OSSFP: Precise and Scalable C/C++ Third-Party Library Detection using Fingerprinting Functions

Jiahui Wu*, Zhengzi Xu*§, Wei Tang†, Lyuye Zhang*, Yueming Wu*, Chengyue Liu *,
Kairan Sun*, Lida Zhao*, Yang Liu*
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
jiahui004@e.ntu.edu.sg, zhengzi.xu@ntu.edu.sg, zh0004ye@e.ntu.edu.sg, wuyueming21@gmail.com,
liuchengyuechina@gmail.com, sunk0013@e.ntu.edu.sg, lida001@e.ntu.edu.sg, yangliu@ntu.edu.sg
†School of Software, Tsinghua University, China, tang-w17@mails.tsinghua.edu.cn

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00034

***Abstract*—Third-party libraries (TPLs) are frequently used in software to boost efficiency by avoiding repeated developments. However, the massive using TPLs also brings security threats since TPLs may introduce bugs and vulnerabilities. Therefore, software composition analysis (SCA) tools have been proposed to detect and manage TPL usage. Unfortunately, due to the presence of common and trivial functions in the bloated feature dataset, existing tools fail to precisely and rapidly identify TPLs in C/C++ real-world projects. To this end, we propose OSSFP, a novel SCA framework for effective and efficient TPL detection in large-scale real-world projects via generating unique fingerprints for open source software. By removing common and trivial functions and keeping only the core functions to build the fingerprint index for each TPL project, OSSFP significantly reduces the database size and accelerates the detection process. It also improves TPL detection accuracy since noises are excluded from the fingerprints. We applied OSSFP on a large data set containing 23,427 C/C++ repositories, which included 585,683 versions and 90 billion lines of code. The result showed that it could achieve 90.84% of recall and 90.34% of precision, which outperformed the state-of-the-art tool by 35.31% and 3.71%, respectively. OSSFP took only 0.12 seconds on average to identify all TPLs per project, which was 22 times faster than the other tool. OSSFP has proven to be highly scalable on large-scale datasets.**

I. Introduction

Third-party libraries (TPLs) have been rapidly developed as open source software. DéjàVu [1] has shown that developers tend to clone the source code of different TPLs into their software projects to prevent repetitively developing existing similar functions and to enhance the efficiency of the software development cycle. However, these copied TPLs may also introduce severe security issues, such as Common Vulnerabilities and Exposures (CVE), to the software, which endanger the project development. Therefore, it is critical for developers to track the TPLs in projects so that they can rapidly detect and fix existing related security issues. Software composition analysis (SCA) tools are proposed to detect the TPLs so that developers can understand and manage the security risks.

The main algorithm of SCA is to generate a unique and robust signature for each TPL, and match the signature in the target software to test the TPL presence. As the number of TPLs grows, how to generate the signature to accurately and efficiently detect TPLs in large-scale datasets has become a challenge for SCA tools. Existing tools propose different approaches to address the problem, but none of them can completely solve it. First, OSSPolice [2] improves the matching efficiency via using lightweighted features, such as the directory structures and folder names, as the signatures to detect the TPLs. However, these kinds of features are not robust and can be easily modified by the developers. Therefore, this approach would produce a lot of false negatives if the directory structure or names of TPLs is changed in the target projects. To use more robust features, code clone-based approaches, such as SourcererCC [3], uses token-based index generated from source code as the signatures. It improves the robustness of the signature since developers rarely change all the copied functions in the software. However, not all the functions are unique to the regarding TPLs such as functions copied from other TPLs. Therefore, this approach will yield false alarms in the presence of nested code clones. For example, if TPL$_a$ copies the code of TPL$_b$ into its repositories as nested code clones and the target project also uses some code of TPL$_b$, SourcererCC will report both of TPL$_a$ and TPL$_b$ in its detection results even if only TPL$_b$ is used in the target project.

The state-of-the-art C/C++ TPL detection tool CENTRIS [4] tries to ensure the uniqueness of the generated signature by eliminating function clones in the nested reused TPLs. It removes all the duplicated clones among TPLs in its pre-collected database by keeping only the code with the oldest time tag as the original copy. However, after the cloned function elimination, there are still large number of common and trivial functions which are not representative enough to be a signature to identify a TPL. For example, it is difficult to determine the originality of some simple functions such as a function containing the body of ”return 0”. Both project A and project B may contain these common and trivial functions without cloning them from each other.

Unfortunately, CENTRIS is unaware of the existence of common and trivial functions, thus noise is brought into the features generated from all the functions. To mitigate the problem, CENTRIS compromises to set a pre-defined threshold and only reports a TPL when the ratio of matched function signatures exceeds it, which brings three undesirable consequences. First, utilizing a pre-defined threshold for TPL

[§] Zhengzi Xu is the corresponding author.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00034

detection would still introduce false positives and false negatives. For example, if the threshold is set too high, CENTRIS will fail to report those TPLs of which only a small part of the code is copied in the target project. On the other hand, a low threshold will cause many false alarms for mapping those common and trivial functions within the target projects to the pre-collected TPLs. Second, as CENTRIS has to calculate the matched ratios between the target project and each TPL, the searching time would unnecessarily increase linearly as the growth of total number of TPLs. Third, without removing those common and trivial functions, CENTRIS requires more storage space to save the features.

Failing to handle the nested TPLs as well as common and trivial functions, the accuracy and performance of the existing approaches are negatively affected in detecting C/C++ TPLs. Based on the limitations of the existing approaches, we have summarized three requirements for SCA tools to achieve accurate and rapid TPL detection. **Requirement 1 (R1)**: SCA tools should generate representative signatures for accurate TPL matching and detection. Most of the functions in the TPLs do not contain the core logic. Instead, they are functions copied from the other TPLs or common and trivial functions. Utilizing these functions as fingerprints to detect TPLs would generate numerous false positives when they are falsely mapped to the functions in the target projects. Therefore, it is important to summarize distinguishing signatures to ensure the detection precision. **Requirement 2 (R2)**: SCA tools should avoid using thresholds in predicting the TPL existence. The proportion of the TPL functions being cloned into projects could range from nearly 0% to 100%. When the TPLs are partially used, setting a predefined threshold to detect TPLs would cause a high false negative rate. **Requirement 3 (R3)**: SCA tools should be capable of maintaining good performance on detection with a large-scale TPL database. As more TPLs have been developed, the search space is increased correspondingly. Therefore, it is critical to ensure the scalability in terms of feature size and time efficiency.

To this end, we propose OSSFP, an SCA framework to perform rapid and accurate TPL detection on large-scale projects to address the existing challenges. First, for **R1**, to generate representative features and signatures for TPL detection, OSSFP chooses to distinguish the core functions, which possess the real values of the TPL, in the projects. Specifically, it divides the functions into four categories, clone function, supporting function, common function, and core function. By filtering out the first three types of functions, it can generate unique fingerprints for the TPLs based on only the core functions. Second, for **R2**, OSSFP abandons the predefined threshold and utilizes the fingerprint index built from the core functions to detect TPLs. OSSFP ensures that each TPL will have its unique fingerprints so that if any of the fingerprints are detected in the target project, it has high confidence to report the existence of TPL reuse. Therefore, the partially used TPLs will not be missed. Furthermore, OSSFP can also maintain a high precision rate by reducing the noises in the fingerprints, which are caused by supporting and common functions. Third, for **R3**, OSSFP improves the scalability by using core functions, which account for 1.06% of the total functions, to build the feature index. It reduces the size of the TPL search spaces by 98.94%, and improves its performance in TPL detection by utilizing the feature index.

We applied OSSFP to construct a TPL fingerprinting database with 23,427 real-world C/C++ repositories from GitHub. Furthermore, we manually verified 896 TPLs and built the ground truth testing data for evaluation experiments. The experimental results show that OSSFP can achieve 90.34% for precision and 90.84% for recall, which outperforms CENTRIS by 3.71% and 35.31%, respectively. OSSFP took 0.12 seconds on average to identify all TPLs per project, which was 22 times faster than CENTRIS. With the help of our industry collaborator, OSSFP was integrated as a commercial tool [1] for detecting C/C++ TPLs.

The main contributions of this paper are as follows:

- We proposed an SCA framework OSSFP, which provides rapid and accurate TPL detection for large-scale projects via selecting core functions from each TPL and generating unique fingerprints from these core functions.
- We implemented OSSFP and applied it to construct a large-scale TPL fingerprinting database with 23,427 C/C++ repositories, which include 585,683 versions and 90 billion lines of code.
- We conducted experiments to show that OSSFP outperforms the state-of-the-art SCA tools by achieving 90.34% precision and 90.84% recall on detecting 896 TPLs in 100 software projects. The results also show that OSSFP is highly scalable, which takes 0.12 seconds on average to detect all TPLs in one project.

II. Related Works

Software Composition Analysis (SCA) tools are proposed to report TPLs within the target projects. Code clone detection algorithms are proposed for mapping the target source code to the pre-collected code, which can be used as the basic approach for TPL detection. These two types of related works are discussed in the following subsections.

A. Software Composition Analysis

SCA tools have been developed to handle different scenarios to enhance their accuracy. Commercial tools like OWASP [5] and Sonatype [6] utilize the SBOM (software bill of materials) files to list the third-party libraries. Since there is no unified BOM file exists in the open source C/C++ repositories, these commercial tools can not be applied to source code detection. LibD [7] and ATVHunter [8] and LibScout [9] and LibPecker [10] and OSSPolice [2] targets at detecting TPLs in Android applications and focus on the binary level. Xiao et al. [11]–[19] utilizes the vulnerable code snippets to identify vulnerable TPLs in the target projects. These tools involve only the vulnerable TPLs which consist only a small part of the open source field. There are also

[1] Free trial at https://scantist.io

some commercial tools such as BlackDuck [20] and Snyk CLI that try to identify the TPLs by code clone detection while do not eliminate the common and trivial functions. When ignoring these types of functions, the state-of-the-art C/C++ TPL detection tool CENTRIS also fails to yield a promising result, though it tries to balance its precision and recall by setting a pre-defined threshold.

B. Code Clone Detection

When utilizing signature generated from source code for detecting TPLs, code clone detection algorithms are the most fundamental and relevant approaches. Code clone refers to the duplication of source code, which can be divided into four types [21]–[24]. Code clone detection algorithms [3], [25]–[45] have been rapidly developed to precisely and efficiently detect all clone types of reused source code. When applying these algorithms in detecting clone functions, they can match the similar or duplicated function pairs between the target project and the pre-collected libraries with high accuracy and high efficiency. However, if the mapped functions from the pre-collected libraries are not representative and the source libraries of these mapped functions are directly reported as the reused TPLs of the target projects, it would generate numerous false positive cases in the detection result. The nested TPLs mentioned in the Sec. I is one typical situation that causes high false positives when applying the code clone algorithm for TPL detection. Thus, the characteristic of TPLs usage needs to be considered before applying code clone detection algorithms for TPL detection.

III. Background

A. Terminology

In this section, we define the terms used in the paper to avoid confusion.

Repository. Repositories refer to the storage location of the software packages. This paper focuses on GitHub repositories, of which software packages are stored on the GitHub platform.

Library and Third-party library. Libraries are reusable components that support certain functionalities during code development. Third-party libraries denote libraries that are developed to be shared and used in other software programs. They are usually open source and may be hosted as a GitHub repository.

Partially Used TPL. When only part of the source code of the TPL is utilized in the projects, we refer to the TPL as a partially used TPL. Partially used TPLs are common in open source projects. They are usually introduced when developers only need sub-modules of the TPLs in their projects.

Project. A software project is similar to a software repository. It can be a TPL or a real-world application containing commercial code. This paper refers to the project as the target software, which may contain multiple TPLs.

Version. A version of a library refers to a release tag of the GitHub repository. If the GitHub repository has not released any tags, the current commit will be used as the one and only version for the library.

B. Function Definition

Our key algorithm is to determine the core functions in the TPLs to build the fingerprint. Thus, we have classified functions in the repositories into four categories. In this section, we will give the formal definitions and examples for each of the category.

Clone Function.: Clone functions are the ones that are not created on their own but imported via copying from other libraries. For example, in the project Chromium [46] , all the functions in the file /third_party/libpng/png.c (4dc9e5e)[2] are clone functions. The code in the file is not developed by Chromium community but is imported from a TPL named LibPNG [47].

Supporting Function.: Supporting functions are the ones that do not contain any core logic or algorithm. For example, Listing 1 presents supporting functions retrieved from network cryptography TPL OpenSSL (38b051a)[3]. These functions only contain a return statement, which do not have logic related to cryptography and network communications.

```
long ossl_quic_callback_ctrl(SSL *s, int cmd, void
    ↪ (*fp) (void))
{
   return 0;
}
long ossl_quic_ctx_callback_ctrl(SSL_CTX *ctx, int
    ↪ cmd, void (*fp) (void))
{
   return 0;
}
```

Listing 1: Supporting Function Example from OpenSSL

Common Function.: Common functions are the ones that contain certain logic but are not exclusive to specific libraries. These include hashing functions, sorting functions, and mathematical operation functions, which contain certain algorithms. However, the algorithms are commonly known by developers and can be re-implemented in different TPLs. Listing 2 shows a common function in OpenSSH [48] (57ed647)[4]. It calculates the absolute value of a float number, which is common math logic and is not related to cryptography and network communications. The difference between a common function and a clone function is that instead of using others code, developers re-implement and customize the algorithm in a common function.

```
static LDOUBLE abs_val(LDOUBLE value)
{
   LDOUBLE result = value;
   if (value < 0)
      result = -value;
   return result;
}
```

Listing 2: Common Functions Example from OpenSSH

[2] See commit 4dc9e5e4e4cbae3edde77b55839144149ae6ba0a in the Chromium source tree.

[3] See commit 38b051a1fedc79ebf24a96de2e9a326ad3665baf in the /ssl/quic/quic_impl.c of the OpenSSL source tree.

[4] See commit 2dc328023f60212cd29504fc05d849133ae47355 in the /openbsd-compat/bsd-snprintf.c of the OpenSSH source tree.

Core Function: Core functions are the ones that implement the core logic and algorithms of the library. These functions are unique which contain the intellectual property of the TPLs. Thus, we use the core functions to generate the fingerprint for the TPLs.

C. Motivating Example

In an SCA task, we aim to identify a TPL named Stringencoders [49], which is partially copied in the target project Chromium [46]. In Chromium [46], the source code file 'modp_b64.cc' (196797a)[5] from Stringencoders is copied into the folder '/third_party/modp_b64/' (6db05bf)[6]. The file contains three functions, which consist the core algorithm of Stringencoders to process the strings. However, the rest parts of Stringencoders, which contain 251 functions, are not imported.

In this situation, Stringencoders can not be detected by OSSPolice since OSSPolice use the directory information for TPL detection, and the directory structure is changed in the target project. Moreover, since the copy ratio of Stringencoders is 1.19% (3/251) and has not exceeded the predefined threshold (10%) set by CENTRIS, it will miss this case. This problem can not be resolved by directly lowering the threshold since CENTRIS has shown that if the predefined threshold is set close to 0%, the false positive rate will be increased significantly. To detect the TPL, we propose OSSFP to select only the core functions to construct the TPL fingerprint. In this case, the three imported functions are all core functions. If any of the three functions are matched in Chromium, our tool will report the detection of Stringencoders. Furthermore, since the three functions in Chromium are unique, OSSFP will not produce false positives by mistakenly matching them with fingerprints in other TPLs.

IV. Methodology

A. Overview

Figure 1 presents the overview of OSSFP, which consists of three offline phases to build a comprehensive TPL database for C/C++ SCA task. First, in the feature generation phase, we collected the TPL information via cloning all the target GitHub repositories into the local system. We utilized the git tag as the version for each repository and then generated features for each function of each version. Second, in the hash building phase, we built the library function hash index for each library by removing duplicate functions between versions and then built the distinct function hash index by removing duplicate functions between libraries. Third, in the fingerprint selection phase, we used different function properties to filter out clone functions, supporting functions, and common functions. Last, we kept only core functions for each library, and these functions would be used as the fingerprint for TPL detection.

B. Feature Generation

Feature of the library refers to the set of the code clone feature and other code properties of all the functions. In the feature generation steps, we aim to collect the meta information of the libraries from the open source repositories and generate the feature for library. Since GitHub is the most popular platform [50] for hosting open source repositories and C/C++ GitHub repositories over 100 stargazers are often used in previous related works [2], [4], [51], we chose GitHub as the repositories source for collecting libraries. In total, we have collected a set of 23,427 links of GitHub repositories via the official API provided by GitHub. Note that if the repositories are forked from other original repositories, they would not be included in our data set. We pull all source code of the repositories into the local system by the git clone command. The cloned repositories contain all the history messages of the source code, including commit time and version tags. Once the repositories are prepared, the next step is to obtain the version information. Most GitHub repositories release their version by tagging specific points in their commit history. For each repository, we retrieve these tags as their versions. For those repositories which have not released any git tags, we used its current main branch as their only version. we chose Antlr [52] to parse functions within all C source code files. Antlr is an efficient lexical parser that is compatible with most programming languages.

To avoid the repetition of parsing the duplicate source code files between different versions of the same library, we adopted an incremental approach to enhance the efficiency of generating function features for repositories. Some repositories like OpenSSL [53] contain a large number of git tags that may lead to a long processing time. Between two consecutive GitHub release tags, there may not be too much difference in the source code. Thus, we optimized the processing time by only parsing the differences in source code between consecutive versions. The information on the difference between two git tag versions can be quickly retrieved by git log command.

After setting up the meta information and tools, we proceeded to generate the following features for each function of each version of each library.

- **Function Hash**: the MD5 hash value generated from the function content. To avoid missing mapping of the function pairs without semantic differences, the function content will be normalized by removing blank spaces, new line characters, and comments for calculating the MD5 hash value. Function hash is the identity we used to group the functions in the index building phase.
- **Author Time**: the create time of the function content, which is retrieved by the git blame command. The git blame command could show the actual creation time of each line of the function content. The author time of the function is the decisive factor for determining if the regarded library is the original author of the function.
- **Lines of Code**: the number of lines of the function code, excluding those blank lines. The lines of code is one of the

[5] See commit 196797afb2de11ae5381dad7246e25c7eeae2c1a in the stringencoders source tree.

[6] See commit 6db05bf99752c68e667500675c10399e76f04cc8 in the Chromium source tree.

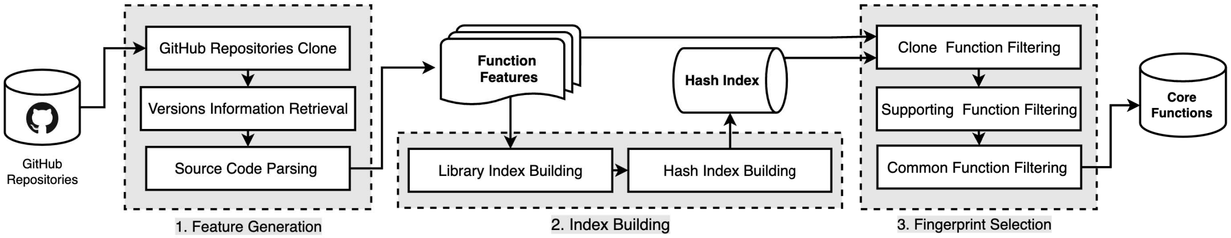


Fig. 1: Overview of OSSFP

measurements for function complexity, which will be used at the Fingerprint Selection Phase.

- **Cyclomatic Complexity** [54]: the number of linearly independent paths within the function content. The Cyclomatic complexity is also used for measuring function complexity.
- **Halstead Volume** [55]: a measuring factor for function complexity which is calculated by the length of the function content and the number of the vocabulary of the function content.

C. Index Building

After generating the feature of each function of all versions of each library, the next phase was to build the distinct hash index for all the functions with their regarded original author library. With the distinct hash index, the clone functions in each repository can be marked out by checking with the hash index. A function can be determined as a non-cloned function or called an original function only if its author time is earlier than those with the same function hash in all other repositories. Vice versa, if the author time of a function is later than the other function in a different repository with the same hash index, it can be recognized as a clone function. The index building phase aims to mark out the author library of each function and thus provide the information for each library to filter the clone functions in the next phase.

To efficiently build the hash index, we divided the process into two steps. The first step called Library Index Building is to collect all the function hashes and their author time for each library, which can be run parallel to reduce the processing time. In this step, we built the independent hash indexes for each library which contain function hash value and their earliest commit time. For different versions of a repository, the author times of the same function hash of different versions in the same repositories may vary due to the non-semantic change in different versions like changing function name or folder name. To select the earliest author time for each function hash in a library, we employed the Algorithm 1 to generate the function hash index of author time.

After building the index of the function hash for each library, the following step called Hash Index Building is to build the function hash index for the whole data set. Only non-clone functions will be stored in the function hash index, which contains the following information.

- **Original Library**: the library with the earliest author time of the same function hash.

Algorithm 1 Library Index Building

```
Input: LFL                                   ▷ List of Library Feature
Output: LHIL                                 ▷ List of Library Hash Index
procedure LIBRARYINDEXGENERATION(LFL)
    $LHIL \leftarrow \emptyset$
    for VFL in LFL do                        ▷ List of Version Feature
        $LHI \leftarrow \emptyset$           ▷ Function Hash Index
        for FL in VFL do                     ▷ List of Function Feature
            $hv \leftarrow FL.hash\_value$
            $ct \leftarrow FL.commit\_time$
            if $hv$ not in $LHI$ then
                $LHI.put$(hv:ct)
            else if $ct < LHI.hv.commit\_time$ then
                $LHI.hv.commit\_time \leftarrow ct$
            end if
        end for
        $LHIL.add(LHI)$
    end for
    return $LHIL$
end procedure
```

- **Author Time**: the creation time of the function content, which is retrieved by the git blame command.
- **Document Frequency**: the number of libraries containing the same function hash.

The function hash index of each library built from the former step will be used as inputs for building the final function hash index, as shown in the Algorithm 2. When iterating each library, we recorded the earliest author time and the corresponding library for each function hash. The occurrence frequency of each function hash will also be recorded as the document frequency for further analysis.

D. Fingerprint Selection

In the last step, we aimed to select the core functions for each library as its fingerprints. As shown in Figure 2, to select the core functions, three steps would be adopted to filter the clone functions, supporting functions, and common functions sequentially. The first step is to eliminate the clone functions. The clone functions in each library would be marked out by checking them with the hash index built in the last phase. The next step targets the supporting functions, which can be filtered by utilizing the function complexity. We chose the maintainability index [56] for measuring the function complexity of each function, which is calculated by the Equation 1. The last step is to distinguish the common functions from the core functions. We selected document frequency to denote the uniqueness of the functions and filter the common functions

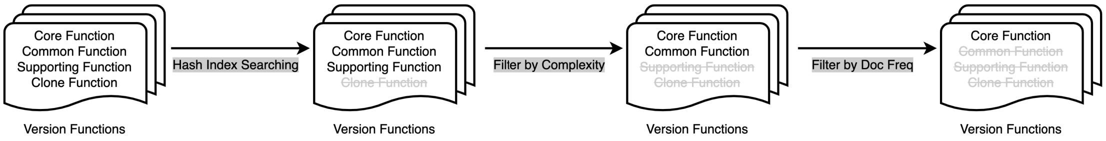


Fig. 2: Fingerprint Selection Algorithm

with it. The algorithm is applied for each version of all repositories.

Algorithm 2 Hash Index Building

```
Input: LHIL                                   ▷ List of Library Hash Index
Output: HI                                    ▷ Final Hash Index
procedure INDEXGENERATION(LHIL)
    $HI \leftarrow \emptyset$                 ▷ Final Hash Index
    for LHI in LHIL do                        ▷ Library Hash Index
        $library_id \leftarrow LHI.name$      ▷ Library Id
        for hv in VFL do                      ▷ Hash Value
            $ct \leftarrow VFL.hv.commit\_time$
            if $hv$ not in $HI$ then
                $HI.hv.commit\_time \leftarrow ct$
                $HI.hv.doc\_freq \leftarrow 1$
                $HI.hv.library \leftarrow library\_id$
            else if $ct < HI.hv.commit\_time$ then
                $HI.hv.commit\_time \leftarrow ct$
                $HI.hv.doc\_freq \leftarrow HI.hv.doc\_freq + 1$
                $HI.hv.library \leftarrow library\_id$
            else if $ct >= HI.hv.commit\_time$ then
                $HI.hv.doc\_freq \leftarrow HI.hv.doc\_freq + 1$
            end if
        end for
    end for
    return $HI$
end procedure
```

Algorithm 3 Clone Function Filtering

```
Input: $LHIL$                                 ▷ List of Library Hash Index
Input: $VFL$                                  ▷ List of Functions of Version
Output: $VFL$                                 ▷ List of Functions of Version
procedure CLONEFILTERING($LHIL$, $VFL$)
    for FF in VFL do                          ▷ Function Feature
        $HL \leftarrow LHIL.hv.library$
        if $HL$ != $FF.library$ then
            $VFL.remove$(FF)
        end if
    end for
    return $VFL$
end procedure
```

Clone Function Filtering. Within the hash index built from the last phase, each function hash is mapped to one library, which is marked as the original author of the function. For each version, if the library of the function hash is not the same as the one in the hash index, it would be marked as a clone function. To check if a function is clone function, the regarding function hash will be searched in the hash index to compare the library information. Each version of all libraries can be run parallel to search the hash index. The algorithm for identifying the clone function can be summarized as Algorithm 3.

The hash index records the library with the earliest author time for each function hash, but it does not mean that all the regarded libraries are the actual original libraries for creating the regarded functions. For example, over 60% of the libraries contain supporting functions like the one shown Listing 1. These widely existing functions are not created by copying from other libraries, and the earliest author library of this type of function can not be determined by the regarded author time. However, whether the libraries with this type of function are marked as the author libraries or not, these functions will eventually be filtered either in this step or the next step for all the libraries. At the end of this step, all the clone functions would be filtered for each library, and only the functions created by themselves remain.

CENTRIS denotes the clone functions as borrowed code and tries to eliminate these functions for each library. To identify the clone function for each library, CENTRIS first calculates the proportion of the overlapped functions between two libraries and then utilizes a predefined threshold to determine if the overlapped functions are clone functions. When the proportion of the overlapped functions is smaller than the predefined threshold, these functions will remain in the non-original library. When the functions from the partially used TPLs remain as the fingerprints of the library, the proportion for calculating the overlapped functions between the regarded library and the target projects would be lessened. If the lessened proportion is under the predefined threshold of CENTRIS, false negatives would be produced in the TPL detection. Thus, using a predefined threshold for determining the function types, the clone functions from the partially used TPLs in each library would be falsely ignored.

Supporting Function Filtering. Since supporting functions do not contain any complex logic, which is different from the common function and core functions, the second step targets filtering supporting functions by utilizing this characteristic. Heuristically, if functions do not contain any complex logic, their function complexity would be lower than the common functions and core functions. Furthermore, maintainability index [56] (**MI**) is a system of measurement for function complexity, which is calculated by Equation 1.

$$\mathbf{MI} = 171 - 5.2 * ln(\mathbf{HV}) - 0.23 * (\mathbf{CC}) - 16.2 * ln(\mathbf{LOC}) \tag{1}$$

HV stands for Halstead Volume, **CC** stands for Cyclomatic Complexity, **LOC** stands for Lines of Code. In the feature generation steps, since the function parser has retrieved the function content, the number of lines of code can be directly calculated by counting the newline characters, while those empty lines would be excluded. Cyclomatic complexity is developed to measure the stability and level of confidence of the program. Thus, with lower Cyclomatic complexity, a function potentially has less complex logic. The Cyclomatic complexity can be calculated for each function by Equation 2.

$$CC = P + 1 \tag{2}$$

P denotes the number of condition nodes in the control flow graph generated by the Antlr parser. The number of the condition nodes can be retrieved in the function parsing step since Antlr provides the types of nodes for the functions. For calculating the Halstead Volume, we adopted the following equation.

$$HV = (N_1 + N_2) * log_2(n_1 + n_2) \tag{3}$$

N_1 and N_2 denote the total number of operators and the total number of operands, respectively. Operand in a function refers to the lexical token that is variable or number or constant string, while other types of lexical tokens are taken as operators. And n_1 and n_2 stand for the the number of distinct operators and and the number of distinct operands, respectively. To avoid the repeat operation of the function parsing, all these three types of function properties would be obtained in the feature generation phase. As shown in the equation of calculating the maintainability index, functions with more straightforward logic would have a higher score on the maintainability index. Thus, after calculating the maintainability index of each function of a library version, the supporting functions will be ranked as the highest part in the increasing order of the scores of the maintainability index. The proportion of the supporting functions may vary in different open source projects. A threshold θ_1 will be set to filter out these parts of supporting functions. The effect of the threshold θ_1 will be evaluated in the Section V.

Common Function Filtering. The last step is to filter out the common functions which are widely used in the repositories. Common functions with lower complexity may be filtered in the last step, while some common functions written with high complexity could not be filtered, such as functions implementing mathematical algorithms or known standard network protocols in different application fields. Heuristically, if the core functions of a library are used, and these functions depend on the common functions, the common functions would be used as well. Besides, since common functions are widely implemented and used within multiple TPLs which contain different core functions, the frequency of the existence of these common functions will be higher than any core functions within one particular library.

Thus, the core functions appeared less to the library than the common functions. To measure the uniqueness of each function, inspired by the term frequency–inverse document frequency (TF/iDF) in information retrieval, we use a similar approach to distinguish the common functions. We define the term frequency and document frequency in this work as the following:

- **Term Frequency**: the number of functions with the same function hash in a library.
- **Document Frequency**: the number of libraries containing the same function hash.

Since the uniqueness of the functions is measured in terms of the library level, the term frequency will be set to 1 to ignore the effect of measuring uniqueness inside one library. Thus, only the document frequency would be used for measuring the uniqueness of the functions. The document frequency of each function hash is already contained in the index building phase, which can be retrieved by searching the function hash. Accordingly, the document frequency of the common functions will be higher than the ones of the core functions of the same library. If the remaining functions are sorted in the increasing order of document frequency, the common functions would appear in the latter part of the function list. Since the number of common functions in each library may vary, a threshold θ_2 will be set to filter out the common functions by the rankings of the document frequency in the libraries. We evaluated the different combinations of the θ_1 and θ_2 in the Section V.

Finally, by filtering out the clone functions, supporting functions, and common functions for each library, we ensure that only the core functions of each library remain in the database.

E. TPL Detection

When any function from the target project is mapped to the function of the regarded library in the hash index built from the core functions, the library would be reported as a TPL of the project. Unlike CENTRIS, we do not adopt any predefined threshold for confirming the TPL since all the functions in our database are unique to the related library and any mapped function from the database is a prove of using the core logic of the related library.

V. Evaluation

A. Research Questions

To evaluate if OSSFP satisfies the three requirements stated in Section I, we conducted experiments to address three research questions. Research Question 1 (**RQ1**) focuses on evaluating the accuracy of OSSFP and comparing it with the state-of-the-art CENTRIS. The experiment will demonstrate if OSSFP can meet the **R1** in Sec I (generating representative signatures for accurate TPL detection) by evaluating its accuracy. **RQ1** will also assess if OSSFP satisfies **R2** in Sec I (abandoning pre-defined thresholds) by comparing its performance with CENTRIS, which utilizes a pre-defined threshold. Research Question 2 (**RQ2**) evaluates the scalability of OSSFP to determine if it meets **R3** in Sec I. Finally, Research Question 3 (**RQ3**) tests the representativeness of features generated from core functions and assesses if filtering other types of functions improves accuracy, in accordance with **R1** in Sec I. The research questions are listed as follows:

- **RQ1**: What is the accuracy of OSSFP in detecting TPLs compared to related works?
- **RQ2**: How is the scalability of OSSFP in terms of time efficiency and data size?

TABLE I: TPLs Distribution of The 100 Target Projects

	Total	Min	Max	Mean	Medium
TPLs	896	1	34	9	7

- **RQ3**: How does each function filtering step contribute to the accuracy improvement of OSSFP?

B. Experiment Setup

Ground Truth Data Set Construction. Since C/C++ GitHub repositories do not adopt any BOM (bill of material) file which lists all its TPLs, we need to build a proper procedure to construct a ground truth set. We manually examined over 1000 projects and found that nearly 20% of them tend to organize and put the source code of all the TPLs into folders with special names, such as ”/3rdparty”, ”/deps” and ”/third_party”. Therefore, we took further steps to build our ground truth data set on the projects which contain these folders. Specifically, we developed scripts to go through each sub-folder inside them and applied the following rules and procedures to establish the TPL ground truth.

Sub-folder Name Mapping. Some sub-folder names contain the full library names. We directly map the names together to identify the TPLs. For example, the sub-folder ’/3rdparty/libz’ is mapped to Zlib [57] library.

Non-source Code Folder Exclusion. If the sub-folder contains only header files or files written in other programming languages, we will exclude it from the ground truth cases.

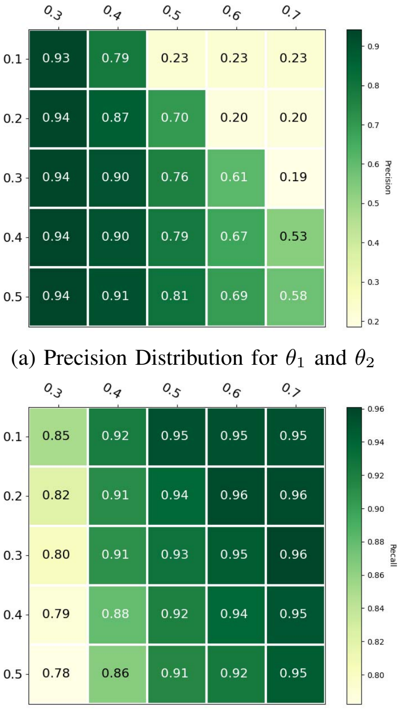
Copyright Confirmation. If the copyright information is stored in the source code files, which usually include vendor name, author name, and library name, we will use the information to identify and confirm the potential TPLs.

README File Mapping. If the README file exists in the sub-folder, which contains information of the TPL usage, we would also use it for mappings with potential TPLs.

Based on the rules, we have examined over 400 C/C++ projects and selected 100 projects, each containing 1 to 34 TPLs. To ensure the correctness of the ground truth, we employed three advanced software researchers to confirm 896 TPL use cases from the 100 projects. The number of TPLs of the target projects is shown in the Table I.

Threshold Selection. After collecting the ground truth data set, we then selected a combination of θ_1 and θ_2 for selecting the desired core functions, which are the thresholds for filtering supporting functions and common functions, respectively mentioned in the Section IV. We randomly selected 80% projects from the ground truth data set for the experiment of the parameter selection. To select the most suitable combination thresholds, we conducted experiments on a group of combination with θ_1 ranging in [0.3, 0.4, 0.5, 0.6, 0.7] and θ_2 ranging in [0.1, 0.2, 0.3, 0.4, 0.5]. As shown in the Figure 3 , the combinations of θ_1 and θ_2 distributed differently in the perspectives of precision and recall.

Figure 3 show that the combinations θ_1 and θ_2 have counter effects in precision and recall. To select a combination of θ_1 and θ_2 for balancing precision and recall, we adopted F_1 [58] which is harmonic mean of the precision and recall.



(a) Precision Distribution for θ_1 and θ_2

(b) Recall Distribution for θ_1 and θ_2

Fig. 3: Accuracy Distribution for θ_1 and θ_2

After calculating the F_1 score for each combination, we select the combination of 0.5 for θ_1 and 0.2 for θ_2 which has the highest F_1 score of 0.91.

Comparison to State-of-the-Arts. To evaluate the accuracy of OSSFP, we selected the CENTRIS and Snyk CLI [59] for comparison. CENTRIS is the state-of-the-art approach for analyzing C/C++ TPLs. While there are many other SCA tools, they do not aim at C/C++ source code, and some are not open-source. For example, ATVHunter [8] and OSSPolice [2] target Android applications and are difficult to apply to the source code of C/C++. Snyk CLI is a free commercial tool that can analyze many programming languages, including C/C++.

Experiment Environment. All our experiments were conducted on Ubuntu 18.04.6 LTS with 2.50GHz Intel(R) Xeon(R) Gold 6248 CPU and 188G RAM. The compared tools used their default settings as stated in their paper or website.

C. RQ1: Accuracy Evaluation

In this section, we compared OSSFP with CENTRIS and Snyk CLI on the ground truth data set. We also discussed the comparison result and analyze the reason for the false positive cases and false negative cases. Table II shows that OSSFP has achieved 90.34% for precision and 90.84% for recall. Table III shows that OSSFP has both higher precision and recall than CENTRIS. Since the source code and data scope of Snyk CLI is not published, we manually checked the accuracy and analyzed the mapping details provided by it.

Comparison with CENTRIS. Since CENTRIS has published its feature set and to ensure the objective of the algo-

TABLE II: Comparison with Different Data Scope

Tool	GT	TP	FP	FN	Precision	Recall
OSSFP (L)	896	814	87	82	90.34%	90.84%
OSSFP (S)	657	607	291	50	67.59%	92.38%

1) L: test with original data scope. S: test with the same data scope with CENTRIS. GT: number of ground truth cases. TP: true positve cases. FP: false positive cases. FN: false negative cases.

TABLE III: Comparison with CENTRIS

SCA Tool	GT	TP	FP	FN	Precision	Recall
CENTRIS	657	375	212	282	63.88%	57.07%
OSSFP	657	607	291	50	67.59%	92.38%

1) GT: number of ground truth cases. TP: true positve cases. FP: false positive cases. FN: false negative cases.

rithm, we conducted the experiments with the same data scope as CENTRIS for comparison. For the ground truth data set, we also excluded those not in the data scope. We ran the detection logic for the 657 cases within the 100 projects with partially used TPLs. Table III shows the accuracy of our OSSFP compared to CENTRIS.OSSFP has both higher precision and significantly higher recall for the 657 test cases. In this data scope, both OSSFP and CENTRIS do not achieve high precision due to the incomplete data set. However, OSSFP can still achieve much higher recall than CENTRIS because OSSFP has abandoned the threshold for confirming the TPL and thus would not miss most of the partially used cases. At the same time, abandoning the pre-defined threshold would not significantly lowering our precision since we ensure our precision by filtering out the common functions and supporting functions for each library and ensure our recall by keeping core functions for TPL detection and abandoning the thresholds.

As shown in Table II, by adding more repositories to our data set, the false positive cases had considerably fallen, and the accuracy had significantly increased. The main reason for causing the false positives cases is the lack of complete data set of C/C++ libraries, as we analyzed later in this section. The high recall of the OSSFP shows that OSSFP are more capable of detecting partially used cases while maintaining higher precision than CENTRIS.

Comparison with Snyk CLI. Since the commercial tool Snyk CLI does not publish its data set and source code, we used our ground truth and the mapping result generated by Snyk CLI to check the accuracy manually. Snyk CLI divides the TPL detection into two steps. It first generates the hash values for each file under the target projects. It then sends all the feature information to their servers for TPLs detection. When the online processing is done, the local client would retrieve the result from the server and show the formatted result followed by the input command. After confirming the total of 896 cases from the result file generated by Snyk CLI , the precision and recall of Snyk CLI are 61.98% and 28.63%, respectively. From the mapping details provided by Snyk CLI, there are still nested TPLs falsely reported as other TPLs which contain them.

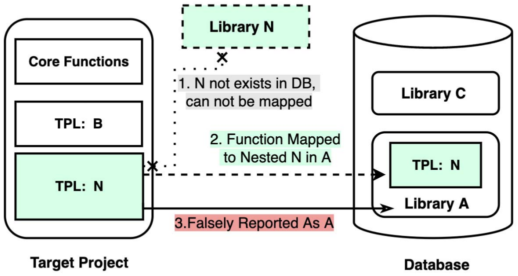


Fig. 4: False Positive Case (a)

False Positive Analysis. Although we tried to filter out all the non-core functions and make sure the selected functions were unique to the regarded library, some false positive cases still existed for the following reasons. The first reason is the lack of libraries in our data scope, which concludes 53 false positive cases. For example, as shown in Figure 4, if library N were not in our data scope, the TPL N in A would be regarded as part of the core logic library A in the feature generation phase. In this situation, if we performed a TPLs detection for the target project, which contains the source code of library N, we would falsely report library A as a TPL since the code of library N had been treated as part of library A. This is also the main reason why both OSSFP and CENTRIS would suffer from low precision as evaluated in the previous experiment. The second reason is the poor version management of the libraries, which concludes 34 false positive cases in our experiments. For example, if library B does not release any git tag as its versions, only the code snapshot of the latest commit would be used as the only version for feature generation. As shown in Figure 5, we denote B2 as the latest code snapshot of library B we collected in our database and B1 as an earlier version of B2, which is utilized as a TPL of library A. If the source code of B2 has non-semantic changes compared to B1, the commit time of the functions of B2 would be later than the commit time of the functions of B1. In this situation, the source of code of B1 in library A would not be considered as clone functions and consequently be considered as part of the core logic of library A. Thus, when scanning the target project containing library B, OSSFP would falsely report library A as its TPL since library A shares the same code with library B in the target project.

The data scope of 23,427 GitHub Repositories is not a complete set of C/C++ libraries in the open-source field. Some C/C++ open source projects are maintained in other source code hosting platforms like Gitlab, SourceForge, and their homepages. If we can collect all the related libraries in our data sets, these problems could be automatically fixed. However, to the best of our knowledge, there is no existing research aiming at collecting the complete library and version list for the field of C/C++ open source projects. We would like to leave it as a future research to build the open source project roadmaps for C/C++.

False Negative Analysis. All the false negative cases come from poor version management of the libraries. Similar to the second reason for the false positive cases, which is shown in Figure 5, library B would be a false negative case if the target project utilized an earlier snapshot of the library. For example, GoogleTest, which is one of the most popular

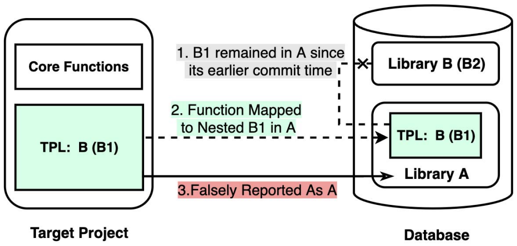


Fig. 5: False Positive Case (b)

TABLE IV: FEATURE SIZE OF EACH FILTERING STEP

Steps	Total Functions
None	1,366,104,539
Clone Function Filtered	43,573,125
Supporting Function Filtered	26,504,053
Common Function Filtered	14,589,744

open source repositories, does not release any tag between 19 September 2013 and 22 August 2016. In this period, there were many continuous updates of the source code, whereas some repositories like FALCONN continued to copy the latest code of GoogleTest. When GoogleTest finally released the version on 22 August 2016, the commit time of the functions from this version might be later than the ones in FALCONN due to the non-semantic change in this period. On the other hand, if B1 is falsely recognized as part of library A, some of the core functions of library A would be filtered if the functions of B1 take the priority place in the Fingerprint Selection algorithm. In this case, if a sub-module of A is used as in the target project and the core functions of this sub-module were filtered, OSSFP cannot correctly report this case.

Answering RQ1: Without utilizing the threshold for TPL detection, OSSFP was evaluated to have a high precision of 90.34% and a high recall of 90.84% at the ground truth data set. Moreover, OSSFP outperformed CENTRIS in precision and recall by 3.71% and 35.31%, respectively, while Snyk CLI showed only 61.98% for precision and 28.63% for recall. Thus, the experiments conducted in RQ1 shows that OSSFP meets the **R1 and R2** in Sec I.

D. RQ2: Scalability Evaluation

To evaluate the scalability of the OSSFP, we measured the feature size and the time of prepossessing and TPL detection compared to CENTRIS.

Feature Size. In our Fingerprint Selection phase, we have filtered out all the clone, supporting, and common functions. There is a total of 1,366,104,539 functions of the total 585,683 versions of 23,427 libraries we collected, while we only need to keep 14,589,744 core functions as the fingerprint of each library for TPL detection. As shown in the Table IV, OSSFP only uses only 1.06% of the total functions, while CENTRIS keeps 2.2% of the total functions. By keeping only half of the total functions of CENTRIS, we also reached higher precision and recall than CENTRIS. The largely reduced feature size proves that OSSFP meets the R3 in Sec I.

TABLE V: TPL DETECTION TIME COMPARISON

Type	OSSFP	CENTRIS	Snyk CLI
Total	12.51	267.14	N.A.
Average	0.12	2.67	N.A.

1) all the time values are measured in seconds.

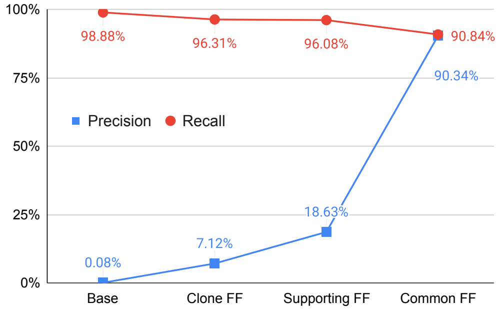


Fig. 6: Improving Accuracy With Accumulative Filtering Step Applied

Time Efficiency. In the TPL detection step, CENTRIS requires extra steps for calculating the overlapping rate for each library, and the CENTRIS utilizes more functions as fingerprints for each library than OSSFP. In terms of the average time of TPLs detection of each target project, OSSFP has a much smaller number than CENTRIS, as shown in Table V. When counting the TPL detection times, we excluded the time cost by feature generation and database loading. Since the source code of Snyk CLI is not publicly available and the time cost by TPL detection can not be independently statistics, the TPL detection time of Snyk CLI will not be put in the table. In the feature generation steps, it took about 177 hours to generate the necessary features for 23,427 libraries, which cost less than 30 seconds for each library. For the index building and Fingerprint Selection phase, OSSFP only took 16.5 hours to perform the algorithm. These processing times imply that OSSFP is highly scalable to be implemented in large-scale data sets, which satisfied the R3 in the Sec I.

Answering RQ2: By selecting the core functions for each library, we significantly reduced the feature size to 1.06% compared to the feature size generated in the first phase. OSSFP took only 0.12 seconds on average to identify all TPLs per project, which was 22 times faster than CENTRIS. Additionally, OSSFP only took less than 30 seconds to process each library, showing that OSSFP is efficient enough to apply to large-scale data sets. Thus, OSSFP conforms the **R3** in Sec I in terms of time and feature size.

E. RQ3: Ablation Study

To evaluate the representativeness of the signature generated from core functions, we also conducted an ablation experiment to compare the improving accuracy of utilizing only core functions with those comprising non-core functions. Since the clone functions, supporting functions, and common functions are filtered sequentially in the Fingerprint Selection phase, we evaluated the accumulative effect of applying each filtering step. Figure 6 shows the precision and recall of applying each algorithm of steps, where FF stands for function filtered. With each filtering step applied, the precision has significantly increased, which proves that our algorithm has successfully filtered the non-core functions for each step. And features generated from the core functions are unique and representative enough for TPL detection, which shows that OSSFP conforms the R1 in Sec I. Although the recall has slightly decreased at filtering steps, it does not means that we filtered the core functions as we analyzed the false negative cases in the RQ1. Without removing any non-core functions, the precision is close to 0 because of the nested TPLs and other trivial functions. Even with eliminating the clone functions for each library, the precision is still at a low level because of the existence of supporting functions and common functions. When filtering the supporting functions, the precision is still not sufficiently high. There is a large number of common functions which cause false positives. After filtering all the non-core functions, we can achieve high precision and high recall at the same time.

Answering RQ3: The ablation study shows that the different types of non-core functions have a significantly negative effect on accuracy. After filtering the clone functions, supporting functions, and common functions, OSSFP can precisely detect the TPLs by utilizing only the core functions. The ablation experiment proves that features generated from the selected core functions are highly representative for TPL detection, which fulfills **R1** in Sec I.

F. Discussion

1) Abandon Threshold: By selecting the core functions for each library for TPLs detection, OSSFP has successfully abandoned the predefined threshold for SCA and thus achieved high precision and high recall for detecting those partially used TPLs. This algorithm is essential for the open source field of C/C++ since the lack of a standard package manager for managing the dependency of source code and the existence of the nested TPL in the open source repositories. The state-of-the-art approach CENTRIS tried to utilize a predefined threshold to eliminate the nested TPLs and detect the TPLs for the target projects. However, the predefined threshold could not balance the precision and recall for those large projects with many partially used TPLs. Our experiment result shows that even in partially used cases, OSSFP can also achieve high precision and recall. These results implied that OSSFP has precisely selected the core functions for each library and thus enhances the accuracy for detecting the partially used cases.

2) Generalizability: For the popular programming language like Java, C/C++, Python, and JavaScript, DéjàVu [1] has shown that 70% of the source code on GitHub are clones from other original repositories. When utilizing TPLs, source code copying is common in many programming languages, even with an official package manager managing the dependency. These findings show that simply applying the code clone technique for TPL detection would generate numerous false positive cases. Thus, filtering the non-core functions is essential for SCA with the code clone technique.

VI. Limitation

OSSFP has some limitations that affect its application. First, OSSFP relies on complete data set of open source repositories. The lack of a complete data set of open source repositories and poor version management are the most significant limitation of our algorithm. As explained in the previous section, the lack of open source repositories and poor version management would affect the algorithm and thus generate false positive and false negative cases. If a shared library was missed in the data set and it was used as a nested TPL of another library, it would be falsely treated as part of another library and further bring the noise to the followed algorithm. To our best knowledge, there is no automatic way to collect all the open source repositories on the whole internet, and thus it would be a research opportunity to find the complete set of C/C++ open source libraries.

VII. Threats to validity

We manually checked the ground truth for the target projects to measure the accuracy of the tools selected for comparison. Because there is no formal BOM (bill of material) file in the target projects and we do not have a complete list of open source libraries, there may be some hidden ground truth cases we have not listed in our ground truth. The problem of the incomplete data set would affect the accuracy measurement for both OSSFP and CENTRIS.

VIII. Conclusion

In this paper, we propose a novel tool named OSSFP to analyze the TPLs with high precision and high recall. By utilizing the function properties, OSSFP has successfully selected the core functions as fingerprints for each library and thus enhances the accuracy and efficiency. With these core functions, OSSFP can outperform the state-of-the-art approaches for detecting the TPLs.

Data Availability

The data sets of the studies and evaluations can be publicly accessed at *https://sites.google.com/view/icse2023sca*.

Acknowledgements

This research is partially supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-019).

References

[1] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[2] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "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*, 2017, pp. 2169–2185.

[3] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[4] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "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, 2021, pp. 860–872.

[5] "Owasp," https://owasp.org/www-project-dependency-track, 2022.

[6] "Sonatype," https://jeremylong.github.io/DependencyCheck/data/ossindex.html, 2022.

[7] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 335–346.

[8] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.

[9] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 356–367.

[10] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 141–152.

[11] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "Mvp: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[13] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 55–64.

[14] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, "Scalable and systematic detection of buggy inconsistencies in source code," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010, pp. 175–190.

[15] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 447–456.

[16] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 533–544.

[17] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 462–472.

[18] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 48–62.

[19] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[20] "Blackduck," https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html, 2022.

[21] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[22] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[23] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[24] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE access*, vol. 7, pp. 86 121–86 144, 2019.

[25] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–9.

[26] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 387–391.

[27] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 96–105.

[28] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[29] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 253–262.

[30] H. Sajnani, V. Saini, and C. Lopes, "A parallel and efficient approach to large scale clone detection," *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 402–429, 2015.

[31] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[32] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.

[33] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 175–186.

[34] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 219–220.

[35] N. Göde and R. Koschke, "Incremental clone detection," in *2009 13th European conference on software maintenance and reengineering*. IEEE, 2009, pp. 219–228.

[36] R. Koschke, "Large-scale inter-system clone detection using suffix trees and hashing," *Journal of Software: Evolution and Process*, vol. 26, no. 8, pp. 747–769, 2014.

[37] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1066–1077.

[38] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 321–330.

[39] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 81–92.

[40] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 301–310.

[41] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International static analysis symposium*. Springer, 2001, pp. 40–56.

[42] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *2016 15th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2016, pp. 1024–1028.

[43] H. Wei and M. Li, "Positive and unlabeled learning for detecting software functional clones with adversarial training." in *IJCAI*, 2018, pp. 2840–2846.

[44] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 70–80.

[45] T. Nakagawa, Y. Higo, and S. Kusumoto, "Nil: large-scale detection of large-variance clones," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 830–841.

[46] "Chromium," https://github.com/chromium/chromium, 2022.

[47] "Libpng," http://www.libpng.org/pub/png/libpng.html, 2022.

[48] "Openssh," https://github.com/openssh/openssh-portable, 2022.

[49] "Stringencoders," https://github.com/client9/stringencoders, 2022.

[50] "Github," https://octoverse.github.com/, 2021.

[51] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards understanding third-party library dependency in c/c++ ecosystem," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[52] "Antlr," https://www.antlr.org/, 2022.

[53] "Openssl," https://github.com/openssl/openssl, 2022.

[54] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.

[55] "Halstead," https://en.wikipedia.org/wiki/Halstead_complexity_measures, 1977.

[56] P. Oman, J. Hagemeister, and D. Ash, "A definition and taxonomy for software maintainability," *Moscow, ID, USA, Tech. Rep*, pp. 91–08, 1992.

[57] "Zlib," https://github.com/madler/zlib, 2022.

[58] "F-score," https://en.wikipedia.org/wiki/F-score, 1992.

[59] "Snky cli," https://snyk.io/, 2022.