Test Coverage and Static Code Analysis in CI/CD of Root

Aniq Javed

Supervisors: Axel Naumann & Bertrand Bellenot

September 21, 2023

Abstract

The predominant challenge observed in extensive, long-lived code bases [1] resides in the increasing difficulty of detecting software defects and security vulnerabilities over time, due to their intricate architectural structures and inter-dependencies across multiple programming languages. Consequently, such complexity renders the software susceptible to malicious activities. To mitigate these risks, preemptive identification of these defects can be accomplished through various methodologies, including code reviews, utilization of static code analysis tools, automated testing frameworks, vulnerability scanning protocols, and penetration testing procedures. One notable software exemplifying the characteristics described herein is ROOT which has been in operation for 25+ years. It constitutes binding between more than 3 programming languages namely C, C++, and Python. Used across the world by several thousand people in the field of high-energy physics. With a substantial code base of over 500k lines written over the course of 2 decades, ROOT provides a perfect use case for this analysis. By analyzing the test coverage of ROOT, we were able to get critical information regarding the presence of code lines that have never been executed, essentially representing untested and potentially vulnerable areas within the code base. Our coverage analysis revealed that approximately 70% of ROOT's code-base had never been touched by test cases, leaving a substantial portion of the software unchecked and exposed to lurking issues. Delving deeper into our analysis, we employed static code analysis tools to scrutinize the intricate web of dependencies and potential code smells hidden within ROOT. Astonishingly, these tools claimed to find over 300 instances of violations of best practices and more than 500 security vulnerabilities. These potential vulnerabilities ranged from memory leaks to data corruption issues, each posing a significant threat to the stability and reliability of ROOT if deemed correct.

1 Test Coverage

Code coverage corresponds directly to the test suite quality, [2]. ROOT is made up of a set of modules which include various histogramming methods in an arbitrary number of dimensions,

curve fitting, function evaluation, minimization, graphics, and visualization classes [3]. It runs 2433 test cases to get the results about the efficiency of the various components of ROOT like RDataFrame, TMVA, RNTuple etc. We use the following command to run the tests for the root repository

```
ctest -jXX
```

with XX representing the number of cores. You can run all the test cases present in the root test suite. During this process .gcno and .gcda files are created in every folder of the root build folder. Now for the generation of code coverage, we need a profiling tool that can work with GCC to analyze a program. For this we used two tools:

1.1 LCOV

lcov is a GNU tool that provides information about what parts of a program are covered while running a particular test case. The extension consists of a set of Perl scripts that are built on the textual GCOV output to implement enhanced functionalities like overview pages allowing quick browsing of coverage data by providing a hierarchical directory structure view. It includes differential coverage and date binning [4], which are used for combining coverage data and project/file history to determine if goals have been met and to identify areas of unexercised code that should be reviewed. It requires a baseline coverage data file to be created which will be combined with other data files.

```
lcov --no-external --capture --initial --directory ../../builddir \
--output-file coverage.info
```

Afterwards in order to generate report from this file we use a tool genhtml

```
genhtml coverage.info --output-directory coverage_report
```

The time taken to produce the lcov report is 17.38 minutes which shows that the actual time taken by the tests cases to run is 179.36 minutes which is far more than that. So, to optimize the operation, we can either run only those test cases that are most important, or we can improve the time efficiency of already available test cases. The results generated by lcov HTML report shows that the total line coverage of core/clingutils is 54.8% as shown in Figure 1.

1.2 GCOV

GCOV is a profiling tool that works in combination with GCC to analyze and discover untested parts of your program. While building the root with -ftest-coverage gcc adds instrumentation code to binary which is used to create .gcda profile output file. The command that was used to create gcov report was as follows:

```
gcovr --output=gcovr_coverage_time.html --html --gcov-ignore
errors=no_working_dir_found --merge-mode-functions=merge-use-
line-min --exclude-unreachable-branches --exclude-
```

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - core/clingutils			
Test:	coverage.info			
Date:	2023-09-11 14:30:07			
	Lines:	2363	4313	54.8 %
	Functions:	357	1163	30.7 %

Filename	Line Coverage		Functions	
G__mapDict.cxx	51.4 %	164 / 319	25.6 %	22 / 86
G__complexDict.cxx	52.5 %	116 / 221	33.9 %	21 / 62
G__valarrayDict.cxx	52.8 %	57 / 108	30.0 %	9 / 30
G__unordered_multimapDict.cxx	54.0 %	271 / 502	28.9 %	39 / 135
G__unordered_mapDict.cxx	54.0 %	341 / 632	28.8 %	49 / 170
G__multimapDict.cxx	54.1 %	131 / 242	29.2 %	19 / 65
G__multimap2Dict.cxx	54.1 %	145 / 268	29.2 %	21 / 72
G__unordered_multisetDict.cxx	54.2 %	103 / 190	29.4 %	15 / 51
G__multisetDict.cxx	54.2 %	103 / 190	29.4 %	15 / 51
G__map2Dict.cxx	55.4 %	191 / 345	31.2 %	29 / 93
G__vectorDict.cxx	56.5 %	210 / 372	33.0 %	33 / 100
G__forward_listDict.cxx	56.8 %	108 / 190	33.3 %	17 / 51
G__setDict.cxx	56.8 %	108 / 190	33.3 %	17 / 51
G__unordered_setDict.cxx	56.8 %	108 / 190	33.3 %	17 / 51
G__dequeDict.cxx	57.3 %	94 / 164	34.1 %	15 / 44
G__listDict.cxx	59.5 %	113 / 190	37.3 %	19 / 51

Generated by: LCOV version 1.14

Figure 1: Results generated using LCOV.

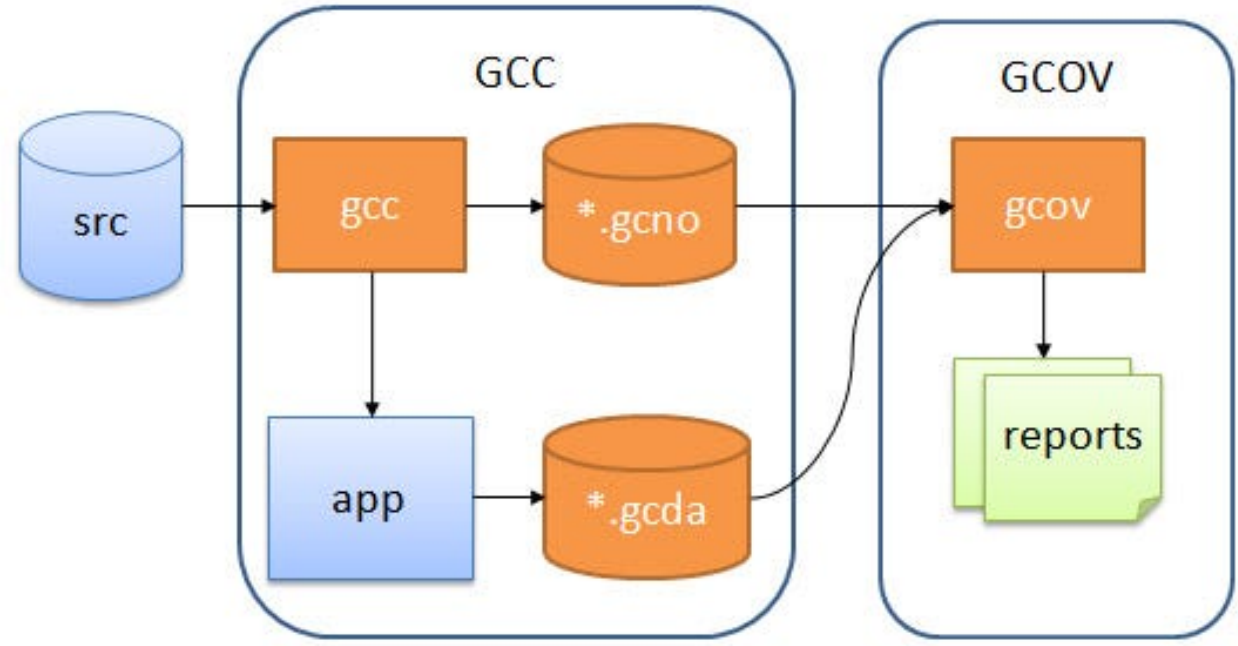


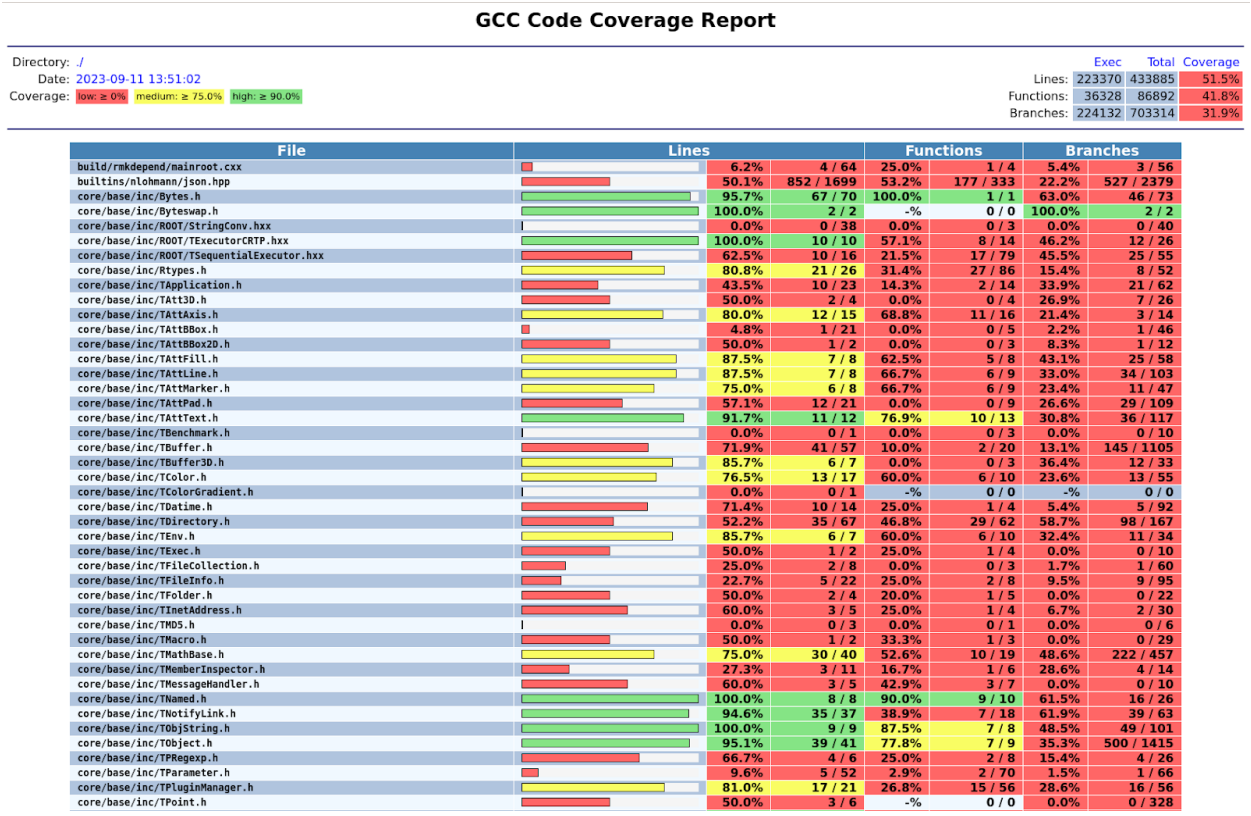
Figure 2: Architecture for gcov report generation

```
directories="roottest|runtutorials" --exclude='.*/G__.*'
--exclude='.*/(roottest|runtutorials|externals|ginclude|googl
etest-prefix|macosx|winnt|geombuilder|cocoa|quartz|win32gdk|x11|x11
ttf|eve|fitpanel|ged|gui|guibuilder|guihtml|qtgsi|qtroot|reco
rder|sessionviewer|tmvagui|treeviewer|geocad|fitsio|gviz|qt|g
viz3d|x3d|spectrum|spectrumpainter|dcache|hdfs|foam|genetic|m
lp|quadp|splot|memstat|rpdutils|proof|odbc|llvm|test|interpre
ter)/.*' --gcov-exclude='.*_ACLiC_dict[.].*' '--
exclude=.*_ACLiC_dict[.].*' --root=../root_src --object-
```

```
directory=../builddir
```

One additional feature about gcov was that it can create XML reports which were instrumental integration in the CI/CD pipeline of ROOT. The actual time taken by the gcov to gather data from 3606 files is 7.31 minutes.

The results generated by gcov HTML report is shown in figure 3.



GCC Code Coverage Report

Directory: ./
Date: 2023-09-11 13:51:02
Coverage: low: ≥ 0% medium: ≥ 75.0% high: ≥ 90.0%

	Exec	Total	Coverage
Lines:	223370	433885	51.5%
Functions:	36328	86892	41.8%
Branches:	224132	703314	31.9%

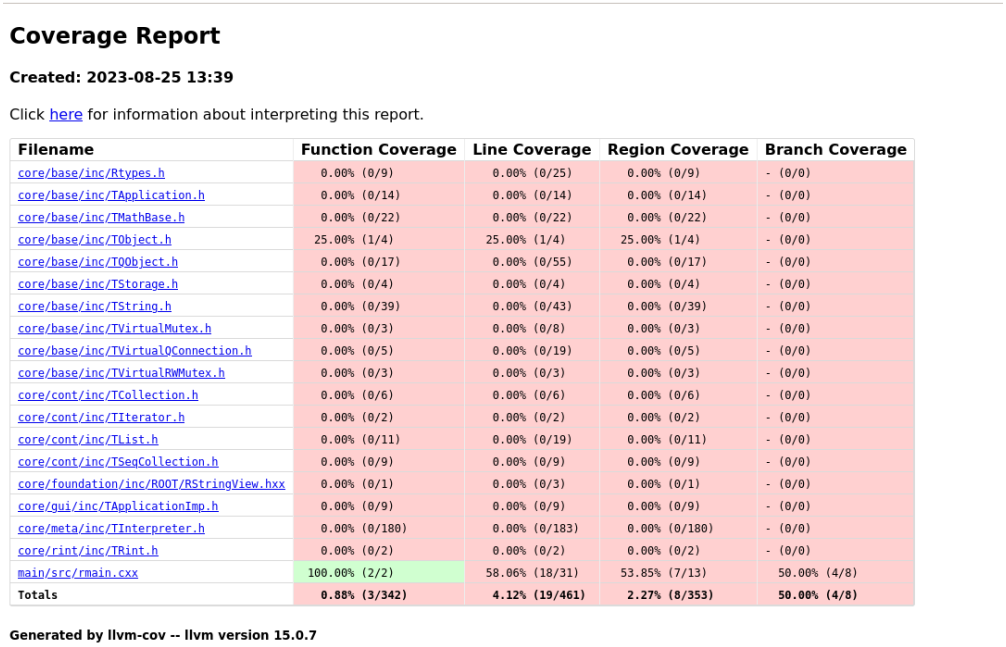
File	Lines			Functions		Branches	
build/rmkdepend/mainroot.cxx		6.2%	4 / 64	25.0%	1 / 4	5.4%	3 / 56
builtins/nlohmann/json.hpp		50.1%	852 / 1699	53.2%	177 / 333	22.2%	527 / 2379
core/base/inc/Bytes.h		95.7%	67 / 70	100.0%	1 / 1	63.0%	46 / 73
core/base/inc/Byteswap.h		100.0%	2 / 2	-%	0 / 0	100.0%	2 / 2
core/base/inc/ROOT/StringConv.hxx		0.0%	0 / 38	0.0%	0 / 3	0.0%	0 / 40
core/base/inc/ROOT/TExecutorCRTP.hxx		100.0%	10 / 10	57.1%	8 / 14	46.2%	12 / 26
core/base/inc/ROOT/TSequentialExecutor.hxx		62.5%	10 / 16	21.5%	17 / 79	45.5%	25 / 55
core/base/inc/Rtypes.h		80.8%	21 / 26	31.4%	27 / 86	15.4%	8 / 52
core/base/inc/TApplication.h		43.5%	10 / 23	14.3%	2 / 14	33.9%	21 / 62
core/base/inc/TAtt3D.h		50.0%	2 / 4	0.0%	0 / 4	26.9%	7 / 26
core/base/inc/TAttAxis.h		80.0%	12 / 15	68.8%	11 / 16	21.4%	3 / 14
core/base/inc/TAttBBox.h		4.8%	1 / 21	0.0%	0 / 5	2.2%	1 / 46
core/base/inc/TAttBBox2D.h		50.0%	1 / 2	0.0%	0 / 3	8.3%	1 / 12
core/base/inc/TAttFill.h		87.5%	7 / 8	62.5%	5 / 8	43.1%	25 / 58
core/base/inc/TAttLine.h		87.5%	7 / 8	66.7%	6 / 9	33.0%	34 / 103
core/base/inc/TAttMarker.h		75.0%	6 / 8	66.7%	6 / 9	23.4%	11 / 47
core/base/inc/TAttPad.h		57.1%	12 / 21	0.0%	0 / 9	26.6%	29 / 109
core/base/inc/TAttText.h		91.7%	11 / 12	76.9%	10 / 13	30.8%	36 / 117
core/base/inc/TBenchmark.h		0.0%	0 / 1	0.0%	0 / 3	0.0%	0 / 10
core/base/inc/TBuffer.h		71.9%	41 / 57	10.0%	2 / 20	13.1%	145 / 1105
core/base/inc/TBuffer3D.h		85.7%	6 / 7	0.0%	0 / 3	36.4%	12 / 33
core/base/inc/TColor.h		76.5%	13 / 17	60.0%	6 / 10	23.6%	13 / 55
core/base/inc/TColorGradient.h		0.0%	0 / 1	-%	0 / 0	-%	0 / 0
core/base/inc/TDatime.h		71.4%	10 / 14	25.0%	1 / 4	5.4%	5 / 92
core/base/inc/TDirectory.h		52.2%	35 / 67	46.8%	29 / 62	58.7%	98 / 167
core/base/inc/TEnv.h		85.7%	6 / 7	60.0%	6 / 10	32.4%	11 / 34
core/base/inc/TExec.h		50.0%	1 / 2	25.0%	1 / 4	0.0%	0 / 10
core/base/inc/TFileCollection.h		25.0%	2 / 8	0.0%	0 / 3	1.7%	1 / 60
core/base/inc/TFileInfo.h		22.7%	5 / 22	25.0%	2 / 8	9.5%	9 / 95
core/base/inc/TFolder.h		50.0%	2 / 4	20.0%	1 / 5	0.0%	0 / 22
core/base/inc/TInetAddress.h		60.0%	3 / 5	25.0%	1 / 4	6.7%	2 / 30
core/base/inc/TMD5.h		0.0%	0 / 3	0.0%	0 / 1	0.0%	0 / 6
core/base/inc/TMacro.h		50.0%	1 / 2	33.3%	1 / 3	0.0%	0 / 29
core/base/inc/TMathBase.h		75.0%	30 / 40	52.6%	10 / 19	48.6%	222 / 457
core/base/inc/TMemberInspector.h		27.3%	3 / 11	16.7%	1 / 6	28.6%	4 / 14
core/base/inc/TMessageHandler.h		60.0%	3 / 5	42.9%	3 / 7	0.0%	0 / 10
core/base/inc/TNamed.h		100.0%	8 / 8	90.0%	9 / 10	61.5%	16 / 26
core/base/inc/TNotifyLink.h		94.6%	35 / 37	38.9%	7 / 18	61.9%	39 / 63
core/base/inc/TObjString.h		100.0%	9 / 9	87.5%	7 / 8	48.5%	49 / 101
core/base/inc/TObject.h		95.1%	39 / 41	77.8%	7 / 9	35.3%	500 / 1415
core/base/inc/TPRegexp.h		66.7%	4 / 6	25.0%	2 / 8	15.4%	4 / 26
core/base/inc/TParameter.h		9.6%	5 / 52	2.9%	2 / 70	1.5%	1 / 66
core/base/inc/TPluginManager.h		81.0%	17 / 21	26.8%	15 / 56	28.6%	16 / 56
core/base/inc/TPoint.h		50.0%	3 / 6	-%	0 / 0	0.0%	0 / 328

Figure 3: GCOV html report

Optimization

To optimize the time taken by the test cases, we used llvm-cov which works differently as compared to GCC tools. It involves clang build which is fast as compared to works both on Linux and MacOS. We can see significant improvement in time but we are unable to get use of llvm because it requires an executable to calculate the coverage and in our case, we do not have executables for every other folder. In the coverage report below we can see that the coverage is calculated for TObject.h and main.cxx because they were the only files that were affected by the root executable. The command which generated the report shown in figure 4 is as follows:

```
llvm-cov show -format=html -output-dir=coverage_report \
../builddir-llvm/bin/root.exe \
-instr- profile=../merged.profdata
```



Coverage Report

Created: 2023-08-25 13:39

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
core/base/inc/Rtypes.h	0.00% (0/9)	0.00% (0/25)	0.00% (0/9)	- (0/0)
core/base/inc/TApplication.h	0.00% (0/14)	0.00% (0/14)	0.00% (0/14)	- (0/0)
core/base/inc/TMathBase.h	0.00% (0/22)	0.00% (0/22)	0.00% (0/22)	- (0/0)
core/base/inc/TObject.h	25.00% (1/4)	25.00% (1/4)	25.00% (1/4)	- (0/0)
core/base/inc/TQObject.h	0.00% (0/17)	0.00% (0/55)	0.00% (0/17)	- (0/0)
core/base/inc/TStorage.h	0.00% (0/4)	0.00% (0/4)	0.00% (0/4)	- (0/0)
core/base/inc/TString.h	0.00% (0/39)	0.00% (0/43)	0.00% (0/39)	- (0/0)
core/base/inc/TVirtualMutex.h	0.00% (0/3)	0.00% (0/8)	0.00% (0/3)	- (0/0)
core/base/inc/TVirtualQConnection.h	0.00% (0/5)	0.00% (0/19)	0.00% (0/5)	- (0/0)
core/base/inc/TVirtualRWMutex.h	0.00% (0/3)	0.00% (0/3)	0.00% (0/3)	- (0/0)
core/cont/inc/TCollection.h	0.00% (0/6)	0.00% (0/6)	0.00% (0/6)	- (0/0)
core/cont/inc/TIterator.h	0.00% (0/2)	0.00% (0/2)	0.00% (0/2)	- (0/0)
core/cont/inc/TList.h	0.00% (0/11)	0.00% (0/19)	0.00% (0/11)	- (0/0)
core/cont/inc/TSeqCollection.h	0.00% (0/9)	0.00% (0/9)	0.00% (0/9)	- (0/0)
core/foundation/inc/ROOT/RStringView.hxx	0.00% (0/1)	0.00% (0/3)	0.00% (0/1)	- (0/0)
core/gui/inc/TApplicationImp.h	0.00% (0/9)	0.00% (0/9)	0.00% (0/9)	- (0/0)
core/meta/inc/TInterpreter.h	0.00% (0/180)	0.00% (0/183)	0.00% (0/180)	- (0/0)
core/rint/inc/TRint.h	0.00% (0/2)	0.00% (0/2)	0.00% (0/2)	- (0/0)
main/src/rmain.cxx	100.00% (2/2)	58.06% (18/31)	53.85% (7/13)	50.00% (4/8)
Totals	**0.88% (3/342)**	**4.12% (19/461)**	**2.27% (8/353)**	**50.00% (4/8)**

Generated by llvm-cov -- llvm version 15.0.7

Figure 4: llvm-cov html report

2 Static Code Analysis

Static code analysis infers to finding the vulnerabilities in the codebase without executing it, [5]. It involves analysis of different frameworks, programming languages, and modules present in the source code. It has been used since the early 1960s to optimize the compilers' operation, [6]. Afterwards, it was used for debugging tools, as well as for software development frameworks. Different tools are available that can be used to analyze the codebase based on requirements like programming language, project size etc. [7]. In our analysis we have used codeql to perform the static code analysis. The choice of codeql was made because it is an open-source tool which can be easily implemented locally and can be integrated with Github's CI/CD pipeline as well. The architecture of the CodeQL analysis pipeline is shown in figure 5.

Codeql requires a database to be created on which multiple queries can be run.

```
codeql database create ../root-db  --language=cpp --command='cmake
--build ../root/builddir  --target install' -threads=16
```

The query required to analyze the databases:

```
codeql database analyze ../root-db codeql/cpp-queries:codeql-suites
/cpp-code-scanning.qls --format=csv --output=cpp-results.csv
--download --ram=20000
```

In our case, we were able to identify 342 code vulnerabilities out of which 3 were critical and 338 were high in severity. The most common defect was "Multiplication result converted to larger type" which can lead to arithmetic overflow resulting in the unexpected behavior as

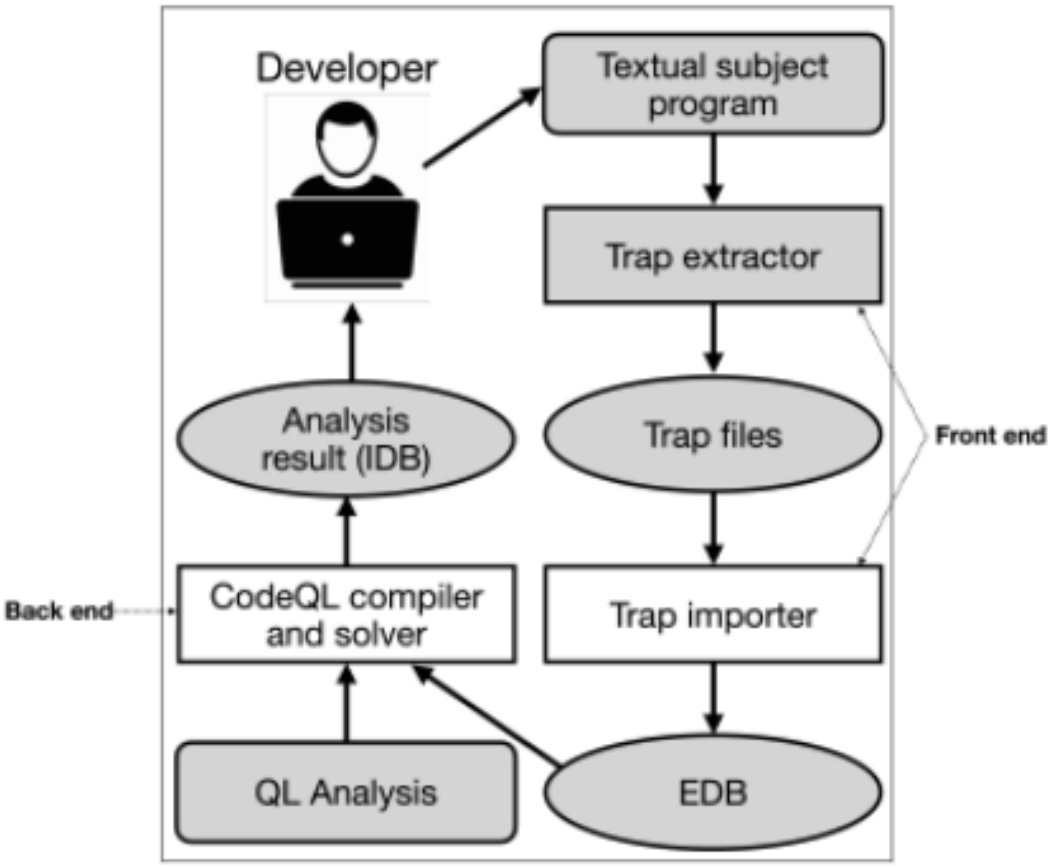


Figure 5: The architecture of Codeql query analysis

shown in Figure 6. Through this we were able to identify Common Weakness Enumerations where we can see which group our defect belongs to and what threats it pose to the system.

Refactoring

Through these tools we can perform refactoring [8] as the data provided by the CodeQl helps us to identify the defects and also gives us potential ways to solve them. As shown in figure 7, for the vulnerability "No space for zero terminator" we can see that the tool is identifying the cause of the error and is also providing the recommendation that can be implemented to correct and remove the defect. Also it provides references to the literature where this common weakness was defined along with the commit that was responsible for introducing this as shown in Figure 8 and 9.

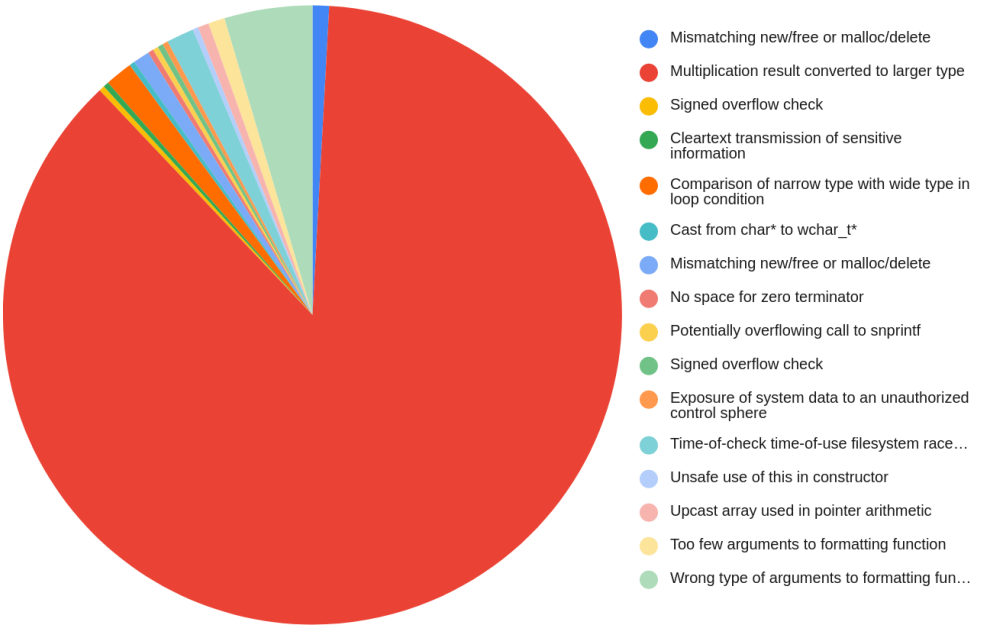


Figure 6: Defects found out by the codeql and their percentage in the codebase.

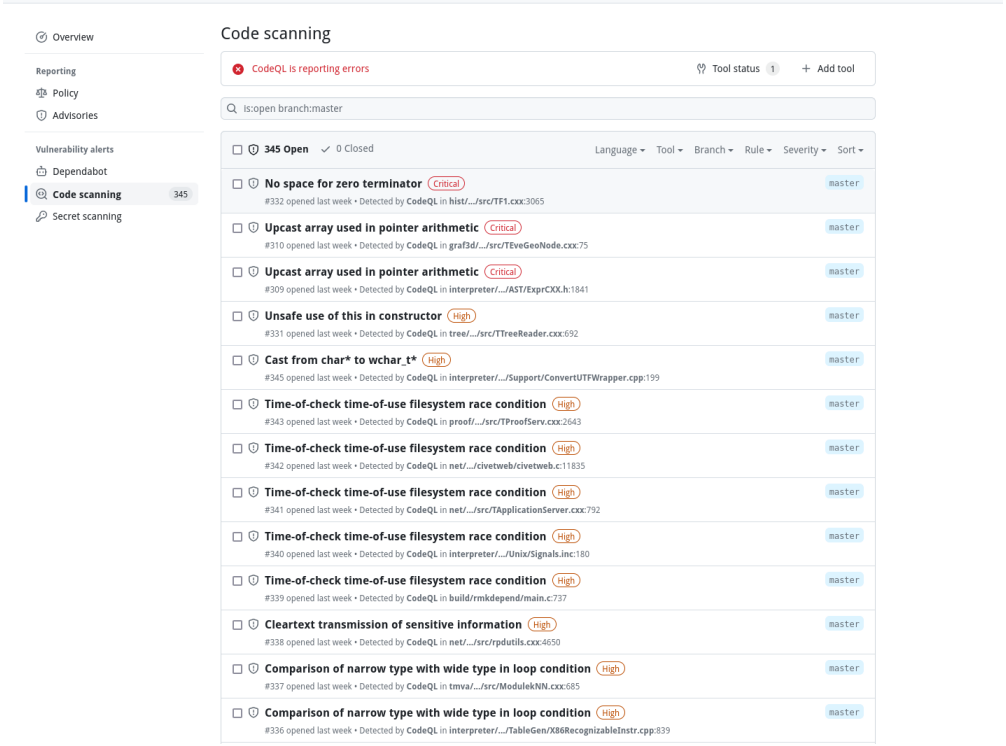


Figure 7: Code scanning results in Github GUI

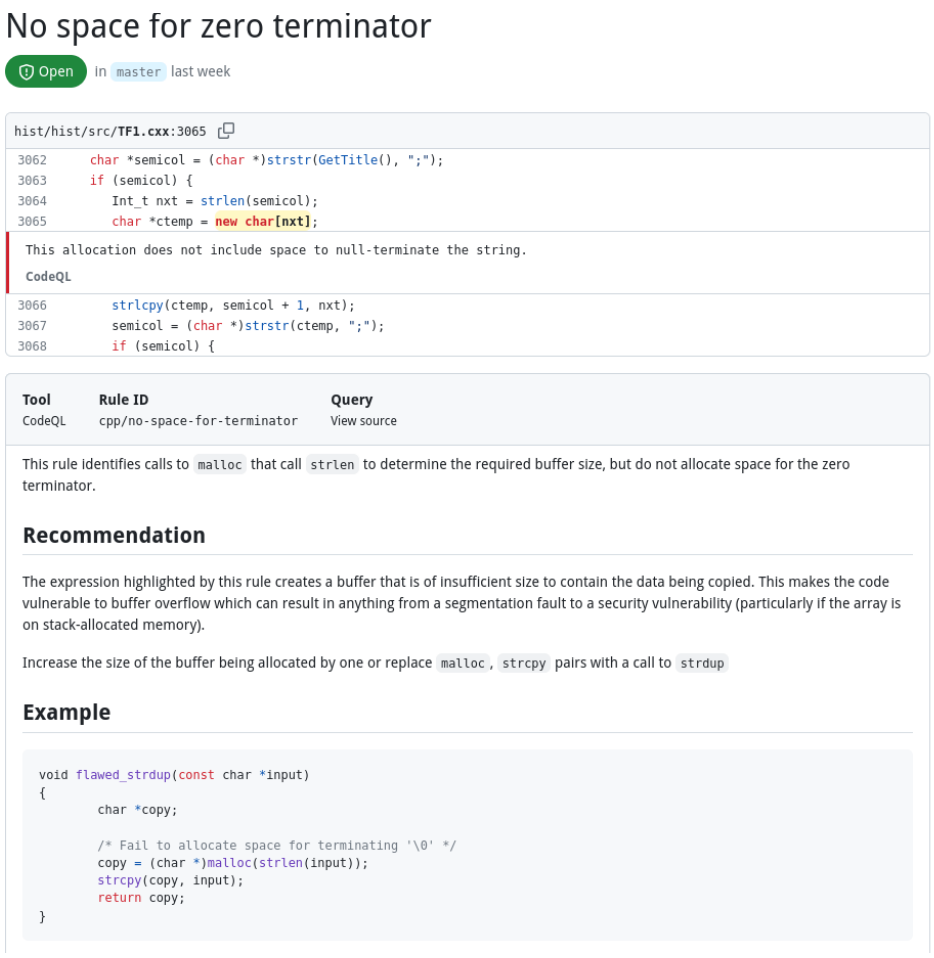


Figure 8: Recommendation provided by CodeQl

Figure 9: Code weakness identification by CodeQl

3 Conclusions

Currently, the size of the database created by the codeql is the main limiting factor which makes it impractical to perform static analysis on every PR created for the ROOT repository. Potential solutions would be to introduce incremental changes in the codeql database build [9]. This will allow us to ease the developers in the process of reviewing their PR's as well as saving the software from future security failures. Similarly in the case of test coverage we should be finding a better technique to reduce the time for build and test in order to the waiting time for the PR review resulting in the optimization of code quality in general. Overall, using of both of these techniques will have significant effect on the the code quality.

References

[1] Niclas Svensson, *Defining and Identifying Legacy Code in Software*, (2018).

[2] P. S. Kochhar, F. Thung and D. Lo, *Code coverage and test suite effectiveness: Empirical study with real bugs in large systems*, 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, Canada, 2015, pp. 560-564, doi: 10.1109/SANER.2015.7081877.

[3] Rene Brun and Fons Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.

[4] Henry Cox, Mediatek and Woburn, *Differential coverage: automating coverage analysis.*

[5] D. Stefanović and D. Nikolić and S. Havzi and T. Lolić and D. Dakić, *Identification of strategies over tools for static code analysis*, IOP Conference Series: Materials Science and Engineering (2021).

[6] Anders Møller and Michael I. Schwartzbach, *Static Program Analysis* 2023.

[7] M. Beller, R. Bholanath, S. McIntosh and A. Zaidman, *Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software* 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 2016, pp. 470-481, doi: 10.1109/SANER.2016.105.

[8] S. A. M. Rizvi and Z. Khanam, *A methodology for refactoring legacy code*, 2011 3rd International Conference on Electronics Computer Technology, Kanyakumari, India, 2011, pp. 198-200, doi: 10.1109/ICECTECH.2011.5942080.

[9] Tamás Szabó, *Incrementalizing Production CodeQL Analyses*, 2023.