UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

Quality And Certification

STATIC ANALYSIS TOOLS FOR LLVM CLANG

EDOARDO DINI, FRANCESCO TERROSI

6326113

Anno Accademico 2019-2020

CONTENTS

1

INTRODUCTION

1.1 PROJECT ASSIGNMENT

The scope of this project is to perform a static analysis of the Clang compiler source code available at https://llvm.org/, https://clang.llvm.org/. In details, the project consists in:

- Analyze the C/C++ source code for the Clang project, using different tools for static analysis. The minimum number of tools that shall be selected is 2, and mandatorily it shall be used Understand++ and Clang static analyzer.
- Discuss the output of the different tools and their performance.

Some possible tools for static analysis are:

- Understand++ `https://scitools.com/student/`
- SonarCube `https://www.sonarqube.org/`
- Cert C Rosechecker (also available pre-installed in a Virtual Machine) `https://www.cert.org/secure-coding/tools/rosecheckers.cfm`
- Clang static analyzer
- Cppcheck
- Many others can be retrieved from:
  - `https://www.owasp.org/index.php/Source_Code_Analysis_Tools`
  - `https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#C,_C++`

Depending on the characteristics of the selected tool, it is recommended to comment on:

- the output of the static analyzers with respect to the computed metrics
- compliance to coding rules as MISRA, CERT C, ISO/IET 17961
- correct/missed/false detection.

It is recommended to compare the output of the tools with the information that is already available about the source code and provided by the developers, especially in terms of existing weaknesses of the software.

1.2 OVERVIEW

The scope of this work is to analyze the Clang compiler with a set of static analyzer tools, in order to detect violations to common and accepted coding rules (such as MISRA) and security weaknesses such as the ones pointed in the CWE (Common Weaknesses Enumerator).

Several tools were used for this purpose:

- Understand
- Clang Static Analyzer
- CppChecker
- Flawfinder
- Sonarqube
- Rosechecker

Unfortunately, not all of them were applicable for this work, due to the complexity of the project's architecture or the inflexibility of the tool. After collecting results from this tool, these were then compared in terms of:

- Violations found
- Performances
- Rules used to detect violations
- Easiness of the tool

1.3 STATIC ANALYSIS

Static Analysis is a technique used to analyse softwares without actually executing them.
In general this methodology relies on tools that inspect the source code in order to detect violations with respect to a set of well-defined rules. These tools usually operate by checking the syntax of the code, the semantic, the execution flow...

There are several advantages when adopting this technique:

- First of all, by checking the actual source code, it is possible to identify the direct cause of a vulnerability/bug
- If it is used during the design/development process of a software, it improves its cleanness and correcteness
- The analysis is done with (almost) zero interactions by the human operator

The tools used to perform the analysis can be distinguished with the respect to the phase in which the analysis is performed:

- Unit Level
  - → The analysis takes place within a specific program (or a part of it) without taking into account interactions with other programs
- Technology Level
  - → Analysis takes into account the interactions between unit programs, having a more general overview of a project
- System Level
  - → The analysis consider the interaction between unit programs but without being limited to a specific technology
- Business Level
  - → The analysis also takes into account aspects related to business processes implemented in the software system

In our work we are interested in **Unit Level Analysis**.

1.4 LLVM-CLANG COMPILER

The LLVM compiler infrastructure project is a "collection of modular and reusable compiler and toolchain technologies" used to develop compiler front ends and back ends [1]. It is a middle-layer between the frontend (C, C++, Python...) and the backend (low-level hardware-dependent assembly). The high-level source code is translated into LLVM bitcode, where optimization and analysis is performed before being translated to low-level code.
The Clang compiler is a C/C++ (and several others) compiler frontend that uses the LLVM infrastracture.
The project is structured in a complex hierarchy of directories and files, referencing each others. This was one of the two reasons that forced us to work on a sub-part of the project: the **tools/libclang** directory. The other reason was that some files made some of the static analyzers crash in unexpected manners, probably because of some sort of overflow.

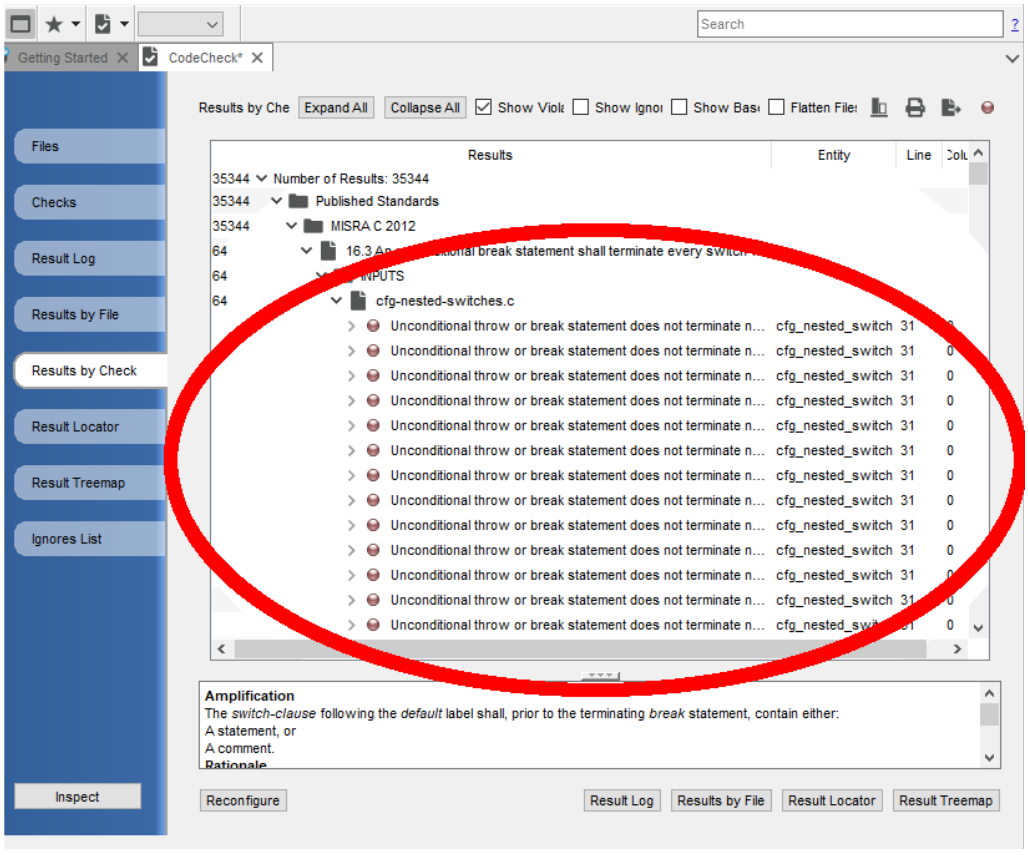

Figure 1: As we can see from this example, the same warning is displayed multiple times. This is most likely an overflow on the specific check

2

CLANG ANALYSIS

2.1 INTRODUCTION

In this chapter it will be described the analysis process for all the tools used.

Understand is indeed the tools that gives the most accurate results in terms of checks, since it incorporates C/C++ MISRA standards, a beta version of the **CLang Static Analyzer**, which is a static analysis tool provided by the LLVM developers, and many other quality checks offered by SciTools itself.
A simpler but also quite effective tool is **Cppcheck** which is designed to "provide unique code analysis to detect bugs and to focus on detecting undefined behaviour and dangerous coding constructs" [2]. Also, as pointed by the developers, its main focus is to "detect only real errors in the code (i.e. have very few false positives)". Cppcheck refers to the *Common Weakness Enumeration* standard for the analysis, a formal list of security issues published by the MITRE institute. It is also possible to check MISRA-C project compliance but it requires to buy the standard so this feature was not used.
The last used tool is **flawfinder** which puts its focus more on security flaws rather than quality issues. This tool incorporates an option to run the analysis in order to detect possible false positives in an automated manner. This tools uses the CWE standard as Cppcheck does.
Other tools such as **SonarQube** and **Cert C Rosechecker** were used but due to their characteristics they were unusable for our purpose.

2.2 UNDERSTAND

BIBLIOGRAPHY

[1] Wikipedia - *https://en.wikipedia.org/wiki/LLVM* (Cited on page 6.)

[2] Cppcheck - *http://cppcheck.sourceforge.net/* (Cited on page 7.)