Symbolic Execution and Applications

Christoph Thiede

TODO: Abstract

1 Introduction

Program analysis covers a wide range of techniques and programming tools for generating insights about the structure and the behavior of a software system. We can distinguish between *static* and *dynamic* analysis tools: static analysis tools examine a program by its source code to derive particular information (e.g., dependency analysis, type flow analysis) or detect occurrences of relevant patterns (e.g., spell checkers, linters). Dynamic analysis tools, on the other hand, examine a program by observing the behavior of the running application (e.g., tests, benchmarks, or assertions).

While static analysis typically offers a higher performance, it often provides a lower precision or selectivity than dynamic analysis. For instance, static type flow analyzers are unable to reason about metaprogramming constructs such as Python's `getattr` function or JavaScript's `Function.prototype.bind()` method, and static performance prediction tools cannot respect bottlenecks of an individual hardware setup. However, while dynamic analyis tools typically offer a higher quality of results, they require *context* for running an application and reaching all its different states. Programmers can provide context by composing a set of possible user interactions or creating a comprehensive test suite, but this is a costly process and often, these artifacts do not exist or only cover a small portion of the code base.

Symbolic execution addresses this issue by automatically uncovering most execution paths of a program and performing analyses on them while requiring no or fewer specifications from programmers. The fundamental operating principle of symbolic execution is to execute a program with *symbolic variables* instead of concrete values for its inputs and to fork the execution of the program every time a conditional expression (e.g., in an `if` statement) can have different results depending on the assignment of the symbolic variables.

In the last four decades, programmers and researchers have applied symbolic execution for different purposes such as vulnerability analysis of programs, program verification, unit testing, or various program understanding tasks such as model generation or reverse engineering.

In this report, we give a summary of several implementations and applications of symbolic execution. In section 2, we describe the fundamental operating principle of symbolic execution. In section 3, we provide an overview of challenges for symbolic execution and existing solution approaches. We present different use cases and tools for symbolic execution in section 4 and examine their impact in research and industry in section 5. In section 6, we discuss some limitations and usage considerations for different symbolic execution tools. We classify some alternative approaches and literature in section 7 and finally give a conclusion in section 8.

2 Approach

In this section, we describe the general concept of symbolic execution and introduce neccessary definitions.

Symbolic execution (also abbreviated to *symbex* or *SymEx*) attempts to systematically uncover all execution paths of a program. It takes a program in an executable form (e.g., x86 binary, LLVM bitcode, or JVM bytecode) and produces a list of inputs that trigger different code paths in the program. Cadar describes it as a technique "to turn code 'inside out' so that instead of consuming inputs [it] becomes a generator of them" [1].

To this end, the *symbolic execution engine* or the *symbolic executor* runs the program with its normal execution semantics but assigns a special *symbolic variable* to each input:

- As the program performs arithmetic or logic operations on symbolic variables, they are composed to *symbolic expressions* that can be propagated through the *symbolic memory* or *symbolic store* of the program execution.
- As the program hits a branch condition such as an `if` statement that depends on a symbolic expression, the executor decides the possible results of the depended expression (e.g., `true` or `false`). If the expression does not have a single solution, the entire execution is *forked* into multiple *execution paths* that assume a different result of the expression, and each fork is assigned a new *path constraint* for the assignment of its symbolic variable in the form of a Boolean first-order expression.

Together, the symbolic memory and the constraint set of an execution path constitute its *symbolic state*. The entirety of execution paths is named the *symbolic execution tree* of the program where each node represents a symbolic state and each edge represents a new path constraint. For each leaf, i.e., each completed execution path, the symbolic execution engine can generate a concrete set of input values from the contraint set that programmers can use to reproduce the same execution path in a regular non-symbolic context.

A crucial component of each symbolic execution engine is an *SMT solver* that decides whether a symbolic branch condition can be fulfilled and generates a concrete solution for a constraint set. SMT solvers are a special kind of *SAT solvers* that test the *satisfiability* of a constraint set *modulo* ("within") a set of *theories*. Theories are axiomatic systems for domain-specific algebras that enable the solver to reason not only about Boolean logic but also about predicates involving various datatypes. Popular theories describe arithmetics, bitwise operations/integers with overflow semantics, or strings.

For example, a symbolic execution of the `foo()` function in listing 1 would produce the symbolic execution tree displayed in fig. 1.

Listing 1: A simple method in C that can be executed symbolically.

```
1  void foo(int a, int b, int c) {
2      int x = 0, y = 0, z = 0;
3      if (a) {
4          x = -2;
5      }
6      if (b < 0) {
7          if (!a && c) {
8              y = 1;
9          }
10         z = 2;
11     }
12 }
```

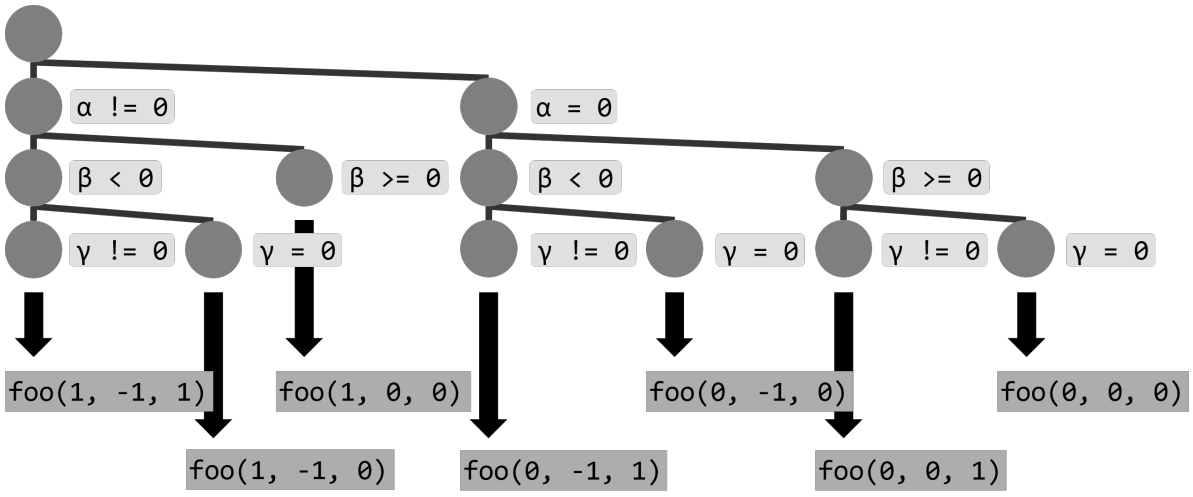

Figure 1: The symbolic execution tree for analyzing the `foo()` method from listing 1. The symbolic variables α, β, γ represent the assigned values for the input variables a, b, and c.

3 Implementation Strategies

4 Applications

5 Impact

6 Discussion

7 Related Work

8 Conclusion

References

[1] Cristian Cadar and Dawson Engler. “Execution generated test cases: How to make systems code crash itself”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 2005, pp. 2–23.