Implementing the Query Count Estimation 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

The central point of this report is the implementation of the heuristic query count estimation (QCE) described in the first part 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, c is a function given in parameter that evaluates the expression e 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:

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

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

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, l' can be seen as the 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 the condition using the function c. 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, this formula can be represented as if we were walking through a tree. When reaching a condition, evaluate the condition using the given function and recursively evaluates the 2 branch. When reaching an other instruction, skip it, When reaching an end, it's a leaf.

As a result, Q_{add} becomes the evaluation of the formula when the function c check if the expression e depends on the variable v and hence will generate a solver query and Q_t becomes the evaluation of the formula when c returns always 1 as it simply counts the number of branch condition.

However, this explanation is not sufficient in itself, as certain instructions are problematic as explained in the paper. Firstly, loops and recursion prevent the formula from stopping because of the recursive nature of its definition. In the paper, a loop bound κ is set to limit the maximum loop iterations and recursion calls. In addition, functions calls are problematic because they must be evaluated too and their number of queries must be added to the total one. Therefore, the next part below will explain the implementation of the LLVM pass with respect to the formulas and constraints stated before.

Implementation

```
- match value function
- the pass in itself
```

Before describing the code in concrete terms, it is important to define the different variables used in the theory in terms of code, and more specifically in terms of LLVM assembly or C++. The equivalent of an instruction l, as explained above, is the `Instruction` class, a variable v as the paper intended will be have for equivalent the `Value` class, the

function q has for implementation the `compute_query` function described below and the function c has for implementation either the `returnTrueFunction` or the `matchValue` also decribed below.

Parameters and Data structure created

The value for β which represents the probability of the feasibility of a branch is fixed to 0.6 and the maximum number of iterations κ is separated into two variables `nloops`and `dcalls` both fixed to `1`. Furthermore, the α parameter which is used to compute if a variable is "hot" is fixed to 0.5. Theses values were chosen from the example at the end of page 5 to facilitate to facilitate comparison with the pass defined in the paper and the tests.

Besides, two data structures were created: `BranchCount` and `CallCount`. The former associates a branch instruction (i.e. `BranchInst` in LLVM assembly) to the number of time the instruction has been encountered, the latter is analog to the former one but it counts the number of times a function (i.e. `Function`) is encountered. Basically, they are used to keep track of the loop iterations and recursions.

Compute Query function

As stated before, the `compute_query` function is the equivalent of $q(l, c)$ which computes the number of query. It takes as argument a pointer to a llvm instruction `Inst` (i.e l), a pointer to a llvm value `researched` (i.e. v), the list of all the `BranchCount` of the backwards branch, in other words all the branch that jump to a label prior to the current one, the list of all the `CallCount` of all the `CallInst` that depends on the value `researched` and finally a function (i.e c) that evaluates if the condition of the branch instruction depends on the value `researched`. The function applies what has been previously defined and handles the constraint such as loops iterations or recursion by checking if the maximum count has been reached.

Match Value Function

The implementation of the function c that check if an expression depends on a particular variable is `matchValue` which takes as argument two llvm value: the first one correspond to the value we are testing and the second one is the value that we are searching (i.e. the variable v in the paper). Although its goal is pretty straightforward, its implementation is more complex as it mights loops. For instance:

```
void loopfunc(int x){
    int index = x;
```

```
    while (index < 10){

        //...//
        index++;
    }
}
```

Results

- Tested case (does it actually work ?)
- case from the paper
- Case on a more complex code
- Does it work on Coreutils
    - Why coreutils ?

Limitations

-

Personnal discoveries

Conclusion