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 infrastructure.
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.

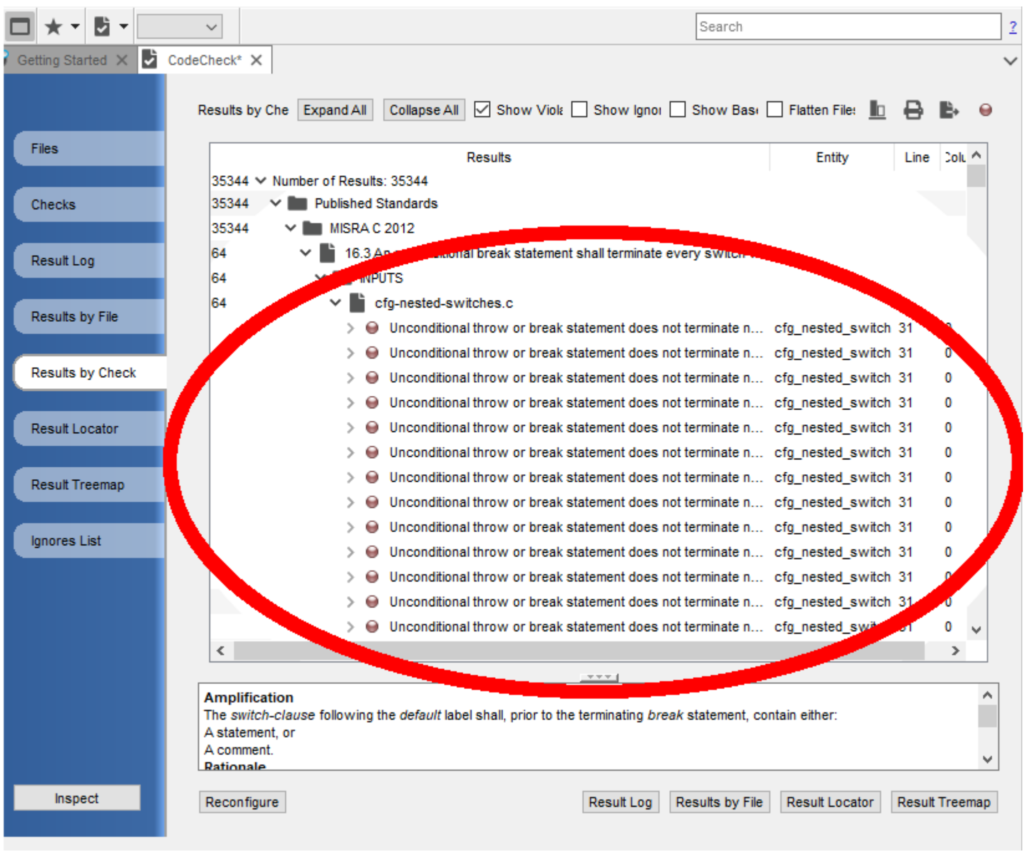

Figure 1: As we can see from this example, the same warning is displayed multiple times. This is most likely an overflow on the specific check

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 ANALYSIS METHODOLOGY

The LLVM Clang compiler was analized with all the tools listed above. Since some issues arose while analyzing the whole project, as it was pointed in the previous section, a representative subset of it was chosen. In particular the folder **src/tool/libclang** was analyzed because it has been observed that the source files in this folder contained much of the compiler logic. This was the input folder for all the static analysis tools used.

After the output was produced, the second phase of the analysis can start. Since the output format of the various tool is hetherogenous, it was necessary to convert them in excel sheets in order to collect evidences about what files were the most vulnerable/contained more bugs.

2.3 UNDERSTAND

Understand is a very powerful tool for static analysis that can be used to analyze software written in multiple languages suchs as Java, Ada, Cobol, Python, C/C++... Among the tools used, it is the only one that comes with a nice and user-friendly user interface that allows users to navigate through the software files.

2.3.1 *Understand Project*

First of all, it must be created an *Understand Project*. In this first step you are asked to select the language of the software (C/C++ in our case study) and the directories to analyze.

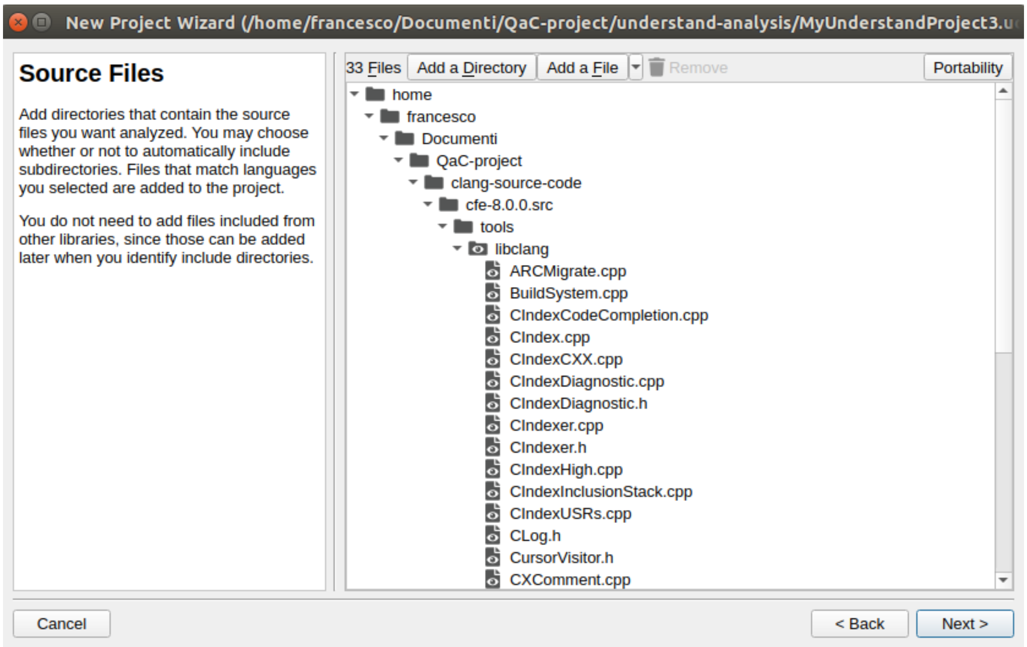

Figure 2: The whole subdirectory tools/libclang is imported in the Understand project in order to run the analysis.

When the files are loaded in the program, the analysis can be run simply by opening the *codecheck perspective* and selecting which standard should guide it.

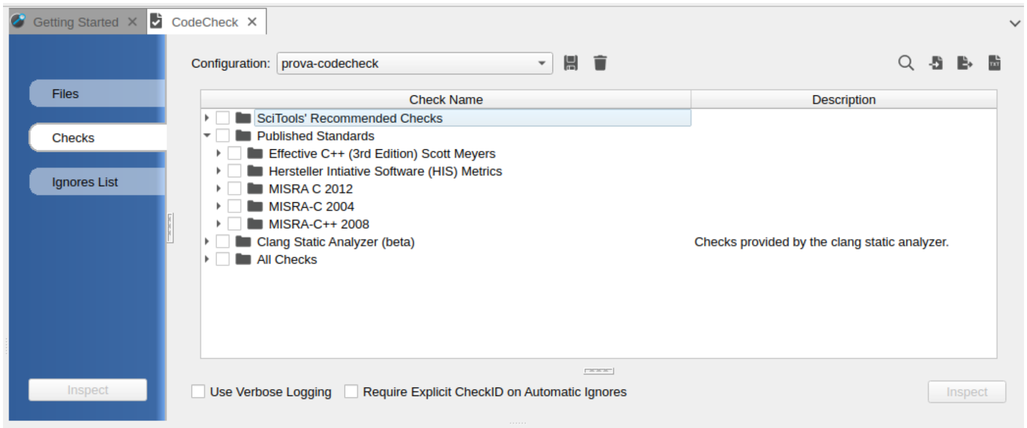

Figure 3: The MISRA standard is incorporated in Understand, as well as the Clang Static Analyzer. Generic checks are also offered by the tool as *SciTools Recommended Checks* and *AllChecks*, some of which are redundant.

- *SciTools' Recommended Checks* - This is a small set (17 items) of generic good programming rules
- *Published Standards* - This section contains the published standards supported by Understand
  - It was used the MISRA-C++ 2008 due to the nature of the source files (.cpp) and because one of the goals of this project was to check the Clang compiler compliance to MISRA rules.
- *Clang Static Analyzer* - Is an implementation of the tool incorporated in Understand.
- *All Checks* - This is a collection of checks which consists of generic good programming rules and some of the MISRA rules. Despite its name, not all the checks are included for real, this is the reason why it is not correct to use only this option for a consistent analysis.

2.3.2 *Understand Output Format*

When the analysis ends, it is possible to navigate through different perspectives of what has been observed. For example it is possible to list results *by file*, in order to check which issues are present in each file (and at which line of code) and what files contains the most issues. Another possibility is to display result *by check*, that is: for each rule (e.g. MISRA) how many times it has been violated and where (in terms of files).
Two very interesting features offered by Understand are the:

- Result Locator
- Result Treemap

The first one offers the possibility to navigate through the findings, filtering them by file, by violation and some other options, giving the possibility to jump to the desired *vulnerable* line of code in the source file. The result treemap instead gives you a graphic representation of the files vulnerabilities, in terms of criticity and quantity. These characteristics can be viewed graphically using colored boxes, where the meaning of the color/dimension of the boxes can be defined by the user.
Mastering the options of these two powerful features gives to the user much more control of the analysis and a wider perspective of the whole project quality.

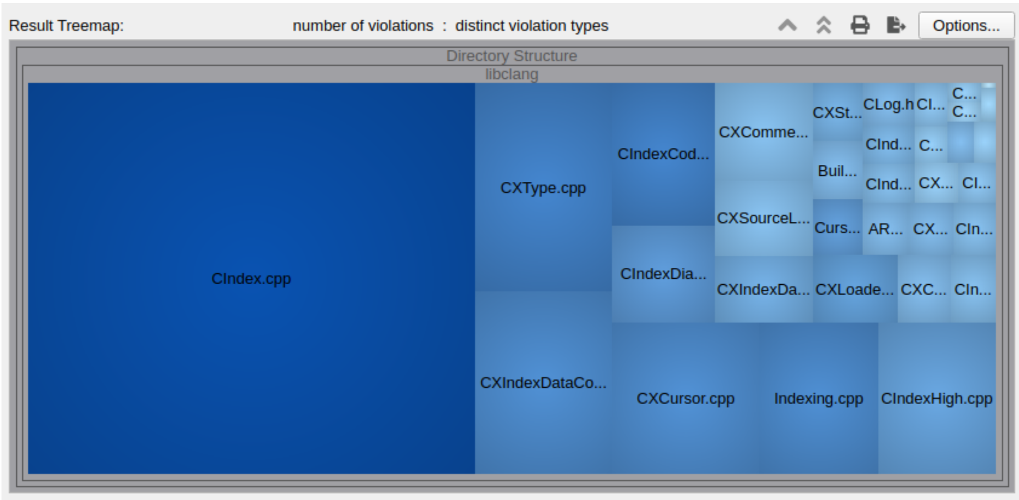

Figure 4: Result Treemap view.

All the output perspectives can be exported in suitable formats (e.g. Treemap is exported in .png files while lists of violations are exported in .txt or .html files) that facilitate the second phase of the analysis.

BIBLIOGRAPHY

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

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