Project Topics for Automated Software Testing FS2023

1. Expanding the Reach of Grammar-based Exhaustive Testing

Grammar-based exhaustive testing is a general, highly effective framework for bounded exhaustive testing which we recently introduced. Its key idea is to generate test cases exhaustively from a grammar up to a fixed depth bound. The testing approach has two key advantages over traditional fuzz testing. First, it generates minimal test cases (wrt. grammar). This is useful as according to the "small scope hypothesis" almost all bugs have small bug triggers. Moreover, test case reduction is not necessary, which is convenient for developers. The second key advantage is that the technique can yield robustness guarantees on software, i.e., on the absence of bugs, and quantitate robustness results over multiple releases. We have implemented the technique in our tool E.T. on top of the parser generator. ANTLR in a implementation and applied it to SMT solvers with great success.

The goal of the project is to adapt E.T. to foundational software besides SMT solvers. Potential domains could be Regex Engines, JSON validators, or SAT solvers. Your task would be to (a) design a suitable ANTLR grammar for the chosen domain, (b) to design a testing oracle, and (c) launch a testing campaign using E.T. and (a) + (b).

Advisor: Dominik Winterer - dominik.winterer@inf.ethz.ch

2. Testing a Visual Programming Language

Algot is a visual programming language that uses a novel programming by demonstration mechanism to make programming simpler and more intuitive. Its current version is a web application built using React, TypeScript and the Next.js development framework. This student can choose between two Algot-related projects:

(i) Algot currently has a basic unit testing module. Currently, this module requires defining the tests cases in React instead of Algot. Your task is to expand this basic testing module so that users can write their own test cases using Algot itself by specifying input-output examples. This is then followed by implementing an automatic mechanism to test the effectiveness or quality of user-generated test cases, for example via mutation testing (in which case, the student needs to come up with a suitable set of mutant operators in Algot). This project requires some experience with React or a similar web development framework.

(ii) The Algot implementation in React has never been tested rigorously and may include some interpreter bugs. Your task is to apply a technique that you may choose yourself to perform automatic testing of the Algot implementation, for example fuzz testing or symbolic execution. The goal is to identify and report any interpreter bugs that are present in the implementation.

Advisor: Sverrir Thorgeirsson - sverrir.thorgeirsson@inf.ethz.ch

3. Finding Missed Optimizations in Binary Code

Compilers apply a series of inter-dependent transformations and analyses on code to optimize it but they often miss optimization opportunities. In this project you will attempt to detect such missed optimizations by analyzing the compiler's output (binary code). Examples include finding dead code (i.e., code that cannot be executed), complex control flow that is not simplified (e.g., loops that could have been completely unrolled but are not), and redundant computation that could have been reduced to a constant expression. This project requires a good understanding of assembly code (X86, unless you'd prefer a different target), the ability to read and understand disassembled code, and it will involve building a small tool that can analyze binaries.

Advisor: Theo Theodoridis - theodoros.theodoridis@inf.ethz.ch

4. Compiler Fuzzing via Guided Value Mutation

Given a seed program, your target is to find a mutant of this program that causes large binary differences across two compilers. Specifically, your jobs would be (a) implementing an algorithm that promotes all constant values of a seed program to the intput; (b) designing a fuzzer that mutates a seed program, i.e., how to mutate the constant values; (c) designing a guidance algorithm for the fuzzer to effectively find mutants that meet our need, i.e., large binary differences. This project will not require any prior knowledge about compilers, but need a good implementation skills. You can choose any programming languages, python, however, is preferred.

Advisor: Shaohua Li - shaohua.li@inf.ethz.ch

5. Testing and Optimizing SMT Solvers via Rewrite Rules

SMT solvers are the tools to solve the first-order logic formula. This project expects the students to come up with some rewrite rules for re-writing the formula while preserving the satisfiability. The rewrite rules can be used to 1) find soundness bugs in SMT solvers, 2) uncover potential performance/incompleteness issues or 3) improve the performance.

Advisor: Chengyu Zhang - chengyu.zhang@inf.ethz.ch

6. Finding Small Code with Large Binaries

In this project, we want to find small pieces of code that generate unusually large binaries. Hence, the task of this project is, given a source code, reduce the source code size while maximizing the binary size to source code ratio. Furthermore, analyse the resulting pieces of code and identify patterns that lead to large binaries.

Advisor: Yann Girsberger - yann.girsberger@inf.ethz.ch

7. Testing of Configuration Management Libraries

Modern infrastructures and systems are rarely configured by hand. Over the last decade, there has been a spike in configuration managenemt systems, such as Puppet and Ansible, whose primary goal is to automate the configuration and provision of computer systems and infrastructures using high-level specification languages. Nowadays, most of data centers rely on these configuration management systems and their associated libraries. Therefore, their reliability is of high importance.

The goal of this project is to design and implement an automated approach for finding bugs in configuration management libraries (e.g, see here for some examples of such libraries). The project also involves the investigation and introduction of suitable test oracles that uncover subtle logic bugs (e.g., misconfigurations).

Advisor: Theodoros Sotiropoulos - theodoros.sotiropoulos@inf.ethz.ch

8. Finding Regression Logical Bugs in Database Management Systems

Logical bugs make DBMSs perform incorrect operations. These kinds of bugs have been researched actively in recent. In this project, we want to understand how logical bugs in DBMSs are related to regression. You are expected to propose an approach that can directedly test the code that is different between two versions of DBMSs. Specifically, you need to (1) choose one DBMS and its two version, and use open-source SQL generator (e.g., SQLancer and SQLsmith). (2) design a strategy to guide SQL generator to generate test cases that reach the code that is different in two DBMS' versions. (3) compare the execution results of two versions of DBMSs, and report a regression logical bug if they are different. C/C++ implemented DBMSs are recommended for testing, as you can collect their code information using LLVM (if you want).

Advisor: Zuming Jiang- zuming.jiang@inf.ethz.ch

*** Submit your preference by 8 Mar on https://forms.gle/Rw4nU1C1eLb2381K7**
* We will assign the topic based on your preference on 10 Mar.