BSc Project: Trust Me, I am a Verifier! (Or should you?)

Proof Assistant Reliability and Testing

Project supervisor: Dr Mohammad Ahmad Abdulaziz Ali Mansour, Dr Karine Even Mendoza

A proof assistant is a program that checks if a given mathematical proof is correct. Proof assistants are used in many applications, ranging from practical software verification, like verifying a linux Kernel [1], a C-compiler [2], or cryptographic software by Apple [3] and AWS [4], to more theoretical applications in mathematics, like confirming the correctness of the proof of Fermat's Last Theorem [5]. In software verification, a proof assistant is used to check proofs that a given program behaves exactly as intended — no bugs, no surprise features, no unexpected behaviour. Proof assistants depend on a multitude of pieces of software that are integrated in complex architectures, e.g. compilers, external solvers, etc. Although the core of a proof assistant might not have bugs, the surrounding software might have bugs that affect the usability, reliability, or usefulness of the proof assistant. These may include the following problems, from most to least serious:

- **False negatives** in verifiers lead to undetected violations of the specification, and thus to erroneously classifying buggy software as correct.
- **False positives** in verifiers lead to correct software being incorrectly rejected, hindering the development cycle and confusing the developers.
- **Crashes** (when a verifier or a compiler fails to act on the input program due to an internal error that occurred during the verifier or the compiler execution) are bugs that can lead to inefficient development of software and can confuse the programmer.
- **Hangs** are similar to crashes, but lead to an infinite run of a verifier or a compiler instead of an internal error.

In this project, you will focus on testing the software stack of one of the widely-used proof assistants, Isabelle/HOL [6]. Isabelle/HOL has been used to verify an OS Kernel [1], and is used to verify software affecting millions of people in companies like Apple [3] and Amazon [4].

This project will expose you to areas in logic, formal methods, compilers, programming languages, and fuzzing. For your project, you will need to choose a part of the Isabelle/HOL software chain/tool stack and a fuzzing method you wish to try on that part. The likely places to find bugs are in the frontend, interfaces between Isabelle/HOL and other tools, or the compiler that compiles Isabelle/HOL. You will need to choose as part of your variant testing either of them.

(Relevant modules: theory/logic modules, compilers, and software engineering ones.)

You shall build a new fuzzer (e.g. by writing a new set of code mutations to AFL[1]) or extend significantly an existing fuzzer and show your extension led to more efficient testing of the target compiler as part of the evaluation of your project.

---

[1] You can use AFL or LibFuzzer to write your own code mutations. Check these links for technical details: AFL Michal Zalewski, "Technical "whitepaper" for afl-fuzz," https://lcamtuf.coredump.cx/afl/technical_details.txt; Google, "AFL dictionaries,": https://github.com/google/AFL/blob/master/dictionaries/README.dictionaries; AFL++ https://aflplus.plus; and LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html.

The evaluation of your project shall include:

- A **manual** explaining how to build and install the verifier or the compiler under **instrumentation** from source code, allowing fuzzing and testing;
- **Test case generation campaign** to discover new bugs;
- **Code coverage** comparison of the baseline (state-of-the-art tool) and your tool, showing whether or not your tool can exercise new parts of the verifier or compiler codebase;
- **Throughput** of tests with a comparison to other existing tools;
- And more.

Requirements:

Each student must choose a unique project variant. You can choose one of the following variants:

1. The PolyML compiler, which compiles the Isabelle/HOL proof assistant. Here you could find crashes and miscompilation bugs.
2. The Isabelle Graphical Frontend with the latest stable versions to find crashes or hangs.
3. The interface between Isabelle and other external provers or tools, which are mostly invoked by sledgehammer.
4. Testing Isabelle's unifier, which is a trusted piece of code in Isabelle's core. Here you might find false negatives/positives.
5. Testing pieces of SW (e.g. sel4) that were verified by Isabelle/HOL. Here you could find bugs in the interface w.r.t. the correctness spec of SeL4, for instance.
6. Your own variance with other theorem provers, as listed below.

Please ensure you can run Docker or a virtual machine on your computer before selecting this project. Knowledge of Unix and Ninja/CMake is required for this project. Please open, if you do not already have one, a GitHub account. During the project, you can apply for Amazon EC2 for resources. This is free for 1 year, unless you have already used your free period. Please contact us if this is the case. (https://aws.amazon.com/free/)

Deliverables:

Mandatory: a **final report** discussing the architecture of your chosen SUT as part of the background, description of your methodology, your implementation including how your code works on the chosen SUT, how to build form source the SUT, how to instrument it, your methodology to generate new tests, your methodology and results of how to evaluate the effectiveness of testing, and bug reports. Beyond that, you are also expected to submit a **presentation**, the **source code** and **executable** file of the working system and data, as part of the general requirements of individual projects.

There will be three different links on KEATS at the end of the project for you to upload:

1. The report
2. Zip/tar of your source code
3. Appendix, which includes a printout of all your code
4. Your presentation, which MUST be under 10 minutes (recommended time is 9 to 9.55 minutes)

We strongly encourage you to publish your **artefact** in Zenodo or figshare. This can include your code and data gathered during your evaluation in a shareable format. We also encourage you to create a **dataset** and publish it.

References for the Application

[1] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T. seL4: Formal verification of an OS kernel. In Proceedings SOSP 2009 (pp. 207-220).

[2] Leroy, X., 2009. Formal verification of a realistic compiler. Commun. ACM 52, 107–115. https://doi.org/10.1145/1538788.1538814

[3] https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web

[4] https://github.com/awslabs/AutoCorrode

[5] https://leanprover-community.github.io/blog/posts/FLT-announcement/

[6] Nipkow, T., Paulson, L.C., Wenzel, M., 2002. Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/3-540-45949-9

Additional resources:

Name	Git
Coq / rocq	https://github.com/rocq-prover/rocq
Isabelle	https://isabelle.in.tum.de/dist/Isabelle2025_linux.tar.gz
Lean	https://github.com/leanprover/lean4
HOL4	https://github.com/HOL-Theorem-Prover/HOL
PolyML	https://github.com/polyml