Program Analysis and Manipulations in Compilers

Fall 2022 Capstone Proposal

Clients	Clément Aubert & Neea Rusch, School of Computer and Cyber Sciences
Contact	caubert@augusta.edu and nrusch@augusta.edu
Topic	Compilation, program analysis and transformation
Preferred Skills	Interest in programming and software architecture, scientific curiosity, formal reasoning.
Tools	LLVM compiler, `C++`, git

1 Introduction

During the 1984 Turing Award acceptance speech, the audience was flabbergasted by the exhibition of the now-infamous exploitation of *miscompilation*: in 3 steps the speaker demonstrated how, by modifying the `C` code compiler, he was able to deliberately cause it to alter the source program whenever a particular pattern of login command was matched, but *without leaving any trace* of this harmful Trojan horse in the compiler source code! That speaker was Ken Thompson, pioneer computer scientist and creator of Unix operating system, and his message was clear:

> You can't trust code that you did not totally create yourself. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect. [1]

This capstone project involves a deep dive into the world of compilation and developing familiarity with topics often obscure to high-level language programmers.

2 Overview

Compilers perform the crucial function of translating programs written in one programming language to another, typically from high-level source code to executable programs. Numerous program analyses and transformations take place during compilation to improve the quality of the target program: error identification, performance optimizations, dead-code elimination, etc. Without expertise in compilation, programmer may be unaware of these transformations and their side-effects, with potentially significant implications to program correctness [2] and security [3].

Goal. This capstone project involves rigorous and hands-on study of modern compiler analyses and program transformations, culminating in implementation of a transformation pass for the LLVM compiler.

Perks. Throughout the project the student will have continuous support from two active mentors. Dr. Aubert is an expert in theory of computation, and Ms. Rusch has several years of industry experience in software engineering to lead the student through practical implementation of the project.

3 Learning Objectives

This project will familiarize the student with:

1. the architecture, operations, and terminology of compilers,
2. theoretical understanding of compilation, program analyses and transformations,
3. thinking critically about correctness and security of compilation, and
4. modern software development practices and open source projects.

During this project the student will also obtain experience working with these tools:

- LLVM – modern, open-source compiler for e.g., `C#`, `Lua`, `Ruby`, `Rust`, `Swift`, …,
- `C++` programming language – prior experience is not required, but will facilitate better outcomes,
- `git` – version control system, and
- Github – host of millions of git projects, including the LLVM source code.

4 Project Timeline and Phases

The student will progress toward learning objectives by completing four phases of study, as outlined below.

Phase 1: Familiarity with compilers and LLVM Duration: 3-4 weeks

The primary focus in this stage is to develop familiarity with the compiler architecture, necessary tooling for working with LLVM source code and the LLVM compiler documentation and compilation terminology.

Phase 2: Understanding the analyses Duration: 3-4 weeks

In this stage the focus in studying the compiler analyses, particularly loop unrolling and splitting: under what conditions and how are these optimizations performed, what impact they have on the compiled code, gathering experimental metrics on resulting executable program performance, and understanding the correctness properties and security implications of these optimizations.

Phase 3: Getting practical with `C++` and implementing first transformation Duration: 3-4 weeks

After developing thorough understanding for various analyses, it is time for practical implementation. At this stage the goal is to practice how to implement a simple "toy" compiler pass, to perform choice optimization from source to target, such as loop splitting, and evaluating the resulting performance before and after.

Phase 4: Putting it all together – implementing an LLVM pass Duration: 4-7 weeks

In the final stage the focus is implementing a compiler optimization pass in `C++` that integrates with the LLVM compiler. Successful completion of this stage will result in a forked version of the LLVM compiler, with the added custom-built pass, for the student to share and showcase in their personal Github profile as demonstration of their successful completion of learning objectives and expertise in compilation.

5 Pushing Further

Some additional directions could be explored:

- Evaluating the performance of the compiler pass developed in phase 4, by benchmarking then visualizing and analyzing the results
- Learning more about correctness in compilation and secure compilation, by studying the CompCert formally verified C compiler, and its variants
- Exploring the path of proving correct compilation passes, by using mechanical proofs

The clients recently published an article on the topic to be presented at VMCAI [4] and submitted a proposal to the CURS' Summer Scholars Program: a successful team would potentially be given the opportunity to co-author future papers and to continue this project over the Summer as a paid internship.

6 Getting Started

- Have a look at the "LLVM Compiler Infrastructure" website: https://llvm.org/
- Practice `C++` @ Codewars: https://www.codewars.com/collections/c-plus-plus/
- Read *"Reflections on Trusting Trust"*: https://dl.acm.org/doi/pdf/10.1145/358198.358210

References

[1] K. Thompson, Reflections on trusting trust, Commun. ACM. 27 (1984) 761–763. https://doi.org/10.1145/358198.358210.

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

[3] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, A. Trieu, Formal verification of a constant-time preserving C compiler, Proc. ACM Program. Lang. 4 (2019). https://doi.org/10.1145/3371075.

[4] C. Aubert, T. Rubiano, N. Rusch, T. Seiller, Distributing and parallelizing non-canonical loops, in: Verification, Model Checking, and Abstract Interpretation (VMCAI 2023), 2023.