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

1. Introduction

Debugging complex software systems can be a time-consuming challenge, particularly when the root cause of an error must be identified. Prof. Pavel Parízek[1] has started an initiative focused on debugging large software systems with long-running computations, and proposed a methodology that combines dynamic runtime analysis/recording of programs, generation of unit tests, 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.

With other members of the project team, they started working on a prototype infrastructure for the Java/JVM ecosystem. The goal of this project is to expand the language/runtime support and provide the majority 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 informed our design decisions. The final approach centers on the LLVM intermediate representation (IR), and the following sections describe our requirements and the alternative solutions we considered.

The project goals require implementing a tracing and capturing infrastructure that satisfies the core requirements:

1. capture: store and capture data (injected constants, function arguments)
2. instrumentation: selectively replace arguments with captured data during execution
3. state preservation: argument replacement within a valid program state
4. reporting: monitor and report results back to the user

We derived three key phases that are performed to achieve this: call tracing, argument capture, and testing. Call tracing is performed to narrow the targets for argument replacement, which in turn provides the data for the testing phase. The concepts are explained in detail in the project README, section Concepts[2].

In the following paragraphs, we list selected studied approaches:

2.1 Purely AST-based approach

Initial prototypes involved directly modifying the target application's source code by walking clang's C++ AST and inserting code fragments. Despite being simple to inspect and modify, we deem the technique not general enough for our purposes. For example, simple syntactic modification of the C++ source code may become extremely convoluted due to the language rules.

[1] https://d3s.mff.cuni.cz/people/pavelparizek/
[2] /LLVM/README.md#concepts

2.2 Low-level dynamic approach: Pin

Following the challenges with the AST approach, we considered a lower-level dynamic instrumentation approach using Intel Pin. While Pin offers a non-intrusive method for runtime inspection and modification, it was deemed too low-level for this project. Our potential reliance on the Application Binary Interface (ABI) makes argument analysis difficult, and the usage of Pin may also pose a significant performance overhead. Additionally, Pin's limitation to the x86 architecture would restrict the project's applicability.

2.3 LLVM IR instrumentation

LLVM Intermediate Representation (IR) modification proved the most useful in our prototypes. The IR presents a balanced tradeoff between the high-level language concepts and the machine-level concepts. Further, the LLVM project offers comfortable APIs for IR modification and a well-documented, high-utility plugin infrastructure, which proved crucial for the project's goals. Despite this, we needed to perform a small modification of `clang` to allow insertion of IR metadata during the AST phase. The patch [3]is small and well-documented, but presents certain challenges discussed later.

Research notes[4] [5] [6] [7] and progress updates [8] providing much more insight are available in the project repository.

3. Contribution

3.1 Components

Our solution uses a custom LLVM pass[9] (C++) to instrument the target application, inserting function calls to our custom library, `hooklib`[10] (C/C++), which cooperates with a central server component, the `llcap-server`[11] (Rust) to realize the tracing and testing processes. For example, to capture an argument, the instrumentation first performs a pre-IR AST pass[12] (C++) to provide crucial argument and function information to the IR pass plugin. The IR pass inserts calls to `hooklib` with the pointer to/value of the argument data.

During runtime, the instrumented application writes captured argument data to a shared memory buffer. This buffer is managed and synchronized by `llcap-server`, which also provides the argument value during the testing phase. The target application is `fork`ed at the tested function start during the testing phase, arguments are replaced, and the result is observed.

This process requires modification of build files and two compilations of the target application, the second one is repeated for a different targeted function set to be instrumented.

3.2 Demonstration

For demonstration, we include a simple C++ program[13], along with instructions on how to perform all the required interactions with the `llcap-server`. We also include a containerized demo environ-

[3] LLVM/sandbox/01-llvm-ir/clang-ir-mapping-llvm.diff
[4] LLVM/notes/00-initial-analysis.md
[5] LLVM/notes/01-llvm-ir-metadata-emission.md
[6] LLVM/notes/00-paper-notes.md
[7] LLVM/notes/000-TODOs.md
[8] LLVM/notes/00-progress-updates.md
[9] LLVM/sandbox/01-llvm-ir/llvm-pass
[10] LLVM/sandbox/02-ipc/ipc-hooklib
[11] LLVM/sandbox/02-ipc/llcap-server
[12] LLVM/sandbox/01-llvm-ir/custom-metadata-pass/ast-meta-add
[13] LLVM/sandbox/02-ipc/example-arg-replacement

ment[14] ready to run (not only) the demo. Finally, the demo includes a detailed explanation of the observed or saved outputs of the target application to provide proof that our approach works.

3.3 Testing

As there are many interacting components, utilizing low-level IPC primitives and delicate IR modification takes place, we also include a set of end-to-end tests[15] which automatically perform the entire testing process, starting with only `.c`/`.cpp` and `CMakeLists.txt` files and scripts that test the outputs. As part of the test suite, we also perform stress testing related to low-level implementation details of the underlying shared-memory protocol.

3.4 Extension

Currently, we support built-in primitive types and the `std::string` (tested only with the `C++11`'s `std::string` ABI). Support for additional *pointer and reference* types, however, can be added by modifying the `llvm-pass` and `hooklib` components. The process of adding custom type support is non-trivial, yet possible and documented[16] in the project repository.

4. Project Execution and Management

Throughout this project, the author maintained a regular consultation schedule with the supervisor, Prof. Pavel Parízek, provided frequent progress updates to the team, and monitored the progress of the Java/JVM system being developed by another team member. Following an initial analysis and evaluation of various approaches, drawing inspiration from related research papers, and the author's prior experience with instrumentation tools, we decided to focus on the LLVM toolchain and IR instrumentation. We placed a strong emphasis on creating detailed documentation for all implemented components, which outlines the technical challenges encountered.

The core components of this project are the `llcap-server`, the LLVM IR pass, the `hooklib` library, and the clang patch. Roman Vašut is responsible for developing all these components, as well as providing their documentation and the end-to-end test suite. In the final stages of the project, the author frequently iterated on various parts of the system, incorporating the invaluable feedback and reviews provided by Prof. Parízek, to whom the author extends their gratitude for his support and guidance.

5. Conclusion

This project adapted a dynamic debugging methodology from the Java/JVM ecosystem to C/C++ applications. By choosing LLVM IR instrumentation, the developed infrastructure, consisting of a custom LLVM pass, a runtime library (`hooklib`), and a central server (`llcap-server`), provides a functional system for generating directed tests. The included demonstration, end-to-end tests, and development documentation serve as a solid basis for future work.

5.1. Evaluation

We chose the `KeePassXC`[17] project to validate the tool's core functionality on a medium-scale, real-world application. With rather small build system modifications, it confirmed that the LLVM-based

[14] LLVM/podman
[15] LLVM/sandbox/02-ipc/e2e-tests
[16] LLVM/notes/development-manual.md
[17] https://github.com/keepassxreboot/keepassxc

approach is viable for both call tracing and argument testing, successfully capturing and replaying program state. Performance analysis showed a minimal impact on build times.

To be precise, we demonstrated that the tool could be used to generate different-length passwords from a command-line interface by injecting different size-specifying arguments captured, proving the concept works as intended. We also evaluated the GUI version of the app with some, albeit lesser, success, presumably due to the GUI architecture. The evaluation document[18] precisely documents the changes to the `keepassxc` repository, as well as the outputs of the developed tools.

5.2. Challenges

`clang` metadata patch

The development of this project presented several key challenges. The most significant was the need for a custom clang patch to pass crucial type information from the AST to the IR pass via IR metadata. This rather small and simple patch, however, requires the users to use a patched build of `clang` or compile it on their own, which is a time-consuming process. Further, some test cases spuriously fail due to the tests' hard-coded value of metadata positions. We have checked and documented this behavior and believe that the patch is benign[19].

Other challenges involved function identification among different compilation units and designing for reasonably performing instrumentation at the same time. This was solved by shortening the identifiers transferred in shared memory, combined with the dual build workflow. A significant amount of time was also spent researching workarounds, the ways to avoid the `clang` patch, and the LLVM IR itself.

Future work

Future development can focus on enhancing the tool's robustness, usability, and functionality. Key areas include ensuring comprehensive exception handling, eliminating the dependency on a custom LLVM patch, improving the extensibility of plugins, and adding support for multithreaded programs. We also suggest exploring fuzzing techniques to automatically generate new test cases, particularly for components whose API is not designed to be reachable with conventional testing tools. For a detailed list, please refer to the project's Future Work document[20].

[18] LLVM/report/EVALUATION.md
[19] LLVM/README.md#note-on-clang-tests
[20] LLVM/notes/000-TODOs.md#future-work