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

6345593 - 6326113

Anno Accademico 2018-2019

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 analysis tools, in order to detect violations to standards and to common 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 the results, 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 analyze 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 an analysis can be distinguished with 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... code) and the backend (low-level hardware-dependent assembly). The high-level source code is translated into *LLVM bitcode*: an intermediate language (such as the JVM bytecode) 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.

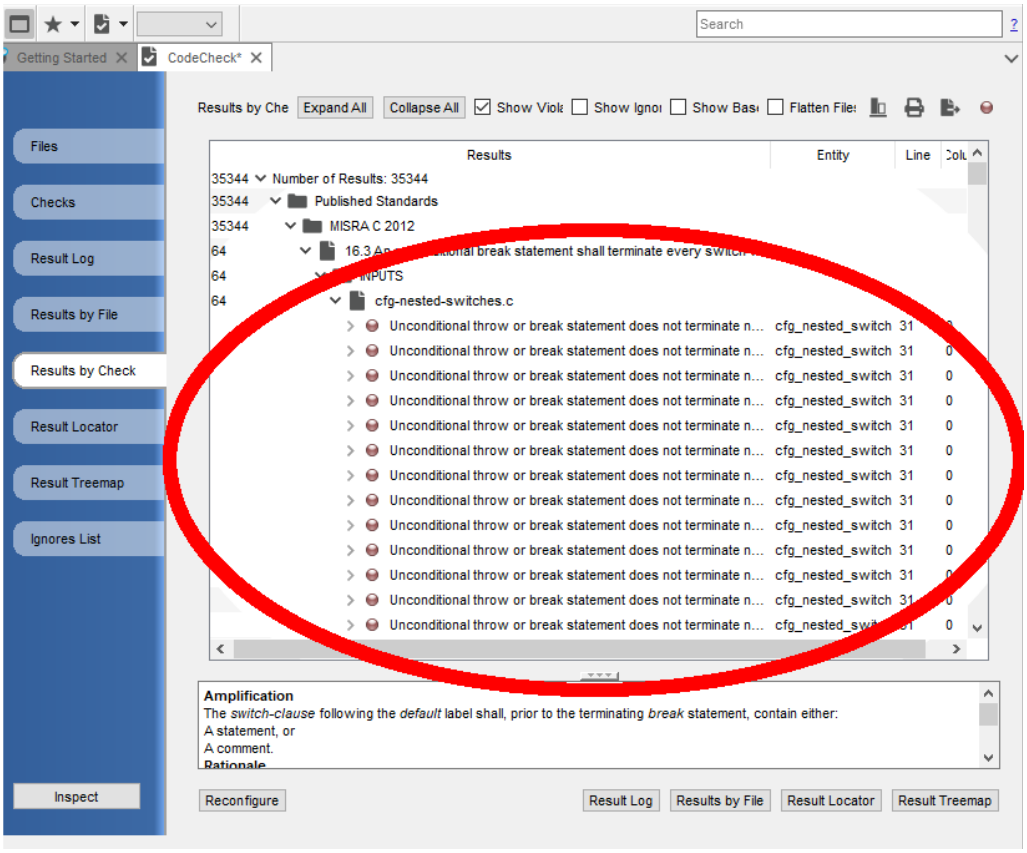

Figure 1: As we can see from this example, the same warning is displayed multiple times. This is most likely an overflow bug on the specific check

2

CLANG ANALYSIS

2.1 INTRODUCTION

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

Understand is indeed the tool 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 quality and 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 there were some technical problems 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/tools/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.

After the output was produced, the second phase of the analysis could start. Since the output format of the various tool is hetherogenous, it was necessary to convert them in files of the same format: data were imported in Excel sheets in order to collect evidences about what files were the most vulnerable/contained more bugs and what kind of bugs/quality issues were the most common.

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 browse the analysis results with different views.

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/source files to be analyzed.

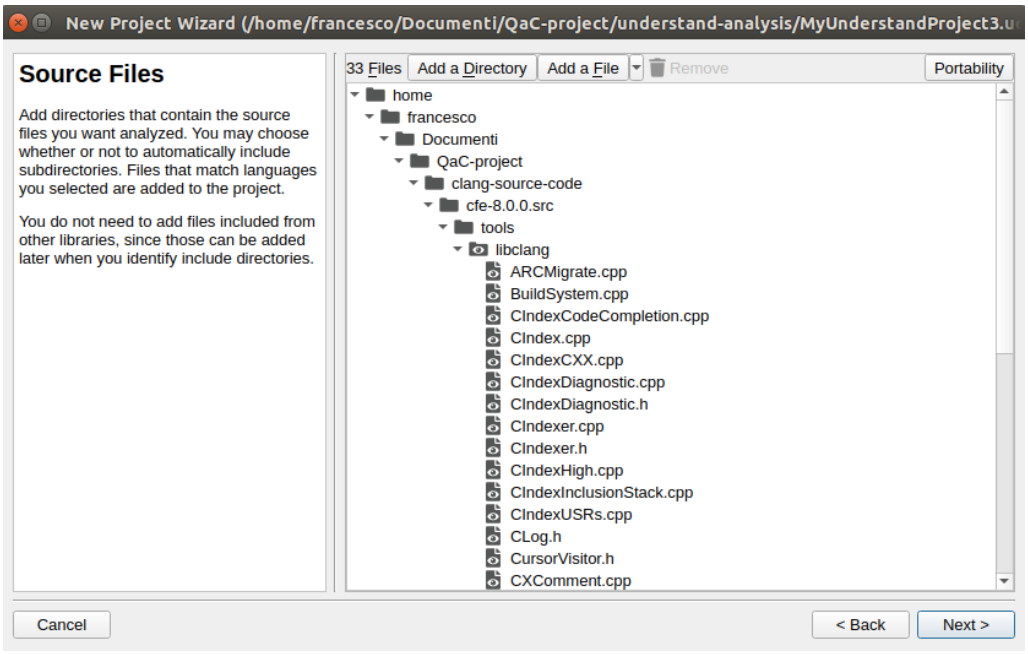

Figure 2: The whole subdirectory tools/libclang is imported in the Understand project in order to run the analysis.

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.

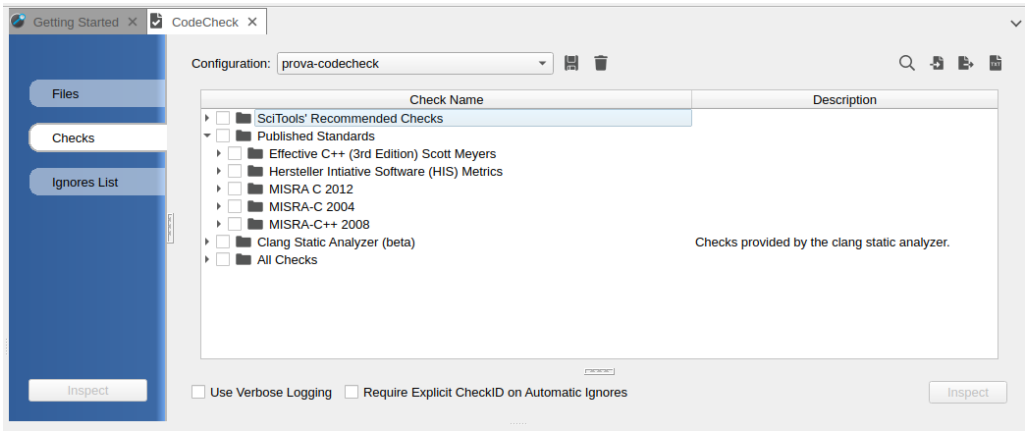

Figure 3: The MISRA standard is incorporated in Understand, as well as the Clang Static Analyzer. Generic checks are also offered by the tool as *SciTools Recommended Checks* and *AllChecks*, some of which are redundant.

- *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* - It's an implementation of the Clang Static Analyzer tool, included 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's incorrect 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 what issues are present in each file (and at which line of code) and what files contain the most issues. Another possibility is to display results *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 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 vulnerabilities found in the files, 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.

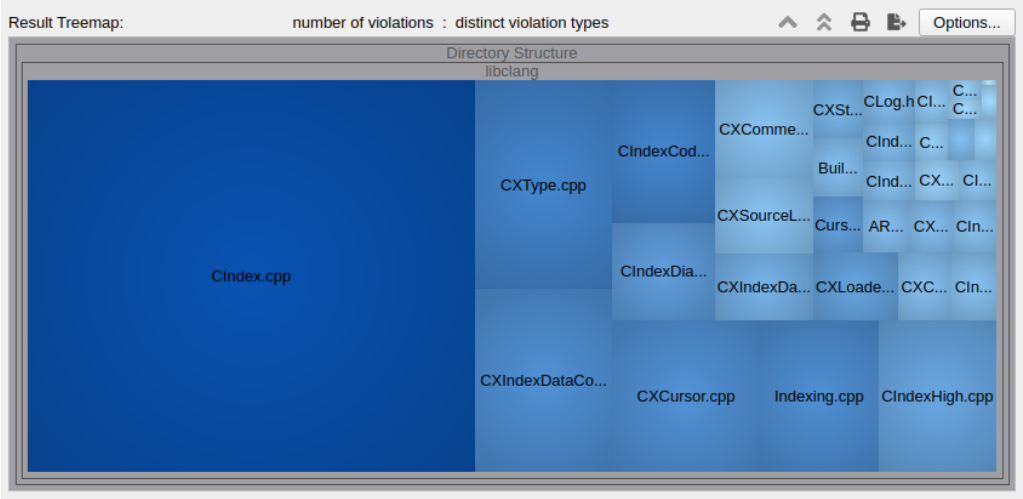

Figure 4: Result Treemap view.

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 facilitated the second phase of the analysis.

2.3.3 *Understand Results*

As it has been said in the introduction, multiple analyses with different checks were run:

a) MISRA-C++ 2008

b) SciTools' Recommended Checks

c) All Checks

d) Clang Static Analyzer

a) This check is based on the MISRA-C++ 2008 standard, which is a standard developed for the quality check of C++ source files.
Results have been sorted by file and by MISRA rule. After that a compact view of these was produced showing the amount of violations for each MISRA and for each file.
Analyzing the reports it can be observed that:

- The total number of violations in the **libclang** folder is 8450
- The first three rules that were violated the most are:
  1. MISRA08_7-1-1 - **A variable which is not modified shall be const qualified** - 1845 violations.
     If a variable does not need to be modified, then it shall be declared with const qualification so that it cannot be modified. A non-parametric variable will then require its initialization at the point of declaration. Also, future maintenance cannot accidentally modify the value
  2. MISRA08_6-4-1 - **An if condition construct shall be followed by a compund statement. The else keyword shall be followed by either a compound statement or another if statement** - 1239 violations.
     If the bodies of these constructs are not compound statements, then errors can occur if a developer fails to add the required braces when attempting to change a single statement body to a multistatement body. Requiring that the body of these constructs shall be a compound statement (enclosed within braces) ensures that these errors cannot arise
  3. MISRA08_0-1-10 -**All defined functions called** - 733 violations.
     Functions or procedures that are not called may be symptomatic of a serious problem, such as missing paths
- The first three files that contain the most violations are:
  1. CIndex.cpp - 3828 violations
  2. CXType.cpp - 757 violations
  3. CXCursor.cpp - 558 violations

Looking at the complete report, considerations can be made on the following aspects:

- Some of the violations found (e.g. MISRA08_0-1-10) include for sure some false-positive, due to the fact that the analysis was performed on a small subset of the complete project. Maybe the violation count for these rules could be reduced by performing a more comprehensive analysis
- The violations distribution is reasonable, that is that most of the files have a similar issues count and the same can be said for MISRA rules

- There is an exception both for files and for MISRA:

FILE: CIndex.cpp (3828 violations wrt 8450 total violations)

MISRA: MISRA08_7-1-1 (1845 violations wrt 8450 total violations)

- Using the Result Treemap feature it is possible to observe that the NumberOfViolations/CountLineCode ratio varies between [0.27 - 1.75]
- Combining the use of the Result Treemap and Result Locator it is observable that the first 3 files in terms of NumberOfViolations/CountLineCode ratio are relatively small files compared to the others

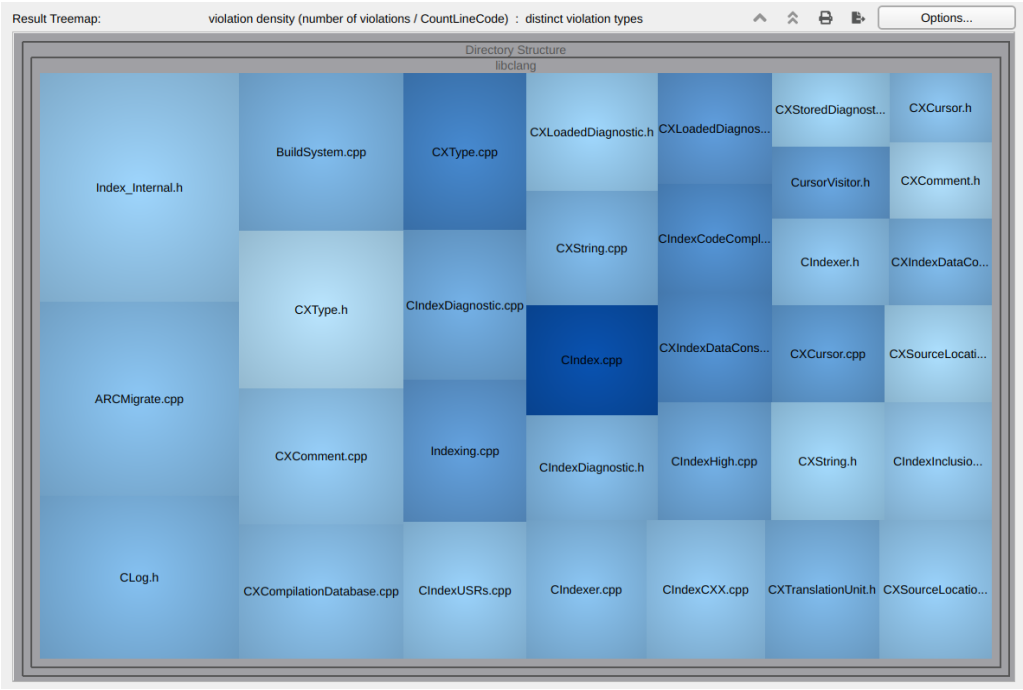

Figure 5: Result Treemap where files are sorted by NumberOfViolations/CountLineCode ratio. Darker boxes indicates more distinct violation types while wider boxes indicate higher ratio.

b) SciTools Recommended Checks is a set of 17 quality checks based on code quality conventions that does not follow any precise standard.
Results has been sorted by files and by the checkID provided by Understand. The postprocess phase was very similar to the previous one in order to obtain a compact view of data.

- The total number of violations in the **libclang** folder is 3299 (roughly 1000 violations less than the MISRA-check)
- The first three checks that were violated the most are:
  1. RECOMMENDED_16 - **Each variable declaration should have a comment** - 1774 violations
  2. RECOMMENDED_13 - **Every defined function shall be called at least once** - 733 violations
  3. RECOMMENDED_08 -**All fixed values will be defined constants** - 405 violations
- The first three files that contains the most violations are:
  1. CIndex.cpp - 1472 violations
  2. CXType.cpp - 266 violations
  3. CXCursor.cpp - 250 violations

As it can be noticed by these data, the three files that have the most violations are the same as in the MISRA analysis.
Looking at the complete dataset and comparing it to the previous one, the following considerations can be made:

- The *RECOMMENDED_13* plays a similar role as the MISRA08_0-1-10. We can expect that when the analysis runs on the whole project the count number of this violation (*Every defined function shall be called at least once*) is reduced
- The *RECOMMENDED_16* check may seem too exaggerated but, if we think to big project as LLVM-Clang is, where multiple teams cooperate writing different chunks of code, this check become reasonable
- The violations distribution with respect to files is quite odd. It can be observed that the average of the violations count is approximately 100. If we exclude the CIndex.cpp file the average drops approximately to 55

- There is an exception both for files and for check, as we saw in the previous section:

FILE: CIndex.cpp (1472 violations wrt 3299 total violations)

MISRA: RECOMMENDED_16(1774 violations wrt 3299 total violations)

- Using the Result Treemap feature it is possible to observe that the NumberOfViolations/CountLineCode ratio varies between [0.09 - 0.39]

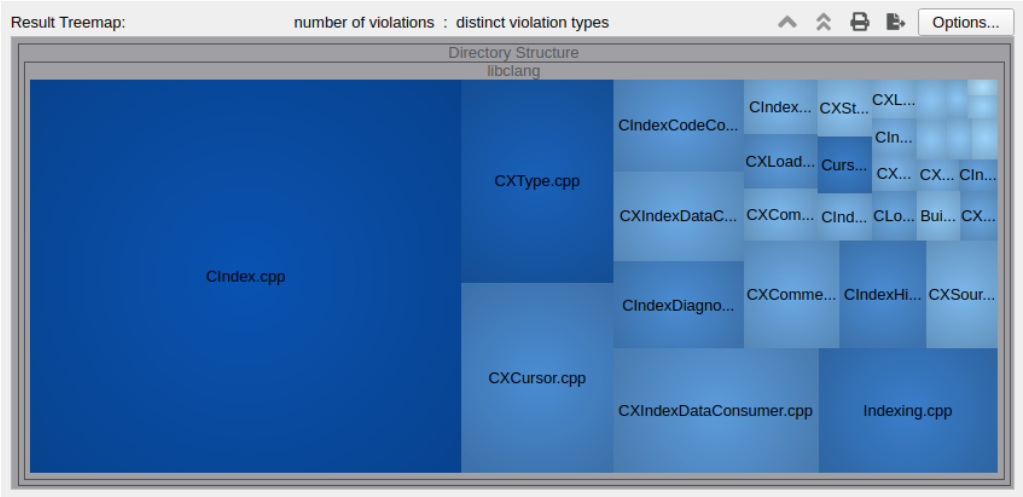

Figure 6: Result Treemap where files are sorted by NumberOfViolations. As you can see the file CIndex.cpp covers almost half of the area.

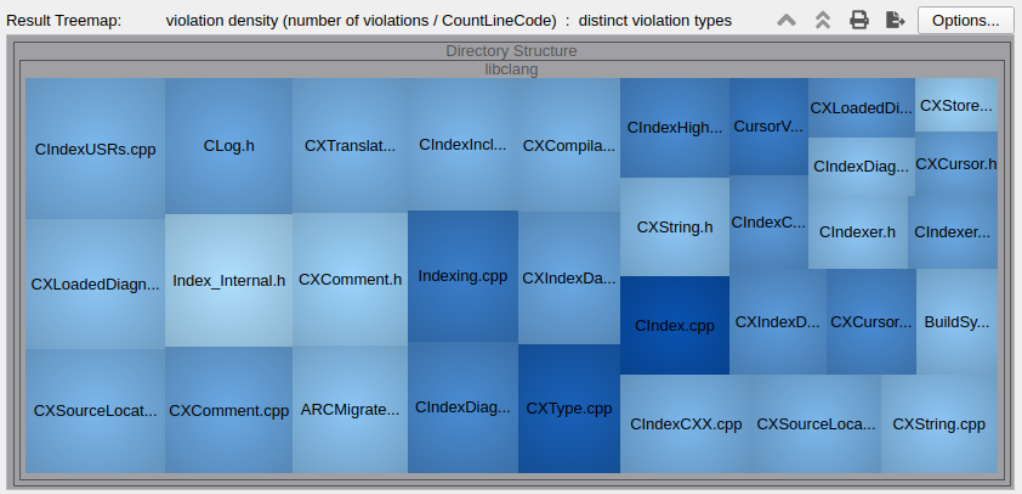

Figure 7: Despite the result in the previous figure, the NumberOfViolations/LineOfCode ratio is quite homogeneous.

c) The *AllChecks* check set is the biggest among the provided ones. It includes most of the checks already seen in the previous sections, plus some completely new checks.
Results have been sorted by files and by checkID, an indentifier provided by Understand. The postprocess phase was very similar to the previous one in order to obtain a compact view of data.

- The total number of violations in the **libclang** folder is 30604 (indeed a very huge number compared to the previous ones)
- The first three checks that were violated the most are:
  1. CPP_L000 - **Calls to COTS (Commercial Off The Shelf) library functions that might throw an exception, must be enclosed in a try block** - 5921 violations
  2. CPP_I005 - **Identifier name reuse** - This check is concerned about variables' names. It requires that, in the same file, a different name is given to every different variable - 3546 violations
  3. CPP_V006 -**A variable which is not modified should be const qualified** - same as in the MISRA analysis - 1845 violations
- The first three files that contains the most violations are:
  1. CIndex.cpp - 14004 violations
  2. CXType.cpp - 2311 violations
  3. CXIndexDataConsumer.cpp - 2078 violations

It is noticeable that, with this set of checks, the third file with most violations is *CXIndexDataConsumer.cpp* instead of *CXCursor.cpp*.
Let's now proceed with some considerations on these results:

- Since this analysis includes the checks of the previous ones, we assume that there are false-positives but they are caused mostly by the fact that the inspection was done on a subpart of the project. We don't have evidences of false-positives among the most common violations (see the complete Excel report for more details)
- The most frequent violation is of a new kind. It is an important vulnerability that was pointed by none of the other analyses. This is quite odd since it is an important rule, well known and well accepted by the developer community

- The violations distribution with respect to files is similar to the previous ones
- The violations distribution with respect to checks tends to a normal distribution. This behaviour is gradually lost when the violations count goes over approximately 400)
- As well as the previous analyses, there is an exception both for files and for checks:

FILE: CIndex.cpp (14004 violations wrt 30604 total violations, almost half of the total)

MISRA: CPP_L000(5921 violations wrt 30604 total violations. Recall that the second most common check had more than 2000 violations less)

- Using the Result Treemap feature it is possible to observe that the NumberOfViolations/CountLineCode ratio varies between [1 - 5.45]

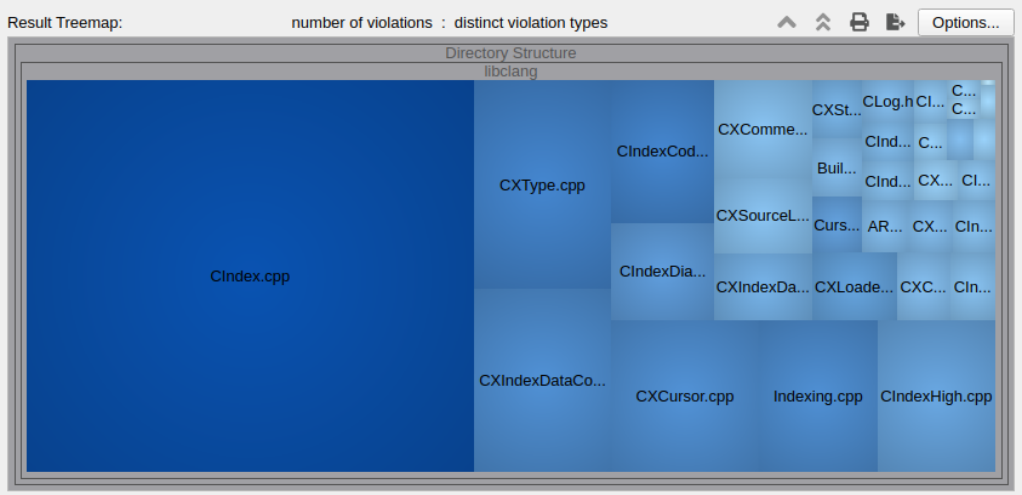

Figure 8: The CIndex.cpp file covers almost half of the total area also in this figure.

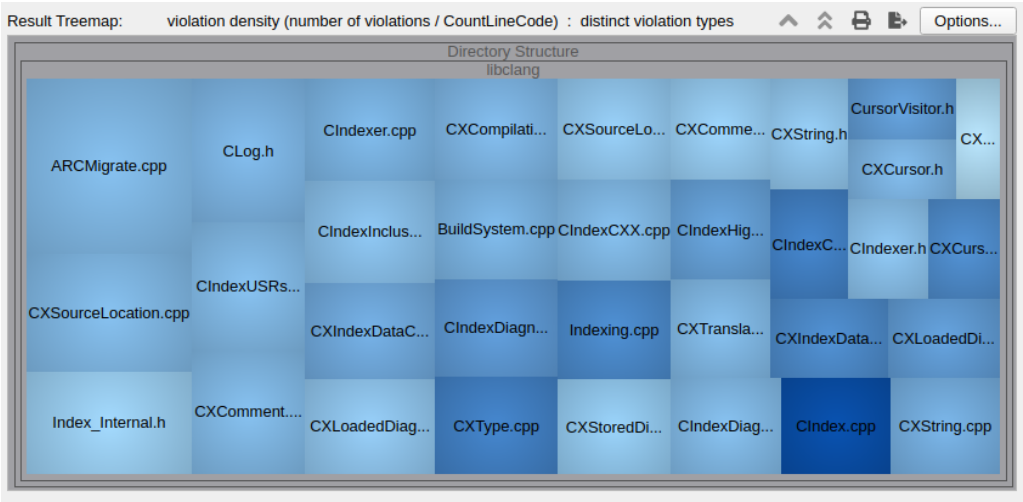

Figure 9: Even if the proportion of the max/min of the ratio is similar to the previous, the ratio distribution itself is quite sparse (it ranges between 1 and 5.5), as well as the first analysis.

d) The *Clang Static Analyzer* check set is an implementation of the Clang Static Analyzer tool, developed as part of the LLVM-Clang project.
The Clang Static Analyzer itself has not been used since it requires the build of the *whole* project to scan it. Moreover the increased knowledge of Understand and the presence of this set of checks embedded in the tool led us to the choice of using this implementation rather than the standalone tool.

This analysis did not find any violation. This seems reasonable since the metrics and quality practices chosen during the development of the LLVM-Clang compiler are probably the same metrics and checks implemented in the analyzer.

2.3.4 *Reports Summary*

In this conclusive section for the Understand analyzer, it will be presented a summary of the most important observations along with some cross-checks. Summaries of the data collected throughout these analyses are reported at the end of the *Understand section*.

- All the analyses agreed that CIndex.cpp is the file with the largest violations count which is very large compared to the other files
- CXType.cpp is always the second file with the largest violations count. There are variations from the third position onwards, for example: CXCursor.cpp appears 2 times in the third position, while CXIndexDataConsumer.cpp appears 1 time
- Looking at the $violationsCount/countLineCode$ ratio, it is observable that across the three analysis the max/min proportion is very similar ($\approx 5.5 \pm 1$)
- It seems reasonable that using the metrics chosen by the project developers to analyze the source code no violations are found while, when different metrics are used, there are some violations
- In some cases the violated metrics could be considered as a warning (e.g. to write a comment for every variable helps to keep the project well documented) while in some other cases (e.g. do not use try-catch construct) they are critical issues

- Very likely, in these analyses, there are some false positives. This is because of the fact that the analyses were not performed on the whole project (e.g. functions that are never called in the analyzed modules could be called somewhere else)
- Some duplicated violations were reported when performing the analysis using the MISRA set. This is likely to be attributable to a possible overflow issue that has been noted also in the very first phase of the use of Understand. The duplicated issues were removed in order to clean the dataset. No duplications were found using the other sets of rules

2.3.5 *Understand Performances*

Understand is a very powerful tool. With its UI it gives a lot of informations about the performed analysis (e.g. result Treemap, result Locator, sort by check/file...). On the other hand, it is not a lightweight program: we experienced unexpected crashes during analyses, issues duplication and a slow execution time in general. These are some of the main reasons that forced us to work on a subset of the LLVM-Clang compiler.
Compared to the tools that will be presented in the next sections, this is indeed the most efficient in terms of *informations* and *versatility* (the offered perspectives are a very powerful tool to navigate through the data). On the other hand, in terms of speed performance, it was the slowest one.

MISRA	Tot
MISRA08_0-1-1	1
MISRA08_0-1-10	733
MISRA08_0-1-11	139
MISRA08_0-1-3	59
MISRA08_0-1-4	51
MISRA08_0-1-5	4
MISRA08_0-1-7	48
MISRA08_10-3-2	24
MISRA08_11-0-1	5
MISRA08_12-1-2	2
MISRA08_12-1-3	11
MISRA08_14-7-1	10
MISRA08_16-0-1	9
MISRA08_16-0-2	30
MISRA08_16-0-3	9
MISRA08_16-0-4	27
MISRA08_16-0-6	9
MISRA08_16-0-7	3
MISRA08_16-2-1	36
MISRA08_16-2-2	27
MISRA08_16-3-1	5
MISRA08_16-3-2	9
MISRA08_17-0-1	1
MISRA08_17-0-2	3
MISRA08_18-0-1	2
MISRA08_18-0-3	1
MISRA08_18-4-1	90
MISRA08_2-10-1	300
MISRA08_2-10-2	55
MISRA08_2-10-4	11
MISRA08_2-10-5	13
MISRA08_27-0-1	4
MISRA08_2-7-3	11
MISRA08_3-1-1	95
MISRA08_3-1-2	3
MISRA08_3-2-1	15
MISRA08_3-2-3	11
MISRA08_3-2-4	53
MISRA08_3-3-1	386
MISRA08_3-3-2	1
MISRA08_3-9-2	176
MISRA08_4-5-1	2
MISRA08_4-5-3	1
MISRA08_5-2-10	33
MISRA08_6-3-1	38
MISRA08_6-4-1	1239
MISRA08_6-4-2	34
MISRA08_6-4-5	695
MISRA08_6-4-6	134
MISRA08_6-4-8	2
MISRA08_6-5-1	63
MISRA08_6-5-2	52
MISRA08_6-5-4	2
MISRA08_6-6-2	3
MISRA08_6-6-4	4
MISRA08_6-6-5	563
MISRA08_7-1-1	1845
MISRA08_7-1-2	148
MISRA08_7-3-1	528
MISRA08_7-3-4	52
MISRA08_7-3-5	2
MISRA08_7-5-4	21
MISRA08_8-0-1	20
MISRA08_8-4-2	2
MISRA08_8-4-4	421
MISRA08_8-5-1	23
MISRA08_9-3-1	3
MISRA08_9-3-3	37
MISRA08_9-5-1	1
Total	8450

File	Tot
ARCMigrate.cpp	46
BuildSystem.cpp	101
CIndex.cpp	3828
CIndexCodeCompletion.cpp	346
CIndexCXX.cpp	55
CIndexDiagnostic.cpp	249
CIndexDiagnostic.h	42
CIndexer.cpp	59
CIndexer.h	30
CIndexHigh.cpp	199
CIndexInclusionStack.cpp	24
CIndexUSRs.cpp	58
CLog.h	70
CursorVisitor.h	62
CXComment.cpp	246
CXComment.h	7
CXCompilationDatabase.cpp	105
CXCursor.cpp	558
CXCursor.h	38
CXIndexDataConsumer.cpp	518
CXIndexDataConsumer.h	120
CXLoadedDiagnostic.cpp	131
CXLoadedDiagnostic.h	20
CXSourceLocation.cpp	162
CXSourceLocation.h	14
CXStoredDiagnostic.cpp	23
CXString.cpp	69
CXString.h	9
CXTranslationUnit.h	28
CXType.cpp	757
CXType.h	3
Index_Internal.h	22
Indexing.cpp	451
Total	8450

Figure 10: Summary of the MISRA checks

File	Tot
ARCMigrate.cpp	9
BuildSystem.cpp	18
CIndex.cpp	1472
CIndexCodeCompletion.cpp	104
CIndexCXX.cpp	22
CIndexDiagnostic.cpp	99
CIndexDiagnostic.h	10
CIndexer.cpp	11
CIndexer.h	11
CIndexHigh.cpp	82
CIndexInclusionStack.cpp	14
CIndexUSRs.cpp	35
CLog.h	19
CursorVisitor.h	27
CXComment.cpp	90
CXComment.h	9
CXCompilationDatabase.cpp	33
CXCursor.cpp	250
CXCursor.h	16
CXIndexDataConsumer.cpp	221
CXIndexDataConsumer.h	102
CXLoadedDiagnostic.cpp	34
CXLoadedDiagnostic.h	15
CXSourceLocation.cpp	67
CXSourceLocation.h	12
CXStoredDiagnostic.cpp	6
CXString.cpp	27
CXString.h	7
CXTranslationUnit.h	14
CXType.cpp	266
Index_Internal.h	4
Indexing.cpp	193
Total	3299

Reccomended Check	Tot
RECOMMENDED_00	11
RECOMMENDED_01	57
RECOMMENDED_02	15
RECOMMENDED_04	3
RECOMMENDED_05	111
RECOMMENDED_06	4
RECOMMENDED_07	30
RECOMMENDED_08	405
RECOMMENDED_10	12
RECOMMENDED_12	1
RECOMMENDED_13	733
RECOMMENDED_14	143
RECOMMENDED_16	1774
Total	3299

Figure 11: Summary of the SciTools Recommended Checks

File	Tot
ARCMigrate.cpp	171
BuildSystem.cpp	648
CIndex.cpp	14004
CIndexCodeCompletion.cpp	1092
CIndexCXX.cpp	183
CIndexDiagnostic.cpp	805
CIndexDiagnostic.h	172
CIndexer.cpp	162
CIndexer.h	119
CIndexHigh.cpp	754
CIndexInclusionStack.cpp	125
CIndexUSRs.cpp	248
CLog.h	175
CursorVisitor.h	246
CXComment.cpp	766
CXComment.h	53
CXCompilationDatabase.cpp	347
CXCursor.cpp	1821
CXCursor.h	475
CXIndexDataConsumer.cpp	2078
CXIndexDataConsumer.h	558
CXLoadedDiagnostic.cpp	467
CXLoadedDiagnostic.h	110
CXSourceLocation.cpp	582
CXSourceLocation.h	78
CXStoredDiagnostic.cpp	141
CXString.cpp	230
CXString.h	64
CXTranslationUnit.h	86
CXType.cpp	2311
CXType.h	11
Index_Internal.h	39
Indexing.cpp	1483
Total	30604

Check	Tot
AC_01	21
CPP_026	33
CPP_B001	65
CPP_C000	11
CPP_C003	652
CPP_C006	2
CPP_C009	11
CPP_C013	38
CPP_C014	33
CPP_C015	63
CPP_C016	1289
CPP_C017	1323
CPP_C020	52
CPP_C021	2
CPP_C023	3
CPP_C024	57
CPP_C025	4
CPP_C026	4
CPP_C027	5
CPP_C029	563
CPP_C031	130
CPP_C032	2
CPP_C033	687
CPP_C034	1
CPP_C036	382
CPP_D001	15
CPP_D002	20
CPP_D007	373
CPP_D009	2
CPP_D011	386
CPP_D012	13
CPP_D013	53
CPP_D014	386
CPP_D016	85
CPP_D022	376
CPP_D023	2
CPP_D025	388
CPP_D026	11
CPP_D029	12
CPP_D031	4
CPP_E004	15
CPP_E015	2
CPP_E016	1

Check	Tot
CPP_F000	12
CPP_F001	12
CPP_F002	3
CPP_F003	731
CPP_F004	1082
CPP_F006	419
CPP_F007	5
CPP_F009	118
CPP_F010	111
CPP_F011	373
CPP_F014	37
CPP_F015	19
CPP_F019	148
CPP_F020	2
CPP_F022	121
CPP_F023	24
CPP_H003	125
CPP_H006	95
CPP_H007	31
CPP_I000	11
CPP_I001	333
CPP_I002	1374
CPP_I003	34
CPP_I004	529
CPP_I005	3546
CPP_I007	4
CPP_I008	44
CPP_I010	55
CPP_I011	72
CPP_I012	13
CPP_I013	31
CPP_I015	139
CPP_I016	8
CPP_L000	5921
CPP_L001	1
CPP_L003	4
CPP_L004	1
CPP_M000	90
CPP_P001	9
CPP_P003	269
CPP_P004	30
CPP_P006	5
CPP_P008	24

Check	Tot
CPP_P009	9
CPP_P012	12
CPP_P013	27
CPP_P015	209
CPP_P017	9
CPP_P019	30
CPP_P020	27
CPP_P021	3
CPP_P022	36
CPP_P023	1
CPP_P024	9
CPP_P025	3
CPP_P026	9
CPP_P029	14
CPP_S000	1
CPP_T000	176
CPP_U001	48
CPP_U002	48
CPP_U003	139
CPP_U005	9
CPP_U006	5
CPP_V000	405
CPP_V001	86
CPP_V002	12
CPP_V003	69
CPP_V004	51
CPP_V005	12
CPP_V006	1845
CPP_V007	143
CPP_V009	52
CPP_V010	1775
CPP_V011	23
CPP_V012	24
EFFECTIVECPP_22	69
METRIC_00	149
METRIC_01	61
METRIC_02	1136
METRIC_03	166
METRIC_04	37
METRIC_05	3
METRIC_06	26
METRIC_07	18
METRIC_08	26
Total	30604

Figure 12: Summary of the AllChecks checks

2.4 CPPCHECK ANALYSIS

Cppcheck is a lightweight, open-source tool for static analysis of C/C++ files. It can be downloaded at `http://cppcheck.sourceforge.net` for Windows (an installer is provided), Mac and Linux distributions:

- sudo apt-get install cppcheck (Linux)
- brew install cppcheck (Mac)

Once installed, the tool can be run from a shell using the following command:

- cppcheck [OPTIONS] [files or directories]

The [OPTIONS] option offers some sort of customization of the analyses, for example it allows the user to:

- Write results to an xml file
- Print the list of all the available checks
- Print the error list in xml format (on the console)
- Suppress specific warnings
- Define/undefine preprocessor symbol
- And more... (see cppcheck –help for more informations)

The analyzer can be fed with a single file, a list of files, or a whole directory. In this study, for compliance with the *Understand* analysis, the check was done on the **tools/libclang** directory.
The output was written to an xml file and postprocessed with Excel.

2.4.1 *Cppcheck Results*

Cppcheck relies on the MITRE-CWE (Commone Weakness Enumeration) list to detect vulnerabilities and quality issues, plus some generic quality checks based on standard code quality practices.
This list associates IDs to code issues categories that can be refined by users (i.e. it was not possible to find a 1:1 correspondence between the checkNames provided by Cppcheck and the checkNames in the CWE list, although the CWE IDs were identical).
In this analysis issues were found related to four CWE categories:

(CWE-398) **Code Quality** - This category represents one of the phyla in the Seven Pernicious Kingdoms vulnerability classification. It includes weaknesses that do not directly introduce a weakness or vulnerability, but indicate that the product has not been carefully developed or maintained. According to the authors of the Seven Pernicious Kingdoms, "Poor code quality leads to unpredictable behavior. From a user's perspective that often manifests itself as poor usability. For an adversary it provides an opportunity to stress the system in unexpected ways." [3]

(CWE-561) **Dead Code** - Dead code is source code that can never be executed in a running program. The surrounding code makes it impossible for a section of code to ever be executed [4]

(CWE-563) **Assignment to Variable without Use** - After the assignment, the variable is either assigned another value or goes out of scope. It is likely that the variable is simply vestigial, but it is also possible that the unused variable points out a bug [5]

(CWE-686) **Function Call With Incorrect Argument Type** - This weakness is most likely to occur in loosely typed languages, or in strongly typed languages in which the types of variable arguments cannot be enforced at compilation time, or where there is implicit casting [6]

Moreover, Cppcheck rises a warning about missing files that are included in source cpp files but that are not found in the analyzed directory. This warning tells the user that the analysis will terminate but "results will probably be more accurate if all the include files are found".

Results have been sorted, as for the previous analyses, by files and by the checkName provided by Cppcheck. Again, results were aggregated in Excel files to ease data observation.
Analyzing the reports it can be observed that:

- The total number of violations in the **libclang** folder is 414 (very small number compared to e.g. MISRA violations)
- The first three rules that were violated the most are:
  1. CWE-561 - **Unused Function** - 375 violations
  2. CWE-398 - **No explicit contructor** - 11 violations

3. CWE-398 -**Uninit Member Var** - 10 violations

- The first three files that contains the most violations are:
  1. CIndex.cpp - 158 violations
  2. CXType.cpp - 47 violations
  3. Indexing.cpp - 35 violations

It is reasonable to think that the **Unused Function** violation count includes a lot of false-positives. First of all it is much greater than the second violation count (375 vs. 11), which is suspicious, moreover, as it was said in the previous sections, this is confirmed by the fact that only one directory was analyzed.

2.4.2 *Cppcheck Performance & Comparison with Understand*

Cppcheck is a much faster tool than Understand. The absence of UI and its simpleness play an important role on its performances: it works directly on the directories instead of creating a *project file* that acts as a link from the source files to the tools; on the other hand this simpleness provides less informations on the project issues.
Cppcheck output is displayed only on console by default. Using a proper option it is possible to write the outputs to an xml file even if this reduces the readability of the reports.
Comparing it to Understand, it has been seen that, in this study, Cppcheck did not produced any duplicate issue (same file, same issue, same line of code).

We now present some comparison of the results achieved with Understand and the ones obtained by Cppcheck.
First of all the issues count distribution for checks and files is almost uniformly distributed, at least compared to the MISRA-checks, although there is always the exception of the false positive *unused functions* and the source file *CIndex.cpp*, which is the biggest file in the directory (therefore it contains much more violations with respect to the others).

Some of the Cppcheck CWE-checks are overlapping with the MISRA checks. The "duplicated" issues that were found are:

- **MISRA08_12-1-3** - All constructors that are callabe with a single argument of fundamental type shall be declared explicit - 11 violations
- **Cppcheck CWE-398** - No Explicit Constructor - 11 violations

These 2 checks refer to a quality practice in which constructors with a single argument should be declared *explicit*. The MISRA check indeed provides much more informations than the Cppcheck check.
It is noticeable that both the tools find the same amount of violations for this check.

- **MISRA08_0-1-3** - A project shall not contain unused variables - 59 violations
- **Cppcheck CWE-563** - Unred variable - 1 violation

The MISRA check is more general, compared to the Cppcheck. This one refers only to variables that are never read, while the MISRA check refers to variables that are declared but never used in general. There is a huge difference on the amount of violations found: 59 for the MISRA, 1 for Cppcheck.

- **MISRA08_8-5-1** - All variables shall have a defined value before they are used - 23 violations
- **Cppcheck CWE-398** - Uninit Member Var - 10 violations

Cppcheck reported a large amount of "Unused Functions" (375 violations on 414 totally found) and, as for Understand where 733 violations of this kind were found with the MISRA set of rules, we suppose that this could be caused by the fact that not the whole project has been analyzed. It is odd that the $UnusedFunctions/TotalViolations$ proportion changes this much: 90% (Cppcheck) versus 9% (Understand).
Generally speaking Cppcheck reported less issues than Understand.

File	Tot
ARCMigrate.cpp	5
BuildSystem.cpp	11
CIndex.cpp	158
CIndexCodeCompletion.cpp	27
CIndexCXX.cpp	4
CIndexDiagnostic.cpp	15
CIndexDiagnostic.h	2
CIndexer.h	2
CIndexHigh.cpp	2
CIndexInclusionStack.cpp	1
CIndexUSRs.cpp	6
CXComment.cpp	34
CXCompilationDatabase.cpp	16
CXCursor.cpp	20
CXCursor.h	1
CXIndexDataConsumer.cpp	3
CXIndexDataConsumer.h	11
CXLoadedDiagnostic.cpp	2
CXLoadedDiagnostic.h	1
CXSourceLocation.cpp	7
CXString.cpp	2
CXString.h	1
CXTranslationUnit.h	1
CXType.cpp	47
Indexing.cpp	35
Total	414

Check	CWE	Tot
unusedFunction	561	375
noExplicitConstructor	398	11
invalidPrintfArgType_sint	686	3
cstyleCast	398	7
unreadVariable	563	1
duplicateExpression	398	1
ConfigurationNotChecked	N.A.	2
passedByValue	398	3
copyCtorAndEqOperator	N.A.	1
uninitMemberVar	398	10
Total	N.A.	414

Figure 13: Summary of the Cppcheck checks

2.5 FLAWFINDER ANALYSIS

Flawfinder has the same characteristics as Cppcheck. It is a light (runnable from console, without UI) and open-source.
The big difference with the previous tools is that Flawfinder is designed to find vulnerabilities more related to *computer security* rather than code quality issues and bugs.
It can be downloaded at `https://dwheeler.com/flawfinder/#downloading`.
Once installed, the tool can be run via terminal using the following command:

- flawfinder [OPTIONS] [files or directories]

Flawfinder has much more [OPTIONS] than Cppcheck, for example:

- Write results to a txt, csv or html file
- Ignore specific files
- Set the minimum risk level to be included in the *hit list*
- Display hits without waiting for the end of the analysis
- An option to don't include false positive. This option was used for one analysis but, as pointed in the documentation, this is risky since it does not follow any standard procedure to exclude likely false positives. For this reason, results found using this options were not considered to be trustable
- And more... (see flawfinder –help for more informations)

As with the previous tools, Flawfinder performed its analyses on the **tools/libclang** directory.

2.5.1 *Flawfinder Results*

The checks performed by Flawfinder are based on the MITRE-CWE list as well as Cppcheck.
The issues categories found during the analyses are listed below:

(CWE-20) **Improper Input Validation** - When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts

of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution [7]

(CWE-120) **Buffer Copy without Checking Size of Input** - A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold, or when a program attempts to put data in a memory area outside of the boundaries of a buffer. The simplest type of error, and the most common cause of buffer overflows, is the "classic" case in which the program copies the buffer without restricting how much is copied. Other variants exist, but the existence of a classic overflow strongly suggests that the programmer is not considering even the most basic of security protections [8]

(CWE-807) **Reliance on Untrusted Inputs in a Security Decision** - Developers may assume that inputs such as cookies, environment variables, and hidden form fields cannot be modified. However, an attacker could change these inputs using customized clients or other attacks. This change might not be detected. When security decisions such as authentication and authorization are made based on the values of these inputs, attackers can bypass the security of the software. Without sufficient encryption, integrity checking, or other mechanism, any input that originates from an outsider cannot be trusted [9]

Some of the issues found by Flawfinder are tagged with the tag: [MS-banned]. This tag refers to particular functions that are in the *banned functions list* published by Microsoft [10]. These are functions that contain critical and well-known security vulnerabilities such, e.g. *strcpy* that introduces a *buffer overflow* vulnerability.

Comparing the issues found with the previous reports, Flawfinder found a small number of vulnerabilities, this is also caused by the fact that this tool only looks for security issues.
The total vulnerability count is 32, distributed in the 3 CWE categories listed above.
The top three vulnerabilities are:

1. CWE-807, CWE-20 - Environment variables are untrustable input if they can be set by an attacker. They can have any content and length, and the same variable can be set more than once - 17 violations

2. CWE-120 - Does not check for buffer overflows when copying to destination - 6 violations

3. CWE-120 **[MS-banned]** - *strncpy* easily used incorrectly; doesn't always \0-terminate or check for invalid pointers - 5 violations

And the top three files are:

1. CIndex.cpp - 14 violations

2. CIndexCodeCompletion.cpp - 6 violations

3. CIndexer.cpp & Indexing.cpp - 3 violations

The number of violations found is very small so the only things that are noticeable are that:

- CIndex.cpp also in this analysis is the file that contains the biggest amount of violations

- Differing from the other analyses, Flawfinder found vulnerabilities only on 8 files out of 33

- The most common causes of vulnerabilities found in this project are related to the use of environmental variables (17 violations out of 32) and the possible arise of buffer overflows (14 violations out of 32, combining different CWE checks)

Flawfinder offers an option to run the analysis without including possible false positives.
As already said in the beginning of this section, results found with this options are not accurate. In particular, it has been reported that the possible false positive is:

(CWE-119/120) Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues

This clearly isn't a false positive since this is a well-known vulnerability.

Looking at performances, in terms of speed of analysis and easiness of the tool, it compares favorably with the considerations made for Cppcheck.

Violation	Tot
It's often easy to fool getlogin. Sometimes it does not work at all, because some program messed up the utmp file. Often, it gives only the first 8 characters of the login name. The user currently logged in on the controlling tty of our program need not be the user who started it. Avoid getlogin() for security-related purposes (CWE-807)	1
Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120)	1
Environment variables are untrustable input if they can be set by an attacker. They can have any content and length, and the same variable can be set more than once (CWE-807, CWE-20)	17
Does not check for buffer overflows when copying to destination (CWE-120)	6
Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119, CWE-120)	2
Strncpy easily used incorrectly; doesn't always \0-terminate or check for invalid pointers [MS-banned] (CWE-120)	5
Total	32

File	Tot
CIndexCodeCompletion.cpp	6
CIndexer.cpp	3
ARCMigrate.cpp	2
CIndex.cpp	14
CLog.h	1
Indexing.cpp	3
BuildSystem.cpp	2
CXLoadedDiagnostic.cpp	1
Total	32

Figure 14: Summary of the Flawfinder checks

2.6 SONARQUBE & CERT ROSECHECKERS

SonarQube is a well known tool for static analysis, capable of detecting both code quality issues (referred as *Code Smells* and security issuds.
CERT Rosecheckers is a tool that follows the CERT C coding standard to perform it analyses, including CWEs entries and MISRA rules.

SonarQube was not used because it requires the build of the whole project in order to perform an analysis. Since Clang is a very huge project this could not be done.
CERT Rosecheckers does not require the project to be built but its inability to use *make files* to link source files caused various issues that will be described later.

2.6.1 *SonarQube*

SonarQube (available also via cloud at `https://sonarcloud.io`) is a general languages analysis tool. It uses different standards such as MISRA, CERT... and it also offers the possibility to define custom rules.
On SonarCloud after an organization's name is defined and a token linked to the project is created, the steps to follow to perform an analysis (on a Linux system) are the following:

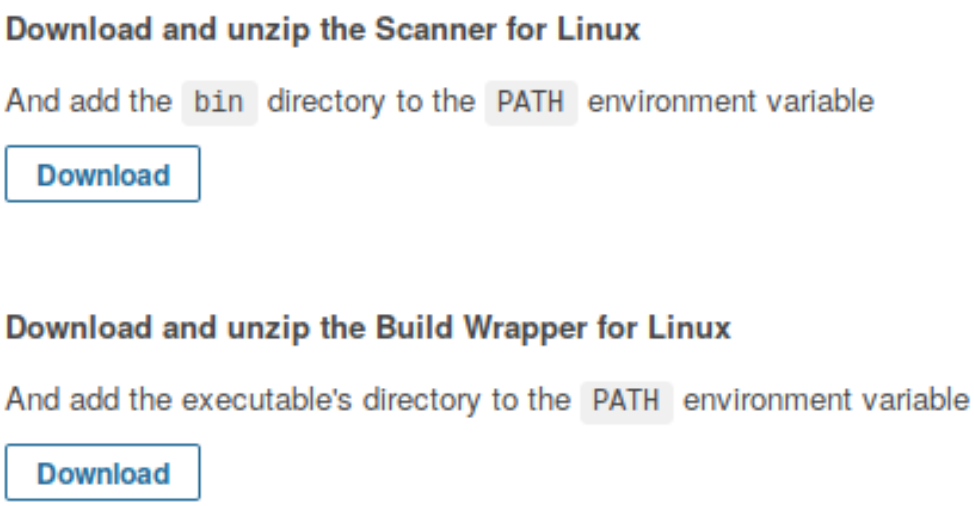

Figure 15: The build-scanner and the build-linux-wrapper have to be downloaded to perform the analysis locally

Figure 16: The first command is used to compile the project.
The second set of commands is used by SonarQube scanner to scan the project.
As already mentioned, given the huge size of LLVM-Clang, this was unfeasible

2.6.2 *CERT Rosechecker*

CERT Rosecheckers is a free tool that can be downloaded from `https://sourceforge.net/projects/rosecheckers/` and it is available pre-installed in a Virtual Machine.
Once the virtual machine is executed it is possible to perform the analysis from the shell of the VM. It works by specifying a file, a set of files or a directory:

- rosecheckers -rose:[OPTIONS] [FILES/DIRECTORY]

Figure 17: Screenshot of the Rosechecker Virtual Machine

Listed below are some of the main options offered by Rosechecker:

- *C_Only*, *Fortran*, *Cxx_Only* and more to select specific languages source files
- *strict* for a strict enforncement of ANSI/ISO standards
- *excludeCommentsAndDirectives* to exclude comments and preprocessor directives
- And more... (see rosechekers –help for more informations)

The problem with this tool was that it requires all the *#include* files to be in the same directory.

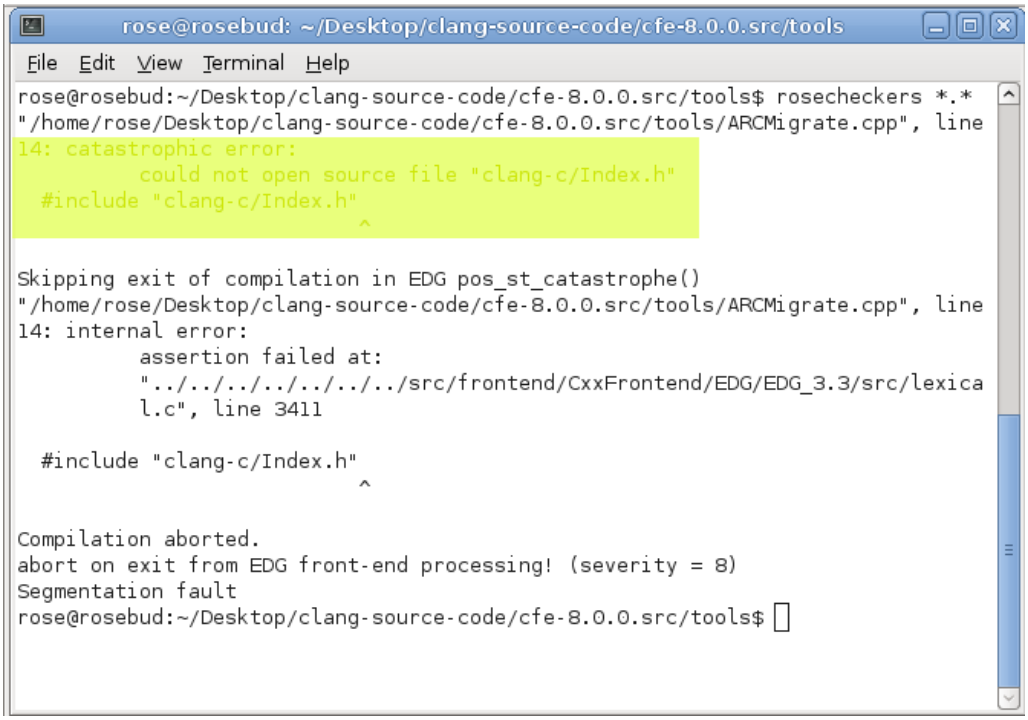

Figure 18: Screenshot of the missing include fatal error

There are two different solutions for this problem:

- Manually remove all the *#include* directives from the source files
- Manually move all the *#include* files in the directory to be analyzed

Obviously, the first solution is conceptually incorrect: the analysis is going to be performed on files that have been modified, compromising the final results.
The second approach is more reasonable (even if the project structure is modified we are interested only on a subset of the whole project and, in this way, we are not changing the interested source code files) but *Rosecheckers* looks in a recursive way for the *#include* files in the files to be included, and so on...

[e.g.]

- CIndex.cpp has the include directive "#include "clang/AST/Attr.h""
- To solve the missing include issue is necessary to copy the *clang/AST)* directory in the analysis-root folder
- Rosechecker will then check the "include files" of the included file "Attr.h"
- Attr.h has the include directive "#include "clang/AST/Type.h""
- To solve the missing include is necessary to to copy the *clang/AST)* directory in this path: *analysis-root/clang/AST*
- At this point we have this situation: *analysis-root/clang/AST/clang/AST*
- This procedure must be followed to solve all the *missing include* issues

This recursive procedure is obviously unfeasible since the number of folders to be copied/created grows exponentially with respect to the number of files to be included in a single source (or header) file.

3

CONCLUSIONS

In this work 5 different static analysis tools were used.
The first difference between these tools is the set of rules used. Different standards have been covered:

- MISRA-C++ and Code Quality practices - **Understand**
- MITRE-CWE - **Cppcheck**, **Flawfinder**
- CERT-C - **Rosechecker**
- Various standards and Code Quality practices - **SonarQube**

These tools are not meant to be mutually exclusive in their use. For example Cppcheck can be used in combination with Flawfinder to detect both *Code Quality* and *Security* issues, using the same standard (MITRE-CWE).
Specific considerations must be made for the tool *Clang Static Analyzer* which was used as an Understand addon.
The analysis with this tool provided no results. Seeing this we can suppose that the LLVM Developer Group defined a custom set of Quality rules without referring to any specific standard and that this set of rules was followed during the implementation of LLVM-Clang compiler and it is the one implemented in the Clang Static Analyzer tool.

During the analyses it emerged that the file *CIndex.cpp*, which is the biggest source file in the analyzed directory *tools/libclang*, has always been the file with the most detected issues (always $\approx$ 50% of the whole issues set). Since this anomaly was observed with all the tools we can consider it to be relevant and that this file needs some refactoring.
Another interesting thing noticed is that if we exclude limit cases such as

CIndex.cpp or very small files (4-5 lines) we can observe a direct proportionality of the issues count with respect to the size of files (i.e. we have a quasi-uniform distribution of the issues in the files wrt to the their sizes).

Looking at the various checks, it is observable that some of them are overlapping (even if different tools found different amounts of them). The most common issue that arises in all the analyses (excluding Flawfinder due to its nature) is the *unusedFunction* issue. However some of the violations reported are probably false positives because of the fact that not the whole project was analyzed. We can suppose that if we analyze the whole project, the violation count of this issue decreases (maybe going to conform to the issues distribution of the other violations).
In general, the analyses performed with the MISRA-C++ standard reported a much greater violations count while all the others standards/checks were much fewer (excluding the Understand *AllCheck* checks that is a collection of a lot of standards and quality practices).

Of all the tools used, Understand is the most useful for static analysis of a project. First of all because it has a user-friendly UI, because it offers a lot of different perspectives of the analysis, and because it includes a lot of different features/standards/checks.
A good idea could be to use in combination both Understand and Flawfinder in order to have a complete view of the project code quality and security robustness.
Instead of using two different tools in combination, it could be used SonarQube solely, since it is able to check both of the aspects described above but, since it requires the build of the project, it wasn't possible to use it in this work.

Looking at the LLVM-Clang compiler a large number of Quality issues was found with all the tools. This is indeed an evidence of the fact that the Code Quality of this project could (must) be improved.
The Security issues count, compared to the Quality issues count, is very little (32 Security problems). However every single Security issue introduces some vulnerability in the software that could be exploited by an attacker or cause a crash of the program (i.e. the *buffer overflow* vulnerabilities found).

Concluding, the big amount of violations found in general evidences the fact the this project should be revised. More accurate results can be achieved by analyzing the complete project.

BIBLIOGRAPHY

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

[2] Cppcheck - *http://cppcheck.sourceforge.net/* (Cited on page 9.)

[3] CWE - *https://cwe.mitre.org/data/definitions/398.html* (Cited on page 28.)

[4] CWE - *https://cwe.mitre.org/data/definitions/561.html* (Cited on page 28.)

[5] CWE - *https://cwe.mitre.org/data/definitions/563.html* (Cited on page 28.)

[6] CWE - *https://cwe.mitre.org/data/definitions/686.html* (Cited on page 28.)

[7] CWE - *https://cwe.mitre.org/data/definitions/20.html* (Cited on page 33.)

[8] CWE - *https://cwe.mitre.org/data/definitions/120.html* (Cited on page 33.)

[9] CWE - *https://cwe.mitre.org/data/definitions/807.html* (Cited on page 33.)

[10] Microsoft - *http://msdn.microsoft.com/en-us/library/bb288454.aspx* (Cited on page 33.)