Lecture 9. Symbolic Execution

Wei Le
Thank Cristian Cadar, Patrice Godefroid, Jeff Foster, Nikolai Tillmann, Vijay Ganesh
for Some of the Slides

2014.11
Outline

- What is symbolic execution?
  - Concrete execution versus symbolic execution
  - Symbolic execution tree
- Applications of symbolic execution: test input generation, infeasible paths detection, bug finding, program repair, debugging
- Code hunt
- History of the research since 1975
Outline

- What is symbolic execution?
  - Concrete execution versus symbolic execution
  - Symbolic execution tree
- Applications of symbolic execution: test input generation, infeasible paths detection, bug finding, program repair, debugging
- Code hunt
- History of the research since 1975
- The three challenges
  - Path explosion
  - Modeling statements and environments
  - Constraint solving
- Implementation and symbolic execution tools
Concrete Execution Versus Symbolic Execution

```c
int foo(int i){
    int j = 2*i;
    i = i++; // i = 1, j = 2
    i = i * j;
    i = -i;  // i = 4, j = 2
    if ( i < 1 )
        i = -i;
    return i;
}
return 4
```
Concrete Execution Versus Symbolic Execution

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

```plaintext
\[ i_{input} \]
\[ i = i_{input}, \ j = 2 * i_{input} \]
\[ i = i_{input} + 1, \ j = 2 * i_{input} \]
\[ i = 2 * i_{input} ^ 2 + 2 * i_{input} \]
Concrete Execution Versus Symbolic Execution

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

$i_{input}$

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

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

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

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

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

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

$(2* i_{input} ^ 2 + 2* i_{input} >= 1)$
Some Insights about 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
int a = α, b = β, c = γ;
// symbolic

int x = 0, y = 0, z = 0;
if (a) {
    x = -2;
} else if (b < 5) {
    if (!a && c) { y = 1; }
    z = 2;
}
assert(x+y+z!=3)
Applications of Symbolic Execution

**General goal:** identifying semantics of programs

**Basic applications:**
- Detecting infeasible paths
- Generating test inputs
- Finding bugs and vulnerabilities
- Proving two code segments are equivalent (Code Hunt)

**Advanced applications:**
- Generating program invariants
- Debugging
- Repair programs
Detecting Infeasible Paths

Suppose we require $\alpha = \beta$

```plaintext
int a = \alpha, b = \beta, c = \gamma;
    // 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)
```

```
v=
|   x=0, y=0, z=0
|   \beta<5
x=-2
|   \beta<5
|   \neg\alpha \land \gamma
   \neg\alpha \land (\beta \geq 5)
   \neg \alpha \land (\beta < 5) \land \neg \gamma
   \neg \alpha \land (\beta < 5) \land \gamma
   \alpha \land (\beta \geq 5)
   \alpha \land (\beta < 5)
   z=2
   y=1
   z=2
   \neg \alpha \land (\beta \geq 5)
   \neg \alpha \land (\beta < 5) \land \neg \gamma
   \neg \alpha \land (\beta < 5) \land \gamma

path condition
```
Test Input Generation

\[
\text{int } a = \alpha, \ b = \beta, \ c = \gamma; \\
\text{int } x = 0, \ y = 0, \ z = 0; \\
\text{if (a) } \{ \\
\quad x = -2; \\
\}\text{ if (b < 5) } \{ \\
\quad \text{if (!a && c) } \{ y = 1; \} \\
\quad z = 2; \\
\}\text{ assert(x+y+z!=3) }
\]

Path 1: \( \alpha = 1, \beta = 1 \)
Path 2: \( \alpha = 1, \beta = 6 \)
Path 3 ...
Bug Finding

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

```c
i_{\text{input}}
```

**True branch:**

```
2* i_{\text{input}}^2 + 2* i_{\text{input}} < 1
i = - 2* i_{\text{input}}^2 - 2* i_{\text{input}}
i == 0
```

**False Branch:**

```
2* i_{\text{input}}^2 + 2* i_{\text{input}} >= 1
i = 2* i_{\text{input}}^2 + 2* i_{\text{input}}
i == 0
```
Bug Finding

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

```plaintext
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: Code Hunt

Code Hunt has had several hundred thousands of users since launch in March 2014.
Stats from Visual Studio Analytics over the period May 22-June 26 indicate 40,235 users.
Stickiness (loyalty) is very high.

% Returning
< 14 days

99.79%
Code Hunt Demo
Code Hunt: Behind the Scene

**Secret Implementation**

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

**Player Implementation**

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

**Class Test**

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

<table>
<thead>
<tr>
<th>x</th>
<th>your result</th>
<th>secret implementation result</th>
<th>Output/Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>Mismatch</td>
</tr>
</tbody>
</table>
History of Symbolic Execution


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 [1]**
- Open source symbolic executor
- Runs on top of LLVM
- Has found lots of problems in open-source software

**SAGE [3]**
- 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  symbolic execution for .NET
- jCUTE  symbolic execution for Java
- Java PathFinder  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

C code → LLVM → LLVM bytecode → KLEE → Constraint Solver (STP)
Three Challenges

- Path explosion
- Modeling program statements and environment
- Constraint solving
Path Explosion

- **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!
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 pseudo-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

**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: Generational Search [4, 5]

See example of DART
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
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

- Build simple versions of library calls
- Summarize the loops
- Simulate system calls
- ...

...
An Example

Program was initiated with a symbolic file system with up to N files. Open all N files + one open() failure.
Solutions: Concretization [4, 5]

- 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
Solutions: Concretization [4, 5]

See example of DART
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 [2]

SMT (Satisfiability Modulo Theories) = SAT++

\[ \sin(x)^3 = \cos(\log(y) \cdot x) \lor b \lor -x^2 \geq 2.3y \]

- 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: \( a = b.\text{replace}('x', 'y') \)
- Quantifies: \( \exists \), \( \forall \)
- Disjunction
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
Coverage Results: KLEE

KLEE vs. random

Graph showing coverage results for various programs with KLEE, Devel, and Random methods.
## Bug Detection Results: KLEE

Mismatch of CoreUtils and BusyBox

<table>
<thead>
<tr>
<th>Input</th>
<th>Busybox</th>
<th>Coreutils</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tee &quot;&quot; &lt;t1.txt</code></td>
<td>[infinite loop]</td>
<td>[terminates]</td>
</tr>
<tr>
<td><code>tee -</code></td>
<td>[copies once to stdout]</td>
<td>[copies twice]</td>
</tr>
<tr>
<td><code>comm t1.txt t2.txt</code></td>
<td>[doesn't show diff]</td>
<td>[shows diff]</td>
</tr>
<tr>
<td><code>cksum /</code></td>
<td>&quot;4294967295 0 /&quot;</td>
<td>&quot;/: Is a directory&quot;</td>
</tr>
<tr>
<td><code>split /</code></td>
<td>&quot;/: Is a directory&quot;</td>
<td></td>
</tr>
<tr>
<td><code>tr</code></td>
<td>[uplicates input]</td>
<td>&quot;missing operand&quot;</td>
</tr>
<tr>
<td><code>[ 0 &quot;&lt;&quot; 1 ]</code></td>
<td></td>
<td>&quot;binary op. expected&quot;</td>
</tr>
<tr>
<td><code>tail -2l</code></td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td><code>unexpand -f</code></td>
<td>[accepts]</td>
<td>[rejects]</td>
</tr>
<tr>
<td><code>split -</code></td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td><code>t1.txt: a</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>t2.txt: b</code></td>
<td>(no newlines!)</td>
<td></td>
</tr>
</tbody>
</table>
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?
Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. 


Koushik Sen, Darko Marinov, and Gul Agha.
Cute: A concolic unit testing engine for C.