SECURITY ANALYSIS OF COMPILER OPTIMISATIONS

A Project Report Submitted

for the Course

CS498 Project-I

by

Dristiron Saikia

(Roll No. 180101022)

under the guidance of

Dr. Chandan Karfa

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781039, INDIA

November 2021

CERTIFICATE

This is to certify that the work contained in this thesis entitled **"Security Analysis of Compiler Optimisations"** is a bonafide work of **Dristiron Saikia (Roll No.: 180101022)** carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.

Supervisor: **Dr. Chandan Karfa**

Assistant Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology Guwahati, Assam.

November, 2021
Guwahati

Acknowledgement

I would like to express my sincere gratitude to my supervisor **Dr. Chandan Karfa** for giving me the opportunity to explore this new field of Compiler Optimisations and the much needed guidance to deep dive into the field of research and to explore some state-of-the-art works.

ABSTRACT

Security Analysis revolves around the design and implementation of systems to trace and analyze potential information leak/security threats. The compiler translates source code from a high-level programming language(e.g., C/C++) to a lower level language (e.g.assembly language, object code, or machine code) to create an executable program. **Optimising Compilers** while compiling programs does additional steps of adding optimisations. These, i.e., program transformation techniques, improve the code by consuming fewer resources (i.e., CPU, Memory) and delivering high speed. The optimisations are program transformations that guarantee correctness according to its language specifications. But certain transformations in the past have proved to be resulting in information leaks/security threats. This gap between correctness and security arises because many optimisations are context-sensitive. To analyse security for optimisation requires researchers to trace and experiment for specific code patterns that could be a potential security threat. In this work, I attempt to build a system to trace for information leakage in programs due to compiler optimisations.

Contents

Chapter 1

Introduction

Threats due to information leaks have posed a significant amount of challenges to developers and system designers. The focus has been on developing secure hardware constructs and abiding by security standards while writing codes for sensitive information of the system. Even after taking care of the two aspects, the Compiler goes through every line of code written by the developer. With growing interest in increasing developer productivity and lesser development time, compilers optimize the source code to decrease resource usages and deliver high speed. Recent works have shown that optimisations done by the Compiler do create additional information leaks.

1.1 Compiler: A Secure Bridge ?

The concept that compiler ensures security guarantees has been proven wrong time and again. The fundamental reason behind such instances is that correctness proofs of compiler transformations are not enough to validate the security guarantees. When we say a transformation is functionally correct, it means that there exists no source code, such that the behavior of the result-

ing compiled code deviates as specified in the language standard. But when we say a transformation is secure, it means potential information leakage after the transformation is at most as before the transformation. Let us see an example of security violations by a functionally correct optimization called dead store elimination.

```
1  // Dead Store Elimination
2  char *getPWHAsh(){
3      long i; char pwd[64];
4      char *sha1 = (char*)malloc(41);
5      // read password
6      fgets(pwd, sizeof(pwd), stdin);
7      //calculate sha1 of password
8      ...
9      //overwrite pwd in memory
10     // Alternative (A): use memset
11     memset(pwd,0,sizeof(pwd)); // (A)
12     for(int i=0;i<sizeof(pwd); i++){
13         pwd[i] = 0; // (B)
14     }
15     //return only hash of pwd
16     return sha1;
17 }
```

Figure 1.1: Dead store elimination

In this optimisation, compilers intend to remove an assignment of a variable that is not read by any subsequent instruction because it seems redundant. Removing such assignments does not violate correctness guarantees. But from a security perspective, eliminating these data scrubbing operations could lead to sensitive information residing in the memory for a longer time than intended by the developer. Though there are solutions where the developer considers the implications of such optimisations in the code, it is pretty impractical to expect every developer to know and understand the effect of every optimization. Therefore it has become necessary that compiler optimisations are either formally proved or are thoroughly tested through proper testing measures before using.

1.2 Problem Statement

In this project, we attempt to build a system, wherein users could test various compiler transformations and check if they result in any information leak. We attempt to build an interface where given an input source code, users can trace potential information leak due to compiler transformations. Considering the increasing usage of LLVM, which provides a collection of reusable compiler and tool-chain technologies giving developers flexibility to construct highly optimized compilers, optimizers, and run-time environments. We therefore, work on the following problem statement: **Tracing information leakage at different Optimization levels in LLVM**.

Chapter 2

Preliminaries

The following section is a brief introduction to how a compiler does program transformations, a method to define information leak in programs due to compiler transformations, an information flow analysis method for untrustworthy inputs called taint analysis, and the popular compiler infrastructure called LLVM.

2.1 Architecture of Optimizing Compiler

An optimizing compiler is a compiler that tries to modify some attributes of an executable computer program. The main objectives of these optimisations are to minimize a program's execution time, memory footprint, storage size, and power consumption. Optimizing compilers guarantees that the resulting program will always generate the correct output for every input instance after the optimisation. Optimisations in compiler are implemented as a sequence of *optimizing transformations*, algorithms which takes a program as input and generates a semantically equivalent output which either has lesser execution time or uses less resources.

In figure 2.1, we see that compiler does lexing, parsing and semantic checks

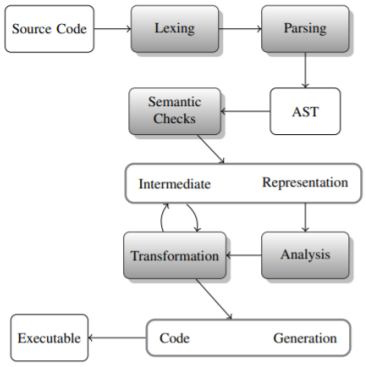

Figure 2.1: Architecture of optimizing compiler

on the source code using a data structure called *Abstract Syntax Tree*. After the checks are done for semantic checks according to language specifications, a combination of transformations and analysis takes place upon the intermediate representation to generate machine code. It is these transformations that result in the optimisation of source code. After the optimisations are over the IR gets transformed into machine-dependent executable code.

2.2 Taint Analysis

Taint analysis is a static program analysis method that tracks the influence of input variables on program state. We identify variables corresponding to user inputs as tainted. At each program state, we check if there is any taint flow. In taint analysis, we look for possible taint flow to sensitive sink functions, ultimately leading to security leaks. The main objective of taint analysis is that we never want our tainted variables to interact with a sink function.

In our system, we will use taint analysis method after every compiler opti-

misation to check if there arise any *information* leak.

2.3 LLVM

LLVM is a compiler framework that provides users with numerous compiler modules and data structures used during the compilation process. LLVM lets compiler developers reuse existing modules and make modifications to the compilation process itself. LLVM uses Clang(for C/C++) as the front

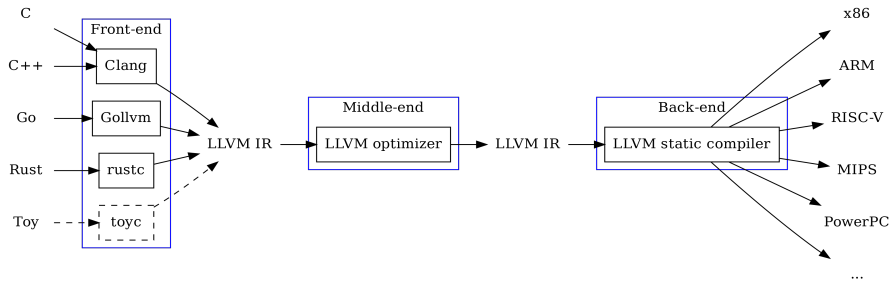

Figure 2.2: LLVM

end, IR(Intermediate Representation) in the middle-end, converted into a machine-dependent executable file. LLVM IR is a platform-independent human-readable assembly code that follows Static Single Assignment(SSA) with infinite local registers, where all of the compiler optimisations happen. LLVM comes with a bunch of inbuilt optimisations which can be modified according to the needs. It exposes users to its internal data structures such as Abstract Syntax Trees, Call Graphs, and Control flow graphs which gives the flexibility to build data-flow solvers or static analyzers without understanding the entire compiler process.

Chapter 3

Methodology

In this section, we briefly discuss the design details of our information leak trace system.

3.1 Design Details

LLVM provides reusable modules to implement *transformation and analysis* passes. Transformation passes are responsible for optimisations, whereas analysis passes let users visualize the program in *Call Graphs, Control Flow Graphs, and Dominator trees*. Since there are a lot of optimisations that require a sequence of transformations to occur one after another, therefore we decided to work upon a framework that solves data flow problems on LLVM IR target code rather than any high-level language code, like *C, C++* For this we refer to the open-source project **PhASAR**.

3.1.1 Architecture

To trace information leakage using taint analysis, it becomes necessary to provide *tainted sources* and *sinks*. At each phase of optimisation we run

taint analysis on the resulting IR and report about leaks, if any. In this case, PhASAR will be used as a library along with *IFDS Framework* for taint analysis. In Fig. 3.1 we present a high-level architecture.

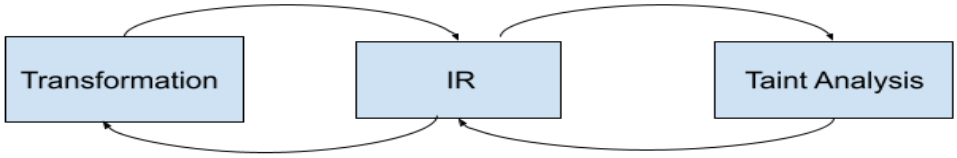

Figure 3.1: High level design

Since, there are many optimisations and running each of them manually is a tiresome task so we need to automate the process of running taint analysis. For this we can make use of *LLVM Pass Managers*. We include the taint analysis procedure in LLVM like an *analysis pass* and encode the flow functions for taint flow analysis, which will be solved using IFDS algorithm[6] by PhASAR. Figure 3.2 shows a high-level diagram.

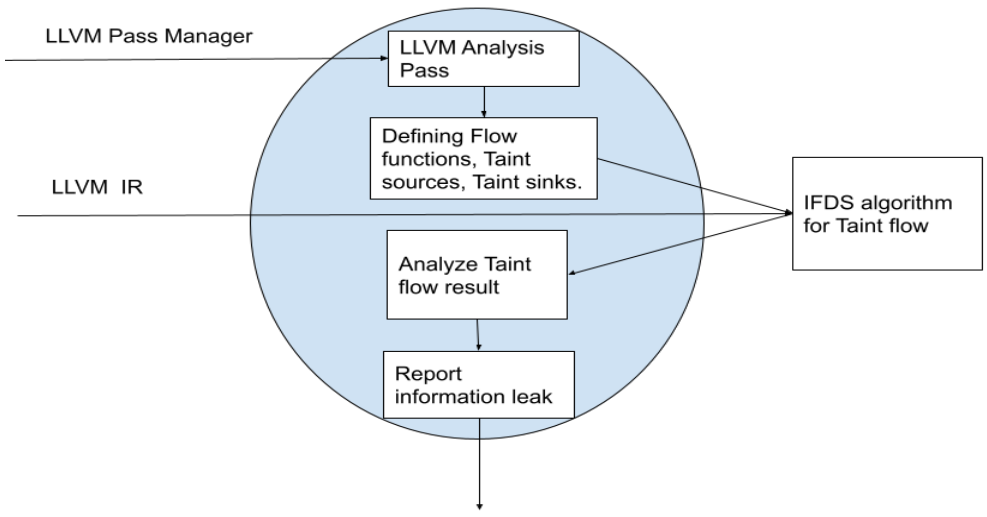

Figure 3.2: Taint analysis design

Firstly, we need to register our new analysis pass into the LLVM Pass Manager and call the analysis pass after every transformation passes. We use PhASAR as a library and call the IFDS Taint Analysis Solver, and define the appropriate data flow, taint source, and sink functions inside it. After each transformation pass, we call *llvm::PrintModulePass()* which will print the resulting IR into a .ll file which we will pass as input to our solver. The solver will do context-sensitive Taint Analysis and return a list of Leaky IR instructions. We then analyze the result and report the information leak.
The approach is generic in the sense that it could be extended to trace the effect of each of the earlier transformations upon the leaky IR instruction.

Chapter 4

Efforts

Being new to the field of LLVM, I faced a lot of challenges in learning the codebase. **LLVM, Clang** has a build of around **59GB** (Figure 4.1) containing packages such as **CMake, GCC, Python, zlib, GNU Make**

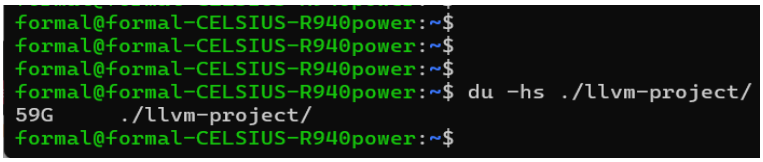

Figure 4.1: llvm-build size

I also experimented as to how to add analysis passes in using the LLVM Pass Manager in Figure 4.2

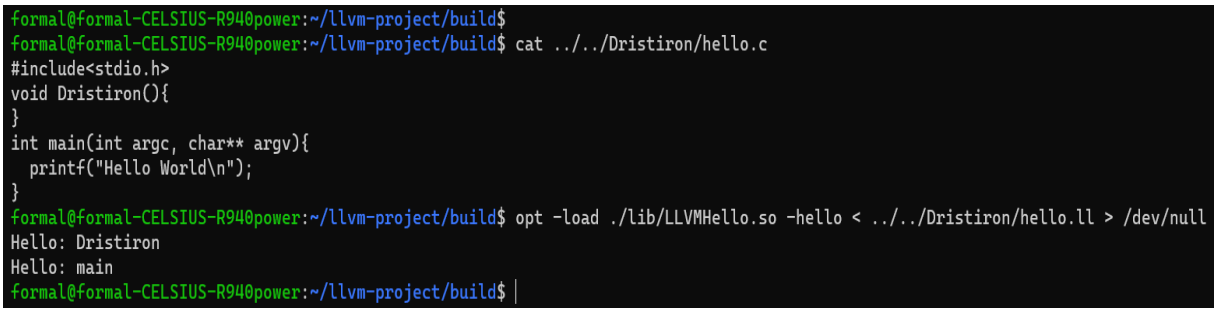

Figure 4.2: llvm hello pass

We build a Shared object file *LLVMHello.so* corresponding to *Hello Analysis Pass*. We generate the unoptimised LLVM IR file for source code *hello.c*. The Pass traverses the IR and returns as output every function that it encounters in the source code.

Since, the entire optimisation phase is dependent on LLVM IR, therefore significant time was being devoted in understanding the IR syntax as well how the transformation passes affect the IR. Following figure 4.3 is one such example of IR transformation:

Figure 4.3: Scalar Replacement of Aggregates

The aim of this optimisation is to reduce unwanted *alloca* instruction present

in the IR. *alloca* instructions are basically allocating registers for a variable. Since, LLVM IR follows Static Single Assignment form therefore unoptimized IR has too many registers alloted. Introduction of phi node occurs in this transformation which is useful in case of branch instructions, and to remove redundant register allocations. Registers having duplicate data over the basic blocks are also removed during SROA optimisation.

4.1 Q & A

Following are some of the constraints/obstacles that led to the current design of the system:

- **Can we use static analyzers for our analysis purposes ?**
  It is not suitable for our system as it will have scalability issues in accommodating many optimisations. There exist dependencies among transformation passes itself, which the user might be required to know to use static analyzers.

- **How can we accommodate in the LLVM environment the subroutine to call Taint solver ?** The best way to accommodate is to use an analysis pass that could take advantage of the internal LLVM structures, transformed during the optimisation phases without modifying the IR.

- **How do we run analysis on intermediate steps ? Do we need to try all possible optimisations one by one ?**
  No, we will automate the testing for each optimisations. We will use our analysis pass within the LLVM codebase, which will be wrapper function to our Taint solver.

- **Since optimised code is present in IR how can we run Taint solver on IR ?**

  Due to the above reason, we had to look for PhASAR, which does the task on IR input. There could be other approaches, but we might need to introduce our data structure which gets modified after every transformation passes, which comes with the cost of scalability issues, and modifying LLVM itself would also be messy.

Chapter 5

Conclusion and Future Work

In this semester, I was able to familiarize myself with the necessary concepts of compiler and compiler optimisations. I also got to know about the correctness security gap that arises and how in the past multiple optimisations had turned out to be information leaky. Moreover, since the LLVM project has a very large code-base, I was able to learn about the basic ways in which we make use of existing LLVM utilities suiting our needs.

Developing the system would ultimately require me to have good grasp of C++ language, OOPS concepts, Data flow analysis approaches and also exploit the various modules,internal data structures used by LLVM to implement transformation and analysis passes.

For the next semester, I plan to proceed with the real implementation of the system and also research on some alternate approaches where we could run taint analysis by just using the internal modules present in the LLVM project. The primary focus would be to develop a working setup at least, where we can correctly trace information leakage for single file programs, without worrying much about scalability and execution times.

References

[1] LLVM Language Reference Manual. `https://llvm.org/docs/LangRef.html`, 2019.

[2] Kedar S. Namjoshi Chaoqiang Deng. Securing a compiler transformation. *Formal Methods in System Design, Volume 53, Issue 2*, pages 166–188, October 2018.

[3] Thomas Jensen. Frēdēric Besson, Alexandre Dang. Information-flow preservation in compiler optimisations. *IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019.

[4] F. Bavera Marcelo Arroyo, Francisco Chiotta. An user configurable clang static analyzer taint checker. *35th International Conference of the Chilean Science Society (SCCC)*, 2016.

[5] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410, Cham, 2019. Springer International Publishing.

[6] Susan Horwitz Thomas Reps and Mooly Sagiv†. Precise interprocedural dataflow analysis via graph reachability. *22nd ACM SIGPLAN-SIGACT*

symposium on Principles of programming languages, pages 49–61, January 1995.

[7] Dawn Song Vijay D'Silva, Mathias Payer. *The Correctness-Security Gap in Compiler Optimization*. IEEE Security and Privacy Workshops (SPW), 2015.