Symoblic Execution

CS261: Security in Computer Systems - Scribe Note

Scriber: Jeongseok Son

October 18, 2017 *

1 Basic concept

Symbolic execution helps finding all execution paths of a program and generate corresponding input for each path. This ensures a program is tested using inputs which exercise all possible execution paths. To this end, symbolic values are assigned to input variables instead of concrete values. Program variables are represented as symbolic expressions and each execution path produces a different path constraint, a conjunction of symoblic expressions encountred. Solving a path constraint results in a set of input values that exercises the path.

1.1 Symbolic input value

In symbolic execution, the input varialbes such as paramters of a function or the user's input are assigend symbolic values instead of concrete values. A symbolic value is a mathematical symbol which represents an arbitrary value that can be assigned to a variable.

```
void foo(int x, int y) {
    int z = 2 * x;
    int k = 3;
    if (z > k) {
        if (y < z) {
            exit(EXIT_FAILURE);
        } else {
            assert(y != z);
        }
    }
}
```

Figure 1: Example C code

Figure 1 is an example C code to illustrate how symbolic executioin works. In this case, the function parameters x and y are the input variables. These variables are initialized with symbolic values in the first place.

$$\{x = x_0, y = y_0\}$$

*This scribe note includes what Caroline and Kevin covered before Prof. Wagner gave a short lecture. I included some important contents not discussed in the class but explained in the reading in addition.

1.2 Execution path

An execution path is a possible flow of control of a program. Each execution path maintains and updates mapping from variables to symbolic expressions during symbolic execution. Control flow statements such as if makes a current execution path diverge into two different execution paths. Execution paths can be expressed as an execution tree. Figure 2 is an execution tree of the example code of Figure 1. The leaf nodes are path constraints which are explained in the next subsection.

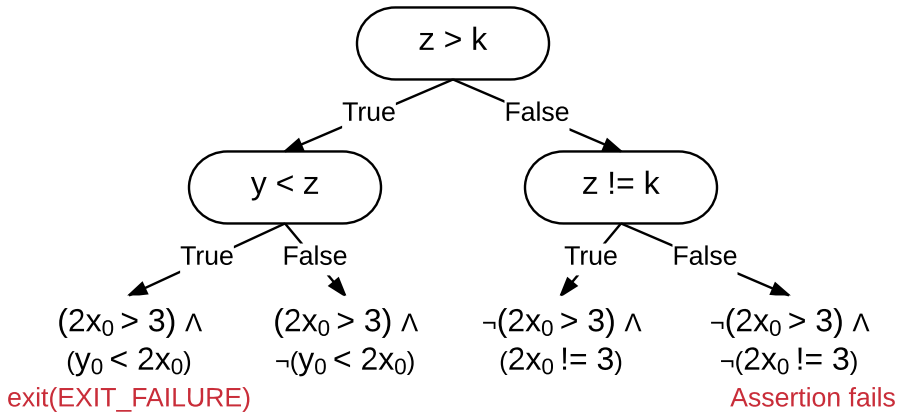

Figure 2: Execution tree of the example code

1.3 Path constraint

During symbolic execution, every variable is evaluated to a symbolic expression using the symbolic values of input variables. Each execution path has its own *path constraint* initially $true$. When the symbolic execution tool encounters a control flow statement, the $true$ and $false$ condition of the statement are conjuncted to the diverged paths respectively. When the tool reaches the end of a program or the termination point, the final path constraint is the condition of the input values that lead the control flow to that result.

The leaf nodes of the execution tree in Figure 2 are denoting path constraints of all possible execution paths. For example, all the input values which satisfy $(2x_0 > 3) \wedge (y_0 < 2x_0)$ make the program exit with a return value indicating the unsuccessful termination.

2 Testing and verification

A path constraint is the condition of input values that leads to the corresponding execution path. By solving path constraints, symbolic execution can generate minimal test cases which exercise all possible execution paths. Also, checking the satisfiability of a path constraint which leads to a certain execution result can verify whether the execution path is never taken under every possible input combination without testing them at all.

2.1 Solving path constraints

Finding an input value which exercises a certain execution path is essentially the same as solving the path constraint and figuring out the values satisfying the condition. If the path constraint is unsatisfiable, there are no input values which can lead to the corresponding execution path. To solve symbolic expressions, SMT

(satisfiability modulo theories) solvers are used generally. Compared to SAT solver, SMT solvers can solve conditions containing variables which are not boolean variables as well.

2.2 Test case generation

Symbolic execution helps the developer generate a minimal set of test cases exercising all possible execution paths. Since each path constraint expresses different input values for the corresponding path, solving all path constraints using the solver creates the test cases.

2.3 Verification

Symbolic execution can be used to verify whether a certain condition is never met under all possible execution scenarios of a program. To do so, a symbolic execution tool checks the satisfiability of the path constraint of an execution path in question. If the path constraint is *unsatisfiable*, there are no input values that can exercise the execution path so the path is never taken.

For example, the assertion condition in Figure 1 can be chcked by solving the path constraint leading to assertion failure in Figure 2, $\neg(2x_0 > 3) \wedge \neg(2x_0! = 3)$, to confirm it is unsatisfiable.

Security bugs can be found by the combination of symbolic execution and instrumentation. To detect buffer overrun vulnerabilities, assertion statements checking an accessed index is within the length of an array are inserted for every array access statement. Running symbolic execution on this code and confirming the unsatisfiability of assertion failure path constraints ensures there are no out-of-bounds bugs in the source code. Another type of security bug, TOCTTOU (time-of-check-to-time-of-use) can also be detected with symbolic execution and source code instrumentation.

3 Limitation of symbolic execution

Although symbolic execution is a powerful tool for generating test cases with high coverage and for verification, it has a few limitations.

3.1 Unsolvable path constraints

The main limitation of classical symbolic execution is that it cannot be used if a path constraint includes a condition which is not solvable by SMT solvers. For example, a symbolic expression may contain non-linear computation which might not be solvable by a constraint solver. Also, classical symbolic execution cannot deal with functions externally referenced whose source code is not available.

To overcome this limitation, dynamic symbolic execution technique such as concolic testing and Execution-Generated Testing (EGT) are used. They mix concrete and symbolic execution to circumvent the cases where symbolic execution cannot be done. Although it may sacrifice the completeness of the technique, it strengthens the usability of the technique by supporting source code which cannot be checked with the classical approach.

3.2 Path explosion

The number of execution paths increases exponentially as more conditional statements are nested within others. If the code base is large and complex, the time it takes to explore all execution paths with symbolic execution can be prohibitively long.

Thus, some heuristics are normally introduced when doing symbolic execution to achieve high coverage without exploring all paths. This includes control-flow graph analysis, random exploration of paths and random testing. Some static analysis techniques are used as well to reduce the number of paths in a sound way. It involves merging the same paths, reusing the summaries of functions tested and pruning redundant paths.

4 Coverage-based Greybox Fuzzing as Markov Chain

Fuzzing is a testing technique which inputs a large amount of random data to the program to find any bugs or vulnerabilities of a program. Symbolic execution-based white box fuzzer is effective in generating different test cases exploring different paths, but it comes at the cost of program analysis and constraint solving. Blackbox fuzzing can generate a large amount of test cases since it does not require time-consuming analysis but may fail to achieve high coverage.

Coverage-based Graybox Fuzzing (CGF) is a technique in between the aforementioned two techniques. GCF uses the coverage information to decide which inputs are finally used for testing. It assigns high priority to random seeds which exercise low-frequency paths. The authors analyze CGF as the exploration of program space of a Markov chain and identified existing tools visit high-frequency paths too often. By prioritizing low-frequency paths when scheduling paths and energy to the state, their implementation of AFLFast achieves an order of magnitude speedup compared to AFL.