assertions could be false
  - Program state has many bits
- **Computers were slow** (not much processing power) **and small** (not much memory)
  - Recent Apple iPads are as fast as Cray-2's from the 80's

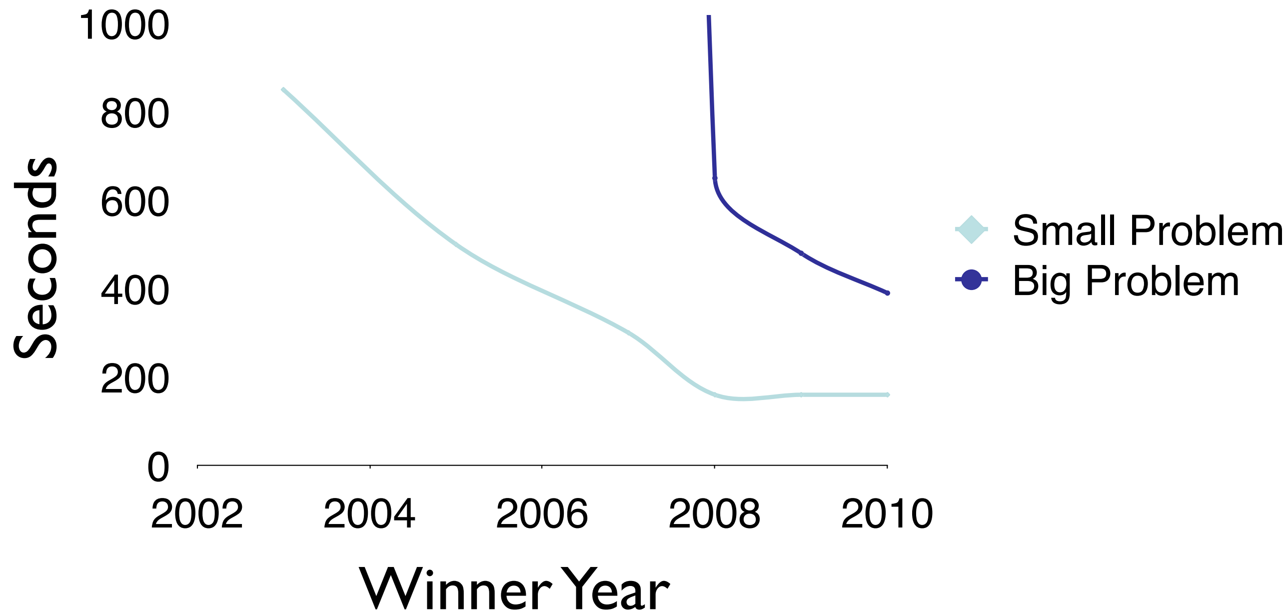
Today

- **Computers** are much **faster**, **bigger**
- **Better algorithms** too: powerful **SMT/SAT solvers**
  - SMT = *Satisfiability Modulo Theories* = SAT++
- Can solve very large instances, very quickly
  - Lets us check assertions, prune infeasible paths

Hardware improvements



SAT algorithm improvements



Results of SAT competition winners (2002-2010)
on SAT'09 problem set, on 2011 hardware

Rediscovery

- 2005-2006 reinterest in symbolic execution
- Area of success: (security) **bug finding**
  - Heuristic search through space of possible executions
  - Find really interesting bugs

Basic symbolic execution

Symbolic variables

- Extend the language's support for expressions e to include **symbolic variables,** representing *unknowns*

$$e ::= \alpha \mid n \mid X \mid e_0 + e_1 \mid e_0 \leq e_1 \mid e_0 \&\& e_1 \mid \ldots$$

  - $n \in N$ = integers, $X \in Var$ = variables, $\alpha \in SymVar$

- Symbolic variables are **introduced** when **reading input**
  - Using `mmap`, `read`, `write`, `fgets`, etc.
  - So if a bug is found, we can recover an input that reproduces the bug when the program is run normally

Symbolic expressions

- We make (or modify) a language interpreter to be able to **compute symbolically**
  - Normally, a program's variables contain *values*
  - Now they can also contain *symbolic expressions*
    - Which are expressions containing symbolic variables
- Example normal values:
  - 5, "hello"
- Example symbolic expressions:
  - α+5, "hello"+α, a[α+β+2]

Straight-line execution

```
x = read();
y = 5 + x;
z = 7 + y;
a[z] = 1;
```

Concrete Memory

x ↦ 5
y ↦ 10
z ↦ 17
a ↦ {0,0,0,0}

Overrun!

Symbolic Memory

x ↦ α
y ↦ 5+α
z ↦ 12+α
a ↦ {0,0,0,0}

Possible overrun!

We'll explain arrays shortly

Path condition

- Program control can be affected by symbolic values

```
1  x = read();
2  if (x>5) {
3    y = 6;
4    if (x<10)
5      y = 5;
6  } else y = 0;
```

- We represent the influence of symbolic values on the current path using a **path condition π**
  - Line 3 reached when $\alpha > 5$
  - Line 5 reached when $\alpha > 5$ and $\alpha < 10$
  - Line 6 reached when $\alpha \leq 5$

Path feasibility

- Whether a path is feasible is tantamount to a path condition being **satisfiable**

```
1  x = read();
2  if (x>5) {
3    y = 6;
4    if (x<3)
5      y = 5;
6  } else y = 0;
```

$\pi = \alpha > 5$

$\pi = \alpha > 5 \wedge \alpha < 3$

$\pi = \alpha \leq 5$

Not satisfiable!

- **Solution** to path constraints **can be used as inputs** to a concrete test case that will execute that path
  - Solution to reach line 3: $\alpha = 6$
  - Solution to reach line 6: $\alpha = 2$

Paths and assertions

- Assertions, like array bounds checks, are conditionals

1	`x = read();`	$\pi = \text{true}$
2	`y = 5 + x;`	$\pi = \text{true}$
3	`z = 7 + y;`	$\pi = \text{true}$
4	`if(z < 0)`	$\pi = \text{true}$
5	`  abort();`	$\pi = 12+\alpha<0$
6	`if(z >= 4);`	$\pi = \neg(12+\alpha<0)$
7	`  abort();`	$\pi = \neg(12+\alpha<0) \wedge 12+\alpha\geq 4$
8	`a[z] = 1;`	$\pi = \neg(12+\alpha<0) \wedge \neg(12+\alpha\geq 4)$

- So, if either lines 5 or lines 7 are reachable (i.e., the paths reaching them are feasible), we have found an out-of-bounds access

Forking execution

- Symbolic executors can **fork** at branching points
  - Happens when there are **solutions** to both the **path condition *and* its negation**
- How to systematically explore both directions?
  - **Check feasibility during execution** and queue feasible path (condition)s for later consideration
  - **Concolic execution**: run the program (concretely) to completion, then generate new input by changing the path condition

Execution algorithm

1. Create initial task
   - $pc = 0$, $\pi = \varnothing$, $\sigma = \varnothing$
2. Add task (pc, π, σ) onto worklist
3. While (list is not empty)
   3a. pull some task (pc, π, σ) from worklist
   3b. execute. if it potentially forks at (pc_0, π_0, σ_0)
   3ba. add task $(pc_1, (\pi_0 \wedge p), \sigma_0)$ if $\pi_0 \wedge p$ feasible
   3bb. add task $(pc_2, (\pi_0 \wedge \neg p), \sigma_0)$ if $\pi_0 \wedge \neg p$ feasible

```
pc0  if (p) {
pc1     …
pc2 } else { …
```

Note: Libraries, native code

- At some point, symbolic execution will reach the “edges” of the application
  - Library, system, or assembly code calls
- In some cases, could pull in that code also
  - E.g., pull in libc and symbolically execute it
  - But glibc is insanely complicated
    - Symbolic execution can easily get stuck in it
  - So, pull in a simpler version of libc, e.g., newlib
- In other cases, need to make models of code
  - E.g., implement ramdisk to model kernel fs code

Concolic execution

- Also called *dynamic symbolic execution*
- **Instrument the program** to do symbolic execution as the program runs
  - Shadow concrete program state with symbolic variables
    - Initial concrete state determines initial path
      - could be randomly generated
  - **Keep shadow path condition**
- **Explore one path at a time**, start to finish
  - The next path can be determined by
    - negating some element of the last path condition, and
    - solving for it, to produce concrete inputs for the next test
  - Always have a concrete underlying value to rely on

Concretization

- Concolic execution makes it really easy to **concretize**
  - Replace symbolic variables with concrete values that satisfy the path condition
    - Always have these around in concolic execution
- So, could **actually do system calls**
  - But we lose symbolic-ness at such calls
- And can **handle cases** when conditions **too complex for SMT solver**