Technische Universität München

Testing against a verified oracle: using Symbolic Execution and Fuzzing

HiWi / Bachelor Thesis

[Skills involved: C/C++, OCaml, Administration, (Isabelle, KLEE, AFL)]

Fakultät für Informatik
Lehrstuhl 21
Logik und Verifikation
Prof. Nipkow

Supervisor:
Maximilian Haslbeck
(haslbema@in.tum.de)

Boltzmannstraße 3, 85748
Garching bei München

Tel: +49 89 289 17315

Web: www21.in.tum.de

Context

Automatic ***testing*** is an important measure to ensure software quality. While testing can only uncover bugs, it can not ensure correctness on all possible scenarios because of combinatorial explosion – there are just too many sets of input to test! Also it is not at all clear what it means that a program is correct in the first place. Detecting obvious errors (e.g. buffer overflows and null pointer exceptions) is straightforward, but ensuring functional correctness is not – how do you decide that a program emits the correct output to the test input you provided?

Verification on the other hand is used to prove that some piece of software is correct on all possible inputs. The drawback is that software verification today still is too costly and laborious and does not yet scale to large software projects. Furthermore simple algorithms are simple to verify, but the more optimized software is the more complex the verification gets.

This project aims to ***bring together*** these ***two worlds*** by using already verified reference implementations as a *test oracle* for testing real-world software.

There are many algorithms already formally verified in the interactive theorem prover Isabelle [1] and it is possible to extract executable code from these formalizations. Some examples are sorting algorithms, Network-Flow algorithms [2], SAT-solvers [3] and a verified timed automaton model checker [4].

The starting point is the set of algorithms that are already formally verified in Isabelle, allowing us to construct a verified test oracle. Collecting real-world implementations for those algorithms and gathering available benchmark test-suites for them is the first step to take. Using these, one can already test the different implementations and compare results. If they differ, a potential bug has been found. Not only the emitted results but also run-time and other resource usage may be of interest, possibly uncovering run-time bugs.

A next step would be to generate new sets of test input and use them in the above methodology. Symbolic Execution [5] could be used on the gathered implementations to generate small sets of new interesting test cases, while Fuzzing [6] may be used in order to efficiently generate a larger set of inputs by random mutation.

One concrete example is the DBM Library [7] that is employed by the widely used timed automaton model checker UUPAAL [8]. There is already a verified DBM Library in Isabelle and “an extensive test suite” for UPAAL’s DBM Library, which in turn is implemented in C/C++.

Goal

The overall idea of this thesis is to use verified algorithms as a test oracle for testing real-world software. Building up infrastructure to systematically test software and conducting the technique in one use case are the main goals.

This entails many sub-goals:

- Reviewing formalizations of efficient algorithms in Isabelle and extract executable code from them
- Find real-world implementations of these algorithms and prepare executable code for them
- Collect available benchmark suites and sets of test cases
- Implement and deploy infrastructure to conduct the experiments in a structured way
- Document the usage of this infrastructure, to be easily usable for further use-cases

- Extract, document and report bugs that were found
- Use Symbolic Execution and Fuzzing to generate additional test cases

Workplan

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

1. Familiarize with Isabelle, and get an overview of the formally verified algorithms
    a. Get to know Isabelle, Verification Frameworks and code export facilities.
    b. Collect interesting algorithms and their formalizations.
    c. Extract executable code and compile it.
2. Collect real-world algorithms and available test suits
    a. Find (open-source) implementations for suitable algorithms.
    b. Summon available test suits and benchmark sets for these algorithms.
3. Create infrastructure to carry out structured experiments
4. Decide for one use case and carry out experiments
5. Evaluate and document the findings.
6. Write a thesis document describing all the above steps, with clear and concise ideas, methodologies and results.
7. Optional: generate new test cases
    a. Use KLEE [8] as a symbolic execution engine to generate interesting test cases.
    b. Employ AFL [9] for Fuzzing new random test cases.

Fakultät für Informatik
Lehrstuhl 21
Logik und Verifikation
Prof. Nipkow

Supervisor:
Maximilian Haslbeck
(haslbema@in.tum.de)

Boltzmannstraße 3, 85748
Garching bei München

Tel: +49 89 289 17315

Web: www21.in.tum.de

We are looking for a student that

- Likes to make software more trustworthy and secure,
- Is pro-active in coming up with design ideas before implementation,
- Has good programming skills in C/C++, and
- Is interested in formal verification and testing.

We are interested in highly motivated students for this study, who will help utilizing our research in verification for testing real-world software. We expect the student to produce high-quality deliverables, both in terms of scientific and engineering contributions.

References

[1] https://isabelle.in.tum.de/
[2] Lammich, Sefidgar - Formalizing the Edmonds-Karp Algorithm. http://www21.in.tum.de/~lammich/pub/itp2016_edka.pdf
[3] Fleury, et al. - A Verified SAT Solver with Watched Literals Using Imperative HOL. https://people.mpi-inf.mpg.de/~mfleury/sat_twl.pdf
[4] Wimmer, Lammich - Verified Model Checking of Timed Automata. https://home.in.tum.de/~wimmers/#tacas-18-paper
[5] King, J. C. *Symbolic execution and program testing*
[6] Sutton, M. et al. *Fuzzing: Brute force vulnerability discovery*
[7] http://people.cs.aau.dk/~adavid/UDBM/
[8] Cadar, C. et al. KLEE: Unassissted and automatic generation of high-coverage tests for complex systems programs
[9] http://lcamtuf.coredump.cx/afl/