EECS 4314 - Bit Theory
Dependency Extraction Report

Amir Mohamad
amohamad@my.yorku.ca

Arian Mohamad Hosaini
mohama23@my.yorku.ca

Dante Laviolette
dantelav@my.yorku.ca

Diego Santosuosso Salerno
nicodemo@my.yorku.ca

Isaiah Linares
isaiah88@my.yorku.ca

Joel Fagen
joefagan@my.yorku.ca

Misato Shimizu
misato1@my.yorku.ca

Muhammad Hassan
furquanh@my.yorku.ca

Yi Qin
aidenqin@my.yorku.ca

Zhilong Lin
lzl1114@my.yorku.ca

York University

March 24, 2023

Abstract

Dependency extraction for source code is a critical task in software engineering which involves identifying and extracting software components from the repositories. It is a process of obtaining source code from compiled executables, and it is essential for reverse engineering, software analysis and debugging. We can find the relationships, including function calls, variable references, and inheritance relationships among different components in the system. We will be able to maintain the software architecture features. By achieving these features, we can minimize changes' impact and ensure the system is always scalable and maintainable. It will also help refactor the code by identifying the components that might need to change from different files. In the report, we will extract the source code from the FreeBSD operating system and find the relationships between the components inside the system, including Understand, SrcML and our custom method. During the research process, we used two different techniques to get more precise results and better compare the three methods. Based on these techniques, the quantitative and qualitative helped to show more secrete about dependency extraction. After those processes, we successfully compared three extraction methods and summarized their advantages and limitations.

Keywords— dependency extraction, Understand, SrcML, quantitative, qualitative

Contents

1 Introduction

In the report, we will be analyzing dependency extraction in software architecture development. As software changes over time, the architecture might be changed a lot. The process of dependency extraction will be able to reconstruct the structure of the software architecture. Dependency extraction allows developers to understand the meaning of the code-based information better and can improve the accuracy of automated text analysis and processing. Understand is a commercial software tool that supports many programming languages. It can generate different reports such as metrics, dependency, and call graphs by analyzing the source code from the software. After analysis of the source code, it will also create a dependency graph which shows the relationships between different components and entities inside the source code. These dependency graphs will give the developers a clear view of the effect of each component and how these components will work together. The official website briefly and concisely explains the functionalities of SrcML, which is an infrastructure for the exploration, analysis, and manipulation of source code and a lightweight, highly scalable, robust, multi-language parsing tool to convert source code into SrcML. Compared to Understand, SrcML is an open-source software system that provides a way to generate the source code in an XML format, and XML format can be easily analyzed and manipulated using other software tools by the developers. SrcML can use different programming languages as input, and SrcML captures both the syntax and semantics of the code. By analyzing the source code, SrcML will depend on the code's structure, including the variables' names and types.

1.1 Overview

The report contains several parts. Firstly, we will discuss the derivation process of the three methods of extracting dependencies, and we will show the use cases and diagram. We will discuss three main extraction strategies we decide to use in the dependency extraction: understand, SrcML and custom. Then, we will discuss the quantitative and qualitative analysis to explain the performance metrics. Before concluding, we will analyze the limitations of the techniques used and explained in the report.

2 Dependency Extraction

2.1 Understand

The first technique we used to extract dependencies of the FreeBSD kernel is using the Understand static code analysis IDE. We used it to analyze the FreeBSD source code and generate the extracted dependencies in `freebsd_FileDependencies.csv` format. This CSV file is then converted to a `raw.ta` file using `csv-to-ta.pl` perl script. The rest of the process after the `Start A`, in figure 1, state is complete is the same as the assignment 2 iterative architectural decomposition. This process takes roughly over 3 hours to complete from start to end. A more comprehensive set of instructions of extracting dependencies with Understand can be found at [3].

2.2 SrcML

The second technique we used to extract dependencies of the FreeBSD kernel is using the SrcML tool to convert our source code into xml and perform `xPath` queries. We used it to analyze the FreeBSD source code and generate the extracted dependencies in `freebsd_FileDependencies.xml` format. Similarly, the rest of the process after the `Start A` state, is the same iterative process. The SrcML method uses an `xPath` query, `src:unit[@type='include']/@filename | cpp:include/cpp:file`, on the XML to extract the include dependencies. This method takes about 20 minutes for the entire process.

2.2.1 Installing and Configuring SrcML

we installed the SrcML tool on our local machines. SrcML is an open-source tool designed to convert source code into an XML format, which makes it easier to analyze and manipulate using XPath queries. After installing SrcML, we familiarized ourselves with its documentation and basic usage, enabling us to use it effectively in the next steps.

2.2.2 Converting the FreeBSD Kernel Source Code to XML

With SrcML installed and configured, we proceeded to convert the entire FreeBSD kernel source code into an XML format. Using the command:

```
srcml freebsd/src/sys -o freebsd_kernel.xml
```

we directed SrcML to process the source code located in the `freebsd/src/sys` directory and output the resulting XML to a file named `freebsd_kernel.xml`. This step was essential for enabling us to perform XPath queries on the source code data.

2.2.3 Extracting Include Units and Files with XPath

Now that we had the kernel source code in XML format, we used an XPath query to extract all the include units and their respective files. We executed the following command:

```
srcml freebsd_kernel.xml --xpath="//src:unit[@type='include']/@filename | //cpp:include/cpp:fi
```

This XPath query searched for all `src:unit` elements with a `type` attribute set to `include`, as well as all `cpp:include/cpp:file` elements. The query then returned the `filename` attribute of each matching element. This step provided us with a comprehensive list of all the included files and their dependencies.

2.2.4 Generating a Raw TA File

With the data extracted, we created a custom script to parse the output from the previous step and generate a raw TA file. Our custom script iterated through the list of include units and files, reformatting the data into the desired format and outputting it to the raw TA file.

2.2.5 SrcML Conclusion

In conclusion, we have successfully extracted file-level dependencies within the FreeBSD kernel using the SrcML tool and a series of XPath queries. SrcML has some limitations that users should be aware of when using it for source code analysis. it can be computationally intensive and time-consuming when processing large codebases, potentially impacting the efficiency of the analysis. Finally, SrcML's XML output can be cumbersome to work with and may require additional tools or expertise to manipulate and analyze effectively, However, being efficient in XPath Queries can help save time in writing multiple lines of code.

2.3 Custom Script

The last technique we used to extract dependencies of the FreeBSD kernel is using a custom script to recursively scan our source code and match include directives with regex. This is all done in our `gen-ta-file.py` script. We used this script to generate the `freebsd_FileDependencies.raw.ta` file without any intermediate tools. Similarly, the rest of the processes after the `Start A` state is complete, is the same iterative process mentioned earlier. This method is the fastest of the three since it does not have intermediate steps for `Start A` state. It takes about 10 minutes.

2.4 Diagrams

Figure 1, 2, and 3 below are a more detailed overview of the derivation process of each dependency extraction method.

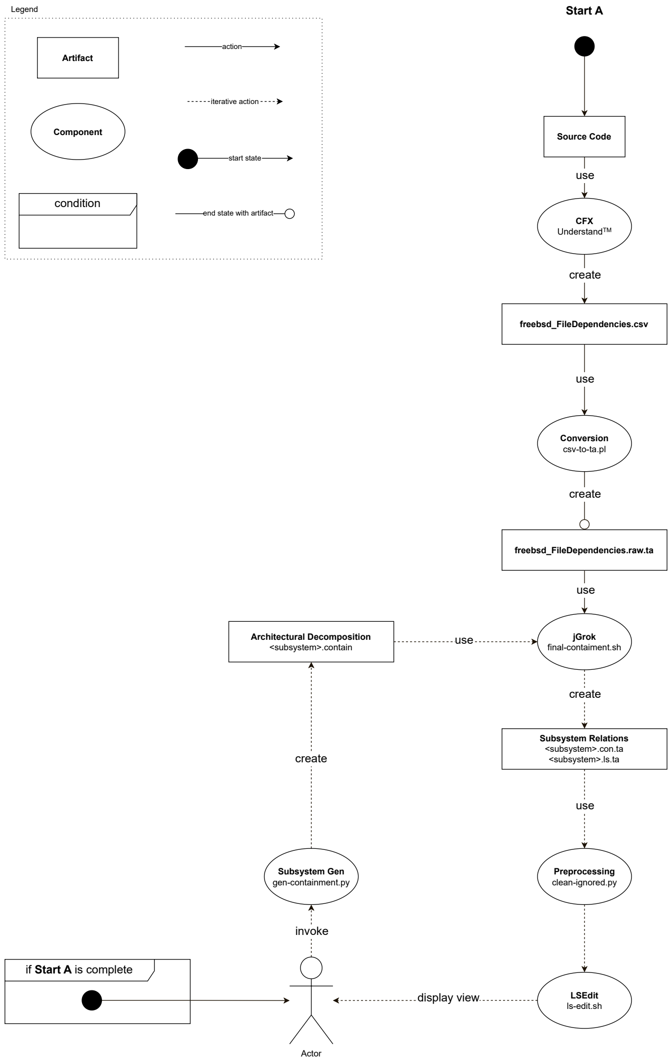


Figure 1: Understand dependency extraction derivation process.[3][2]

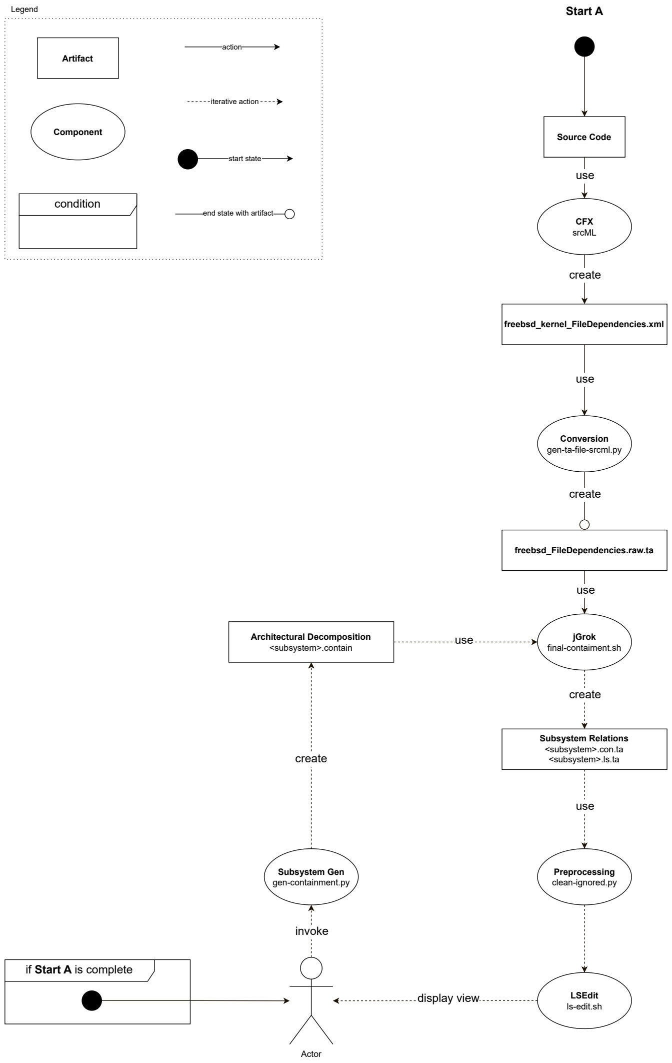


Figure 2: SrcML dependency extraction derivation process. [2]

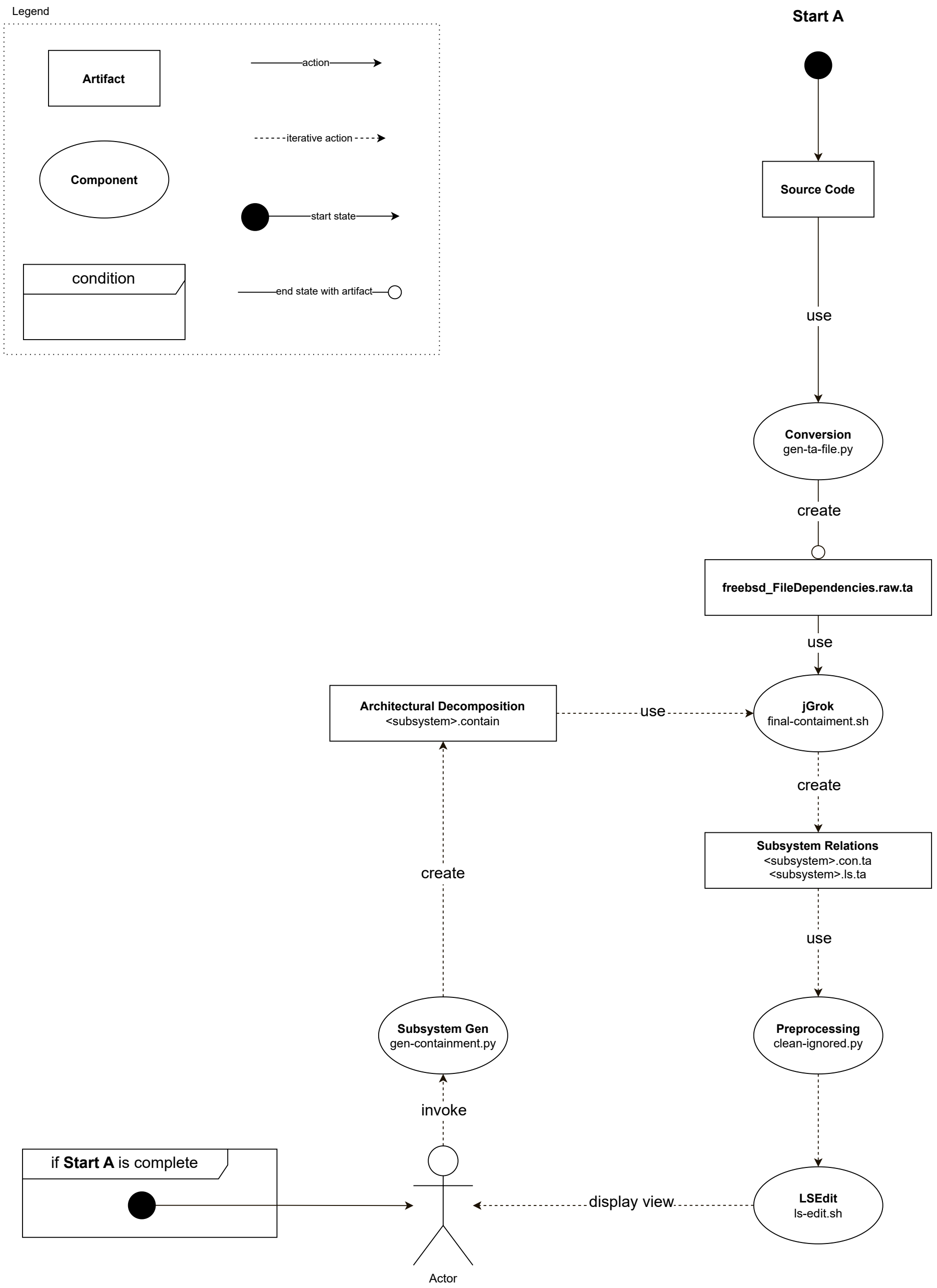


Figure 3: Custom dependency extraction derivation process.[2]

2.5 Generation of Raw TA Files

The main difference between our various dependency extraction processes is the generation of the `*.raw.ta` file. Once this file has been generated, the processes generally follow the same steps. With that being said, the following diagrams show how the `*.raw.ta` files are generated in our 2 custom methods.

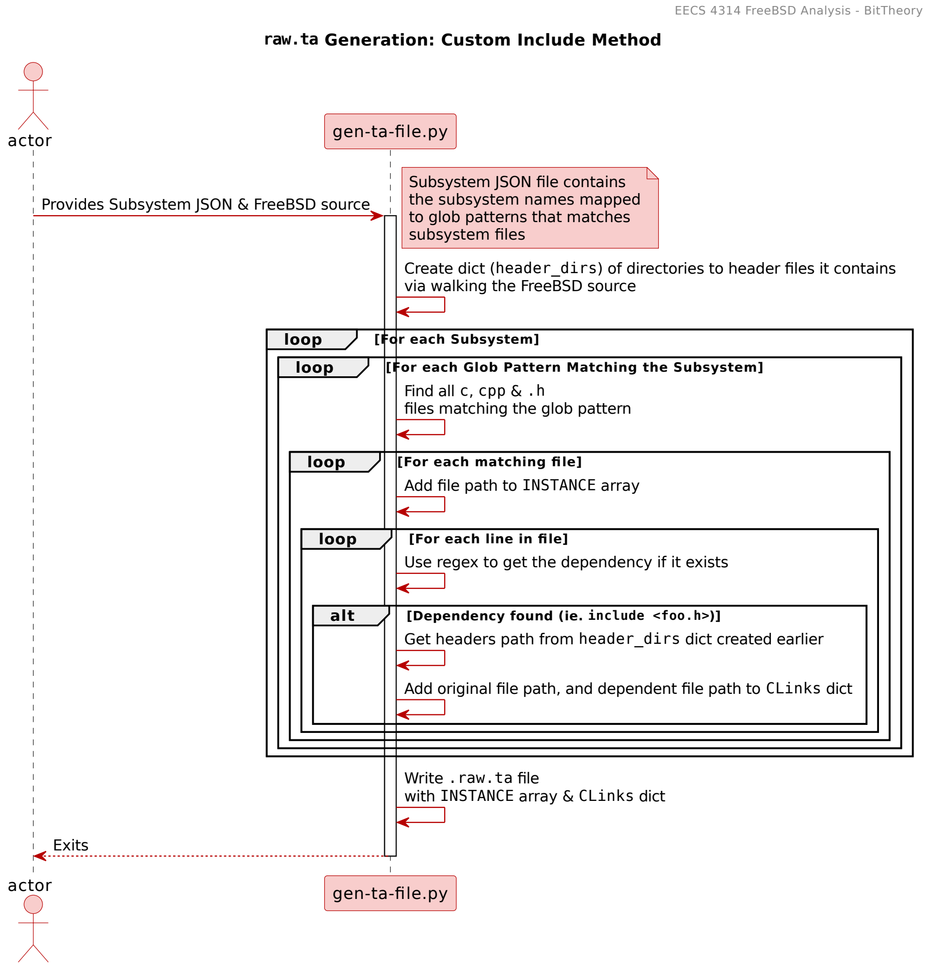

EECS 4314 FreeBSD Analysis - BitTheory
raw.ta Generation: Custom Include Method
actor
gen-ta-file.py
Provides Subsystem JSON & FreeBSD source
Subsystem JSON file contains the subsystem names mapped to glob patterns that matches subsystem files
Create dict (header_dirs) of directories to header files it contains via walking the FreeBSD source
loop
[For each Subsystem]
loop
[For each Glob Pattern Matching the Subsystem]
Find all c, cpp & .h files matching the glob pattern
loop
[For each matching file]
Add file path to INSTANCE array
loop
[For each line in file]
Use regex to get the dependency if it exists
alt
[Dependency found (ie. include <foo.h>)]
Get headers path from header_dirs dict created earlier
Add original file path, and dependent file path to CLinks dict
Write .raw.ta file with INSTANCE array & CLinks dict
Exits
actor
gen-ta-file.py

Figure 4: Raw TA file generation using custom includes script.[2]

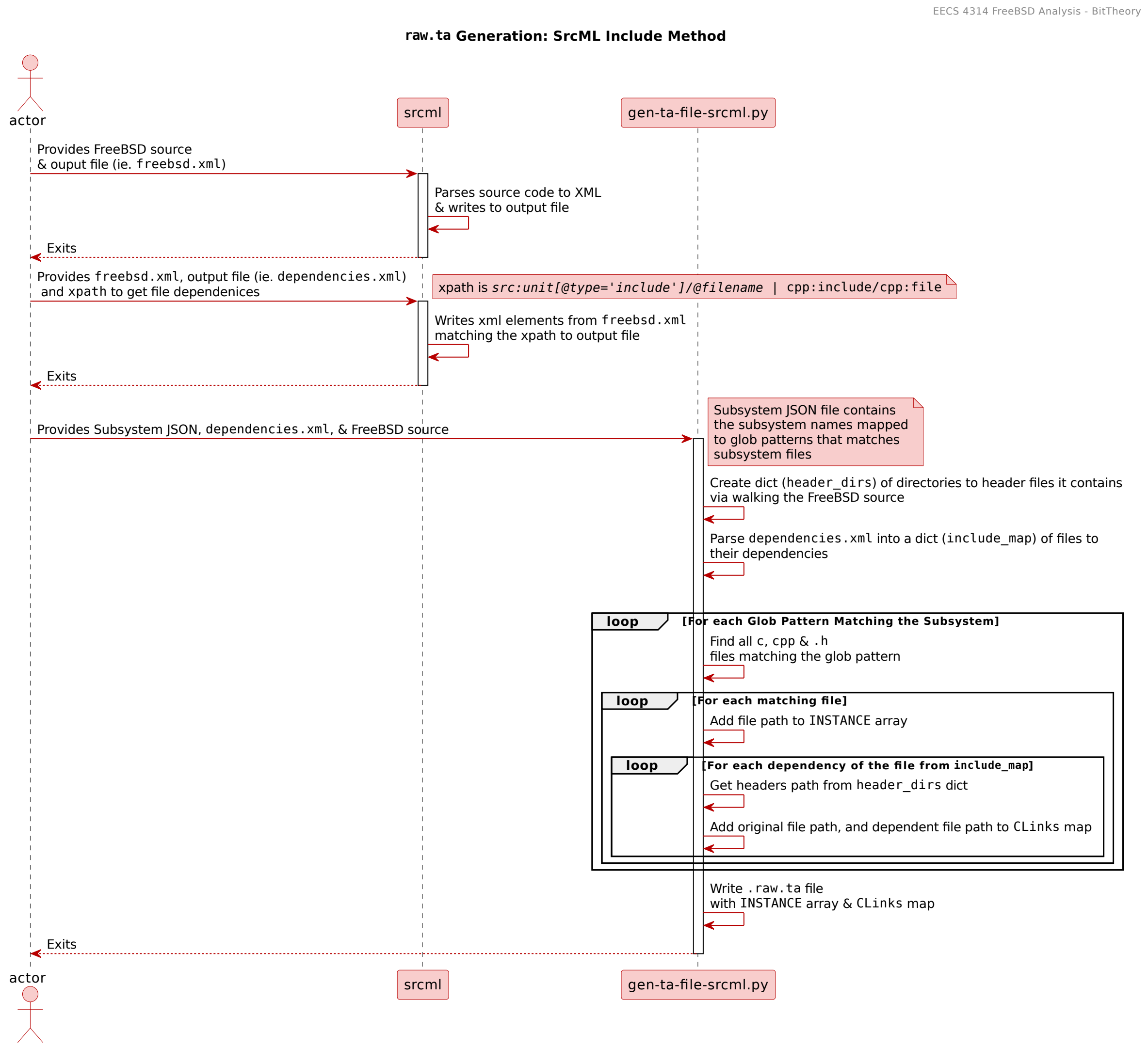

EECS 4314 FreeBSD Analysis - BitTheory
raw.ta Generation: SrcML Include Method
actor
srcml
gen-ta-file-srcml.py
Provides FreeBSD source & ouput file (ie. freebsd.xml)
Parses source code to XML & writes to output file
Exits
Provides freebsd.xml, output file (ie. dependencies.xml) and xpath to get file dependencies
xpath is src:unit[@type='include']/@filename | cpp:include/cpp:file
Writes xml elements from freebsd.xml matching the xpath to output file
Exits
Provides Subsystem JSON, dependencies.xml, & FreeBSD source
Subsystem JSON file contains the subsystem names mapped to glob patterns that matches subsystem files
Create dict (header_dirs) of directories to header files it contains via walking the FreeBSD source
Parse dependencies.xml into a dict (include_map) of files to their dependencies
loop
[For each Glob Pattern Matching the Subsystem]
Find all c, cpp & .h files matching the glob pattern
loop
[For each matching file]
Add file path to INSTANCE array
loop
[For each dependency of the file from include_map]
Get headers path from header_dirs dict
Add original file path, and dependent file path to CLinks map
Write .raw.ta file with INSTANCE array & CLinks map
Exits
actor
srcml
gen-ta-file-srcml.py

Figure 5: Raw TA file generation using custom SrcML script.[2][5]

3 Quantitative Analysis

3.1 Process of Quantitative Analysis

We used a python script `gen-statistics.py` in order to generate our results for this analysis. This script would take 3 `ls.ta` extraction files for each tool that we implemented in this project, then read every line from each extraction file looking for `cLinks` dependencies. The script then adds the dependencies to a respective set for each tool using the “from file” and “to file” as a key. Then we use python set theory functions to find useful statistics such as: the size of the intersection between each two tool dependency sets, the size of the intersection of all dependency sets, the size of the exclusive dependency set for each tool, and the size of the union of all dependency sets. Finally this information is logged into a txt file for future reference. Finally we use this file to generate the following visualizations.

3.2 Direct Comparison

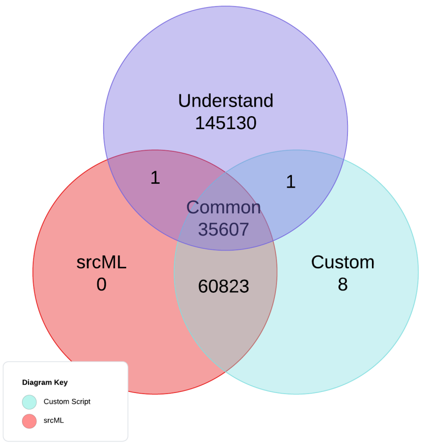


Figure 6: The exclusive and common dependencies of the 3 tools we used for extraction, broken into 7 sets.

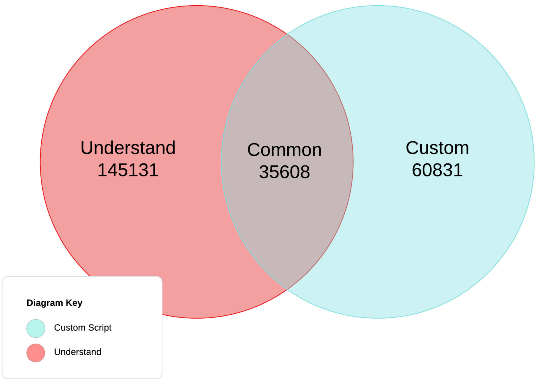


Figure 7: Direct comparison between the number of dependencies found from the Understand extraction tool and the Custom “include” extraction

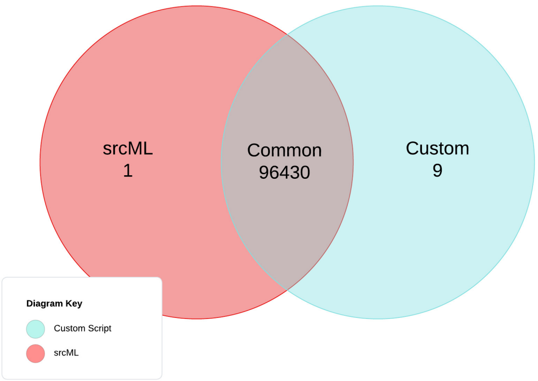


Figure 8: Direct comparison between the number of dependencies found from the SrcML extraction tool and the Custom “include” extraction

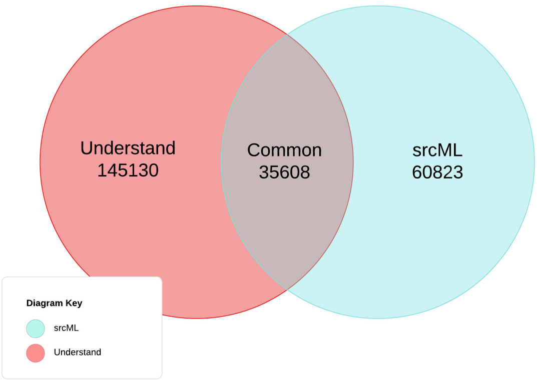


Figure 9: Direct comparison between the number of dependencies found from the Understand extraction tool and the scrML extraction tool

3.3 Quantitative Analysis Table

Number of dependencies		
Extraction Tool	Total number of dependencies	Number of unique dependencies
Understand	180,739	145,130
SrcML	96,431	1
Custom Script	96,439	9
$Understand \cap SrcML$	35,608	1
$Understand \cap CustomScript$	35,608	1
$SrcML \cap CustomScript$	96,430	60823
All three (union)	241,570	N/A
All three (common)	35,607	N/A

The above table above shows the number of dependencies extracted from each extraction tool. Understand has the most dependencies out of the three extraction tools, with 180,739 dependencies total. If we divide the number of unique dependencies of Understand by its total number of dependencies, we get $(145130 \div 180739) \approx 80\%$. From this we can conclude that 80% of Understand's extracted dependencies are unique to that tool.

One observation was that Understand was able to extract far more unique dependencies than SrcML and our custom script put together. Combined, $SrcML \cap CustomScript$ extracted 60,823 unique dependencies, but Understand managed to extract 145,130 dependencies, which is more than double. The SrcML by itself only captured one unique dependency, and our custom script only captured 9 unique dependencies, compared to the large number that Understand gathered.

One thing to note is that SrcML and our custom script have similar statistics. SrcML extracted 96,431 dependencies in total, while our custom script extracted 96,439. The reason why SrcML and the custom script have similar statistics is because they both focus on parsing include statements, in contrast to Understand which prioritizes extracting functions calling other functions.

4 Qualitative Comparison

4.1 Process of Qualitative Comparison

This is the process of our qualitative comparison. After obtaining the extracted dependencies from each technique, we applied the Stratified Sampling to get samples. Here, the population is the union set of all the techniques minus the common dependencies, which is equivalent to 205,963. By utilizing a calculator from Creative Research Systems [4], we got 383 as a needed sample size. We divided all the dependencies into 7 stratas and fetched the appropriate number of dependencies from them. For example, the number of dependencies extracted only by our Custom method is 8, and the proportion of the stratum is 8 divided by 205,963, which is approximately 0.000039. That is, the sample size needed for the particular stratum is 383 multiplied by 0.000039, which is approximately 0. After completing all the calculations, we obtained 270 samples from dependencies which only Understand extracted and 113 samples from the common dependencies, extracted by both Custom and SrcML methods. Moreover, we fetched a single random dependency from the other stratas to introduce more variety of reasons for the existence of dependencies in each strata. After the Stratified Sampling, we observed each dependency to find any differences, examined its file code, and updated our extractors when necessary. We repeated these steps until we classified all the dependencies.

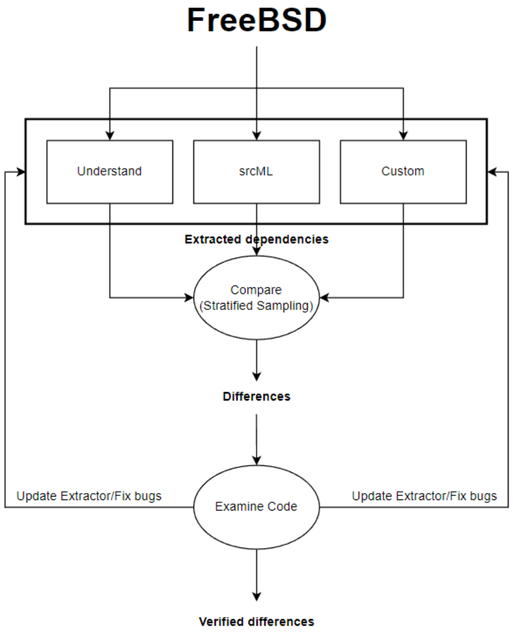


Figure 10: Qualitative Comparison's Process Diagram

4.2 Results of Qualitative Comparison

4.2.1 Comparison Between Understand and Custom

Understand-Only Dependencies

Example : `freebsd/sys/fs/nfsclient/nfs_clsubs.c`
→ `freebsd/sys/fs/nfsclient/nfs.h`

This dependency was extracted because this `nfs_clsubs.c` file includes `nfs.h` as an imported file. Only Understand was able to fetch this dependency as our Custom script is built to fetch the first match, which results in capturing `nfs.h` under `sontib/tcpdump` folder.

Custom-Only Dependencies

Example : `freebsd/sys/dev/pms/RefTisa/sallsdk/spc/sahw.c`
→ `freebsd/sys/dev/pms/RefTisa/sallsdk/hda/64k/iopimg.h`

The dependency is found here because ioparray within `iopimg.h` is included and utilized in `sahw.c`. However, Understand did not include this dependency as the used location is within a comment line but within the actual run code. Therefore, it was assumed by Understand that it is an unnecessary dependency to extract.

Common Dependencies Between Understand And Custom

Example : `freebsd/sys/dev/pms/freebsd/driver/ini/src/agtiapi.c`
→ `freebsd/sys/cam/cam_sim.h`

`cam_sim.h` is included and utilized in `agtiapi.c` for creating structs. For this dependency, Understand simply extracted it, and the Custom script also extracted it without any issues as `cam_sim.h` with this particular file name can be found only once in the system.

4.2.2 Comparison Between Understand and SrcML

Understand-Only Dependencies

Example : freebsd/sys/contrib/libnv/cnvlist.c
→ freebsd/sys/sys/nv.h

For this Understand-only dependency, cnvlist.c uses data structures and functions defined in nv.h. nv.h is a header file that defines the "name-value" (NV) pair data structure. The cnvlist.c source file in libnv provides functions for creating, manipulating, and storing NV pairs.

Example : freebsd/sys/contrib/dev/acpica/compiler/aslmaputils.c
→ freebsd/sys/contrib/dev/acpica/include/acobject.h

Although not a direct dependency understand was able to still catch indirect dependencies between these files. acobject.h is a header file that defines various ACPI object data structures and functions used throughout the ACPI subsystem. The aslmaputils.c source file provides utilities for mapping ACPI names to objects and vice versa, as well as for searching and validating ACPI namepaths.

SrcML-Only Dependencies

Example : freebsd/sys/dev/cfe/cfe_resource.c
→ freebsd/contrib/sendmail/include/sm/cdefs.h

The reason for the apparent dependency between these two files is that cfe_resource.c includes a header file that is defined in cdefs.h.

Common Dependencies Between Understand And SrcML

Example : freebsd/sys/dev/smartpqi/smartpqi_event.c
→ freebsd/sys/dev/smartpqi/smartpqi_includes.h

The smartpqi_event.c source file in the smartpqi driver for FreeBSD includes the smartpqi_includes.h header file because it contains various data structures and function declarations declared in smartpqi_includes.h. smartpqi_includes.h provides definitions for the data structures used to represent various events that can occur in the driver.

4.2.3 Comparison Between Custom and SrcML

Custom-Only Dependencies

Example : freebsd/sys/dev/pms/RefTisa/sallsdk/spc/sahw.c
→ freebsd/sys/dev/pms/RefTisa/sallsdk/hda/64k/iopimg.h

It is the same dependency as the one, brought up earlier for the comparison between Understand and Custom. As well as Understand, SrcML also ignored the dependency as iopimg.h was included within commented-out lines.

SrcML-Only Dependencies

Example : artpqi/smartpqi_event.c
→ freebsd/sys/dev/smartpqi/smartpqi_includes.h

For this SrcML-only dependency, this was included only in a SrcML extracted list, and the reason for this was due to the missing white space in include reference.

Common Dependencies Between Custom And SrcML

Example : freebsd/sys/dev/beri/virtio/network/if_vtbe.c
→ freebsd/sys/amd64/include/cpu.h

For this dependency, we were not able to find a clear rationale. Because both Custom and SrcML contain this sample, we first assumed that if_vtbe.c includes cpu.h. However, after examination, we found out that the if_vtbe.c imports cpu.h, but not in the path shown in the list of extracted dependencies. Instead, the imported cpu.h was under a machine folder. Because the file location is still under the machine folder within a XML file we obtained, we assume that the SrcML script we created might need to be modified to show the correct path.

4.2.4 Performance Metrics

For the Performance metrics, we utilized recall and precision for our Custom and SrcML by including Understand as Oracle. Here, we have two formulas to calculate recall and precision. As well as these formulas, we included a diagram which shows which dependency sets we examined to obtain solutions.

$$Recall = \frac{TruePositive}{(TruePositive + FalseNegative)}$$

$$Precision = \frac{TruePositive}{(TruePositive + FalsePositive)}$$

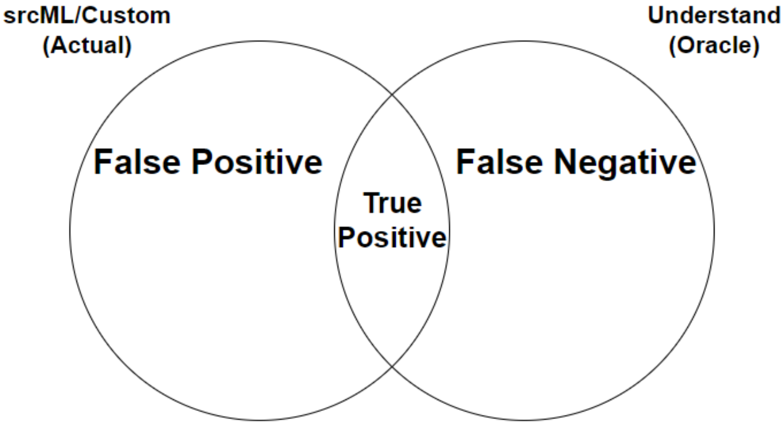


Figure 11: Classification Diagram

For both Custom and SrcML, we obtained the same results shown below :
For recall, we obtained 66 True Positive dependencies and 270 False Negative dependencies.

$$\begin{aligned} Recall &= \frac{66}{(66 + 270)} \\ &= 0.196428571\ldots \\ &\approx 20\% \end{aligned}$$

For precision, we obtained the same number of True Positive dependencies and 133 False Positive dependencies.

$$\begin{aligned} Precision &= \frac{0 + 66}{(66 + 113)} \\ &= 0.3687150\ldots \\ &\approx 37\% \end{aligned}$$

5 Limitations & Recommendations

Code analysis tools, both static and dynamic, play an essential role in software development. They help identify issues in code, improve code quality, and increase the overall reliability of software. However, like all tools, they have limitations. We will now discuss the limitations of Understand, SrcML, and our custom analysis. We will also recommend how we could have improved the process if we had more time to explore other dependency extraction techniques. Code analysis tools are essential in software development but have limitations since dependency extraction is non-trivial. Understanding the limitations of these tools is crucial to use them to extract dependencies effectively.

5.1 Limitations of SrcML and Custom

The SrcML and custom methods have similar limitations as they extract "include" directive dependencies. For our custom method, the regular expression used to find the include directives in source files is simple and may only capture some possible include statement formats. For example, `#include"test.h"` will not be matched as it is a missing edge case for the space after the include directive. The script also assumes that all files with a "`.c`," "`.cpp`," or "`.h`" extension are C or C++ source files, which may not be accurate in all cases and could result in incorrect or missing dependencies in other types of projects. One of our most significant limitations was extracting the correct include file path. The SrcML and custom method exit early after finding the first match of a header file. This limitation introduces false positives in the case multiple directories have the same header file.

Furthermore, the scripts do not handle circular dependencies between files, and it only looks for header files in directories containing C or C++ source files. Additionally, these include methods do not account for conditional compilation directives, such as `#ifdef` and `#ifndef`, which could result in missing or incorrect dependencies. Lastly, these include methods do not handle call graph dependencies and other types of dependencies, which is evident in our reduced extracted dependencies compared to Understand.

The SrcML and custom method are similar, as seen by our performance metrics and quantitative analysis. The include method is generally less effective than how Understand processes dependencies, as Understand had a significantly more significant number of mutually-exclusive dependencies that the include methods could not find due to the limitations.

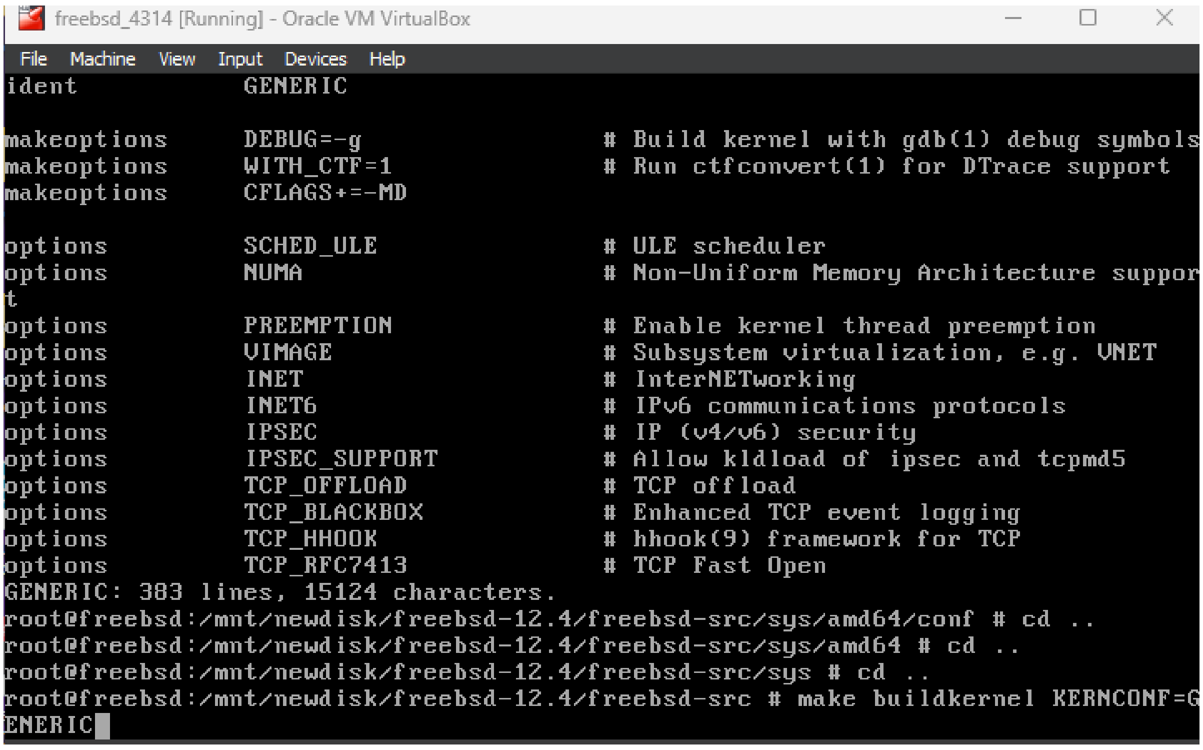
5.2 Understand

The Understand static code analysis tool also has limitations despite being a robust commercial tool. One of the primary limitations is a high false-positive rate, and understanding sometimes identifies dependencies that are not in the source code, leading to a high false-positive rate. Additionally, while Understand supports a wide range of programming languages, it may only support some features, making it challenging to analyze code that uses those features.

Furthermore, Understand cannot analyze dynamic code since it is a static code analysis tool. This means it cannot analyze dynamic code or code generated at runtime. Lastly, Understand can be resource-intensive and time-consuming when analyzing large codebases, making it challenging to use on large projects and may impact its effectiveness.

5.3 Difficulty in navigating CLI for LLVM-IR extractions

The process of extracting dependecies through LLVM-IR from the FreeBSD kernel source code can be a daunting task, particularly for those who are not well-versed in command line interface (CLI) navigation. One of the primary challenges is the fact that the FreeBSD kernel source code can only be built within the FreeBSD environment, which adds an additional layer of complexity to an already intricate process. Moreover, working with the CLI requires scripting expertise, a skill that may not have been taught or had the opportunity to acquire.



```
ident           GENERIC

makeoptions     DEBUG=-g                # Build kernel with gdb(1) debug symbols
makeoptions     WITH_CTF=1              # Run ctfconvert(1) for DTrace support
makeoptions     CFLAGS+=-MD

options         SCHED_ULE               # ULE scheduler
options         NUMA                    # Non-Uniform Memory Architecture suppor
t
options         PREEMPTION              # Enable kernel thread preemption
options         VIMAGE                  # Subsystem virtualization, e.g. VNET
options         INET                    # InterNETworking
options         INET6                   # IPv6 communications protocols
options         IPSEC                   # IP (v4/v6) security
options         IPSEC_SUPPORT           # Allow kldload of ipsec and tcpmd5
options         TCP_OFFLOAD             # TCP offload
options         TCP_BLACKBOX            # Enhanced TCP event logging
options         TCP_HHOOK               # hhook(9) framework for TCP
options         TCP_RFC7413             # TCP Fast Open
GENERIC: 383 lines, 15124 characters.
root@freebsd:/mnt/newdisk/freebsd-12.4/freebsd-src/sys/amd64/conf # cd ..
root@freebsd:/mnt/newdisk/freebsd-12.4/freebsd-src/sys/amd64 # cd ..
root@freebsd:/mnt/newdisk/freebsd-12.4/freebsd-src/sys # cd ..
root@freebsd:/mnt/newdisk/freebsd-12.4/freebsd-src # make buildkernel KERNCONF=G
ENERIC
```

Figure 12: The process of perform LLVM dependency extractions

The expectation to perform LLVM-IR extractions on the FreeBSD kernel source code, despite the lack of formal training in command line scripting, left us feeling overwhelmed and ill-equipped. This difficulty in navigating the CLI in FreeBSD for dependency extractions through LLVM highlights the need for importance of teaching students complex scripting task in CLI interface.

5.4 Recommendations

If we had more time, one way to improve these limitations would be to explore other dependency extraction techniques, such as compiling the source code and extracting linked library dependencies instead of using two methods that use include directives. Alternatively, we can expand using SrcML and create a custom parser that generates a call graph based on XPath queries on the XML function units.

6 Conclusion

To conclude, based on the results and observations presented, it is clear that the choice of dependency extraction tool can have a significant impact on the number and types of dependencies identified in a codebase. Understand appears to be the most thorough and comprehensive tool, identifying over 180,000 dependencies, 80% of which were exclusive to that tool. However, it is worth noting that this may lead to more false positives or irrelevant dependencies being identified. SrcML and the custom script both focused primarily on include files and identified around 96,000 dependencies, with SrcML having no exclusive dependencies and the custom script having only 8. The similarity in their statistics suggests that they may have similar strengths and limitations.

The observation that Understand analyses dependencies based on functions calling other functions while SrcML and the custom script prioritize "include" files only suggests that the former may be more effective at identifying dynamic or runtime dependencies, while the latter may be more appropriate for identifying static or compile-time dependencies. However, it is important to note that both types of dependencies can be important in understanding the architecture of a codebase.

In terms of choosing a methodology for dependency extraction in FreeBSD, the specific goals and characteristics of the analysis should be taken into account. For example, if the goal is to identify dynamic dependencies, Understand may be the most appropriate tool. If the focus is on static dependencies based on include files, SrcML or the custom script may be more effective. However, it is worth noting that no single tool or methodology may be sufficient on its own, and a combination of techniques may be needed for a more comprehensive understanding of the architecture.

Overall, the process of extracting and analyzing architectural dependencies in FreeBSD can be complex and may require careful consideration of the specific needs of the analysis. The tools and methodologies used should be chosen based on the specific goals of the analysis and the characteristics of the codebase being analyzed. By taking these factors into account and combining multiple techniques as necessary, a more complete understanding of the architecture of FreeBSD can be achieved.

7 Lessons Learned

The importance of choosing the right tool: The choice of tool for dependency extraction can have a significant impact on the results and effectiveness of the analysis. It is important to carefully consider the strengths and limitations of each tool and select the one that is most appropriate for the specific goals and characteristics of the analysis.

The importance of understanding the codebase: A thorough understanding of the codebase is crucial for effective dependency extraction and analysis. Without a good understanding of the code structure, it can be difficult to interpret the results and identify relevant dependencies.

The value of combining multiple techniques: While each tool or methodology may have its own strengths and limitations, combining multiple techniques can provide a more comprehensive understanding of the architecture. For example, using both Understand and a custom script may help to identify both dynamic and static dependencies.

The risk of false positives and irrelevant dependencies: Some tools may identify a large number of dependencies, many of which may not be relevant to the specific analysis. It is important to carefully review and validate the results to avoid being misled by false positives or irrelevant dependencies.

The need for ongoing analysis and refinement: As codebases evolve over time, the architecture and dependencies may change as well. Ongoing analysis and refinement of the dependency extraction process is needed to ensure that the results remain accurate and relevant. This may involve updating the tools and techniques used, as well as revisiting the goals and objectives of the analysis.

8 Data Dictionary

- Stratified Sampling: A sampling technique which divides the whole dependency set into multiple strata based on their number proportion
- SrcML: A light weight tool to transform and analyze source code in XML format
- XPath: A query syntax to perform syntactic analysis on SrcML generated XML
- Oracle: A source of comparison, in this case Understand dependency extraction
- False Positive: Dependency that was incorrectly identified as present
- False Negative: Dependency that was incorrectly identified as not present
- True Positive: Dependency that was correctly identified as present

9 Naming Conventions

- XML: Extensible Markup Language
- IDE: Integrated Development Environment
- CSV: Comma Separated Values

References

[1] Clarke, Arthur C. 2001: A Space Odyssey. New York: Roc, 1968. 297.

[2] "BitTheory A3 source code." README.md, https://github.com/BitTheoryProject/eecs4314-reports/tree/main/a3/src.

[3] "BitTheory A2 Understand steps." README.md, https://github.com/BitTheoryProject/eecs4314-reports/tree/main/a2/src.

[4] "Sample Size Calculator." Sample Size Calculator Confidence Level, Confidence Interval, Sample Size, Population Size, Relevant Population Creative Research Systems, 2012, https://www.surveysystem.com/sscalc.htm.

[5] "SrcML Documentation ." https://www.srcml.org/documentation.html.