Locating Bugs in C/C++ Programs by Generating Directed Unit Tests

1. Introduction

Debugging complex systems may present a time-consuming challenge, especially if the bugfix requires finding the root cause of the error. The initiative started at Charles University, focused on techniques and tools for debugging large systems, led by Pavel Parizek, Ph.D., already addresses this area of research in the Java/JVM ecosystem.

The initiative proposes a methodology that combines dynamic runtime analysis/recording of programs, generation of, and execution of unit tests. The methodology aims to help discover root causes of errors by user-targeted testing of application components via e.g., inspecting the differences between passing and failing tests in terms of method call arguments and input program states.

The goal of this project is to expand the language/runtime support and provide a significant part of the necessary infrastructure for C/C++ applications. This includes plugins, tools, and orchestration that allow recording of relevant program state, generation of tests, and their execution.

2. Analysis & Research

We evaluated the viability of various tools and approaches to solving the problem and studied related papers that helped steer our ideas throughout the initial phases. The approach we chose heavily involves the LLVM infrastructure, and in the following sections, we describe our needs and the alternative solutions we considered.

- roughly describe what we need (call tracing, capturing program state, testing, …)
    - link to concepts

2.1 Purely AST-based approach

- best for "inspection" - changes seen in the source code
- requires manual header insertion
- while more comfortable in terms of modification, not general enough (C++ syntax)

2.2 Dynamic approach: Pin

- program state “saving” requires forking (cite usage of syscalls)
- syscall usage also limits the programs that could be targeted
- slow
- hard to analyze the function arguments on the instruction level
- x86-only - hard, not theoretical limit (or a limit imposed by us)

2.3 Combination: LLVM IR instrumentation

- Instrument program on the IR level (nice LLVM APIs)
- Perform calls to an custom (linked) library to generalize functionality
- Dynamic configuration via shared memory (managed by the linked library)
- monitoring of the tested application

3. Contribution

- AST plugin
- LLVM IR plugin
- llcap-server
- hooklib
- aside: llvm patch, explained later

4. Conclusion

- what’s done

Evaluation

- keepassxc distilled

Challenges

- LLVM patch
- identifying functions (ID, uniqueness, persistence)
- filtering of functions (so far the reason AST plugin and LLVM patch exist)

Future work

- distill Future work