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 The L_{cfi} Language and Its Role in Obfuscation

L_{cfi} (the input language for Aporia) is specifically designed to support instruction decorrelation. It employs:

- **Predicate-based execution**, where each instruction is executed only if a dynamic condition holds, making control flow reconstruction difficult.

While L_{cfi} facilitates instruction decorrelation, its design constraints make it challenging to write directly. Developers must handle explicit predicates and low-level constructs, which significantly differ from high-level programming languages like Python.

2.4 Relation to Our Project

The goal of our project is to create the missing "bridge" between Python and the Aporia framework. Writing code directly in L_{cfi} is impractical due to its low-level nature and strict requirements. Our compiler translates programs written in a subset of Python (L_{if}) into L_{cfi}.

This transformation allows developers to:

- Write secure code in a familiar, high-level syntax.
- Leverage Aporia's obfuscation techniques without directly interacting with L_{cfi}.

By bridging this gap, our project streamlines the workflow for developers, enabling them to write Python-like programs and automatically obtain the obfuscation benefits provided by Aporia.

3 Implementation

3.1 Overview

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris.

Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

cmp	::=	!= | == | < | <= | > | >=
bop	::=	+ | - | * | / | %
pred	::=	true | false | bool_var
bool_exp	::=	pred | (bool_exp) | !bool_exp
	|	bool_exp && bool_exp
	|	bool_exp cmp bool_exp
num_exp	::=	number | num_var | (num_exp) | -num_exp
	|	num_exp bop num_exp
inst	::=	print("string",bool_exp) | bool_exp
	|	print("string", num_exp) | num_exp
	|	bool_var = bool_exp
	|	num_var = num_exp
stmt	::=	($label)? | pred : inst
declar	::=	bool bool_var$^+$ | int num_var$^+$ | int float_var$^+$
L_{cfi}	::=	declar* stmt*

Fig. 1: Backus Naur Form Grammar of L_{cfi}

cmp	::=	!= | == | < | <= | > | >=
exp	::=	int | input_int() | -exp
	|	exp + exp | exp - exp
	|	(exp) | var
	|	True | False | exp and exp
	|	exp or exp | not exp | exp cmp exp
	|	exp if exp else exp
stmt	::=	print(exp) | exp | var = exp
	|	if exp : stmt$^+$ else : stmt$^+$
L_{if}	::=	stmt$^+$

Fig. 2: Backus Naur Form Grammar of L_{if}

3.2 First pass: Remove complex operands (L_{if}^{sc})

cmp	::=	!= | == | < | <= | > | >=
atm	::=	True | False | var
exp	::=	int | -exp | exp + exp
	|	exp - exp | (exp)
	|	exp and exp | exp or exp
	|	not exp | exp cmp exp | atm
stmt	::=	print(exp) | exp | var = exp
	|	if atm: stmt$^+$
L_{sc}	::=	stmt$^+$

Fig. 3: Backus Naur Form Grammar of L_{sc}

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

3.3 Second pass: Flatten and Make everything an if (L_{if}^{flat})

The goal of this pass is to eliminate nested `if` statements, ensuring that all statements are wrapped in top-level `if` statements in the resulting intermediary language. We call this language L_{flat}, which inherits from L_{sc} but restricts all `if` statements to be top-level only, as described by the BNF in Figure 4. Consequently, each `if` statement body now contains exactly one statement, which can not be an `if` statement, thereby preventing any form of nesting.

The compiler takes an L_{sc} AST as input and produces an L_{flat} AST. Applying this pass to the sample code shown above yields the flattened version in Figure 5. Statements that were not originally inside any `if` are wrapped in an `if`

cmp	::=	!= | == | < | <= | > | >=
atm	::=	True | False | var
exp	::=	int | -exp | exp + exp
	|	exp - exp | (exp)
	|	exp and exp | exp or exp
	|	not exp | exp cmp exp | atm
stmt	::=	print(exp) | exp | var = exp
if_stmt	::=	if atm: stmt
L_{flat}	::=	if_stmt$^+$

Fig. 4: Backus Naur Form Grammar of L_{flat}

`True` statement, while statements that were nested in an `if` block are guarded by newly introduced boolean variables.

```
if True:
    a = 3
if True:
    pred_0 = a > 0
if True:
    pred_1 = not pred_0
if pred_0:
    pred_2 = a > 2
if pred_0:
    pred_3 = not pred_2
if True:
    pred_4 = pred_0 and pred_2
if pred_4:
    b = 1
if True:
    pred_5 = pred_0 and pred_3
if pred_5:
    b = 0
if pred_0:
    print(b)
if pred_1:
    print(2)
```

Fig. 5: L_{flat} example code

When the compiler encounters an `if` statement nested within another `if`, it generates a temporary boolean variable holding the conjunction of the outer and

inner conditions. The statements in the nested `if` block are then wrapped in a new `if` whose predicate is this temporary variable. By repeating this process, we flatten all nested `if` statements without changing the original program's logical flow.

3.4 Third pass: Select instructions (L_{cif})

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

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.