Technische Universität München

Fakultät für Informatik
Lehrstuhl 22
Software Engineering
Prof. Dr. Alexander Pretschner

Supervisor:
Saahil Ognawala
(ognawala@in.tum.de)

Boltzmannstraße 3, 85748
Garching bei München

Tel: +49 89 289 17386

Web: www22.in.tum.de

Constraint size thresholding in symbolic execution for broader path coverage

Master Thesis

[Skills involved: C/C++, Python, elementary machine learning and statistics]

Symbolic execution [1] is a powerful way to analyze programs by executing them using "symbolic" values instead of concrete values, such that all the possible program paths may be traversed, theoretically. Most symbolic execution tools achieve this by finding solutions for predicates, or "constraints", that represent the execution tree nodes. As such, this is an efficient way to automatically generate test cases that execute different paths of the programs. As a by-product of executing different paths, symbolic execution also helps us in finding those specific program paths that lead to vulnerabilities (overflows, code execution, memory corruption etc.) [2].

However, due to the complex nature of modern large-scale applications, the execution trees typically grow very large and the problem of finding *all* paths from start to end becomes undecidable [3] (*path-explosion problem*). Moreover, symbolic execution engines rely heavily on constraint solvers to return satisfiability and counter-examples for these path constraints (*constraint solving problem*). We would only focus our efforts on the constraint solving problem in this thesis. By applying a dynamic threshold to the size of path constraints in symbolic execution, we might be able to increase coverage of those branches whose guard conditions are "easier" for the constraint solver. One benefit of this approach would be that the new test-suite with higher branch coverage may be, then, used by helper techniques (such as *fuzzing* [4]) that are more effective at deeper branch coverage than symbolic execution.

Goal

The overall goal of this thesis would be to learn trends that find program specific indicators that lead to performance bottlenecks in modern-day constraint solvers.

This entails many sub-goals –

- Perform a state-of-the-art study in SMT solvers in the context of symbolic execution.
- Find representation features of path constraints, w.r.t. symbolic inputs, degree of the constraint formulae, domain ranges of involved datatypes etc.
- Using the above features, create linear regression models to preempt path constraints before they grow too large. This may include finding correlations between path constraints' features and time taken for the SMT solver to return.
- Compare the effectiveness (in terms of branch coverage and path coverage) of the implemented method to basic symbolic execution without constraint size thresholding.

Workplan

In order to fulfill the above goals, your tasks will include the following –

1. Develop an understanding of SMT-solver-based program analysis techniques
    a. Study symbolic execution and most common static analysis techniques.
    b. Compile a state-of-the-art report in usage of SMT solvers in symbolic execution engines.

Fakultät für Informatik
Lehrstuhl 22
Software Engineering
Prof. Dr. Alexander Pretschner

Supervisor:
Saahil Ognawala
(ognawala@in.tum.de)

Boltzmannstraße 3, 85748
Garching bei München

Tel: +49 89 289 17386

Web: www22.in.tum.de

2. Develop understanding of KLEE concolic (*conc*rete+symb*olic*) execution engine [5].
    a. Using small programs, learn the internal working of KLEE.
    b. Understand how KLEE interacts with symbolic environment and symbolic user-inputs.
    c. Understand how KLEE converts branching conditions to SMT queries, including
        i. KLEE-specific internal representation of path constraints
        ii. Representation of path constraints as queries issued to an SMT solver, e.g. Z3 [6].
    d. Understand how KLEE caches and/or subsumes some SMT queries, instead of issuing them to the SMT solver.
3. Use existing benchmark C programs to create supervised learning dataset of raw SMT queries issued to the solver and the time taken to return a SAT solution.
4. Using the above dataset, implement a predictive model.
    a. Using C language source code and raw SMT queries, manually engineer features of path constraints.
    b. Use the above features to train a regression model on the time taken to return satisfiability value (SAT or UNSAT) for an SMT query.
5. Implement dynamic constraint size thresholding for SMT queries issued by KLEE.
    a. Using the prediction from step 4, assign a *dropout probability*, P_d, to queries issued to SMT solver.
    b. With the calculated probability, P_d, pre-empt the query before completion of the path, and issue to SMT solver to generate counter-example.
6. Evaluate the performance of KLEE with dropout probabilities against basic KLEE, in terms of path and branch coverage.
7. Write a thesis document describing all the above steps, with clear and concise ideas, methodologies and results.

We are looking for a student that

- Likes to hack on existing C-like codebases,
- Is pro-active in coming up with design ideas before implementation,
- Has good programming skills in C/C++, and
- Has a basic knowledge of Python and machine learning.

We are interested in highly motivated students for this study, who will help further our research in the field of automatic analysis of programs for security vulnerabilities. We expect the student to produce high-quality deliverables, both in terms of scientific and engineering contributions.

References

[1] King, J. C. *Symbolic execution and program testing*
[2] Cadar, C. et al. *EXE: Automatically generating inputs of death*
[3] https://en.wikipedia.org/wiki/Halting_problem
[4] Sutton, M. et al. *Fuzzing: Brute force vulnerability discovery*
[5] Cadar, C. et al. *KLEE: Unassissted and automatic generation of high-coverage tests for complex systems programs*
[6] Moura, L. et al. *Z3: An efficient SMT solver*