FreeBSD Kernel Dependency Extraction Using 3 Different Techniques

EECS 4314: Advanced Software Engineering
Dr. Zhen Ming (Jack) Jiang
Assignment 3 Report
March 24rd, 2023

Group Sudo:
Michael Koiku
Swapnilkumar Parmar
Henry Gu
Amin Mohammadi
Omar BaGunaid
Michelangelo Granato
Hieu (Aaron) Le
Vikramjeet Gill
Jai Ramotar

Abstract

This report investigates the pros and cons of various dependency extraction techniques for FreeBSD, a popular open-source operating system. The report examines three tools: the Understand tool, the Include ".headers" tool, and the Include linker objects tool.

Diagrams and tables are used to demonstrate the design of the new extraction approaches. In addition, the extraction results from different tools are compared quantitatively and qualitatively by using precision and recall, and the comparison process, outcomes and implications are analyzed. The report concludes with lessons learned and potential risks and limitations of the current comparison approach for FreeBSD. The findings show that the choice of tool depends on the specific needs of the FreeBSD project and that a combination of different tools may provide the most comprehensive results for extracting dependencies in FreeBSD.

Introduction

FreeBSD is an open-source operating system with advanced functionality, security, and reliability. It is constructed using a modular architecture, allowing different system components to be created independently and communicate using explicit interfaces. One of the most critical elements of this architecture is its capacity to manage dependencies between system components.

The identification of dependencies between different software system components is achieved by dependency extraction techniques, which are essential in the software development process. These methods allow developers and engineers to comprehend how multiple system parts interact, which is crucial for maintaining and improving software systems. The advantages and disadvantages of three different dependency extraction methods used in FreeBSD are analyzed in this report. In particular, we compare the data extracted through the Understand tool, the program dependencies extracted using include directives, and the include linker objects tool.

The "understand" tool is a software analysis and visualization tool that offers a high-level understanding of the dependencies of a system. It can provide a graphical representation of the system's structure and extract many dependencies, including code-level, database, and configuration dependencies. It is beneficial for determining the system's general architecture and identifying potential issues.

The "include directives" tool is straightforward for extracting header file dependencies. It examines a program's source code to determine which header files should be included before compiling the program. It helps ensure all required header files are available and for spotting potential problems like circular dependencies.

The "include linker objects" tool is a tool that extracts the dependencies of linker objects, which are binary files that contain compiled code and data. It analyzes the object files of a program and identifies the external libraries and object files that the program depends on to link and execute correctly. It benefits to confirm that all necessary libraries and object files are included and for detecting potential issues, such as missing dependencies.

Understand Extraction Technique

Understand is a software tool that is designed to extract dependencies from codebases, including legacy code. Legacy code can be challenging to work with due to its lack of organization and documentation. However, by using tools like Understand, developers can gain insight into how different parts of the codebase are connected and identify any issues that may exist. By improving the overall structure and reducing the risk of encountering bugs or other issues, Understand can help make the code easier to understand and maintain, ultimately saving developers time and effort. Some features of understand include:

1. Inheritance Relationships: Understand analyzes software codebases to find classes and their inheritance relationships, identifying which classes are derived from others and which methods or variables are inherited from a parent class. This tool helps developers identify potential problem areas in the codebase, giving them the opportunity to revise the code for improved structure and ease of maintenance.
2. Import Statements: Import statements are used to get functions and variables from other code files. Understand analyzes these imports to identify which files are being referenced and how they are used with the codebase. It helps the developers to better organize their code.
3. Function Calls and parameters: When one function calls another function, there is a relationship between them. Understand analyzes these function calls to recognize how different parts of the code depend on each other. Parameters passed to each function are also analyzed to understand how they are used and the relationships they create. This helps developers understand how different parts of their code are connected and recognize any potential dependencies that might cause problems.
4. Control Flow Analysis: Understand can analyze the control flow of a program, identifying how different parts of the code depend on each other, such as conditional statements and loops. This helps developers identify areas of the code that may be more complex. By performing control flow analysis, developers can better understand how the code flows and make changes to simplify it.
5. Call Graph Analysis: The call graph analysis provides a clear graphical view of the program, helping to identify which functions and modules are heavily used. This technique creates a graph of the dependencies between these functions. By analyzing the call graph, developers can optimize the performance of their codebase by focusing on frequently used functions and modules.

Understand Extraction Derivation Process

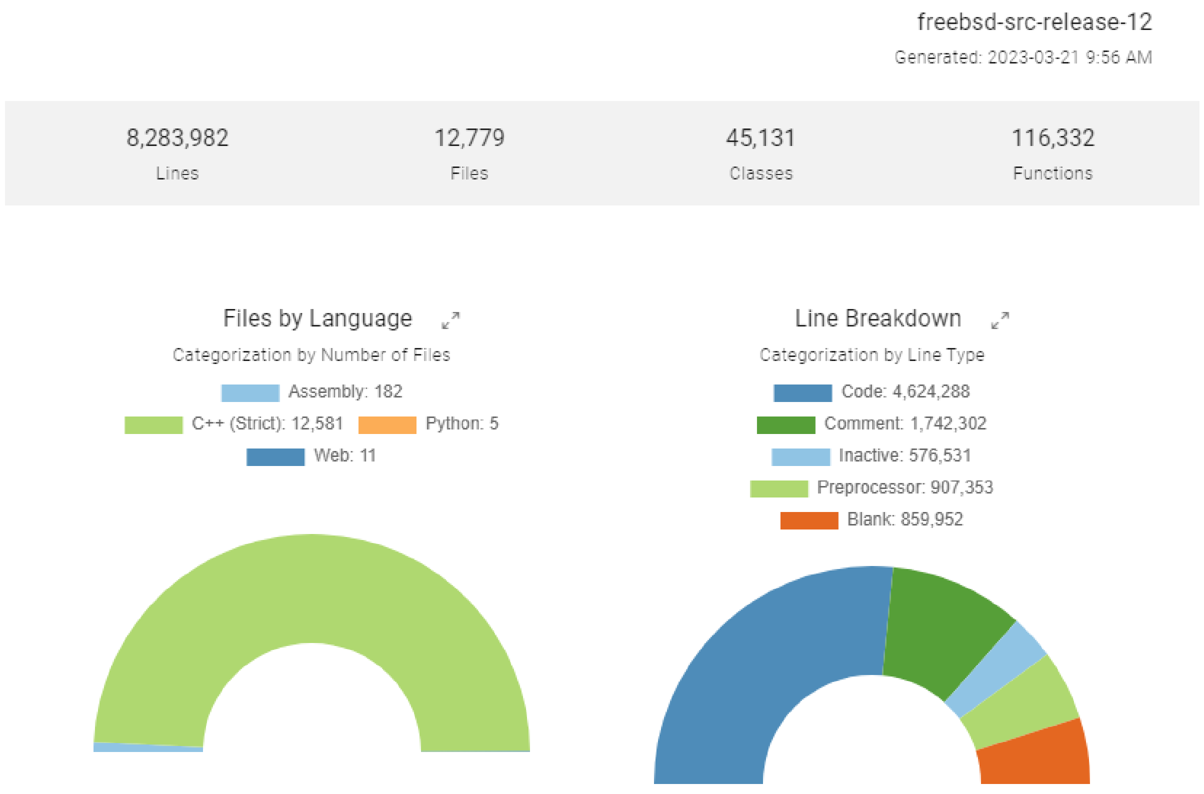
To begin with, in order to analyze the kernel code of FreeBSD release 12.4.0, we need to create a new Understand project. This can be achieved by navigating to the Understand software and pointing it to the root directory of the kernel code, which in this case is the “sys” directory. Once the directory is selected, Understand will automatically build the project by scanning through all the relevant files and directories.

After the project has been built, the next step is to export the dependencies of the project. To do this, we can go to the “Project” menu and select “Export Dependencies”. From here, we can choose the “File Dependencies” option and select “Relative Path Name” to generate a .csv file that contains all the file dependencies of the project.

Once the .csv file has been generated, we can run the transformUnderstand.pl script from A2 to transform the file into a .raw.ta file. This script will read the .csv file and perform the necessary transformations to generate a .raw.ta file, which can then be used for further analysis.

It is worth noting that the .raw.ta file is essentially a text file, and as such can be opened and inspected using any text editor. This can be useful in cases where the formatting of the file needs to be adjusted or modified in some way. For example, our customs tools were developed on Linux and produced dependencies with the path having forward slashes (“/”), while the Understand extraction was done on Windows and produced an output with paths having backward slashes (“\”), we had to use Visual Studio Code as a text editor to perform a regex operation to covert the backward slashes on the Understand output to forward slashes.

Overall, these steps provide a basic framework for analyzing kernel code using Understand. By following these steps, we can generate a .raw.ta file that can be further processed and analyzed using various tools and techniques, ultimately allowing us to gain a deeper understanding of the underlying code and how it functions.


freebsd-src-release-12

Generated: 2023-03-21 9:56 AM

8,283,982	12,779	45,131	116,332
Lines	Files	Classes	Functions

Figure 1: Our Sci Tools Understand Analysis Results

Advantages & Limitations of Understand Extraction Technique

Scitool Understand is a software with a variety of comprehensive built-in features that aid users in learning and analyzing software systems. It is a powerful tool with many perks but like any other software, it also comes with some limitations.

The advantages of this technique include:

1. Excellent code visualization feature that helps generate from code a diagram of files and dependencies in an organized hierarchy. This interactive feature allows users to explore and study the dependencies at different levels and have a better understanding of software
2. User-friendly UI
3. Supports multiple programming languages from C/C++, Java, Assembly, Python, etc
4. Provides reports with in-depth analysis, annotated code and customizable metrics

The limitations of this technique include:

1. It's a commercial tool and therefore requires a license to use. Even though it depends on each individual's purpose and how much they use, the expensive subscription cost of 100-120 USD/ month may be a barrier for users with a limited budget.
2. Despite its user-friendly UI, the software itself has a learning curve and requires users to spend time to be familiar with

Include Header Files Extraction Technique

This technique refers to a process in software development where a program's dependencies on header files are analyzed and extracted. Header files typically contain declarations for functions, variables, and other elements that are used in a program's source code. When a program is compiled, the compiler needs to know about these declarations in order to generate object code that can be linked with other parts of the program. The process of extracting header file dependencies involves analyzing a program's source code to identify all of the header files that are included by the program. Once the header file dependencies have been identified, they can be used to ensure that the correct header files are included in the program's source code. This can help to avoid errors that can occur when a header file is missing or when conflicting declarations are present in multiple header files.

Include Header Files Extraction Derivation Process

We implemented our own tool for this which works as follows:

1. To begin, we first find all ***.c*** and ***.h*** files.
2. Once we have identified these files, we then proceed to find and parse all ***#include*** statements in each file. This will allow us to determine which header files are needed by the program.
3. Next, we attempt to search for the relevant ***.h*** files. We start by looking in the same directory as the source files. If the necessary header files are not found there, we use a set of heuristic rules to determine additional locations to search. These rules are as follows, suppose we are looking for the file “a/b/c.h”, we will search the following files:
    - a/b/c.h
    - include/a/b/c.h
    - a/b/include/c.h
    - If none of the above are found, then go up to the parent directory and repeat the above
    - If you keep going up and reach the root of the code, then check /usr/include
    - If still not found, then give up
4. Finally, once we have identified all of the necessary header files, we can generate a ***.ta*** file using ***#include*** references. This file can be used by the compiler to ensure that the correct header files are included in the program's source code, thus helping to avoid errors that can occur when a header file is missing or when conflicting declarations are present in multiple header files.

Advantages & Limitations of Include Header Files Extraction Technique

While using the include “.header” files technique, we quickly noticed it came with more disadvantages than advantages which will be elaborated upon. An advantage of using this technique for dependency extraction is that it is easy to use. In the sense that, once we have our header files, it is just a matter of finding which files include them. This can be done by reading files for the “#include sample_header_file.” statements. A custom tool we developed performed this process.

While this technique is easy to use, the number of disadvantages that come with it outnumbers its advantage. For example:

1. Naming conflicts: During the extraction process, we discovered that we had multiple header files of the same name in different directories. This made it difficult to determine which exact header was being used by a file including it. To overcome this, we had to develop some heuristic rules to determine the appropriate header file.
2. Unused Includes: During the development process of the system (FreeBSD), developers might have included header files that are not needed as a formality. These header files are still counted by our tool as dependencies, even though they are not used. Furthermore, Some headers might be #ifdef'd out, but our tool will not interpret preprocessor commands.
3. Not header files exist: During the extraction process, we also discovered that some header files are generated automatically during the build process and are not available in th freeBSD source and therefore could not be identified by our tool.
4. Not header files were found: Our custom tool was not able to detect all header files. Even if a header file was available, it might have been somewhere so obscure that it could not be found by our tool.
5. Limited dependency scope: This technique does not discover and extract all the dependencies. It only extracts dependencies between .c or .h files to other .h files.

Linker Objects Extraction Technique

This technique involves analyzing the dependencies between object files to identify which files depend on which other files. This information can be used to optimize the build process by only recompiling and relinking the necessary files when changes are made to the code.

The first step of this technique is to generate a list of all the object files. Next, we link all the object files together to create a single executable file. During this linking process, we generate a list of the symbols called the symbol table (functions, variables, etc.) that are referenced by each object file. Once we have obtained the symbol table we can extract the dependencies by analyzing the list of symbols for each object file and identifying which symbols are defined in other object files.

Linker Objects Extraction Derivation Process

We implemented linker object extraction by implementing a program to emulate the steps a linker would take and record the results during the process. We started by compiling the FreeBSD kernel to get the object files we want to analyze. After this, we search for all ".o" object files. All these object files are in the "Executable and Linkable Format" (ELF) format.

We implemented a parser for ELF files and used it to extract the list of sections in the file. From here, we can find the ".symtab" section, which stores the list of symbols used in the object file. This list contains all symbols referenced by the object file, including symbols that are used but not defined and symbols that are defined in the file. The basic linking process is finding the undefined symbols and matching them with a definition in another file. Symbols in the symbol table have various flags indicating the type, properties, and section of a symbol. Our linker extracts the symbol names and flags and handles the various categories depending on how they need to be handled:

- If a symbol is "local", then it is not visible to any other file, we can ignore these symbols as they have no influence whatsoever outside of the file they belong to.
- If a symbol is "hidden", then it is ignored for the same reason as local symbols.
- Special symbols, such as the "file" symbol, which is simply a symbol containing the name of the source file are also ignored.

If the symbol has an associated section, then it is a defined symbol. We store these symbols in a map from name to list of symbol for the linking step. We store a list of possible symbols for name as there are cases where we can have multiple definitions of a symbol. We also store the associated file alongside the symbol to ensure we are able to generate the results properly at the end.

If the defined symbol is weak, then the linker will link to any of the definitions of the symbol. As these symbols are essentially interchangeable, we add the symbol to the corresponding list in the map. However, we add it if and only if the list contains other weak symbols.
If the defined symbol is strong, then add the symbol to the list of in the map. If the list contains any weak symbols, remove them all first. If there are multiple strong symbols, then this would be a linker error with a traditional linker. However, there are some cases where we do encounter this, as parts of the kernel define the same symbol multiple times but only make one available depending on specific behaviour (such as 32-bit vs 64-bit). Unfortunately, since we do not parse the build files (due to the complexity involved and limited time for the project), we assume that they are interchangeable (that is, there is a scenario when the actual linked symbol is each of the symbols), we can add the symbol and store it alongside the other strong symbols.

Finally, we can handle the linking step. We process all the undefined symbols, for each undefined symbol, we look up the list in the map of defined symbols using the symbol name and retrieve the map of definitions and their corresponding files. We use this list to generate a link between the source file and all the definition files for each entry in the list.

At this point, we have the links between .o object files, and we could generate the .ta file using these links. However, this would give inconsistent results with our other techniques (Understand and #include), which report links between .c and .h files. To overcome this, we parse the DWARF debugging information in the file. The debugging information contains the path to the source file.

Our DWARF parser was implemented as follows:

- If there exists a ".debug_str" section, treat this as a contiguous array of null-terminated strings, and read the second and third strings. The path to the source file is the third string concatenated with the second string.

- Otherwise, if there exists a ".debug_info" section, read the null-terminated string 0x20 bytes past the start of the section. This string is the path to the source file
- Otherwise, if there exists a ".debug_line" section, read the null-terminated string at 0x1c and the null-terminated string immediately after the first. The path to the source file is these two strings concatenated in order.
- Otherwise, the file likely has no "source file". This occurs when some files are created by *objcopy*ing data into ELF files. Since these are not generated by a compiler, no debugging data exists.

We are aware this method of testing for sections and blindly jumping to various offsets and reading strings blindly in the object file and assuming that we landed on the correct path is terrible and extremely unreliable. However, the results were very reliable but only for object files built by the FreeBSD toolchain. Unsurprisingly, this does not work for object files built with any other method. Due to the complexity of DWARF and the limited time available, this was the best we could do.

Finally, we could put this all together and generate a .ta file that has links between source files using the data collected in previous steps.

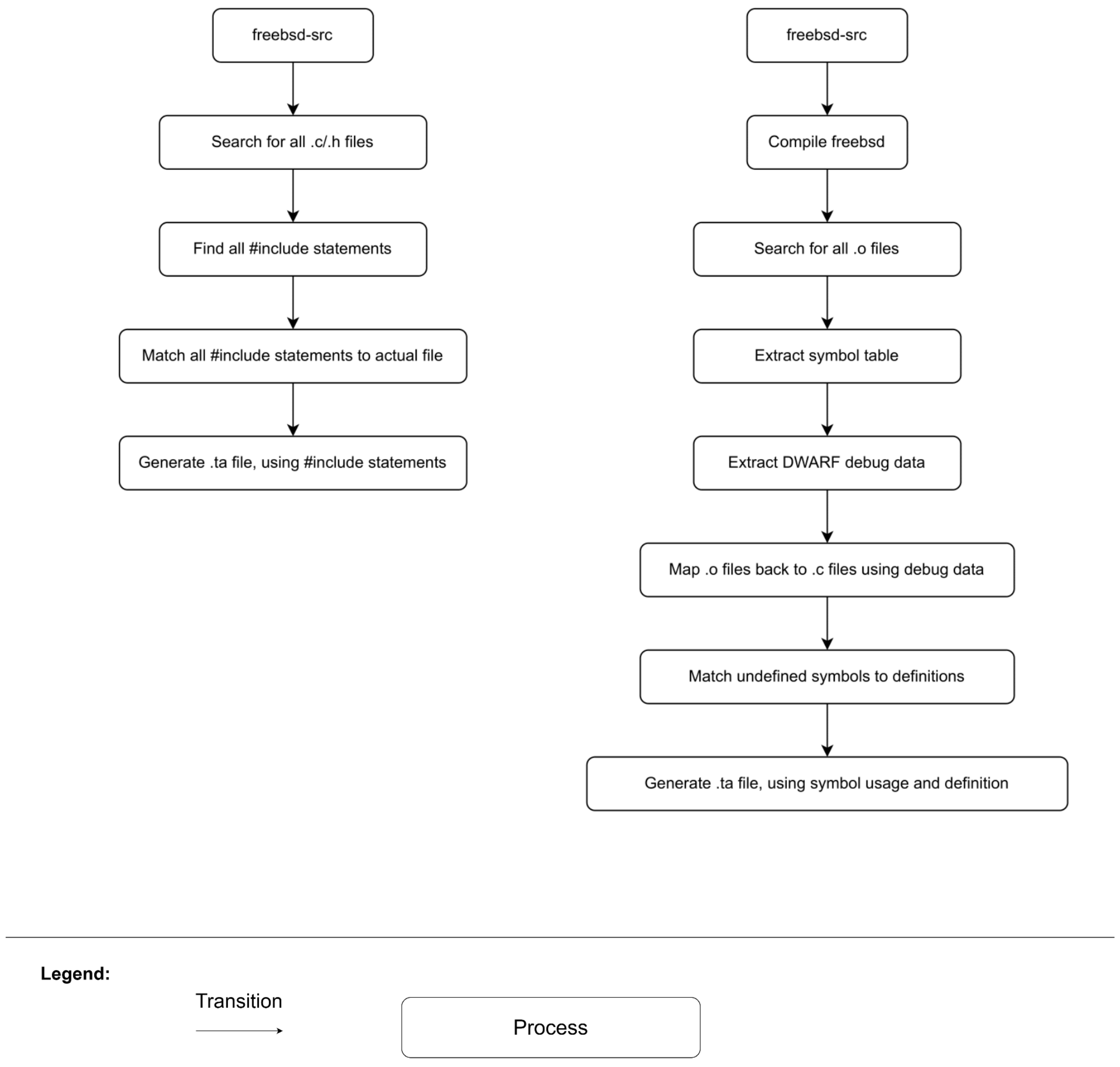
Advantages & Limitations of Linker Objects Extraction Technique

The advantages of this technique include:

1. Combines all the object files and system libraries to create a single executable program along with the dependency graph
2. Analyzes the binary object code rather than the source code meaning more accurate results (due to the preprocessor removing comments, expanding includes, macros, ect to best simulate runtime behaviour)
3. Able to find assembly files (.S) that implement low-level features, that are not found by understand/include method

The limitations of this technique are:

1. Linker objects technique cannot process incorrect code (ie code with syntax errors or code that is uncompilable)
2. Dependencies produced as a result of the linker objects technique do not match the expectations of developers used to viewing the source code unpreprocessed (as they are used to viewing the dependencies unexpanded)
3. Cannot produce detailed information below the file level



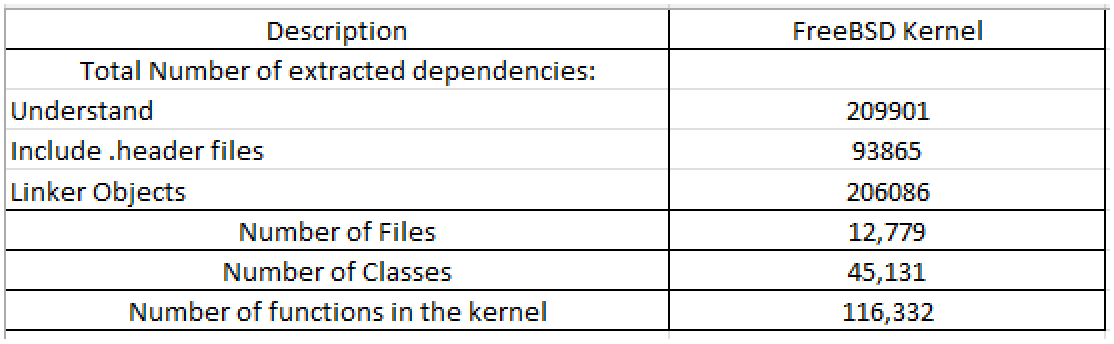
freebsd-src
Search for all .c/.h files
Find all #include statements
Match all #include statements to actual file
Generate .ta file, using #include statements
freebsd-src
Compile freebsd
Search for all .o files
Extract symbol table
Extract DWARF debug data
Map .o files back to .c files using debug data
Match undefined symbols to definitions
Generate .ta file, using symbol usage and definition
Legend:
Transition
Process

Figure 3: A diagram showing the extraction process for both the include header file and linker objects dependency extraction techniques

Quantitative Analysis

We will now perform a quantitative analysis on the kernel, to achieve this we wrote custom tools to generate statistics that will used for the analysis.

Prior to beginning the analysis, we expected Understand to produce the most dependencies as it performed a thorough analysis of the system for dependency extraction, followed by linker objects since new objects will be produced when the source is compiled, and then the include header files technique as we are just analyzing source code. Our expectations were met as Understand produced 209,901 extracted dependencies, the include header files technique produced 93,865 extracted dependencies, and the linker objects technique produced 206,086 extracted dependencies.



Description	FreeBSD Kernel
Total Number of extracted dependencies:	
Understand	209901
Include .header files	93865
Linker Objects	206086
Number of Files	12,779
Number of Classes	45,131
Number of functions in the kernel	116,332

Figure 2: The statistics extracted using our tools

As for the commonalities of the extracted dependencies between the 3 techniques, we observed that Understand and the Include header files technique shared 66,406 extracted dependencies, while Understand and the Linker objects technique shared 98,863 extracted dependencies, and Include header files technique and linker objects technique having no shared extracted dependencies.

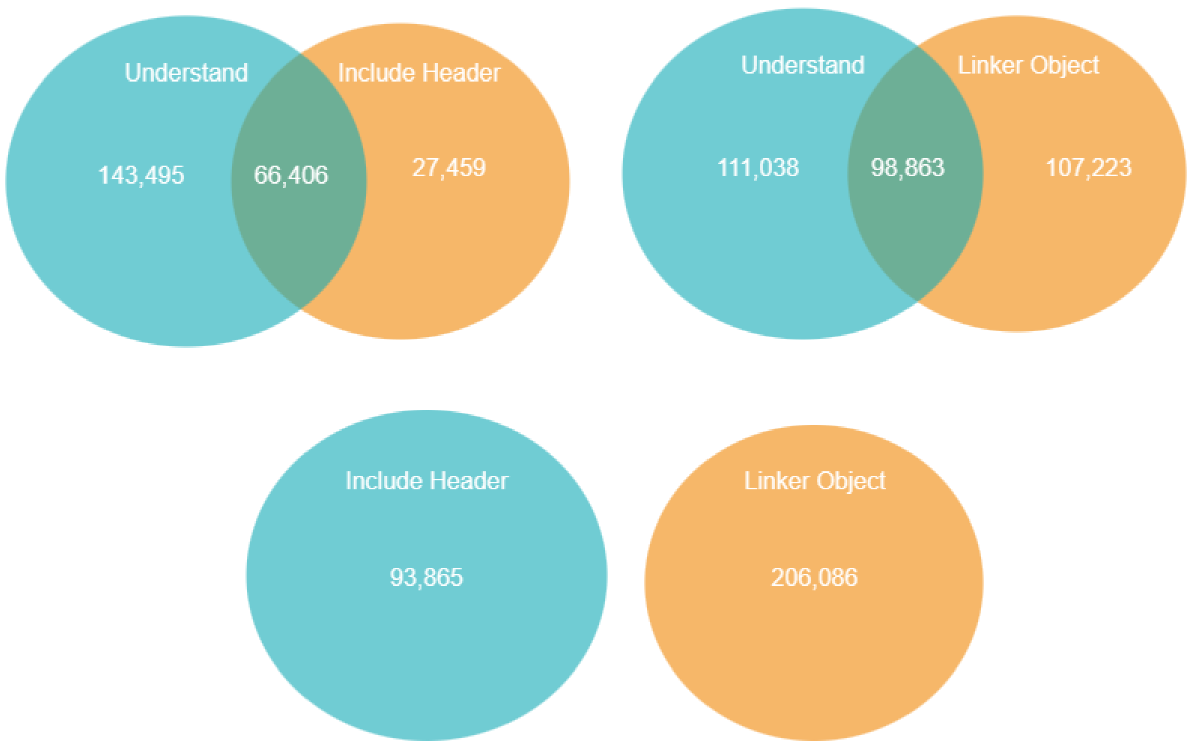


Figure 4: A Venn diagram showing the unique and shared extracted dependencies between the three dependency extraction techniques

Qualitative Analysis

After the quantitative analysis, we then performed qualitative analysis on the extraction. We leveraged statistical inference techniques to sample a portion of the extracted dependencies.

Understand and Include Header files Analysis

Total population = 143,495 + 66,406 + 27,459 = 237,360
Sample size needed: 384

Dependencies	Equivalent Sample Size	Percentage of Population
Understand Unique	(143,495 / 237,360) * 384 = 232 cases	(232 / 384) * 100 = 60.4 %
Include Header Files Unique	(27,459 / 237,360) * 384 = 44 cases	(44 / 384) * 100 = 11.5 %
Shared	(66,406 / 237,360) * 384 = 107 cases	(107 / 384) * 100 = 27.9 %

Understand and Linker Objects Analysis

Total population: 111,038 + 98,863 + 107,223
Sample size needed: 384

Dependencies	Equivalent Sample Size	Percentage of Population
Understand Unique	(111,038 / 237,360) * 384 = 180 cases	(180 / 384) * 100 = 46.9
Linker Objects Unique	(107,223 / 237,360) * 384 = 173 cases	(173 / 384) * 100 = 45.1
Shared	(66,406 / 237,360) * 384 = 107 cases	(107 / 384) * 100 = 27.9

We also focused on comparing the scope and level of detail provided by three dependency extraction tools. The first tool, Understand, is capable of extracting various types of dependencies, including code-level, database, and configuration dependencies. This makes it a valuable tool for understanding the overall architecture of a project and identifying any potential issues. The second tool we examined was the Include Directives tool, which is specifically designed to extract header file dependencies. This tool is particularly useful for ensuring that all necessary header files are included in a project and for detecting any potential issues related to missing or incorrect header files. Finally, we looked at the Include Linker Objects tool, which is designed to extract the dependencies of linker objects - binary files that contain compiled code and data. This tool is essential for ensuring that all necessary libraries and object files are included in a project and for detecting any potential issues related to missing or incorrect linker objects. Overall, these three tools provide a comprehensive suite of options for examining and managing dependencies in software projects, and can greatly improve the efficiency and effectiveness of development efforts.

Precision and Recall

To perform our precision and recall analysis, we chose our Understand extracted dependency data minus the entries containing "C:/Program Files/SciTools/" as our source of truth which is 209,401 dependencies. We then compared the other two extraction methods against this using the formulas below:

Precision = True Positive / True Positive + False Positive
Recall = True Positive / True Positive + False Negative

The following are the results:

Include Header files Technique

From the Quantitative Analysis, we determined that 66,406 extracted dependencies were shared between Understand and the Include Header files technique, this is our True Positive. Our True Positive + False Positive is the total result from this extraction method which is 93,865. The results of our precision and recall calculations were:

Precision = 66,406 / 93,865 = 0.707 = 71%
Recall = 66,406 / 209,401 = 0.317 = 32%

Linker Objects Technique

Our True Positive is the 98,863 shared extracted dependencies between Understand and this technique. Our True Positive + False Positive is the total result from this extraction method which is 206,086 extracted dependencies.

Precision = 98,863 / 206,086 = 0.479 = 48%
Recall = 98,863 / 209,401 = 0.472 = 47%

Rationale For The Differences Among The 3 Extraction Techniques

The understand technique is expensive to use and has a steep learning curve, therefore if these are major concerns, it can be rationalized to use the linker objects technique or include header files technique.

Include header files only shows relations between .c/.h files and .h files with other .h files, so if you need to examine relations between multiple .c files it can be rationalized to use the linker objects technique.

On the other hand, the linker objects technique only shows relations between multiple .c files so if you need to examine relations between c/.h files and/or .h files with other .h files it can be rationalized to instead use the include header files technique. The linker object technique also found assembly files that Understand did not find.

Conclusions & Lessons Learned

There were 2 main lessons we learned. First, the three methods we covered involved making tradeoffs and none could be said to be better than the others for all use cases. For example, The include headers tool would be useful for ensuring all necessary header files are present but its functionality otherwise is far more limited than the Understand tool but the Understand tool is far more expensive and hence not as cost-effective. Second, the tools that we developed ourselves took a long time to extract dependencies and so using them on larger systems to perform quantitative analysis could potentially take a long time.

References

Https://docs.freebsd.org/en/. https://docs.freebsd.org/en/. (n.d.). Retrieved February 10, 2023, from https://docs.freebsd.org/

McKusick, M. K. (2006). *The design and implementation of the 4.4BSD operating system*. Addison-Wesley

FreeBSD. (n.d.). Retrieved from https://github.com/freebsd/freebsd-src

"Sample Size Calculator - Confidence Level, Confidence Interval, Sample Size, Population Size, Relevant Population - Creative Research Systems." Survey Software - The Survey System, https://www.surveysystem.com/sscalc.htm. Accessed 27 November 2022

ISO/IEC JTC1/SC22/WG14, "N3054: Proposal to add Regular Expression Matching to the Standard Library," September 2010. Available: https://open-std.org/JTC1/SC22/WG14/www/docs/n3054.pdf