Symbolic Execution

for finding bugs

Michael Hicks

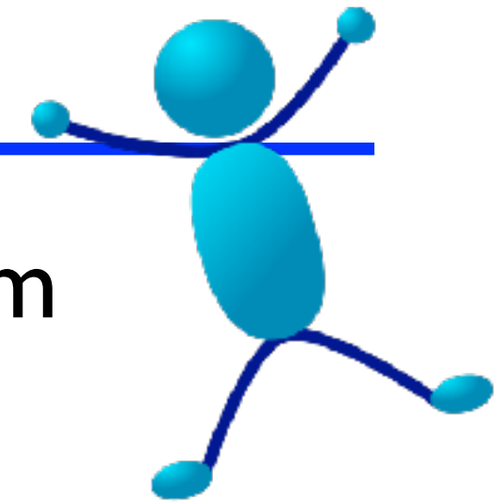
CMSC 631, Fall 2017

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
  - Lots of interesting ideas and tools
  - Commercial companies sell, use static analysis
  - It all looks good on paper, and in papers

- But can developers use it?
  - Our experience: Not easily
  - Results in papers describe use by static analysis experts
  - Commercial viability implies you must deal with developer confusion, false positives, error management,..

One Issue: Abstraction

- Abstraction lets us scale and model all possible runs
  - But it also introduces conservatism
  - *-sensitivities attempt to deal with this
    - * = flow-, context-, path-, field-, etc
  - But they are never enough

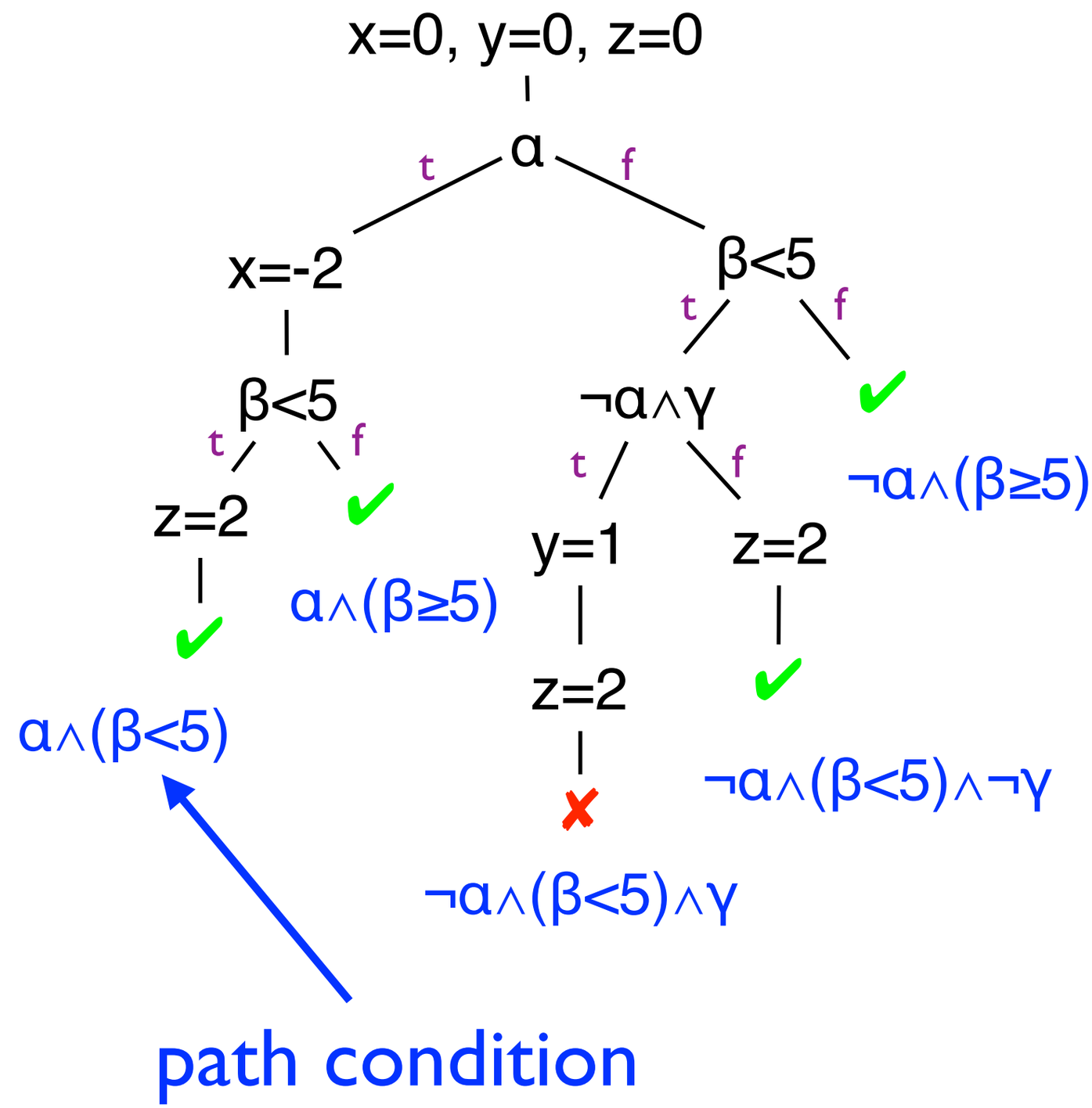
- Static analysis abstraction ≠ developer abstraction
  - Because the developer didn't have them in mind

Symbolic execution: a middle ground

- Testing works
  - But, each test only explores one possible execution
    - assert(f(3) == 5)
  - We *hope* test cases generalize, but no guarantees
- Symbolic execution generalizes testing
  - Allows *unknown* symbolic variables 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* actually 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

Early work on symbolic execution

- 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.

The problem

- Computers were small (not much memory) and slow (not much processing power)
  - Apple's iPad 2 is as fast as a Cray-2 from the 1980's
- Symbolic execution can be extremely expensive
  - 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

Today

- Computers are much faster, memory is cheap
- There are very powerful SMT/SAT solvers today
  - SMT = Satisfiability Modulo Theories = SAT++
  - Can solve very large instances, very quickly
    - Lets us check assertions, prune infeasible paths
  - We've used Z3 (Microsoft), STP (Stanford), Yices (SRI)
- Recent success: bug finding
  - Heuristic search through space of possible executions
  - Find really interesting bugs

Remainder of the tutorial

- The basics, in code
- Scaling up
  - The search space
  - Hard-to-handle features
- Existing tools
  - KLEE: one industrial grade tool
- KLEE lab: using KLEE to find bugs
  - Including vulnerabilities

Symbolic Execution for IMP

a ::= n | X | -a | a0+a1 | a0-a1 | a0×a1 | a0/a1

b ::= bv | ¬b | b0∧b1 | b0∨b1 | a0=a1 | a0<a1 | a0>a1

c ::= skip | input(s) | X:=a | if b then c else c

| c0; c1 | while b do c | assert b

- $n \in N$ = integers, $X \in Var$ = variables, $bv \in Bool = \{true, false\}$
- Syntax stratified into commands (c) and expressions (a,b)
  - Expressions have no side effects
- No functions, no unstructured control flow (goto), no heap (or stack), no arrays/data structures
  - But these can all be handled
- See ast.ml

Interpretation for IMP

- See concrete.ml
  - States are maps from strings to ints
  - `Impl.run` is the entry point: returns a state

- How to extend this to be a symbolic executor?
  - See symbol.ml and symbolic.ml

Symbolic Expressions

- May have variables with no particular assignment
  - Arise from `input(s)` expressions

```
type arith =
  | AEVar of string
  | AENum of int
  | AENegate of arith
  | AEPlus of arith * arith
  | AEMinus of arith * arith
  | AEMult of arith * arith
  | AEDiv of arith * arith
```

```
type int_t =
  | LVar of string
  | LNum of int
  | LNegate of int_t
  | LPlus of int_t * int_t
  | LMinus of int_t * int_t
  | LMult of int_t * int_t
  | LDiv of int_t * int_t
```

  - Likewise have `type t` for symbolic boolean expressions

Symbolic States

- Concrete states are maps from variables to ints

```
type conc_state = int StringMap.t
```

- Symbolic states are maps from variables to *symbolic expressions*, and a *path condition*

```
type sym_state = (Symbol.int_t StringMap.t) * Symbol.t
```

  - might result in x being mapped to expressions like α+5, where α is a symbolic variable

Forking Execution

- How to decide which branches are *feasible*?
  - Combine path condition with branch cond and ask solver!

```
| SIf (b, s1, s2) ->
 let l = t_of_boolean b env in    (* branch cond *)
 let cond_true = LAnd (l, pc) in (* … and path cond *)
 let cond_false = LAnd ((LNot l), pc) in
 let sat_true = check (z3_of_t cond_true) in
 let sat_false = check (z3_of_t cond_false) in
(match sat_true with        (* might do both branches *)
 | Some _ -> …(eval s1 (env, cond_true))
 | None   -> …);
 (match sat_false with
 | Some _ -> …(eval s2 (env, cond_false))
 | None   -> …);
```

Top-level Evaluation Strategy

```
1. create initial state
   - pc = 0, path cond = true, state = empty
2. evaluate each statement symbolically
3. whenever execution forks, evaluate both sides (depth first)
4. when done, may return multiple symbolic states
```

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!

Basic search

- Simplest ideas: algorithms 101
  - Depth-first search (DFS)
  - Breadth-first search (BFS)

- Potential drawbacks
  - Neither is 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

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(n1,n2) = can transition from state n1 to state n2
- Then we need some kind of graph exploration strategy
  - 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, RP)
  - Idea 2: randomly restart search if haven't hit anything interesting in a while
  - Idea 3: when have equal priority paths to explore, choose next one at random
    - All of these are good ideas, and randomness is very effective
- One drawback: reproducibility
  - Probably good to use psuedo-randomness based on seed, and then record which seed is picked
  - (More important for symbolic execution implementers than users)

Coverage-guided heuristics

- Idea: Try to visit statements we haven't seen before
- Approach
  - Score of statement = # times it's been seen and how often
  - 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
- KLEE = RP + coverage-guided

Generational search

- Hybrid of BFS and coverage-guided
- Generation 0: pick one path 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 path prefix, and then take the resulting path
  - Note will semi-randomly assign to any variables not constrained by the path 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, which 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 and higher-level optimizations are critical

- Some examples
  - 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

Libraries and 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
  - ⇒ pull in a simpler version of libc, e.g., newlib
    - libc versions for embedded systems tend to be simpler
- In other cases, need to make models of code
  - E.g., implement ramdisk to model kernel fs code
  - This is a lot of work!

Concolic execution

- Also called *dynamic symbolic execution*
- Instrument the program to do symbolic execution as the program runs
  - I.e., shadow concrete program state with symbolic variables
- Explore one path at a time, start to finish
  - Always have a concrete underlying value to rely on
  - Basis for generational search, mentioned before

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
  - But can do the same in pure symbolic system

Resurgence of symbolic exection

- 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)
      - Theory of arrays
    - Cadar and Engler = systems
    - Won CCS 2016 Most Influential Paper award

Recent successes, run on binaries

- SAGE
  - Microsoft (Godefroid) concolic executor
  - Symbolic execution to find bugs in file parsers
    - E.g., JPEG, DOCX, PPT, etc
  - Now a Microsoft service (Springfield)
- Mayhem (CMU), Angr (UCSB & NEU), Triton
- Java Symbolic PathFinder — runs on JVM byte code

KLEE

- Symbolically executes LLVM bitcode
  - LLVM compiles source file to .bc file
  - KLEE runs the .bc file
- Works in the style of our example interpreter
  - Uses fork() to manage multiple states
  - Employs a variety of search strategies
  - Mocks up the environment to deal with system calls, file accesses, etc.

KLEE: Coverage for Coreutils

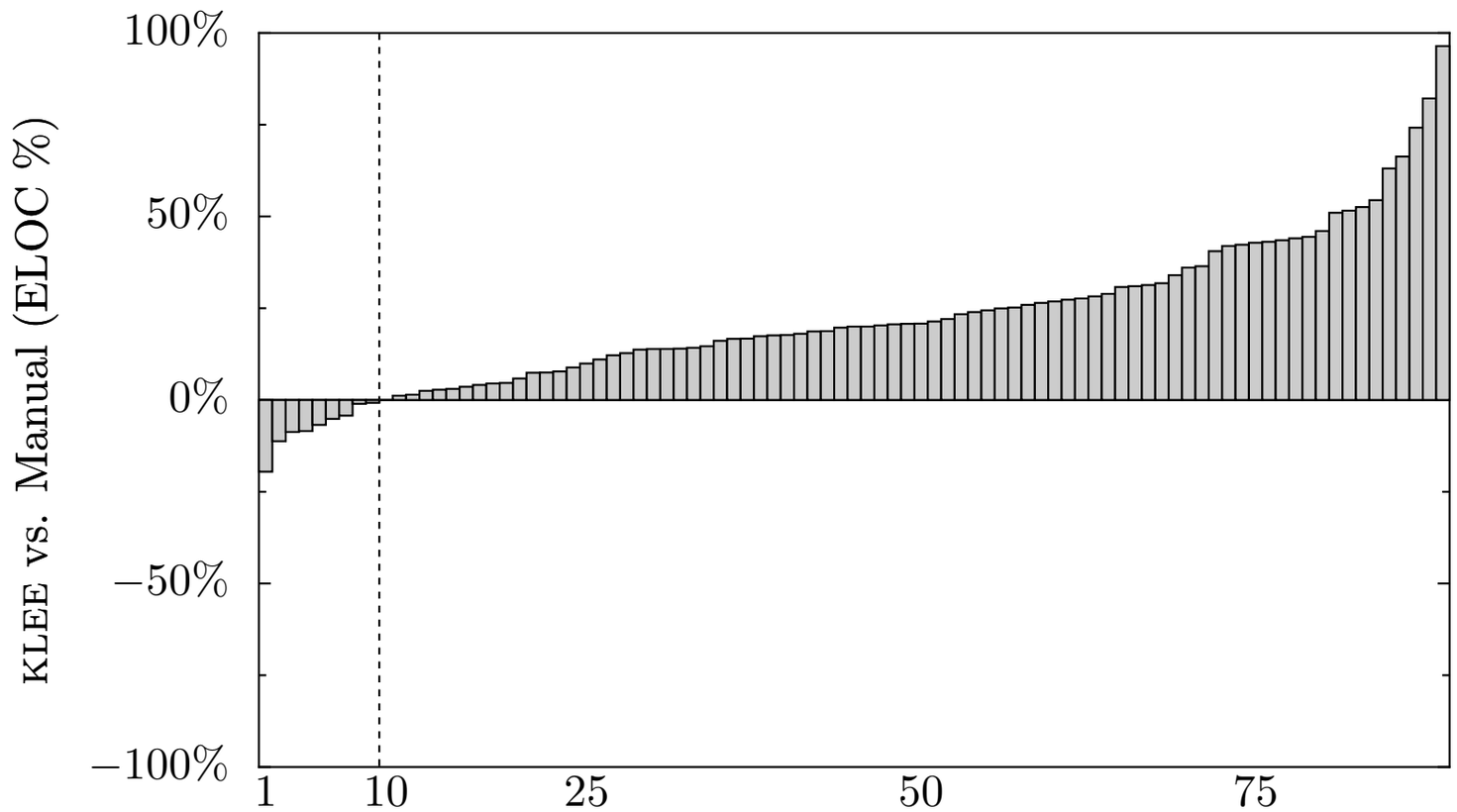


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE,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

Demo

- Now we will try out KLEE
  - I'm using the Docker image; much easier to install!
- We will try the "maze tutorial"
  - https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/