Symbolic Execution for Java executables using angr

Sebastiano Mariani, Sydney Ma

Runtime System Spring 2018

Abstract

Nowadays, Java is still a very popular programming language because it is widely used in Android systems and web servers. Most of these applications need to deal with the user inputs provided in a string format. In most of the cases, the user input needs to be validated or sanitized, but it is hard to prove if these routines work correctly or not. The problem of understanding if a string might lead to a vulnerability or not is known as "string analysis". In the past few years, popular constraints solver, such as z3 (1) from Microsoft, managed to solve the problem. These solvers require constraints to be provided explicitly, however, it is not practical to do so because most of the applications are closed source and using reverse engineering to extract those constrains is time consuming and sometimes hard. In this project, we built a tool, that given an apk or a jar file, it can correctly identify string manipulation routines and correctly extracts constraints imposed on those strings.

1. Introduction

When looking for bugs or vulnerabilities in a program, the source code is not available most of the time, and so the analyzer needs to manually do the reverse engineering, which can be hard and time consuming. Another way that the analyst might do to find bugs is to try a lot of different inputs and hopefully find a particular one which would crash the program. This technique is known in the literature as **fuzzing**. The main limitation of this technique is that the space of all possible inputs is sometimes huge and it is very unlikely to find one which can crash the program. Symbolic execution tries to solve this problem in a clever way: inputs are no more considered as a series of concrete values but rather be assumed as symbolic values, and modifications to these values are collected in form of constraints. A constraint can be seen as an equation where each variable is a symbolic value representing an input. In other words, symbolic execution is a method of analyzing a program to determine what inputs cause each part of the program to execute.

1.1. Example of symbolic execution

The whole process is better explained step by step with the code below.

```
1 void foobar(int a) {
2     int x = 4, y = 0;
3     if(a != 0) {
4         y = a + 3;
5     }
6     assert(x-y != 0);
7 }
```

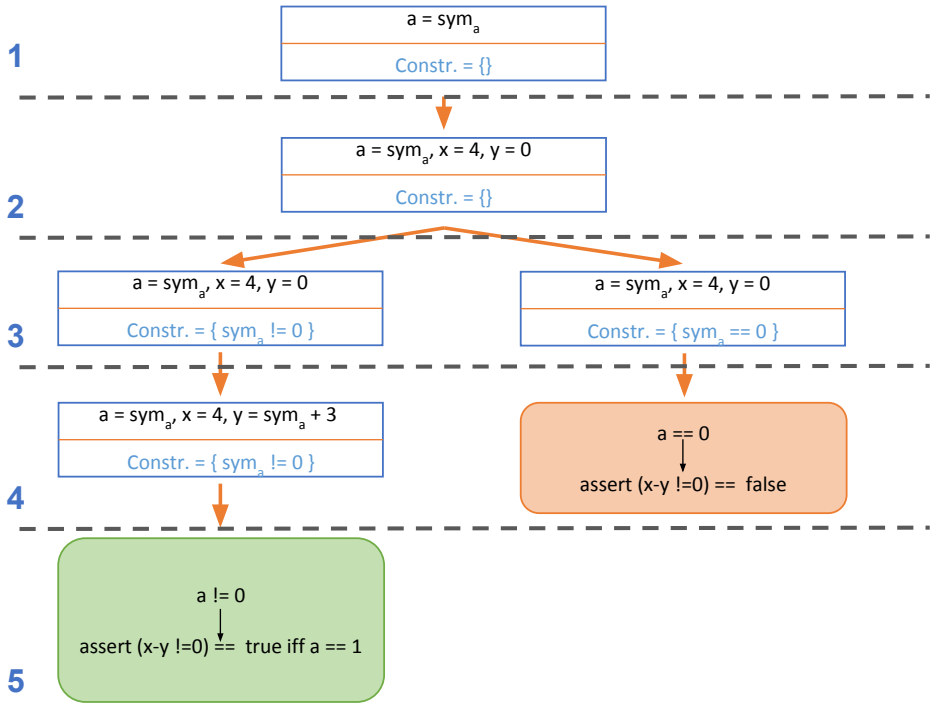

Figure 1: Simple example of symbolic execution flow

When this program gets executed symbolically, the flow is illustrated in Figure 1.

The key component of symbolic execution is the **state of the program** and it is identified by the following properties:

- **Variables** : Every symbolic or concrete value found during the symbolic execution of the program.
- **Constraints** : Conditions on symbolic variables which need to be satisfied in order to reach that specific state.

The execution proceeds in parallel in all possible paths. In this example, every step is labeled from 1 to 5.

1. At the initial state, the constraints are empty since we are at the beginning of the execution, and the parameter of the function (int a) is treated as a symbolic value.
2. At this step, two variables x and y are added to the program state as concrete values.
3. Here we have the first fork in the execution flow. The left branch requires the condition a!=0 to be satisfied, while the right branch requires a==0.
4. The execution proceeds in parallel. In the left branch, the concrete value of y is updated. In the right branch, the assert is evaluated. Given our constraints on the input, we see that it is impossible for the assert to be true.
5. At last, the assert is evaluated in the left branch. The assert is evaluated to be true if and only if the input value a is equal to 1.

2. Building blocks

This project is entirely based on angr [1], one of the state of art frameworks for symbolic execution nowadays. It provides a barebone infrastructure to manage program states and collect constrains during the analysis of binaries. Unfortunately, angr deals with binaries, which uses register based machines, however, java bytecode uses stack based machines. This problem is directly solved by our second main component of the project: Soot [2]. Soot is a java optimization framework, which lifts java bytecode to an intermediate representation, called Soot IR, and then performs various kinds of optimizations on it. Soot provides four different intermediate representations:

- **Baf**: A streamlined representation of bytecode which is simple to manipulate.
- **Jimple**: A typed 3-address intermediate representation suitable for optimization.
- **Shimple**: An SSA variation of Jimple.
- **Grimp**: An aggregated version of Jimple suitable for decompilation and code inspection.

The intermediate representation used in this project is shimple. It addresses the problem of different types of machines directly when lifting (i.e. the lifted intermediate representation uses a register based machine), and on top of that, it guarantees SSA property within methods, making the process of updating states easier during symbolic execution.

The only problem with this solution is that Soot is written in java and angr is written in python. To integrate this two different paradigms, we use another component called pysoot. Pysoot is a wrapper to all the functions exposed by Soot. It uses jython, which enables python code to call java directly. The workflow of our project is summarized in Figure 2.

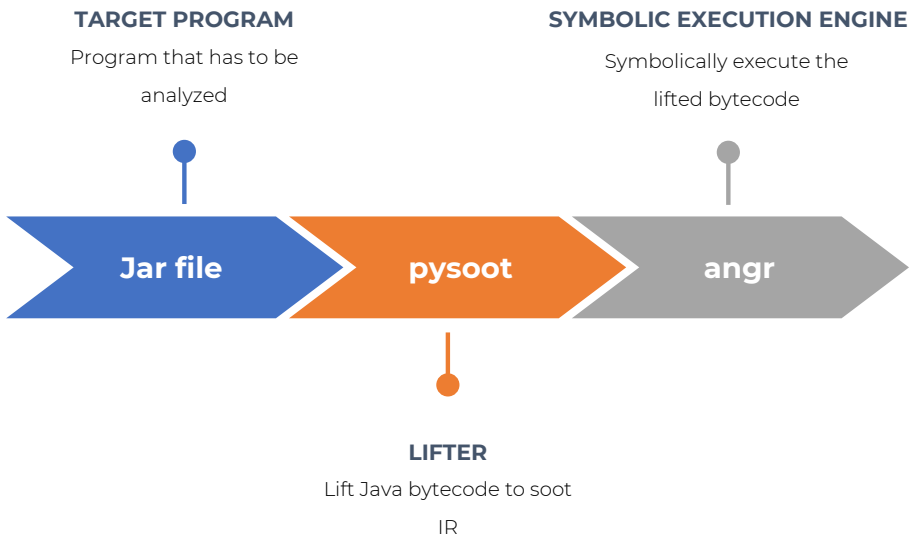

Figure 2: Project workflow

3. Implementation details

The symbolic execution engine proceeds as follows:

1. The java bytecode that has to be analyzed is passed to angr
2. The first basic block is lifted using pysoot
3. All the instructions in the basic block are symbolically executed and the program state is updated accordingly
4. The next basic block is lifted. And step 3 is repeated until the program ends

A program state is uniquely identified by three main properties: Stack, Heap and Static table. The stack keeps track of all the stack frames of the functions currently active. The heap keeps track of all the dynamically allocated objects and the static table contains all the static fields of all the classes currently initialized.

For every soot IR instruction, there is a dedicated handler associated with it. These python functions emulate the semantics of the instructions, and update the state of the program at that given point. For example, if the executed instruction is to store a SootStaticFieldref, some new dedicated space on the static table is created and the reference to the space will be returned.

Library calls is another common problem in symbolic execution. It might be solved in two ways. The first solution is to include all the library codes in the analysis and symbolically execute them. However, this solution might lead to states explosion. The second solution is to use summarization. Summarization is a technique that tries to reflect the side effects of a function (i.e. the modification to the program state) without analyzing it. Angr supports both of the solutions and for the second soltuion, it provides an interface called SimProcedure, which allows writing plug-in summarized functions. In our project we chose to summarize a couple of functions from the library java.lang, targeting specifically the ones which deal with strings and user inputs.

Other implementation details such as virtual dispatcher, class hierarchy support, and etc, can be found in our Github repositories.

4. Limitations and Future work

There are several limitations regarding the work. We will briefly illustrate two here using two simple examples.

4.1. Array with symbolic indexes

The program below is intended to assign a value to the variable ele. The value to be assigned is retrieved from an array with one million elements and the index is read from the standard input.

```
1 public static void main(String[] args) {
2     Scanner scanner = new Scanner(System.in);
3     int index = scanner.nextInt();
4     int[] arr = {0,1,2,    , 1000000};
5     int ele = arr[index];
6 }
```

The problem is that the value of the index is not known and could be any possible integers. In this case we don't know which element to retrieve, and therefore we don't know which value the variable ele will be assigned. One possible solution would be to generate all the possible states, where in each of them, ele assumes one of the values of the array. But this would lead to **state explosion**.

4.2. Inheritance with symbolic types

The program below is intended to call a function called foo. There are multiple subclasses inheriting form superclass A, as shown in Figure 3. In each of the subclass, the method foo is overwritten.

```
1 public static void foo(A param) {
2     int res = param.foo();
3 }
```

The problem is that the variable param can assume any subclass of A as an dynamic type. When calling the function foo, we don't know which foo is called since all of the subclasses have the function implemented.

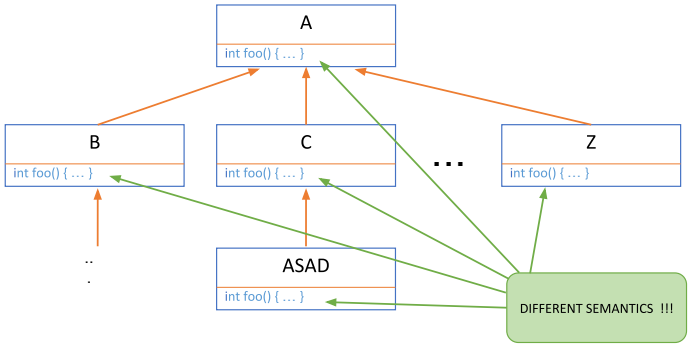

Figure 3: Class hierarchy of A

5. Conclusion

The goal for this project is to build a tool that, given an apk or a jar file, it can correctly identify string manipulation routines and correctly extracts constraints imposed on those strings. Our tool is built on top of two frameworks: soot and angr. And the tool is capable of lifting the Jave bytocode to an intermediate representation and use angr to perform symbolic execution on the newly created IR. Of course, there are still a lot of work that needs to be done, but this project is a good start.

[1] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[2] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.