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

Bachelor Project - Spring 2023

Adrien Alain Bouquet

Introduction

When you're trying to test code, symbolic execution is an effective way of doing this by automatically generating tests and finding bugs. However, its cost in terms of performance can be heavy due to path explosion and this state explosion. To counter this problem, one approach is to merge states obtained on different paths in order to reduce the total exploration cost. However, merging two states can potentially increase costs and therefore decrease performance rather than the opposite. In order to determine which states to merge or not, researchers at the Dependable Systems Lab (DSLAB) at the Ecole Polytechnique Fédéral de Lausanne (EPFL) have published "Efficient State Merging in Symbolic Execution" in 2012, in which they present two methods for reducing costs of state merging. The first one is a query count estimation which computes the impact of each symbolic variable on solver queries and indicates which one provides benefits to merge. The second one is dynamic state merging which "interacts favorably with search strategies in automated test case generation and bug finding tools".[1] In this report, we will be working on an implementation of the query count estimation based on its description. The goal of the report is first to explain the implementation of the pass chosen to be as close as possible to that described in the paper, and then to test its effectiveness and robustness on different examples and the `GNU COREUTILS`.

Definition of the problem

The central point of this report is the implementation of the heuristic query count estimation (QCE)[2] It is essential to explain the purpose of this heuristic in order to understand the implementation described below: In the context of symbolic execution,

[1] page 1 from the paper
[2] Described in the first part of the paper

one of the objectives is to reduce the computational cost that the introduction of symbolic variables can generate to 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 sensitive (or hot) at location ℓ if the number of additional queries $Q_{add}(\ell, v)$ is greater than a fixed ratio α of the total number of queries $Q_t(\ell)$ (see §3.2). Consequently, the paper presents a formula to count recursively the number of queries that are selected by a function c:

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

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 for example. Therefore, it employs this formula to define the two numbers of queries defined above by:

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

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

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, 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 straightforward: respectively 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 checks 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 recursive 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 an LLVM pass with respect to the formulas and constraints stated before.

Implementation

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 loop iterations and recursions.

Compute Query function

As stated before, the `compute_query` function is the equivalent of $q(\ell, c)$ which computes the number of queries. It takes as argument a pointer to an LLVM instruction `Inst` (i.e ℓ), a pointer to an LLVM value `researched` (i.e. v), the list of all the `BranchCount` of the backwards branch, in other words all the branches 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 checks 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. In fact, we have to backtrack in the code by going from the operands of an instruction, checking if they are the value we search and if not checking their operands until the variable searched is found or there is nore more instruction and hence the initial condition does not depends on the variable. However, this logic might causes `matchValue` to loop if they are loops or recursion. For instance:

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

    while (index < 10){
        //...//
        index++;
    }
}
```

When reaching the condition of the while loop, the index has been redefined and thus there are two paths leading to it. In LLVM, there exist `PHINode` which are instructions that regroup the differents paths of the variable, in other words the differents values and the label where they are defined.

The pass

The pass computes twice the query count of a function for each its argument: once using the `matchValue` function described above and once using the `returnTrueFunction` to count the total number of queries. It then outputs the value using both function and if the variable is a hot variable.

Results

Basic cases

In order to evaluate the correctness on basic code, a set of 21 cases[3] as been established to tests implementation and constraint handling, and we are going to demonstrate 3 of them. As a reminder, the probability β that a branch is feasible is fixed to 0.6 and the parameter κ which correspond to the maximum loop iterations and recursion depth is fixed to 1.

[3]https://github.com/abouquet27/BachelorProject/tree/main/project/cases

Basic condition

```
void basicCondition(int a){
  int x = 0;
  int b = 19;
  if (a >= 5) {
    x = 5;
  } else {
    x = 10;
  }
  int y = x + a;
}
```

This case is pretty straightforward when applying the pass as the variable tested a is in the condition of the if and hence `matchValue` will return 1. Furthermore, the evaluation of the two branch will return 0 each as there are not any conditions left. Thus, the query count will be equals to 1.0 which is the value we theoretically expect. Here is the full expectation computation (line 1 correspond to the function header):

$$\begin{aligned} Q_{add}(1, a) = q(1, c) = q(4, c) \\ = \beta q(7, c) + \beta q(5, c) + c(a >= 5, a) \\ = \beta q(10, c) + \beta q(10, c) + 1 = 0 + 0 + 1 = 1 \end{aligned}$$

Recursion

```
void recursionFunction(int x){
    if (x > 0){
        printf("recursion needed");
        recursionFunction1(x-1);
    }
}
```

The interesting aspect of this case is recursion. When evaluated the first time, `matchValue` will return 1 on the first condition, the `printf` call will be skipped as it does the x variable and it will evaluate a second time the condition:

$$\begin{aligned} Q_{add}(1, x) = q(1, c) = q(2, c) = \beta q(5, c) + \beta q(3, c) + c(x > 0, x) \\ = \beta q(6, c) + \beta q(1, c) + 1 = \beta(\beta q(5, c) + \beta q(3, c) + c(x > 0, x)) + 1 \\ = \beta(0 + 0 + 1) + 1 = 0.6 + 1 = 1.6 \end{aligned}$$

More generally, the different base cases have all been computed manually before being submitted to the pass and comparing the results. About twenty base cases have been established, computed and tested to make sure the pass correctly work on these cases.

Double while loop

```
void doublewhileloopfunction(int* tab, int size) {
    int i = 0;
    int j = 0;
    while (i < size) {
        while (j < size) {
            tab[i] = tab[i] + tab[j];
            j++;
        }
        i++;
    }
}
```

The case is rather interesting because the functions has 2 arguments and there is a double while loop. The result of evaluating the function `compute_query` on `tab` will be equals to 0 as the argument is not part of any condition in the program. However, the argument `size` will have 1.6 has result because it is inside two conditions.

Example from the paper

After creating cases to test the different issues the pass has to handle, the next step was testing it on the example from the paper[4]. The paper uses this simplified version of the `echo` program. However, the example has been truncated to make it work using the pass, mainly by adding the variable 'arg' in the arguments of the function. Consequently, the pass computes 1.6 which is the same value as the one computed by the paper and hence slightly strengthens the correctness of the pass

Applying the pass on `COREUTILS`

The next step in evaluating the pass is to test its robustness, i.e. its ability not to crash when applied to programmes that are much more complex than the examples shown above. To this end, the pass was applied to 30 `COREUTILS` of different purposes and sizes. They were chosen for 2 reasons: firstly, because they are programmes rich in complexity and variety, and secondly, because the paper also uses them to carry out its evaluations.

Out of 30 coreutils tested, 21 worked (i.e. the pass worked on all the functions of the coreutils tested and did not crash), 7 crashed (i.e. the pass did not finish and an error was thrown) and 2 did not terminate. The results is rather satisfying as it is 70% of the `COREUTILS` tested that works. Regarding those who does not terminate, the main explanation is the increasing complexity of the code. In fact, the more complex code becomes (by using function calls, recursion, etc), the more time it will need as the number

[4]Figure 1 page 5

of exploration path can increase exponentially. About those who crashed, the error thrown is `Illegal Instruction: 4` which is an issue with compile flags and might not happen on other OS.

Limitations

Although the pass works on a number of cases including some complex ones, it still has flaws, for instance hardly pointers to a function that is not constant or switch case that are not taken in account even though they are condition. As a side note, the former is not treated by Klee because it is difficult to analyze a function and the pass could do the same. Furthermore, the correctness of pass has not been proved and hence its application might generate wrong results.

On the other hand, the pass could still be improved on its efficiency. Indeed, the pass enters and analyzes every function that are being called instead of computing them once as the paper planned to do. Thus, the cost of the computation is really heavy and not very useful as function that have already being called will be analyzed again. Moreover, function from external library such as `printf` initally caused the program to crash and hence have been skipped to facilitate the computation. However, this assumption of skipping function of external libraries can be discussed: on the one hand, it alters the final result but on the other hand the impact on it will become less significant as the size of the code and therefore the depth of the analysis increases. Losing some millionths may be a good way of improving efficiency.

Finally, some existing LLVM pass might exist to make the code easier to analyse. For instance, one major issue where the load and store because it was costly when encountering a load to trackback until finding the corresponding store. Hopefully, we did find the pass `mem2reg` which allows us to get rid of the majority of the stores and loads and thus to lower the cost.

Conclusion

In this report, we have presented the implementation of the query count estimation (QCE) heuristic from the paper "Efficient State Merging in Symbolic Execution" in the form of an LLVM pass. We explained the purpose of the heuristic and its formulas, and then provided a detailed description of the implementation. We also discussed the results of testing the pass on basic cases, the example from the paper, and a set of coreutils programs.

Overall, the implementation of the pass closely follows the description in the paper, and it successfully computes the query count for the given examples. The pass demonstrates its effectiveness in identifying hot variables and providing insights into the potential performance impact of state merging in symbolic execution. However, it is important to note that the pass has limitations and may not work on all programs. It may crash or

fail to terminate on certain complex programs. Nevertheless, the pass shows promise and can serve as a starting point for future research.

In conclusion, this project contributes to a better understanding of the query count estimation heuristic and provides a practical implementation that can be used to analyze and optimize symbolic execution techniques.

Personnal discoveries

This project has been a great learning experience on a large range of aspects. To start, it was my first big personal project in computer science and at EPFL. It required a lot of autonomy and a capacity of managing the hours for a result that is closer to reality than exercices seen during courses.

Speaking of courses, this project and working with LLVM was a good application of concepts seen during courses as Compilers or a good introduction to new concept such as Symbolic Executions. Another interesting point was the notion of research. Working on a project based on a paper from the laboratory in question is very motivating and stimulating because it requires you to do research and understand more than just what you need to implement.

Finally the main challenge was working with new languages with their particularities, more specific libraries with tougher documentation requiring more effort to understand how to use them or tools that requires a lot attempts to install and make them work. It took me a while to understand how does LLVM work and to use in my code to achieve the pass. The project required a lot of adaptation, but it developed cross-disciplinary skills that I'm sure will come in very useful later on.

References

- « Getting started with writing LLVM passes - Sebastian Österlund ». https://osterlund.xyz/posts/2017-11-28-LLVM-pass.html.
- GitHub. « Coreutils/Src at Master · Coreutils/Coreutils ». https://github.com/coreutils/coreutils.
- KLEE http://klee.github.io/.
- Kuznetsov, Volodymyr, Johannes Kinder, Stefan Bucur, et George Candea. « Efficient State Merging in Symbolic Execution », s. d.
- « The LLVM Compiler Infrastructure Project ». https://llvm.org/.