

Towards Understanding Third-party Library Dependency in C/C++ Ecosystem

Wei Tang¹, Zhengzi Xu^{2*}, Chengwei Liu², Jiahui Wu², Shouguo Yang³, Yi Li²
Ping Luo¹ and Yang Liu²

¹ School of Software, Tsinghua University, Beijing, China

² School of Computer Science and Engineering, Nanyang Technological University, Singapore

³ Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

tang-w17@mails.tsinghua.edu.cn, {zhengzi.xu, yi_li, yangliu}@ntu.edu.sg
{chengwei001, jiahui004}@e.ntu.edu.sg, yangshouguo@iie.ac.cn

ABSTRACT

Third-party libraries (TPLs) are frequently reused in software to reduce development cost and the time to market. However, external library dependencies may introduce vulnerabilities into host applications. The issue of library dependency has received considerable critical attention. Many package managers, such as Maven, Pip, and NPM, are proposed to manage TPLs. Moreover, a significant amount of effort has been put into studying dependencies in language ecosystems like Java, Python, and JavaScript except C/C++. Due to the lack of a unified package manager for C/C++, existing research has only few understanding of TPL dependencies in the C/C++ ecosystem, especially at large scale.

Towards understanding TPL dependencies in the C/C++ ecosystem, we collect existing TPL databases, package management tools, and dependency detection tools, summarize the dependency patterns of C/C++ projects, and construct a comprehensive and precise C/C++ dependency detector. Using our detector, we extract dependencies from a large-scale database containing 24K C/C++ repositories from GitHub. Based on the extracted dependencies, we provide the results and findings of an empirical study, which aims at understanding the characteristics of the TPL dependencies. We further discuss the implications to manage dependency for C/C++ and the future research directions for software engineering researchers and developers in fields of library development, software composition analysis, and C/C++ package manager.

CCS CONCEPTS

• **Software and its engineering** → **Reusability; Software product lines.**

* Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE '22, October 10–14, 2022, Rochester, MI, USA

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9475-8/22/10...\$15.00

<https://doi.org/10.1145/3551349.3560432>

KEYWORDS

Mining Software Repositories, Third-Party Library, Package Manager

1 INTRODUCTION

Reusing third-party libraries (TPLs) as dependencies has been widely adopted as a common practice to save time and manpower in software development. However, TPL reuse could also constantly expose downstream projects to potential risks of being attacked via vulnerabilities from dependencies. Therefore, acknowledging, monitoring, and managing TPLs that are introduced as dependencies, and further mitigating potential vulnerability threats, are critical demands in modern software development.

To facilitate the reuse of TPLs, many practical package managers have been proposed to manage dependencies, especially for younger languages. For example, NPM, Pip, and Maven are powerful package managers for Node.js, Python, and Java, respectively. However, for C/C++, which is an ancient programming language, although millions of components are available and various solutions for reusing TPLs have been proposed and adopted, there is still no unified package manager [38, 50] that can manage the dependencies of C/C++ projects well, which significantly complicates the dependency management of C/C++ projects and hinders the process of embracing agile DevOps [8] in C/C++.

Unlike other languages in that researchers have gained a lot of in-depth insights, for C/C++, people might have only few understanding of the fundamental aspects. For example, questions, such as how developers introduce C/C++ TPLs into their projects and what the TPL data scope is, are not comprehensively studied. Due to the lack of a standard package format and unified package manager, there are no available methods and tools to extract dependencies against large-scale C/C++ repositories. Consequently, people have no insights on TPL dependencies at large scale, and even no awareness of TPL data scope in the C/C++ ecosystem. Insights on TPL dependency landscape are important and beneficial for numerous areas. For example, OSSPolice [24] detects C/C++ TPL dependencies in Android applications. It relies on a local TPL database for detection. With unawareness of TPL data scope, OSSPolice collects GitHub repositories as TPLs that contain a large number of internal code clones [33, 36], further resulting in false positives. To bridge this knowledge gap, we seek to conduct a large-scale empirical study on dependencies in C/C++ ecosystem.

For an empirical study on dependencies to be successful and insightful, it is crucial to build a **comprehensive, precise, and efficient** detector for TPL reuse detection in large-scale repositories. A large number of existing researches [26, 32, 36, 49] conducted code clone detection to extract TPL dependencies in C/C++ repositories. However, not all dependencies are introduced by code clone. Miranda et al. [38] conducted a survey with a questionnaire involving 343 C/C++ developers. They report that there are multiple methods that developers prefer to handle dependencies in C/C++ projects. The top favorite methods to add dependencies are system package manager, header-only libraries, and git submodules. Only 10% of developers put the source code of libraries in host repositories, which suggests that code clone detection methods might only have a limited ability to detect dependencies.

In this paper, to unify the dependency management and detect dependencies of C/C++ properly, we first undertake a detailed investigation on how TPL dependencies are handled from the original published TPL data to the final dependent application and summarize the lifecycle for dependencies. Based on the dependency lifecycle, we propose a dependency detector that analyzes every step where dependencies might be introduced in the lifecycle and parses all possible TPL reuse methods which create dependencies. As shown in Figure 1, we divide TPL reuse methods into two categories depending on whether tools are used or not: **1)** package management tools and **2)** code clone. For both reuse methods, we design two corresponding modules for the TPL detection. We build a dependency detector, CCScanner, with the ability to detect dependencies introduced through 21 management tools. Moreover, its clone detection module integrates a state-of-the-art tool [49] for identifying C/C++ TPL reuse. Our experiments show that CCScanner can precisely and efficiently extract dependencies from large-scale repositories with the highest recall of 80.1%.

For a large-scale empirical study, we download a collection of 24K GitHub C/C++ repositories that own more than 100 stargazers and scan all repositories to extract dependencies using CCScanner. Totally, we obtain over 150K TPL dependencies from 24K repositories, based on that, we conduct an empirical study on dependencies in the C/C++ ecosystem. Through our empirical study, we conclude some key findings as follows. **1)** Developers prefer to add dependencies unintentionally in build scripts (over 70% of dependencies), not explicitly managing dependency installation. This convention significantly constrains the effectiveness of existing dependency detection tools. **2)** TPL data fragmentation exists in the C/C++ ecosystem. Libraries are inconsistent between databases, which may threaten the effectiveness of TPL detection and vulnerability reporting. **3)** System libraries on OS are the most important libraries for the C/C++ ecosystem. A group of 10 popular system libraries has an impact on the entire ecosystem. However, system libraries are always neglected in TPL management and detection. **4)** Management tools that rely on system libraries will introduce vulnerabilities from OS environments (22% of repositories). Half of version specifications of vulnerable libraries use vulnerable versions, that require developers to update the dependencies manually. Based on our findings, we present some recommendations for practitioners in related areas and future directions.

In summary, our contributions are as follows:

- We describe the lifecycle of C/C++ dependencies including mechanisms to import dependencies, dependency management tools, and TPL database for C/C++ dependencies.
- We propose a comprehensive and precise C/C++ dependency detector, CCScanner, based on the dependency lifecycle. The evaluation result demonstrates that CCScanner can achieve a precision of 86% and a recall of 80.1%. Besides, CCScanner is capable of efficiently scanning large-scale C/C++ repositories for empirical studies.
- We conduct a large-scale empirical study on C/C++ dependencies and discuss the research questions about the method to import dependencies, TPL data scope for C/C++ dependencies, key library in the C/C++ ecosystem, and TPL version constraints. We provide useful insights for helping improve the capability of package management for C/C++. We make our code and data available¹.

2 DEPENDENCY LIFECYCLE OVERVIEW

Dependency lifecycle is a concept that describes the process of TPL from the database where it is stored to applications where it is reused. Understanding how dependencies are processed in their lifecycles would guide us to build an effective dependency detector.

Referring to Maven for Java, we present how a TPL is processed in its lifecycle according to TPL location. Maven downloads all TPLs from the Maven central repository and put them under a local directory, like `~/m2`. Then, dependencies are packaged into the dependent project. We summarize that general TPL management mainly consists of two phases, Install and Build, that are separated by TPL location. The Install phase is responsible for installing third-party packages under local directories and the Build phase compiles and links libraries into final artifacts.

Unlike other languages, there is no unified and language-specific package management systems for C/C++, which means it is impossible to collect complete datasets from a single data source. Moreover, as far as we have reviewed, there is no relevant literature on what the library reuse mechanisms are and how to collect the aforementioned datasets. Therefore, it is the foundation and prerequisite for further investigation to figure out the TPL lifecycle when C/C++ projects reuse TPLs. With extensive research on previous studies [38, 50] and related real-world projects, we have summarized the C/C++ dependency lifecycle including TPL databases and TPL reuse methods as shown in Figure 2.

There are two types of colorful boxes, TPL data and TPL reuse methods. TPL data consists of data sources from which the reusable libraries can be retrieved. It includes default installed libraries on OS, mirrors of system-level package managers, central repositories of application-level package managers, and source repositories that are hosted on platforms such as GitHub or official websites. TPL reuse methods include package management tools and code clone, that introduce reused TPL in the two phases, Install and Build. Package management tools contain system-level and application-level package managers for installing dependencies in Install phase and build systems for Build phase. Different TPL databases and reuse methods work together as toolchains to complete the dependency

¹<https://github.com/lkpsg/ccscanner>

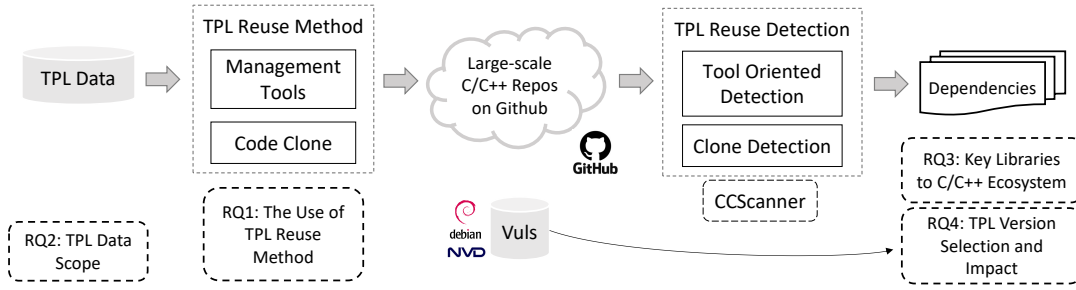


Figure 1: Overview of our work

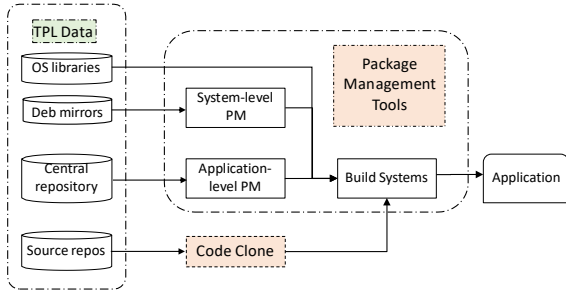


Figure 2: C/C++ TPL dependency lifecycle and toolchains. PM: package manager

management task during the entire lifecycle, and the toolchains can be classified into three types as follows:

- System library toolchain:** This toolchain retrieves TPLs from default installed libraries on operating systems (OS) and the mirrors of system-level package managers, such as Debian mirrors [6]. System-level package managers like APT [1] install libraries under system paths globally on OS. Build systems search reused libraries under system paths to compile and link [4]. It is clear that system-level package managers are system-specific, not language-specific. Therefore, different systems have different toolchains. For example, APT on Linux, Homebrew [12] on MacOS and winget [19] on Windows are toolchains for mainstream OS.
- Application-level package manager toolchain:** C/C++ has long been lacking a popular language-specific application-level package manager. C/C++ application-level managers are used to download TPLs from their central repositories for C/C++. Different from system-level package managers, this kind of package manager is language-specific like Maven and installed at the application level. They reside within a directory that is not maintained by the system environment. Generally, an official central repository of TPLs is built and maintained along with the application-level package manager. One of the famous C/C++ package managers is Conan [3]. Package managers need to integrate with build systems to specify the library directory path. Subsequently, build systems could link the libraries and compile the whole project.
- Code clone toolchain:** In this toolchain, libraries are reused by code clones that are not downloaded from specified TPL databases. The code clone method directly puts the source code of libraries into host repositories. It is a primary and quick way

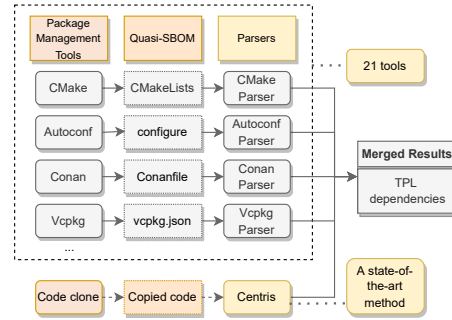


Figure 3: CCScanner

to import a dependency. Some techniques like `git submodules` help to manage the TPL versions and fetch source code automatically. The access to source code allows developers to modify the code of libraries to improve security, remove unnecessary code, or add new functionalities. However, reusing libraries through source code means that developers need to compile everything from source code on their own.

The differences between the three types of toolchains are the data source in the Install phase. An end-to-end toolchain completes the management functionality for the whole dependency lifecycle like Maven. However, not all repositories utilize the complete toolchain. For example, Ifopt [10] only contains build scripts without the use of package managers. It provides a `README.md` to instruct developers to install required dependencies manually through `apt-get` command. We consider this reuse method as an incomplete system library toolchain. It is possible to miss any step in the toolchain for a C/C++ project since there is no unified end-to-end package manager.

In addition to these toolchains, SBOM [17] (software bill of materials) files can be used to describe dependency relationships and specify the source of libraries. There are some standard SBOM formats, such as SPDX [13] and CycloneDX [14]. Large-scale development teams may define their own formats like Firefox [2] and Chromium [9]. However, less than 10 repositories in our 24K C/C++ repositories adopt SBOM to manage dependencies, and SBOM files only provide the description of dependencies. Therefore, we would disregard these SBOM rules in this paper.

```

Conan (Conanfile.txt) :
[requires]
nlohmann_json/3.7.3
[generators]
cmake

CMake (CMakeLists.txt) :
find_package(GLIB2 "2.50.0" REQUIRED)
find_package(GCRYPT "1.8.0" REQUIRED)

```

Figure 4: Examples of quasi-SBOM files for Conan [39] and CMake [48].

3 DEPENDENCY DETECTION

3.1 CCScanner

Dependencies in Java projects could be easily extracted by parsing SBOM files like `pom.xml` in Maven. However, it is difficult for C/C++ since there is no central SBOM to store all package information. As mentioned in Section 2, three types of toolchains work as Maven to retrieve TPLs, compile and package artifacts. Numerous available methods and tools can be selected at every step in the toolchain. Depending on whether tools are used, we divide elements that can be used to extract dependencies into two categories, **quasi-SBOM files** in package management tools and **copied code** with code clone method as shown in Figure 3.

Quasi-SBOM files Package management tools always need configuration files or specific statements to specify what libraries will be introduced. Therefore, these files can be recognized as a kind of quasi-SBOM files that describe required dependencies. Examples of quasi-SBOM and statements for Conan and CMake are shown in Figure 4. When using Conan to install libraries, `Conanfile.txt` file is written to describe package information. Dependencies can be extracted by parsing the `[requires]` field in `Conanfile.txt`. CMake finds external packages using `find_package` method. Libraries would be specified in `CMakeLists.txt` as arguments of `find_package`. A corresponding analyzer could be built to parse quasi-SBOM files based on the syntax of the package management tool.

After extensive research, we have summarized all tools that can be utilized in toolchains. Tools can be divided to package managers and build systems as shown in Table 1. In total, we have found 21 package management tools that are used by C/C++ developers. For system-level package managers, we select the Debian package manager since it is widely used in the C/C++ ecosystem. Other system package managers, such as RPM [16] and Homebrew [12], are similar and not considered in this paper. The Name column in Table 1 shows the names of corresponding quasi-SBOM files we parse for each tool. More details can be seen in our public code repo of CCScanner [5].

Copied code The code clone method copies and pastes reused code into the host repository and does not utilize any management tool. Consequently, there are no available quasi-SBOM files that describe the library information. In this case, the copied code can be used to generate features for code clone detection systems. TPL dependencies would be detected by matching similar code against a local TPL code database. For TPLs imported by code clone method, we utilize CENTRIS [49], a state-of-the-art system, to detect C/C++ TPL dependencies.

Table 1: Elements used in CCScanner.

Type	Tools	SBOM/Method
Package Managers	Deb	Control
	Conan	conanfile*, conaninfo.txt
	Vcpkg	vcpkg.json
	Clib	package.json, clib.json
	CPM	CMakeLists.txt
	Buckaroo	buckaroo.toml
	Dds	package.json5
	Hunter	CMakeLists.txt
	Cppget	manifest
	Xrepo	xmake.lua
Build Systems	Gitsubmodule	.gitmodules
	Pkg-config	*.pc
Build Systems	Make	Makefile
	CMake	CMakeLists.txt, *.cmake
	Autoconf	configure, configure.*
	Bazel	bazel.build, BUILD
	Meson	meson.build
	MSBuild	*.vcxproj, *.vbproj, *.props
	Xmake	xmake.lua
Build2	manifest	
Buck	BUCK	
Code clone	Centris	Code clone detection

Based on all management tools and TPL reuse methods, we build a comprehensive C/C++ dependency detector, CCScanner. It has the capability to parse the quasi-SBOM files of 21 package management tools and integrates CENTRIS to detect cloned TPLs. To the best of our knowledge, CCScanner has the best capability to detect C/C++ dependencies compared to existing techniques. Each existing tool is only capable of dealing with one or two elements in Table 1. For example, CENTRIS can only detect dependencies with cloned code. The existing state-of-the-art SBOM parser tool, OWASP [7] can only deal with CMake and Autoconf for C/C++.

3.2 Evaluation of CCScanner

In this section, we construct the ground truth for evaluation and present the accuracy evaluation results of CCScanner to prove that it is effective to scan large-scale databases for empirical study. We compare CCScanner with related works including two clone detection tools, CENTRIS (a state-of-the-art tool for cloned C/C++ TPL detection) and FOSSID (a famous commercial software composition analysis tool), and two state-of-the-art SBOM scanners, OWASP Dependency-Check and Sonatype. We present our findings on C/C++ dependency detection and discuss the effectiveness of existing tools.

3.2.1 Ground truth Construction. Since there is no public test data for C/C++ dependency detection, we need to establish the ground truth dataset containing a labeled mapping of C/C++ repositories and reused TPL dependencies. We adopt 15 real-world projects that cover all categories, such as audio processing, database management, and Internet connection, from the dataset of previous work [38] that analyzed the use of package managers by C/C++ developers. To label the dependency relationships, we manually check all files including source code files and non-source code files in projects to find TPL dependencies. After manually checking that

Table 2: Performance comparison of SCA tools. P: Precision, R1: Recall on the full ground truth, R2: Recall on dependencies that are supported to detect. F1: $2 * P * R1 / (P + R1)$.

Method	P (%)	R1 (%)	R2	F1
CENTRIS	33.6	5.3	22.4	0.09
FOSSID	12.7	2.1	12.5	0.04
OWASP	93.9	27.2	83.3	0.42
Sonatype	/	0	0	/
CCScanner	86.0	80.1	80.8	0.83

took over 100 hours, we obtained 589 TPL dependencies as the ground truth.

3.2.2 Accuracy analysis. We evaluate CCSanner and baselines on our ground truth and calculate the precision and recall. There are two kinds of recall rates, R1 and R2. R1 refers to the recall rate against full ground truth. However, each baseline is designed for only one or two elements in Table 1. We calculate R2 for each baseline based on the dependencies that the baseline supports to detect. From the results in Table 2, we can see that CCSanner outperforms all baselines in terms of F1 score.

The R1 and R2 of CCSanner are nearly identical, which means the detection capability of CCSanner can cover most elements used in ground truth. More than 90% of false results of CCSanner are caused by the code clone module provided by CENTRIS since we integrate unchanged CENTRIS for the detection. Without CENTRIS, CCSanner achieves both accuracy and recall rate of 97%. Sometimes, the SBOM parser in CCSanner makes mistakes in processing sophisticated syntax, such as complex variable replacement. Even though, our results prove that the SBOM parser module of CCSanner significantly outperforms OWASP Dependency-Check and Sonatype.

We inspected the false results reported by CENTRIS module and found that even though the CENTRIS module reports false original reused libraries, there truly exist code reuse and TPL dependency for each reported positive case since CENTRIS applies identical hashes to match reused code. Besides, we have found that many dependencies are reused by code clone method, but they only account for a small part of the total dependencies. In our ground truth, code clone introduces 24.4% of dependencies, that are not managed by other tools. As a result, CENTRIS can still be integrated with the SBOM module for large-scale empirical study. Reasons for false results of CENTRIS are illustrated in Section 3.2.3.

Our experiments show that CCSanner is comprehensive and precise for C/C++ TPL dependency detection. We use it to scan a large-scale database containing 24K C/C++ repositories and conduct an empirical study on dependencies in C/C++ ecosystem.

3.2.3 The effectiveness of existing SCA tools. Based on our experiments, we discuss the effectiveness of existing SCA tools for better understandings of C/C++ dependency detection.

SBOM parser-based tools generally have high precision, since the SBOM formats are strictly defined by package managers. However, both state-of-the-art SBOM scanners, OWASP and Sonatype, have

low recall rates. Sonatype does not report any dependency, because it only supports scanning SBOM files in Conan. There is no unified package manager or SBOM format in C/C++ ecosystem and multiple tools are used in the dependency lifecycle. The comprehensiveness of SBOM parser is the most important factor to the recall rate. Therefore, it is critical to understand the use of package managers. It would be discussed in our empirical study for RQ1 in Section 5.1.

For code clone detection-based tools, both CENTRIS and FOSSID perform poorly, especially on the recall. Such tools search similar or identical context hashes against a local TPL database. The reason for false positives is that duplication exists across different libraries. FOSSID has a lower precision since it adds non-source code files as features, unlike CENTRIS which only uses source code files. Duplicated context of license or description would cause false positive results. The reason for false negatives is the lack of reused libraries in TPL database. In general, the quality of TPL database is a determining factor for the effectiveness of the tool. How to build a TPL database is a critical research question for the field of dependency detection. It would be discussed for RQ2 in Section 5.2.

4 EXPERIMENTAL DESIGN

4.1 Dataset

We collect two datasets for large-scale empirical study including a large-scale database of popular C/C++ repositories and a TPL database.

C/C++ Repo Dataset To investigate dependencies in the C/C++ ecosystem, we extract dependencies from a large-scale collection of C/C++ repositories that are popular and receive recognition from the C/C++ community. C/C++ GitHub repositories with more than 100 stargazers are often used in previous works [24, 49] and the number of repositories increased to 24,001 in Mar 2022, which is adequate for the study. Therefore, we pull all 24K git repositories containing 3.5 billion lines of code as detection targets for the analysis on C/C++ ecosystem.

TPL Dataset As aforementioned, TPL databases can be divided into three categories: system libraries including OS libraries and mirrors of system-level package managers, central repositories of application-level package managers, and source repositories on the Internet. For OS libraries, We scan the OS environment and collect default installed libraries under the standard directory such as "/usr/lib" to form an OS library database. In addition to them, the metadata of all libraries in the Debian mirror is crawled. We combine OS libraries and Debian mirror together to build the system library database. For the central repository of application-level package managers, we collected and merged repositories of 7 package managers including Conan, Vcpkg, Clib, Cppget (Build2), Hunter, Buckaroo and Xrepo (Xmake). The merged repository contains 3380 reusable libraries. For source repositories, we select the database of the CENTRIS module as the third category of TPL database.

5 EMPIRICAL STUDY

In the experiments, we use CCSanner to conduct a large-scale C/C++ dependency analysis against 24K GitHub repositories with more than 100 stars. In total, we obtain over 150K TPL dependencies from 24K repositories with the latest versions to study the

TPL dependencies in the C/C++ ecosystem. Since the lack of understanding of many fundamental aspects, we first look into TPL reuse method and TPL data in the C/C++ ecosystem as shown in Figure 1. Then, we study key libraries and version constraints caused by large-scale reuses. In summary, we try to investigate the following research questions (RQs):

- RQ1: The use of TPL reuse methods to handle dependencies.
- RQ2: TPL data scope for C/C++.
- RQ3: Key libraries in the C/C++ ecosystem.
- RQ4: TPL version selection and the impact of vulnerable versions.

5.1 RQ1: The Use of TPL Reuse Method

In this section, we will investigate the use of TPL reuse methods in dependency lifecycle. Even though existing studies [38] have discussed what TPL reuse methods are adopted to add dependencies in C/C++ projects, insufficient empirical evidence on dependencies and details of methods were provided. Therefore, we aim to re-investigate this question via large-scale analysis. As described in Section 2, multiple methods and tools are combined to form a toolchain in two phases. We investigate each step in the dependency lifecycle to answer three main questions, (1) which phase is preferred to introduce TPL dependencies? (2) what tools are reused more often? (3) how are the tools combined as toolchains? The following three subsections answer these questions respectively.

5.1.1 Preferred Phase. We first count how many dependencies are handled and how many repositories are involved in each phase. Statistics indicate that 71.5% of dependencies are extracted from Build phase and 80.3% of repositories adopt build systems. However, only 47.5% of repositories adopt methods and tools in Install phase that introduces 37.5% of dependencies. It proves that TPL management is separated in the dependency lifecycle. Build phase is complicated for C/C++ projects to generate binaries and support enormous platforms, therefore the build scripts are provided along with source code in TPLs. However, C/C++ projects are not required to provide management scripts to install dependencies. Sometimes, developers write customized documents to guide system-level package managers to install dependencies. However, it is not an explicit and formal method for TPL management and detection. It can be seen that C/C++ projects do not perform well on explicitly managing dependency installation using tools in Install phase. However, most of the existing SCA tools focus on Install phase including code clone detection and SBOM scan. It means they would neglect dependencies added in Build phase that occupy a large portion. Only 9% of dependencies are handled in both two phases from Install to Build, which means a lot of manual efforts are necessary to handle the whole lifecycle for C/C++ dependencies.

Table 3: The use of package management tools.

Install	Dep.(%)	Repo.(%)	Build	Dep(%)	Repo(%)
Gitsubmod	4.94	16.79	Make Only	2.27	25.00
Deb	9.32	5.90	Cmake	51.33	39.11
Pkg-config	6.94	1.83	MSBuild	5.34	16.86
Conan	0.45	0.99	Autoconf	12.3	15.27
Vcpkg	0.56	0.39	Meson	3.0	2.11
Hunter	0.18	0.2	Bazel	0.81	1.37
Clib	0.05	0.12	Buck	0.0	0.27
Cpm	0.07	0.1	Xmake	0.08	0.09
Buckaroo	0.0	0.02	Build2	0.0	0.01

Finding-1: ① Developers prefer adding dependencies unintentionally in Build phase (over 70% of dependencies), not explicitly managing the installation of dependencies in Install phase. This convention significantly constrains the effectiveness of existing dependency detection tools. ② Due to the separation of dependency lifecycle for C/C++, developers usually do not maintain complete toolchains for dependencies. Only 9% of dependencies are handled by complete toolchains. A lot of manual efforts are required for TPL reuse.

5.1.2 Usage of Tools. For the usage of each tool, we count the proportion of dependencies that are managed by the tool and the proportion of repositories that adopt it. Results are shown in Table 3. The code clone method is excluded since it is a general method rather than a specific tool. Two package managers, Cppget and Xrepo are excluded since they are a part of Build2 and XMake respectively. Dds is not in the table, because it is not used by any repository in C/C++ Repo Dataset.

Shown in Table 3, the most popular tool in Install phase is Gitsubmodule, that is actually a tool for automatic code clone. Besides, we notice that 32.6% of repositories adopt code clone method and introduce 15.77% of dependencies. Code clone is more popular than tools in Install phase. This is a disappointing fact that C/C++ developers prefer to explicitly specify resources in Install phase by code clone, which will hinder the package management process.

Only 1.8% of repositories (442/24,001, the sum of Conan, Vcpkg, Hunter, Clib, Cpm, and Buckaroo) adopt application-level package manager toolchains that introduce 1.3% of dependencies, even if it provides the best capability to manage TPLs like Maven. Between application-level package managers, we notice that Conan and Vcpkg contribute about 80% of usage. Other tools are rarely used in real-world C/C++ projects. Interestingly, Conan and Vcpkg choose a different design philosophy from others. They directly use the ready-to-use binary packages rather than source code packages to construct their central repositories. This strategy eases the effort of users to compile the code on their own. We believe it is the future direction for C/C++ package managers.

We notice that traditional tools that do not have the capability to manage TPLs still have a high prevalence. Modern tools, including six application-level package managers mentioned above and two build systems (Xmake and Build2) that integrated a C/C++ package manager, need to extend their prevalence.

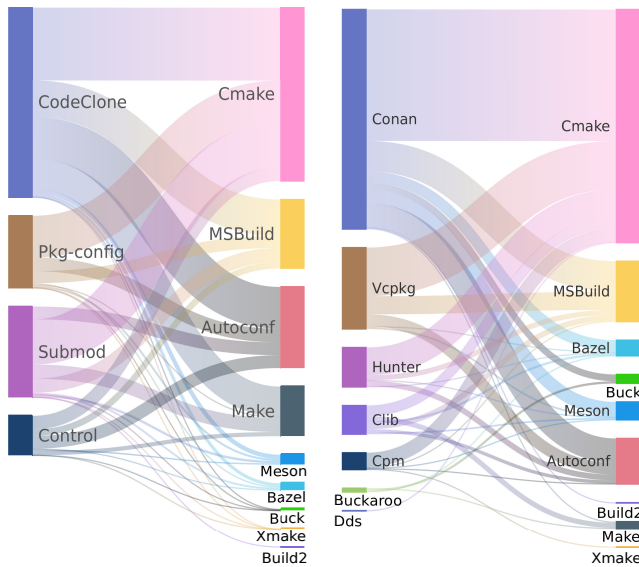


Figure 5: Combinations of reuse methods in dependency lifecycle.

Finding-2: ① For repositories that use formal methods to explicitly specify dependency installation, developers prefer to implement dependency installation through the naive code clone method. There is no convention at present to use a package manager in the C/C++ ecosystem. ② Less than 2% of repositories adopt application-level package managers, which means package managers are still in the primary stage in the C/C++ ecosystem.

5.1.3 Combinations of reuse methods. Since C/C++ dependency lifecycle is separated into two phases. Toolchains are formed by the combination of dependency installation tools and build systems. For each phase, there are many available tools. As a result, countless toolchains can be formed and used in C/C++ projects. We extract all combinations of tools in C/C++ Repo Dataset to investigate toolchains in the C/C++ ecosystem.

Results are shown in Figure 5. Two graphs are divided by traditional tools (shown on the left) and modern tools (shown on the right) in Install phase. For the sake of typography, the sub-graph on the right is scaled up to the same size of the left sub-graph. The figure shows that each tool in Install phase can be combined with any tool in Build phase to form a toolchain. It reveals a serious problem that TPL management methods in the C/C++ ecosystem are chaotic. There is no unified package management tool for C/C++, moreover, the separation of C/C++ dependency lifecycle further increases the complexity and fragmentation of the usage of tools with an exponential growth rate. Chaotic usage of tools increases the development cost on library reuse, collaboration and integration.

For both traditional tools and modern tools in Install phase, Cmake, MSBuild, and Autoconf are always the main options to build the software, which is consistent with Table 3. Interestingly, we find that Make is a main choice for traditional tools in Install phase, however, application-level package managers rarely work with Make. Furthermore, Table 3 shows that Make is too naive to handle dependencies. Despite the fact that it is the most important build

system in C/C++ ecosystem and more than 80% projects depend on it in Build phase, only 2.27% of dependencies are added through it.

Even for modern tools in Install phase, they rarely work with modern build systems. This is because package managers need to first consider integrating with mainstream build systems due to the separation of C/C++ dependency management. The separation impedes the formation of advanced toolchains.

Finding-3: ① Due to the lack of a unified package manager and the separation of TPL dependency lifecycle, the usage of toolchains is chaotic in the C/C++ ecosystem. ② Compatibility issues exist in toolchains for C/C++. Make is a usual option for traditional TPL reuse methods, but it is not effective to work with modern application-level package managers. ③ Basic build systems, including Make, Cmake, MSBuild, and Autoconf, still dominate the usage in the ecosystem despite the fact that many modern build systems have been proposed.

5.2 RQ2: TPL Data Scope

The goal of this section is to understand the TPL data scope in the C/C++ ecosystem. As described in Section 2, there are three categories of TPL databases, including system mirrors, central repositories of package managers, and source code hosts. For source code hosts, GitHub repositories are often collected as the local TPL database for detection [20, 24, 49]. However, it is not clear which database is better and whether libraries in databases are sufficient and high-quality. Therefore, we conduct a comparative analysis on three categories of databases to answer two questions in the following two subsections: (1) what is the data coverage of each TPL database? (2) is it reasonable to use GitHub repositories as the local TPL database like what related work does?

5.2.1 Database Coverage. It is important for TPL database to have a large coverage since it defines the availability and effectiveness of package managers in practice. Moreover, it is beneficial for the improvement of the recall rate of detection as mentioned in Section 3.2.3 From extracted 150K dependencies, we have identified 23,322 reused libraries. Figure 6 presents the data coverage of the three individual databases and a merged database that contains all libraries of three databases. We sort all reused libraries by their popularity (i.e. the number of reuses) and generate a batch every 100 libraries on the horizontal axis. The data coverage indicated by blue lines is calculated by counting how many libraries of a batch exist in the database. For example, the first point (0, 1) in the sub-figure of system library means that all top 100 popular libraries exist in the system library database.

From Figure 6 we can see that system library database has the largest coverage on all reused libraries and it contains the largest number of popular libraries. All top 100 popular libraries and 5,163 reused libraries can be found in the system library database that covers a total number of 65.5% of dependencies (98,896). Central repositories of application-level package managers have a lower coverage (70%) on top 100 popular libraries than system library database, but results show that 58% of libraries in central repositories are top 20% popular libraries which means these central repositories have high-quality libraries that may be commonly used by developers. In contrast, more than 60% of libraries in the

other two databases are not detected as reused libraries. For source code repositories, the database of CENTRIS has the lowest coverage on popular libraries that drops to 41% and the coverage on top 100 popular libraries is less than 50%. From the merged database, we notice that the three categories of databases cover 77% of dependencies with 8,505 reused libraries.

Even though the system library database has the largest data coverage, it only contains 22% of reused libraries. The merged database of three categories contains 36% of reused libraries. After eliminating overlapped libraries between three categories of databases, each one still has more than 50%, 73% and 98% left as unique libraries, respectively for central repository database, GitHub repositories, and system library database. It suggests that TPLs in C/C++ ecosystem are scattered on the web. There is no complete TPL database for the ecosystem.

Through checking overlapped libraries, we find that data fragmentation causes a problem that overlapped libraries are not consistent with each other. For example, Debian maintains LibPNG which is downstream of original official repository. New versions are created to fix vulnerabilities by Debian team and a Debian revision version is attached following original semantic version. This causes differences in terms of versioning and security information. Version number would be ineffective to report potential vulnerabilities. For instance, CVE 2019-7317 exists in libpng 1.6.x before 1.6.37, however, it does not affect Debian libpng after 1.6.36-4. Moreover, there are two forms of packages for a C/C++ library, source code repository and binary package. This kind of difference would likewise affect the vulnerability reporting in security management since a source repository might be compiled into multiple binary packages. If a vulnerability is bundled with a source repository, it may not be contained in some binary packages. For example, CVE-2021-38171 only affects the libavformat package in FFmpeg, we scan the dependencies in Debian mirrors and find 112 repositories depend on FFmpeg, however, 12/112 repositories do not depend on the vulnerable libavformat package.

Finding-4: ① There is no unified and complete TPL database in the C/C++ ecosystem. Each category of database has more unique libraries than overlapped libraries. The merged database only contains a small portion of reused libraries (36%). ② TPL data fragmentation exists in C/C++ ecosystem. Libraries are inconsistent between databases, that may threaten the effectiveness of TPL detection and vulnerability reporting.

5.2.2 GitHub TPL Coverage. Nowadays, GitHub is the main platform to host source code repositories. Researchers naturally consider collecting GitHub repositories as a TPL database. The mainstream solution to local TPL database construction in related work is to collect GitHub repositories with a significant stargazer count [20, 24, 49]. However, C/C++ is an older programming language than GitHub. There are a large number of C/C++ libraries that are hosted in Linux distributions, SourceForge, etc. Therefore, it is questionable whether GitHub repositories have convincing representativeness as TPL database. Collecting GitHub repositories to build TPL database is based on an assumption that popular repositories on GitHub are likely to be reused as TPLs in software

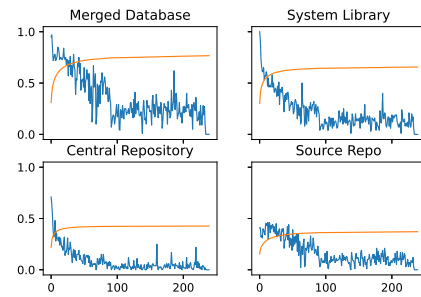


Figure 6: TPL data coverage of all database. The blue line represents data coverage of every 100 libraries. Yellow line indicates the accumulated number of dependencies that are contributed by covered libraries.

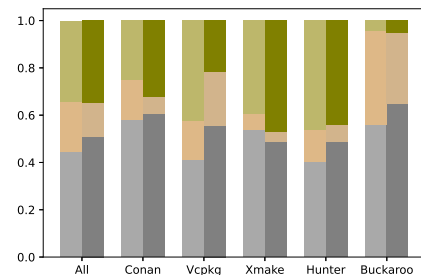


Figure 7: Distribution of data sources for overall libraries (left) and reused libraries (right) for each package manager. The bottom layer represents GitHub repositories with more than 100 stars, the middle layer: less than 100 stars, the top layer: not GitHub repositories.

development. There are two serious problems with this assumption: (1) it assumes that most libraries are hosted on GitHub. (2) repositories on GitHub are reusable libraries.

To evaluate the first assumption, we use the collection of libraries in central repositories as the ground truth to figure out how many libraries are hosted on GitHub. Since application-level package managers are designed for TPL reuse management, packages in their central repositories can be treated as TPLs. To keep consistent with previous works [24, 49], we divide all source code repositories into three categories, GitHub repositories with more than 100 stars, GitHub repositories with less than 100 stars, and non-GitHub repositories.

Figure 7 shows the distribution of data sources for overall libraries and reused libraries for each package manager. We can see that overall, 43% of libraries and 50% of reused libraries are hosted on GitHub with more than 100 stars. It means that collecting repositories with more than 100 stars will miss half of libraries. About 20% of libraries are GitHub repositories with less than 100 stars. However, we do not recommend practitioners to crawl all repositories with few stars, since a lot of noise data would be collected due to the long-tail distribution of stars. On GitHub, 41K C/C++ repositories have more than 50 stars and more than 2 million non-folk C/C++ repositories are hosted on GitHub.

To evaluate the second assumption, we inspected the false positives mentioned in Section 3.2.3. We noticed that there are a significant number of repositories that are not TPLs and would not be reused by other developers. Such repositories are the main reason for false positives in results of CENTRIS. We propose two concepts for classifying source repositories, application and library. An application is a computer program that is designed for specific users. A library is a collection of code to make other programs work. They provide special features to developers. The most remarkable difference is that applications are user-oriented, and libraries are developer-oriented. It is crucial to distinguish applications and libraries to build a TPL database.

Finding-5: Collecting GitHub repositories with more than 100 stars is not a reasonable approach to building TPL database. Only half of libraries of C/C++ package managers could be collected in this way. Moreover, it would collect a significant number of applications that are not TPLs, leading to a low precision (33.6%) of CENTRIS for TPL detection.

5.3 RQ3: Key Libraries to C/C++ Ecosystem

A library is considered more popular if it is reused by more repositories. Key libraries have a considerable impact on the whole ecosystem. It is a critical research question in previous empirical studies for other ecosystems [28, 35, 42]. We conduct an analysis on the popularity of libraries in the C/C++ ecosystem to answer questions: (1) what is the library popularity distribution? (2) what are the commonly reused libraries and how do they affect the ecosystem?

5.3.1 Library popularity distribution. A total number of 150,927 TPL dependencies are extracted from C/C++ Repo Dataset. On average, each C/C++ repository depends on 6.3 external libraries. As for the library usage intensity, 42.4% (10179 / 24001) of repositories do not depend on TPLs and 41.4% of repositories (9,934) adopt at most 10 libraries. Only 2.1% of repositories (502) reuse more than 50 libraries. 23,322 distinct libraries are identified and each library is reused 6.5 times on average. The median of reuse times is 1. It means that a highly polarized popularity structure exists in C/C++ libraries. Top 1%, 5%, 10% and 20% of libraries contribute 45%, 68%, 77% and 84% of dependencies, respectively. The popularity distribution highly meets the pareto distribution that describes social behaviors, i.e., most dependencies are contributed by a small group of libraries. In economics, the Gini coefficient is a measure of statistical dispersion intended to describe the income polarization. We calculate the Gini coefficient of popularity as 0.79 that is extremely unequal distribution in human society.

We scan C/C++ Repo Dataset over the last ten years and find that the polarized popularity structure has intensified over time. In 2011, the Gini coefficient of popularity is 0.69 and Top 1%, 5%, 10% and 20% of libraries contribute 29%, 54%, 65% and 76% of dependencies, respectively.

A highly polarized popularity structure is a double-edged sword. On one hand, the majority of the dependencies can be covered with a small TPL database. It would be effective and efficient to build such a small database for library management and dependency detection. Data collection becomes much easier and less time-consuming. On the other hand, the C/C++ ecosystem heavily depends on a small

Table 4: Top 10 popular libraries. STD Lib means standard library.

Direct	Type	Repo	Transitive	Type	Repo
Threads	STD Lib	2825	Threads	STD Lib	6459
Zlib	Compression	1977	Zlib	Compression	5212
Boost	Programming	1676	Libm	STD Lib	5124
OpenCV	CV	987	Libdl	STD Lib	3830
OpenSSL	Crypto	929	OpenSSL	Crypto	3387
Libm	STD Lib	890	Cuda	DEV ENV	3291
OpenGL	Graphics	876	Librt	STD Lib	3191
Eigen	Math	776	OpenMP	Programming	2999
OpenMP	Programming	774	Libsocket	Network	2971
X11	GUI	737	Libnsl	Network	2934

group of critical libraries that may threaten the security of the entire ecosystem. If a library contains a vulnerability with high severity, it would be a disaster to the ecosystem such as the spread of the Heartbleed Vulnerability [25] in the most popular communication TPL, OpenSSL.

Finding-6: A highly polarized popularity structure exists in C/C++ TPLs and has intensified over the last ten years. Practitioners in areas of TPL management and detection could focus on a small group of TPL data. However, the highly polarized popularity structure may threaten the security of the entire ecosystem.

5.3.2 Popular TPL Influence. To explore the extent to which popular libraries affect the whole ecosystem, we select the top 100 libraries to observe the direct and transitive dependencies. Through checking all direct dependencies, we notice that some repositories are reused as libraries by other repositories. Therefore, we can generate the dependency chain between repositories to identify transitive dependencies. We find that it is a big research challenge to construct a whole dependency graph for the C/C++ ecosystem. More details are described in Section 6.2. Therefore, we only resolve the transitive dependencies for top 100 libraries. Table 4 shows top 10 popular libraries from direct dependencies and transitive dependencies. We ignore compile-time dependencies such as pkg-config and test-time dependencies like googletest since they are not required in the runtime phase.

Considering direct dependencies, a set of top 10 popular libraries affect 28% of repositories (6754) and the top 100 libraries affect 42% (10100). Since 13,822 repositories have dependencies, the top 100 popular libraries affect 73% of repositories with dependencies. For transitive dependencies, the top 10 popular libraries have an impact on the entire ecosystem.

Moreover, compared to the results of previous studies, we notice that a large number of dependencies of system libraries are ignored by previous studies. However, our statistical results show that system libraries are the most important category of libraries. They occupy a large portion of dependencies, especially transitive dependencies. All top 10 popular libraries of transitive dependencies are low-level system libraries that are default installed on OS. Previous empirical studies investigate C/C++ dependencies using clone-based detection techniques. As a consequence, only TPLs contained in detection targets can be detected. It would reduce the number of libraries discovered. The libraries (e.g. system libraries) that are not used by code clone would be missing. Since C/C++ are low-level programming languages, a large number of

C/C++ libraries are installed on OS by default, that are downloaded through system-level package managers and installed globally. If an application relies on system libraries, build tools can directly locate the reused libraries under system paths without copying the code of libraries. Therefore, the previous studies lack the capability to extract dependencies of system libraries and all of their findings are explored around non-system libraries.

Finding-7: System libraries on OS are the most important libraries for the C/C++ ecosystem. A group of 10 popular system libraries have an impact on the entire ecosystem. However, system libraries are always neglected in TPL management and detection.

5.4 RQ4: TPL Version Selection and Impact

Version selection is an important behavior in TPL reuse. It could directly affect library update and vulnerability propagation [29, 35]. To demystify TPL version selection and its impact in the C/C++ ecosystem, we try to answer two questions in the following two subsections respectively: (1) do developers specify version constraints for dependencies? do they update them? 2) How do vulnerabilities in TPLs affect the C/C++ ecosystem?

5.4.1 TPL Version Management. Version specification indicates the compatibility of dependencies and well-maintained versioning constraints can bring many benefits, such as avoiding using vulnerable versions. Many works [21, 27, 31, 35, 40, 41, 53] investigate the security landscape based on version specifications in other language ecosystems. However, specifying version constraints is not always a requirement and there are few relevant regulations and conventions for C/C++. To investigate version constraints in the C/C++ ecosystem, CCScanner parses statements about dependency version for every package management tool to extract version specifications.

In total, we obtain 35,349 dependencies that specify versions. Unlike other modern languages that embrace semantic versioning to specify their required TPL versions, dependency declaration in C/C++ projects has a quite low ratio of explicitly specifying the versions of dependencies (27%). Such a low ratio is an indication of inadequate control of version constraints in the C/C++ ecosystem. To understand the reasons, we examine the stages in dependency lifecycle that introduce the TPLs without version specification. We find that 51.7 % of dependencies handled by tools during Install phase specify versions. However, the number drops to only 8.09% in the Build phase. Furthermore, for package managers that contain multi-version in their central repository like Conan, 97% of dependencies specify versions. For package managers that do not support multiple versions for one package, like system-level package managers, only 18.4% specify versions. Moreover, we notice that no low-level system libraries specify version constraints in their dependencies. The OS environment takes responsibility for maintaining system libraries, such as patching and updating. That explains why the system package managers have a low ratio of version constraints. It is not necessary to consider the problem of library versions once the OS environment is determined. As for updates, we select the top 10 popular libraries with version constraints and calculate how many dependencies adopt the latest

versions. Results show that only 9.5% of constraints specify the latest versions.

Finding-8: ① Version specifications are more widespread in Install phase, especially for package managers that support multiple versions. ② The versions of system libraries are determined by OS. Therefore, dependencies that have requirements on system library versions might be not compatible with some systems.

5.4.2 Vulnerable TPL Influence. An important version-based analysis is to investigate how vulnerable libraries affect the language ecosystem [22, 27, 35, 52, 53]. Different from other language ecosystems where the allowed versions are usually explicitly specified and vulnerabilities can be easily tracked, only a small part of dependencies are declared with specified versions in C/C++ projects so that Build phase could find the default version for system libraries on OS. To study the impact of vulnerable libraries, we use the latest Debian security information to match dependencies without version constraints, supposing that these dependencies reuse the latest versions of libraries on OS. We have got 6661 unpatched vulnerabilities existing in C/C++ repositories on Debian. As for dependencies with version specifications, we match specified versions to a vulnerability database of a commercial security company that contains the vulnerability list of 299 reused libraries.

For dependencies without version specifications, 13% of dependencies (13463) reuse vulnerable libraries involving 22% of repositories (5315) through direct dependencies. Moreover, the affection rate will become much higher if the transitive dependencies are taken into consideration. By checking management tools related to vulnerable dependencies, we find that all vulnerable dependencies come from build systems and management tools in Linux ecosystem like Deb, i.e., the system library toolchain. It relies heavily on OS environment and the system libraries. An advantage is that security issues would be fixed by OS maintainers. It lightens and simplifies the tasks of developers.

For dependencies with version specifications given by developers, 1143 dependencies are found to use vulnerable versions involving 118 reused libraries and 758 upstream dependent repositories. We find that 49.3% of dependencies with version specifications libraries use vulnerable versions. It means that poorly maintained versions cause a significant security risk to dependent applications. We also find that it is challenging to recall introduced vulnerabilities based on library name and version for C/C++.

Finding-9: Management tools that rely on system libraries will introduce vulnerabilities from OS environments. Developers would not explicitly specify versions using these tools and the OS maintainer is responsible for eliminating vulnerabilities. About half of dependencies with version specifications of vulnerable libraries use vulnerable versions, which requires developers to update the dependencies manually.

6 DISCUSSION

6.1 Implications

For C/C++ Developers. We recommend C/C++ developers to use application-level package managers in the development and avoid

using customized documents to describe dependency installation. Application-level package managers require developers to facilitate the use of dynamic linking rather than code clone. Due to the small size of central repositories, reused libraries may be not hosted in public official repositories of package managers. It poses challenges and difficulties for the use of package managers in practice. To address this issue, developers could create a package repository to store TPLs for public release and join the ecosystem construction for C/C++. Developers should be aware of implicit OS library dependencies and convert them to explicit ones through management tools while it would require more effort. When code clone, especially partial clone, appears, package managers do not apply to cloned code. It is suggested to adopt a SBOM mechanism to manage cloned code if code clone is unavoidable.

For C/C++ TPL Auditors. Due to the lack of a unified package manager and the features of C/C++, it is more complicated for C/C++ to detect reused TPLs than other languages. Existing detection tools only have a limited capability to resolve and track the whole dependency lifecycle. Our findings help practitioners to develop more effective TPL detection systems for C/C++ and build TPL databases more effectively. We conclude some recommendations for C/C++ TPL Auditors: ① parsing build scripts in Build to recall more dependencies; ② scanning OS environment for system libraries; ③ collecting TPL data from three categories of databases; ④ distinguishing applications and libraries for source code repositories; ⑤ collecting a database from popular libraries if resources are limited; ⑥ focusing on differences caused by data fragmentation.

For C/C++ Package Managers Designers Existing system package managers are not designed for C/C++ package management and lack some advanced features. As shown in Figure 2, system libraries are installed through commands like `apt install` and imported directly by build systems. Designers should add a SBOM file to describe dependencies and reproduce the environment automatically. `Dockerfile` in `docker` is a good practice, but there is no such mechanism for system package managers on Linux. Many developers rely on system package managers, even they are not designed for development. Since system package managers only provide one version for each package, they do not support evolving needs of developers. Besides, they do not provide security alert when users import a vulnerable package. Designers could consider to provide multiple versions and track introduced vulnerabilities to support effective and secure development. For application-level package managers, it is almost impossible to build a central and trusted database that contains sufficient C/C++ libraries. We recommend designers should consider making application-level package managers compatible with existing library databases, such as Debian mirrors, that have the largest coverage and massive packages. Due to the separation of dependency lifecycle and chaotic combinations of tools, package managers should integrate with all mainstream build systems to complete the whole toolchain automatically and become more ease-of-use.

6.2 Limitations

Code clone detection In our work, we integrate unchanged CENTRIS for code clone detection. CENTRIS is a state-of-the-art tool to

detect C/C++ dependencies. However, it is not precise or efficient for large-scale empirical studies as mentioned in Section 3.2.2. It makes our empirical study lack dependencies that are caused by code clone in the evolutionary analysis.

Dependency chain Macros and compilation settings change dependencies of a project. It is decided in the period of compilation and cannot be identified by static code analysis. For example, there are dozens of flags in the build scripts of FFmpeg [11]. Each flag can be set to enabled to control whether a TPL needs to be introduced during the Build phase. Besides, as mentioned in Section 5.2.1, dependencies on source code repositories may not keep the same with binary packages after compilation. It also affects the dependency chain construction.

7 RELATED WORK

Dependency Management & Detection. To properly manage dependencies of user projects, lots of research has investigated the way how dependencies are integrated and the accompanied problems.

For C/C++, Miranda et al. [38] performed a questionnaire survey and collected opinions on the use of package managers by 343 C++ developers from 42 open-source projects. Centris [49] and DéjàVu [36] and the commercial product FOSSID detect C/C++ TPL dependencies through code clone detection. OSSPolice [24] and Modx [51] detect TPL dependencies in binaries. Besides, there are some famous SBOM scan tools, like OWASP [15] and Sonatype [18].

Besides, there are also many work investigating the dependency management in other languages. Dietrich et al. [23] studied the choices developers made on defining dependencies in 17 different package managers. BuildMedic [37], Riddle [46], and Sensor [47] study on build issues in Maven. Wang et al. [34, 43–47] had done many interesting work to investigate dependency management issues in different languages.

Dependency based Ecosystem Analysis. For C/C++, there is few work, especially large-scale studies due to the lack of unified C/C++ package manager. Wu et al. [50] analyzed 30 applications to discuss the usage of C++ standard libraries. OSSPolice [24] conducts a large-scale usage analysis in Apks.

Many researchers analyzed dependencies for other language ecosystems. Huang et al. [30] and He et al. [28] conducted studies on usages and migrations of libraries for Java. Zimmermann et al. [53], Decan et al. [22], Chengwei et al. [35] investigate on security risks of dependencies for JavaScript. However, all of these vulnerability impact analysis are based on clear dependency relations, and our work can provide dependency relations for C/C++ projects in a confident manner, therefore can inspire more in-depth analyses on vulnerabilities in the ecosystem of C/C++.

8 CONCLUSION

In this paper, we first undertake an extensive investigation on how TPL dependencies are handled in C/C++ projects and summarize the lifecycle for C/C++ TPL dependencies. Based on the dependency lifecycle, we propose a comprehensive and precise C/C++ dependency detector, CCScanner. Experiments demonstrate that CCScanner is capable of scanning large-scale C/C++ repositories

for empirical studies. We apply C/C++ to scan 24K Github repositories to conduct a large-scale empirical study on dependencies in C/C++ ecosystem. Our study unveils a lot of findings regarding TPL reuse method, TPL data, key TPLs and TPL version selection.

ACKNOWLEDGMENTS

This work is supported by the Key Research Program of the Ministry of Science and Technology of China (no.2018YFF0215901), Nanyang Technological University (NTU)-DESAY SV Research Program under Grant 2018-0980, Singapore Ministry of Education (MOE) Academic Research Funding (AcRF) Tier 2 under Grant MOE-T2EP20120-0004, and the program of China Scholarships Council award (202006210393).

REFERENCES

- [1] 2022. APT Package Manager. [https://en.wikipedia.org/wiki/APT_\(software\)](https://en.wikipedia.org/wiki/APT_(software)). (Accessed on 05/05/2022).
- [2] 2022. Automation for updating third party libraries for Firefox. <https://github.com/mozilla/protecdiscretionary{\char\hyphenchar\font}}services/updatebot>. (Accessed on 04/04/2022).
- [3] 2022. C/C++ Package Manager. <https://conan.io>. (Accessed on 04/04/2022).
- [4] 2022. CMake command: find-library. https://cmake.org/cmake/help/latest/command/find_library.html. (Accessed on 05/05/2022).
- [5] 2022. The code repository of CCScanner. <https://anonymous.4open.science/r/ccscanner-7491/>. (Accessed on 05/05/2022).
- [6] 2022. Debian Mirrors. <https://www.debian.org/mirror/list>. (Accessed on 05/05/2022).
- [7] 2022. dependency-check – File Type Analyzers. <https://jeremylong.github.io/DependencyCheck/analyzers/index.html>. (Accessed on 04/04/2022).
- [8] 2022. DevOps - Wikipedia. <https://en.wikipedia.org/wiki/DevOps>. (Accessed on 05/05/2022).
- [9] 2022. docs - chromium/src.git - Git at Google. <https://chromium.googlesource.com/chromium/src.git/+master/docs>. (Accessed on 04/04/2022).
- [10] 2022. An Eigen-based, light-weight C++ Interface to Nonlinear Programming Solvers. <https://github.com/ethz-adrl/ifoft>. (Accessed on 04/04/2022).
- [11] 2022. Files - debian/master - Debian Multimedia Team / ffmpeg - GitLab. <https://salsa.debian.org/multimedia-team/ffmpeg/-tree/debian/master>. (Accessed on 04/04/2022).
- [12] 2022. Homebrew The Missing Package Manager for macOS (or Linux). <https://brew.sh>. (Accessed on 05/05/2022).
- [13] 2022. International Open Standard (ISO/IEC 5962:2021) - Software Package Data Exchange (SPDX). <https://spdx.dev>. (Accessed on 04/04/2022).
- [14] 2022. OWASP CycloneDX Software Bill of Materials (SBOM) Standard. <https://cyclonedx.org>. (Accessed on 04/04/2022).
- [15] 2022. OWASP Dependency-Track. <https://owasp.org/www-project-dependency-track>. (Accessed on 05/05/2022).
- [16] 2022. rpm.org - Home. <https://rpm.org>. (Accessed on 04/04/2022).
- [17] 2022. SBOM Software Bill of Materials. https://en.wikipedia.org/wiki/Software_bill_of_materials. (Accessed on 05/05/2022).
- [18] 2022. Sonatype Dependency-Check. <https://jeremylong.github.io/DependencyCheck/data/ossindex.html>. (Accessed on 05/05/2022).
- [19] 2022. Windows Package Manager. <https://docs.microsoft.com/en-us/windows/package-manager/>. (Accessed on 05/05/2022).
- [20] Gu Ban, Lili Xu, Yang Xiao, Xinhua Li, Zimu Yuan, and Wei Huo. 2021. B2SMatcher: fine-Grained version identification of open-Source software in binary files. *Cybersecurity* 4, 1 (2021), 1–21.
- [21] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (2021), 1–28.
- [22] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the NPM package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 181–191.
- [23] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 349–359.
- [24] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2169–2185.
- [25] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*. 475–488.
- [26] Chunrong Fang, Xixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 516–527.
- [27] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2021. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* 172 (2021), 110653.
- [28] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A large-scale empirical study on Java library migrations: prevalence, trends, and rationales. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 478–490.
- [29] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. 2020. Interactive, effort-aware library version harmonization. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 518–529.
- [30] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 1–41.
- [31] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsalantalis. 2021. Dependency smells in Javascript projects. *IEEE Transactions on Software Engineering* (2021).
- [32] Lingxiao Jiang, Ghassan Mishserghi, Zhendong Su, and Stephane Gloudu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.
- [33] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.
- [34] Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. 2022. Nufix: Escape From NuGet Dependency Maze. In *2022 International Conference on Software Engineering*. <https://www.microsoft.com/en-us/research/publication/nufix-escape-from-nuget-dependency-maze/>
- [35] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. *arXiv preprint arXiv:2201.03981* (2022).
- [36] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.
- [37] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 106–117.
- [38] André Miranda and João Pimentel. 2018. On the use of package managers by the C++ open-source community. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 1483–1491.
- [39] Nlohmann. 2022. Conan project, nlohmann-json. <https://github.com/nlohmann/json>. (Accessed on 07/25/2022).
- [40] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. *A Qualitative Study of Dependency Management and Its Security Implications*. Association for Computing Machinery, New York, NY, USA, 1513–1531. <https://doi.org/10.1145/3372297.3417232>
- [41] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E Santosa, Asankhaya Sharma, and David Lo. 2021. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering* 26, 4 (2021), 1–34.
- [42] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45.
- [43] Ying Wang, Liang Qiao, Chang Xu, Yepang Liu, Shing-Chi Cheung, Na Meng, Hai Yu, and Zhiliang Zhu. 2021. Hero: On the Chaos When PATH Meets Modules. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 99–111. <https://doi.org/10.1109/ICSE43902.2021.00022>
- [44] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 125–135. <https://doi.org/10.1145/3377811.3380426>
- [45] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 319–330.
- [46] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I Have a Stack Trace to Examine the

- Dependency Conflict Issue?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 572–583. <https://doi.org/10.1109/ICSE.2019.00068>
- [47] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, S.C. Cheung, Hai Yu, Chang Xu, and Zhi-liang Zhu. 2021. Will Dependency Conflicts Affect My Program’s Semantics. *IEEE Transactions on Software Engineering* (2021), 1–1. <https://doi.org/10.1109/TSE.2021.3057767>
- [48] Wireshark. 2022. CMake Project,. <https://github.com/wireshark/wireshark>. (Accessed on 07/25/2022).
- [49] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 860–872.
- [50] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. 2015. How do developers use C++ libraries? An empirical study. In *Proceedings of the Twenty-Seventh International Conference on Software Engineering and Knowledge Engineering*. 260–265.
- [51] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. Modx: Binary Level Partial Imported Third-Party Library Detection through Program Modularization and Semantic Matching. *arXiv preprint arXiv:2204.08237* (2022).
- [52] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2021. On the Impact of Security Vulnerabilities in the npm and RubyGems Dependency Networks. *arXiv preprint arXiv:2106.06747* (2021).
- [53] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.