# EECS 4314 - Advanced Software Engineering

# OpenCV Dependency Analysis

Eric Rodrigues, Laura Marin, Alp Baran Sirek, Negar Khalilazar, Danny Le

ericr100@my.yorku.ca, nemira@my.yorku.ca, hiangel@my.yorku.ca, nkhazar@my.yorku.ca, dannyle3@my.yorku.ca

Dr. Zhen Ming (Jack) Jiang

Tuesday, December 2, 2025

## Table of Contents



# Abstract

This report investigates and compares three architectural dependency extraction techniques applied to OpenCV v4.12.0, where the goal is to evaluate how the three different approaches capture dependencies in OpenCV and mainly to assess their accuracy and coverage while also analyzing the practicality of these dependency extraction techniques. The first technique utilizes the SciTools Understand to extract the set of dependencies based on its own analysis engine. The second technique implements a custom python script that tries to identify the dependency relationships based on the include directives. The third technique used for the data extraction used for the analysis was the srcML, an XML based source code transformation tool to extract the dependencies by analyzing the program's abstracted structure [1]. Different extraction methods exist for large software programs which are built for specific tasks. Analyzing how they work and comparing the results based on a quantitative and qualitative analysis on these tools on OpenCV, an open source and a relatively large project, the aim is to extract the strengths, weaknesses, limitations and the trade offs of each technique. We have found that Understand excels when we want a fuller picture of the architecture, that the include extraction technique is fast and simple, and that srcML can be used when we need to audit specific disagreements. However, none of the tools provide complete dependency extraction, and the most complete and accurate results would be obtained through a combination of more than one tool.

# Introduction and Overview

OpenCV is a popular tool used on tasks such as image processing, computer vision and real-time video analysis. It provides optimized filters, feature extraction, tracking, and deep-learning inference across multiple programming languages [2]. It was developed at Intel Research in 1999 and publicly released in 2000, OpenCV has since been extended by many contributors and is now maintained by OpenCV.org. Early versions focused on efficient C and C++ implementations, currently with the shifted focus on C++ and Python for more modern work [3].

SciTools's Understand application is a tool that helps developers analyze the source code of large projects with access, by building a database of various structural elements, for our case most importantly being the dependencies. srcML on the other hand is a different extraction tool which allows us to look at dependencies in an xml format, where it converts source code into XML based interpretations/representations while keeping the structure of the program intact, not just a text based interpretation of the program.

# Program Dependency Extractions

**Include**
Our approach for extracting dependencies from OpenCV based on the include directives revolved around a python script where first each file directory in OpenCV 4.12.0 is searched with the matching file extensions ".c",".cpp",".cxx",".h",".hpp",".hh",".cc",".cpp". This was done using the "findFilesFirst" function which returns all the directories to files with those extensions. Once the function yielded all the directories, each and every file is scanned line by line looking for the "include" lines using the regex generated by an LLM model: (r'^\s*#\s*include\s*(<|")([^">]+)(>|")') within the findImports helper function.

Once an include statement is caught, the script attempts to resolve the referenced path using the helper method, edtPth. This function first checks whether the include path exists directly within the project root directory, if it does not then it performs a recursive walk to locate any file in OpenCV whose "basename" matches the include path. The behaviour of the script can be seen on the flow diagram section Figure 8.

The csv file is generated in the very beginning of the main function which every time that an include line is resolved, the line in a very specific format is written into the csv file. After the main double loop is done, the csv file is closed and the script terminates.
For the include method, every file was considered including the 3rdparty/, samples/, modules/js/, tests/, apps/, and platforms/ directories which could explain why a larger amount of dependencies than expected were found.

**SourceML**

Our second approach for extracting dependencies from OpenCV leveraged srcML, a tool that converts source code into a structured XML representation. Before describing the full extraction process, it is important to introduce srcML, the core tool used in our workflow, srcML converts source code into an XML representation while preserving the original structure and semantic information of the code. This transformation enables structural analysis through XML querying, allowing dependencies and relationships to be extracted programmatically. In practice, srcML performs efficiently; in our case, converting the entire OpenCV codebase required only a few minutes. With this foundation in place, the following section outlines the procedure we used to apply srcML for architectural dependency extraction.

To extract OpenCV's architectural dependencies using srcML, we divided the process into several stages. First, we wrote a shell script to recursively traverse the entire OpenCV source tree and record every file path. These file paths were collected into a text file named "files", which served as the input list for batch XML conversion. Next, we executed a second shell script that applied srcML to each source file in the list, generating corresponding XML representations for all OpenCV components. These XML files were then processed by our custom Python program, "extract_dependencies". The program systematically traverses the XML structure, identifies file-level program dependencies, and outputs these relationships in TA format. This output format ensures compatibility with our broader analysis workflow and enables direct comparison with the dependency sets produced by the Understand tool and by our include-based extraction approach. This entire extraction process is also visually described in Figure 9.

# Quantitative Comparison

The total dependencies extracted were 66,379, 54,676 being unique. For easier comparison, the dependencies extracted for each tool are listed in the table below.

| Dependency Technique | Total Dependencies |
|---|---|
| Understand | 26,058 |
| Include | 31,041 |
| srcML | 9,280 |

Figure 3.

When looking at the overlap in the venn diagrams from Figure 4, we see that Understand and Include have 3,359 dependencies in common (meaning there is an overlap coverage of about 6%), Understand and srcML have 3,417 dependencies in common (overlap coverage of about 10%), and Include and srcML have 4,927 dependencies in common (overlap coverage of about 12%).
It is important to note that Understand didn't include folders at the top-layer of OpenCV's source-code folder, like the "samples" folder. But the srcML and our "include" directive analyses did. So there also existed unique edges per technique, where some dependencies didn’t show up at all in the Understand, but did in Include and srcML.

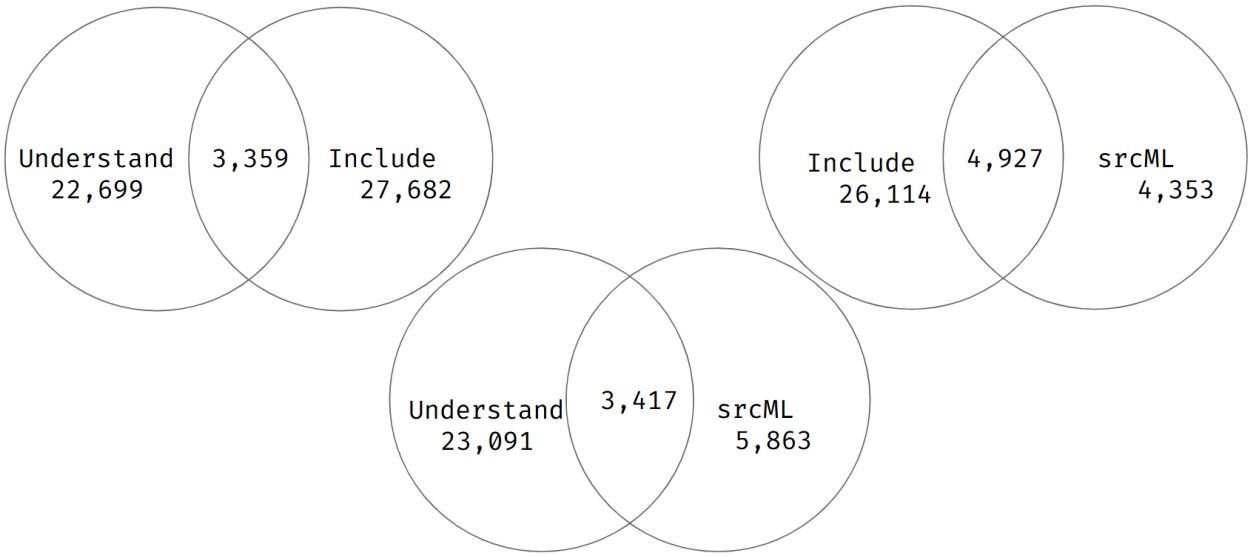


Figure 4.

To dive deeper into the quantitative comparison, we needed to manually review a subset of the dependency differences among the three tools. Because the total number of differences

was very large, examining every case manually would not be realistic. To handle this, we used a standard statistical approach, following the guidelines provided by the Sample Size Calculator from Creative Research Systems [4]. Precision (also called positive predictive value) is the fraction of relevant instances among the retrieved instances. Recall (also known as sensitivity) is the fraction of relevant instances that were retrieved. [5]
We chose to compare the venn diagrams in three steps. First, we compared Understand vs Include. Using the calculator we got a sample size of 382. Then applying a script to determine the unique Understand (130), unique Include (182), and common cases (70) within the sample size, we applied the formula of precision and Recall to each tool. The precision and recall of Include was 27.78% and 35%, respectively. The precision and recall of Understand was 35% and 27.78%, respectively.

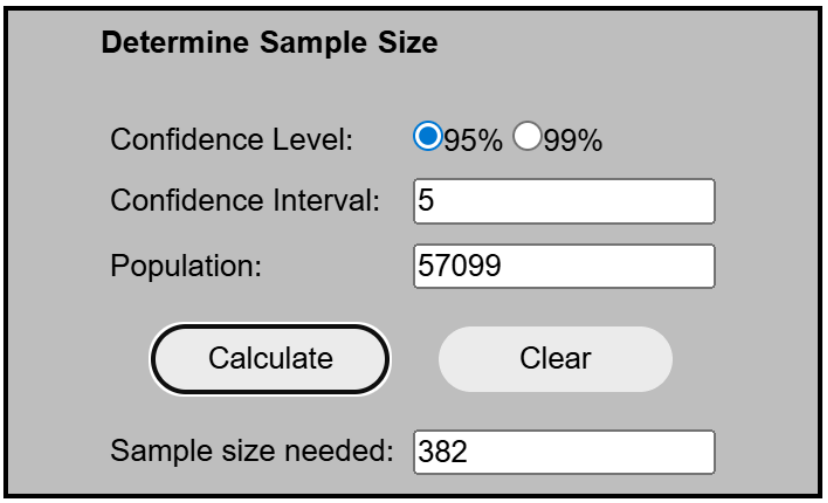


Figure 5. Understand vs Include Sample Size Calculator

Secondly, we compared Understand vs srcML. Using the calculator we got a sample size of 380. Then applying a script to determine the unique Understand (320), unique srcML (40), and common cases (20) within the sample size, we applied the formula of precision and recall to each tool. The precision and recall of srcML was 66.67% and 11.11%, respectively. The precision and recall of Understand was 11.11% and 66.67%, respectively.

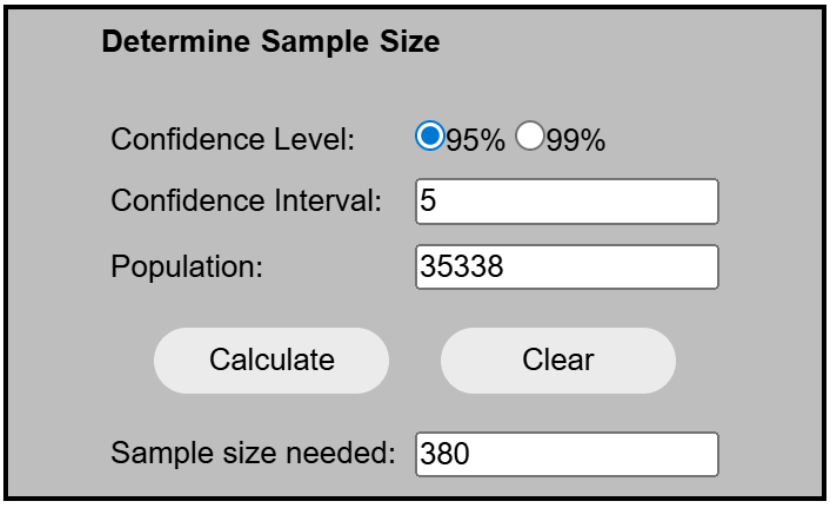

Figure 6. Understand vs srcML Sample Size Calculator

Thirdly and finally, we compared Include vs srcML. Using the calculator we got a sample size of 381. Then applying a script to determine the unique Include (298), unique srcML (35), and common cases (48) within the sample size, we applied the formula of precision and recall to each tool. The precision and recall of srcML was 42.17% and 10.51%, respectively. The precision and recall of Include was 10.51% and 42.17%, respectively.

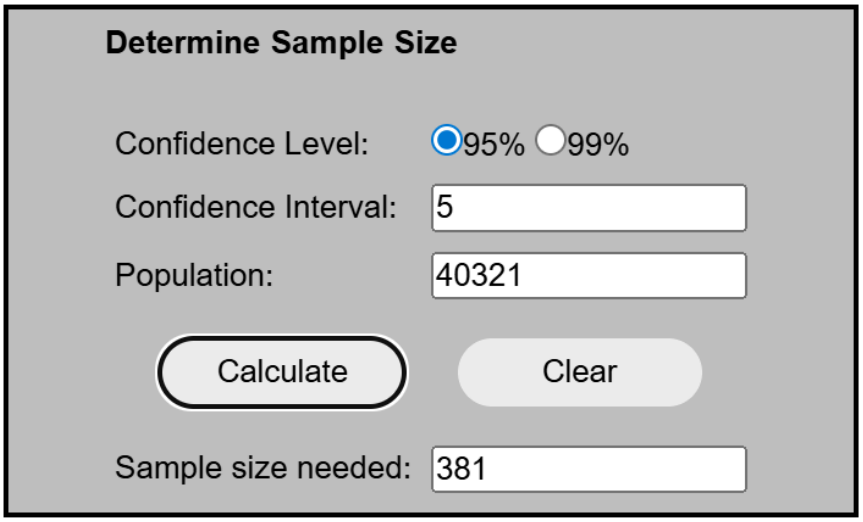


Figure 7. Include vs srcML Sample Size Calculator

The quantitative comparison across the three tools reveals the differences in their coverages and agreement. Each tool identified a substantial number of unique dependencies, indicating the existence of limited overlap, which means each tool captures different aspects of the dependencies extracted.

Understand and Include were somewhat similar to one another, but neither found everything the other did. srcML found the least dependencies overall, but the ones it found were more likely to match what Understand and Include found when analyzed pairwise.

This demonstrates that a single tool does not provide complete coverage. So to get the most complete and most accurate results, it would be best to use more than one tool.

# Qualitative Comparison

For the qualitative comparison, instead of counting the dependencies, we used how each tool extracted the dependencies and why their results differ as our metric. This involves looking at how the tools work, which relationships they detect, and where they over-report or under-report dependencies.

Understand, Include, and srcML each analyze the source code in different ways. Include performs a syntactic level extraction, meaning it only detects the raw #include directives in the source files. It does not interpret whether these directives are active, commented out, or conditionally compiled, so it tends to report the highest number of dependencies. This also means that Include over-reports, since it does not distinguish between dependencies that are actually used and those that are in inactive source files. If we were to use Include again, we would find a way to remove inactive or commented-out includes to reduce the noise and focus on the real dependencies.

Understand performs a semantic-level analysis, It examines more that include relationships by also detecting function calls, variable references, type usage, inheritance relationships, and other semantic connections between files. Its extraction is therefore more meaningful than that of Include. Nonetheless, Understand missed certain top-level folders in the OpenCV codebase, which caused certain dependencies that the other tools detected to be missed completely. This is part of the reason why the gaps in coverage exist, although it has more advanced analysis capabilities.

Lastly, srcML performs a structural-level extraction. It converts the source code into an XML representation and detects elements like <call> for function-call expressions and <cpp:include> for the include dependencies. Due to the immense size of the OpenCV codebase, srcMl struggled with a system-wide extraction and reported the fewest dependencies. The results tended to be precise when the dependencies were successfully detected, but the limits in scalability led to under-reporting compared to the other tools.

All in all, the differences of the depth of analysis, design, and the scalability between each tool explains the discrepancies seen in the venn diagram overlaps and precision and recall scores. Each tool finds a different aspect of OpenCV’s dependency structure, and no single tool provides complete coverage across all dependency types. We would recommend using Understand as the primary tool, to get all the semantic details, but also use Include and srcMl to cover missing files and structural edges.

# Potential Risks and Limitations

The only limitation that we found is that srcML could not parse every file of OpenCV, these being “3rdparty/libjpeg-turbo/src/jdhuff.c”, “zlib/deflate.c”, “libspng/spng.c”, and “3rdparty/libjpeg-turbo/simd/arm/aarch64/jchuff-neon.c”. Even though those files were syntactically correct for a proper C compiler, they were too complicated for srcML’s partial parser and limited macro expansion engine to fully process, which led to segmentation faults. We originally had srcML convert all of OpenCV’s source files into a single, large XML file; but this file would break our script to convert it into a TA file. To work around this issue, we instead developed a short script to convert each file of OpenCV into an individual XML file.

It is also possible OpenCV’s source code contains include directives that were commented out, or are no longer needed but unnecessarily remain in the code. In such cases, our script to find all of the include statements from the XML files produced by srcML, along with the script used to extract the “include” directives, would count these extra dependencies.

# Flow Diagrams of Include & srcML Extractions

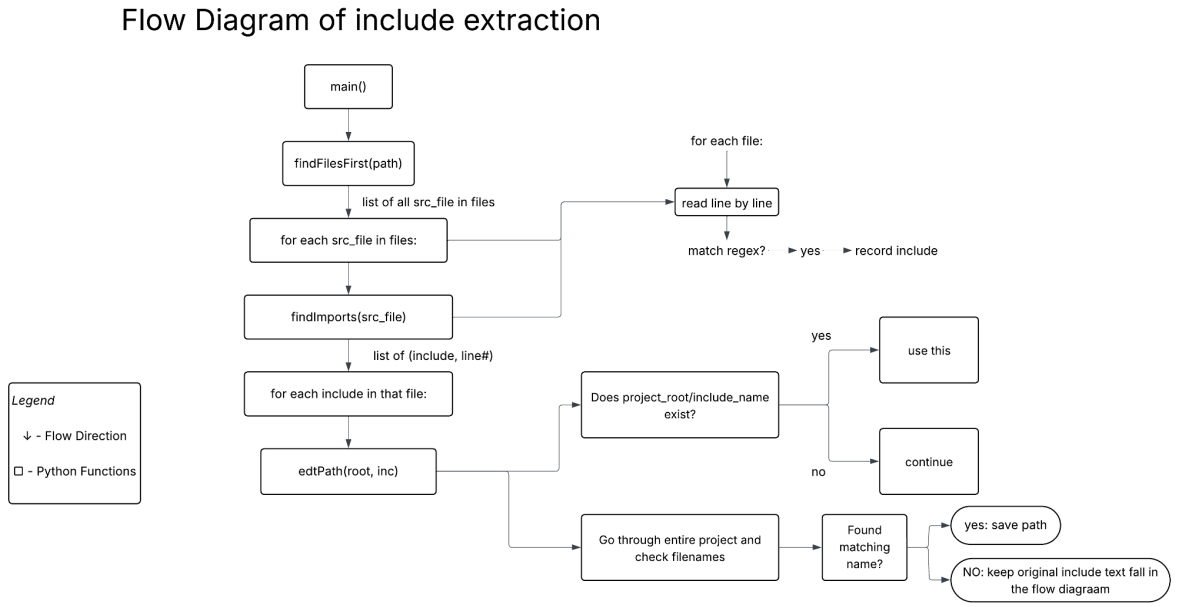


Figure 8.

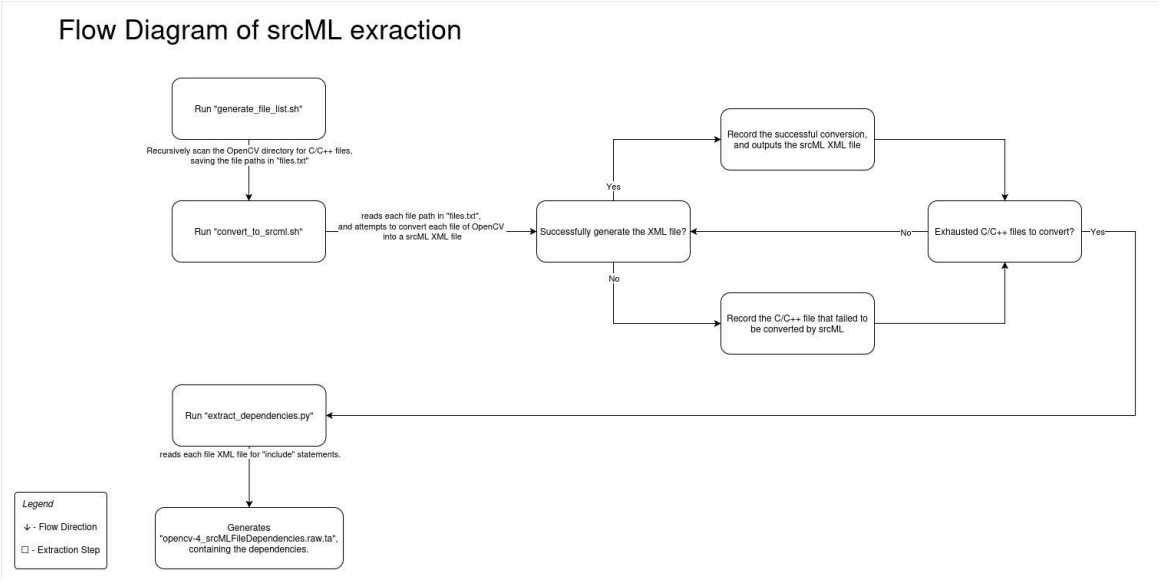


Figure 9.

# Conclusion

This report compared three static techniques for extracting architectural dependencies from OpenCV 4.12.0 and showed that they do not agree on a single “correct” view of the system. By running Understand, an include-based Python extractor, and a srcML-based pipeline on the same codebase and converting all outputs to a common format, we were able to measure how many dependencies each one reported, how much they overlapped, and where they disagreed in practice. The analysis confirmed that configuration choices, directory coverage, and the level of abstraction each tool targets all have a direct impact on the dependencies that end up in the final dataset.

Going forward, there are a few natural next steps. One is to tighten the extraction techniques themselves, for example by improving path resolution, filtering out unused includes, and pushing srcML further toward function-level or class-level relationships. Another is to repeat the same comparison process on other large systems, or on newer versions of OpenCV, to see whether the same patterns hold. Finally, integrating these dependency datasets with higher-level architectural views, such as subsystem diagrams or evolution histories, could make the extracted information more directly useful for refactoring, impact analysis, and future maintenance work.

# Lessons Learned

Include extraction ended up being the quickest way to get a dependency snapshot of OpenCV. It was easy to script, ran over the whole tree in a short time, and gave us a simple file-to-file view based purely on #include lines. The downside is that it only reflects what shows up in the source text, so it can miss real relationships and also report edges that are never actually used or compiled.

Understand gave us a much better picture of the full architecture. It captured function calls, type usage, and other semantic links that the include script and srcML could not see, which made its results more meaningful when we cared about how subsystems really interact. We also saw that its output is very sensitive to configuration details, such as which top-level folders are part of the project, so getting that setup right early is important.

Putting everything together, the most useful approach was a hybrid one. Using the include script as a fast first pass, then relying on Understand to refine and confirm key dependencies, and finally using srcML or small manual samples to investigate disagreements gave a more reliable view than any single technique. Working at this scale also taught us that we need basic statistics like sampling, precision, and recall to compare extraction methods in a way that is realistic and defensible.

# References

[1] *srcML*. “About.” srcML, www.srcml.org/about.html.

[2] “Article Title.” *ACM Queue*, 2012, spawn-queue.acm.org/doi/10.1145/2181796.2206309.

[3] “About.” *OpenCV*, opencv.org/about/.

[4] *The Survey System*. “Sample Size Calculator.” www.surveysystem.com/sscalc.htm.

[5] “Precision and Recall.” *Wikipedia*, en.wikipedia.org/wiki/Precision_and_recall.