Symbolic/Concolic Execution

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: Prof. Michael Hicks of PSU and Prof. Wei Le from iastate

Overview

- Static vs dynamic analysis
- Control flow analysis
  - CFG
    - Basic block
    - Dominator
  - CG (ICFG)
    - Context sensitivity

Symbolic/Concolic Execution

Software has bug

- 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
- But can developers use it?
  - Results in papers describe use by static analysis experts
  - Commercial viability implies you must deal with developer confusion, false positives, error management,..

Static analysis

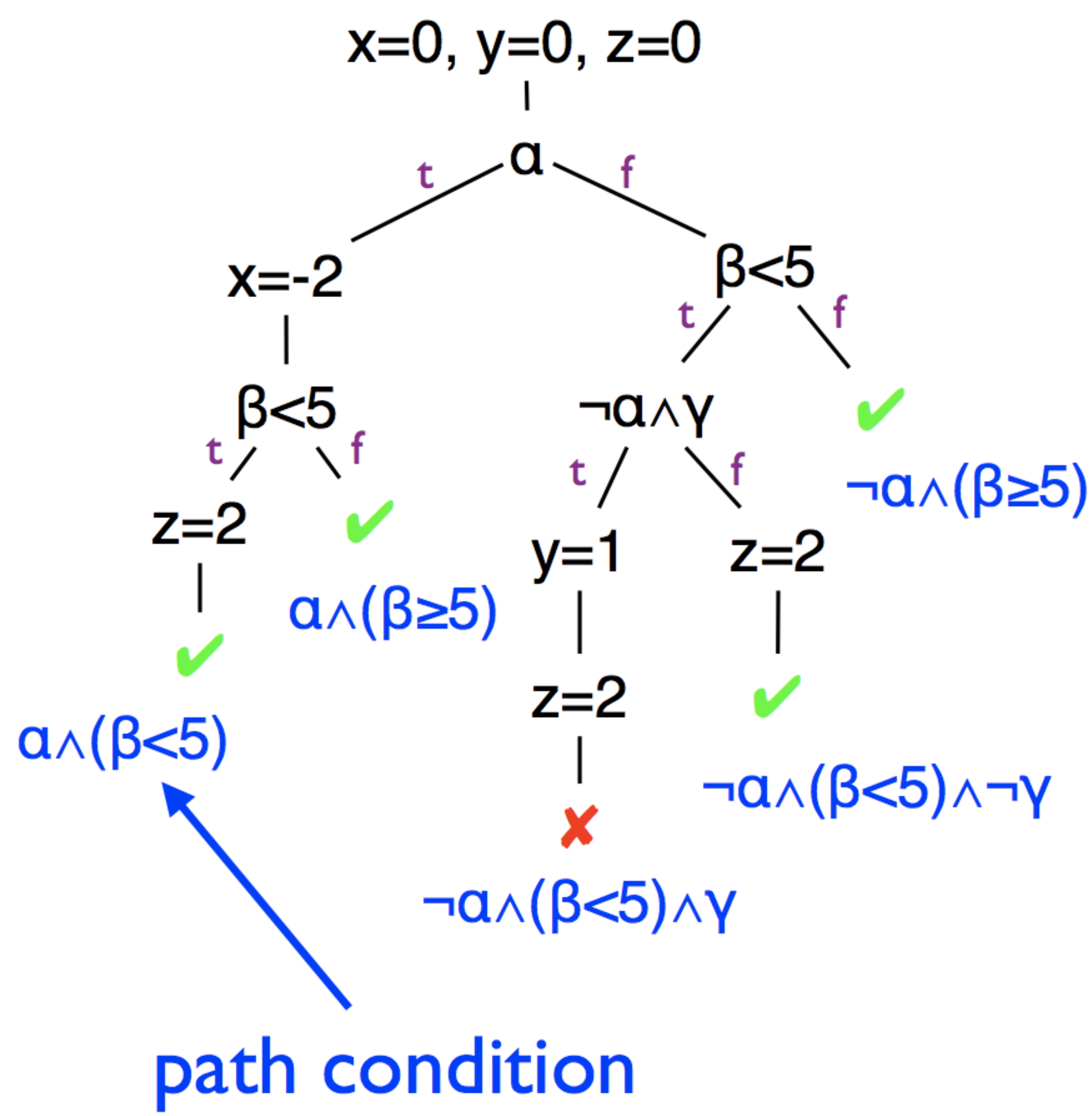
- Abstraction lets us scale and model all possible runs
  - But it also introduces conservatism
  - *-sensitivities attempt to deal with this - * = flow-, context-, path-, field-, etc
  - But they are never enough
- Static analysis abstraction ≠ developer abstraction

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

```
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 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

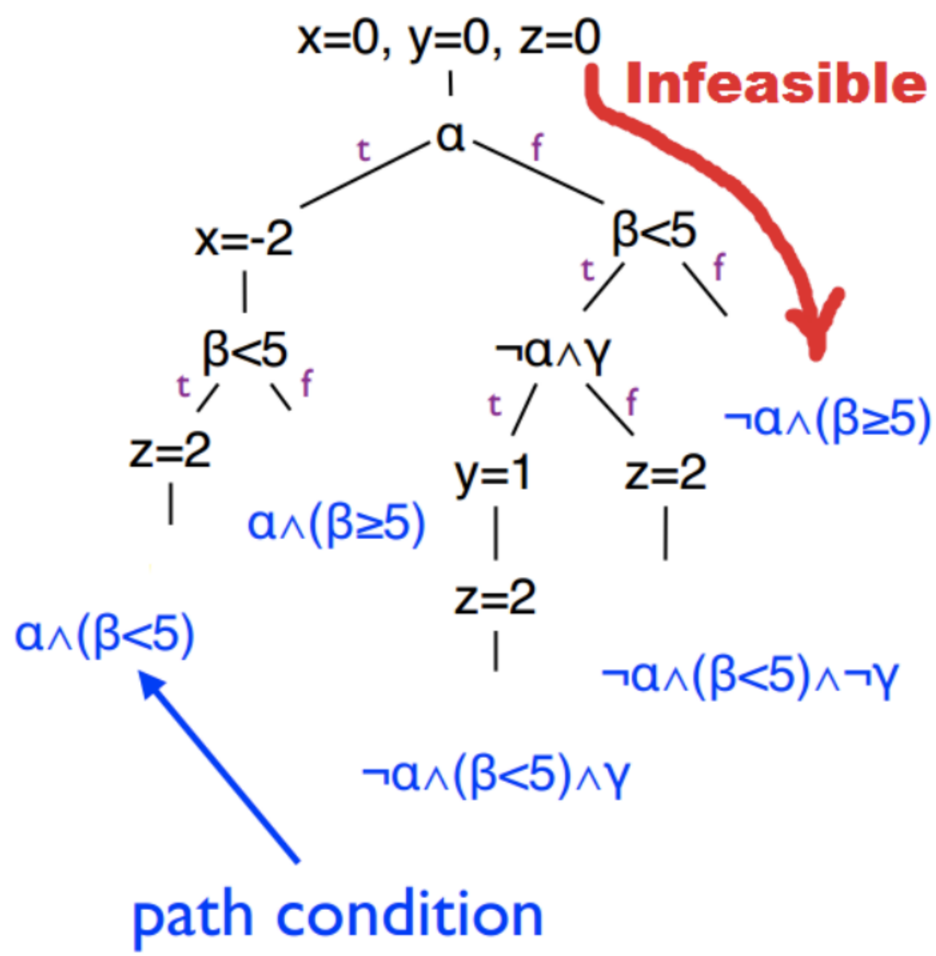
Applications of Symbolic Execution

- General goal: identifying semantics of programs
- Basic applications:
  - Detecting infeasible paths
  - Generating test inputs
  - Finding bugs and vulnerabilities
- Advanced applications
  - Generating program invariants
  - Repair programs

Detecting Infeasible Paths

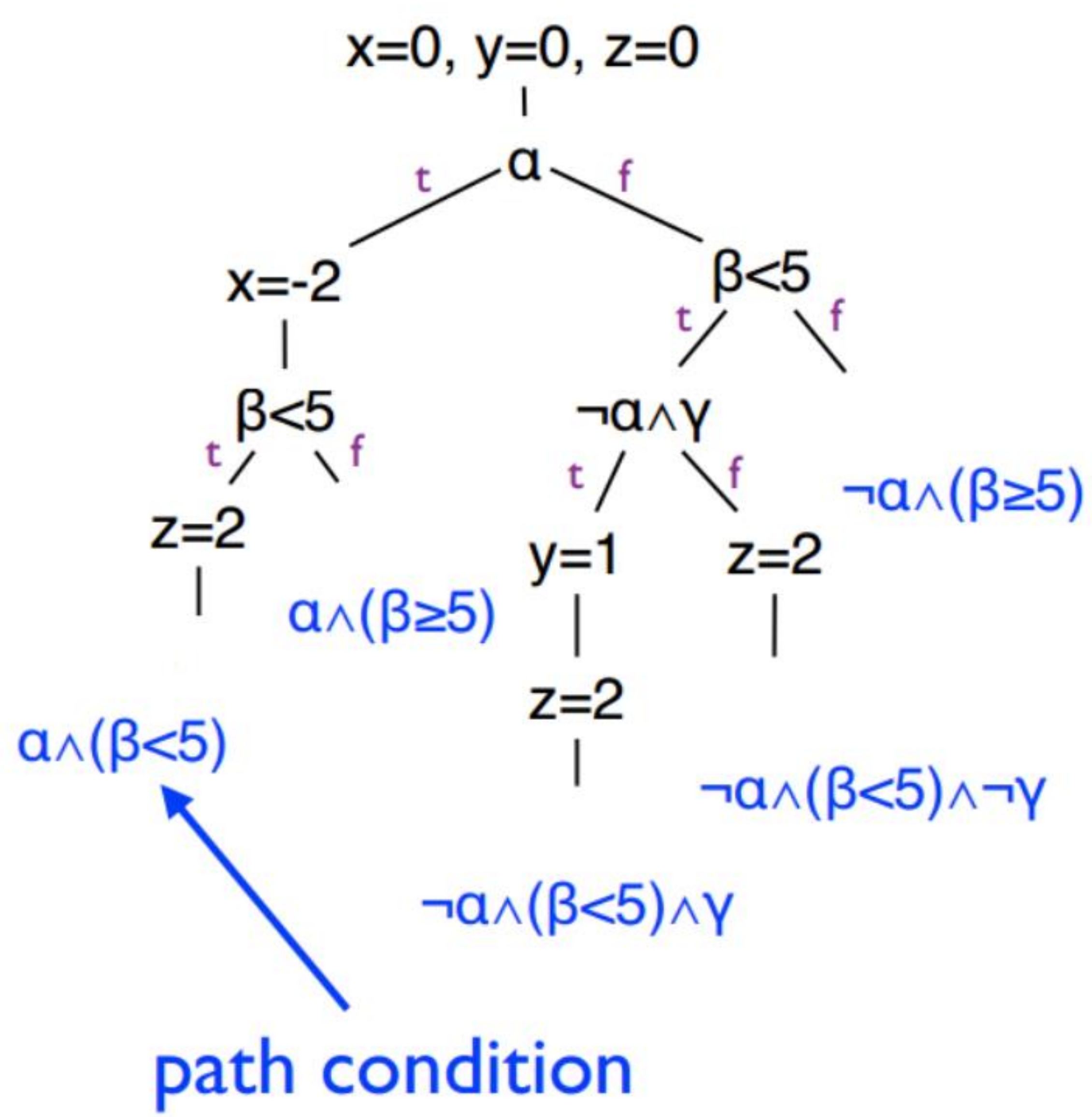
Suppose we require $\alpha = \beta$

```
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)
```



Test Input Generation

```
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)
```



Path 1: $\alpha = 1, \beta = 1$
Path 2: $\alpha = 1, \beta = 6$
Path 3 ...

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
- Recent success: bug finding
  - Heuristic search through space of possible executions
  - Find really interesting bugs

Resurgence of Symbolic Execution

The block issues in the past:

- Not scalable: program state has many bits, there are many program paths
- Not able to go through loops and library calls
- Constraint solver is slow and not capable to handle advanced constraints

The two key projects that enable the advance:

- DART Godefroid and Sen, PLDI 2005 (introduce dynamic information to symbolic execution)
- EXE Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006 (STP: a powerful constraint solver that handles *array*)

Moving forward:

- More powerful computers and clusters
- Techniques of mixture concrete and symbolic executions
- Powerful constraint solvers

Today: Two Important Tools

- KLEE [2008:OSDI:Cadar]
  - Open source symbolic executor
  - Runs on top of LLVM
  - Has found lots of problems in open-source software
- SAGE [PLDI:Godefroid:2008]
  - Microsoft internal tool
  - Symbolic execution to find bugs in file parsers - E.g., JPEG, DOCX, PPT, etc
  - Cluster of n machines continually running SAGE

Other Symbolic Executors

- Cloud9: parallel symbolic execution, also supports threads
- Pex, Code Hunt: Microsoft tools, symbolic execution for .NET
- Cute: concolic testing
- jCUTE: symbolic execution for Java
- Java PathFinder: NASA tools, a model checker that also supports symbolic execution
- SymDroid: symbolic execution on Dalvik Bytecode
- Kleenet: testing interaction protocols for sensor network

Three Challenges

- Path explosion
- Modeling program statements
- Constraint solving

Challenge 1: Path Explosion

- Exponential in branching structure

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

  - Ex: 3 variables, 8 program paths

- Loops on symbolic variables even worse

```
int a = α;    // symbolic
while (a) do ...;

```

  - Potentially 2^31 paths through loop!

Search Strategies: Naive Approach

- DFS (depth first search), BFS (breadth first search)
  - The two approaches purely are based on the structure of the code
  - You cannot enumerate all the paths
  - DFS: search can stuck at somewhere in a loop
  - BFS: very slow to determine properties for a path if there are many branches

Search Strategies: Random Search

- How to perform a random search?
  - Idea 1: pick next path to explore uniformly at random
  - 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
- Drawback: reproducibility, probably good to use psuedo-randomness based on seed, and then record which seed is picked

Search Strategies: Coverage Guided Search

- Goal: Try to visit statements we haven't seen before
- Approach:
  - Select paths likely to hit the new statements
    - Favor paths on recently covering new statements
    - Score of statement = # times its been seen and how often; Pick next statement to explore that has lowest score
- Good: Errors are often in hard-to-reach parts of the program, this strategy tries to reach everywhere.
- Bad: Maybe never be able to get to a statement

Search Strategies: Generational Search

- Hybrid of BFS and coverage-guided search
- 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
- …
- Generation n: similar, but branching off gen n-1 (also uses a coverage heuristic to pick priority)

Challenge 2: Complex Code and Environment Dependencies

- 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
  - In other cases, need to make models of code
    - E.g., implement ramdisk to model kernel fs code
    - This is a lot of work!

An Example

```
int fd  = open("t.txt", O_RDONLY);
```

- If all arguments are concrete, forward to OS

```
int fd  = open(sym_str, O_RDONLY);
```

- Otherwise, provide *models* that can handle symbolic files
  - Goal is to explore all possible *legal* interactions with the environment

Solutions: Concretization

- 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

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

Challenge 3: Constraint Solving - SAT

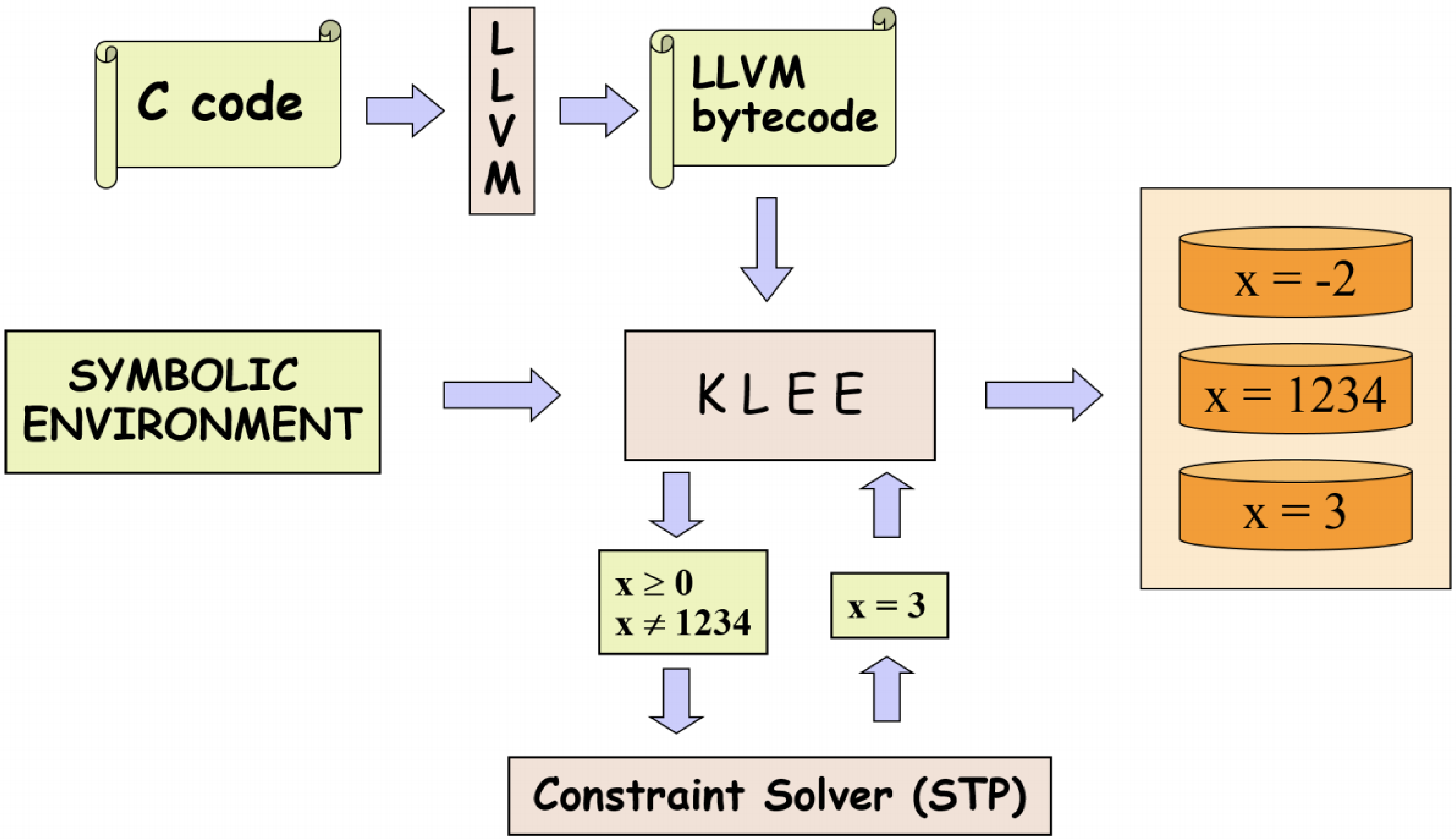
SAT: find an assignment to a set of Boolean variables that makes the Boolean formula true

Complexity: NP-Complete

Constraint Solving - SMT

- The State of the Art: Handle linear integer constraints
- Challenges
  - Constraints that contain non-linear operands, e.g., sin(), cos()
  - Float-point constraints: no theory support yet, convert to bit-vector computation
  - String constraints: a = b.replace('x', 'y')
  - Quantifies: ∃, ∀ | Disjunction

Internal of Symbolic Executors: KLEE



Tool Design KLEE - Path Explosion

- Random, coverage-optimize search
- Compute state weight using
  - Minimum distance to an uncovered instruction
  - Call stack of the state
  - Whether the state recently covered new code
- Timeout: one hour per utility when experimenting with coreutils

Tool Design KLEE - Tracking Symbolic States

- Trees of symbolic expressions
  - Instruction pointer
  - Path condition
  - Registers, heap and stack objects
  - Expressions are of C language: arithmetic, shift, dereference, assignment
  - Checks inserted at dangerous operations: division, dereferencing

Tool Design KLEE - Tracking Symbolic States

- Modeling environment:
  - 2,500 lines of modeling code to customize system calls (e.g. open, read, write, stat, lseek, ftruncate, ioctl)
  - How to generate tests after using symbolic env: supply an description of symbolic env for each test path; a special driver creates real OS objects from the description

Tool Design KLEE - Constraint Solving

- STP: a decision procedure for Bit-Vectors and Arrays
- Decision procedures are programs which determine the satisfiability of logical formulas that can express constraints relevant to software and hardware
- STP uses new efficient SAT solvers
- Treat everything as bit vectors: arithmetic, bitwise operations, relational operations

Symbolic/Concolic Execution in Practice using KLEE

Install KLEE

- We use the docker image
  - docker pull klee/klee
- Run the docker image

```
>>cat run_docker.sh
docker run --rm -ti  --volume  $(PWD):/home/klee/klee --ulimit='stack=-1:-1' klee/klee
```

Example I: Testing a simple function

get_sign.c

```c
/*
 * First KLEE tutorial: testing a small function
 */

#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
     return 0;

  if (x < 0)
     return -1;
  else
     return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

Make an input as a symbolic value

- To mark a variable as symbolic, we use the **klee_make_symbolic()** function (defined in klee/klee.h)
- Arguments
  - the address of the variable (memory location) that we want to treat as symbolic
  - its size,
  - a name (which can be anything).

```
int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

Compiling to LLVM bitcode

- clang **-i ../../klee/klee_src/include/ -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone** get_sign.c
  - -I argument is used so that the compiler can find klee/klee.h
  - -g: add debug information to the llvm bitcode

Run KLEE

- klee get_sign.bc

```
klee@7fd91ba21ea4:~/klee$ klee get_sign.bc
KLEE: output directory is "/home/klee/klee/klee-out-2"
KLEE: Using STP solver backend

KLEE: done: total instructions = 33
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
```

```
klee@7fd91ba21ea4:~/klee$ ls -l /home/klee/klee/klee-out-2
total 36
-rw-r--r-- 1 klee klee 5477 Apr 22 16:12 assembly.ll
-rw-r--r-- 1 klee klee  631 Apr 22 16:12 info
-rw-r--r-- 1 klee klee   31 Apr 22 16:12 messages.txt
-rw-r--r-- 1 klee klee 1652 Apr 22 16:12 run.istats
-rw-r--r-- 1 klee klee  565 Apr 22 16:12 run.stats
-rw-r--r-- 1 klee klee   53 Apr 22 16:12 test000001.ktest
-rw-r--r-- 1 klee klee   53 Apr 22 16:12 test000002.ktest
-rw-r--r-- 1 klee klee   53 Apr 22 16:12 test000003.ktest
-rw-r--r-- 1 klee klee    0 Apr 22 16:12 warnings.txt
```

Using the test cases

```
klee@7fd91ba21ea4:~/klee$ ktest-tool /home/klee/klee/klee-out-2/test000002.ktest
ktest file : '/home/klee/klee/klee-out-2/test000002.ktest'
args       : ['get_sign.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x01\x01\x01\x01'
object 0: hex : 0x01010101
object 0: int : 16843009
object 0: uint: 16843009
object 0: text: ....
```

Example II: crack a password

```
int main(int a1, char **a2, char **a3)
{
  signed  long long v4; // rbx@10
  signed int v5; // [sp+1Ch] [bp-14h]@4

  if ( a1 == 2 )
  {
    if ( 42 * (strlen(a2[1]) + 1) != 504 )
      goto LABEL_31;
    v5 = 1;
    if ( *a2[1] != 80 )
      v5 = 0;
    if ( 2 * a2[1][3] != 200 )
      v5 = 0;
    if ( *a2[1] + 16 != a2[1][6] - 16 )
      v5 = 0;
    v4 = a2[1][5];
    if ( v4 != 9 * strlen(a2[1]) - 4 )
      v5 = 0;
    if ( a2[1][1] != a2[1][7] )
      v5 = 0;
    if ( a2[1][1] != a2[1][10] )
      v5 = 0;
    if ( a2[1][1] - 17 != *a2[1] )
      v5 = 0;
    if ( a2[1][3] != a2[1][9] )
      v5 = 0;
    if ( a2[1][4] != 105 )
      v5 = 0;
    if ( a2[1][2] - a2[1][1] != 13 )
      v5 = 0;
    if ( a2[1][8] - a2[1][7] != 13 )
      v5 = 0;
    if ( v5 ) {
      printf("Good good!\n");
      klee_assert(0);
    }
    else
LABEL_31:
      printf("Try again...\n");
  }
  else
  {
    printf("Usage: %s <pass>\n", *a2);
  }
}
```

Compile and Run KLEE

- clang -I ../../klee/klee_src/include/ -emit-llvm -g -o password.bc -c password.c
- Run KLEE
  - klee --optimize --libc=uclibc --posix-runtime password.bc --sym-arg 100
  - --sym-arg: under 100 chars

```
KLEE: done: total instructions = 230254
KLEE: done: completed paths = 2148
KLEE: done: generated tests = 2148
```

```
klee@7fd91ba21ea4:~/klee$ ls -l klee-out-1/*err
-rw-r--r-- 1 klee klee 378 Apr 22 16:05 klee-out-1/test000101.assert.err
klee@7fd91ba21ea4:~/klee$ ktest-tool klee-last/test000101.
test000101.assert.err  test000101.kquery      test000101.ktest
klee@7fd91ba21ea4:~/klee$ ktest-tool klee-last/test000101.ktest
ktest file : 'klee-last/test000101.ktest'
args       : ['password.bc', '--sym-arg', '100']
num objects: 2
object 0: name: 'arg00'
object 0: size: 101
object 0: data: b'Pandi_panda\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
f\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
f\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
object 0: hex : 0x50616e64695f70616e646100fffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
object 0: text: Pandi_panda........................................................................
object 1: name: 'model_version'
object 1: size: 4
object 1: data: b'\x01\x00\x00\x00'
object 1: hex : 0x01000000
object 1: int : 1
```

```
>>./password Pandi_panda
Good good!
```