Announcements

- HW7 will be due in about ten days
- HW8 will be optional

Presentations
- Some of you haven’t sent paper selection
- Please do!

Quiz 6

Outline

- Symbolic Execution
  - Overview and applications
  - Challenges
  - Tools and techniques


Classical References


Resurgence and Applications

- More powerful computers lead to much more powerful reasoning tools (e.g., Z3)
- Systems that started a resurgence
  - DART by Godefroid and Sen, PLDI 2005
  - EXE by Cadar, Ganesh, Pawlowski, Dill and Engler, CCS 2006

Symbolic Execution, Example 1

```c
void foobar(int a, int b) {
    if (a>b) {
        a = a+b;
        b = a-b;
        a = a-b;
        if (a-b>0) {
            assert false;
        }
        return 0;
    }
}
```
**Example 1**

```
T = (a+b) \land \neg (a+b)
```

```
F = (a+b) \land (a+b)
```

**Symbolic Execution, Example 2**

```c
void foobar (int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0) {
            x = 2*(a+b);
            assert (x-y != 0) OK.
        }
    }
    assert (x-y != 0) NOT OK.
}
```

**Motivation for Symbolic Execution**

- Why?
  - One symbolic execution path covers **many** actual inputs
  - Exactly the set of inputs that satisfy the path condition
  - Thus, we cover a lot more of the program input space than testing

**VC Generation Works Too**

```c
void foobar (int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0) {
            x = 2*(a+b);
        }
    }
    assert (x-y != 0) NOT OK.
}
```

**VC generation vs. Symbolic Execution?**

- VC generation = Backward reasoning
  - HW7
- Symbolic execution = Forward reasoning
  - HW8 (optional): Add a symbolic execution engine as another "interpreter" of IMP programs
Challenges to Symbolic Execution

- Loops and recursion
- Infinite execution trees
- Path explosion
- n conditionals generate \(2^n\) paths
- Heap memory: pointers and arrays
- Modeling pointers and arrays symbolically
- Solvers
- Cannot handle complex path conditions
- “Edge” of the program
- Libraries, binary code, etc.

Loops and Recursion

- Limit size of path condition, i.e., number of loop iterations (bounded execution). E.g., \(k=2\)

```plaintext
t = x - y;
while \((t>0)\) {
  x = x - 1;
  y = y + 1;
  t = t - 1;
}
```

- Use the loop invariant

```plaintext
\sigma = \{ x \to x_0, y \to y_0, t=x_0-y_0 \}
\pi = \text{true}
```

- Path explosion: Search Strategies

- Program execution as a tree or a DAG
- Nodes \(n_i\) represent states
- Edges \((n_i,n_j)\) represent state transitions
- We need strategies/heuristics for graph exploration
  - At each step, how do we chose which paths to explore and which paths to drop
- There are many strategies and heuristics!

Search Strategies

- DFS
- BFS
- Advantages
  - Simplicity
- Drawbacks
  - Generally, unguided by other “knowledge”
  - DFS can get stuck in one part of program
  - BFS considered the better one
Search Strategies
- Heuristics try to steer towards paths “more likely” to fail assertions
- Run symbolic execution engine for a limited period of time
- One big idea: randomness
  - At each step choose T or F branch at random
  - Consensus: randomness works very well!
  - Any new heuristic must compare with random
- A drawback: reproducibility

Concolic Execution
- Another big idea: Sen’s CUTE and DART tools (FSE 2005, PLDI 2005)
- Mixes concrete and symbolic execution
- One variation: dynamic symbolic execution
  - Select some inputs
  - Run path from start to finish, maintaining concrete state and symbolic state
  - When finished, generate a new path condition by negating the last clause of path condition
  - Solve new path condition and if satisfiable, generate an input and run program again

Concolic Execution, Example 1
Suppose we chose inputs a=1, b=1.

Example 2
1. int a = α; b = β; c = γ;
   // symbolic values
2. int x = 0, y = 0, z = 0;
3. if (a) {
   4.   x = -2;
   }
5. if (b < 5) {
6.   if (!a && c) { y = 1; }
7.   z = 2;
8. }
assert (x+y+z!)=3

Concolic Execution
- Why concolic execution works?
  - Search is guided by a concrete path, therefore there are shadow concrete values for most symbolic variables
  - SMT formula becomes easier to solve
Heap Memory

- Classical techniques include state forking and if-then-else formulae

```c
foobar(unsigned i, unsigned j) {
    int a[2] = {0}; // a[0] = 0, a[1] = 0
    if (i>1 || j>1) return;
    a[i] = 5;
    assert(a[j] != 5)
}
```

State Forking

At array read or write, the state is forked considering all possible results of the operation!

If-then-else Formulae

- State is encoded in an if-then-else formula. No forking of state.

Heap Memory

- Other approaches
  - Address concretization
  - Partial memory modeling
  - Lazy concretization

“Edge” of the Program

- Libraries, binary code
- One way
  - Pull in library code (libc, glibc)
  - Hard. Symbolic execution easily gets stuck
- Another way
  - Summaries (stubs) for library code
  - Also hard. A lot of work and often unsound
  - Concolic execution gets around it!

Recent Success

- SAGE
  - Microsoft, concolic execution
  - Finds bugs in file parsers
  - Microsoft continuously runs SAGE!
- Mayhem
  - Combines BFS and advanced search techniques
  - Runs on binary code
  - Automatically generates exploits when bug found
- KLEE
  - Symbolically executes LLVM bitcode
Next class: Presentations!

- Attend the presentations!
- Ask questions!