Managing Path Explosion in Lazy Symbolic Execution
Maganty Rushyendra
September 2019

Supervised by:
William Hallahan, PhD Candidate, Department of Computer Science, Yale University
Prof. Ruzica Piskac, Department of Computer Science, Yale University

1 Background

1.1 Symbolic Execution

As software programs balloon in size and complexity, it becomes increasingly difficult for developers to ensure that their code meets specifications and is free of subtle errors. Symbolic Execution tools enable developers to specify high-level properties that their programs should satisfy, and check that their code does not violate any of these properties, generating sample test inputs in the event that they do.

The central idea behind symbolic execution involves replacing test inputs to a program with symbolic values, and then running an interpreter through the code, continually updating a mathematical expression (path condition) that constrains the values these symbols can take. At each point where the executor encounters a branch condition that depends on the symbolic values, the execution splits into two separate states, each with its own path condition. If the execution reaches an assertion statement or error, the path condition is analyzed to see if there exists some combination of concrete input values that violates the assertion statement and fits the constraint. This combination of concrete values is returned as a test input which can be used by the developer to verify the error. The path conditions are usually expressed as a first-order logic formula, and solved using a Satisfiability Modulo Theories (SMT) Solver. For instance, consider the following simple program. Our goal is to figure out if any set of inputs \(x, y\) would result in an ERROR.

```
1. int foo(int x, int y){
2.   int z = x-10;
3.   if(z > y){
4.     if(2y > x){
5.       ERROR
6.   }
7.   }
8.   return z;
9. }
```

The symbolic executor runs through the code as shown in Fig 1. At each step, it maintains a heap \(H\) that stores the mapping between symbolic values and expressions over the concrete values \(x, y\) and \(z\), and updates the path condition \(P\). When it encounters the ERROR statement, it invokes the SMT solver with the path condition \(P_{4}\), which then tries to find a set of values of \(x\) and \(y\) that satisfies the condition. In this case, there exists such a possible set of values: \(x = 24\) and \(y = 10\). Hence, the symbolic executor could return such a pair of values as a test input that causes the function to fail.

1.2 Lazy Evaluation

Professor Piskac and her group have developed G2, a symbolic execution engine to analyze programs written in Haskell, a language with non-strict semantics.
Consider the following function application:

\[(\lambda x. (+ x x))(\ast 5 4)\]

There are various sequences by which this function can be reduced. Most programming languages are strict and employ an *eager* evaluation strategy, evaluating an expression as soon as it is bound to a variable.

\[(\lambda x. (+ x x))(\ast 5 4)\]
\[= \Rightarrow (\lambda x. (+ x x)) 20\]
\[\Rightarrow (+ 20 20)\]
\[\Rightarrow 40\]

In contrast, Haskell’s non-strict semantics enables expressions to have a value even if some of their sub-expressions do not. In practice, this is implemented with lazy evaluation, a strategy that comprises delaying the evaluation of an expression until its value is needed.

\[(\lambda x. (+ x x))(\ast 5 4)\]
\[= \Rightarrow (+ (\ast 5 4) (\ast 5 4))\]
\[\Rightarrow (+ 20 (\ast 5 4))\]
\[\Rightarrow (+ 20 20)\]
\[\Rightarrow 40\]

Strategies used by symbolic execution tools that work on strict languages are usually not compatible with non-strict languages. They would result in spurious errors or a failure to feasibly complete symbolic execution. For example, consider the Haskell program below, which checks if the \(y + 1^{th}\) element in a list formed by repeatedly squaring \(x\) is even.

1. let squares a = a : squares (a*a)
2. x = ? y = ?
3. in assert (rem (squares x !! y) 2 == 0)

The length of the recursively-defined list at line 1 is infinite. Strict symbolic execution would first attempt to create the entire infinite list, missing a potential assertion violation in the case where \(x\) is odd. Under non-strict semantics, only a finite region of the list is actually evaluated, allowing for termination.
2 Proposed Work

2.1 Project

A major stumbling block in the use of G2 is the path explosion problem: in programs with conditional branches, loops, or recursion, the total number of states that need to be explored often increases exponentially. This slows down the analysis, and often makes symbolic analysis intractable.

Over the past summer, we have been attempting to address this problem by implementing state merging in G2. By merging states from different paths, we aim to combine different paths into one to avoid unnecessary execution of common code multiple times. State merging is a tricky endeavor, because different program paths encode different constraints about the symbolic variables, and it is essential to be able to recover such accumulated information for each individual path even after merging. While techniques to perform state merging have been proposed for strict languages, they require special modification to interact well with non-strict semantics.

To date, we have developed formal rules to describe the state merging operation, taking into account the operational semantics of the programming language. The rules take into account whether it is possible to merge states, and if it were possible, (i) how to merge the current program expressions at that point, (ii) how to merge the respective accumulated path conditions, and (iii), how to reconcile different values for the same symbolic variables. We also partially implemented state merging, though more work needs to be done to correctly integrate it into the control flow of the tool. Over the course of this semester, I aim to finish the implementation of state merging and its integration into G2.

2.2 Deliverables

2.2.1 Implement State Merging in G2

The symbolic values used and constraints detected at each point in the symbolic execution of a program is represented by G2 as a tuple of values called a symbolic state \( S \). It is of the form \((E, H, P)\), where the expression \( E \) corresponds to the term being evaluated, the heap \( H \) is a mapping from bound variables to terms, and the path constraint \( P \) is a logical formula representing the constraints on the symbolic values thus far in order for the execution to have reached that state.

We aim to finish implementing a state merging operation that takes states \( S_1 = (e_1, H_1, P_1) \) and \( S_2 = (e_2, H_2, P_2) \), and combines them into a new state \((e'_1, H'_1, P'_1)\), which encodes information from both \( S_1 \) and \( S_2 \) in a recoverable manner.

Secondly, we need to modify the order of evaluation of different program paths to integrate state merging into G2. There exists a trade-off between the time and space savings we could achieve by combining states from separate paths together, and the additional complexity of the constraints that the SMT solver would have to solve. Hence, there is a need to evaluate various heuristics about when to merge different states together and when to avoid doing so. Crucially, we need to account for the non-strict semantics when considering possible future states.

2.2.2 Evaluate Performance

We aim to evaluate the effectiveness of state merging by running G2 on two separate versions of a large independent benchmark of Haskell programs, one containing errors and the other without. We aim to measure the time taken for G2 to find differences between the error-free version and the buggy version, with and without state merging enabled.