Translating L_{if} to L_{cfi}: A Multi-Step Compilation Approach (Team 2)

Ephraim Siegfried, Luca Gloor and Ruben Hutter

University of Basel
Interpretation and Compilation of Programming Languages (ICPL) seminar
Autumn Semester 2024

1 Introduction

"Every program is open source if you know Assembly." - Unknown. But must this always be the case? What if we could write programs in a high level programming language like python and interleave them with many different programs without changing input-output behavior making it (almost) impossible to reverse engineer the original program? The interleaving part of this problem has already been solved by Ali Ajorian (TODO Luca: Insert Aporia Paper reference here) but the first part was where this project came into play. The goal was to create a compiler from the python subset L_{if} to the aporia language L_{cfi} allowing the original program to be obfuscated at a later point in time. This objective was achieved by generating two new intermediary languages L_{if}^{sc} and L_{if}^{flat}, each with their own restrictions and compiling the input program in three separate steps in order to finally reach the desired L_{cfi} representation.

In the following report, we will give an overview of the technical background before we explain the implementation of the project. We will also discuss the difficulties that were encountered while coming up with solutions, evaluate the achieved outcomes, and reflect on the lessons learned during the course of this project.

2 Technical Background

2.1 Program Obfuscation and Its Challenges

Program obfuscation is the process of transforming a program into a functionally equivalent version that is difficult to analyze or reverse-engineer. It is widely used for *software protection, digital watermarking, and cryptographic applications*, such as secure multiparty computation and homomorphic encryption [1].

Traditional obfuscation techniques rely on ad hoc transformations like control flow flattening, junk code insertion, or opaque predicates. However, these methods often fail against modern deobfuscation tools that leverage symbolic execution and AI-assisted static analysis [1].

2.2 Aporia: Instruction Decorrelation as an Obfuscation Strategy

Aporia introduces **instruction decorrelation**, a technique that breaks the one-to-one mapping between source and compiled instructions. Unlike traditional obfuscation, decorrelation **rearranges, delays, and duplicates instructions in a way that depends on runtime execution states** [1].

This is achieved through two key mechanisms:

- **Randomized Scheduling**: Instructions are not executed in their original order, forcing an adversary to reconstruct execution flow dynamically.
- **Context-Dependent Execution**: Execution conditions are based on hidden states, preventing static analysis.

2.3 Security Model and Side-Channel Resistance

Aporia's security model assumes an *adversary capable of both static and dynamic analysis*. However, unlike most obfuscation methods, instruction decorrelation also offers protection against **speculative execution attacks** (e.g., Spectre, Meltdown) by eliminating direct data dependencies [1].

Additionally, the system **introduces phantom instructions**—instructions that appear valid to an attacker but do not contribute to actual program execution. This makes symbolic execution impractical.

2.4 The Lcfi Language and Its Role in Obfuscation

Lcfi (the target language of our compiler) is specifically designed to support instruction decorrelation. It employs:

- **Predicate-based execution**, where each instruction is executed only if a dynamic condition holds.
- **Phantom instructions**, which introduce noise in the control flow to mislead attackers.
- **Delayed branching**, which defers the resolution of execution paths, making control flow reconstruction NP-hard [1].

2.5 Relation to Our Project

Our project builds upon Aporia's work by designing a **compiler from Lif to Lcfi**, enabling automatic program transformation for obfuscation. The compiler translates a subset of Python (Lif) into Lcfi through multiple intermediate representations, enforcing instruction decorrelation at various stages.

By leveraging Aporia's **randomized scheduling, context-aware execution, and phantom instructions**, our compiler provides a practical tool for obfuscation that remains resilient against modern reverse-engineering techniques.

3 Implementation

How we implemented the compiler, what we used, how we structured the code, etc.

4 Difficulties

Write some difficulties you encountered during the project. This can be technical difficulties, difficulties with the group, etc.

5 Evaluation

In this section, we will evaluate the project and the results. We will discuss the project's objectives, the implementation, and the results. We will also discuss the project's limitations and possible improvements.

6 Lessons Learned

What we learned from this project...

7 Conclusion

A nice conclusion...

8 Individual Contributions

8.1 Ephraim

- **Technical Background:** Wrote the section about the technical background of the project.
- **Introduction:** Wrote the section about the introduction of the project.
- **Lessons Learned:** Wrote the section about the lessons learned from the project.
- **Conclusion:** Wrote the section about the conclusion of the project.

8.2 Luca

- **Introduction:** Wrote the section about the introduction of the project.
- **Technical Background:** Wrote the section about the technical background of the project.
- **Lessons Learned:** Wrote the section about the lessons learned from the project.
- **Conclusion:** Wrote the section about the conclusion of the project.

8.3 Ruben

- **Introduction:** Wrote the section about the introduction of the project.
- **Technical Background:** Wrote the section about the technical background of the project.
- **Lessons Learned:** Wrote the section about the lessons learned from the project.
- **Conclusion:** Wrote the section about the conclusion of the project.

References

[1] Ali Ajorian, Erick Lavoie, and Christian Tschudin. "Obfuscation as Instruction Decorrelation". In: *arXiv* (2024). arXiv: 2411.05570 [cs.CR]. URL: https://arxiv.org/abs/2411.05570.