Library Usage Analysis in the C++ Codebase of Fedora Linux 37

by

Jiachao Deng

B.Sc., Zhejiang University, 2020

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Library Usage Analysis in the C++ Codebase of Fedora Linux 37

by

Jiachao Deng

B.Sc., Zhejiang University, 2020

Supervisory Committee

Dr. Michael D. Adams, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Stephen W. Neville, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

C++ source code analysis is conducted at scale. A framework is proposed for analyzing the C++ codebase of operating systems that employ the `dnf` package manager, such as Fedora Linux and Red Hat Enterprise Linux. The framework can run an arbitrary static analysis tool over software packages that contain C++ code from compatible operating systems. In order to evaluate the effectiveness of the framework and to better understand how the C++ language is used in practice, a C++ analysis tool is developed to study library usage with a fine level of granularity, considering instances of uses of types, type aliases, member/non-member functions, variables, and enumerators.

Our framework, combined with the C++ library usage analysis tool, is used to analyze 2 379 software packages from the codebase of Fedora Linux 37. The number of packages analyzed is two to three orders of magnitude larger than that of previous C++ research. We applied our library usage analysis tool to nearly 400 million lines of C++ code across these packages. Leveraging the Clang compiler front-end libraries, our tool extracts information from correctly parsed C++ code, which is an improved approach compared to many existing studies. As a result, the tool provides an accurate collection of library usage instances from C++ software.

Numerous observations are made regarding various aspects of library usage that can facilitate improved teaching of C++, aid in the refinement of C++ libraries, and help guide the future evolution of the C++ standard. For example, our analysis reveals that C++ programmers rarely use some C++ standard library algorithms designed for specialized purposes or combined operations. These algorithms often appear in less than 1% of all C++ software packages investigated. We suggest that the standard library exercise caution when adopting infrequently needed algorithms to maintain a streamlined interface. Such observations summarize current trends in C++ library usage and provide recommendations for improving the C++ language and its libraries.

Table of Contents

List of Tables

List of Figures

List of Listings

Preface

Acknowledgments

This thesis would never have been finished without the help of numerous people. I want to take this chance to express my gratitude to:

My supervisor, Dr. Michael Adams. Your guidance has been invaluable to me. The way we spent time and effort together to develop parallel implementations of our work to ensure correctness is unheard of in other research projects. You taught me most of the software development skills and principles I know. Through your instructions, I learned to think and work like an engineer, a great gift that I will cherish in the future.

My committee member, Dr. Stephen Neville. Thank you for being on my supervisory committee and for your effort in reviewing my thesis.

My family. I would like to thank my parents, Xiaoyu Huang and Zhimin Deng. Your unconditional support and love have always been the foundation of my journey.

My friends. I am deeply grateful for my friends: Shamim Shihab, Yuantao Tu, Xinwei Chen, Ziling Zhao, Hongjin Chen, Xinye Li, Yuanru Zheng, Mirek Drozdel, Paul Forbes, and many others. Your company and encouragement have given me great strength to overcome my difficulties.

Other students in my research group. I wish to thank Yibo Yuan, Min Liu, Hongming Yu, Young-su Ryu, and Diyu Hu. I have learned a lot from you through our group meetings and discussions. I am very grateful for our time together in the research group.

Digital Research Alliance of Canada. Thank you for providing some of the computing resources for this research.

This thesis is dedicated to my family.

Chapter 1

Introduction

1.1 C++ Library Usage Analysis

C++ has been a cornerstone of software development since its introduction by Bjarne Stroustrup [34]. Renowned for its ability to provide high-level and low-level programming features, C++ possesses capabilities such as object-oriented programming, generic programming, and direct memory management [21]. This unique combination allows for precise control over system resources while enabling the creation of complex software, making C++ particularly powerful for performance-critical projects. Its versatility and efficiency have led to widespread adoption across various domains such as network infrastructures, gaming, database management, scientific computing, and artificial intelligence. Renowned software projects employing C++ include Unreal Engine, MySQL, Cuda, and many others. A crucial part of C++ programming is the use of libraries, which often significantly enhances the productivity and robustness of C++ software projects. C++ libraries provide software building blocks that developers can utilize to perform various common programming tasks without reinventing the wheel, such as graphical user interface creation, database interaction, and complex scientific calculations. The code quality found in widely used libraries often surpasses code produced with typical daily development efforts, largely due to rigorous testing and optimization efforts that went into these established libraries, for example the Clang C++ standard library (i.e., `libc++`) [29]. Consequently, the strategic adoption of such refined libraries may improve the code quality of C++ software projects. Therefore, understanding how libraries are utilized during C++ software development is extremely important, as it can bring valuable insights to C++ users, library developers, and language standard contributors.

One way to gain such insights into C++ library usage is through the static analysis of C++ source code. Static analysis over extensive collections of real-world C++ software is highly challenging in C++. When parsing and analyzing a C++ source file, a few kinds of information other than the source file itself are required. First, the compiler flags used to compile the source file can change the meaning of C++ code and specify the search locations of necessary header files. Second, the header files included by the C++ source file will contain declarations of functions, classes, and variables that are used in the source file, which are all essential information to understand the source file. In many cases, included header files come from the software dependencies of the package, which means they are external to the software package itself. The above information

must be fully determined to ensure that static analysis of C++ source code yields accurate results. Therefore, knowledge of how a package was built is required, including the compiler flags used to compile each source file of the package. Also, (external) dependencies of the package used during the build process must be identified and located. Such information, however, is extremely difficult to obtain since C++ has no standardized build system and no standardized method for resolving dependencies.

1.2 Historical Perspective

Over the years, researchers have adopted various methods to study how programmers use the C++ language. We consider two vital aspects when reviewing these studies. First, we look at how the researchers understand the analyzed C++ code. If the C++ code is not fully parsed, the interpretation of the code can be inaccurate, leading to unreliable results. Second, we evaluate the C++ codebase of the C++ language usage studies. If the analyzed C++ code is limited in diversity and quantity, or if the code does not belong to real-world software, the study's capability to reflect the actual trends of C++ programming may be compromised.

Hanenberg et al. [15, 16, 35] adopted empirical evaluations to review the effect of certain C++ features on programmers through controlled experiments involving human participants. The researchers manually reviewed metrics, including usability, productivity, and error rates associated with using the C++ features of interest. Verdi et al. [36] manually reviewed 72 483 C++ code snippets from the Stack Overflow website. Using collaborative sessions involving multiple human reviewers, they discovered 31 types of vulnerabilities. These studies require no automated interpretation of C++ source code and are not troubled by the complexity of compiling C++ software. Such approaches, however, are often limited by the scale of the study or become labor-intensive and error-prone when the size of the dataset increases. In addition, these studies often investigate short C++ code pieces produced by participants or from online forums, which may not represent real-world C++ programming practices.

To automate the analysis of C++ code, some studies of C++ usage perform string processing on C++ source code to extract information. Bengtsson et al. [20] utilized regular expressions to perform raw string matching in C++ source code to study the usage of lambda expressions. Koppler [22] discussed a systematic approach to create fuzzy parsers for C++ source code. Moonen [24] used island grammar analysis to enhance parsers handling irregular or incomplete code. Based on the string pattern extractions of source code, these studies are limited in their ability to understand the C++ code. Therefore, performing complex analyses with high accuracy becomes difficult using these methods.

Beyond the level of raw string processing, researchers sometimes rely on C++ parsing engines from the integrated development environment (IDE) software or standalone tools. Such parsing engines can convert C++ source code to structured language representations based on language rules, providing a more accurate understanding of the semantics of C++ source code. Chen et al. [5, 37, 38] performed a series of C++ source code analyses using the C++ analysis engine provided in the Understand IDE. One of the studies looked at library usage, which is the focus of the analysis tool presented in this thesis as well, and produced some observations about the C++ standard

library. They also studied the adoption of C++ concurrency and template features to summarize C++ programming practices. Nielebock et al. [26] utilized the C/C++ development tooling (CDT) engine [12] from the Eclipse IDE to study the uses of lambda expressions in C++ source code and discovered that lambda expressions have limited adoption in concurrent programming. Balanyi et al. [4] used the Columbus parsing engine, which is an open-source C++ analysis platform, to learn design patterns from C++ source code. Zhang et al. [39] utilized the cppcheck tool to detect vulnerabilities in C++ code from the Stack Overflow website. The parsing engines and tools used in these studies may not be customizable enough to perform flexible tasks needed by source code analysis. Furthermore, the parsing engines may not align with the latest C++ language standard and can produce different results compared to C++ compilers that actually compile the source code. For example, some accuracy issues of the Balanyi et al. results were identified by Lucia et al. [23].

Compiler front-end libraries are capable of parsing C++ code to the same level of accuracy as the C++ compiler. Thus, this thesis advocates that utilizing such libraries is an ideal approach for developing C++ source code analysis tools. The Clang compiler front-end libraries [28], which are part of the LLVM compiler and toolchain libraries [30], are a suitable choice for this purpose. Some earlier studies have adopted the Clang libraries to develop C++ source code analysis tools. Schubert et al. [32, 33] developed the Phasar framework for dataflow analysis over C++ programs using the Clang libraries, which can be used to detect vulnerabilities, identify performance bottlenecks, and understand the behavior of C++ programs. Fornaia et al. [11] implemented the JSCAN framework to facilitate cross-language refactoring and optimization. Shaub et al. [9] utilized the Clang libraries to evaluate the code complexity in three releases of a C++ project by measuring cyclomatic complexity. In another study, Shaub et al. [31] also compared the accuracy of their Clang-based analysis tool against partial parsing tools over three C++ projects and found accuracy advantages of their tool. Hartogs [17] implemented Clang-based tools to examine various C++ programming metrics in the LLVM project itself. Because these studies used libraries from an actual compiler project, the accuracy of understanding C++ code semantics can surpass that of other approaches.

Analyzing C++ code that is limited in scale and diversity is a common issue in many previous C++ studies. This limitation arises from the absence of a standardized build system and a dependency management tool in C++, which often forces researchers to manually build C++ projects and, as a result, study only a small number of projects. In addition, some studies that use compiler front-end libraries focus on addressing specific static analysis tasks by developing tools without applying the tools to C++ software. Many previous studies, based on compiler front-end libraries or not, limit the dataset to one to a few dozen C++ projects, hindering the representation of general trends in C++ usage. Gygi [14] developed a scheme to build a large number of C++ projects but did not analyze the built projects. Studies that perform accurate source code analysis over a wide range of software projects exist in other languages with less complexities in building the software. For example, Dyer [10] analyzed billions of Java language nodes with precise parsing and observed general trends of Java feature usage. Such studies, however, are very much absent in C++.

1.3 Overview and Contributions of the Thesis

This thesis is concerned with C++ source code analysis at scale. Part of the work presented in this thesis was previously published [8]. A significant contribution of the thesis is a framework for analyzing the C++ codebase of operating systems that employ the `dnf` package manager, such as Fedora Linux and Red Hat Enterprise Linux. The framework can run an arbitrary static analysis tool over software packages that contain C++ code from compatible operating systems, enabling various large-scale C++ research. In order to evaluate the effectiveness of the framework and to better understand how the C++ language is used in practice, a library usage analysis tool is developed, considering instances of uses of types, type aliases, member/non-member functions, variables, and enumerators. Based on the Clang compiler front-end libraries, our analysis tool is capable of extracting information from accurately parsed C++ code, which is a major improvement over the approaches of many previous studies.

Our framework is used in conjunction with our library usage analysis tool to successfully process 2 379 software packages from the Fedora Linux 37 codebase. By investigating the library usage instances extracted from these packages, we gained numerous valuable insights. For example, we found that C++ programmers perform the majority of indexing operations without built-in bounds checking when using sequential container and view types from the standard library, posing a significant security risk. Therefore, we recommend that the standard library include bounds checking in the default behavior of indexing operations to enhance the safety of the C++ language. Such observations can facilitate improved C++ teaching, refine C++ libraries, and support the future evolution of the C++ standard. The remainder of this thesis is organized into six chapters, and we provide an overview of these chapters below.

Chapter 2 introduces the background information necessary to understand the work presented herein. First, the chapter explains the concept of binary and source packages. Next, we discuss the compilation of package source code and the LLVM/Clang libraries, which help perform C++ parsing for analysis. The importance of compiler flags and how the flags are collected are then introduced. The chapter explains build systems and dependency management issues when compiling C++ source files of a package. Container technology, which addresses some dependency management issues, is also explained. Finally, the chapter introduces the Fedora Linux 37 software repositories from which the source packages are collected.

Chapter 3 presents our proposed library usage analysis tool. The tool accepts an arbitrary piece of C++ software and categorizes encountered source code into user code and library code based on the source code's location. The tool extracts information from references in the user code that point to entities declared in the library code. Kinds of entities considered include types, type aliases, member/non-member functions, variables, and enumerators. The chapter then provides a code example to illustrate library uses. Lastly, the chapter discusses the information required when running the tool and the output structure of the tool.

Chapter 4 presents our framework that applies an arbitrary analysis tool to C++ software packages from an operating system that employs the `dnf` package manager. The potential expansion to support systems that utilize other package managers is later discussed in Chapter 7. The chapter starts by explaining the selection process of candidate source packages that may contain C++ code from software repositories of the target system. When analyzing each package, we need to

prepare a few types of information for the package, namely the package's source/header files, the collection of compiler flags to compile each package source file, and the header files included in package source code from dependencies. We then introduce processing steps for obtaining such information for each package. The chapter concludes by discussing how to use the framework and the output structure of the framework.

Chapter 5 provides the results of package processing after the framework processes 2 980 candidate source packages selected from the Fedora Linux 37 software repositories and applies the library usage analysis tool to the processed packages. The chapter presents a table showing the outcomes of each processing step. Among the candidate source packages, 2 379 are successfully processed and contain C++ source code. Examples are provided to illustrate reasons that may cause the framework to fail in processing a package or the analysis tool to fail in analyzing package source files. The overall success of package processing and tool application is highlighted. The application of our analysis tool produces a collection of instances of library uses from the C++ source code. Next, we discuss the characteristics of the C++ codebase obtained after processing. The size distribution of packages is presented. We then provide a table of the largest source packages and the application types of these large packages, demonstrating the diversity of the C++ codebase considered. Lastly, the chapter identifies factors that should be considered when interpreting the results and conclusions produced by this study.

Chapter 6 explores several topics of C++ library usage in the collected instances of library uses produced in the previous chapter. Before individual topics, the chapter discusses the statistical measures that help understand the popularity of libraries or library entities. The first topic explored is the use of top-level namespaces in the C++ codebase. Numerous observations have been made. For example, the Boost libraries are widely adopted by C++ programmers. Many popular Boost components find only partial or no counterpart implemented by the C++ standard library, suggesting potential candidates for expanding the standard library features. Several other topics are discussed, such as the uses of standard library containers, algorithms, and indexing methods with built-in bounds checking. Each section provides correlated observations and recommendations for improvements.

Chapter 7 concludes the thesis with a summary of our key contributions. Some recommendations for future work are also provided.

Chapter 2

Background

2.1 Overview

This chapter provides necessary background information to help understand the work presented. First, we introduce the software packages containing the C++ code analyzed in this study and explain the relationship between binary and source packages. C++ code in a source package needs to be compiled before the analysis. We introduce the compilation of a C++ source file and discuss the relationship between compilation and source code analysis. The analysis tools developed using the LLVM/Clang compiler libraries perform partial compilation (i.e., parsing) of a C++ source file and extract information from parsed constructs. The chapter proceeds to introduce the compiler flags used to configure the compilation of C++ source files and the importance of collecting correct flags for source code analysis. Next, we explore build systems, which automate the compilation of C++ source files in a package. We then discuss the build wrapper used to collect compiler flags when compiling one or more C++ source files. The compilation database file containing the collected flags is explained with an example. The chapter then introduces software dependencies, which must be resolved before building a source package. A package manager is employed to resolve software dependencies. The complexities of dependency management when processing a large number of source packages are examined. Container technology can help address these complexities. Finally, the chapter introduces software repositories from which source packages are selected and obtained.

2.2 Software Packaging

We will first discuss the concept of software packaging to provide a clear understanding of the origins of our data. The C++ code analyzed in this study is distributed as software packages. These packages are collected from Fedora Linux, a well-known Linux distribution for its rapid innovation and inclusion of the latest software technologies. Because Fedora Linux is among the top popular Linux distributions, most major C++ software on Linux can usually find corresponding packages on this distribution. Consequently, we are able to study a much larger codebase compared to the few selected software projects analyzed in many previous research.

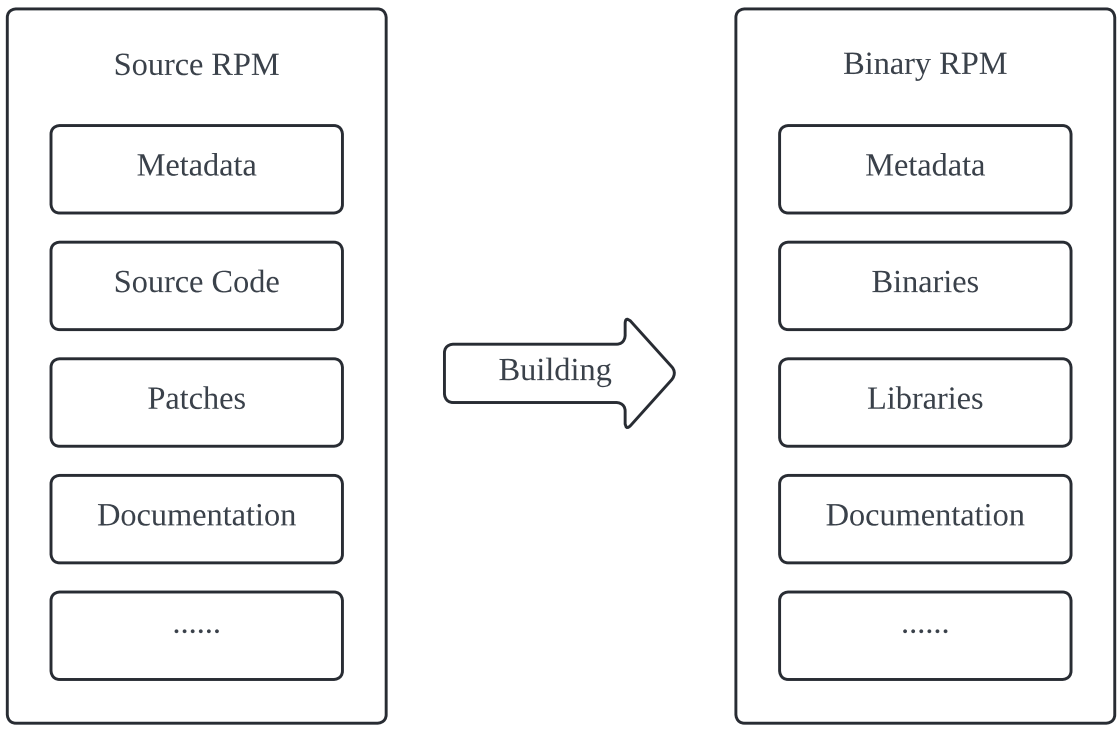

Figure 2.1: Major contents of binary and source packages, and their relationship.

Software packages on Fedora Linux are distributed in the Red Hat package manager (RPM) format [19]. RPM packages are designed to streamline software distribution, installation, and management on Fedora Linux and other Red Hat-based Linux distributions. Each RPM file includes metadata describing the package and the actual data, such as program binaries, libraries, documentation, and configuration files. The metadata includes the package name, version, release number, and architecture (such as x86 or ARM). Additionally, it contains a list of the other packages upon which the package depends.

Each (binary) package is typically built from a corresponding source package. A source package contains the source code of the software, as well as a recipe that describes how to prepare the source code and compile it into binary files. A source package also contains documentation similar to a (binary) RPM package and software patches to be applied to the source code. Because source packages contain both the source code and the information necessary to compile it into binaries, this thesis uses source packages to prepare C++ software to be correctly understood by analysis tools. A diagram showing the major contents of binary and source packages and the relationship between them is presented in Figure 2.1.

The `rpmbuild` software tool is specifically designed to build (binary) packages from source packages. When building a source package, `rpmbuild` interprets the package's metadata and verifies that the external packages required by the package (i.e., dependencies) are installed. Next, `rpmbuild` extracts the package and applies software patches to ensure the completeness of the

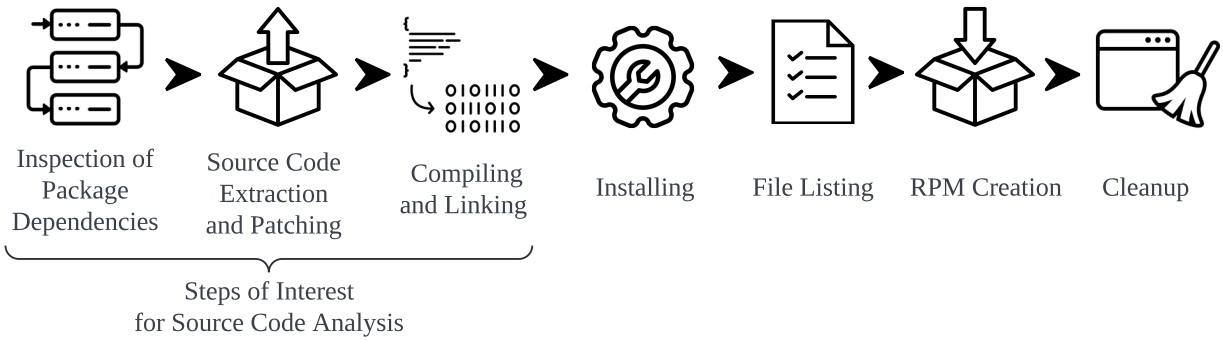

Figure 2.2: Steps performed by `rpmbuild` to build binary packages from source packages.

package content. Following this, `rpmbuild` runs the build tool that invokes the compilation of package source code. Figure 2.2 illustrates the major steps performed by `rpmbuild` when building a source package. The first two steps and the compilation phase of the third step are involved in preparing C++ software packages for source code analysis. Other steps of `rpmbuild` are irrelevant to this study and, therefore, not discussed.

2.3 Compiler and LLVM/Clang Libraries

The previous section introduced the source packages that need to be built before being analyzed. Building a source package that contains C++ involves compiling one or more C++ source files of the package into binary object files. Before introducing the compilation of multiple source files when building a package, we will first discuss compiling a single C++ source file and how the compilation relates to source code analysis. The section then introduces the compiler libraries that can be used to develop analysis tools to perform partial compiling to extract code information from C++ files.

C++ compilation involves translating a C++ source file and its included header files into a binary object file, as depicted in Figure 2.3. Before generating an object file, the C++ compiler parses the C++ source file with included header files to generate an abstract syntax tree (AST). The AST is a tree-like data structure that represents the syntax of the C++ code and is easily traversable by analysis tools. Therefore, the AST can be used to extract various information from the C++ code. Partial compilation (i.e., parsing) of a C++ source file is crucial for source code analysis. This study does not consider other steps of compilation and the production of executable/library files using object files after compilation.

Compiler front-end libraries can be utilized to perform precise parsing to generate the AST and extract information from the AST [1, 2]. Thus, these libraries are essential for developing tools that can accurately analyze C++ code. The Clang compiler front-end libraries, part of the LLVM compiler and toolchain libraries, are a suitable choice for this purpose. The Clang libraries provide a compiler front-end for the C-family languages, including C++. The functionalities of compiler front-end libraries include the generation of AST and providing interfaces to traverse/manipulate the AST. A variety of callback entries are provided by the Clang libraries. Using these callback

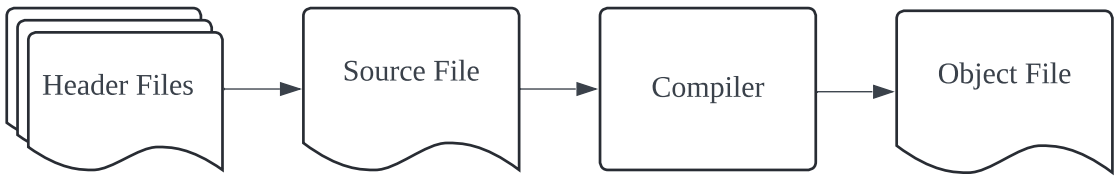

Figure 2.3: Compilation of a C++ source file.

entries, developers can write custom operations to retrieve information from the AST or modify the AST, performing various analysis tasks such as source code analysis, code completion, and refactoring. Some supportive functionalities from the LLVM libraries are also used in this study, including command-line option parsing, type casting, and input/output. Running a Clang-based analysis tool on a C++ source file involves partial compilation of the file. Thus, we must be able to configure and perform the source file compilation correctly before the analysis. The analysis tool reuses the compile configurations to interpret the C++ code accurately.

2.4 Compiler Flags and C++ Code Semantics

Having introduced the compilation process of a C++ file and how analysis tools can perform partial compilation to extract C++ code information, we will now discuss the configuration required for compiling/analyzing a C++ source file. The compile configuration is achieved by using compiler flags. Compiler flags are arguments passed to compilers when compiling C++ files. Some compiler flags specify the search path for header files, which are necessary for successful compilation. Some other compiler flags toggle specific code blocks or control the version of the C++ standard and language extensions used, which change the semantics of the C++ code. Therefore, collecting the correct set of compiler flags is essential to ensure the successful and accurate interpretation of a C++ source file.

Listing 2.1 demonstrates a compiler flag that turns code blocks on or off, changing the meaning of a C++ source file. In Listing 2.1, a function `foo` is defined with a conditional block based on the `ENABLE_FOO_FEATURE` flag. When the `ENABLE_FOO_FEATURE` flag is passed to the compiler through the compiler's command-line interface, the code block that defines the `foo` function and the code block that invokes the `foo` function are included in the compilation. Otherwise, if the `-DENABLE_FOO_FEATURE` option is not specified, `foo` related code blocks are excluded. This practice is commonly used to improve the flexibility of C++ programs.

Other than the change of C++ code semantics shown in Listing 2.1, the compiler flags that specify the search paths for header files are often crucial for the compilation and the analysis. The compiler needs to know the location of header files included by the C++ source file. Declarations in header files contain the necessary information for C++ entities, such as types and functions. Without the compiler flags that specify the location of header files internal and external to the package, the compiler may not find some required declarations and may produce compile errors.

Listing 2.1: Toggling an optional feature using a compiler flag.

```
#include <iostream>

// An optional foo feature is defined.
// This feature can be turned on or off based on
// the compiler flags specified.
#ifdef ENABLE_FOO_FEATURE
void foo() {
    std::cout << "Foo feature enabled\n";
}
#endif

int main() {
#ifdef ENABLE_FOO_FEATURE
    foo();
#else
    // Print a message if the foo feature is not available.
    std::cout << "Foo feature not available.\n";
#endif

    return 0;
}
```

2.5 Build Systems

Having introduced the importance of compiler flags, we will now discuss the build systems used to automate the compilation. Building a source package may involve compiling one or more C++ source files. During the build process, compiler flags for each source file are usually not specified manually. Instead, a build system is often used to automatically configure compilations. Most packages that contains C++ code require the use of build systems for several reasons.

First, modern software typically adheres to modularity principles, dividing C++ code into many individual files. Compiling all package source files and applying the correct compile flags to each file is error-prone. Additionally, software development and updates are often incremental. If the content of one header file is modified, only the source files that include this header file need to be recompiled. Without the capability to detect affected source files, a total recompilation of all compiled source files becomes necessary, which is time-consuming. Furthermore, tailoring the compilation for different platforms becomes arduous when a package is cross-platform. Challenges such as the above necessitate using build systems to orchestrate the compilation of C++ source files for most packages containing C++ code.

Figure 2.4 demonstrates the relationship between the build system and the compiler when compiling multiple C++ source files of a package. When using a build system, programmers usually specify high-level characteristics and attributes of the package source code. The build

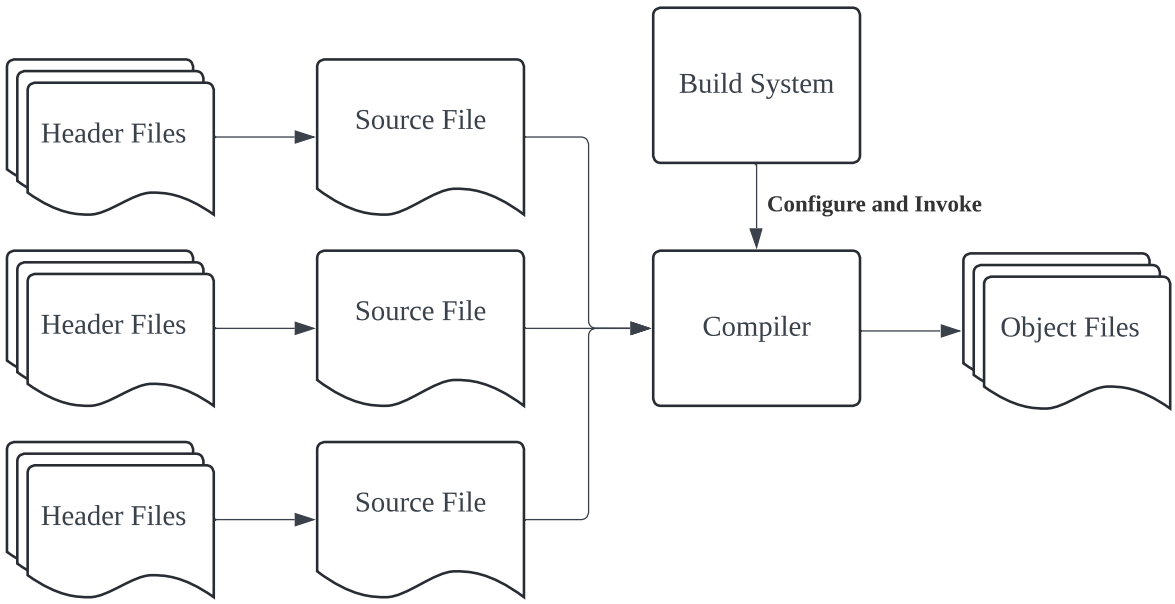

Figure 2.4: Build system directs the compilation of C++ source files.

system interprets these specifications, generates compile flags for each source file, and invokes the compiler to compile the source files. This approach allows programmers to avoid manual configuration of compile jobs, ensuring the correct compilation of package source files. Other functionalities of build systems, such as directing the linking of object files, are not related to source code analysis and are not discussed. In C++, there is no standardized build system. Instead, C++ programmers can choose from a variety of build systems, such as CMake, Meson, Ninja, and Bazel, each offering a different collection of features.

Researchers aiming to obtain compile flags for analyzing C++ code in software projects often encounter the challenge of invoking the appropriate build system for each project. Due to the diverse range of build systems in use, manual invocation of build systems is often required, which limits the number of software projects investigated in many previous studies. The build instructions specified in source packages help unify the build system invocations of different packages. The `rpmbuild` tool extracts build instructions from the metadata of a source package and calls the specified build system of the package. Notably, the invocation of `rpmbuild` is a standardized operation across different source packages. Utilizing source packages and the `rpmbuild` tool makes it feasible to build thousands of packages for analysis.

2.6 Build Wrapper and Compilation Databases

With the knowledge of how the build systems automate the compilation of C++ source files, we now turn our focus to the collection of compiler flags when building a source package. A build wrapper is utilized to collect compiler flags. In this study, we use the Build EAR (`bear`) program

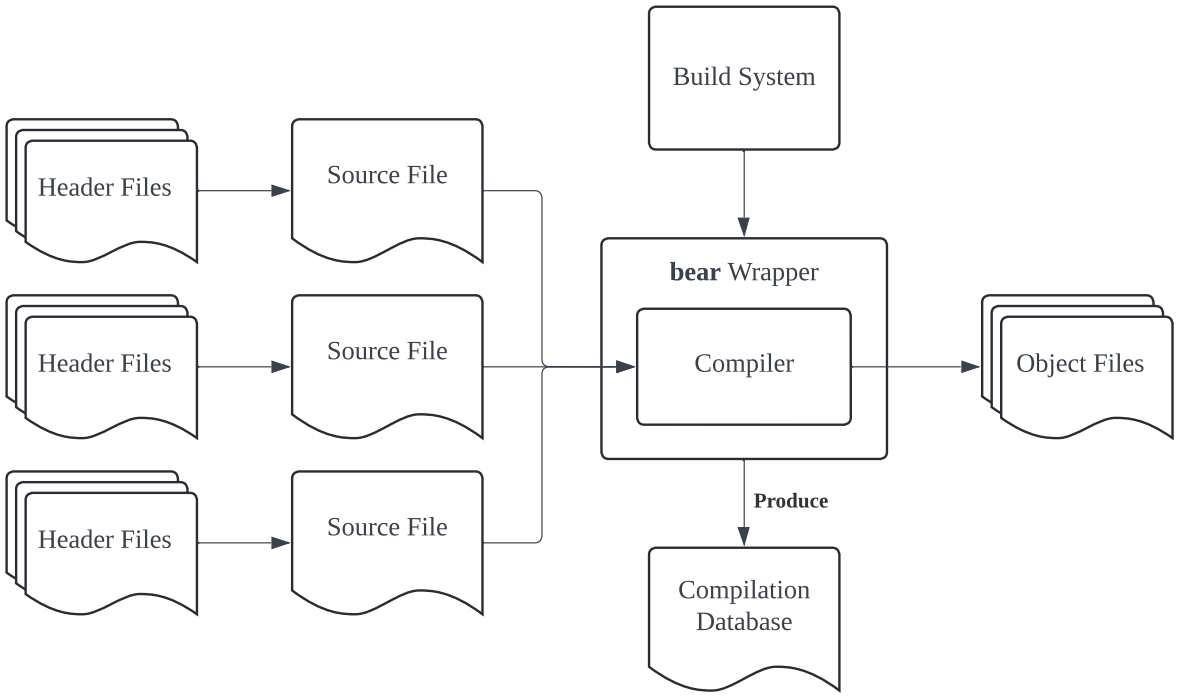

Figure 2.5: Capturing compiler flags using the `bear` wrapper.

[25] as the build wrapper. When running build systems, the build wrapper intercepts the commands sent to the compiler by build systems and records the compiler flags used. The original commands are then forwarded to the actual compiler. When the build process finishes, `bear` produces an output file called a compilation database that contains the captured compiler flags. Figure 2.5 illustrates the process of using `bear` to collect compiler flags while compiling multiple C++ source files.

The output of `bear` is a Clang-style compilation database file formatted in JavaScript Object Notation (JSON) [6], a lightweight data-interchange format that organizes data in a structured and easy-to-navigate manner. Each entry in the compilation database file contains the pathname of a source file, the compile command containing compiler flags for the source file, and the home directory of the package to which the source file belongs. Therefore, when C++ source files are compiled, the compilation information is saved and can be reused by analysis tools to interpret the C++ code accurately.

Listing 2.2 provides an example of a compilation database, which contains the compiler flags for the `foo.cpp` and `bar.cpp` source files of the package `foo-1.0.0`. This package is located under a hypothetical Linux home directory `/home/jdoe/`. The compiler flags in the compilation database specify that the compiler should search for header files in the `foo-1.0.0/include/` directory, and the C++ standard used to compile the files is C++11.

Listing 2.2: Compilation database containing the compiler flags of two source files of the `foo-1.0.0` package.

```
[
    {
        "directory": "/home/jdoe/foo-1.0.0/src",
        "command": "/usr/bin/clang++ -I../include -std=c++11 -
           c foo.cpp -o foo.o",
        "file": "foo.cpp"
    },
    {
        "directory": "/home/jdoe/foo-1.0.0/src",
        "command": "/usr/bin/clang++ -I../include -std=c++11 -
           c bar.cpp -o bar.o",
        "file": "bar.cpp"
    }
]
```

2.7 Software Dependencies and Package Management

Before building a source package that contains C++ code, other packages on which the package depends must be installed. These packages are called software dependencies of the package. Software dependencies contain pre-implemented code that does not belong to the package but is referenced by the package's code from external locations. A source package usually contains only the source code developed for the package, not including all its transitive (i.e., direct and indirect) dependencies. Therefore, the dependencies must be installed separately before performing the software build. The typical packaging approach reduces the package size and facilitates software updates, as dependencies can often be updated independently of the packages that rely on them.

Figure 2.6 illustrates an example of a software dependency structure. Each arrow in the figure denotes that a package depends on another package. The software package in Figure 2.6 relies on several dependencies, some of which have their own dependencies, forming a graph structure. The software package can only be built when all dependencies of the correct versions are installed.

Managing dependencies is often a challenging task for numerous reasons. In real-world scenarios, software packages typically have dependency graphs comprising dozens or even hundreds of dependencies, creating much more complex dependency structures than shown in Figure 2.6. The versions of dependencies are also crucial, as software often requires a minimum version and sometimes even a specific version of a dependency to build appropriately. Some dependencies cannot coexist, causing dependency conflicts that require specific rules to resolve. Lastly, the package may have dependencies relied on by other dependencies but not the package itself. Figure 2.6 shows the indirect dependencies D, E, and F. Such indirect dependencies are difficult to follow.

The approaches to perform dependency management are not standardized in C++. For smaller projects, dependency management may still be manually achievable. Instructions for installing the dependencies may exist in the package's written documents, which require manual enforcement.

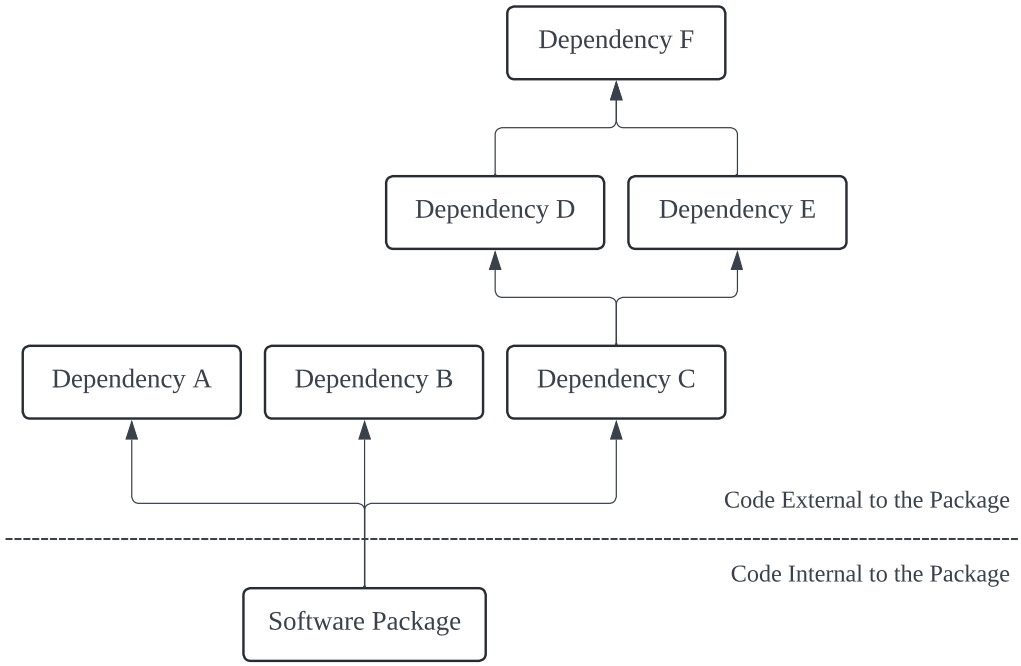

Figure 2.6: Graph structure of a package's software dependencies.

Build systems sometimes offer dependency-managing support to varying degrees, such as CMake and Bazel. In this study, because the analyzed C++ software is in the form of software packages, dependency management is performed by using a package manager. A package manager is a tool that automates the process of installing, upgrading, configuring, and removing software packages. Before installing a package, the package manager detects and resolves the package's dependencies.

In this study, the Dandified YUM (`dnf`) tool [3] is used as the package manager. The possibilities for making our work compatible with other package managers will be discussed in Chapter 7. The `dnf` package manager can interpret the metadata of a source package to detect all transitive dependencies of the package. The `dnf` software then installs the detected dependencies and addresses dependency conflicts. Typically, `dnf` is used before invoking the `rpmbuild` tool on a source package to ensure `rpmbuild` has a complete set of dependencies for building the package. Similar to invoking `rpmbuild`, invoking `dnf` for a source package is consistent across different source packages. Therefore, the `dnf` tool can be utilized to automate dependency installation processes when building a large number of packages.

2.8 Dependency Complexities and Containers

The use of the `dnf` package manager introduced in the previous section does not address all the complexities associated with managing dependencies for software packages. When building large quantities of source packages, thousands of dependencies are installed. The chance of dependen-

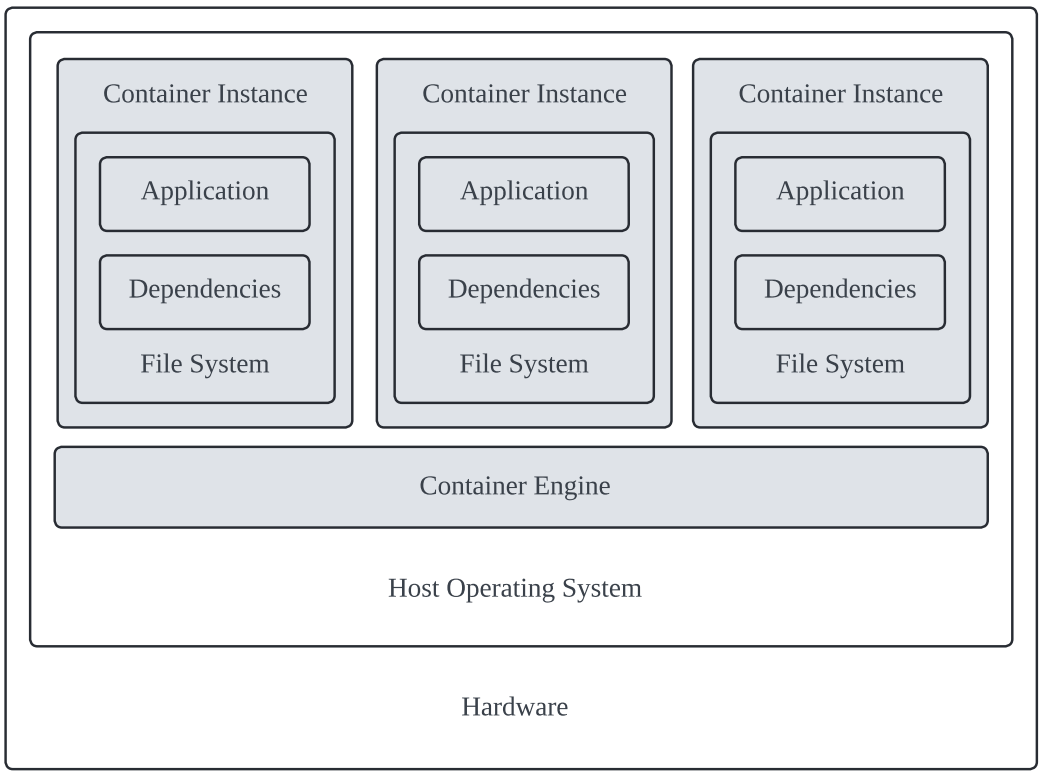

Figure 2.7: Architecture of containers running on a host system.

cies increases rapidly during the process and the disk consumption of these dependencies also becomes problematic. Also, because multiple versions of a dependency are often all acceptable, the consistency of analysis results may be compromised if different versions of a dependency are installed.

In this study, the container technology is adopted to address the above complexities. A container provides an isolated environment to encapsulate applications with a collection of dependencies. First, the application (i.e., processing a software package) and its dependencies are packaged into a file called a container image. When running the application, the container image is deployed to create a container instance. The container instance is managed by the container engine and runs on the host system. A container instance has its own filesystem, network interfaces, and computing resources, ensuring that the application and its dependencies are isolated from other processes on the host system. Figure 2.7 illustrates the structural diagram of containers running on a host system.

Because a container is an isolated environment, dependencies installed when processing a software package do not interfere with those of other packages, reducing the chance of dependency conflicts. Another key feature of containerization is statelessness, which means that each container instance operates independently without retaining any context or state from previous instances. Consequently, as long as the container instances used to process a package are constructed from the same image, the `dnf` manager will start resolving dependencies from the same initial state,

installing a consistent set of dependencies. This consistency ensures that the result of analysis of the packages is reproducible. The container filesystem stores the installed dependencies when processing a package and can be eliminated after processing. Thus, the disk space consumed by the installed dependencies can be reclaimed after processing a package. Additionally, the performance overhead associated with frequently creating and eliminating large quantities of container instances is significantly lower than that of virtualization technology, as containers do not run their own operating systems. This performance advantage is crucial when processing thousands of software packages.

The container engine utilized in this study is Podman [18]. Compared to other container engines, Podman offers several particularly advantageous features for this study. The command-line interface of Podman is compatible with Docker, a commonly used container environment, which facilitates a smooth adoption process. Additionally, Podman containers run without root privileges. Applications running within a Podman container can only obtain root privileges in the container environment but not on the host system, providing a significant security advantage for this study. Furthermore, Podman does not require a daemon process to oversee running containers, simplifying management. These characteristics make Podman a suitable choice for this research.

2.9 Software Repositories and Package Selection

The preceding sections provided all the necessary background information on preparing C++ software packages for source code analysis tools. We now shift our focus to selecting and obtaining thousands of C++ software packages.

Software repositories are centralized locations where software packages are distributed. The C++ software packages considered in this study are obtained from software repositories of Fedora Linux 37. Different types of repositories are available for various purposes, such as fixed, update, and third-party repositories. Software packages in fixed repositories remain unchanged once they are determined with each release of Fedora Linux, except for security reasons. Since the consistency of analysis results is crucial, fixed repositories are ideal for this study. Some repositories, including the fixed repositories, have corresponding source repositories from which the contents of (binary) repositories are built. The source packages analyzed in this thesis are selected from the fixed source repositories of the Fedora Linux 37 release.

The tool used to navigate these repositories is the `dnf` package manager, which is introduced for its capabilities to resolve package dependencies in Section 2.7. Other than resolving dependencies, `dnf` can query packages from Fedora Linux software repositories with specific criteria, such as package name, version, and dependencies. The querying capability of `dnf` can be utilized to select source packages that may contain C++ code. If a source package has a dependence on one of the packages that contains a C++ compiler, namely, `gcc-c++` [13] or `clang`, the source package may have C++ code. Section 4.2 discusses the package selection process in detail. Figure 2.8 shows how `dnf` can help to interact with software repositories.

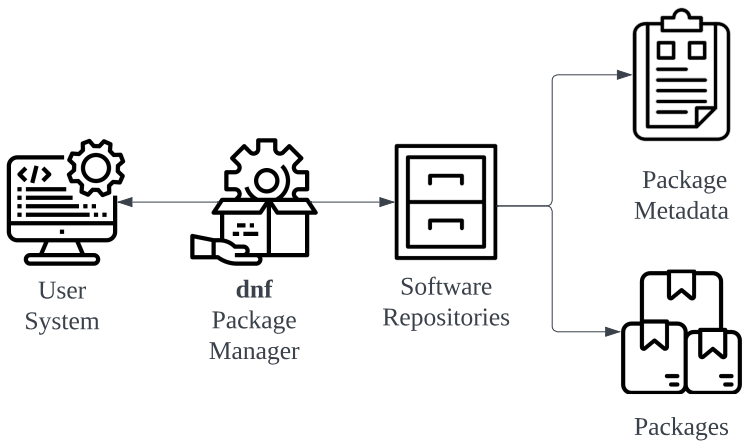

Figure 2.8: Interaction with software repositories using `dnf`.

Chapter 3

Proposed Library Usage Analysis Tool

3.1 Overview

With the necessary background knowledge introduced, we present our proposed Clang-based library usage analysis tool. This tool extracts information from code locations in the compiled source files of a given C++ software project that refer to entities declared in libraries, such as types and functions. The chapter then provides examples demonstrating the identified instances of library usage. Finally, we discuss the inputs and configurations required to run the tool and the outputs it generates.

3.2 Functionality of Analysis Tool

As introduced in Section 2.3, the Clang libraries can help perform C++ parsing to produce the AST. The AST is a syntactic representation of a source file that can be used for source code analysis. Based on the Clang libraries, the particular analysis tool developed in this study focuses on extracting instances of library usage from C++ software.

The tool extracts information about the use of various entities declared in libraries, such as types and functions. Our tool takes one or more C++ source files listed in a compilation database as input to be analyzed. The compilation database of the analyzed software is produced by our framework, which will be later introduced in Chapter 4. Source files in the compilation database, along with directly or indirectly included header files, are each classified as either user code or library code. User code is defined as the code within the software build directory, while library code is defined as the code located outside of the build directory. Based on the above classification, our tool identifies each location in the user code referencing entities declared in the library code. We deem each such reference to be a use of a library. Figure 3.1 illustrates a *type use* and a *function use* found in the source file `user.cpp`. The kinds of library usages identified by our tool include:

1. **Type use.** Types are some of the most important library entities used in C++ code. Functionalities provided by libraries are often encapsulated into types. For example, the C++ standard library provides the `std::vector` type. An `std::vector` object declared by user

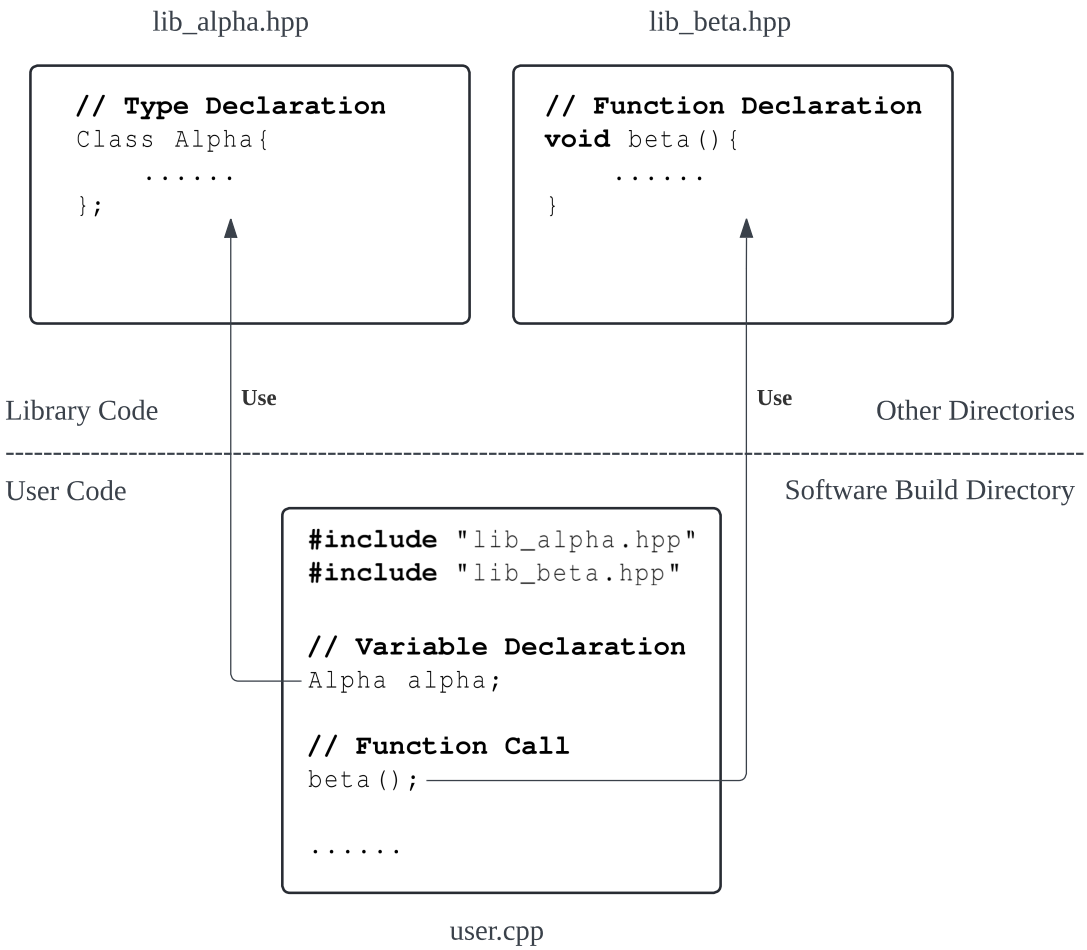

Figure 3.1: Type and function usages in user code referring to entities declared in library code.

code manages a dynamically sized array, providing various operations to manipulate the elements. Declaring variables is the most typical type utilization. Other type usages include user code declaring type aliases, function parameters, and data members of library types.

2. **Type alias use.** Libraries often provide type aliases to simplify complex type names or to offer common template instantiations. For instance, the C++ standard library includes the `std::string` type alias for `std::basic_string<char>`, which is frequently used for declaring string objects. In addition to identifying the instantiation of objects using library type aliases, our tool also detects type aliases declared by user code based on existing library aliases.

3. **Function/method/constructor use.** Functions package a set of operations that can be invoked by user code, providing critical encapsulation for improving code organization and maintainability. For example, the C++ standard library provides the function `std::sort` to sort a given range of elements in a container. Methods are functions that belong to classes. Non-static methods operate on class objects, while static methods are invoked without requiring an object. The `std::vector` type from the C++ standard library provides methods like `begin` and `rbegin` to access the first and last elements of the container. Constructors are special member functions that are invoked when objects are created. The `std::vector` type has different constructors for creating `std::vector` objects in different manners, such as specifying the initial size or initializing the container with a range of elements. Our tool identifies situations where functions, methods, and constructors are invoked. In addition, our tool detects instances where functions and methods are taken addresses by user code.

4. **Variable use.** Sometimes, variables are created in library code. User code refers to these variables without creating user instances, such as the `std::cin` and `std::cout` objects from the C++ standard library, which is commonly employed for input and output operations. The analysis tool finds utilization of such library variables in user code.

5. **Enumeration use.** Enumerations are programmer-defined data types that consist of named integer constants. Enumerations improve code clarity by providing meaningful names to integer values and enforce type safety by restricting variable values to the defined options. Enumerations are often used to represent a set of related constants. For example, the C++ standard library defines an enumeration `std::ios_base::openmode` to specify file opening modes. The enumeration values `std::ios_base::in`, `std::ios_base::out`, and `std::ios_base::app` represent the input, output, and append modes, respectively. Our tool examines instances where user code employs library enumerations.

For each instance of library use, our tool records detailed information, including the kind of library usage (e.g., enumeration), the source location (i.e., the pathname of the source/header file and the line/column number in that file) where the entity is referenced in user code, the source location where the entity is declared in library code. More importantly, the tool records extensive information about the declared entity, such as identifier name, declaration context, and type/template information. The extracted information is organized in JSON format as output. The output file can then be further analyzed to explore various aspects of library usage in the software. For

example, namespace information can be found in the declaration context of a recorded instance. The C++ standard library uses the `std` namespace. By examining the collected instances under the `std` namespace, we can compare the popularity of different functionalities offered by the standard library. Chapter 6 will present our explorations in detail.

3.3 Examples of Library Usage

In the preceding section, we introduced the functionalities of the library usage analysis tool. To better illustrate the instances of library use collected by the tool, we now present code examples of a C++ library named `foo_lib`, as well as an application that employs the `foo_lib` and the C++ standard library. Our analysis tool processes the application source code and identifies references to entities declared in libraries.

The `foo_lib` library provides basic logging capabilities. Listing 3.1 shows the header file `foo_lib.hpp` that contains the declarations of `foo_lib`. Library entities defined in this file that can be referenced by user code are highlighted. The header file first defines an enumeration `LogLevel` to represent various logging levels and a type alias `Messages` of `std::vector<std::string>` to manage an array of messages. The library then declares a `Logger` type responsible for printing messages and managing the log level. The default log level of `LogLevel::Info` is specified as a library variable. Lastly, the library provides a helper function `formatLogMessage`.

Listing 3.1: Header file `foo_lib.hpp` contains declarations of the `foo_lib` library.

```
1  #ifndef FOO_LIB_HPP
2  #define FOO_LIB_HPP
3
4  #include <string>
5  #include <vector>
6  #include <string_view>
7
8  namespace foo {
9
10 enum class LogLevel {
11     Debug,
12     Info,
13     Warning,
14     Error
15 };
16
17 using Messages = std::vector<std::string>;
18
19 class Logger {
20 public:
21     Logger(std::string_view loggerName, LogLevel logLevel)
          noexcept;
```

```
22
23      bool log(std::string_view message) const;
24      LogLevel getLogLevel() const noexcept;
25
26  private:
27      std::string loggerName;
28      LogLevel logLevel;
29  };
30
31  inline constexpr LogLevel defaultLogLevel = LogLevel::Info;
32
33  std::string formatLogMessage(LogLevel level, std::string_view
        message);
34
35  } // namespace foo
36
37  #endif // FOO_LIB_HPP
```

Listing 3.2 presents the source file `foo_lib.cpp`, which provides the definitions of the entities declared in the header file `foo_lib.hpp`, highlighting the identifier names of the library items. The `Logger` type is implemented with a constructor, a method `log` to print messages, and a method `getLogLevel` to retrieve the log level. The `formatLogMessage` function is also defined in the source file, which assembles log messages with log levels.

Listing 3.2: Source file `foo_lib.cpp` contains definitions of the `foo_lib` library.

```
1  #include "foo_lib.hpp"
2  #include <iostream>
3
4  namespace foo {
5
6  Logger::Logger(std::string_view loggerName, LogLevel logLevel)
       noexcept
7      : loggerName(loggerName), logLevel(logLevel) {}
8
9  bool Logger::log(std::string_view message) const {
10     std::string formattedMessage{formatLogMessage(logLevel,
           message)};
11     std::cout << formattedMessage << '\n';
12     return std::cout.flush() ? 0 : 1;
13 }
14
15 LogLevel Logger::getLogLevel() const noexcept {
16     return logLevel;
17 }
```

```
18
19 std::string formatLogMessage(LogLevel level, std::string_view
      message) {
20     std::string levelStr;
21     switch (level) {
22         case LogLevel::Debug: levelStr = "DEBUG"; break;
23         case LogLevel::Info: levelStr = "INFO"; break;
24         case LogLevel::Warning: levelStr = "WARNING"; break;
25         case LogLevel::Error: levelStr = "ERROR"; break;
26     }
27     return levelStr + ": " + std::string(message);
28 }
29
30 } // namespace foo
```

Listing 3.3 demonstrates an application that consists of a single source file named `main.cpp`, which employs the entities declared in the `foo_lib` library and the C++ standard library. The locations of identified library usages are highlighted. The program performs several logging operations and returns the number of logging errors encountered.

Listing 3.3: Source file `main.cpp` employs entities declared in libraries.

```
1  #include "foo_lib.hpp"
2  #include <iostream>
3
4  int main() {
5      foo::Logger logger{"MainLogger", foo::LogLevel::Info};
6
7      if (!logger.log("Starting application...")) {
8          std::cerr << "Failed to log message: "
9              << "Starting application...\n";
10     }
11
12     int loggingErrors = 0;
13
14     const foo::Messages messages{
15         "Hello", "World", "This is a test"};
16     for (const auto& message : messages) {
17         if (!logger.log(message)) {
18             std::cerr << "Failed to log message: "
19                 << message << '\n';
20             ++loggingErrors;
21         }
22     }
23
```

```
24      if (logger.getLogLevel() == foo::defaultLogLevel) {
25          if (!logger.log("Log level matches default.")) {
26              std::cerr << "Failed to log message: "
27                  << "Log level matches default.\n";
28              ++loggingErrors;
29          }
30      }
31
32      if (logger.getLogLevel() == foo::LogLevel::Info) {
33          if (!logger.log("Current log level is Info.")) {
34              std::cerr << "Failed to log message: "
35                  << "Current log level is Info.\n";
36              ++loggingErrors;
37          }
38      }
39
40      return loggingErrors;
41  }
```

Suppose we give the compilation database of the application to our tool, which contains the compile instructions for `main.cpp`. The tool takes `main.cpp` and its included header files as input. In this case, the user code is the `main.cpp` file, and the library code comprises `foo_lib.hpp` as well as C++ standard library headers directly/indirectly included by `main.cpp`. Our analysis tool examines the user code `main.cpp` and identifies references to entities declared in the library code. Table 3.1 lists the library uses detected in Listing 3.3. For each instance of library usage, Table 3.1 shows some of the information collected by the tool, including the kind of used entity, declaration context, identifier name, and line/column number where the entity is referenced in `main.cpp`. From Table 3.1, we can see that numerous instances of library usage are detected despite the application and the library being relatively simple. Referencing library entities is frequently performed when developing most C++ software. Therefore, observing the library uses can provide various insights into C++ programming practices.

3.4 Tool Invocation and Output

The previous sections described the functionalities of the library usage analysis tool and provided examples of library usage instances captured by the tool. Next, we discuss the inputs/configurations required when running the tool and the outputs produced by the tool.

Figure 3.2 illustrates the inputs and outputs of the library usage analysis tool. In the command-line interface of our tool, we specify the compilation database file of the software to be analyzed. For each compiled source file stated in the compilation database, our tool applies the corresponding compiler flags to parse the source file and included header files.

Some configuration information must also be provided to the tool via command-line options. A pathname prefix is passed to the tool to identify code locations that belong to the user code.

Table 3.1: Library uses identified in `main.cpp`

Kind	Declaration Context	Identifier Name	Line : Column
Type	`foo`	`Logger`	5:5
Type Alias	`foo`	`Messages`	14:5
	`std::vector`	`const_iterator`	16:30
Function	`std`	`operator<<`	8:19, 9:13, 18:23, 19:17, 19:28, 26:23, 27:17, 34:23, 35:17
Method	`foo::Logger`	`log`	7:10, 17:14, 25:14, 33:14
	`foo::Logger`	`getLogLevel`	24:9, 32:9
	`std::basic_string`	`operator basic_string_view`	17:25
	`std::vector`	`begin`	16:30
	`std::vector`	`end`	16:30
Constructor	`foo::Logger`	`Logger`	5:17
	`std::basic_string_view`	`basic_string_view`	5:24, 7:21, 25:25, 33:25
	`std::vector`	`vector`	14:25
	`std::basic_string`	`basic_string`	15:9, 15:18, 15:27
Variable	`std`	`cerr`	8:9, 18:13, 26:13
	`foo`	`defaultLogLevel`	24:33
Enumeration	`foo`	`LogLevel`	5:38, 32:33, 34:13

Typically, this prefix is the absolute pathname of the build directory of the analyzed software. Code locations that start with the prefix are considered user code. Additionally, a library pathname prefix can be provided to study the usage of libraries installed in a particular directory. If no library prefix is specified, all code locations not identified as user code are considered library code. Our tool then detects all instances in the user code that reference entities declared in the library code.

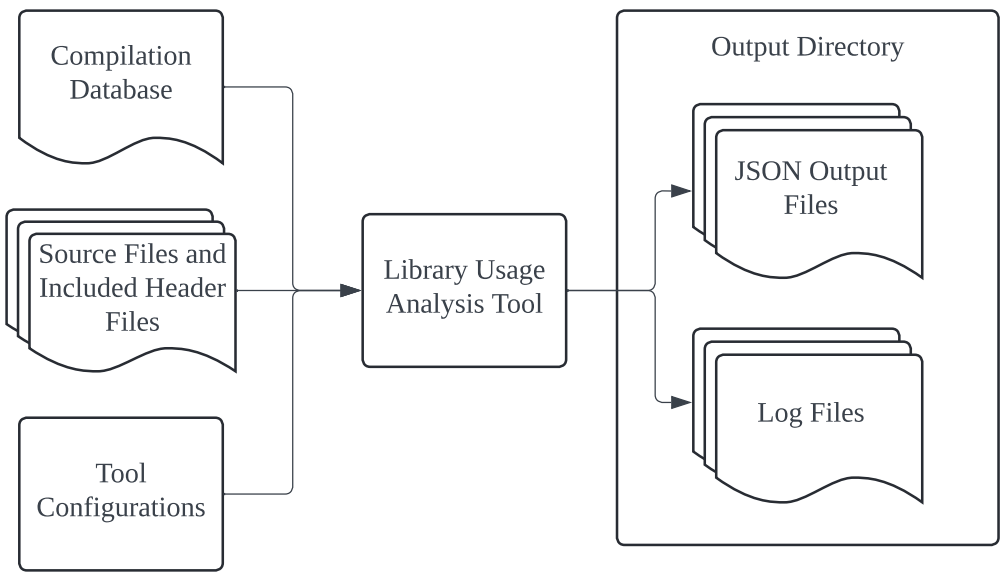

Figure 3.2: Invocation of the library usage analysis tool.

The tool generates a JSON output file containing all identified library uses for each source file in the compilation database. Supportive log files are also created for debugging purposes. These output files are stored in a designated output directory. After the tool finishes processing all source files listed in the software's compilation database, we can explore the output files to study various aspects of library usage in the analyzed software.

Chapter 4

Proposed Package Processing Framework

4.1 Overview

Having introduced our library usage analysis tool, we present the framework that obtains and processes source packages to apply an arbitrary C++ analysis tool to a wide range of C++ software. This chapter starts by explaining the selection process of source packages that may contain C++ code from software repositories of a `dnf` based operating system. Next, we introduce the necessary types of information for source code analysis and the methods used by the framework to collect such information. The steps for processing each package are then explained in detail. To conclude the chapter, we discuss how the framework is used and the output structure of the framework.

4.2 Software Package Selection

We developed a framework for source package processing to apply an arbitrary C++ analysis tool (e.g., the library usage analysis tool introduced in Chapter 3) to a large codebase. This framework obtains source packages containing C++ code from operating systems that use the `dnf` package manager, such as Fedora Linux and Red Hat Enterprise Linux. The framework then prepares the source packages for analysis and applies the specified analysis tool to the packages.

The first step of the framework is to select source packages that may contain C++ code. As introduced in Section 2.9, the querying capability of the `dnf` package manager facilitates such selection. To identify source packages with C++ code, we rely on their dependency on packages that provide a C++ compiler, specifically `gcc-c++` or `clang`. First, the framework uses `dnf` to query all source packages that depend on `gcc-c++`, saving the identified package names in a candidate list. Next, since the `clang` package includes both C and C++ compilers, some packages might depend on `clang` solely for its C compiler. Therefore, we query source packages that depend on `clang` and apply a language classification tool called `cloc` [7]. The `cloc` tool examines the content of a source package and categorizes package source files by programming language based on file extensions. Source packages that depend on `clang` and contain at least one C++ source file, as identified by `cloc`, are also added to the candidate list.

A source package can falsely indicate a dependency on another package (i.e., claim that it has a dependency on another package when, in fact, it does not). Consequently, some packages selected based on the above approach might not contain any C++ code. Our library usage analysis tool, developed using the Clang compiler front-end libraries, is capable of detecting whether each source file is written in C or C++. During the analysis, C source files are excluded, and statistics from packages without C++ code are discarded.

4.3 Analysis Prerequisites

The selection process introduced in the previous section produces a list of candidate source packages that may contain C++ code. To apply an arbitrary C++ analysis tool to each source package, the framework must ensure the availability of the following types of files:

1. Any header files external to the package that are included by one or more C++ source files in the package.
2. All C++ source files in the package that would be compiled during the build process.
3. All header/source files that would be generated when building the package.

To meet the first prerequisite, we ensure that all software dependencies of the package are installed prior to building it. The installation is executed by using the `dnf` package manager. The `dnf` tool extracts dependency information from the source package's metadata, resolves direct/indirect dependencies, and installs all necessary dependencies. To meet the last two prerequisites, we fully build the package using the `rpmbuild` tool.

When the framework is used in conjunction with an analysis tool based on compiler front-end libraries, such as our library usage analysis tool, the compiler flags used to compile each C++ source file are also required to ensure accurate parsing of C++ code. Therefore, when invoking the `rpmbuild` tool, the framework utilizes the `bear` build wrapper introduced in Section 2.6. The `rpmbuild` tool runs the build tools specified in the source package's metadata. The `bear` wrapper intercepts the compile instructions and records the pathname of the compiled source file with the compiler flags used. The wrapper then relays compile commands to the actual compiler. After the build process, `bear` summarizes the recorded compile instructions and outputs a Clang-style compilation database file. Figure 4.1 illustrates the above mechanism.

For each package in the candidate list that may contain C++ code, the above types of information are collected before applying an analysis tool. In the next section, we present the four steps performed by the framework to gather the information for each package and apply the analysis tool.

4.4 Package Processing Steps

To collect the necessary information for each package and invoke an analysis tool, the framework performs four processing steps: prebuild, build, postbuild, and analysis. These procedures are

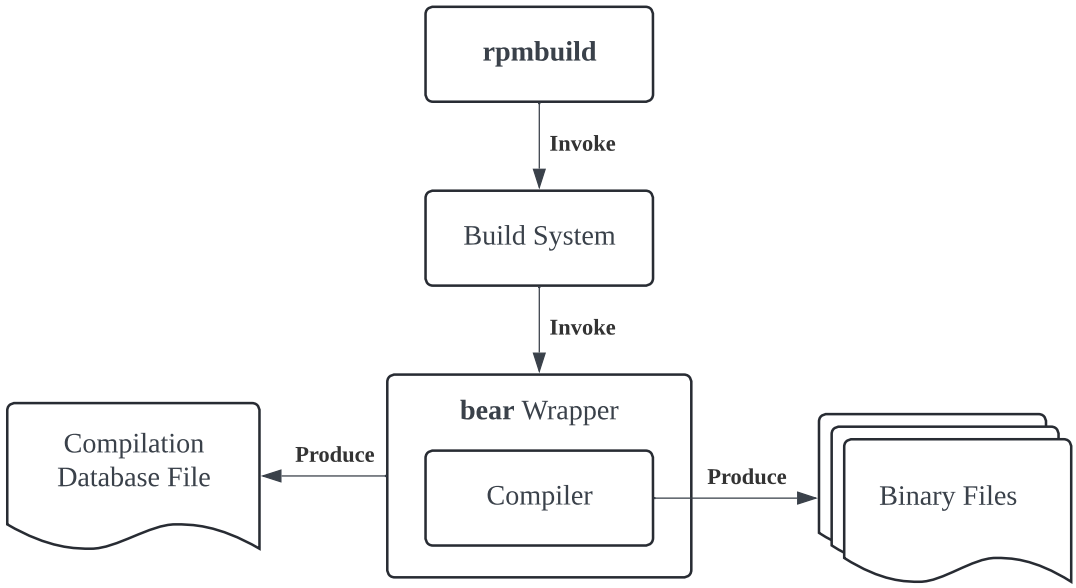

Figure 4.1: The `bear` tool wraps around the compiler to capture compiler flags.

executed in a Podman container instance to ensure that the processing of each package is consistent and reproducible. In what follows, we describe each step in detail.

1. **Prebuild.** A source package from the candidate package list is downloaded from the Fedora software repository using the `dnf` package manager. The package manager extracts the dependency information from the metadata of the source package and installs all necessary dependencies on the container filesystem. Next, the `rpmbuild` tool extracts and prepares the package content for the building process. All C++ source/header files from the package and the header files from the package's dependencies are available after the prebuild step.

2. **Build.** The build step is executed by the `rpmbuild` tool to compile the package's source files. The `rpmbuild` tool follows the instructions in the metadata of the source package to invoke the build tool chosen by the package creator. The `bear` wrapper intercepts the compile instructions sent by the build tool to the C++ compiler. When the build process finishes, `bear` outputs the compilation database that contains the list of compiled source files and the compiler flags used. All necessary information for the source code analysis of the package is available after this step.

3. **Postbuild.** Before invoking the analysis tool, the framework performs cleanup operations to reduce disk space consumption. We developed another Clang-based tool named `file_cleaner` to identify all C++ source files in the compilation database generated in the build step. The `file_cleaner` tool examines each of the compiled source files and finds all header files directly or indirectly included by the source files. An output list contains the path names of all C++ source/header files associated with the build process. The `file_cleaner` tool removes

all files in the package build directory that are not on the list. Typically, C++ source and header files do not consume the most disk space in a source package. Instead, resource files, files in other languages, binary files, and documentation usually occupy most of the disk space. Keeping only the related C++ source/header files reduces disk space consumption significantly, which is highly desirable when processing a large number of packages.

4. **Analysis.** In the analysis step, the framework runs the specified static analysis tool on each C++ source file listed in the compilation database generated during the build step. The author's supervisor, Michael Adams, developed a helper program that runs an arbitrary tool over each source file in a compilation database in parallel. The helper program is invoked with the specified analysis tool to reduce processing time.

When the above processing steps are finished, the Podman container instance is destroyed, deleting the container-specific filesystem with the installed dependencies. The package build directory and the output directory of the analysis tool are mounted as external directories to the container, which are not deleted. As a result, we retain the package's source code and analysis results for further exploration.

4.5 Framework Utilization and Output Structure

Having introduced the functionalities of the framework, we discuss the information that needs to be specified when using the framework and the output structure of the framework. In summary, the framework is a collection of complicated invocations of underlying tools (e.g., `dnf`, `rpmbuild`, `bear`, `file_cleaner`, and analysis tool). We specify some initial information when using the framework, and the framework generates the command-line options required by each tool.

The package selection process of the framework introduced in Section 4.2 is implemented as an independent script, which produces a list of candidate packages that may contain C++ code from the software repositories of a `dnf` based operating system. We start using the framework by invoking the script to turn on the fixed source repository of the target operating system and turn off other software repositories. The script then performs the package selection process and saves the list of candidate packages in a text file.

The candidate package list, the pathname of the analysis tool, and the pathname of the framework output directory are necessary information for using the framework. A subdirectory under the framework output directory is created for each package in the candidate list. The framework prepares each package for analysis and applies the specified analysis tool. Analysis outputs and log files generated by individual tools are stored in each package processing directory. Figure 4.2 illustrates the invocation of the framework and the output structure. After the framework finishes processing all packages in the candidate list, the analysis results and package source code are available for further exploration.

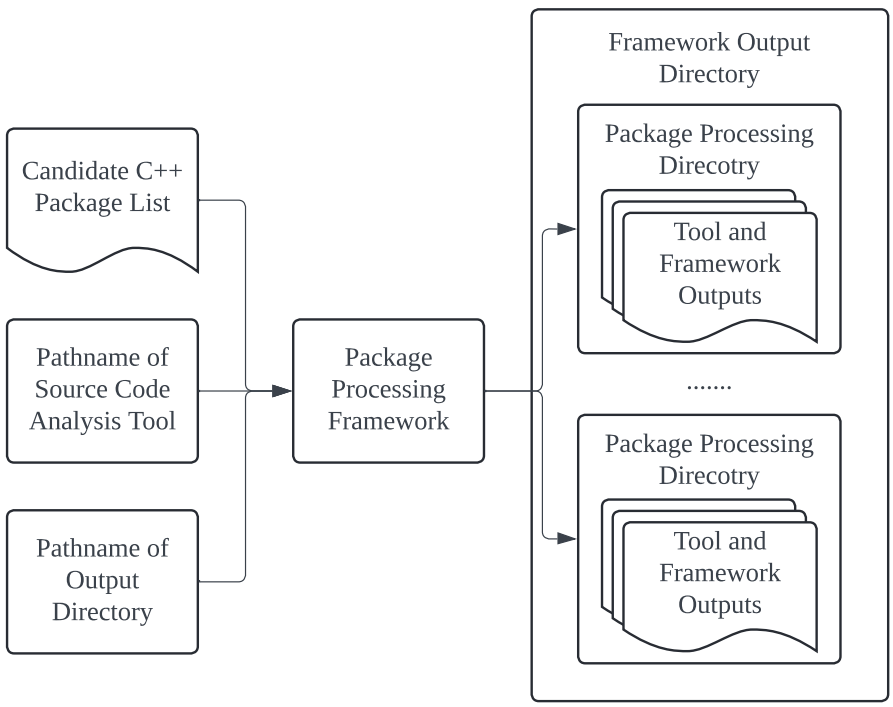

Figure 4.2: Using the package processing framework.

Chapter 5

Results of Applying Framework

5.1 Overview

Having introduced our library usage analysis tool and package processing framework in the previous chapter, we present results obtained by using the framework in conjunction with our analysis tool over the C++ codebase of Fedora Linux 37. The overall success of processing source packages is demonstrated. Reasons for failures to handle some packages are discussed with examples. We then present the characteristics of the analyzed C++ codebase to showcase the scale and variety of the considered C++ software. Finally, the chapter identifies several factors that should be considered when interpreting the results of this study.

5.2 Results of Package Processing

The previous chapters presented our library usage analysis tool and our framework that applies an arbitrary tool over a wide range of software packages. To evaluate the effectiveness of our work, we use our framework in conjunction with our analysis tool to process the source packages in the software repositories of Fedora Linux 37.

The fixed software repositories in the Fedora Linux 37 release contain approximately 23 030 source packages. Using the package selection criteria introduced in Section 4.2, 2 980 candidate packages are selected that may contain C++ source code when building the packages. Our framework processes all 2 980 packages. Based on the processing steps presented in Section 4.4, the outcomes produced by our framework when handling each package can be classified into seven mutually exclusive categories. Table 5.1 summarizes the processing outcomes. In Table 5.1, the notation $[x, y)$ denotes the half-open interval from x to y, including x but not including y. The analysis success rate is defined as the ratio of successfully analyzed C++ source files (those analyzed without error) to the total number of C++ source files in a package. We use a 50% threshold to determine whether the majority of a package's source files can be successfully analyzed.

Overall, Table 5.1 illustrates the success of our framework in processing source packages. Despite the complexities of building and managing dependencies in C++ projects, we successfully built and analyzed 2 725 packages, representing 88.62% of all packages considered. A total of 339

Table 5.1: Package processing outcomes

Outcome	Number of Packages	% of Packages
prebuild failed	17	0.57
build completed but failed	196	6.58
build failed due to hanging	31	1.04
postbuild failed	11	0.37
$[0\%, 50\%)$ analysis success	84	2.82
$[50\%, 100\%)$ analysis success	372	12.48
100% analysis success	2 269	76.14
total	2 980	100.00

packages failed at various stages of the framework, accounting for 11.38% of the packages. In what follows, we discuss the different types of processing failures, along with examples for each type:

1. **Prebuild failed.** The prebuild failure happens when some packages cannot be prepared for the build process. A sum of 17 packages (i.e., 0.57% of packages considered) fail at the prebuild step of the framework. The most common reason for prebuild failures is that a package's dependencies conflict with the minimal set of pre-installed dependencies in our Podman containers. When conflicting dependencies cannot be installed, the build process cannot be initiated, causing our framework to fail on these packages. For example, the package `botan2-0:2.19.1` requires a 32-bit version of OpenSSL, while a conflicting 64-bit version of OpenSSL is already installed in our Podman image. Another common situation is that a package is not designed for the x86-64 architecture and cannot be processed, such as the package `s390utils-2:2.23.0` developed for the s390 architecture.

2. **Build completed but failed.** This build failure occurs when the build process of a package completes with a non-zero exit status, indicating build errors. This type of failure happens to 196 packages (i.e., 6.58% of considered packages). For these packages, even if some build outputs were created, the outputs cannot be trusted. Packaging errors are a common reason identified for such build failures. For example, the package `dbus-qt3-0:0.9` requires a Qt version between 3.3 and 4.0. This dependency is not correctly stated in the package's metadata, causing a build failure. Another kind of build failure is due to software authors not addressing compile errors. In the C++ source code of `clazy-0:1.9`, some non-virtual methods are marked as `override`. These incorrect uses of the `override` keyword were only warned by some older versions of C++ compilers, but newer versions of C++ compilers produce compile errors and abort the compilation process. In conclusion, many of the 6.58% of packages are defective and do not build correctly even without our framework. Such problems should be fixed by the package maintainers or the software authors.

3. **Build failed due to hanging.** The build hanging failure is when a package's build process freezes before it can finish. This type of failure occurs with 31 packages (i.e., 1.04% of considered packages). Though extremely helpful to our framework, the `bear` build wrapper

unfortunately contains some bugs. When the `bear` wrapper is applied when building the 31 packages, the wrapper will freeze and fail to produce a compilation database.

4. **Postbuild failed.** The postbuild failure occurs when the build process of a package completes, but the produced compilation database is empty. This type of failure happens to 11 packages (i.e., 0.37% of considered packages). The `bear` wrapper may fail to capture the compilation commands during the build process and cause the postbuild failure.

5. **Analysis success rate less than 50%.** When the analysis success rate is less than 50%, we fail to analyze the majority of the C++ source files in the package, and we consider this situation another type of failure. Some source files are not intended to be compiled by `clang`, and our tool is based on the Clang libraries. If a source file cannot be compiled by the `clang` compiler, our analysis tool will also fail to parse the file and produce results. The command-line interface of the `clang` compiler is designed to be generally compatible with `gcc-c++`. The compatibility, however, is not always guaranteed. For example, the package `colobot-0:0.2.0` tries to suppress a compiler warning `-Wrestrict` using pragma directives in some of its source files. This warning is specific to `gcc-c++` and is not recognized by `clang`. Other than differences in warning types, `clang` also differs from other compilers in supported options, precompiled header formats, and strictness in enforcing language rules. As shown in Table 5.1, our analysis tool is mostly successful in handling C++ source code. Only 84 packages (i.e., 2.82% of considered packages) are successfully built but have less than half of their source code analyzed without error, likely due to their heavy reliance on compiler-specific features.

Table 5.1 shows that our framework is generally successful in processing source packages and applying the analysis tool. A total of 2 269 packages, which account for 76.14% of all considered packages, have all their C++ source files analyzed without error. Furthermore, 2 641 packages have the majority (i.e., equal to or more than 50%) of their source files analyzed without error, accounting for 88.62% of considered packages. An extensive collection of library usage instances is obtained from the built and analyzed packages, which enables us to identify real-world programming trends in C++.

Before proceeding further, it should be noted that two different implementations of the framework and the static analysis tool were developed to improve the consistency of the results of our study. The framework and the tool are both challenging to implement and prone to errors. To ensure accuracy, the author and the author's supervisor independently each implemented a version of the framework and the static analysis tool. We compared the processing output of thousands of packages and fixed many bugs in the implementations until the results of the two implementations were generally aligned. The supervisor's version collects additional supportive data, such as extra log files. Because the debugging process between the two implementations was time-consuming, the thesis project took longer than expected. As a result, we decided not to implement the extra logging features in the author's version and discuss the processing results from the supervisor's version in this thesis. It is important to emphasize, however, that analysis results from either implementation would not have been reliable without the thorough debugging of both implementations.

Table 5.2: Distribution of processed source code size for packages with at least one successfully analyzed C++ source file

Lines of Code	Number of Packages	% of Packages
$[10^0, 10^1)$	7	0.29
$[10^1, 10^2)$	6	0.25
$[10^2, 10^3)$	201	8.45
$[10^3, 10^4)$	645	27.11
$[10^4, 10^5)$	1 018	42.79
$[10^5, 10^6)$	448	18.83
$[10^6, 10^7)$	49	2.06
$[10^7, 10^8)$	5	0.21
total	2 379	100.00

5.3 Characteristics of Considered Codebase

Having examined the outcomes of C++ source code processing, we present the characteristics of the considered C++ codebase. For each package processed successfully in the prebuild, build, and postbuild steps, we compute the total number of lines of compiled C++ code of the package. This number is calculated by summing the number of lines of each compiled source file and header file included by at least one of the package's compiled source files. If a source file is compiled multiple times in the build process, or if a header file is included by multiple source files, the file will only contribute once to the total number of lines of C++ code. Throughout this thesis, when measuring the number of lines of code for a C++ file, we count the total number of lines in the file, which includes any comments and blank lines. We successfully process 398 065 762 lines of code from 339 786 C++ source files and 416 439 header files. Table 5.2 shows the distribution of the packages' sizes in terms of the number of lines of compiled C++ code. Only packages with at least one successfully analyzed C++ source file are included in the table.

Table 5.2 shows that the most common sizes of considered packages are within the range of $[10^4, 10^5)$ lines of code. A total of 346 packages, though successfully processed by the framework, have no analyzed C++ code. Several factors account for this situation. The package may falsely declare dependencies on C++ compilers but only contain C source code or require compiler features not supported by `clang`, as previously discussed. Another possibility is that the package is a header-only library with no C++ source files compiled in a default build that includes the package's headers. A tiny fraction of packages (i.e., 0.54%) have C++ code but are less than 100 lines. These packages are usually not primarily written in C++ but provide a C++ interface or are header-only libraries with minimal compiled test/demo source code. Table 5.3 shows the 20 packages with the most lines of C++ code in our codebase. These largest packages belong to the following types of software:

Table 5.3: 20 packages with the most lines of C++ code

Name of Source Package	Number of Lines
libint2-0:2.6.0	28 724 722
swig-0:4.0.2	14 987 941
godot-0:3.4.5	14 451 968
qt5-qtwebengine-0:5.15.10	12 959 278
cross-gcc-0:12.1.1	10 843 595
nextpnr-0:1-12.20220912gitf1349e1	10 763 584
root-0:6.26.06	7 421 955
libint-0:1.2.1	7 094 630
swift-lang-0:5.7	7 004 290
llvm9.0-0:9.0.1	6 662 584
llvm12-0:12.0.1	6 645 502
qgis-0:3.26.1	6 330 395
llvm8.0-0:8.0.1	6 146 641
llvm11-0:11.1.0	5 764 368
mingw-gcc-0:12.2.1	5 641 138
llvm10-0:10.0.0	5 334 565
vtk-0:9.1.0	5 195 544
llvm7.0-0:7.0.1	5 118 867
scummvm-0:2.6.0	4 199 865
qt-1:4.8.7	4 198 967

1. **Software development tools and libraries.** This category includes software such as LLVM, Cross-GCC, Swig, Swift, and MinGW-GCC. These packages are essential for modern software engineering. They provide the infrastructure for compiling, linking, and optimizing code, facilitating cross-platform development, and integrating various programming languages. These tools enhance code portability, streamline the development process, and improve the performance and reliability of software.

2. **Graphical libraries.** This category includes software such as Qt and VTK. These graphical libraries are crucial for creating and managing graphical user interfaces and visualizations in software development. They provide the tools and frameworks needed to design, render, and manipulate graphical elements, enabling the creation of visually appealing and interactive applications across various platforms. These libraries are widely used in commercial software development, scientific visualization, and other graphical applications.

3. **Scientific and engineering tools.** This category includes software such as LibInt, Nextpnr, QGis, and Root. The functionalities of the scientific and engineering tools include quantum chemistry computations, circuitry design, geographic information systems, and data analysis frameworks. These tools encapsulate complex computations, streamline workflows, and support advanced analysis and visualization, making them invaluable in their respective fields. Notably, the LibInt libraries contain a significant amount of machine-generated code, contributing to the high number of lines in the libraries.

4. **Game Engines.** This category includes packages such as ScummVM and Godot. Game engines are software frameworks that provide features such as rendering, audio management, and physics simulations for game development. The performance and flexibility of C++ make it a popular choice in the gaming industry.

The above categorization aids in understanding the composition of our codebase and highlights the diverse range of software considered. Beyond the larger packages, the codebase contains a variety of smaller software projects. Consequently, various C++ programming techniques are likely present in the codebase, from widely adopted to more specialized practices. By applying analytical tools to this codebase, we can identify trends in the programming choices made by C++ developers,

5.4 Threats to Validity

Although our C++ codebase surpasses those in previous studies in both scale and diversity, several factors should be considered when interpreting the results of this study. These factors could potentially affect the codebase's ability to accurately represent real-world C++ programming practices, influencing the conclusions and recommendations drawn from the analysis.

We begin by discussing scenarios where some C++ code is excluded from our analysis. The excluded code may utilize C++ libraries differently from the rest of the codebase. Consequently, the observed library usage trends may differ from results obtained by analyzing all C++ code in the codebase. Certain types of C++ code exclusions were previously discussed, such as header-only libraries with minimal or no compiled C++ source code and C++ source files that fail to compile with `clang`. Our analysis tool does not process such C++ code. Additionally, some C++ code may not be compiled in a default build. For instance, in the Qt package, specific network protocols, profilers, and third-party components are turned off by default. Library usage instances within these disabled components are also excluded from our analysis results.

Another factor to consider is the presence of multiple versions of some libraries in the codebase. Some library packages offer different versions simultaneously to support backward compatibility and provide evolving feature sets. Similar code in different versions of the same library, however, all contribute to the analysis results, causing observed programming trends to bias towards the libraries with multiple versions. Additionally, libraries that adopt such versioning strategies are often large projects with more lines of C++ code, further skewing the analysis results. The LLVM libraries are a good example. Because there are multiple versions of LLVM libraries in the codebase, and each version contains a large amount of C++ code, the analysis results may reflect more on the LLVM developers' programming choices.

Furthermore, the codebase shows a strong presence of graphical and scientific packages. These particular categories of software may utilize certain C++ libraries more frequently than other software types, causing the analysis results to emphasize these libraries. For example, the Qt libraries are widely adopted by many graphical software packages, and the analysis results show high usage counts for various Qt library entities. Although the prominence of these libraries reflects the real-world demand, such an emphasis may not be desirable for some kinds of analysis.

In addition, machine-generated code, as seen in the LibInt package and other packages, is not excluded from the codebase. We consider machine-generated code to be valid C++ code, as its practicality in specific contexts is indispensable, such as aiding in complex tests for scientific libraries. Library usage behaviors of machine-generated code, however, may differ from human-written code. Therefore, the analysis results may deviate from the results obtained by analyzing a codebase without machine-generated code.

Lastly, we do not perform any kind of dynamic analysis in our work. If a library item is referenced frequently in C++ source code, the item does not necessarily execute often or at all when the C++ applications that include it are run. In situations where runtime occurrence frequency is more important, the results obtained using our approach might not be as applicable.

The factors discussed above do not diminish our codebase's ability to represent real-world C++ programming practices. Many of these factors have a mild impact. For example, in most software packages, the analysis tool successfully processed the majority of the C++ source files. Thus, the C++ code exclusion because of source files incompatible with `clang` is limited. Additionally, some factors reflect the real-world complexities of software development, such as the presence of machine-generated code in some packages. While these factors need to be considered when evaluating analysis results and making C++ programming recommendations, they do not undermine the overall reliability of our findings.

Chapter 6

Results of Library Usage Analysis

6.1 Overview

After applying our library usage analysis tool with our package processing framework, we obtained extensive raw data for C++ library usage, totalling 341 181 165 instances. Further extraction and summarization are necessary to derive meaningful conclusions from the raw data. This chapter quantifies the popularity of different library items. Some useful measures of popularity are extracted from the raw data. After that, we analyze various aspects of library usage and provide valuable insights.

6.2 Quantifying Library Usage

The previous chapter demonstrated the successful processing of the C++ codebase of Fedora Linux 37 by our framework and library usage analysis tool. This chapter explores various aspects of C++ library utilization within the considered codebase.

Before discussing each analysis topic, we introduce several measures for comparing the popularity of different library items. These measures are specifically designed to mitigate the impact of large software packages, such as the LLVM and Qt packages, on the analysis results. These measures provide an accurate overview of C++ library utilization practices across the codebase. In contrast, emphasizing measures such as raw item occurrence counts could cause the trends observed to disproportionately reflect the preferences of authors of larger projects.

First, we introduce the **occurrences per line of code (OPLOC)** measure. For a particular library item used in a package, we define its OPLOC f as $f = c/s$, where c is the number of instances of the item being used in the user code that is processed for the package, and s is the total number of lines of code in the C++ source files and user header files encountered during the processing of the package. A header file is counted at most once, even if included in multiple places in the source code.

For a particular library item, we define the **average OPLOC** of this item as the arithmetic mean of the OPLOC values for the item taken across all packages. The average OPLOC provides a measure of the overall popularity of the library item across the C++ codebase under consideration.

Sometimes, a library item may not be the most frequently used per line of C++ code as measured by the average OPLOC measure, but the item is used by a large number of packages. Therefore, we also consider the **fraction of packages** using an item. In future studies, more sophisticated measures could be employed to examine additional characteristics of library usage patterns.

6.3 Top-Level Namespaces Analysis

The first aspect of C++ library usage investigated is top-level namespaces. In C++ programming, namespaces play a crucial role in preventing name collisions in large projects and when multiple libraries are included, enhancing code maintainability. A top-level namespace is simply the highest level of a namespace hierarchy. Many C++ libraries organize their components into namespaces, with the top-level namespace often corresponding to the library itself. By examining the most frequently used top-level namespaces, we can gain insights into the preferred C++ libraries in software development. These findings can guide programmers in selecting appropriate libraries for their projects and help library developers assess the adoption of their work.

From our raw data, we extracted the top-level namespace of used library items across the 2 379 packages that contain C++ code. For each instance of C++ library usage within a software package, we check whether the top-level namespace is a programmer-specified namespace, an anonymous namespace, or the global namespace. We then summarize the usage of all top-level namespaces within each package to generate package-level OPLOCs. The arithmetic mean of the package-level OPLOCs is calculated to determine the average OPLOC for each identified namespace. Additionally, the fraction of packages where each top-level namespace appears is calculated. A total of 699 unique top-level namespaces were identified.

Table 6.1 lists the most frequently used top-level namespaces, ranked by average OPLOC, and includes the fraction of packages where each namespace appears. From Table 6.1, we can see that the global namespace is heavily used with the highest average OPLOC of $2.41 \cdot 10^{-1}$, and appears in 94.07% of all packages. We also verified that 90.96% of the used library items under the global namespace are constructors and methods, which means that these items mostly come from C++ libraries rather than C libraries. The C++ libraries where these items belong have the option to manage entities under programmer-specified namespaces but choose to use the global namespace instead. Such reliance on the global namespace may increase the risk of name conflicts. We recommend that C++ libraries organize more items into programmer-specified namespaces to improve code maintainability.

Next, we will examine some other popular top-level namespaces in Table 6.1. The `std` namespace of the C++ standard library is the most frequently used with an average OPLOC of $2.35 \cdot 10^{-1}$. The popularity of the `std` namespace is to be expected due to its rich collection of features and its integral role in the C++ language. The 12.78% of packages that do not use the `std` namespace are typically very small packages that are not written primarily in C++ and contain little C++ code. Among third-party library namespaces, the `boost` namespace of the Boost libraries is the most used and has an average OPLOC of $1.17 \cdot 10^{-2}$. The Boost and the C++ standard libraries are both general utility libraries. The high frequency of Boost usage suggests that incorporating some of the functionalities provided by Boost libraries (but not the C++ standard library) into the C++

Table 6.1: Usage statistics for the most popular top-level namespaces of used library items

Namespace Name	Average OPLOC	Appear in % of Packages
`(global namespace)`	$2.41 \cdot 10^{-1}$	94.07
`std`	$2.35 \cdot 10^{-1}$	87.22
`boost`	$1.17 \cdot 10^{-2}$	11.85
`fcitx`	$9.88 \cdot 10^{-3}$	0.71
`llvm`	$9.06 \cdot 10^{-3}$	0.80
`testing`	$6.99 \cdot 10^{-3}$	2.86
`Catch`	$6.63 \cdot 10^{-3}$	0.97
`google`	$3.87 \cdot 10^{-3}$	2.10
`Glib`	$3.58 \cdot 10^{-3}$	2.98
`doctest`	$2.16 \cdot 10^{-3}$	0.42
`Gtk`	$2.11 \cdot 10^{-3}$	2.14
`folly`	$1.77 \cdot 10^{-3}$	0.34
`Qt`	$1.31 \cdot 10^{-3}$	18.24
`Eigen`	$1.04 \cdot 10^{-3}$	1.43
`sigc`	$8.62 \cdot 10^{-4}$	3.11
`pybind11`	$7.92 \cdot 10^{-4}$	1.09
`libebml`	$7.67 \cdot 10^{-4}$	0.08
`librevenge`	$7.48 \cdot 10^{-4}$	0.88
`apache`	$7.32 \cdot 10^{-4}$	0.13
`spv`	$6.53 \cdot 10^{-4}$	0.13
`(anonymous namespace)`	$9.35 \cdot 10^{-5}$	1.09

Table 6.2: Categorization of the software types of the most popular top-level namespaces

Software Category	Namespace Name
General purpose	`std, boost, folly`
Graphical interface and computing	`Qt, Gtk, Glib, spv`
Software testing	`testing, Catch, doctest, google`
Media and file formats	`libebml, librevenge`
Compiler and toolchain	`llvm`
Language interoperability	`pybind11`
Web and server	`apache`
Scientific	`Eigen`
Input method	`fcitx`
Event handling	`sigc`

standard library would likely be highly beneficial. Another noticeable namespace is `Qt` from the Qt graphical library. This namespace appears in an even larger fraction of packages than `boost`, but its average OPLOC ranks lower in Table 6.1. This discrepancy is due to the fact that the Qt library often does not manage items using namespaces. The importance of the Qt library is reflected in its presence in a large fraction of the considered packages, indicating the significance of graphical applications in C++ software development.

We then categorize the libraries to which the most popular top-level namespaces belong into different software types, as shown in Table 6.2. The software types of the most used libraries are highly diverse. Besides utility and graphical libraries, the software types include scientific computing, media formatting, and network/web libraries. This diversity reflects the versatility of the C++ language, demonstrating its capability to satisfy various software development needs.

6.4 Boost Second-Level Namespace Analysis

The previous section discussed that the Boost libraries are the most popular general-purpose libraries besides the standard library, and the standard library might benefit from incorporating some of the pioneering features that Boost provides. The Boost libraries are a collection of individual libraries, each providing a specific set of functionalities. While many of the Boost library items are placed under the top-level `boost` namespace, the Boost libraries sometimes group related functionalities under a second-level namespace. By studying the usage of these second-level namespaces, we can identify some of the most popular Boost libraries. Then, we can investigate the implementation status of the counterparts of these Boost libraries in the C++ standard library to identify potential areas for improvement of the standard library.

For each instance of library use, if the library entity is under the `boost` top-level namespace, we examine whether the entity has a second-level namespace. If a second-level namespace is present, the name of the namespace represents the particular Boost library used. Then, we calculate each second-level namespace's average OPLOC and the fraction of packages where the namespace appears.

Table 6.3: Most frequently used Boost second-level namespaces

Boost Second-Level Namespace	Average OPLOC	Appear in % of Packages
`unit_test`	$1.58 \cdot 10^{-3}$	1.77
`test_tools`	$9.57 \cdot 10^{-4}$	1.72
`python`	$5.60 \cdot 10^{-4}$	0.84
`program_options`	$3.46 \cdot 10^{-4}$	2.61
`mpl`	$1.83 \cdot 10^{-4}$	0.55
`asio`	$1.73 \cdot 10^{-4}$	1.05
`spirit`	$1.71 \cdot 10^{-4}$	1.35
`filesystem`	$1.36 \cdot 10^{-4}$	3.11
`property_tree`	$1.21 \cdot 10^{-4}$	1.22
`container`	$8.86 \cdot 10^{-5}$	0.92

Table 6.4: Implementation status of the C++ standard library counterparts of popular Boost libraries

Implemented in C++ Standard Library?	Boost Libraries
Yes	`filesystem`
Partial	`mpl`, `asio`, `spirit`, `container`, `property_tree`
No	`unit_test`, `test_tools`, `python`, `program_options`

Table 6.3 lists the most frequently used second-level namespaces under `boost`, ranked by average OPLOC. From Table 6.3, we can see that the most popular Boost libraries provide functionalities such as software testing, Python interoperability, and option parsing. The implementation status of the C++ standard library counterparts of Boost libraries can be categorized into several groups, as shown in Table 6.4. Among the most popular Boost libraries, only the counterpart of `filesystem` is fully implemented in the C++ standard library. Standard library counterparts of other popular Boost libraries, such as `mpl` for template meta-programming and `asio` for asynchronous I/O, provide fewer functionalities. Notable features commonly needed by programmers, such as `unit_test` for software testing and `python` for Python interoperability, are not supported in the C++ standard library.

The C++ standard library is cautious about adopting new features to ensure backward compatibility, maintain reliability, and preserve performance. New features must integrate well with existing code, address significant user demand, and be feasible to implement across different platforms. Therefore, the incorporation of Boost features into the C++ standard library should be carefully considered. A prudent approach is to start with the most popular Boost libraries, such as `unit_test` and `python`. By focusing on the most utilized Boost libraries, the C++ standard library can meet the active demands of C++ software developers. Additionally, because these libraries have been widely applied and tested in real-world software packages, the standard library can benefit from proven implementations, maximizing the chances of successful integrations.

6.5 Standard Library Containers Analysis

The next analysis of library usage that we performed relates to the C++ standard library containers. The C++ standard library containers provide class templates for data collections. These containers are essential for efficient data handling, enabling programmers to concentrate on implementing complex logic rather than managing data storage details. Analyzing the use of standard library containers by C++ developers yields valuable insights, such as finding the most popular and essential containers. Understanding the uses of standard library containers also helps prioritize the learning and integration of key containers into development practices. Additionally, identifying usage patterns aids in making informed decisions when selecting an appropriate standard library container for a specific task.

We extracted all usages of the containers, including constructors, types, type alias, and methods. Different kinds of usages of each container across all software packages are summed together when calculating our popularity measures. All the standard library container types are templated. The template arguments are ignored, and different instantiations of a container type are treated as uses of the same container.

Table 6.5 lists the use statistics for standard library containers. Examining the results in Table 6.5, we can see that the `vector` is the most frequently used container type by a large margin, followed next by the `map` and `array` container types. We recommend that new C++ developers start learning the most-used containers first because of the importance of these containers in real-world C++ projects. Additionally, an often recommended programming practice is that the use of the `array` type be preferred over the built-in array types. The fact that `array` did not rank higher than the third most used, however, suggests that this practice is frequently not followed. Consequently, C++ developers may benefit from an improved awareness of the advantages of the `array` type.

Table 6.5 also shows that for associative containers, the ordered variant of a container (e.g., `map`) is consistently more frequently used than the unordered variant (e.g., `unordered_map`). Similarly, the unique-key variant of a container (e.g., `map`) is consistently more frequently used than the corresponding non-unique-key variant of a container (e.g., `multimap`). Knowing which container types are most frequently used is potentially helpful to guide the selection of new container types in future versions of the C++ standard library.

Additionally, Table 6.5 provides insights into the utilization of pioneering and outdated features. For the C++ codebase from the Fedora Linux 37 software repositories, the `pmr` containers (e.g., `pmr::map`) introduced in C++17 are still relatively new. The `pmr` containers provide more flexible memory management. Although `pmr` containers only appear in a few packages, their presence indicates that such memory management features are helpful for some C++ software. Conversely, the use of the `tr1` containers is worrying. The `tr1` containers are part of an early collection of proposed library extensions for C++. These containers are outdated and superseded by their C++11 (and later) equivalents long ago. We recommend that C++ developers replace the `tr1` containers with the modern equivalents provided by the C++ standard library.

The C++ standard library also provides container adapters, which are classes that provide a specific interface and behavior on top of existing containers. We summarized the uses of container adapters separately, as shown in Table 6.6. Table 6.6 demonstrates the prevalent use of container

Table 6.5: Usage frequency table of the C++ standard library containers

Container Name	Average OPLOC	Appear in % of Packages
vector	$3.17 \cdot 10^{-2}$	64.40
map	$5.74 \cdot 10^{-3}$	47.50
array	$3.50 \cdot 10^{-3}$	17.44
list	$2.43 \cdot 10^{-3}$	32.37
set	$1.51 \cdot 10^{-3}$	32.32
unordered_map	$1.06 \cdot 10^{-3}$	17.15
deque	$5.54 \cdot 10^{-4}$	17.78
unordered_set	$2.62 \cdot 10^{-4}$	10.76
multimap	$2.02 \cdot 10^{-4}$	12.69
forward_list	$4.13 \cdot 10^{-5}$	2.44
unordered_multimap	$4.05 \cdot 10^{-5}$	3.19
multiset	$2.86 \cdot 10^{-5}$	3.83
unordered_multiset	$2.59 \cdot 10^{-6}$	0.59
__cxx1998::vector	$2.24 \cdot 10^{-6}$	0.08
tr1::unordered_set	$2.04 \cdot 10^{-6}$	0.25
tr1::unordered_map	$1.95 \cdot 10^{-6}$	0.55
__cxx1998::map	$1.40 \cdot 10^{-6}$	0.08
__cxx1998::unordered_map	$2.69 \cdot 10^{-7}$	0.04
tr1::array	$1.24 \cdot 10^{-7}$	0.04
__cxx1998::unordered_set	$6.72 \cdot 10^{-8}$	0.04
__cxx1998::list	$6.20 \cdot 10^{-8}$	0.04
__cxx1998::set	$5.68 \cdot 10^{-8}$	0.04
__cxx1998::deque	$4.91 \cdot 10^{-8}$	0.04
__cxx1998::multimap	$7.75 \cdot 10^{-9}$	0.04
pmr::unordered_set	$3.59 \cdot 10^{-9}$	0.08
pmr::map	$2.14 \cdot 10^{-10}$	0.04

Table 6.6: Usage frequency table of the C++ standard library container adapters

Container Adapter	Average OPLOC	Appear in % of Packages
`stack`	$2.77 \cdot 10^{-4}$	13.91
`queue`	$7.75 \cdot 10^{-5}$	12.19
`priority_queue`	$2.28 \cdot 10^{-5}$	5.88

adapter types from the C++ standard library. Notably, the `stack` and `queue` adapters exhibit higher average OPLOC and appear in more packages than some container types. This data suggests that despite container adapters being wrapper types for underlying containers, the cognitive benefits of using container adapters are valued by C++ developers. For instance, the `stack` adapter, typically a wrapper for the `deque` container, allows programmers to concentrate on managing a last-in-first-out data structure without worrying about the additional functionalities provided by the underlying `deque` container. We conclude that container adapters are often helpful, and C++ developers may benefit from better understanding and utilizing these adapters.

6.6 Standard Library Algorithms Analysis

The next library usage analysis that we performed focuses on the C++ standard library algorithms. The algorithm library, part of the C++ standard library, offers various functions for performing operations like searching, sorting, counting, manipulating, and examining data collections. These algorithms are rigorously optimized and tested, providing efficient and reliable solutions for common programming tasks. Despite its utility, the algorithm library is often underused due to the learning cost of its extensive API. Understanding the utilization patterns of the algorithm library benefits both C++ developers and contributors to the C++ standard. For developers, recognizing the most frequently used functions within the algorithm library may encourage more effective and extensive use of the library. For those involved in developing and maintaining the C++ standard, identifying algorithms with minimal usage can offer insights into potential areas for deprecation. The deprecation may streamline the standard API, making it more accessible and easier to maintain.

We extracted all uses of the standard library algorithms from our codebase by examining the namespace, function names, and function parameters. First, a list of function names that belong to the algorithm library was created by referencing the C++ standard draft document. For each instance of library use, we checked whether the library entity is under the `std` namespace and whether the function name is in our list. C++ supports function overloading, which means that multiple functions can have the same name but serve different purposes. For example, the `move` function in the C++ standard library has two overloads: one that helps with type conversion and comes from the standard utility library, and another that moves a range of elements between two containers and comes from the algorithm library. To ensure that a called function belongs to the algorithm library, we checked its parameter list and excluded function overloads not related to the algorithm library.

Table 6.7 lists the most frequently used functions in the C++ standard algorithm library, ranked

Table 6.7: Most frequently used functions in the C++ standard algorithm library

Function Name	Average OPLOC	Appear in % of Packages
`min`	$7.27 \cdot 10^{-4}$	40.31
`max`	$6.00 \cdot 10^{-4}$	37.24
`copy`	$1.75 \cdot 10^{-4}$	22.36
`sort`	$1.16 \cdot 10^{-4}$	34.59
`transform`	$9.19 \cdot 10^{-5}$	20.39
`find`	$9.03 \cdot 10^{-5}$	29.80
`find_if`	$7.49 \cdot 10^{-5}$	20.47
`equal`	$5.12 \cdot 10^{-5}$	10.51
`fill`	$4.82 \cdot 10^{-5}$	12.90
`for_each`	$3.41 \cdot 10^{-5}$	12.15
`fill_n`	$3.15 \cdot 10^{-5}$	7.02
`copy_n`	$2.25 \cdot 10^{-5}$	3.45
`reverse`	$2.18 \cdot 10^{-5}$	13.37
`move`	$2.15 \cdot 10^{-5}$	4.96
`lower_bound`	$2.08 \cdot 10^{-5}$	12.40

by average OPLOC. In Table 6.7, we can see that the most popular algorithms are those with straightforward and commonly demanded purposes, such as `min`, `max`, `copy`, and `sort`. The `min` and `max` algorithms, in particular, appear in 40.31% and 37.24% packages, respectively, showcasing their widespread application. The `transform`, `find`, and `find_if` algorithms are the next most frequently occurring in C++ source code and are also used by a relatively high fraction of packages (i.e., over 20%). We recommend emphasizing the most frequently used algorithms in C++ learning materials.

Table 6.8 lists the least frequently used functions in the algorithm library. In Table 6.8, we can see that rarely used algorithms often serve specialized purposes. For example, the function `set_symmetric_difference` performs a symmetric difference operation on two sorted ranges, a less common task in C++ software development. As a result, this algorithm has an average OPLOC of $1.02 \cdot 10^{-7}$, which means it appears only once in approximately every 10 million lines of C++ code. Similarly, algorithms designed for combined operations are also less frequently used. The `rotate_copy` function, for instance, combines the `rotate` and `copy` functions. The combination of operations may aid in performance optimization but is seldom utilized by C++ developers. Table 6.8 shows that the average OPLOC of `rotate_copy` is only $5.98 \cdot 10^{-8}$.

We recommend considering the deprecation of the least frequently used algorithms to streamline the C++ standard algorithm library. If deprecation is deemed too impactful, an alternative suggestion is to be more cautious in the future expansion of the C++ standard algorithm library. Potential additions to the algorithm library, particularly specialized or combined-operation algorithms, should be carefully examined to ensure they are essential. Notably, in Table 6.8, the lowest usage of the `sample` algorithm is likely due to its recent introduction in C++17 when Fedora Linux 37 was released. Therefore, our comment does not apply to this algorithm.

Table 6.8: Least frequently used functions in the C++ standard algorithm library

Function Name	Average OPLOC	Appear in % of Packages
`partition_point`	$1.41 \cdot 10^{-7}$	0.92
`set_symmetric_difference`	$1.02 \cdot 10^{-7}$	0.34
`find_end`	$5.54 \cdot 10^{-8}$	0.50
`prev_permutation`	$5.42 \cdot 10^{-8}$	0.17
`unique_copy`	$5.00 \cdot 10^{-8}$	0.92
`replace_copy_if`	$4.72 \cdot 10^{-8}$	0.13
`partial_sort_copy`	$3.15 \cdot 10^{-8}$	0.38
`remove_copy`	$8.86 \cdot 10^{-9}$	0.25
`rotate_copy`	$5.98 \cdot 10^{-9}$	0.08
`partition_copy`	$5.66 \cdot 10^{-9}$	0.21
`search_n`	$3.98 \cdot 10^{-9}$	0.04
`is_sorted_until`	$3.98 \cdot 10^{-9}$	0.04
`replace_copy`	$2.24 \cdot 10^{-9}$	0.08
`is_partitioned`	$1.99 \cdot 10^{-9}$	0.04
`sample`	$9.56 \cdot 10^{-10}$	0.04

6.7 Bounds-Checking When Indexing Analysis

The next topic to consider is how C++ programmers use indexing methods with built-in bounds checking. C++ is often criticized for not performing bounds checking by default for all potentially unsafe indexing operations on containers, such as `std::vector`, with the provision for an opt-out for performance reasons. This philosophy towards bounds checking makes out-of-bounds accesses a more serious issue in C++ (i.e., undefined behavior) than in other languages that have well-defined semantics for out-of-bounds accesses. Thus, we investigated bounds checking for container/view types in the C++ standard library that can be indexed using potentially unsafe interfaces.

In the case of the `vector`, `deque`, `basic_string`, `array`, and `basic_string_view` container/view types, two different interfaces for indexing are provided:

1. The subscripting operator, `operator[]`, does not perform bounds checking, leading to undefined behavior in cases of out-of-bounds access.

2. The `at` method includes bounds checking and has well-defined behavior, throwing an exception if out-of-bounds access occurs.

For each of the five container/view types under consideration, if a package uses at least one of the indexing operations, the percentage of `operator[]` usage of both indexing operations is calculated. These percentages are then averaged across all packages using each container/view type, yielding a mean `operator[]` usage rate for each type, as shown in Table 6.9. In this table, we can see that the `operator[]` method, which lacks built-in bounds checking, is predominantly used for indexing operations with the container/view types. C++ programmers utilize `operator[]`

Table 6.9: Fraction of indexing operations without built-in bounds checking when using sequential container/view types

Container/View Type Name	% of Indexing Operations
`array`	95.54
`basic_string`	91.96
`vector`	91.69
`deque`	90.91
`basic_string_view`	87.52

Table 6.10: Fraction of packages using only indexing operations without built-in bounds checking when using sequential container/view types

Container/View Type Name	% of Packages
`array`	86.80
`basic_string_view`	85.42
`vector`	77.20
`basic_string`	74.57
`deque`	63.01

in 87.52% to 95.54% of all indexing operations, favoring it over the `at` method. This preference is especially pronounced with `array`, likely due to its compile-time size determination, which simplifies manual bounds checking for programmers. In any case, this reliance on `operator []` may increase the risk of out-of-bounds errors when programmers neglect to perform manual checks, potentially leading to crashes and security vulnerabilities.

Furthermore, we summarized the fraction of packages that exclusively use `operator[]` without using any built-in bounds checking when performing indexing operations, as shown in Table 6.10. Table 6.10 reveals that a significant portion of packages (ranging from 63.01% to 86.80%) never uses the `at` method for indexing operations when using each container/view type. This finding highlights a greater risk of out-of-bounds errors. When a package mixes the use of `operator[]` and the `at` method, the developers may be mindful of bounds checks, using `operator[]` for convenience and performance reasons. When a package exclusively uses `operator []`, however, it is possible that the package developers are unaware of out-of-bounds errors and may not perform necessary checks.

Based on our findings, we recommend that the C++ standard library consider incorporating bounds checking into the `operator[]` methods of sequential data types, while also providing options to explicitly disable bounds checking when necessary. This approach would effectively promote safer bounds checking practices with sequential containers and view types in C++. Additionally, we suggest that C++ developers prioritize the use of the `at` method for indexing operations in sequential containers and views. Educational materials for C++ should highlight the risks of out-of-bounds errors and the advantages of using the `at` method. These measures would improve the safety and reliability of C++ software, addressing a critical area for enhancement within the language.

Table 6.11: Fraction of packages using smart pointer APIs from the C++ standard library or Boost libraries

Smart Pointer Name	Appear in % of Packages
`std::unique_ptr`	31.57
`std::shared_ptr`	22.07
`std::weak_ptr`	6.35
`std::auto_ptr`	4.12
`boost::shared_ptr`	3.15
`boost::scoped_ptr`	1.43
`boost::intrusive_ptr`	0.80
`boost::intrusive_ptr`	0.76
`boost::weak_ptr`	0.50
`std::tr1::shared_ptr`	0.38
`boost::interprocess::scoped_ptr`	0.04
`std::tr1::weak_ptr`	0.04

6.8 Smart Pointer Analysis

The next topic of analysis is the usage of smart pointers in C++ software development. Smart pointers manage the lifetime of dynamically allocated objects. Unlike raw pointers, smart pointers automatically handle memory de-allocation when the object is no longer in use, preventing memory leaks and dangling pointers. By using smart pointers, developers can focus more on the logic of their software rather than on the intricacies of memory management. Understanding the adoption of smart pointers helps guide the programming practices of C++ developers and promotes the use of modern C++ features.

We examine whether each software package utilizes any APIs of smart pointers from the C++ standard library and Boost libraries. If a package refers to types, type aliases, methods, or constructors of a smart pointer, the smart pointer is considered to be used in that package. Table 6.11 shows the fraction of packages adopting each smart pointer. In Table 6.11, we find that `std::unique_ptr` and `std::shared_ptr` are the most popular smart pointers, appearing in 31.57% and 22.07% of all packages, respectively. Additionally, `std::weak_ptr` is widely used, appearing in 6.35% of all packages. Although Boost smart pointers, such as `boost::shared_ptr` and `boost::scoped_ptr`, were implemented earlier, they are less popular than their C++ standard library counterparts. This suggests that the smart pointers from the C++ standard library have effectively learned from earlier implementations and are well-received by C++ developers. The `std::auto_ptr` is still used in 4.12% of all packages, despite its deprecation from the C++ standard library in C++11. Smart pointers under the `std::tr1` namespace are also outdated. Similar to our findings in Section 6.5, some C++ developers continue to use outdated features. These developers should make an effort to update their code to utilize modern supported features.

Additionally, we summarized the fraction of packages using supporting utility functions for smart pointers from the C++ standard library and Boost libraries in Table 6.12. By comparing Table 6.12 and 6.11, we can see that although `std::unique_ptr` appears in more packages than

Table 6.12: Fraction of packages using supporting utility functions for smart pointers from the C++ standard library or Boost libraries

Smart Pointer Helper Function Name	Appear in % of Packages
`std::make_shared`	16.23
`std::make_unique`	11.22
`std::enable_shared_from_this`	4.62
`boost::make_shared`	1.01
`boost::enable_shared_from_this`	0.50
`std::allocate_shared`	0.17
`boost::make_unique`	0.13

Table 6.13: Fraction of packages using smart pointer functionalities from the C++ standard library and/or Boost libraries

Use Smart Pointers From	% of Packages
C++ standard library	38.55
C++ standard library and/or Boost	40.56

`std::make_unique`, the helper function `std::make_unique` for `std::unique_ptr` is used in 11.22% of all packages, lower than the 16.23% of packages using `std::make_shared` for `std::shared_ptr`. This inversion of popularity suggests that C++ developers may underutilize the helper function for `std::unique_ptr`, possibly because `std::make_unique` was introduced in C++14, which is later than the introduction of `std::unique_ptr` in C++11. These helper functions combine memory allocation and object construction into a single step, preventing memory leaks if an exception is thrown during object construction. We recommend that C++ developers use smart pointer helper functions to enhance their coding practices.

We combine the usages of smart pointers and supporting utility functions to provide an overview of the fraction of packages adopting any smart pointer functionalities. The result is shown in Table 6.13. Only 38.55% of packages use smart pointer features provided by the C++ standard library. This number increases slightly to 40.56% when considering offerings from both the C++ standard library and Boost libraries. This result is potentially concerning, suggesting that many packages may not utilize smart pointers. While some packages may implement their own custom smart pointers, it is likely that a significant number of packages do not use smart pointers at all. This indicates a need for better education among C++ developers about the advantages of using smart pointers.

6.9 Tuple and Pair Analysis

The next aspect of library usage that we consider is the use of the `tuple` and `pair` types from the C++ standard library. The `tuple` type from the C++ standard library allows the grouping of multiple objects of potentially different types into a single object, enabling the storage and

Table 6.14: Number of constructed `std::tuple` and `std::pair`

Container Type	Use Count	% of Uses
`std::pair`	2 974 716	85.38
`std::tuple`	509 392	14.62

Table 6.15: Number of elements in constructed `std::tuple` objects

Number of Elements	Use Count	% of Uses
0	17 610	3.46
1	79 562	15.62
2	200 486	39.36
3	139 613	27.41
4	61 970	12.17
$[5,8)$	8 985	1.76
$[8,16)$	1 119	0.22
$[16,32)$	42	below 0.01
$[32,64)$	4	below 0.01
$[64,128)$	1	below 0.01

manipulation of a heterogeneous collection of elements. A `pair` is a specific case of a `tuple` that stores exactly two elements. By observing how `tuple` and `pair` are used, we can learn the general practice of C++ developers and aid programmers in making informed decisions when using these types. Also, knowing the distribution of tuple element counts can help implementers of the `tuple` class to know how much they should be concerned about optimizing tuples with a large number of elements.

We examined the invocation of constructors of `tuple` and `pair`, as well as the factory functions for these types, namely `make_tuple` and `make_pair`. The total numbers of constructed `tuple` and `pair` objects are shown in Table 6.14. Since objects holding zero to four elements are very commonly used, separate entries for each of these element counts are included in Table 6.14. As the number of elements increases, the occurrence of such objects declines rapidly. Therefore, we used progressively larger bins for the subsequent entries in Table 6.14.

Overall, 85.38% of constructed objects are pairs, while tuples account for the remaining 14.62%. In addition, 39.36% of all `tuple` objects contain two elements. These results suggest that C++ developers frequently need to pack two elements together. The three-element tuples are the next most common, making up 27.41% of all `tuple` uses. Programmers also create 15.62% of one-element tuples, likely for generic programming purposes. Constructed tuples with more than eight elements are rare, with the largest tuple object containing 81 elements. Understanding the distribution of tuple element counts can aid implementers in optimizing the `std::tuple` class for common use cases. Also, it is quite likely that much more efficient implementations of tuple will be possible in the future C++26 standard due to the new pack-indexing language feature recently added to the C++26 working draft.

Table 6.16: Usage frequency of functions of the C++ standard cfenv library

Function Name	Average OPLOC	Appear in % of Packages
`fesetround`	$6.47 \cdot 10^{-6}$	0.71
`fegetround`	$2.33 \cdot 10^{-6}$	0.55
`fetestexcept`	$2.73 \cdot 10^{-7}$	0.71
`feclearexcept`	$2.65 \cdot 10^{-7}$	0.84
`feenableexcept`	$1.77 \cdot 10^{-8}$	0.46
`fesetenv`	$1.64 \cdot 10^{-8}$	0.21
`fedisableexcept`	$1.38 \cdot 10^{-8}$	0.42
`fegetenv`	$1.33 \cdot 10^{-8}$	0.13
`feraiseexcept`	$5.87 \cdot 10^{-9}$	0.04
`feholdexcept`	$4.71 \cdot 10^{-9}$	0.17
`feupdateenv`	$6.63 \cdot 10^{-10}$	0.04
`fegetexcept`	$3.93 \cdot 10^{-10}$	0.08

6.10 Floating-Point Environment Analysis

The next library usage analysis considered in our work focuses on the floating-point environment library cfenv from the C++ standard library. Introduced in C++11, the cfenv library offers various functionalities, such as manipulating rounding modes and handling floating-point exceptions. This library ensures consistent behavior of floating-point calculation, which is valuable for areas like scientific computing and financial software. A major downside of the cfenv library, however, is its reliance on global state manipulation at runtime. This reliance obstructs compile-time computation features, complicates multi-threaded programming, and raises portability issues across different platforms. To assess the impact of potential modification or substitution of the cfenv library, understanding how it is used in C++ software is crucial.

The cfenv library provides a series of functions for managing the floating-point environment. We examined all invocations of cfenv library functions in the C++ codebase we considered. The results are summarized in Table 6.16. The most frequently used function per each line of C++ code is `fesetround`, but its average OPLOC is merely $6.47 \cdot 10^{-6}$. The `feclearexcept` function appears in the largest fraction of packages (i.e., 0.84%) and is also infrequently seen. Overall, we find that the cfenv library functions are not used often in the C++ codebase. We also verified that the majority of packages that use cfenv functions do so sparingly, with 81.82% of packages invoking cfenv functions fewer than ten times.

Based on our findings, we suggest the C++ standard library provides an alternative to the cfenv library that does not rely on global states. Only a small fraction of packages would need to consider upgrading to use the new library, as the cfenv library is not widely used. Also, because the packages using the cfenv library often invoke cfenv functions a small number of times, migrating to an upgraded implementation should require minimal effort.

Table 6.17: Fraction of packages using random number functionalities from the cstdlib library and/or random library

Random Number Functionality Usage	Package Count	% of All Packages
Used only `std::rand`, `std::srand` from the cstdlib library	484	17.24
Used only items from the random library	128	4.56
Used both	121	4.31

6.11 Random Number Analysis

Lastly, we consider the use of random number generation features from the C++ standard library. The cstdlib library provides basic functions for this purpose, namely `std::rand` and `std::srand`, inherited from the C language. Random numbers generated by these functions tend to have poor statistical properties, making them less suitable for applications requiring high-quality randomness [27]. If a software package relies on the randomness of generated numbers to operate correctly, the use of these functions may compromise the software's functionality and even security. In contrast, the random library under the C++ standard library generates random numbers with better statistical properties. Additionally, the random library is more powerful, providing various random number engines and distributions. Understanding how these random number features are utilized in C++ software helps promote best practices in random number generation.

Based on the C++ standard draft document, we created a list of entities provided by the random library. We examined the usage of each item in the list in our raw data. We also checked whether a package uses `std::rand` and `std::srand` from the cstdlib library. Table 6.17 summarizes the results of our analysis and shows that the biggest fraction of packages (i.e., 17.24%) uses only `std::rand` and `std::srand` from the cstdlib library. A total of 4.56% of packages exclusively use the random library. Another 4.31% of packages use both libraries for random number generation. Many packages that adopt the random library also continue to use `std::rand` and `std::srand`, likely due to the simplicity and convenience of the older functions.

The data indicates that `std::rand` and `std::srand` are still the most frequently used random number generation functions in C++ software. C++ developers are not fully leveraging the superior features of the random library. The C++ standard library could be enhanced by modifying `std::rand` and `std::srand` to use higher-quality random number generators, similar to those in the random library. This update would automatically improve the quality of random number generation in existing C++ software, requiring no modifications from developers.

Chapter 7

Conclusions and Future Research

7.1 Conclusions

In this thesis, we presented a framework for applying an arbitrary analysis tool to the software packages from an operating system that employs the `dnf` package manager and a source code analysis tool designed to extract information about library usage in C++ software. Our framework can process a diverse collection of source packages, enabling various large-scale C++ analyses. Our analysis tool, developed using the Clang compiler front-end libraries, addresses the C++ parsing accuracy issues found in many existing studies and produces a reliable collection of instances of library uses from C++ software.

We demonstrated the effectiveness of our work by using the framework in conjunction with the library usage analysis tool to process the C++ codebase of Fedora Linux 37. A total of 2 379 software packages are analyzed, containing 398 065 762 lines of C++ source code. The number of considered packages is two to three orders of magnitude larger than in previous C++ studies, enabling a more comprehensive understanding of real-world C++ programming practices. Our library usage analysis tool extracted 341 181 165 instances of library usage from the codebase, allowing us to explore various aspects of C++ library usage behaviors.

The extracted instances were further analyzed to gain insights into various aspects of C++ library usage. For example, the Boost libraries are identified as the most popular third-party utility libraries in the codebase. Many frequently used Boost components, however, only have partial or no counterparts implemented in the C++ standard library, suggesting that the C++ standard library can benefit from incorporating more commonly needed features from Boost. In addition, we identified the worrying trend that programmers perform most indexing operations on sequential containers and views from the standard library without built-in bounds-checking, which may increase security risks in C++ software. We recommend that the C++ standard library include bounds-checking in the default behavior for indexing operations to effectively enhance the safety of C++ software. Observations and recommendations such as the above can benefit the C++ community by facilitating better education for C++ developers, offering guidance to C++ library maintainers, and enabling more informed decisions regarding the future evolution of the C++ language.

7.2 Future Research

This thesis has proposed a novel framework for applying an arbitrary analysis tools to a large C++ codebase and has developed a library usage analysis tool to extract information from accurately parsed C++ source code. Numerous research opportunities, however, remain unexplored. In this section, we discuss potential areas for future research.

We used the framework to process the C++ codebase of Fedora Linux 37, but the framework can be used with other operating systems that employ the `dnf` package manager. For instance, applying the framework to the C++ codebase of different major releases of Fedora Linux might reveal C++ programming trends over time. Other Linux distributions, such as Red Hat Enterprise Linux and Rocky Linux, can also be analyzed using the framework, broadening the scope of observation. Additionally, expanding the framework to support other package managers, such as `apt` and `pacman`, would further increase its versatility across a broader range of Linux variants.

Our analysis of library usage instances emphasizes the C++ standard library and Boost libraries, as these are the most commonly adopted utility libraries in the codebase. Future work could explore other widely used libraries, such as Qt, Eigen, and Poco. In addition, the statistical analysis of library usage instances presented in this thesis is exploratory. More comprehensive statistical research could be conducted to uncover further characteristics of C++ library usage. Moreover, the time when software packages are written and/or packaged can prevent software from adopting C++ library features introduced after release, which warrants exploration in future research. Lastly, analyzing the parameters used in library template instantiations can reveal common patterns that may guide future developments in C++ generic programming features.

Beyond the library usage analysis tool, other C++ tools can also be applied in our framework. For example, code quality tools can be integrated to identify common C++ programming issues. Language conversion tools can be enhanced by addressing errors when handling uncommon C++ programming practices in a large codebase. Additionally, code generation tools can leverage our findings that many existing C++ projects exhibit code quality issues and are slow to adopt modern features. Therefore, code generation tools should develop methods to filter high-quality software projects in their training input, thereby improving the quality of the generated code. In conclusion, various types of C++ analysis tools can benefit from being applied within our framework to large C++ codebases.

Bibliography

[1] Michael D. Adams. Companion git repository for *Lecture Slides for the Clang Libraries*, 2024. https://github.com/mdadams/clang_libraries_companion.

[2] Michael D. Adams. *Lecture Slides for the Clang Libraries [LLVM/Clang 17] (Edition 0.2.0)*. 2024. https://books.google.ca/books?id=WtrwEAAAQBAJ.

[3] James Antill, Jaroslav Mracek, et al. Dandified YUM project on GitHub, 2024. https://github.com/rpm-software-management/dnf.

[4] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from C++ source code. In *Proceedings of the International Conference on Software Maintenance*, pages 305–314, 2003.

[5] Lin Chen, Di Wu, Wanwangying Ma, Yuming Zhou, Baowen Xu, and Hareton Leung. How C++ templates are used for generic programming. *ACM Transactions on Software Engineering and Methodology*, 29(1):1–49, January 2020.

[6] Douglas Crockford. JSON data interchange format, 2024. https://json.org.

[7] Albert Danial, Sietse Snel, et al. Count Lines of Code project on GitHub, 2024. https://github.com/AlDanial/cloc.

[8] Jiachao Deng and Michael D. Adams. An analysis of library usage in the C++ code base of Fedora Linux 37. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, August 2024.

[9] Edward B. Duffy, Brian A. Malloy, and Stephen Schaub. Exploiting the Clang AST for analysis of C++ applications. In *Proceedings of the 52nd annual ACM southeast conference*, 2014.

[10] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering*, pages 779–790, 2014.

[11] Andrea Fornaia, Stefano Scafiti, and Emiliano Tramontana. JSCAN: Designing an easy to use LLVM-based static analysis framework. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 2019.

[12] Eclipse Foundation. Eclipse C/C++ Development Tooling engine, 2024. https://projects.eclipse.org/projects/tools.cdt.

[13] Free Software Foundation. GNU Compiler Collection, 2024. https://gcc.gnu.org.

[14] Lukas Gygi. CppBuild: Large-scale, automatic build system for open source C++ repositories. Bachelor thesis, ETH Zurich, Zurich, January 2021.

[15] Stefan Hanenberg. Empirical, human-centered evaluation of programming and programming language constructs: Controlled experiments. In *Lecture Notes in Computer Science*, pages 45–72. 2017.

[16] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, October 2014.

[17] Siegfried Hartogs. Code-driven language development: Framework for analysis of C/C++ open-source projects. B.S. thesis, ETH Zurich, Department of Computer Science, 2021.

[18] Red Hat. Podman container engine, 2024. https://podman.io.

[19] Red Hat. RPM package manager, 2024. https://rpm.org.

[20] Heidi Hokka, Felix Dobslaw, and Jonathan Bengtsson. Linking developer experience to coding style in open-source repositories. In *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 516–520, 2021.

[21] ISO/IEC 14882: Information technology — programming languages — C++. Standard, 2020. https://www.iso.org/standard/79358.html.

[22] Rainer Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27(6):637–649, June 1997.

[23] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery through visual language parsing and source code analysis. *Journal of Systems and Software*, 82(7):1177–1193, 2009.

[24] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22, 2001.

[25] László Nagy, Ben Reser, et al. Build EAR project on GitHub, 2024. https://github.com/rizsotto/Bear.

[26] Sebastian Nielebock, Robert Heumüller, and Frank Ortmeier. Programmers do not favor lambda expressions for concurrent object-oriented code. *Empirical Software Engineering*, 24(1):103–138, February 2019.

[27] Contributors of cppreference.com. Documentation of the std::rand function, 2024. https://en.cppreference.com/w/cpp/numeric/random/rand.

[28] LLVM Project. Clang: a C language family frontend for LLVM, 2024. https://clang.llvm.org.

[29] LLVM Project. Clang C++ Standard Library, 2024. https://libcxx.llvm.org.

[30] LLVM Project. LLVM compiler infrastructure, 2024. https://llvm.org.

[31] Stephen Schaub and Brian A. Malloy. Comprehensive analysis of C++ applications using the libClang API. pages 131–136, January 2014.

[32] Philipp D. Schubert, Ben Hermann, and Eric Bodden. PhASAR: An inter-procedural static analysis framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410, 2019.

[33] Philipp D. Schubert, Richard Leer, Ben Hermann, and Eric Bodden. Into the woods: Experiences from building a dataflow analysis framework for C/C++. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation*, September 2021.

[34] Bjarne Stroustrup. An overview of C++. In *Proceedings of the SIGPLAN workshop on object-oriented programming*, pages 7–18, 1986.

[35] Phillip M. Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of C++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering*, May 2016.

[36] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Fouste Khomh, Gias Uddin, and Alireza K. Motlagh. An empirical study of C++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering*, 48(05):1497–1514, May 2022.

[37] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. How do developers use C++ libraries? an empirical study. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, July 2015.

[38] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. An extensive empirical study on C++ concurrency constructs. *Information and Software Technology*, 76:1–18, August 2016.

[39] Haoxiang Zhang, Shaowei Wang, Heng Li, Tse-Hsun Chen, and Ahmed E. Hassan. A study of C/C++ code weaknesses on Stack Overflow. *IEEE Transactions on Software Engineering*, 48(7):2359–2375, 2022.