CNEPS: A Precise Approach for Examining Dependencies among Third-Party C/C++ Open-Source Components

Yoonjong Na
Korea University
Seoul, Republic of Korea
nooryyaa@korea.ac.kr

Seunghoon Woo*
Korea University
Seoul, Republic of Korea
seunghoonwoo@korea.ac.kr

Joomyeong Lee
Korea University
Seoul, Republic of Korea
security428@korea.ac.kr

Heejo Lee*
Korea University
Seoul, Republic of Korea
heejo@korea.ac.kr

ABSTRACT

The rise in open-source software (OSS) reuse has led to intricate dependencies among third-party components, increasing the demand for precise dependency analysis. However, owing to the presence of reused files that are difficult to identify the originating components (*i.e.*, indistinguishable files) and duplicated components, precisely identifying component dependencies is becoming challenging.

In this paper, we present CNEPS, a precise approach for examining dependencies in reused C/C++ OSS components. The key idea of CNEPS is to use a novel granularity called a module, which represents a minimum unit (*i.e.*, set of source files) that can be reused as a library from another project. By examining dependencies based on modules instead of analyzing single reused files, CNEPS can precisely identify dependencies in the target projects, even in the presence of indistinguishable files. To differentiate duplicated components, CNEPS examines the cloned paths and originating projects of each component, enabling precise identification of dependencies associated with them. Experimental results on top 100 C/C++ software show that CNEPS outperforms a state-of-the-art approach by identifying twice as many dependencies. CNEPS could identify 435 dependencies with 89.9% precision and 93.2% recall in less than 10 seconds per application on average, whereas the existing approach hardly achieved 63.5% precision and 42.5% recall.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

Open Source Software Reuse; Supply Chain Security; Third-party Library Dependency; Software Bill of Materials (SBOM).

*Corresponding authors.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639209

ACM Reference Format:
Yoonjong Na, Seunghoon Woo, Joomyeong Lee, and Heejo Lee. 2024. CNEPS: A Precise Approach for Examining Dependencies among Third-Party C/C++ Open-Source Components. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639209

1 INTRODUCTION

The surge of open-source software (OSS) has accelerated third-party OSS reuse, making software development more cost-effective. As the prevalence of Open-Source Software (OSS) reuse grows, dependency tracking becomes increasingly crucial in today's software development environment [3, 40]. Incorrect tracking of dependencies can lead to security threats (*e.g.*, vulnerability issues [1, 48]). In addition, it is particularly important given the increasing significance of uncovering supply chain transparency, as exemplified by Software Bill of Materials (SBOM) [6, 7], which declares the components and pieces that the software was built with.

However, for the following two main reasons, examining dependencies in a precise manner is becoming challenging, especially in C and C++ languages, where components are primarily reused at the code level rather than those managed by package managers [38].

- **Indistinguishable file:** This refers to a source file, or occasionally a function, that is widely used in various OSS projects or cannot be clearly determined as belonging to a specific OSS project (*e.g.*, implementation of cryptographic functions).
- **Duplicated component:** This refers to a case where the same OSS project is cloned in the target project multiple times [12, 34, 46]. As an OSS project usually includes another sub-component, the same OSS project can be duplicated in a target project during the third-party OSS reuse process.

To examine dependencies comprehensively, it is essential to consider the components that indistinguishable files belong to and the dependencies that result from them. In addition, to reduce false positives and negatives (*e.g.*, misidentified or unidentified dependencies), duplicated components should be differentiated and then involved in dependency analysis. This is because unidentified dependencies can cause vulnerabilities to remain untouched, allowing threats to remain in the system [18, 32]. Furthermore, misidentified dependencies can increase the time and effort required to resolve potential security threats in target software [28].

Existing code-based software composition analysis (SCA) techniques (*e.g.*, [10, 16, 31, 38]) overlook indistinguishable files, which rarely contribute to identifying reused components, and duplicated components, thereby producing many false positives and negatives in dependency identification. For example, Centris [38] considers only the unique functions of OSS projects and overlooks the remaining reused files, resulting in missed dependencies. It also does not consider duplicated components, thus yielding false dependencies (see Section 2.3). On the other hand, several existing approaches (*e.g.*, [13, 24, 33, 48]) have attempted to identify dependencies based on metadata (*e.g.*, package manager files). However, in the C/C++ languages, third-party OSS reuse using package managers is not actively pursued [31], and metadata may not exist in the target program because partial OSS reuse is prevalent [38]. Therefore, these approaches are insufficient for identifying dependencies precisely.

Proposing approach. In this paper, we present Cneps (Component dependency scanner), which is a precise approach that can identify dependencies in reused third-party OSS components (at the source code level) while overcoming the aforementioned challenges.

The key idea of Cneps that clearly distinguishes it from existing approaches is the use of a novel granularity called a *module*. A module is a unit comprising a header file and a set of source files that are reusable by importing the header file (see Section 2.1). In other words, a module is a minimum subset of a reusable project that can be reused as a library from another project.

To address indistinguishable files, Cneps performs a module-level dependency analysis. First, Cneps constructs modules from the target project's codebase (see Section 3.1). Subsequently, Cneps inspects the component from which a module has been cloned and generates a module dependency graph (see Section 2.1). This graph comprehensively includes *code dependencies* within a module and *library dependencies* between modules (see Section 3.2). By analyzing modules instead of examining individual files, Cneps can identify the component where the entire module is cloned, thus enabling the precise component identification and dependency examination of modules with indistinguishable files.

Next, Cneps consolidates the generated module dependency graphs to create a dependency graph for the entire input target software (Section 3.3). This is achieved by integrating the nodes (*i.e.*, components) that commonly exist in multiple module dependency graphs. To handle the duplicated component problem, Cneps examines both the (1) cloned paths and (2) originating projects of the target nodes, integrating only the nodes determined to be non-duplicated components. Consequently, a dependency graph of the input software is generated, including components of indistinguishable files, with duplicate components clearly differentiated.

Evaluation. We evaluated Cneps on 100 real-world popular C/C++ projects obtained from GitHub. Although indistinguishable files and duplicate components are prevalent, Cneps successfully identified more than twice as many dependencies than the state-of-the-art code-based SCA technique (*i.e.*, Centris [38]). For the 100 target projects, Cneps identified 534 dependencies with 89.9% precision and 93.2% recall, whereas Centris identified only 345 dependencies with 63.5% precision and 42.5% recall (see Section 4.1). Notably, Cneps could precisely identify the components to which indistinguishable files belong, and differentiate duplicated components with high accuracy (see Section 4.2).

Moreover, when we applied Cneps to 1,000 popular C/C++ software with varying sizes, Cneps could generate a dependency graph in an average of 8.22 seconds (see Section 4.3). The experimental results, in which the elapsed time did not significantly increase even when the code size of the input porject increased, demonstrated that Cneps had sufficient scalability to be used in practice.

Contributions. This paper makes the following contributions:

- We present Cneps, a precise approach for analyzing dependencies in reused OSS components. Notably Cneps can find more dependencies than existing approaches by identifying unnoted components and discovering their dependencies with high accuracy.
- We investigated the impact of indistinguishable files and duplicated components on precise dependency analysis, and devised effective solutions to address these problems.
- Cneps, which has demonstrated higher accuracy and speed in dependency analysis, can be effectively used to examine dependencies in real-world software to prevent potential threats.

2 BACKGROUND

2.1 Term definitions

Basic terms. An *OSS component* refers to a distinct set of codes cloned from an OSS project [16, 31, 38] (called a component for short). *OSS reuse* refers to the process of cloning existing OSS codes or importing OSS code for reuse in a library form. In addition, *component dependency* indicates a state in which an OSS component is reusing the code of another OSS component.

Granularity notations. We then clarify the granularity notations.

- **Files.** A file F refers to a set of one or more functions (f).
$$F = \{f_1,\ f_2,\ \ldots,\ f_n\} \quad (n \geq 1)$$
- **Components.** A component C refers to a set of source files reused from another OSS project.
$$C = \{F_1,\ F_2,\ \ldots,\ F_n\} \quad (n \geq 1)$$
- **Projects.** A project P refers to source code that combines one or more components and a set of files.
$$P = \{C_1,\ C_2,\ \ldots,C_m\} \cup \{F_1,\ F_2,\ \ldots,F_n\} \quad (n, m \geq 1)$$

Building on the aforementioned notations, we introduce new granularity known as header and module for clarity.

- **Headers.** A header file F_h is an importable file that facilitates the reuse of a subset of a project. Let H represent a set of files that can be reused by importing the header file F_h. A file that imports F_h can reuse a set of files listed in H as a library.
$$H = \{F_1,\ F_2,\ \ldots,\ F_n\} \quad (n \geq 1)$$
- **Modules.** A module M is a unit that combines a header file (F_h) with a set of files that are reusable by importing the header (H).
$$M = \{F_h\} \cup H = \{F_h, F_1,\ F_2,\ \ldots,\ F_n\} \quad (n \geq 1)$$
(where F_h and H need to be cloned together)

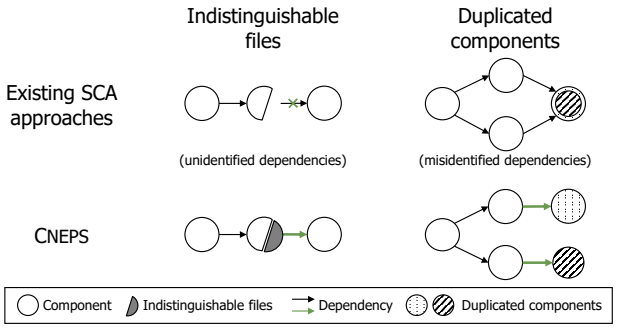

Figure 1: Illustration of challenges in dependency analysis.

Dependency graph. We represent component dependencies by leveraging a graph-based structure. A *dependency graph* (G) is a directed graph, and is represented as $G = (C, E)$, where C denotes components and edge ($E \subseteq C \times C$) indicates the dependency direction. This edge includes dependencies generated by both code and library (see Section 2.2). Note that a dependency graph can contain cycles. An example dependency graph is illustrated in Figure 2.

2.2 Problem statement

In this paper, we address the problem of analyzing dependencies among components. To express this more formally, let C_1 and C_2 be two components in the target project. We can define the conditions for dependency between two components as follows.

> ◇ **Dependency between two components.**
> We define a component C_1 has dependency on C_2 if C_1 and C_2 satisfy one of the following two conditions.
> **(1) Code dependency.** If a function f in C_2 is cloned in a file F in C_1, we determine that C_1 has a dependency on C_2.
> **(2) Library dependency.** If a file F in C_1 imports a header file F_h from another component C_2, we conclude that C_1 has a dependency on C_2.

However, identifying dependencies between components is difficult owing to the following two main challenges.

Indistinguishable files. Indistinguishable files refer to files that exist widely in OSS projects, or cannot be clearly determined as cloned from a specific component (*e.g.*, cryptographic functions and hash table). To analyze the dependencies precisely, the components of indistinguishable files and the dependencies that arise from these files should be considered. However, indistinguishable files are widely present in various projects, making it difficult to identify their components and thus complicating dependency analysis. Existing SCA approaches (*e.g.*, [10, 16, 31, 38]) fail to properly determine the components of indistinguishable files, thereby missing existing dependencies. Figure 1 (*i.e.*, indistinguishable files) illustrates the missing dependency caused by indistinguishable files.

Duplicated components. Duplicated components exist when the same project is reused in the target project multiple times [12, 34, 46]. In general, an OSS project contains many sub-components within its codebase. Therefore, even when reusing a single OSS, in practice, many other sub-components are also reused and a certain OSS project may be unintentionally reused multiple times in the target project. Existing SCA approaches do not distinguish between duplicated components, resulting in false positives and negatives in the dependency identification process. Figure 1 (*i.e.*, duplicated components) illustrates the false positive dependency produced by duplicated components.

Table 1: *MongoDB* dependency identification results of CNEPS and CENTRIS. Note that 4,000 indistinguishable files were detected in the *MongoDB* codebase.

	CENTRIS	CNEPS
#Total Dependencies	77	
#Identified reused files	2,463	**6,463**
#Identified components	30	**40**
#Identified dependencies	40	**82**
#Correct dependencies (TP)	30	**72**
#Incorrect dependencies (FP)	10	**10**

2.3 Motivating Example

To describe the importance of addressing the challenges, we introduce a motivating example, the *MongoDB* case (latest version as of July 2023). We applied CNEPS and CENTRIS (*i.e.*, a code-based SCA approach [38]) to *MongoDB* and examined the dependencies identified by each approach (the method of applying CENTRIS to dependency identification is introduced in Section 4.1 in detail).

Figure 2 illustrates a part of the *MongoDB* dependency graph identified by CNEPS, where the solid black lines indicate dependencies discovered in both approaches and the green dotted lines indicate dependencies identified only by CNEPS.

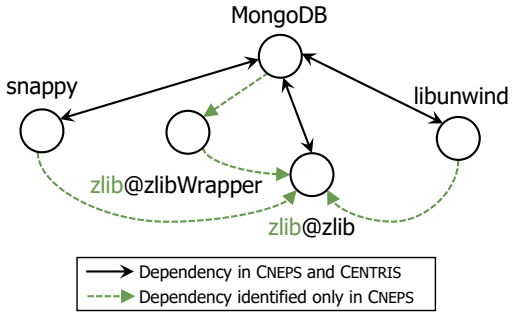

Figure 2: Depiction of a part of MongoDB dependency graph.

CENTRIS. Because CENTRIS did not consider the indistinguishable file and duplicated component problem, it misidentified several dependencies (75% precision) and also overlooked a number of correct dependencies. In the case of CENTRIS, most of the indistinguishable files that exist in *MongoDB* were not clearly identified as reused code in a certain component. As shown in Table 1, CENTRIS was only able to identify 2,463 OSS files (38.1%) reused in *MongoDB*, which prevented it from identifying dependencies that actually exist in *MongoDB* (*i.e.*, CENTRIS missed at least 42 dependencies).

In addition, Centris failed to address duplicated components, yielding false positives and negatives. As shown in Figure 2, the *zlib* OSS was reused twice in *MongoDB*: the original *zlib* and modified *zlib* (called *zlibWrapper*). Interestingly, the *MongoDB* team was reusing *zlibWrapper*, which imports *zlib* as a library; that is, *zlibWrapper* is linked to reuse *zlib*. Owing to these duplicate components, Centris did not correctly identify dependencies on *zlib*, reporting false positives and negatives.

Our proposed Cneps approach. By overcoming both challenges, Cneps discovered 42 more dependencies (*i.e.*, 72 dependencies) in *MongoDB* than Centris. In particular, by leveraging module granularity, Cneps could address the indistinguishable file issue and thus identified 62% more reused files (*i.e.*, 4,000 additional reused files) than Centris. Thus, Cneps discovered two more dependencies that existed in the indistinguishable files. For example, one of the indistinguishable file in the *snappy* component had a dependency on *zlib*, therefore, Centris failed to identify this dependency.

Moreover, through effective dependency graph generation and consolidation techniques, Cneps could handle duplicate components, resulting in the discovery of 10 more dependencies than Centris. For example, three components in *MongoDB* were reusing *zlib*; if *zlib* and *zlibWrapper* are not clearly distinguished, this leads to misidentified or unidentified dependency issues. Cneps could differentiate these duplicate components, thus identifying more dependencies than Centris with higher accuracy.

3 DESIGN OF CNEPS

In this section, we describe the design of Cneps, which is a precise approach for identifying dependencies in C/C++ projects.

Overview. Figure 3 shows the workflow of Cneps, which comprises three phases: *module construction* (P1), *dependency graph generation* (P2), and *graph consolidation* (P3). In P1, Cneps constructs modules from the input source code project. In P2, Cneps generates a module dependency graph for each constructed module by considering the dependencies established by both code and library dependencies. By leveraging module granularity, Cneps can identify the components in which indistinguishable files have been reused and their dependencies. In P3, Cneps consolidates the module dependency graphs, thereby completing the dependency graph of the input software. Here, Cneps addresses duplicate components to generate a more precise dependency graph.

3.1 Module construction (P1)

To identify the dependencies between components, Cneps first aims to systematically construct *modules* for a given input software.

3.1.1 ***Module construction.*** We first clarify the term "*module*".

> ◇ **Module granularity.**
> We define the module M as a unit comprising a header file (F_h) and files that are reusable by importing the header (H). Note that these reusable files should be cloned together for reuse and can be leveraged by importing the header file (*e.g.*, using the "`#include`" command in C/C++ languages).

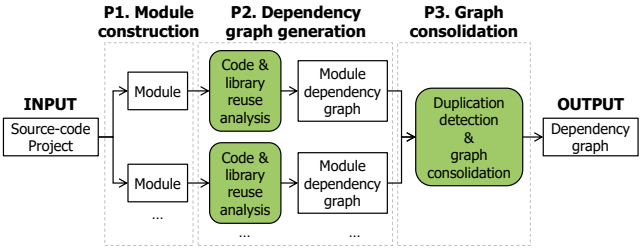

Figure 3: High-level workflow of Cneps.

Module construction comprises the following three steps.

- First, Cneps parses all functions in a given target source code project. In particular, Cneps parses the declarations and definitions of all functions in the input software with the extensions: `c`, `cc`, `cpp`, `h`, `hpp`, and `hxx`.
- Next, Cneps examines each source file to determine whether it is a header file (F_h) by examining the presence of function declarations (f_{dec}).
- Using the function name and its parameters, Cneps attempts to detect the corresponding function definitions. If a file F contains a definition (f_{def}) for f_{dec}, Cneps bundles F_h and F into one module. Here, a function may exist only in the file F with its definition, and its declaration may not be present in other files. In this case, Cneps classifies F as both a header and a reusable file: Cneps bundles F into a single module because functions in F remain reusable by importing F.

As a working example, we introduce a *c-ares* case that is reused in *MongoDB* (see Figure 4). When Cneps parses the **`ares.h`** (*i.e.*, header file) in the *MongoDB*, we can identify many function declarations, including the following three functions: **`ares_gethostbyaddr`**, **`ares_timeout`**, and **`ares_free_string`**.

Next, Cneps discovers the definition of each function; in the working example, the definitions for corresponding functions are contained in **`ares_gethostbyaddr.c`**, **`ares_timeout.c`**, and **`ares_free_string.c`**, respectively. Therefore, Cneps bundles the header file (**`ares.h`**) and the aforementioned three files, each containing the definition of a function, to create a single module. As a result, a single module is generated, which can be reused as a standalone library by importing the header file to access the following three functions: **`ares_gethostbyaddr`**, **`ares_ timeout`**, and **`ares_free_string`**.

3.2 Dependency graph generation (P2)

Cneps generates dependency graphs using the constructed modules. Cneps examines code dependency through an internal examination of modules (Section 3.2.1) and inspects library dependency by identifying components that import a header file of modules (Section 3.2.2). Finally, Cneps generates dependency graphs (Section 3.2.3).

3.2.1 ***Code dependency analysis.*** First, Cneps examines the functions within a module to identify the components from which this module is cloned. Here, Cneps leverages Centris [38], an approach for identifying reused C/C++ components in a target project. Using Centris, we can identify whether functions in the target project were cloned from other OSS; in other words, we can identify (1) the functions reused from other OSS and (2) their originating

Listing 1: Example of the declarations for the three functions in `ares.h` of *MongoDB*.

```
...
CARES_EXTERN void
  ares_gethostbyaddr...;
CARES_EXTERN struct timeval
  *ares_timeout...;
CARES_EXTERN void
  ares_free_string...;
...
```

Listing 2: `ares_gethostbyaddr` function defined in `ares_gethostbyaddr.c` of the *c-ares* component.

```
void
  ares_gethostbyaddr
  (/* Paremeters */)
{
  struct addr_query *aquery;
  ...

}
```

Listing 3: `*ares_timeout` function defined in `ares_timeout.c` of the *Curl* component.

```
struct timeval
  *ares_timeout
  (/* Paremeters */)
{
  struct query *query;
  ...

}
```

Listing 4: `ares_free_string` function defined in `ares_free_string.c` (indistinguishable file).

```
void
  ares_free_string
  (/* Paremeters */)
{
  ares_free(str);
  // a single line function
}
```

Figure 4: An example of module construction for the `ares.h` file in the *c-ares* component reused in *MongoDB*.

OSS projects. Note that Centris is only used to identify the components of distinguishable files and cannot identify components of indistinguishable files or examine component dependencies.

Then, Cneps counts the number of functions cloned into the module from each component. Based on the counting results, we determine that the component with the highest number of reused functions is the one from which the entire module was cloned (called the prevalent component). As an additional validation to support this decision, after initially identifying the prevalent component of a module, Cneps verifies that the file F containing function declarations is present in the codebase of the prevalent component. If not, Cneps performs the same task with the second-most prevalent component. While repeating this task, the component discovered first, which includes F, becomes the component of this module. Because indistinguishable files have been cloned together from the codebase of the module, Cneps identifies the component of indistinguishable files with the same component of the module (further discussion on this method is provided in Section 5).

For example, among the three functions declared in the **`ares.h`** file (see Figure 4), Cneps can identify that **`ares_gethostbyaddr`** was cloned from *c-ares* and that **`ares_timeout`** was cloned from *Curl*. The **`ares_free_string.c`** file including the single line **`ares_free_string`** function does not belong to any component identified by Centris in *MongoDB*; thus, Cneps determines that it is an indistinguishable file (*i.e.*, its originating project is unknown). In particular, the **`ares.h`** file in the module contains 23 functions cloned from *c-ares*, seven functions reused from *Curl*, and 34 indistinguishable functions. After verifying that the **`ares.h`** file is included in the *c-ares* codebase, Cneps determines this module is cloned from *c-ares* and has a dependency on *Curl* as code dependency. Because indistinguishable files are cloned together with the module, Cneps identifies that the component of indistinguishable files is *c-ares*.

3.2.2 ***Library dependency analysis.*** In addition to reused functions at the code level, there are cases of reusing functions by importing the header files of the module (*e.g.*, using the "`#include`" command). For all files belonging to a module, Cneps examines whether a file is importing other header files as a library, which can be performed by parsing strings. Note that the previous component identification was performed on files within one module, whereas library dependency analysis is performed between modules.

One issue we should consider is that there can be multiple files with the same name in the input software codebase. For example, when a library import statement named "`#include <zlib.h>`" is found in file F of *MongoDB*, there may actually be more than one

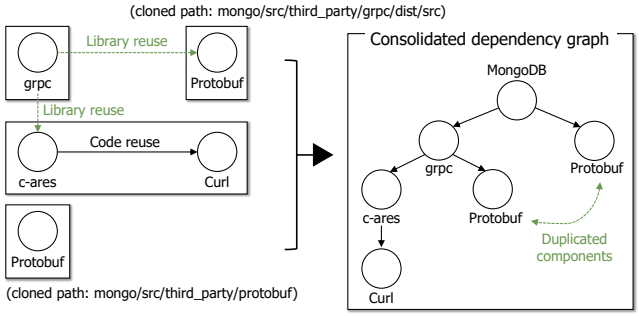

Figure 5: Illustration of graph consolidation in *MongoDB*.

`zlib.h` file in the *MongoDB* codebase. In this case, it is necessary to correctly determine which file to link with. To address this, we follow the resolving rules of the GNU GCC compiler [5]. That is, for every **`zlib.h`**, Cneps compares the paths of F and **`zlib.h`**. For all detected **`zlib.h`** occurrences, Cneps determines that **`zlib.h`** with the path closest to the path of F is an imported file.

3.2.3 ***Module dependency graph generation.*** By combining the list of components, previously identified code dependencies, and the relationships discovered through library dependency, Cneps generates module dependency graphs.

Establishing code dependencies. First, for each module, Cneps organizes a parent node representing the module's component (see Section 3.2.1). Next, based on the code dependency information in a module, Cneps identifies the components that become child nodes; Cneps connects the dependency (*i.e.*, edge) from the parent node to the child nodes. If the file F where function declarations are declared does not exist any component, Cneps does not find parent-child relationships between components and does not connect edges between nodes (*i.e.*, independent nodes). In our working example, two nodes are initially created: *c-ares* and *Curl*. Since we confirmed that *c-ares* is a module component in Section 3.2.1, *c-ares* becomes the parent node and *Curl* becomes the child node. Thus, the dependency from *c-ares* to *Curl* is established (*c-ares* → *Curl*).

Establishing library dependencies. Next, Cneps considers library dependency. This task involves connecting two nodes in different modules. Suppose a file F reused in a component C_1 imports a header file F_h of component C_2 (*i.e.*, `#include` <F_h>). Then, the dependency is connected from node C_1 of the module to which F belongs to node C_2 of the module to which F_h belongs. An example of a dependency establishment is shown in Figure 5.

After performing this task on all modules, several module dependency graphs of the input target software are generated. However, because there are overlapping and unconnected nodes (*e.g.*, dead code), a single dependency graph for the input software is created by consolidating the module dependency graphs in P3 (Section 3.3).

One of the biggest advantages of CNEPS is that it does not overlook the dependencies generated by indistinguishable files. If we identify components and dependencies based on each reused file or function (*i.e.*, using finer-grained granularity), we may miss some dependencies because we cannot specify the components of indistinguishable files. However, the module unit leveraged in CNEPS is a combination of a header file and a set of files that are reusable by importing. In other words, the module is the smallest subset of the project required to operate as a standalone library. Subsequently, by analyzing components and dependencies in groups rather than individual files or functions, we can identify the component where the entire module is cloned (*i.e.*, the prevalent component); hence, even the dependencies of modules including indistinguishable files can be identified.

3.3 Graph consolidation (P3)

Finally, CNEPS constructs a dependency graph for the input project by consolidating the constructed module dependency graphs.

Although a straightforward approach could be applied here (*i.e.*, simply merging identical components), this may result in the production of false alarms in the dependency analysis owing to duplicated components (see Section 2.3).

To overcome this problem, CNEPS uses the following approach. Essentially, the same nodes (*i.e.*, components) that exist in different module dependency graphs are merged into one node. Herein, CNEPS identifies whether these components are identical components sharing the same codebase or duplicated components utilizing different codebases.

To this end, CNEPS uses two features: (1) a cloned path and (2) an originating project (*i.e.*, a parent component). Suppose the two components to be compared are C and C'. Figure 6 illustrates the flow chart of the graph consolidation process.

3.3.1 ***Cloned path comparison (S1).*** CNEPS first compares the directory paths where C and C' are cloned in the input target software. Note that a node of a graph contains a list of reused files (Section 3.2.1). Here, instead of considering each file path, CNEPS considers *a common path* for all reused files in each component. We define a cloned path of a component as follows.

> ◇ **Cloned path of a component.**
> We define the cloned path of the component C as the longest common directory path among the reused file sets of C.

For example, suppose component C contains three reused files (*e.g.*, `target/C/src/a.c`, `target/C/src/b.c`, and `target/C/src/temp/c.h`). CNEPS splits the file path based on "/", and then compares each level from the root directory to the file path. Then, CNEPS extracts the common path with the longest length (*i.e.*, `target/C/src/`). This common path becomes the cloned path of C.

If the cloned paths of C and C' are the same or have an inclusion relationship (*e.g.*, $path(C) \subset path(C')$), CNEPS determines that the two components are cloned in the same path and are the same component. Therefore, in the process of graph consolidation, the two nodes C and C' are merged into one node. If not, their originating projects are compared in the next step.

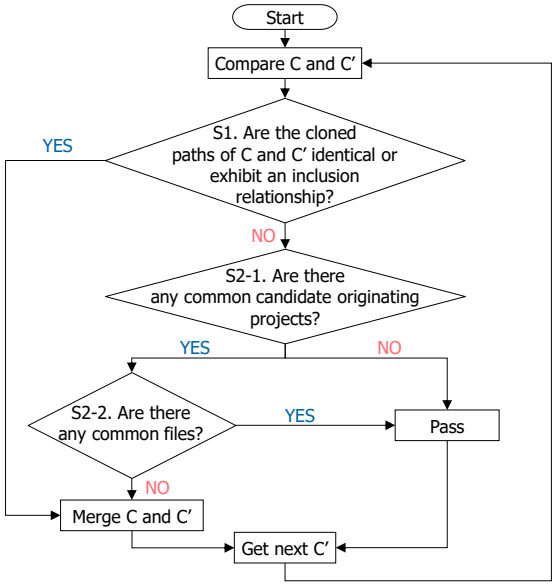

Figure 6: Illustration of the graph consolidation flow chart.

3.3.2 ***Originating project comparison (S2).*** Thereafter, CNEPS compares the originating projects of the two nodes (components) to examine whether they are duplicated components.

In a code reuse relationship where a component is a child node (*e.g.*, *Curl* in Figure 5), the originating project can be easily identified as the parent node (*i.e.*, *c-ares* in Figure 5). In this case, if the originating projects (P) of C and C' are identical, both C and C' are considered to be cloned together with P as sub-components, and thus CNEPS merges C and C' into one node.

However, when a component has a "library dependency" (*e.g.*, *Protobuf* in Figure 5), it is difficult to clearly identify an originating project because numerous nodes may have dependencies on the corresponding component. To address this issue, CNEPS identifies candidate originating projects for the component. CNEPS selects nodes that satisfy the following two conditions as the originating project candidates for a component: (1) they have a dependency on the target component through a library dependency relationship, and (2) they have a cloned path that is the same as or has an inclusion relationship with the cloned path of the component.

Thereafter, CNEPS verifies whether C and C' are duplicated components as follows.

S2-1. First, CNEPS examines whether the same component exists in the candidate originating projects of C and C'. If so, CNEPS performs **S2-2** operations.

S2-2. CNEPS then examines whether there are common files between C and C' (based on the file name) as additional verification. If C and C' are the same component, we determine that the same file name has already been created as one node during the module construction phase. Nevertheless, if C and C' contain the

same file name, CNEPS considers them duplicated components and does not merge the C and C' nodes in graph consolidation.

For example, the module dependency graphs in Figure 5 have two *Protobuf* nodes. Here, we confirmed that their originating projects are different (*i.e.*, *grpc* and *MongoDB*), the cloned paths are different, and even common files exist for both *Protobuf* components. Therefore, CNEPS determines that *Protobuf* nodes are duplicated components, and does not merge them (see consolidated dependency graph in Figure 5). This process is performed for all nodes with the same name until no more components are mergeable. Finally, CNEPS determines the consolidated dependency graph as the dependency graph of the input software.

4 EVALUATION

In this section, we evaluate CNEPS. In Section 4.1, we evaluate the accuracy of CNEPS. In Section 4.2, we evaluate the effectiveness of CNEPS and demonstrate the need to address indistinguishable file and duplicated component problems. Finally, Section 4.3 investigates the performance and scalability of CNEPS. We executed CNEPS on a machine equipped with an AMD Ryzen 9 3900X 12-Core 3.8GHz Processor, 64GB RAM, and 1TB SSD, running Ubuntu 22.04.

CNEPS architecture. CNEPS comprises two subsystems: (1) a *dependency graph generator* and (2) a *graph consolidator*. The dependency graph generator constructs modules and generates a dependency graph for each module. Here, CENTRIS [38] is used to identify reused functions, and Ctags [4] is leveraged to parse functions. The graph consolidator merges the generated dependency graphs and discovers duplicated components. CNEPS is implemented with approximately 2,500 lines of Python code, excluding external libraries. The source code of the CNEPS is publicly available[1].

4.1 Accuracy of CNEPS

Dataset. To evaluate the accuracy of CNEPS, we selected target projects based on the following criterion: real-world OSS to which many people actively contribute, thus having a high probability of containing complicated dependencies. We chose GitHub, one of the most popular OSS hosting services. Based on the number of stargazers (*i.e.*, a popularity indicator on GitHub), we selected the top 100 C/C++ OSS as the target software. Next, we constructed an OSS dataset to identify dependencies by analyzing the dependencies in the target project. We leveraged a dataset composed of all functions from all versions of more than 10,000 popular C/C++ OSS projects on GitHub, provided by CENTRIS.

Evaluation methodology. The accuracy of CNEPS is measured by examining the edges (*i.e.*, dependencies) in the dependency graphs. We compared CNEPS with a state-of-the-art approach (*i.e.*, CENTRIS [38]), which is capable of discovering components that a source code project depends on. Because the output of CENTRIS is not a graph-based structure, we implemented a tool that uses CENTRIS results to generate a graph, where nodes indicate the components identified by CENTRIS and edges represent the dependencies between the input software and all identified components.

[1]https://github.com/sodium49/CNEPS-public

CENTRIS provides dependencies between the target project and its components but does not provide dependencies between reused components. To ensure fair comparisons, we implemented the same principle that CENTRIS used to establish the ground truth for identifying dependencies between components: we considered component dependencies identifiable in CENTRIS through reused file path analysis. For example, suppose that CENTRIS identifies C and C' components in the target project. If the names of C' are included in any paths of reused files identified in C (*e.g.*, **`target/C/thirdParty`**`/C'/a.c`), then we determine that C has a dependency on C', and the two nodes are connected in the graph ($C \rightarrow C'$). For each target project, we compared the dependency graphs generated by CNEPS and CENTRIS and measure the accuracy of each tool.

To evaluate accuracy, we used the following five metrics: true positives (*TP*), false positives (*TP*), false negatives (*FN*), precision ($P = \#TP/(\#TP + \#FP)$), and recall ($R = \#TP/(\#TP + \#FN)$). FPs indicate the misidentified dependencies between two components (*i.e.*, one component does not depend on the other). In contrast, FNs refer to unidentified dependencies (*i.e.*, one component depends on another component but the dependency is not established). We only consider FNs in which the components are included in the graph but the correct dependency between the components does not exist.

Specifically, we implemented a monitoring system to verify the results of CNEPS. This system tracks two main aspects: (1) source files involved in dependencies (code and library dependencies), and (2) every header file that each component's source file attempts to import (*i.e.*, using `#include` directives); the former are monitored to verify FPs, and the latter are investigated to assess FNs of CNEPS. The detection results of CNEPS were manually verified by two security analysts over two days. We analyzed the results of CNEPS by examining the files tracked by the monitoring system and also manually investigating the source code of components. Finally, the results were evaluated through cross-checking.

Overall result. Table 2 and Figure 7 summarize the dependency examination results of CNEPS and CENTRIS. In general, as the number of nodes in the dependency graph increases, the dependency relationship becomes more complex, and more edge-case dependencies occur. As a result, the dependency identification accuracy of each tool decreases slightly. Nevertheless, we observed that CNEPS significantly outperformed CENTRIS; CNEPS identified 534 dependencies from the 100 target projects with 89.9% precision and 93.2% recall, whereas CENTRIS discovered 345 dependencies with 63.5% precision and 42.5% recall. Table 3 shows dependencies identified by CNEPS and CENTRIS from target projects with 10 or more nodes in their graph.

Accuracy of CENTRIS. We observed that CENTRIS overlooked many existing dependencies in the target projects (296 FNs). The main reason is that CENTRIS does not consider indistinguishable files and duplicated components; these challenges respectively led to 57 and 50 cases of CENTRIS failing to detect dependencies in the target projects (see Section 4.2). Also, CENTRIS is an approach specialized in detecting code dependency; its inability to detect dependencies caused by library dependency also contributed to the high number of FNs.

Table 2: Dependency examination accuracy of CNEPS and CENTRIS for the 100 target projects.

Approach	Graph classification*	#Included nodes	#Identified reused files	#Identified dependencies	#TPs	#FPs	#FNs	Precision	Recall
CENTRIS	*Small*	68	8,843	17	11	6	0	64.7%	100%
	Moderate	21	7,741	102	56	46	52	54.9%	51.9%
	Large	11	23,998	226	152	74	244	67.3%	38.4%
	Total	100	40,582	345	219	126	296	63.5%	42.5%
CNEPS	*Small*	68	18,212	11	11	0	0	100%	100%
	Moderate	21	15,160	108	106	2	2	98.1%	98.1%
	Large	11	41,821	415	363	52	33	87.5%	92.8%
	Total	**100**	75,193	**534**	**480**	**54**	**35**	**89.9%**	**93.2%**

*: Graph classification according to the number of nodes contained in the graph (*Small*: one node; *Moderate*: 2-9 nodes; *Large*: 10 or more nodes).

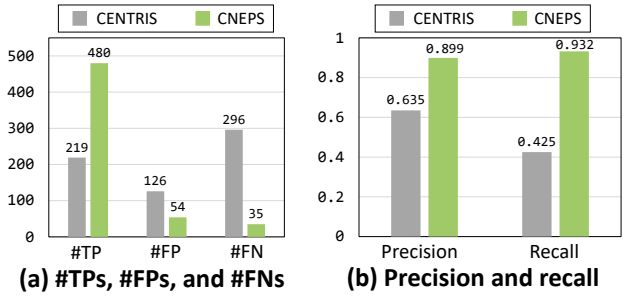

Figure 7: #TPs, #FPs, #FNs, precision, and recall of each tool.

Moreover, CENTRIS generated 126 FPs in dependency identification (63.5% precision). Many FPs appeared because CENTRIS did not differentiate duplicated components, as we argued throughout this paper. Also, if a component is a sub-component of another component, CENTRIS did not consider this case, thus generating FPs. For example, in the *MongoDB* case in Figure 5, while *MongoDB* has a dependency on *grpc*, it did not have a direct dependency on *c-ares* and *Curl*, which are sub-components of *grpc*. In this case, CENTRIS misidentifies that *MongoDB* has direct dependencies on all of *grpc*, *c-ares*, and *Curl*, resulting in producing FPs.

Accuracy of CNEPS. CNEPS could achieve higher dependency analysis accuracy than CENTRIS by effectively addressing indistinguishable files and duplicated components (see Section 4.2).

Although CNEPS outperforms the existing approach, several FPs and FNs occurred. The main cause of FPs and FNs is an error that occurs during the module construction process. As will be introduced in detail in Section 4.2, an error that occurs in module construction can result in incorrectly identifying the component of indistinguishable files and failing to differentiate duplicated components, thus generating FPs and FNs in dependency analysis. Also, FPs and FNs were produced owing to errors in resolving the exact target of "`#include`" directives. Our method of detecting the target header file according to the rules of the GNU GCC compiler (see Section 3.2) may generate false alarms if the target header file is not included in the codebase (*e.g.*, system library) or if there are multiple header files with the same name. This can unintentionally cause false alarms in library dependency analysis.

Table 3: Dependency examination results of CENTRIS and CNEPS for 11 large projects.

Project	#Total Dependencies	CENTRIS		CNEPS	
		#Nodes*	#Deps**	#Nodes*	#Deps**
MongoDB	77	30	30	40	72
Hhvm	72	16	26	27	63
Linux	62	11	18	22	59
Tasmota	38	21	18	31	31
Srs	32	8	10	14	28
Godot	29	16	16	20	28
Winget-cli	21	12	9	12	18
Emscripten	20	7	8	16	19
Radare2	18	10	8	10	18
OpenCV	16	10	2	16	16
Php-src	11	4	7	9	11
Total	396	145	152	**217**	**363**

*: The number of nodes when considering only correctly identified components. CNEPS successfully differentiates duplicated components while CENTRIS does not.
**: The number of correctly identified dependencies.

Case study: A dependency in the Linux kernel. In real-world software, dependencies are not always one-directional, posing challenges for dependency analysis. Here, we present a case detected by CNEPS where a dependency changes during the reuse of other components. As shown in Listing 5, the **`lz4.c`** file, whose originating project is *lz4*, was reused in the *Linux kernel*. Because the *Linux kernel* reused *lz4*, the dependency from the *Linux kernel* to *lz4* was essentially created (*i.e.*, *kernel* → *lz4*). However, after this **`lz4.c`** file was reused, it was modified to have dependencies on header files existing in the *Linux kernel*(*i.e.*, **`init.h`** and **`lz4.h`**), resulting in a newly generated dependency of opposite direction (*i.e.*, *lz4* → *kernel*). If only code dependencies are considered, the dependency of *lz4*→*kernel* (*i.e.*, library dependency) would go undetected. On the other hand, if only library dependencies are considered, the dependency of *kernel*→*lz4* would not be identified. By considering both dependencies and by leveraging module granularity, CNEPS can precisely analyze component dependencies even in such situations where dependencies are intricately woven.

Listing 5: Code snippet of the `lz4.c` file reused in the *Linux kernel*.

```
...
#include <linux/init.h>
#include <linux/lz4.h>
...
```

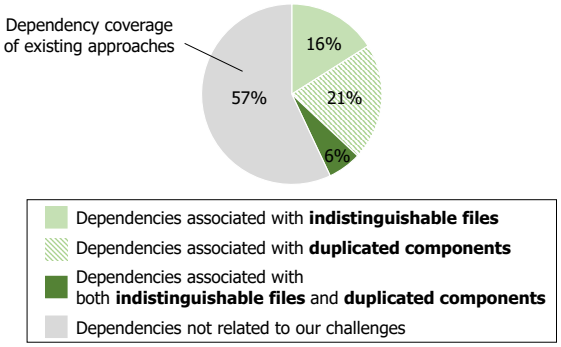

Figure 8: Distribution of dependencies affected by indistinguishable files and duplicated components in 1,000 popular GitHub projects.

4.2 Effectiveness of CNEPS

To demonstrate the importance of addressing *indistinguishable file* and *duplicated component* problems in dependency analysis, we performed the following two evaluations for each challenge.

(1) Impact of indistinguishable files and duplicated components. We examined the ratio of dependencies that were affected by indistinguishable files and duplicated components.

(2) Efficacy of CNEPS. We evaluated the accuracy of CNEPS in identifying components for indistinguishable files and differentiating duplicated components.

4.2.1 ***Impact of our challenges in OSS ecosystems.*** We first investigated the impact of two challenges that make dependency analysis difficult in real-world OSS ecosystems. To do this, we collected 1,000 popular C/C++ OSS projects from GitHub (based on the number of stargazers) and executed CNEPS on the collected projects. Here, rather than measuring the accuracy of dependency detection, we focused on identifying the distribution of dependencies existing in indistinguishable files and duplicated components.

Figure 8 shows the examination results. Notably, we observed 22% (*i.e.*, 16%+6%) of dependencies are detected from indistinguishable files, and 27% (*i.e.*, 21%+6%) of dependencies are detected from duplicated components. We confirmed that 6% of the dependencies could only be identified correctly by resolving both challenges. In other words, existing approaches that do not overcome the two challenges we present only identify up to 57% of total dependencies; even this can only be achieved by considering both code and library dependencies comprehensively.

Therefore, the aforementioned two challenges should be addressed in dependency analysis. Notably, CNEPS demonstrated its high accuracy by effectively resolving these challenges during the accuracy evaluation. In the following sections, we describe a more detailed analysis of the effectiveness of CNEPS in resolving both challenges.

4.2.2 ***Addressing indistinguishable files.*** First, we introduce the number of dependencies related to indistinguishable files that appeared in the accuracy evaluation and how effectively CNEPS resolved them. Table 4 summarizes the results of experiments.

Table 4: Accuracy of CNEPS in identifying components of indistinguishable files.

Approach	#Identified reused files	#Identified indistinguishable files	#TP†	#FP‡	Precision
CNEPS	75,193	34,611	31,681	2,930	91.5%

†: CNEPS identifies the correct component for the indistinguishable files.
‡: CNEPS identifies the incorrect component for the indistinguishable files.

Table 5: Accuracy of CNEPS in differentiating duplicated components.

Approach	#All components	#Unique components	#Duplicated components	#TP†	#FP‡	Precision
CNEPS	297	257	40	33	7	82.5%

†: CNEPS successfully differentiates duplicated components.
‡: CNEPS does not merge nodes even though they are not duplicated components.

We confirmed that a total of 34,611 indistinguishable files (46% of all reused files) were discovered in the 100 target projects in Section 4.1. It is worth noting that the dependencies generated from these indistinguishable files account for 11% (57 dependencies) for all identified dependencies.

CNEPS could identify the correct components of the indistinguishable files in most cases (91.5%) by considering module-level granularity (see Section 3.2). In particular, the approach of CNEPS to detect prevalent components of a module was demonstrated to be highly effective in practice.

However, CNEPS produced several FPs. The FPs mainly occur when the definition of a function is misidentified in module construction. During module construction, CNEPS detects the definition of a function using the function names and parameters obtained from function declarations. However, if there are multiple functions with the same function name and parameters, errors may arise in the module construction process, leading to the incorrect identification of components for indistinguishable files.

Because the dependencies generated from these indistinguishable files account for a considerable portion of the 100 target projects, we can demonstrate the efficiency of CNEPS, which is capable of identifying the components to which indistinguishable files belong and their dependencies.

4.2.3 ***Addressing duplicated components.*** We then evaluate the efficacy of CNEPS in terms of addressing duplicated components.

Because there is no ground truth for duplicated components, we determined TPs and FPs using the cloned path of each component (see Section 3.3). For example, suppose CNEPS determined that two *zlib* projects are duplicated components. If the cloned paths of two *zlib* projects are different, we determine them as duplicated components because they were cloned from different projects. However, if the cloned paths are the same but CNEPS identifies the two *zlib* projects as duplicated components, we consider this case an FP. Also, we excluded components that are identified from the input project (*e.g.*, *MongoDB* in *MongoDB*) as they can not be duplicated.

Table 5 shows the evaluation results. In the 100 target projects, CNEPS identified 297 components, of which 257 were unique components; 40 were determined to be duplicated. Among them, we

confirmed that 33 components (82.5%) were actually duplicated (TPs). By differentiating these 33 duplicated components, Cneps identified 50 more dependencies compared to Centris in Section 4.1.

The main cause of FPs, similar to the reason for FPs of the indistinguishable file problem, was the incorrect identification of function definitions during module construction. Incorrect identification of the function definition results in the wrong file being included in the module, which causes the cloned path of the module to be incorrectly identified, thereby producing an error in differentiating duplicated components. Although there were some FPs, Cneps could precisely differentiate duplicated components in most cases, which allowed it to identify more dependencies, thereby demonstrating its efficacy.

4.3 Performance and scalability

We then evaluate the performance and scalability of Cneps. To do this, we executed Cneps on 1,000 popular GitHub C/C++ projects (based on the number of stargazers) with varying code sizes, ranging from less than 10 to more than 31 million Lines of Code (LoC). We measured the total time elapsed for Cneps to generate a dependency graph for an input C/C++ source code project, excluding the time taken for executing Centris.

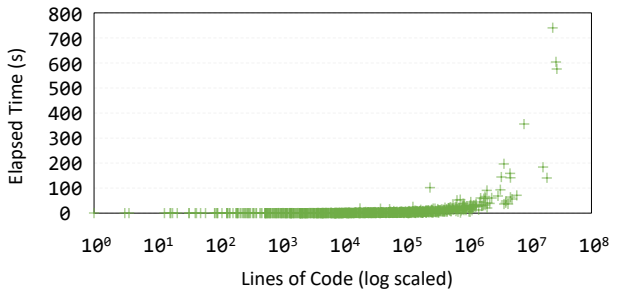

Figure 9: Execution time of Cneps on 1,000 target projects.

Figure 9 shows the measurement results. Remarkably, we observed that Cneps took an average of 8.22 s to examine the dependencies of an input project (the median value was 1.12 s). The scalable design of Cneps enables rapid dependency analysis; Cneps scans the target project codebase only once during the dependency analysis. In this scanning process, Cneps extracts all necessary information, including functions, file paths, and "`#include`" directives. Additionally, Cneps utilizes a high-performance function parser (*i.e.*, Ctags). Because of its scalable design, Cneps can analyze the dependencies of the target project within an average of 10 s. It is noteworthy that the time required for Cneps did not increase significantly even when the code size of the input project increased. The fact that Cneps could identify dependencies between components within 10 s even with millions of LoC input projects indicates that Cneps has sufficient speed for practical use.

5 DISCUSSION

In this section, we discuss several considerations related to Cneps, including its applications and limitations.

Application. We first discuss the possible applications of Cneps: vulnerability management and SBOM generation.

- **Vulnerability management:** Many existing approaches have attempted to detect vulnerable codes propagated by third-party OSS reuse (*e.g.*, [19, 36, 41]). However, in practice, not every identified vulnerable code is prioritized for management at the moment of disclosure, especially if it has not been verified as exploitable [18, 43, 44]. Currently, exploitability assessments rely on manual analysis by security experts [40], necessitating the need for a rapid triage approach. Given the importance of precise dependency analysis in exploitability assessments [18], the inclusion of Cneps in the vulnerability detection process allows for swift triage in vulnerability management.
- **SBOM Generation:** The challenge of providing accurate dependency information often complicates the SBOM (Software Bill of Materials) generation [2, 7, 40]. Cneps, delivering precise dependency details for a specified source code project, stands out as a valuable tool in this regard. It can significantly enhance the accuracy and reliability of the SBOM, ensuring that component dependencies are precisely represented.

Module component identification. When Cneps identifies the component where a module is cloned, it investigates the number of reused files for components associated with the module. Consequently, the component to which the highest number of reused files belong is determined as the prevalent component of the module. To reinforce the rationale behind our decision, we perform additional verification to ensure that the file containing function declarations in the module belongs to the prevalent component (see Section 3.2). However, in practice, the prevalent component of the module may be incorrectly determined even though the two conditions are satisfied. Although this case has not been observed in our experiments, with the process of experimenting with various target projects in the future, we plan to identify the components from which the module was cloned in a more sophisticated way (*e.g.*, analysis of call relationships between functions in modules).

Threats to validity. While we employed a monitoring system to minimize human errors in result verification, we acknowledge the possibility of misidentifying TPs and FPs, as well as the risk of overlooking dependencies in the target project (FNs). Also, to the best of our knowledge, no ground truth data are available that exhibit the originating project of indistinguishable files included in our dataset. As it is challenging to determine the component indistinguishable files belong to, there might be a possibility of undiscovered false alarms in our analysis. In addition, we used more than 10 K OSS projects in our experiments (*i.e.*, leveraging the dataset of Centris), but this dataset might not fully represent the diversity and breadth of OSS projects. Lastly, we utilized Centris to show the accuracy of Cneps in dependency analysis; however, the purpose of Centris is to precisely detect uniquely identifiable reused components, not on detecting component dependencies. Our intention is not to discredit the original purpose of Centris but to demonstrate that Cneps performs better in dependency analysis in the presence of indistinguishable files and duplicated components.

Limitations. Although we demonstrated the effectiveness of CNEPS, it contains several limitations that can limit its application.

First, CNEPS can analyze dependencies only when the source code of the target project is provided. Second, although we follow the rules of the GNU GCC compiler, CNEPS may fail or detect an incorrect header path in the process of resolving "`#include`" directives. For example, when multiple header files with the same name exist in the same path distance, CNEPS fails to differentiate the header files, thereby producing false alarms (see Section 4.1). To overcome this problem, we are planning to devise a new resolving technique, such as utilizing metadata [11, 23] (*e.g.*, build scripts) or leveraging control and data flow analysis approaches (*e.g.*, LLVM and SVF [20, 30]). Last, because CNEPS uses CENTRIS in its component analysis, it cannot identify dependencies on components that CENTRIS fails to detect. In the future, if a more precise and practical tool is developed, we plan to use it alongside our own proprietary component identification algorithm.

6 RELATED WORKS

Software composition analysis approaches. Several approaches have attempted to detect reused OSS components using unique characteristics [10, 16, 21, 27, 31, 38, 47]. For example, CENTRIS [38] aims to identify modified OSS components by focusing on the unique functions of OSS projects, combined with code segmentation and redundancy elimination techniques. LIBD [21] utilizes feature hashing to detect reused third-party libraries. Additionally, some approaches attempted to detect reused OSS components from binary software [10, 42, 45]. For example, OSSPOLICE [10] identifies reused third-party libraries by comparing similarities between binaries. However, these approaches fail to address indistinguishable file and duplicated component problems, and thus yield false positives and negatives in dependency analysis (see Section 4.1).

Several approaches focus on analyzing precise dependencies across various package management systems, such as Node Package Manager (NPM) [8, 9, 13, 24, 33, 48]. In recent work, Liu *et al.* [24] proposed an approach to analyze vulnerability propagation in the NPM ecosystem by resolving dependency graphs including transitive sub-components. However, metadata (*e.g.*, package dependency files) is not always available [26, 39] for every component especially when the component is partially reused or modified, which is prevalent in the C/C++ ecosystem [31, 38]. Therefore, they are not suitable for solving our target problem.

Code clone detection approaches. Several approaches have been proposed to identify code clones within source code projects [15, 17, 25, 29]. Additionally, some approaches have employed code clone detection techniques to identify vulnerabilities [14, 19, 22, 35–37]. However, there is a gap between code clone detection and dependency analysis, and an efficient algorithm should be developed to bridge this gap. Therefore, although code clone detection approaches can be used together with CNEPS to provide a more effective vulnerability management process, they are difficult to be used in dependency analysis directly.

7 CONCLUSION

With the growth of OSS reuse, precisely examining dependencies among reused components has become critical in preventing potential threats in software. Accordingly, we present CNEPS, which is a precise approach that analyzes dependencies among reused components by addressing indistinguishable files and duplicated component problems. Our evaluation showed that CNEPS outperformed existing SCA approaches in dependency analysis by achieving 89.9% precision and 93.2% recall. CNEPS can be applied to manage risks by helping understand potential risks associated with dependencies; thus, it can be used to render a safer software ecosystem. The source code of CNEPS, all datasets, and the experimental results will be rendered publicly available at the publication time to foster future research.

8 ACKNOWLEDGMENTS

This work was supported by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains, No.2022-0-01198, Convergence Security Core Talent Training Business(Korea University), and IITP-2024-2020-0-01819, ICT Creative Consilience program).

REFERENCES

[1] 2021. State of the Software Supply Chain. https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021
[2] 2022. The State of Software Bill of Materials (SBOM) and Cybersecurity Readiness. https://www.linuxfoundation.org/research/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readiness
[3] 2023. 2023 State of Open Source Security. https://go.snyk.io/state-of-open-source-security-report-2023
[4] 2023. Ctags: Universal Ctags. https://github.com/universal-ctags/ctags
[5] 2023. GCC: the GNU Compiler Collection. https://gcc.gnu.org/
[6] 2023. Software Bill of Materials (SBOM). https://www.cisa.gov/sbom
[7] 2023. Software Identity: Challenges and Guidance. https://www.ntia.gov/page/software-bill-materials
[8] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the IEEE/ACM 15th international conference on Mining Software Repositories (MSR)*. 181–191.
[9] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 349–359.
[10] 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 (CCS)*. 2169–2185.
[11] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 463–474.
[12] 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 (ESEC/FSE)*. 518–529.
[13] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2021. Dependency Smells in JavaScript Projects. *IEEE Transactions on Software Engineering* 48, 10 (2021), 3790–3807.
[14] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *33rd IEEE Symposium on Security and Privacy (SP)*. 48–62.
[15] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the IEEE/ACM 29th International Conference on Software Engineering (ICSE)*. 96–105.

[16] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-Party Library Dependency for Large-Scale SCA in the C/C++ Ecosystem: How Far Are We?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1383–1395.

[17] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE transactions on Software engineering* 28, 7 (2002), 654–670.

[18] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S Păsăreanu, and David Lo. 2022. Test Mimicry to Assess the Exploitability of Library Vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 276–288.

[19] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *38th IEEE Symposium on Security and Privacy (SP)*. 595–614.

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 75–86.

[21] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: Scalable and Precise Third-party Library Detection in Android Markets. In *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 335–346.

[22] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering* 32, 3 (2006), 176–192.

[23] Nándor Licker and Andrew Rice. 2019. Detecting Incorrect Build Rules. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1234–1244.

[24] 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. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 672–684.

[25] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A Map of Code Duplicates on GitHub. *Proceedings of the ACM on Programming Languages (OOPSLA)* 1 (2017), 1–28.

[26] 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 (SAC)*. 1483–1491.

[27] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. AdDetect: Automated Detection of Android Ad Libraries using Semantic Analysis. In *IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. 1–6.

[28] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1513–1531.

[29] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1157–1168.

[30] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*. 265–266.

[31] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. 2022. Towards Understanding Third-party Library Dependency in C/C++ Ecosystem. In *Proceedings of the IEEE/ACM 37th International Conference on Automated Software Engineering (ASE)*. 1–12.

[32] Wentao Wang, Faryn Dumont, Nan Niu, and Glen Horton. 2020. Detecting Software Security Vulnerabilities Via Requirements Dependency Analysis. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1665–1675.

[33] 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 (ICSE)*. 125–135.

[34] 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 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 319–330.

[35] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. 2023. V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*.

[36] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. 3037–3053.

[37] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. 3041–3058.

[38] 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 *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 860–872.

[39] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. 2015. How do developers use C++ libraries? An empirical study. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 260–265.

[40] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. 2023. An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. (2023).

[41] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 1165–1182.

[42] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: Binary Level Partially Imported Third-Party Library Detection via Program Modularization and Semantic Matching. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 1393–1405.

[43] Awad Younis, Yashwant K Malaiya, and Indrajit Ray. 2016. Assessing Vulnerability Exploitability Risk Using Software Properties. *Software Quality Journal* 24 (2016), 159–202.

[44] Awad A Younis, Yashwant K Malaiya, and Indrajit Ray. 2014. Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability. In *IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE, 1–8.

[45] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. 2019. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1038–1049.

[46] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2021. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Transactions on Software Engineering* (2021).

[47] Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. 2019. LibID: Reliable Identification of Obfuscated Third-Party Android Libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 55–65.

[48] 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 *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. 995–1010.