Symbolic Execution

Wei Le

February 14, 2020

Outline

- What is *symbolic execution*?
- Applications
- History
- Interal Design: The three challenges
  - Path explosion
  - Modeling statements and environments
  - Constraint solving
- Implementation and symbolic execution tools

Concrete execution vs. symbolic execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =  -i ;
        return i;
}
```

i = 1

i = 1, j = 2

i = 2, j = 2

i = 4, j = 2

return 4

Concrete execution vs. symbolic execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        return i;
}
```

i_{input}

$i = i_{input}$, $j = 2 * i_{input}$

$i = i_{input} + 1$, $j = 2 * i_{input}$

$i = 2 * i_{input}$ ^2 + $2 * i_{input}$

Concrete execution vs. symbolic execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        return i;
}
```

i_{input}

$i = i_{input}, \; j = 2 * i_{input}$

$i = i_{input} + 1, \; j = 2 * i_{input}$

$i = 2 * i_{input}\text{^}2 + 2 * i_{input}$

$\boxed{\begin{array}{l} i = - 2 * i_{input}\text{^}2 - 2 * i_{input} \\ (2 * i_{input}\text{^}2 + 2 * i_{input} < 1) \end{array}}$

OR

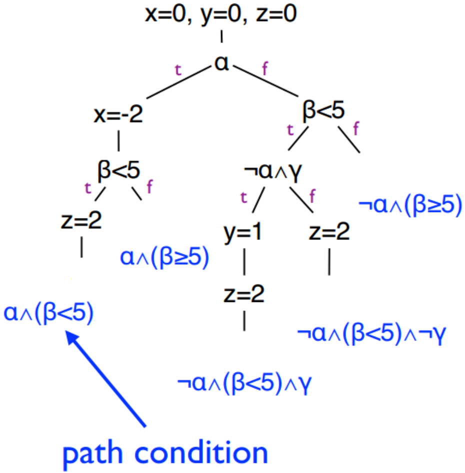
$\boxed{\begin{array}{l} i = 2 * i_{input}\text{^}2 + 2 * i_{input} \\ (2 * i_{input}\text{^}2 + 2 * i_{input} >= 1) \end{array}}$

Intuitive understanding of symbolic execution

- 'Execute' programs with symbols: we track symbolic state rather than concrete input
- 'Execute' many program paths simultaneously: when execution path diverges, fork and add constraints on symbolic values
- When 'execute' one path, we actually simulate many test runs, since we are considering all the inputs that can exercise the same path
- System calls and library calls: the symbolic execution engine creates an environment to mimic real executions
- Other variants
  - *Symbolic Analysis*: Model library and system calls rather than 'executing' it [Le2013]
  - *Concolic Testing*: Mixture of symbolic and concrete inputs [Sen2005]

Symbolic execution tree

```
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)
```

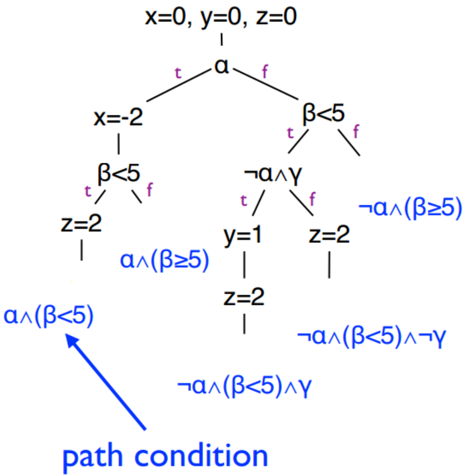


Applications

- Generating test inputs
- Finding bugs and vulnerabilities
- Detecting infeasible paths
- Proving two code segments are equivalent
- Repair programs
- Compare two programs
- Generate program specifications
- ...

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)
```



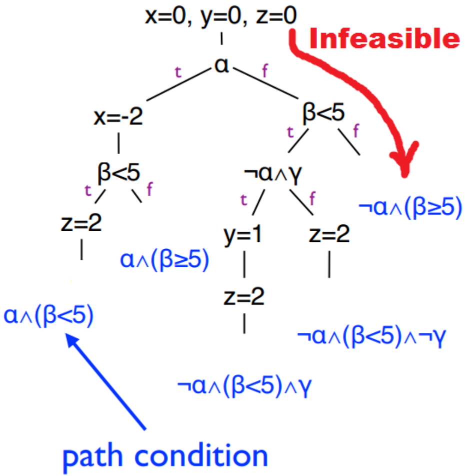
Path 1: $\alpha = 1, \beta = 1$
Path 2: $\alpha = 1, \beta = 6$
Path 3 ...

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)
```



Finding bugs

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        i = j/i;
        return i;
}
```

i_{input}

True branch:

$2^* i_{input}$ ^2 + $2^* i_{input}$ < 1

i = - $2^* i_{input}$ ^2 - $2^* i_{input}$

i == 0

False Branch:

$2^* i_{input}$ ^2 + $2^* i_{input}$ >= 1

i = $2^* i_{input}$ ^2 + $2^* i_{input}$

i == 0

Finding bugs

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        i = j/i;
        return i;
}
```

i_{input} **i_{input} = -1 Trigger the bug**

True branch:

$2 * i_{input}{}^2 + 2 * i_{input} < 1$

$i = -2 * i_{input}{}^2 - 2 * i_{input}$

$i == 0$

False Branch: **always safe**

$2 * i_{input}{}^2 + 2 * i_{input} >= 1$

$i = 2 * i_{input}{}^2 + 2 * i_{input}$

$i == 0$

Test input generation: CodeHunt

Secret Implementation

```
class Secret {
    public static int Puzzle(int x) {
      return 2*x-1;
    }
}
```

Player Implementation

```
class Player {
    public static int Puzzle(int x) {
          return x;
    }
}
```

```
class Test {
public static void Driver(int x) {
  if (Secret.Puzzle(x) != Player.Puzzle(x))
     throw new Exception("Mismatch");
  }
}
```

```
class Test {
public static void Driver(int x) {
  if (2*x-1 != x)
     throw new Exception("Mismatch");
  }
}
```

	x	your result	secret implementation result	Output/Exception
	1	1	1	
	3	3	5	Mismatch

Test input generation: DART

see ppt slides from Patrice for another example

History of 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.

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

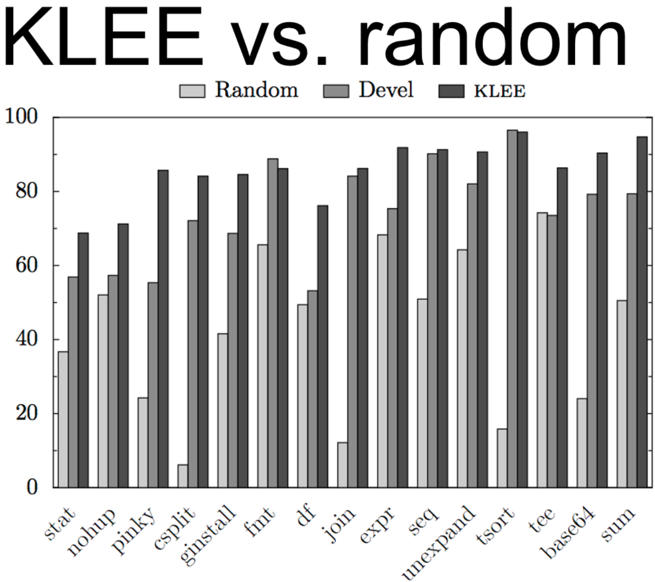
KLEE [2008:OSDI:Cadar]

- 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

Coverage Results: KLEE



Bug Detection Results: KLEE

Mismatch of CoreUtils and BusyBox

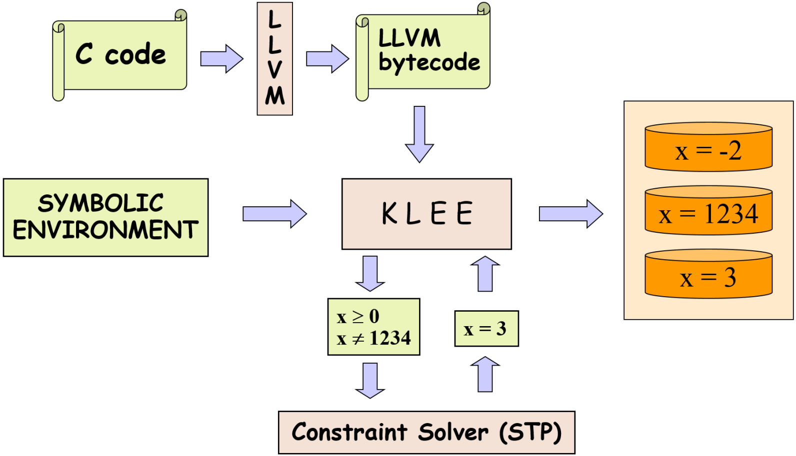
Input	Busybox	Coreutils
tee "" <t1.txt	[infinite loop]	[terminates]
tee -	[copies once to stdout]	[copies twice]
comm t1.txt t2.txt	[doesn't show diff]	[shows diff]
cksum /	"4294967295 0 /"	"/: Is a directory"
split /	"/: Is a directory"	
tr	[duplicates input]	"missing operand"
[0 "<" 1]		"binary op. expected"
tail -2l	[rejects]	[accepts]
unexpand -f	[accepts]	[rejects]
split -	[rejects]	[accepts]

t1.txt: a   t2.txt: b   (no newlines!)

Other symbolic executors

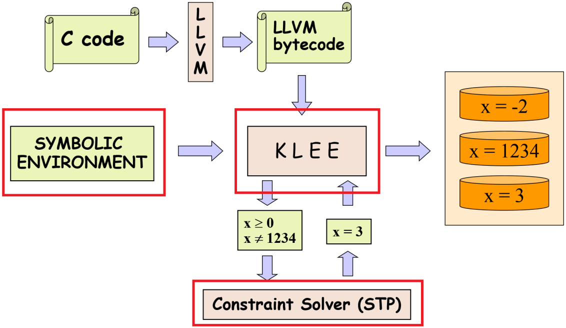
- Cloud9: parallel symbolic execution, also supports threads
- Pex, Code Hunt: Microsoft tools, symbolic execution for .NET
- Cute: concolic testing
- jCUTE: symbolic execution for Java
- Java PathFinder: NASA tools, a model checker that also supports symbolic execution
- SymDroid: symbolic execution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

Internal of symbolic executors: KLEE



Three Challenges

- Path explosion
- Modeling program statements and environment
- 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

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

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

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 it's been seen and how often; Pick next statement to explore that has lowest score

Pros and cons:

- 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)

Search Strategies: Combined Search

- Run multiple searches at the same time and alternate between them
- Depends on conditions needed to exhibit bug; so will be as good as *best* solution, with a constant factor for wasting time with other algorithms
- Could potentially use different algorithms to reach different parts of the program

Challenge 2: Complex Code and Environment Dependencies

- System calls: open(file)
- Library calls: sin(x), glibc
- Pointers and heap: linklist, tree
- Loops and recursive calls: how many times it should iterate and unfold?
- ...

Solutions

- Simulate system calls
- Build simple versions of library calls
- Assign random values after library calls
- Run library an system calls with a concrete value
- Summarize the loops
- ...

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

Program was initiated with a symbolic file system with up to N files.
Open all N files + one open() failure.

Solutions: Concretization [2005:PLDI:Godefroid],[2005:FSE:Sen]

- Concolic (concrete/symbolic) testing: run on concrete random inputs. In parallel, execute symbolically and solve constraints. Generate inputs to other paths than the concrete one along the way.
- Replace symbolic variables with concrete values that satisfy the path condition
- So, could actually do system 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 [2011:ACM:DeMoura]

SMT (Satisfiability Modulo Theories) = SAT++

$$\sin(x)^3 = \cos(\log(y) \cdot x) \vee b \vee -x^2 \geq 2.3y$$

- An SMT formula is a Boolean combination of formulas over first-order theories
- Example of SMT theories include bit-vectors, arrays, integer and real arithmetic, strings, ...
- The satisfiability problem for these theories is typically hard in general (NP-complete, PSPACE-complete, ...)
- Program semantics are easily expressed over these theories
- Many software engineering problems can be easily reduced to the SAT problem over first-order theories

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: $\exists$, $\forall$
- Disjunction

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

Modeling environment:

- 2500 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.

Tool Usage KLEE

- Using LLVM to compile to bytecode
- Run KLEE with bytecode

Discussions

- Symbolic environment interaction - how reliable can the customized modeling really be? think about concurrent programs, inter-process programs.
- What is more commonly needed - functional testing or security/completeness/crash testing?

Reference and Further Reading

Please find the references in [] in the above slides