EECS 4314 Assignment 3

Madison Hartley
215062417

Alberto Mastrofrancesco
214107742

Arsh Punia
215199961

Ashish Ranjan
215580327

Richard Robinson
215353287

Daniel Santaguida
214790695

Doris Zhou
214746283

Pamir Amiry
215077373

November 2021

Abstract

PostgreSQL is a popular, open-source relational database management system. To further understand the system's architecture, this paper tries to use different dependency extraction techniques to get the most accurate set of dependencies of the system and discusses the differences between the techniques. The three techniques that are discussed are the GCC, Understand, and Include methods. The paper compares a subset of the discrepancies dependencies and performs a qualitative/quantitative analysis. The paper also details the different risks and limitations to our techniques and comparison methods.

Contents

1 Introduction

PostgreSQL is a free, open-source relational database management system that has been backing many software projects for over 20 years. This system has been quite refined and in order to accurately extract the different dependencies accurately we cannot rely on one dependency extraction technique. This paper examines the different dependency extraction techniques that can be used on postgres and analyzes the qualitative and quantitative results.

The three different dependency extraction methods that this paper examines are the GCC, Understand and Include methods. The GCC method is made from the same compiler that postgres uses. The GCC method is able to find dependency at an object file level and find all user level dependencies. The include method uses a script to go through all the source code files and find the "include" line for each dependency. The understand method uses the dependencies extracted from the understand tool.

2 New Dependency Techniques

2.1 Technique 1

This technique was used to extract the dependencies by traversing each source code file and parsing each `#include` directive to find the associated dependency. This was automated using a small Kotlin script.

Specifically, the script recursively walks top down through each of the source code files throughout PostgreSQL. Then, it filters the files to only include `.c` or `.h` files, since the source code contains other miscellaneous types of files within it. For each file, it then writes to the output `$INSTANCE [name] cFile` where `[name]` is the file path. Then, the file is parsed for all `#include` directives using the regex string `#include "(.*)"`, and the dependencies are extracted from there.

Once the dependency is known, the script then goes through the process of constructing its path relative to the base directory, which is relatively complex. The vast majority of the dependencies are `.h` files that reside within the `/src/include` directory. However, this is not always the case:

1. For files within the `ecpg` subdirectory, the `#include` directives may reference dependencies that are either within `/src/include` like normal, or those that exist within the `include` subdirectory of `ecpg` itself, which adds complexity if multiple files share the same base name in both directories.

2. Some subdirectories contain both `.c` and `.h` files. In this case, the `#include` directives may reference dependencies that are either within such subdirectory or within `/src/include` like normal.

If the file is in the `ecpg` directory, the script first looks to see if the dependency is within the `ecpg/include` directory. Then, if either of these are false, the script then looks in the normal `/src/include` directory to see if the dependency exists there. This covers the majority of situations. If the dependency still has yet to be found, then it is assumed that the dependency is located in the same directory as the file.

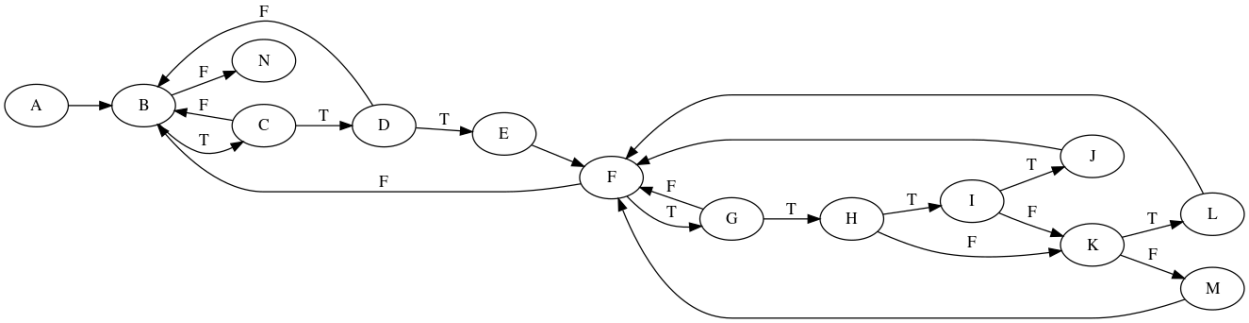

Figure 1: The program graph for the script used in Technique 1.

Node	Statement
A	Start
B	There are files that have yet to be traversed
C	`file.isFile`
D	`it.extension == "c" || it.extension == "h"`
E	`appendLine("$INSTANCE $it.toRelativeString(relativeTo) cFile")`
F	There are lines left in the file
G	`regex.find(line) != null`
H	`"ecpg" in file.path`
I	The dependency is within the `ecpg/include` directory
J	Write the `cLinks` line relative to the `ecpg/include` directory
K	The dependency is within the default `include` directory
L	Write the `cLinks` line relative to the `src/include` directory
M	Write the `cLinks` line relative to the directory the file is in
N	End

Table 1: The legend for Figure 1, mapping the nodes to the psuedo-code statements corresponding to the script source.

Using this algorithm, results have the potential to be incorrect if they do not match any of the above cases, which have not been accounted for. Additionally, if there are multiple files with the same base name, the dependency path may be ambiguous, and so the incorrect one may be chosen.

Finally, for each dependency, the script writes to the output `cLinks [name1] [name2]`, where `[name*]` is the path of the dependency and dependent. The result of the script is then output to a destination file using the TA format.

The program graph for the script is illustrated in Figure 1. [1] The legend mapping the nodes to the pseudo-code of the program statements is given in Table 1.

[1]Sequence diagram, use case diagrams, nor state diagrams are applicable for scripts, so this type of diagram was chosen instead.

2.2 Technique 2

For the third dependency-extraction method, we effectively leveraged the command-line options that are shipped with the GNU Compiler Collection (GCC).

As the reader might be aware, GCC is one of the most commonly used compiler families in the world and supports a variety of languages including C, Ada, and Fortran. GCC also happens to be the compiler of choice for the PostgreSQL project.

As part of its pre-processing stage, GCC provides the ability to automatically generate dependency information at the object-file level [1] [2] for any compiled source-code file. It must be noted here that dependencies at the object-file level are more than just an enumeration of the `#include` macros present in the source-code file. They include all the system and user-level dependencies that are needed (even if transitively [2]) to build that object-file. One can generate these dependencies using the `-M` flag as part of the overall `gcc` command [3]. This flag can then be refined to `-MMD` that will ultimately write user dependencies (it will omit system dependencies) after the pre-processing stage to a `.d`file.

However, compiling every single source code file manually using the aforementioned `gcc` flags would have been an arduous undertaking, requiring extensive trial-and-error to ensure that we have the right compiler command. We soon realized that the `Makefile` exists to allay that concern and provides a single point of compilation for a project. Makefiles are a simple way to organize code compilation [4] and contain the compiler commands (in this case gcc commands) needed to build any codebase from source.

We realized that we could generate the dependencies for all PostgreSQL source files if we made a few simple changes to the `gcc` commands in the project's `Makefile`. In other words, we could automatically obtain all object-file level dependencies while building the project from source in a single command ('make all'). This would give us a single source of truth for all compilation-related activities and save us the effort needed for manual dependency generation of every single source file. The question was to then find the appropriate `Makefile` and make the requisite changes to the flags. Further investigation showed us that all the PostgreSQL makefiles use the variables set in `/src/Makefile.global` file, which are in turn put there by the `configure` script. [2]

We consequently concluded that if we edit the 'configure' script such that it would add the `-MMD` flag to the `CFLAGS` variable in `/src/Makefile.global` file, we could generate all dependency files with a single run of the `make all` command. We eventually succeeded in getting the requisite `CFLAG` in the `/src/Makefile.global` file by editing the `CFLAG` variable in the `configure` script from `-Wall -Wmissing-prototypes -Wpointer-arith` to `-MMD -Wall -Wmissing-prototypes -Wpointer-arith`. [3] We then ran a `make all` command (after a `make clean` to remove all existing object and executable files). This yielded not only new object and executable files, but also dependency files (files with the `.d` extension). For an example, Figure 2 contains the list of dependency files created in the `/src/fe_utils` folder.

All dependency files were created alongside the respective object (`.o`) files. We copied the generated dependency files (consistent with the directory structure) to a separate location where we sought to parse all the dependencies and put them into a single file.

[2]Comments in the `/src/makefile.global` file

[3]A similar change was also applied to the `CXXFLAGS` variable for redundancy purposes

```
red 335 % pwd
/eecs/home/arshp98/eecs4314/postgresql-13.4/src/fe_utils
red 336 % ls -l *d
-rw------- 1 arshp98 ugrad 1044 Nov 20 18:48 archive.d
-rw------- 1 arshp98 ugrad  553 Nov 20 18:48 cancel.d
-rw------- 1 arshp98 ugrad  370 Nov 20 18:48 conditional.d
-rw------- 1 arshp98 ugrad  432 Nov 20 18:48 mbprint.d
-rw------- 1 arshp98 ugrad  504 Nov 20 18:48 print.d
-rw------- 1 arshp98 ugrad  559 Nov 20 18:48 psqlscan.d
-rw------- 1 arshp98 ugrad  574 Nov 20 18:48 recovery_gen.d
-rw------- 1 arshp98 ugrad  370 Nov 20 18:48 simple_list.d
-rw------- 1 arshp98 ugrad  566 Nov 20 18:48 string_utils.d
red 337 %
```

Figure 2: The list of dependency files created in the `/src/fe_utils` folder

To that end, we wrote a Python script that put all the generated dependencies for each source code file into an output file (that was meant to contain all the dependencies for all the source files). We were able to achieve this by parsing every single dependency file created and appending the correctly formatted combination of source and dependency files to the aforementioned output file. The dependency files that are created by the `-MMD` flag follow a standard convention that is fairly easy to parse using standard Python libraries. There were a few edge-cases we had to account for in generating absolute file paths, but we ultimately succeeded in parsing and collating the output of over 1000 dependency files.

Figure 3 illustrates the generated dependency file for `/src/fe_utils/simple_list.c`. The object file is `simple_list.o`, the source file is `simple_list.c` and everything else is a dependency header file. The equivalent data in the TA format is shown in Figure 4.

```
red 350 % cat simple_list.d
simple_list.o: simple_list.c ../../src/include/postgres_fe.h \
 ../../src/include/c.h ../../src/include/postgres_ext.h \
 ../../src/include/pg_config_ext.h ../../src/include/pg_config.h \
 ../../src/include/pg_config_manual.h ../../src/include/pg_config_os.h \
 ../../src/include/port.h ../../src/include/common/fe_memutils.h \
 ../../src/include/fe_utils/simple_list.h
red 351 %
```

Figure 3: The generated dependency file for `/src/backend/simple_list.c`

```
68411 $INSTANCE postgresql-13.4/src/fe_utils/simple_list.c cFile
68412 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/postgres_fe.h
68413 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/c.h
68414 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/postgres_ext.h
68415 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/pg_config_ext.h
68416 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/pg_config.h
68417 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/pg_config_manual.h
68418 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/pg_config_os.h
68419 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/port.h
68420 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/common/fe_memutils.h
68421 cLinks postgresql-13.4/src/fe_utils/simple_list.c postgresql-13.4/src/include/fe_utils/simple_list.h
```

Figure 4: The equivalent data in the TA format

Our solution was elegant in capturing the myriad complexity of compiling over a 1000 source-code files and generating dependencies for them. By effectively making changes to the `Makefile` itself, we were able to generate dependencies (from the compiler) with a single command while simultaneously avoiding any syntactic issues that would have surfaced with manual compiling.

2.2.1 Procedure

Figure 5 describes our second technique of dependency extraction, albeit in a graphical format. A description of the events in the flowchart is given below:

1. We start by editing the configure script such that the CFLAGS and the CXXFLAGS variables include the "-MMD" flag
2. Run the configure script
3. This will generate the `Makefile.global` file. You might recall that all other Makefiles within the Postgres Project get their variables from `Makefile.global`.
4. We then run the `make clean` command to delete any object files that may already exist
5. We then run the `make all` command to generate the object files (.o files) and the subsequent dependency files (.d files)
6. We then copy the `.d` files (while preserving the dependency structure) to a separate location. A command similar to `cp --parents 'find -name .d' ./dependencyfiles/` was run from the postgresql-13.4 folder. This created a folder `depenedencyfiles` that contained all the `.d` files.
7. We wrote and ran a Python script that parsed all the `.d` files. Effectively this Python script looked at every file in the `dependencyfiles` folder, parsed the contents and output them in the raw TA format.
8. The Python script gave us the dependencies for every source file, that we were then able to leverage for the analysis.

2.3 Differences between Approaches

For the second method, we used the GCC pre-processor to generate source-file dependencies at the object-file level. The dependencies can be obtained by including the "-MMD" flag as part of the gcc compilation command. As mentioned before, the dependencies at the object-file level are more than just an enumeration of the `#include` directives present in the source-code file. They include all the system and user-level dependencies that are needed (even if transitively [5]) to build that object-file. This set of dependency files can only be determined by pre-processing the source file. [5]

To generate dependencies for all the source files then, manually compiling every source file would have been an extremely time-consuming process. To solve that issue, we made one small change to the `configure` script such that it would include the "-MMD" flag in the compiler commands in the Makefiles, therefore giving us the ability to generate the dependency files while building the project. In taking this approach, we were effectively able to obtain dependencies for all the source files with a single run of the `make all` command.

This second method is different from the source code technique in three important ways:

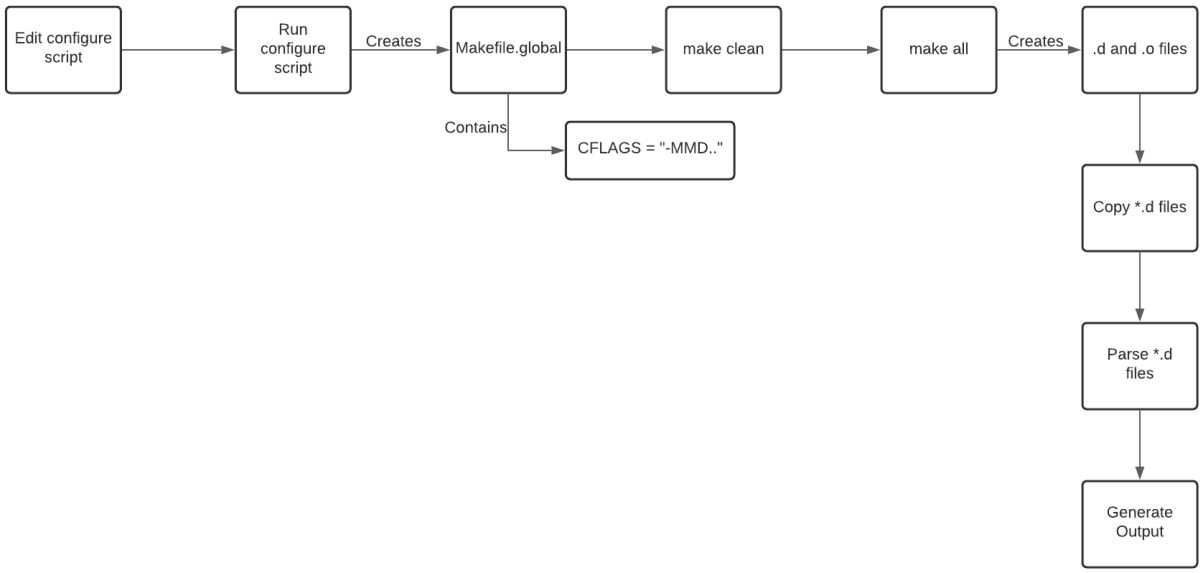

Figure 5: The flow chart for Technique 2.

- Our second method, in contrast to the first approach uses the compiler's in-built preprocessor options to generate dependency files. Consequently, our method was self-contained within the compiler and needed little-to-no manual intervention to generate dependencies. Instead, all the magic was done by the compiler itself at the object-file level. Indeed, our second method did not require us to look within the source file at all.

- Unlike our first method, our second method relied on the Makefile to generate dependencies while building the project. Again, this was achieved not by explicitly writing a script, but instead making a clever edit to the `configure` script itself.

- Parsing the `#include` directives of a source file would only give us the dependencies that are directly mentioned in the file. Our second method of using the GCC compiler gave us all the dependencies needed to build the object-file for a particular source file, including any transitive dependencies.

Therefore, through a combination of compiler options and Makefiles, we were able to develop an approach for generating dependencies at the object-file level while building PostgreSQL from source

3 Quantitative Analysis

3.1 Comparison Process

A python script was made to parse the output files created by each of the dependency extraction methods. Each output file was broken down into a dictionary, with each key

being the dependent file, and the corresponding value being a list of the dependencies. This was done by reading each line in the output files that begin with 'cLinks', and splitting it up into its dependent and its dependency. The output dictionary was then updated to reflect the new dependent-dependency pair.

Each of the dependency extraction methods were then compared against another technique. Using the created dictionaries, the total number of common dependencies were found as well as the percentage of common dependencies found between each dependency extraction method. Next, the common dependencies among all three methods were found. Each individual dependency extraction method's results were compared among this list of common dependencies to find its percentage of common dependencies.

3.2 Comparison Results

Table 2 compares the results of one dependency extraction method against one other dependency extraction method. The GCC method found the most dependent-dependency pairs at 70,022 total dependencies. This total is much greater than the other two methods, and will be further explored in the qualitative comparison section. Of the compared pairs of methods, the GCC and Understand methods had the most total common dependencies at 16,065 dependencies. The include and GCC methods had the least total common dependencies at 10,525 dependencies.

Method		**Total Deps.**		**Common Deps.**	**% Common Deps.**	
A	**B**	**A**	**B**		**A**	**B**
Include	Understand	13,779	31,433	11,218	81.41%	35.69%
Include	GCC	13,779	70,022	10,525	76.38%	15.03%
Understand	GCC	31,433	70,022	16,065	51.11%	22.94%

Table 2: Comparing each dependency extraction method against another dependency extraction method

Table 3 shows the results of comparing all three dependency extraction methods against each other. Overall, there were 9609 common dependencies. Of the 9609 common dependencies, 69.74% were found in the Include methods dependencies, 13.72% were found in the GCC methods dependencies, and 30.57% were found in the Understand methods dependencies. These results will be explored further in the qualitative comparison section.

Total Common Dependencies		9,609
% Total Dependencies	Include	69.74%
	GCC	13.72%
	Understand	30.57%

Table 3: Comparing each dependency extraction technique with the total common dependencies

4 Qualitative Analysis

4.1 Comparison Process

Three output files from three different methods were compared line by line: Each line in a file was compared to the content of the other two files. If none of the other files contain this line, the dependency was identified as a discrepancy. The discrepancy was then written to the discrepancy file. Three discrepancy files were generated through this process.

The Understand discrepancy file has 17,867 discrepancies. The include method file has 552 discrepancies. The GCC compiler file has 52825 discrepancies. There are 71,244 discrepancies in total. According to the sample calculator, to achieve a 95% confidence interval, 382 samples must be investigated.

Using stratified sampling, 96 samples (25% of total samples) from the Understand discrepancy file, 4 samples (1% of total samples) from the include method file, and 282 samples (74% of total samples) from the GCC compiler file were randomly selected. Each of the discrepancies was manually analyzed, and the result of the analysis is listed below.

	Understand	**Include**	**GCC**	**Total**
All Discrepancies	17,867	552	52,825	71,244
Sampled Discrepancies	96	4	282	382

Table 4: Discrepancies sampled using stratified sampling techniques

4.2 Comparison Results

The sampled discrepancies for each method were analyzed manually and the causes of each discrepancy were identified.

The include method generated the smallest number of dependencies and had only 552 discrepancies compared to the other two methods. All 4 sample discrepancies were because the dependency path was incorrectly identified. These errors were mostly caused by unhandled edge cases while parsing the include header. All discrepancies were invalid dependencies.

Out of the 282 sample GCC discrepancies, 251 (89%) of them were transitive dependencies that other methods failed to identify since the GCC method identifies all files needed for a c file to run. 18 (6.4%) of the discrepancies were because the dependency path was incorrectly identified. Some of the dependencies could not be found manually and were thus marked as non-existent. These non-existent dependencies accounted for 13 (4.6%) of the total discrepancies. 31 (11%) of the discrepancies were invalid dependencies.

For the Understand method, the majority of the discrepancies (72.9%) were caused by incorrectly-identified dependency paths. The other discrepancies were because the Understand method looks into dependencies between header files and conditional dependencies. 20 (20.8%) of the total discrepancies were caused by dependencies between header files, while the remainder of them were because they are conditional dependencies. 26 (21.1%) of the discrepancies were invalid dependencies.

5 Implications & Recommendations

According to the qualitative analysis conducted, the vast majority of dependencies (85%) generated by the GCC method were unique due to this method finding transitive dependencies. This result could be exaggerated due to a significant portion of transitive dependencies being sampled. A larger sample size can counteract this issue.

A substantial amount of discrepancies analyzed by the Understand method (72.9%) were found to be invalid due to incorrect dependency paths. 31 of the dependencies analyzed in the GCC method were found to be invalid either because the path could not be found manually or the dependency path itself was invalid. Because these qualitative analyses were done manually there is the possibility of human error. Automated checking for dependency path verification will reduce the possibility of human error and improve the Understand and GCC methods.

The include method only had 4 discrepancies analyzed due to the small number of dependencies generated. Furthermore, all 4 dependencies were invalid due to unhandled edge cases in the include method. A larger sample size and appropriate handling of edge cases will result in a more accurate analysis.

6 Comparison Metrics

The precision and recall was calculated for each of the dependency extraction methods. An 'oracle' was put together consisting of the common dependencies amongst all results. Precision was calculated by dividing the number of common dependencies between a method and the oracle by the total number of dependencies from that particular method. Recall was calculated by dividing the number of common dependencies between a method and the oracle by the number of dependencies in the oracle.

Method	**Precision**	**Recall**
Include	0.49	0.70
GCC	0.05	0.16
Understand	0.09	0.82

Table 5: Calculated precision and recall for each dependency extraction method

7 Risks & Limitations of Comparison Approaches

There are certain risks and limitations associated with the comparison approaches. For the calculation of precision and recall, the lack of a real oracle poses some risks. An invalid dependency may not be discovered if it is produced by at least two methods. Such limitation may result in missing discrepancy cause and incorrect precision or recall.

Since the include method only produced a smaller amount of dependencies and most of its dependencies were also generated by the other methods, only 552 discrepancies were found. Since there are many discrepancies produced by the other two methods, only 4 discrepancies

from the include method were sampled. These 4 samples may not be a representative sample of the entire population. Some causes behind discrepancies might be missed. This limitation affects the result of the qualitative analysis.

The GCC method is the only method that produces transitive dependencies. Many discrepancies of this method are transitive dependencies. Out of the 282 discrepancies sampled, more than 85% are unique due to this reason. Some other causes behind discrepancies may be missed since a large number of transitive dependencies were sampled. A sampling issue like this may result in an incomplete qualitative analysis.

Also, transitive dependencies are hard to analyze manually. To confirm a dependency is transitive, one must look through a long line of files to confirm such dependency exists. Some transitive dependencies may be mislabelled as non-existent dependencies for this reason. A limitation like this may affect the calculation of precision and recall.

The risks and limitations listed above can be resolved by utilizing an oracle and/or running the methods on a smaller system.

8 Conclusion

Through the last assignment we learned that agreeing on an output format for the dependency extraction techniques simplifies the qualitative and quantitative comparison process because the output file for the total different dependencies was huge and having them in the same format would make it easier to compare. We also learned that using multiple dependency extraction techniques can increase the quality of the results due to some techniques not picking up certain types of dependencies. An example of this is from the GCC extraction method. The GCC method picked up a lot of transitive dependencies that the other two methods did not pick up, increasing the quality of the results. Another lesson learned was that constant communication was required to ensure that all members were working together in regards to the comparison process since there were over 300 different dependencies to analyze.

In conclusion we were able to build a python script to extract the dependencies discrepancies from the three methods, GCC, Include and Understand and manually make a qualitative and quantitative comparison. Through the results it is evident that using different dependencies methods helped increase the accuracy of the results.The GCC method was the most significant technique because it found 251 transitive dependencies discrepancies which are considered valid dependencies. The cost of using different methods is the increase of non valid dependencies caused by incorrectly defined dependency paths and dependency files that could not be found.

References

[1] P. D. Smith, "Auto-dependency generation," GNU make, 23-Jun-2000. [Online]. Available: https://make.mad-scientist.net/papers/advanced-auto-dependency-generation/. [Accessed: 20-Nov-2021].

[2] E. Jones, "Builds are complicated," Builds are complicated: C/C++ (evanjones.ca), 03-Mar-2012. [Online]. Available: https://www.evanjones.ca/build-c.html. [Accessed: 21-Nov-2021].

[3] "Options Controlling the Preprocessor" GCC Online Documentation. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html. [Accessed: 18-Nov-2021].

[4] B. A. Maxwell, "A Simple Makefile Tutorial," Maxwell's Tutorials. [Online]. Available: https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/. [Accessed: 20-Nov-2021].

[5] E. Jones, "Builds are complicated," Builds are complicated: C/C++ (evanjones.ca), 03-Mar-2012. [Online]. Available: https://www.evanjones.ca/build-c.html. [Accessed: 21-Nov-2021].

[6] "Options Controlling the Preprocessor" GCC Online Documentation. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html. [Accessed: 18-Nov-2021].