Symbolic Execution for Software Testing

Scribe: Andrew Liu

November 10, 2015

1 Motivation

Goal: We want to generate high coverage tests and be able to find deep errors in software programs.

- There is a trade-off between test coverage and cost; usually, it is not enough resources (time, computational power, etc) to explore every possible path. We want to be able to explore as many execution paths in the program as possible in a given time constraint.
- We want to identify program endpoints (errors, termination, etc) and generate a set of concrete inputs that reach the program endpoints.
- We also want to check for reachability of errors and assertion violations.

2 Overview

2.1 Definitions

Variables - Classical symbolic execution uses symbols as opposed to concrete values to define inputs. The variables are represented as an expression involving the symbolic inputs and can be solved in order to find concrete values that satisfy the conditions.
Why? Most of the time, multiple input sets can lead to the same execution path since the values all satisfy the same conditions. Instead of checking every possible input, we can group classes of inputs together if they execute the same path.

Execution Path - a set of boolean values where the ith value is true if the ith conditional encountered was true. The set of all execution paths can be represented as an execution tree (similar to a decision tree).

The function testme() in the example below has 3 possible execution paths.

```
int twice (int v) {
    return 2v;
}

void testme (int x, int y) {
```

```
    z = twice (y );
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
int main() {
    x = sym input();
    y = sym input();
    testme (x, y );
return 0;
}
```

Symbolic Execution - uses symbolic state, σ, as opposed to concrete state. Symbolic execution maps variables to symbolic expressions.

Path Constraints - Defined as a formula over symbolic expressions.

```
// Path constraint here is PC
if e:
    s1
else:
    s2
```

When you reach a conditional, such as as the one listed above, you construct a new path constraint for each branch of the conditional. Given the path constraint, PC, that led to the conditional, and the conditional expression, e, then the path constraint for the true branch, $PC_{true} = PC \wedge \sigma(e)$. Similarly, $PC_{false} = PC \wedge \neg\sigma(e)$.

2.2 Execution

Traditionally, symbolic execution takes two steps.

1. First, the compiler executes the program symbolically by following each branch and updating the symbolic state, σ and path constraint on the current branch.

2. Once it reaches a program endpoint, a constraint solver tries to solve the path constrains in order to produce a set of concrete inputs.

3 Challenges

There are multiple challenges with modern symbolic execution.

3.1 Functional Challenges

1. What if a function is not defined in the code?

2. What if the function is non-linear?

3. What if the function cannot be solved efficiently?

In these cases, it is much harder to solve the path constraint. One possible solution is to use a hybrid approach called execution-generated testing which is similar to concolic testing. In execution-generated testing, a subset of the compiler replaces a subset of the symbols with concrete values. For example, the function $y = x * x * x (mod 50)$ is non-linear and inefficient to solve. In execution-generated testing, we would use a concrete value x=5 for the input.

3.2 Path Explosion

Often times, the number of program paths grows exponentially with the number of conditional branches in the code. As a result, it is usually infeasible to explore every possible execution path during a limited amount of time. Often times, symbolic execution programs will use some sort of heuristic to prune paths or prioritize some paths over others. Most often, the heuristics try to optimize for high coverage (both line coverage and branch coverage). Some heuristics rely on a measure of path closeness (measured using the control flow graph). Other heuristics give more weight to paths that have been less explored. Similarly, other heuristics mix symbolic execution with random testing, using symbolic execution to reach a specific deep state and then random testing to check available paths.

3.3 Modeling Memory and Concurrency

Another scalability problem for symbolic execution is the difficulty of representing and abstracting complex data types. Similarly, it is often hard to model concurrency in complex systems due to the difficulty of handling the non-deterministic behavior.

4 Presentation: EXE

EXE is a symbolic execution tool for C programs. It applies input fuzzing in parallel in order to iteratively (and more efficiently) explore new paths. To do this, it randomly applies mutations to well-formed inputs and tests for crashes or other errors. EXE defines endpoints as 1) exit(), 2) crash, 3) assertion failure, or 4) some kind of detected error. Once it reaches an endpoint, it uses STP in order to efficiently solve the path constraints for concrete values.