Implementing the Query Count Estimator from 'Efficient State Merging in Symbolic Execution'

Bachelor Project - Spring 2023

Adrien Alain Bouquet

Introduction

```
- Presentation du projet initial
- What is a query
- "" Du langage
- "" De LLVM (particulièrement llvm assembly)
- "" De klee
```

Definition of the problem

```
- Comment comprendre la théorie
- Explication des problèmes rencontrer
    - Branch conditional
    - Load and store.
    - For loop
    - Recursion and Call
```

The central point of this report is the implementation of the heuristic query count estimation (QCE), the formula for which is described on page 5 of the paper. It is essential to explain the purpose of this heuristic in order to understand the implementation described below: In the context of symbolic execution, one of the objectives is to reduce the computational cost that the introduction of symbolic variables can generate so that we increase its efficiency. To do this, the paper proposes this QCE heuristic whose aim is to determine whether a variable v at a location l in the program is a sensitive variable (or a "hot variable" as denoted in the paper). As described in section 3.2, a variable v is sensible (or hot) at location l if the number of additional queries $Q_{add}(l, v)$ is greater than a fixed fraction α of the total number of queries $Q_t(l)$ (see §3.2). Consequently, the

paper presents a formula to count recursively the number queries that are selected by a function c:

$$q(l', c) = \begin{cases} \beta q(succ(l'), c) + \beta q(l'', c) + c(l', v) & \text{instr(l') = if (e) go to l" (1.a)} \\ 0 & \text{instr(l') = halt (1.b)} \\ q(succ(l', c)) & \text{otherwise (1.c)} \end{cases} \quad (1)$$

Where e is an expression and β represents the probability that a branch is feasible. The paper fixes it to 0.6 as example. Therefore, it employs this formula to define the two numbers of queries defined above by:

$$Q_{add}(l, v) = q(l, \lambda(l', e).ite((l, v) \triangleleft (l', e), 1, 0)) \quad (2)$$

$$Q_t(l, v) = q(l, \lambda(l', e).1) \quad (3)$$

Presented in this way, the formulas do not allow us to understand clearly how to implement the LLVM pass and it is necessary to explain them. First of all, a program location l can be understood as an instruction. Thus l' can be seen as next instruction (the next line in terms of code) and $succ(l)$ as the successor of l. Then, (1) separates 3 cases which return a value. The first one $(1.a)$ is a branch condition with an expression (an if-else for instance), the second one $(1.b)$ is a program termination and the last one $(1.c)$ is all the other type of instructions. Hence, $(1.a)$ evaluates recursively the branch if e is true and the branch if e is false. It also evaluates if the condition e depends on the variable v and therefore will need a query. The two last cases are pretty straight forward: repectively return 0 if the program ends and return the value of the next instruction. In summary,

Implementation

- Comment passer de la formule de la page 5 à un llvm pass
  - First llvm pass
  - Chosen Parameters
  - static analyze function
  - match value function
  - block come before function

Results

```
- Tested case (does it actually work ?)
- Does it solve the issues
- case from the paper
```

- Case on a more complex code
- Does it work on Coreutils
    - Why coreutils ?

Limitations

-

Personnal discoveries

Conclusion