Introducing **Symbolic Execution**

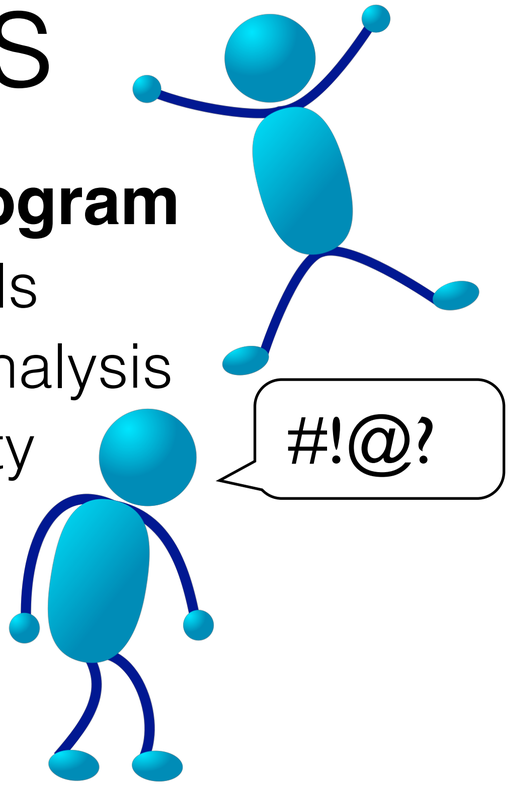
Slide deck courtesy of Prof. Michael Hicks, University of Maryland, College Park (UMD)

Software has **bugs**

- To **find them**, we use **testing** and **code reviews**

- But some **bugs are** still **missed**
  - *Rare features*
  - *Rare circumstances*
  - *Nondeterminism*

Static analysis



- Can **analyze all possible runs of a program**
  - An explosion of interesting ideas and tools
  - Commercial companies sell, use static analysis
  - Great potential to improve software quality

- But: **Can it find deep, difficult bugs?**
  - Our experience: yes, but **not often**
  - Commercial viability implies you must deal with developer confusion, false positives, error management,..
  - This means that companies specifically aim to keep the **false positive rate down**
    - They often do this **by purposely missing bugs**, to keep the analysis simpler

One issue: Abstraction

- Abstraction lets us model all possible runs
  - But **abstraction introduces conservatism**

- ***-sensitivities add precision**, to deal with this
  - * = **flow**-, **context**-, **path**-, *etc.*
  - But **more precise abstractions are more expensive**
    - Challenges scalability
    - Still have false alarms or missed bugs

- **Static analysis abstraction ≠ developer abstraction**
  - Because the developer didn't have them in mind

Symbolic execution

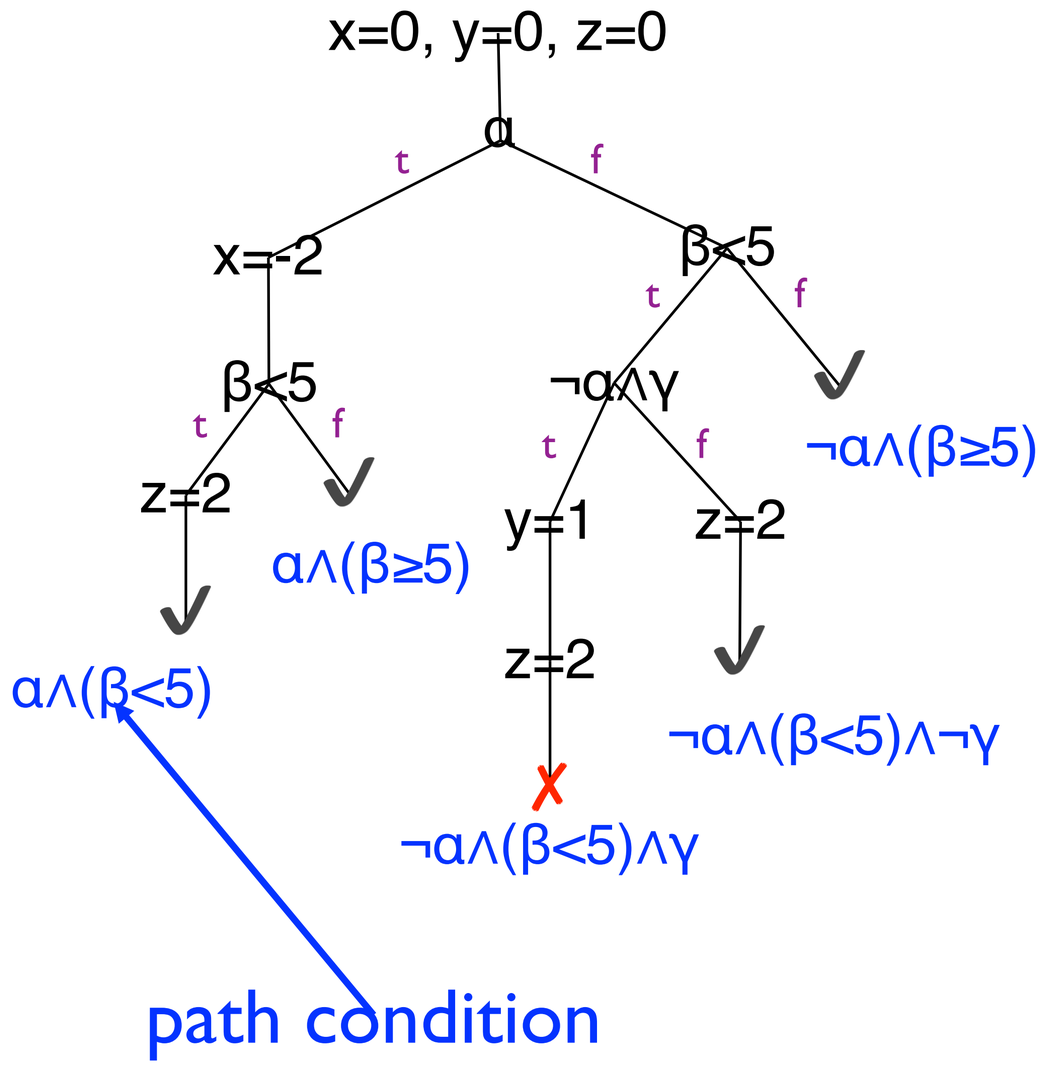
A middle ground

- 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

```
1.  int a = α, b = β, c = γ;
2.                        // symbolic
3.  int x = 0, y = 0, z = 0;
4.  if (a) {
5.    x = -2;
6.  }
7.  if (b < 5) {
8.    if (!a && c)  { y = 1; }
9.    z = 2;
10. }
11. 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

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;
```

π = α>5

π = α>5 ∧ α<3

π = α≤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: α = 6
  - Solution to reach line 6: α = 2

Paths and assertions

- Assertions, like array bounds checks, are conditionals

```
1 x = read();
2 y = 5 + x;
3 z = 7 + y;
4 if(z < 0)
5    abort();
6 if(z >= 4);
7    abort();
8 a[z] = 1;
```

π = **true**
π = **true**
π = **true**
π = **true**
π = $l2+\alpha<0$
π = **¬(**$l2+\alpha<0$)
π = **¬(**$l2+\alpha<0$) ∧ $l2+\alpha$**≥4**
π = **¬(**$l2+\alpha<0$) ∧ ¬($l2+\alpha$**≥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, $\pi = \emptyset$, $\sigma = \emptyset$
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, ($\pi_0 \wedge p$), σ_0) if $\pi_0 \wedge p$ feasible
   3bb. add task (pc_2, ($\pi_0 \wedge \neg p$), σ_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**

Symbolic execution as search, and the rise of solvers

Search and SMT

- Symbolic execution is appealingly **simple and useful**, but **computationally expensive**
- We will see how the effective use of symbolic execution **boils down to a kind of search**
- And also take a moment to see how its feasibility at all has been aided by **the rise of SMT solvers**

Path explosion

- Usually can't run symbolic execution to exhaustion
  - Exponential in branching structure

```
1. int a = α, b = β, c = γ;    // symbolic
2. if (a) ... else ...;
3. if (b) ... else ...;
4. if (c) ... else ...;
```

    - Ex: 3 variables, 8 program paths
  - Loops on symbolic variables even worse

```
1. int a = α;    // symbolic
2. while (a) do ...;
3. …
```

    - Potentially 2^31 paths through loop!

Compared to static analysis

- Stepping back: Here is a **benefit of static analysis**
  - Static analysis will actually **terminate** even when considering **all possible program runs**
- It does this by approximating multiple loop executions, or branch conditions
  - Essentially **assumes all branches**, and **any number of loop iterations**, are **feasible**
- But can **lead to false alarms**, of course

Basic (symbolic) search

- Simplest ideas: algorithms 101
  - Depth-first search (*DFS*) — **worklist = stack**
  - Breadth-first search (*BFS*) — **worklist = queue**

- Potential drawbacks
  - **Not guided** by any higher-level knowledge
    - Probably a bad sign
  - **DFS could easily get stuck** in one part of the program
    - E.g., it could keep going around a loop over and over again
  - Of these two, BFS is a better choice
    - But more intrusive to implement (can't easily be concolic)

Search strategies

- Need to **prioritize search**
  - Try to steer search towards paths more likely to contain assertion failures
  - Only run for a certain length of time
    - So if we don't find a bug/vulnerability within time budget, too bad
- Think of **program execution as a DAG**
  - Nodes = program states
  - Edge(n_1,n_2) = can transition from state n_1 to state n_2
- We need a kind of **graph exploration algorithm**
  - At each step, pick among all possible paths

Randomness

- We don't know *a priori* which paths to take, so adding some randomness seems like a good idea
  - Idea 1: **pick next path to explore uniformly at random** (*Random Path*, or RP)
  - Idea 2: **randomly restart search** if haven't hit anything interesting in a while
  - Idea 3: **choose among equal priority paths at random**
    - All of these are good ideas, and randomness is very effective
- One drawback of **randomness**: reproducibility
  - Probably good to use pseudo-randomness based on seed, and then record which seed is picked
    - Or bugs may disappear (or reappear) on later runs

Coverage-guided heuristics

- **Idea**: Try to **visit statements we haven't seen before**
- Approach
  - Score of statement = # times it's been seen
  - Pick next statement to explore that has lowest score
- Why might this work?
  - Errors are often in hard-to-reach parts of the program
  - This strategy tries to reach everywhere.
- Why might this *not* work?
  - Maybe never be able to get to a statement if proper precondition not set up

Generational search

- Hybrid of **BFS and coverage-guided**
  - *Generation 0*: pick one program at random, run to completion
  - *Generation 1*: take paths from *gen 0;* negate one branch condition on a path to yield a new path prefix; find a solution for that prefix; then take the resulting path
    - Semi-randomly assigns to any variables not constrained by the prefix
  - *Generation n*: similar, but branching off *gen n-1*
- Also uses a coverage heuristic to pick priority

Combined search

- Run **multiple searches at the same time**
  - Alternate between them; e.g., Fitnext
- Idea: no one-size-fits-all solution
  - Depends on conditions needed to exhibit bug
  - So will be as good as “best” solution, within a constant factor for wasting time with other algorithms
  - Could potentially use different algorithms to reach different parts of the program

SMT solver performance

- SAT solvers are at core of SMT solvers
  - In theory, could reduce all SMT queries to SAT queries
  - In practice, SMT-level optimizations are critical
- Some example extensions/improvements
  - Simple identities (x + 0 = x, x * 0 = 0)
  - Theory of arrays (read(x, write(42, x, A)) = 42)
    - 42 = array index, A = array, x = element
  - Caching (memoize solver queries)
  - Remove useless variables
    - E.g., if trying to show path feasible, only the part of the path condition related to variables in guard are important

Popular SMT solvers

- **Z3** - developed at Microsoft Research
  - http://z3.codeplex.com/
- **Yices** - developed at SRI
  - http://yices.csl.sri.com/
- **STP** - developed by Vijay Ganesh, now @ Waterloo
  - https://sites.google.com/site/stpfastprover/
- **CVC3** - developed primarily at NYU
  - http://www.cs.nyu.edu/acsys/cvc3/

But: Path-based search limited

```
int counter = 0, values = 0;
for (i = 0; i<100; i++) {
  if (input[i] == 'B') {
    counter++;
    values += 2;
  }
}
assert(counter != 75);
```

- This program has 2^{100} possible execution paths.
- Hard to find the bug:
  - $\binom{100}{75} \approx 2^{78}$ paths reach buggy line of code
  - *Pr(finding bug)* = $2^{78} / 2^{100} = 2^{-22}$

Symbolic execution systems

Resurgence

- Two key systems that triggered revival of this topic:
- **DART** — Godefroid and Sen, PLDI 2005
  - Godefroid = model checking, formal systems background
- **EXE** — Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006
  - Ganesh and Dill = SMT solver called STP (used in implementation), Cadar and Engler = systems
- Now on to next-generation systems

SAGE

- **Concolic executor** developed at **Microsoft Research**
  - Grew out of Godefroid's work on DART
  - Uses generational search
- Primarily **targets bugs in file parsers**
  - E.g., JPEG, DOCX, PPT, etc
  - Good fit for concolic execution
    - Likely to terminate
    - Just input/output behavior

SAGE Impact

- **Used on production software at MS**. Since 2007:
  - 500+ machine years (in largest fuzzing lab in the world)
    - Large cluster of machines continually running SAGE
  - 3.4 Billion+ constraints (largest SMT solver usage ever!)
  - 100s of apps, 100s of bugs (missed by everything else…)
    - Ex: *1/3 of all Win7 WEX security bugs found by SAGE*
  - Bug fixes shipped quietly to 1 Billion+ PCs
  - Millions of dollars saved (for Microsoft and the world)
  - SAGE is now used daily in Windows, Office, etc.

http://research.microsoft.com/en-us/um/people/pg/public_psfiles/SAGE-in-1slide-for-PLDI2013.pdf

KLEE

- **Symbolically executes LLVM bitcode**
  - LLVM compiles source file to .bc file
  - KLEE runs the .bc file
  - Grew out of work on EXE
- Works in the style of our basic symbolic executor
  - Uses `fork()` to manage multiple states
  - Employs a variety of search strategies
    - Primarily **random path + coverage-guided**
  - Mocks up the environment to deal with system calls, file accesses, etc.
- **Freely available with LLVM distribution**

KLEE: Coverage for Coreutils

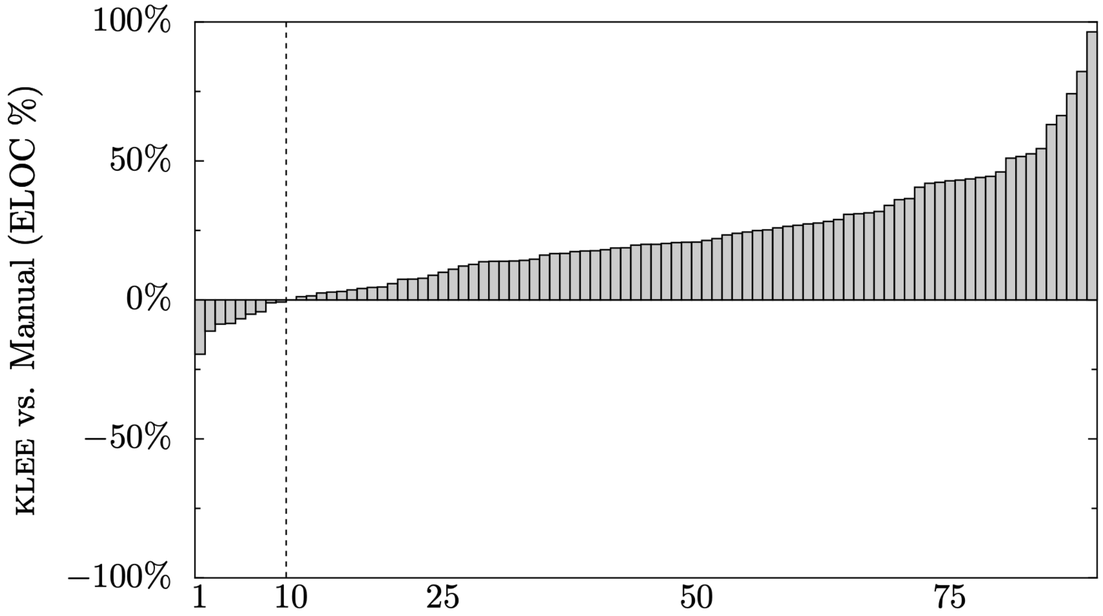


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.

Cadar, Dunbar, and Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008

KLEE: Coreutils crashes

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

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

Cadar, Dunbar, and Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008

Mayhem

- Developed at CMU (Brumley et al), **runs on binaries**
- Uses BFS-style search and native execution
  - **Combines best of symbolic and concolic strategies**
- **Automatically generates exploits** when bugs found

Mergepoint

- Extends Mayhem with a technique called **veritesting**
  - ***Combines* symbolic execution** with **static analysis**
  - Use static analysis for complete code blocks
  - Use symbolic execution for hard-to-analyze parts
    - Loops (how many times will it run?), complex pointer arithmetic, system calls
- Better **balance** of time **between solver and executor**
  - **Finds bugs faster**
  - **Covers more of the program** in the same time
- Found 11,687 bugs in 4,379 distinct applications in a Linux distribution
  - Including new bugs in highly tested code

Other symbolic executors

- **Cloud9** — Parallel, multi-threaded symbolic execution
  - Extends KLEE (available)
- **jCUTE**, **Java PathFinder** — symbolic execution for Java (available)
- **Bitblaze** — Binary analysis framework (available)
- **Otter** — directed symbolic execution for C (available)
  - Give the tool a line number, and it try to generate a test case to get there
- **Pex** — symbolic execution for .NET

Summary

- **Symbolic execution generalizes testing**
  - Uses static analysis to direct generation of tests that cover different program paths
- Used in practice to find **security-critical bugs** in **production code**
  - SAGE at Microsoft
  - Mergepoint for Linux
- **Many tools freely available**