Automated Testing & Verification

Dynamic Symbolic Execution

Galeotti/Gorla/Rau
Saarland University
Symbolic Execution

- King [Comm. ACM 1976]
- Analysis of programs with unspecified inputs
  - "Execute" a program on symbolic inputs
- Symbolic states represents sets of concrete states
- For each path, build a path condition
  - Condition on inputs – for the execution to follow that path
  - Check path condition satisfiability – explore only feasible paths
1: int x, y;
2: if (x > y) {
3:   x = x + y;
4:   y = x - y;
5:   x = x - y;
6:   if (x > y) {
    7:     assert false;
  }
}

1: x == 1 && y == 0
2: 1 > 0 ? true
3: x = 1 + 0 = 1
4: y = 1 - 0 = 1
5: x = 1 - 1 = 0
6: 0 > 1 ? false
1: int x, y;
2: if (x > y) {
3:   x = x + y;
4:   y = x - y;
5:   x = x - y;
6:   if (x > y) {
7:     assert false;
7:   }
8: return;
7: }
Symbolic Execution

- Use an off-the-shelf constraint solver to solve path conditions
  - SMT-Solver / SAT-Solver
- Solutions are inputs to force that traversal
- Path constraints:
  - Solve(X<=Y) $\rightarrow$ x=-1 && y=54
  - Solve(X>Y && Y<=X) $\rightarrow$ x=10 && y=5
  - Solve(X>Y && Y>X) $\rightarrow$ UNSAT
Symbolic Execution Limitations

- Too many path conditions
  - Exploring a very small set of executions

```cpp
test_me(int x, int y) {
    if (((x % y) * 4 != 17)) {
        ERROR
    } else {
        ERROR
    }
}
```
Constraint solver capabilities

```cpp
test_me(int x, int y) {
    if (bbox(x, y) = 17)) {
        ERROR
    } else {
        ERROR
    }
}
```
Native code (no access to source code/environment)
Executions Paths of a Program

- Can be seen as a binary tree with possibly infinity depth
  - Computation tree

- Each **node** represents the execution of a “if then else”

- Each **edge** represents the execution of a sequence of non-conditional statements

- Each **path** in the tree represents an equivalence class of inputs
How many feasible execution paths do you have in this program?

```c
void testme1(int x) {
    for (int j=0; j<2; j++) {
        if (x==j) {
            printf("Good\n");
        }
    }
}
```
Add assertions as new branches

- Divide by zero
  - $x = 3 / i$

- Buffer overflow
  - $a[i] = 4$

```
if (i != 0) {
    x = 3 / i;
} else {
    ERROR
}
```

```
if (i >= 0 && i < a.length) {
    a[i] = 4;
} else {
    ERROR
}
```
Random testing

- Generate random inputs
- Execute the program on generated inputs
- Probability of reaching an error can be astronomically lesser.

```c
1: testme(int x) {
2:   if (x==94389) {
3:     assert false;
3:   }
3: }
```
Dynamic Symbolic Execution

Code to generate inputs for:

```csharp
void CoverMe(int[] a) {
  if (a == null) return;
  if (a.Length > 0) {
    if (a[0] == 1234567890)
      throw new Exception("bug");
  }
}
```

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Data</th>
<th>Observed constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>a!=null</td>
<td>null</td>
<td>a==null</td>
</tr>
<tr>
<td>a!=null &amp;&amp; a.Length&gt;0</td>
<td>{}</td>
<td>!(a.Length&gt;0)</td>
</tr>
<tr>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]!=1234567890</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]==1234567890</td>
<td>{123..}</td>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]==1234567890</td>
</tr>
</tbody>
</table>

Negated condition

Choose next path

Solve

Execute&Monitor

Done: There is no path left.
Dynamic Symbolic Execution

- Efficiently traverse all execution paths one by one to detect errors
  - Assertion violations
  - Program crash
  - Uncaught exceptions
- In the path build a test suite
  - Branch coverage
Dynamic Symbolic Execution

1: void testme(x,y) {
2:   z=foo(y);
3:   if (z==x) {
4:     if (x>y+10) {
5:       ERROR
6:     }
7:   }
8:   return;
9: }

Initial data:
x==15 && y==-541

Path constraint:
foo(Y)!=X

Don’t know how to solve foo(Y)==X! Stuck?
1: void testme(x,y){
2:     z=foo(y);
3:     if (z==x) {
4:         if (x>y+10) {
5:             ERROR
6:         }
7:     }
8:     return;
9: }

Initial data:
\(x==15 \land y==-541 \land \text{foo}(-541)==85\)

Path constraint:
\(85!=X\)

Solve(85==X) \(\Rightarrow X==85 \land Y==-541\)
We can use concrete values in the constraint system instead of symbolic representation

- Sound
- Incomplete (we stop trying to reason about some part of the code)
Tools! Tools!

- CUTE (C), jCUTE (Java), CREST (C)
- PEX
  - Visual Studio 2010 Power Tool
  - http://pexforfun.com
- SAGE
  - X86 assembly
  - Targets security problems
  - 24x7 use (million-dollar bugs)
- EXE and KLEE
Challenges

- **Scalability**
  - Key challenge!
  - Path Space of a Large program is Huge

- **Complex Non-Linear constraint**
  - Floating point numbers

- **Testing web apps and security problems**
  - String constraints
  - Mixed numeric and string constraints
Dynamic Symbolic Execution

- Dynamic Technique
  - In contrast symbolic execution is static

- More coverage: Guide exploration using collected path conditions

- Whenever you can’t reason about something, use a collected concrete value