Test Generation for Worst-Case Complexity

Anna L. Villani
Department of Computer Science, University of Colorado, Boulder, CO 80309, USA

I. Introduction

There are many ways a vulnerability in a program can be exposed and utilized by a hacker. One way is by using a single malicious input to trigger high-complexity behavior in a networked system, causing the server to exhaust its resources and be unavailable to other clients [3]. This is called a denial-of-service attack, which commonly exploits availability program vulnerabilities. Availability refers to the possibility of access to a resource that a given program offers. An attacker that exploits a weakness in a program, denying appropriate user access to it is compromising its availability. This paper aims to develop a strategy for generating test inputs for program paths that could potentially be vulnerable, and exposed to a denial-of-service attack. Since a single packet across a server can contain an "input of death" [4] causing the server to crash, it is important to observe what inputs of a particular program can cause high-complexity behavior. A good way to begin is to observe what inputs lead to the 'worst-case path' of a program. This paper presents an automated test generation technique using symbolic execution to generate inputs that result in the worst-case complexity of a given Java program. A naive symbolic test generation algorithm would enumerate all possible paths, generate a path constraint for each path, and solve each constraint to generate a test input for each possible path. However, this technique does not scale to large programs, and for denial-of-service attacks only the paths leading to high complexity behavior are important. The algorithm builds on the complexity testing technique, Worst-case Inputs from Symbolic Execution (WISE) [1] and uses Java Path Finder for performing symbolic execution [6].

II. Background

The algorithm presented in this paper builds on the complexity technique, Worst-case Inputs from Symbolic Execution (WISE). The WISE tool has achieved generating worst-case complexity tests for each possible input size of a program. The algorithm in this paper has, thus far, only achieved generating worst-case complexity inputs for programs of constant input size. The technique can be run on any Java program P. It is assumed that P is a sequence of labeled statements consisting of: an input statement, an assignment statement, a conditional statement, and an exit statement. Thus, the algorithm is not handling loops and a simple assumption can be made regarding the worst-case complexity of a program. For a given program P, an execution path is considered a worst-case execution path if it contains the maximum number of conditional statements contained by any feasible execution of P [1]. In the example in **Figure 1**, the worst-case execution path is lines 3-5 because that is the path that contains the maximum number of conditionals.

```
1 public class AssertionLifting {
2    public static void test(int x){
3        if(x>0) {
4            if(x<=5) {
5                System.out.println("assert violated "+Debug.getSolvedPC());
6            }
7            System.out.println("br2");
8        } else
9            System.out.println("br3");
10   }
11   public static void main(String[] args) {
12       test(0);
13   }
14}
```

Figure 1.

III. Overview

This section gives an informal overview of the algorithm using the example in **Figure 1**. This example is a simple Java program that asserts that the input is less than or equal to five and greater than zero. If the assert is violated it then calls some debug function. First, symbolic execution and constraint solving is used to generate test inputs. If the algorithm is run in this way, it will create a test input for each possible execution path in the program. This approach works fine or relatively small Java programs with constant input. However, it would not scale to large programs with variable input. The next step is to restrict the paths considered to be one(s) that result in worst-case complexity. To do this the algorithm keeps a count of the number of conditionals encountered in each possible execution path during symbolic execution, and matches it with its corresponding constraint. In the example presented in **Figure 1**, the constraint x_1_SYMT[1] <= CONST_ 5 && x_1_SYMT[1] > CONST_0 is matched with a count of size two. The constraints x_1_SYMT[1] > CONST_ 5 && x_1_SYMT[1] > CONST_0 and x_1_SYMT[1] <= CONST_0 are paired with counts of size one. The constraint with the largest count size is returned, and a test input is generated from that constraint. The idea is that for programs with variable size of input, this technique can be run on small input sizes to find the longest path. The constraint returned can be used to reduce the enumeration of paths on a large program with large input size.

```
1 public class AssertionLifting {
2    public static void test(int x){
3        if(x>0) {
4            if(x<=5) {
5                System.out.println("assert violated "+Debug.getSolvedPC());
6            }
7            System.out.println("br2");
8        } else
9            System.out.println("br3");
10   }
11   public static void main(String[] args) {
12       test(0);
13   }
14}
```

Figure 2.

IV. ALGORITHM

The algorithm presented in this paper uses the symbolic execution and constraint solving techniques to generate test inputs. Symbolic execution is a program analysis technique that uses symbolic values as program inputs and symbolic expressions to represent values of program variables. In particular, assignment statements are represented as functions of their symbolic arguments and conditionals are represented as constraints on symbolic values. The outputs of a program are expressed as a function of the symbolic inputs that exactly characterizes the inputs that cause the program to execute along the path it took during that iteration of symbolic execution. Symbolic execution maintains a symbolic state σ, which maps variables to symbolic expressions, and a symbolic path constraint PC, which is a quantifier-free first-order formula over symbolic expressions. At the beginning of a symbolic execution, σ is initialized to an empty map and PC is initialized to true [2]. The idea is that symbolic execution generalizes testing by allowing symbolic variables in evaluation. When an execution path depends on an unknown value the symbolic executor will fork, conceptually executing every possible path. All the execution paths of a program can be represented using a tree. For example, the program in **Figure 3** has five possible execution paths, which can be seen from the tree in **Figure 2**. When symbolic execution is run on the code from **Figure 3**, it starts with an empty state σ and PC = true. At every statement in the program that takes input, a mapping from that variable to a symbolic value is added to the state σ. After line is executed in **Figure 3**, σ = { a -> ao, b -> bo, c -> co}. At every conditional statement if (e) S1else S2, the PC is updated to PC ∧ σ(e) (for the "then" branch), and new PC' is created to be PC ∧ ¬ σ(e) (for the "else" branch). If both of these execution paths are satisfiable, then two instances of symbolic execution are created to execute the tow different paths. After line 3 is executed in the code, two instances of symbolic execution are created, σ1 with PC = a and σ2 with PC' = ¬a. When line 6 is executed, two more instances of created. The resulting states are σ1 with PC = a && b < 5, σ2 with PC' = ¬a && b < 5, σ3 with PC'' = a && b >=5, and σ4 with PC''' = ¬a && b >=5. When line 7 is executed, new instances are created only from the states in which b < 5: σ1 with PC = a && b < 5, σ2 with PC' = ¬a && b < 5. Since ¬a && c is unsatisfiable within state σ1, it can be ignored. Thus, only two new instances are added, σ21 with PC' = ¬a && b < 5 && c and σ22 with PC'''' = ¬a && b < 5 && ¬c. To generate test inputs, a satisfying assignment to the symbolic path constraint is generated using an off- the-shelf SMT solver [2].

V. GENERATING WORST-CASE TEST INPUTS

In order to generate tests that only result in worst-case complexity, restrictions must be added to the symbolic executer so that only worst-case paths are symbolically executed. To restrict the paths executed during symbolic execution, a class of functions called generators are used. Generators are formally defined to be functions mapping ever prefix of an execution path to either true or false. A generator G an execution path or path prefix σ, if G(σ) is true [1]. A generator function can be defined in many different ways, one class of generators are called branch generators. Branch generators determine which branches a particular program path can contain. Based on the assumption that worst-case execution paths are the paths with the most conditionals, this seems like a good means for pruning the path space. In order to use generators some kind of symbolic execution must be performed on the program in question so that the generator function can 'learn' something about the program. In this case, the function must know which execution path contains the maximum number of conditionals. To achieve this a modified version of the symbolic execution is used. A counter variable is created at the beginning of the execution. Each time a conditional is reached, new counters are created for each possible branch path. Each counter is paired with its corresponding path constraint, so there is one counter and one path constraint for each possible path in the program. The path constraint paired with the biggest count is returned, the constraint leading to the worst-case complexity of the program. The idea is that this technique can be applied to larger, more complex programs by first symbolically executing the program on small input sizes. The modified version of symbolic execution can return a constraint for the worst-case path, which can be used to generate test inputs for larger programs with large inputs.

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

Figure 3.

VI. CONCLUSION AND FURTHER WORK

The algorithm implemented here is able to generate test inputs that result in worst-case complexity for simple, relatively

small Java programs. One challenge that has been observed so far with this implementation is when it is run on a program with multiple paths that have the same number of conditionals contained in the path. When this situation is encountered, the algorithm currently will consider and generate test inputs for both paths. A more sophisticated approach for determining worst-case complexity in programs would allow for this algorithm to run on more complex and real programs.
Right now the technique is designed to run on programs with a constant input size. In other words, there is a predetermined number of inputs the program can take and it cannot be changed. A big and important next step is to design the algorithm to accept programs that can
take an arbitrary number of inputs. Thus, the number of inputs the program takes is not constant, but probably depends on something else. If this can be achieved, then pruning the path space of a program will be a useful for generating tests for programs of large input sizes.

REFERENCES

[1] Burnim, Jacob, Sudeep Juvekar, and Koushik Sen. "WISE: Automated test generation for worst-case complexity." Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE, 2009.

[2] Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." Communications of the ACM 56.2 (2013): 82-90.

[3] Chang, Richard, et al. "Inputs of coma: Static detection of denial-of-service vulnerabilities." Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE. IEEE, 2009.

[4] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically generating inputs of death," in CCS, 2006.

[5] Godefroid, Patrice. "Higher-order test generation." ACM SIGPLAN Notices. Vol. 46. No. 6. ACM, 2011.

[6] Java Path Finder. http://babelfish.arc.nasa.gov/trac/jpf/wiki

[7] Wilhelm, Reinhard, et al. "The worst-case execution-time problem—overview of methods and survey of tools." ACM Transactions on Embedded Computing Systems (TECS) 7.3 (2008):36.