Third-Party Library Dependency for Large-Scale SCA in the C/C++ Ecosystem: How Far Are We?

Ling Jiang[†]
Southern University of Science and Technology
Shenzhen, China
11711906@mail.sustech.edu.cn

Hengchen Yuan
Southern University of Science and Technology
Shenzhen, China
11911202@mail.sustech.edu.cn

Qiyi Tang
Tencent Security Keen Lab
Shanghai, China
dodgetang@tencent.com

Sen Nie
Tencent Security Keen Lab
Shanghai, China
snie@tencent.com

Shi Wu
Tencent Security Keen Lab
Shanghai, China
shiwu@tencent.com

Yuqun Zhang[*]
Southern University of Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

ABSTRACT

Existing software composition analysis (SCA) techniques for the C/C++ ecosystem tend to identify the reused components through feature matching between target software project and collected third-party libraries (TPLs). However, feature duplication caused by internal code clone can cause inaccurate SCA results. To mitigate this issue, *Centris*, a state-of-the-art SCA technique for the C/C++ ecosystem, was proposed to adopt function-level code clone detection to derive the TPL dependencies for eliminating the redundant features before performing SCA tasks. Although *Centris* has been shown effective in the original paper, the accuracy of the derived TPL dependencies is not evaluated. Additionally, the dataset to evaluate the impact of TPL dependency on SCA is limited. To further investigate the efficacy and limitations of *Centris*, we first construct two large-scale ground-truth datasets for evaluating the accuracy of deriving TPL dependency and SCA results respectively. Then we extensively evaluate *Centris* where the evaluation results suggest that the accuracy of TPL dependencies derived by *Centris* may not well generalize to our evaluation dataset. We further infer the key factors that degrade the performance can be the inaccurate function birth time and the threshold-based recall. In addition, the impact on SCA from the TPL dependencies derived by *Centris* can be somewhat limited. Inspired by our findings, we propose *TPLite* with *function-level origin TPL detection* and *graph-based dependency recall* to enhance the accuracy of TPL reuse detection in the C/C++ ecosystem. Our evaluation results indicate that *TPLite* effectively increases the precision from 35.71% to 88.33% and the recall from 49.44% to 62.65% of deriving TPL dependencies compared with *Centris*. Moreover, *TPLite* increases the precision from 21.08% to 75.90% and the recall from 57.62% to 64.17% compared with the SOTA academic SCA tool B2SFinder and even outperforms the well-adopted commercial SCA tool BDBA, i.e., increasing the precision from 72.46% to 75.90% and the recall from 58.55% to 64.17%.

CCS CONCEPTS

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

KEYWORDS

Software Composition Analysis, Code Clone Detection, Mining Software Repositories

ACM Reference Format:
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 '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598143

1 INTRODUCTION

Software composition analysis (SCA) [9, 27] intends to identify and manage the open-source components contained in software projects, where the components refer to the reused TPLs with their corresponding versions. Relying on the SCA results, developers can effectively track the potential threats introduced to software projects, such as vulnerability propagation [8] and license violation [13, 81]. For the widely-spread C/C++ ecosystem, while many proposed SCA techniques [11, 13, 58, 68, 78] advance the component identification by matching the features between target software project and collected TPLs based on their similarities, they have also been shown ineffective when encountering internal code clone [13, 68] where one TPL depends on other TPLs via code reuse. Such an issue can further cause inevitable feature duplication across collected TPLs which can compromise the SCA results, i.e., incurring false positives during feature matching.

† Ling Jiang is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China.
* Yuqun Zhang is the corresponding author. He is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, China.

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598143

Recently, *Centris* [68], a state-of-the-art SCA technique for the C/C++ ecosystem, has been proposed to mitigate the issue of feature duplication by deriving the TPL dependencies when building the SCA database (in this paper, one TPL dependency represents the code reuse relation between TPLs, e.g., a dependency from tpl_s to tpl_r indicates that tpl_s reuses tpl_r). To derive the TPL dependencies, since there exists no unified package management tool (e.g., Maven [3] in JAVA) to manage TPL dependencies in the C/C++ ecosystem, *Centris* adopts function-level code clone detection to identify duplicated functions between TPLs. Accordingly, *Centris* determines the exact TPL dependency under the assumption that the reused TPL should have earlier function birth time and then eliminates the redundant features, e.g., redundant function signatures, introduced by the reused TPLs. Although *Centris* has shown significant enhancement of SCA precision (from 5% to 95%) in the original paper, it does not evaluate the accuracy of the derived TPL dependencies. Meanwhile, the evaluation of how the derived TPL dependencies impact the SCA results is limited (only on four software projects in the original paper). Moreover, the SCA tasks are performed at the source code level which leads to a potential availability issue for real-world applications. Therefore, it is unclear whether the effectiveness of *Centris* can be generalized to other evaluation datasets and setups.

In this paper, to extensively understand the efficacy and limitations of *Centris*, we investigate the accuracy of its derived TPL dependencies and their impacts on the SCA results. To this end, we construct two ground-truth datasets composed of 2,150 TPL dependencies out of 1,035 TPLs for TPL reuse detection and 128 binary files for SCA respectively. To our best knowledge, they are both the largest ground-truth datasets of their respective domain in the existing literature. Our study results suggest that the accuracy of TPL dependencies derived by *Centris* and their impacts on SCA may not well generalize to our evaluation dataset. Specifically, *Centris* only achieves 35.71% precision and 49.44% recall for deriving the TPL dependencies based on the ground-truth data, which are rather unsatisfying as suggested by prior works [38, 57, 63]. Moreover, by adapting the TPL dependencies derived by *Centris* in all the 10,241 TPLs of its original paper to a well-established commercial binary SCA engine BinaryAI [6], the precision only increases from 25.76% to 56.12% and the recall even decreases from 56.34% to 53.28%, indicating that the performance advantage is somewhat limited compared with the original paper (i.e., increased precision from 5% to 95% with the same recall). We further investigate the major factors that impact the performance of *Centris* and find that the inaccurate function birth time and the threshold-based recall can potentially compromise the effectiveness of the TPL reuse detection by *Centris*.

Inspired by the findings of our study, we propose *TPLite* to improve over *Centris* in terms of the accuracy of TPL reuse detection. First, *TPLite* applies *function-level origin TPL detection* which supplements the birth time with hierarchical path information and meta information, i.e., included header files, software bill of materials files, and licenses, as the input to detect the TPL from which the target function originates (defined as the *origin TPL* in this paper). Furthermore, *TPLite* adopts *graph-based dependency recall* to derive the TPL dependencies upon the collected origin TPLs. More specifically, *TPLite* first applies *coarse-grained detection* to initialize the TPL dependency collection and then uses *centrality-based filter* to identify and remove the invalid TPL dependencies based on the PageRank [46] algorithm. Our evaluation results indicate that *TPLite* effectively enhances the accuracy of deriving TPL dependencies, i.e., the precision increases from 35.71% to 88.33% and the recall increases from 49.44% to 62.65% compared with *Centris*. Meanwhile, we also apply *TPLite* for SCA via adapting their derived TPL dependencies to BinaryAI [6]. We find that *TPLite* can also significantly outperform *Centris* by improving the precision from 56.12% to 75.90% and the recall from 53.28% to 64.17% for SCA. Moreover, *TPLite* even outperforms the well-adopted state-of-the-art commercial SCA tools, e.g., BDBA [11], by improving the precision from 72.46% to 75.90% and recall from 58.55% to 64.17%.

To summarize, our paper makes the following contributions.

- **Dataset:** We construct two ground-truth datasets with 2,150 dependencies across 1,035 TPLs and 128 binary files for the C/C++ ecosystem, which can serve as the benchmark suites for future studies of TPL reuse detection and binary-level SCA.
- **Study:** We perform an extensive study on the state-of-the-art SCA technique for the C/C++ ecosystem *Centris* on our large-scale dataset. We find that both its accuracy of deriving TPL dependencies and its impact on SCA may not well generalize to our evaluation dataset. Moreover, inaccurate function birth time and threshold-based recall can be the key factors to degrade the performance of *Centris*.
- **Technique:** We propose *TPLite* based on the study findings to significantly improve the accuracy of deriving TPL dependencies. Moreover, we are the first to adapt the TPL dependencies to the binary-level SCA engine and find that it outperforms the existing state-of-the-art binary-level SCA tools.

2 BACKGROUND

2.1 Software Composition Analysis

Software composition analysis (SCA) typically refers to the component identification in the target software project for tracking the security threats and license violations introduced by TPLs [9, 13, 27]. Many existing SCA techniques [13, 35, 68, 78, 80] construct a large-scale TPL database with their extracted software features (generally the syntactic features such as string literal [13, 21, 78] and function signature [29, 31, 68]) and then identify the third-party components by matching the TPL features based on their similarities with the target software project. In particular, when the number of the common features between the target software project and a TPL exceeds a preset threshold, we recognize the TPL as a component of the target software project.

Prior works [13, 68, 78] indicate that identifying the third-party components is challenging in large-scale SCA due to the inevitable internal code clone [13, 68] (i.e., the given TPL reuses the source code of other TPLs directly) which can potentially cause vast false positives. In particular, when one TPL highly reuses the code of other TPLs, it is likely that such TPLs would be also recognized as the components of target software project. As a result, the accuracy of component identification can be degraded by introducing irrelevant vulnerabilities/licenses [13, 78]. Figure 1 presents an internal

code clone example where TPLs *nodejs*, *openMVG* and *assimp* directly reuse the whole project of TPL *zlib*. More specifically, *nodejs* even indirectly reuses *zlib* via directly reusing TPL *v8*. Applying many existing SCA techniques [13, 21, 61, 78] in this case can easily result in false positives, e.g., while the target software project actually only reuses *zlib*, other TPLs reusing *zlib* are also recognized as the components in the target software project. Note that multiple existing SCA techniques attempt to alleviate such false positives. Specifically, OSSPolice [13] and B2SFinder [78] propose hierarchical schemes based on source directory and filtering policies respectively to reduce the false positives in SCA. However, their strategies are only designed for the intact code reuse of the TPLs (i.e., allowing no structural or source code modifications of the TPL) and thus compromise the SCA results.

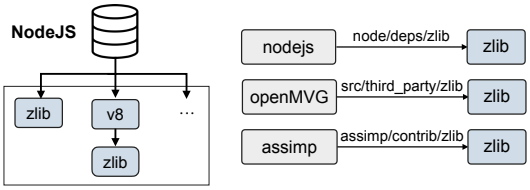


Figure 1: Illustration of internal code clone

2.2 State-of-the-Art *Centris*

Centris [68], a state-of-the-art SCA technique for the C/C++ ecosystem, utilizes code clone detection to derive the TPL dependencies upon large-scale TPL collection and eliminates the redundant features introduced by internal code clone. Figure 2 presents the workflow of *Centris*. Specifically, *Centris* first extracts the function signature information (i.e., hash value of the function body) across all versions/tags as the feature for each collected TPL. Moreover, *Centris* performs the TPL reuse detection to derive the TPL dependencies in the form of a directed graph where each TPL is denoted as a vertex and each TPL dependency is denoted as an edge. Accordingly, *Centris* removes the redundant features for each TPL, i.e., for a given TPL, removing the union of the features of all its reused TPLs. Next, *Centris* generates the TPL database via inverted index mapping of features to the corresponding TPLs and identifies the components online via feature matching as other SCA techniques [13, 61, 62, 76, 78].

Then we further illustrate how *Centris* performs the TPL reuse detection. First, *Centris* obtains the duplicate functions between two TPLs via code clone detection. Meanwhile, *Centris* extracts the birth time for each duplicate function, i.e, the earliest release time in the corresponding TPL. Next, assuming that for a given TPL, its reused TPL should enable earlier function birth time, *Centris* computes whether the ratio of the amount of the duplicate functions with earlier birth time to the total function amount of the reused TPL surpasses a preset threshold θ (i.e., 0.1) to derive the TPL dependency. Note that while many approaches [42, 71, 74, 77] have been proven effective for clone detection, they are not applicable for deriving TPL dependency and thus fail to alleviate redundant features.

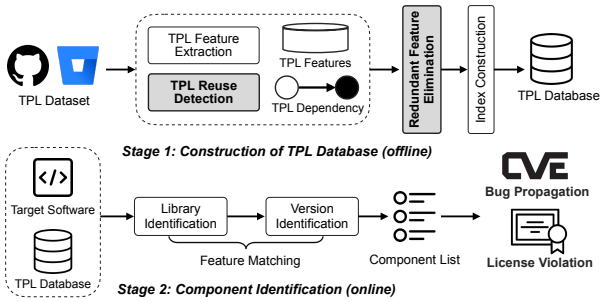


Figure 2: The workflow of *Centris*

We notice that while *Centris* is proposed to detect TPL reuse, the accuracy of its derived TPL dependency is not evaluated in the original paper. Meanwhile, the evaluation of the impact of the derived TPL dependencies on SCA is limited. Such facts potentially compromise the effectiveness of *Centris*. Thus, there is a pressing need for an extensive study on *Centris* to comprehensively delineate its effectiveness and limitations in TPL reuse detection and SCA.

3 STUDY ON CENTRIS

3.1 Dataset

To extensively study how *Centris* performs in TPL reuse detection and SCA tasks respectively, we need to construct the datasets of ground-truth TPL dependencies and SCA results, as many existing works [62, 68, 76, 78, 80].

3.1.1 Ground-truth TPL dependencies. To our best knowledge, there is no existing well-formed dataset of TPL dependencies of C/C++ programs. In this paper, considering the workload of manual calibration, we strictly follow previous works [13, 63, 76, 80] to select top 1K mostly reused TPLs from the dataset adopted by the original *Centris* paper which includes the open-source projects from the GNU/Linux community (e.g., *glibc* [5] and *e2fsprogs* [1]) and C/C++ GitHub repositories with over 1K stars (e.g., *zlib* [54] and *libjpeg* [50]). We then derive their dependencies after rigorously analyzing the directory, file paths, included headers, and the information contained in the copyright, license, and README for each TPL. As a result, we extract 2,150 TPL dependencies as the ground truth.

3.1.2 Ground-truth SCA results. Note that in the original paper of *Centris*, the ground-truth SCA results are not publicly available. In this paper, we determine to derive the ground-truth SCA results on top of the collected 10,241 TPLs of the *Centris* paper. In particular, we target binary-level SCA rather than the original source-code-level SCA due to following reasons. First, binary-level SCA can be generally more reliable than source-code-level SCA when source code is unavailable [20]. Next, binary-level SCA can be integrated at the development or deploy phase during DevOps [61, 62, 76, 78]. However, source-code-level SCA like *Centris* in the original paper can only analyze the source files before the build phase which can be potentially risky since the components introduced during compilation can potentially have vulnerabilities, e.g., dynamic-link library [21]. Note that although *Centris* performs source-code-level

Table 1: SCA test case samples

Software	Binary	Version	Sys/Arch†	#TPLs	Sample TPLs
terarkdb	db_bench	v1.3.6	arch linux/x86_64	15	bzip2, zlib, lz4, xxHash
ClickHouse	clickhouse	v22.1.2.2	macOS/arm64	61	libxml2, grpc, libexpat
TIC-80	tic80.exe	v0.90.1723	windows/x86_64	15	blip-buf, libpng, dirent
kvrocks	kvrocks	v2.0.5	ubuntu/i386	12	glibc, libevent, rocksdb
Tendis	tendisplus	v2.4.3	ubuntu/x86_64	15	glibc, rapidjson, snappy

† The system and architecture applied to compile the binary

SCA in the original paper, we can still utilize its derived TPL dependencies to remove redundant features and construct the TPL database for binary-level SCA.

We then construct our binary-level SCA testset compiled by 75 C/C++ open-source software projects with manually labeled components as the test cases. In particular, we first attempt to identify influential projects with over 1K stars in their corresponding GitHub repositories, and further select the ones with over 10 submodules since such projects are more likely to have multiple reused TPLs to facilitate SCA. Next, we compile the source code of the 75 selected projects into 128 binary files as input for online component identification based on multiple compiler configurations. More specifically, we follow previous works [13, 61, 78] to strip all the binary files where variable names and function names are removed. Furthermore, we hook the compiler (i.e., `gcc` [4]) to parse the DWARF information [2] during compilation and extract the compiled files for deriving their contained components as the labeled data. Table 1 presents multiple SCA test case samples. For instance, project *ClickHouse* (an open-source database management system with 27K+ stars in GitHub [51]) is compiled into the binary file *clickhouse* which contains 61 manually labeled components (e.g., *grpc*) as the ground-truth SCA results.

To our best knowledge, the two ground-truth datasets above are the largest datasets in their respective domain, which are both presented in our GitHub repository [55].

3.2 Experimental Setup

To evaluate the accuracy of TPL reuse detection (i.e., deriving TPL dependencies) of *Centris* and its impact on the SCA results respectively, we adopt the following metrics: true positives (TP, correct TPL dependencies derived by *Centris*), false positives (FP, incorrect TPL dependencies derived by *Centris*), false negatives (FN, missed TPL dependencies in the ground-truth data), *Precision* (i.e., the ratio of true positives to the resulting TPL dependencies by *Centris*), *Recall* (i.e., the ratio of true positives to the ground-truth TPL dependencies), and *F1* score (i.e., measuring accuracy by combining precision and recall). All our evaluation results are averaged out of 10 runs to reduce the impact from randomness. All the experiments are conducted on a machine with Linux VM-187-4-centos 5.4.119, AMD EPYC 7K62 48-Core Processors and 32 GiB RAM.

3.3 Research Questions

We investigate following research questions for studying *Centris*:

- **RQ1:** *How does Centris perform in TPL reuse detection and SCA?* For this RQ, we evaluate the performance of *Centris* with the accuracy of the derived TPL dependencies and the SCA results.
- **RQ2:** *What are the major factors that impact the performance of Centris?* For this RQ, we further investigate the factors that impact the performance of *Centris* based on the results from RQ1.

3.4 Results and Analysis

3.4.1 RQ1: Performance of Centris. We first investigate the accuracy of the derived TPL dependencies by *Centris* on top of our collected ground-truth 2,150 TPL dependencies (as in Section 3.1.1). Table 2 demonstrates the evaluation results in terms of diverse reuse ratio threshold (defined as θ) setups. Note that in addition to retain the θs adopted by the original *Centris* paper (i.e., 0.05, 0.1, 0.15, and 0.2), we also include more θs. Specifically, we adopt 0.01 to mainly assess the FN performance and 0.5 and 0.75 to mainly assess the precision of the TPL dependencies when only highly reused TPLs can be recognized. We can observe while the original version of *Centris* (i.e., θ is 0.1) achieves the optimal performance in terms of the *F1* score (41.47%), it is somewhat unsatisfying. To illustrate, *Centris* results in around 2X false positives (1,914) compared with true positives (1,063) and misses nearly half of the ground-truth TPL dependencies (1,087). We further investigate the 1,914 false positives and find that 286 out of them are opposite to the ground truth. For example, while in fact, project *tigervnc* reuses project *zlib*, *Centris* derives that *zlib* reuses *tigervnc*. Such an opposite TPL dependency can lead to feature elimination errors, i.e., the features can be mistakenly removed from *zlib* and retained in *tigervnc*, and further cause component identification errors in SCA. Moreover, we can also observe from Table 2 that no matter how to set θ, *Centris* results in limited precision/recall. To illustrate, even when θ is set to 0.75 which indicates more than three-quarters of the functions are reused between TPLs, the precision is only 62.71% with 37 true positives and 22 false positives. On the other hand, *Centris* only recalls 69.35% TPL dependencies when θ is set to 0.01 and misses 659 dependencies in the ground truth. Therefore, we can summarize that the *Centris* is somewhat ineffective in deriving TPL dependencies.

Finding 1: The accuracy of TPL dependencies derived by Centris may not well generalize to other datasets.

Table 2: Accuracy of TPL reuse detection by *Centris*

Threshold	Verification of TPL dependency						
	Total	#TP	#FP	#FN	Precision(%)	Recall(%)	F1(%)
0.75	59	37	22	2,113	62.71	1.72	3.35
0.50	261	159	102	1,991	60.92	7.40	13.20
0.20	1,611	662	949	1,488	41.09	30.79	35.20
0.15	2,118	839	1,279	1,311	39.61	39.02	39.31
0.10	2,977	1,063	1,914	1,087	35.71	49.44	41.47
0.05	4,349	1,300	3,049	850	29.89	60.47	40.01
0.01	8,447	1,491	6,956	659	17.65	69.35	28.14

We further evaluate how TPL dependencies impact on the large-scale binary-level SCA upon a well-established commercial binary SCA engine BinaryAI [6], which uses string literals as features to match target binary files and source code of TPLs. As described in

Section 2.1, we can expect to delete the duplicate software features based on the correct TPL dependencies for augmenting the SCA accuracy. Table 3 presents the results of the online component identification of SCA in terms of multiple TPL dependency options, i.e., ground-truth TPL dependencies (denoted as "Ground truth-1k"), the TPL dependencies derived by *Centris* on top of 1K mostly reused TPLs (denoted as "*Centris*-1k"), the TPL dependencies derived by *Centris* on top of all TPLs (denoted as "*Centris*-10k"), and no TPL dependency adopted (denoted as "No dependency"). We can observe that adopting ground-truth TPL dependencies can effectively improve the SCA precision (from 25.76% to 45.90%) and recall (from 56.34% to 61.17%) compared with adopting no TPL dependency. Such results indicate that redundant feature elimination based on even limited number of TPL dependencies (2,150) can already effectively alleviate the issue of internal code clone and improve the accuracy of SCA. Moreover, adopting *Centris*-1k incurs inferior SCA precision compared with adopting ground-truth TPL dependencies (32.44% vs. 45.90%). It even incurs lower recall than adopting no TPL dependency (50.71% vs. 56.34%). Such results indicate that *Centris* tends to derive invalid TPL dependencies which lead to erroneous redundant feature elimination in TPLs during SCA. Moreover, we also find adopting larger-scale TPL dataset for *Centris*, i.e., *Centris*-10k, incurs limited $F1$ score improvement (from 52.45% to 54.66%) compared with adopting ground-truth TPL dependencies. Meanwhile, noticing that *Centris* evaluates the impact of TPL dependency only on four projects in the original paper where the SCA precision is improved from 5% to 95% after performing the redundant feature elimination, our study on the 128 binary files compiled from the 75 selected projects shows that the SCA precision is increased from 25.76% to 56.12% which indicates that the performance of *Centris* may not well generalize to diverse scenarios. In summary, we can derive from all the facts that *Centris* delivers limited effectiveness of applying the TPL reuse detection for SCA.

Finding 2: While the redundant feature elimination based on TPL dependencies can advance the SCA results, Centris may be limited in leveraging the power of TPL reuse detection.

Table 3: SCA results with TPL dependency

TPL dependency	Verification of SCA results		
	Precision(%)	Recall(%)	F1(%)
No dependency	25.76	56.34	35.35
Ground truth-1k	45.90	61.17	52.45
Centris-1k	32.44	50.71	39.57
Centris-10k	56.12	53.28	54.66

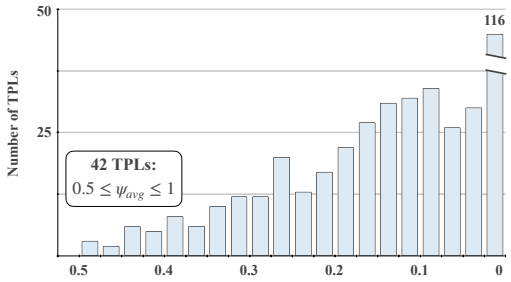
3.4.2 RQ2: Impact factors of Centris. Previous findings indicate that the accuracy of the TPL dependencies derived by *Centris* is limited and further compromises the SCA results. We then analyze the potential reasons behind the limitations of *Centris*. In particular, we follow prior works [61, 62, 68, 80] to study the reasons behind limited precision (reflected by false positives) and recall caused by *Centris*. First, for the false positives, we infer that they are largely caused by inaccurate function birth time. To illustrate, Table 4 lists multiple TPLs commonly reusing source file `deflate.c` from TPL *zlib* with their respective corresponding birth time. Surprisingly, all the functions of `deflate.c` share the same birth time (i.e., 2011-09-10) under commit-bcf78a2 [54] while we discover that these functions were actually first introduced in 1995 [49]. We then realize that the *zlib* project had no version control in early years and was migrated to git later. As a result, all its prior time stamps were lost during migration, resulting in inaccurate birth time information. Furthermore, we manually analyze all the false cases (1,914 FP and 1,087 FN) and find that 1,336 FP and 215 FN are caused by inaccurate function birth time. Note that to determine the correct function birth time, we manually search all the information in the repository, including the link to the original project, and the time that appeared in the project copyright. Nevertheless, the reason behind the incorrect birth time is elusive, as there are limited hints to track the explicit reasons–some TPLs (e.g., *zlib*) lose their history due to migration of repository, or some code of TPLs (e.g., *lzma*) is reused before tagging a specific version.

Finding 3: Inaccurate function birth time significantly compromises the accuracy of Centris for TPL reuse detection.

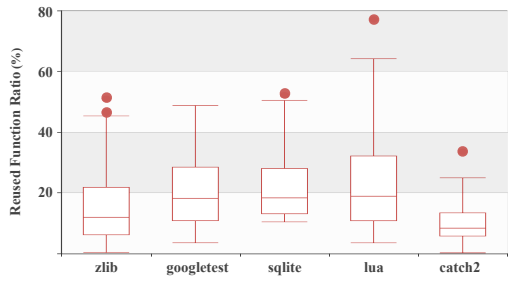
Table 4: Birth time of reused functions from zlib

TPL Name	Source file directories	Birth time
rsync	zlib/deflate.c	1998-05-14 07:22:45
wxWidgets	src/zlib/deflate.c	1998-05-20 14:02:15
gdal	frmts/zlib/deflate.c	2001-09-15 21:50:31
mysql-server	zlib/deflate.c	2002-04-21 10:06:34
llvm-project	llvm/runtime/zlib/deflate.c	2004-03-19 21:59:23
CMake	Utilities/cmzlib/deflate.c	2006-04-18 20:40:40
libpng	deflate.c	2009-04-16 15:46:37
tigervnc	common/zlib/deflate.c	2009-04-30 11:41:03
libwebsockets	tmp/win32port/zlib/deflate.c	2011-04-24 07:12:38
zlib	deflate.c	2011-09-10 06:19:21

We then investigate the potential reason behind limited recall caused by *Centris*. In general, we can easily observe that the recall performance of *Centris* is highly related with the different setups of reuse ratio threshold θ. For instance, *Centris* achieves the recall of 49.44% and 69.35% under the θs 0.10 and 0.01 respectively, i.e., around 20% ground-truth TPL dependencies can be recalled under θ 0.01 and cannot be recalled under θ 0.10. We then infer that the less reuse between TPLs is, the lower recall it causes. Furthermore, we suspect that fixating the reuse ratio threshold θ can hardly optimize the recall performance of *Centris*. To illustrate, for a given tpl_i and any other possible tpl_j, we first compute its reuse ratio $\psi_j(i)$ which is the percentage of function amount that tpl_j reusing tpl_i over the total function amount of tpl_i, and then we derive the average reuse ratio $\psi_{avg}(i)$ upon all the collected $\psi_j(i)$s. Figure 3a presents the number of TPLs distributed in different ψ_{avg} intervals. We can observe that ψ_{avg}s enormously vary for different TPLs. For instance, more than 50% functions are reused in 42 TPLs while less than 2.5% functions are reused in 116 TPLs. We further investigate how ψ distributes in specific TPLs. Specifically, we target the five



(a) Average reuse ratio



(b) TPL reuse distribution

Figure 3: Illustration of the TPL reuse pattern

mostly reused TPLs (i.e., *zlib*, *googletest*, *catch2*, *sqlite* and *lua*) in our ground-truth dataset. Figure 3b presents that ψ can be distributed quite divergently among different dependencies for specific TPLs. For instance, $\psi(zlib)$ ranges from 0.27% to 50.96%. To summarize, setting a constant ψ threshold may not well reflect the divergent ψ distribution for a single TPL, let alone all TPLs.

> *Finding 4: The reuse ratio can be quite divergent for different TPL dependencies. Thus, a fixed threshold to denote the reuse ratio may not well generalize to different TPL dependencies.*

3.5 Discussion

Our study findings so far have indicated that the accuracy of TPL dependency by *Centris* and its impact on the binary-level SCA are somewhat limited in our evaluation dataset. As mentioned in Section 2.2, *Centris* derives the TPL dependencies using the function birth time and recalls them based on the fixated reuse ratio threshold. In this section, we first attempt to discuss the limitations of the mechanism of *Centris*. In particular, as in Finding 3, due to the inaccurate function birth time, it is error-prone to detect the TPL from which the target function originates, i.e., the origin TPL, which thus can decrease the precision of the TPL dependency derived by *Centris* and further cause erroneous redundant feature elimination. Intuitively, one can use more information to augment the accuracy when identifying origin TPLs, such as the source directory information of TPLs. For instance, when *rsync* is incorrectly identified as the origin TPL for the functions in file `deflate.c` solely based on its birth time, using its source directory information (i.e., `zlib/delate.c`) in *rsync* can correctly identify that the inclusive functions in `deflate.c` should originate from *zlib*. In addition, more meta information from the repositories containing the corresponding TPLs can enhance the accuracy of identifying the origin TPL, e.g., for the file `deflate.c`, its included header files (e.g., <#include "zlib.h">), software bill of materials (SBOM) files for the package management tools [63] (e.g., CMakeLists.txt, configure.ac), and licenses, since such meta information can potentially include the hints to infer the dependency in C/C++ projects [13, 16, 45, 48]. Furthermore, noticing that modified code reuse (i.e., functions being reused with modified code) commonly exists in TPL dependencies where similar functions share the same origin TPL, we consider it is essential to identify such modified code reuse prior to applying the aforementioned information for identifying the origin TPL.

Finding 4 indicates that it is rather challenging to derive the accurate TPL dependency with a fixed reuse ratio threshold. Intuitively, we could rebuild the way to derive recall to improve the accuracy of deriving TPL dependencies. More specifically, we can first adopt *Centris* to initialize the dependencies in a coarse-grained manner to include as many potential TPL dependencies as possible. Then we can further develop specific rules to identify and filter the introduced invalid TPL dependencies to increase the precision while maintaining the recall performance. Note that *Centris* models the TPL dependencies as a directed graph in the original paper (as introduced in Section 2.2), which can somewhat compromise the effectiveness of the downstream SCA task since *Centris* would drop all the overlapped features from the TPL database if the corresponding TPL dependencies form a (partial) cyclic graph. For instance, assume tpl_s and tpl_r include functions $\{s_1, s_2, r_1\}$ and $\{r_1, r_2, s_1\}$ respectively. If tpl_s and tpl_r form reuse relations on each other in the derived TPL dependencies, *Centris* would completely drop the overlapped functions $\{s_1, r_1\}$, leaving only $\{s_2\}$ and $\{r_2\}$. Such feature loss can limit the effectiveness of performing the SCA task [13, 61, 78]. However, if we identify the reuse relation from only one side (e.g., tpl_s reuses tpl_r) to retain the features $\{s_1, r_1\}$ in tpl_r, such a performance issue on SCA could be alleviated. Thus, we tend to model the TPL dependencies in the form of a directed acyclic graph (DAG) to contain full features for facilitating SCA. Accordingly, we can assign each edge a weight reflecting the reuse ratio $\psi_j(i)$ given tpl_j reusing tpl_i, and use well-established mechanisms such as the centrality algorithms [18, 28, 46] to further analyze the dependency graph.

Presumably, we can model the centrality to reflect the influence of a node in the graph (i.e., TPL in dependency graph) and adopt multiple metrics to measure the centrality [43] such as the degree centrality which simply reflects the number of neighbors. However, applying degree centrality tends to cause local optima and can hardly manifest the influence of a node in the entire graph [7, 72]. On the contrary, since eigenvector centrality measures the transitive influence of neighbors and iteratively calculates the global influence for each node, we thus can render the dependencies that originate from high-scoring nodes to contribute more to their neighbors. Note that prior works [17, 37] show that degree centrality and eigenvector centrality tend to be highly consistent in large and stable relationship graphs (e.g., web links). Accordingly, their divergences can be potentially used to indicate the validity of the graph relationships. More specifically, nodes with high degree (here we only discuss in-degree which indicates the number of incoming

dependencies to the TPL) centrality and eigenvector centrality in TPL dependency are expected to be widely reused (e.g., *zlib*). On the other hand, nodes with high eigenvector centrality but low degree centrality indicates abnormal TPL dependency, i.e., the influence of these nodes is contributed to by high-scoring neighbors while they have rather limited incoming edges. To illustrate, we consider such edges indicate invalid TPL dependencies because such a TPL reuse pattern generally contradicts the common TPL reuse pattern in real-world software development.

4 APPROACH: *TPLITE*

Inspired by our findings of the study on *Centris* and the discussion thereafter, we propose *TPLite* to improve over *Centris* in terms of the accuracy of TPL reuse detection and SCA. Figure 4 presents the framework of *TPLite* which consists of two components—*function-level origin TPL detection* and *graph-based dependency recall*. In particular, *TPLite* first adopts *function-level origin TPL detection* which not only extracts function birth time but also utilizes source directory information to identify the origin TPLs (Section 4.1). Next, *TPLite* applies *graph-based dependency recall* to derive the TPL dependencies upon the collected origin TPLs via the centrality algorithm (Section 4.2).

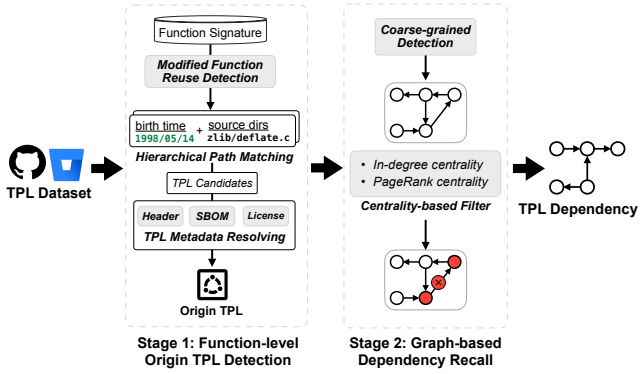


Figure 4: The framework of *TPLite*

4.1 Function-level Origin TPL Detection

Function-level origin TPL detection consists of three steps: (1) modified function reuse detection, (2) hierarchical path matching, and (3) TPL metadata resolving, as illustrated in Algorithm 1.

4.1.1 Modified Function Reuse Detection. Prior works [13, 41, 60, 67, 68] indicate that code reuse with code changes, i.e., modified code reuse, commonly exists in TPL dependencies. For instance, project *SQLiteCpp* reuses a single file *sqlite.c* [52] which is "an auto-generated amalgamation of all the sources files" from *sqlite* [53], where `SQLITE_PRIVATE` is added as an access modifier for all the functions during auto-generation. Thus it is essential to identify all the modified code reuse before detecting that all the functions actually originate from *sqlite*. To this end, *TPLite* is initialized to perform modified function reuse detection to identify the functions similar with input function *func* (line 2 in Algorithm 1) because they are more likely to share the same origin TPL. Note that multiple

Algorithm 1: Function-level Origin TPL Detection

```
   Input: func ;                                   ▷ Source code of function
   Result: origin_tpl
1  Function DetectOriginTPL:
2      similar_funcs ← DetectModifiedReuse(func) ;  ▷ Token-based detection
3      candidate_tpls ← set()
4      for f_i in similar_funcs do
5          tpls_f_i ← set of TPLs containing the function f_i
6          source_dirs ← ExtractSourcePath(tpls_f_i, f_i) ; ▷ Across all versions
7          terms_count ← ∅
8          for p_i in source_dirs do
9              terms ← PathHierarchyTokenizer(p_i)
10             terms_count.update(terms)
11         end
12         terms_sort ← Sort(terms_count) ;        ▷ Sort terms by frequency
13         for t_i in terms_sort do
14             if t_i contains the name of TPL in tpl_f_i then
15                 tpl_path ← earliest TPL with the name in t_i
16                 candidate_tpls.add(tpl_path) ;   ▷ Based on source dirs
17                 break
18             end
19         end
20         tpl_time ← TPL with the earliest birth time t(tpl, f_i) in tpls_f_i
21         candidate_tpls.add(tpl_time) ;   ▷ Based on birth time as Centris
22     end
23     origin_tpl ← ResolveMeta(candidate_tpls) ;   ▷ Header, SBOM, License
24 return
```

existing code clone detectors [13, 24, 56] including *Centris* adopt locality-sensitive hashing (LSH) to detect modified code reuse by measuring the similarity between hashes. However, LSH is argued to be limited in detecting similar functions only with the provided cutoff value [32]. For instance, the cutoff for TLSH (one type of LSH adopted by *Centris*) is set to be 30 (i.e., two functions are considered to share the same origin TPL if the cutoff between two hashes is less than 30) in [44] while the TLSH distance of the reused function `sqlite3SrcListAppend` in *sqlite.c* is 50. Thus, such a modified reuse cannot be detected, degrading the accuracy of origin TPL identification. On the other hand, many recent code clone detectors [15, 66, 77, 84, 85] utilize deep learning models (e.g., DPCNN [26] for CodeCMR [77]) with semantic features of code. However, their overhead of model training and source code semantic feature extraction have been widely recognized to seriously limit the scalability [15, 22, 33, 34, 65]. Thus, to realize the trade-off between accuracy and scalability, we adopt the existing token-based clone detector SourcererCC [57], which is specifically designed for large code base at the function level. In particular, SourcererCC first tokenizes the function source code and then exploits an optimized inverted index to quickly query the cloned functions.

4.1.2 Hierarchical Path Matching. Previous findings indicate that function birth time can be incorrect when deriving TPL dependencies. To tackle this problem, we utilize the hierarchical path features (i.e., source directory information) to help identify the origin TPL since the TPL reuse tends to retain structural similarity (e.g., *rsync* reuses *zlib* where all the reused functions have the source directory containing "zlib") on the large-scale TPL dataset [13, 38].

Specifically, Algorithm 1 (lines 3-22) presents the workflow of hierarchical path matching. First, we extract the source directories for all the TPLs containing the target function f_i (lines 5-6), e.g., the source directories for all the functions of file `deflate.c` in Table 4. Next, we adopt the path hierarchy tokenizer [14] to split on the path separator to generate a directory list, e.g., transforming path "src/zlib/deflate.c" as the directory list ["src", "zlib", "deflate.c", "zlib/deflate.c", "src/zlib/deflate.c"]. Meanwhile, we iteratively update *terms_count* to record the frequency of the directory elements in the list (lines 7-11). Eventually, we sort the directory elements in a descending order of their frequency and fuzzily match them with the names of the TPLs containing f_i, i.e., $tpls\_f_i$. If matched, the corresponding TPL *tpl_path* is retained as the potential origin TPL in *candidate_tpls* (lines 12-19). For instance, by applying *hierarchical path matching* to the reused functions in Table 4, "zlib" and "zlib/deflate.c" are the directory elements with the highest frequency among the TPLs which both match the actual origin TPL *zlib*. Despite structural modification may take place in some TPLs (e.g., *libpng*), it usually exerts limited effect on the directory element frequency in large-scale dataset since the structure tends to be preserved during TPL reuse [13]. In addition to *tpl_path*, we still adopt the TPL with the earliest birth time of f_i, i.e., *tpl_time*, as the potential origin TPL as *Centris* (lines 20 to 21).

4.1.3 TPL Metadata Resolving. To derive the actual origin TPLs out of the potential origin TPL collection from Section 4.1.2, we parse the header files, SBOM files and licenses as follows.

- **Header file.** We parse the statements of header files included in TPL source files to derive the origin TPLs, e.g., the statement <#include "zlib.h"> in *rsync* indicates that the origin TPL for the functions in `deflate.c` should be *zlib* as mentioned in Section 3.4.2.
- **SBOM file.** We adopt the state-of-the-art SBOM parser from *CCScanner* [63] to analyse the SBOMs of the potential origin TPLs. The derived TPLs which are reused by other TPLs are identified as the origin TPLs.
- **License.** We parse the text of license files in the root directory and determine the origin TPL whose name is most frequently present in the license files.

4.2 Graph-based Dependency Recall

To realize *graph-based dependency recall*, we first adopt *coarse-grained detection* which initializes the collection of TPL dependencies. Then we propose the *centrality-based filter* to identify and remove the potentially invalid TPL dependencies via a centrality algorithm.

4.2.1 Coarse-grained Detection. Algorithm 2 (lines 2-9) demonstrates the *coarse-grained detection* process. First, we follow *Centris* to extract all the potential TPL dependencies which are denoted as (tpl_s, tpl_r) pairs. Accordingly, we also collect their corresponding reused functions (denoted as ω, lines 3-4). Furthermore, we retain all the TPL dependencies that satisfy Equation 1 as our initial TPL dependency collection (line 5):

$$\frac{|\omega|}{|R|} > \delta \cdot \frac{|R|}{|R_\varphi|} \tag{1}$$

Algorithm 2: Graph-based Dependency Recall

```
   Input: TPL dataset tpl_set
   Result: tpl_dependencies
1  Function DependencyRecall:
2      tpl_dependencies ← ∅
3      for tpl_s, tpl_r ∈ tpl_set × tpl_set do
4          ω ← reused functions from tpl_s to tpl_r
5          if |ω| > 0 and CoarseGrainedCheck(tpl_s, tpl_r) then
6              R ← set of functions of tpl_r
7              tpl_dependencies.add_edge(tpl_s, tpl_r, weight=|ω|/|R|)
8          end
9      end
10     CentralityFilter(tpl_dependencies)
11 return
12 Function CentralityFilter(tpl_dependencies):
13     tpl_σ_set ← ∅
14     centrality_indegree ← InDegreeCentrality(tpl_dependencies)
15     centrality_pagerank ← PageRank(tpl_dependencies)
16     for tpl in tpl_dependencies.nodes do
17         if Normalize(centrality_pagerank[tpl]) /
             Normalize(centrality_indegree[tpl]) > ϵ then
18             tpl_σ_set.add(tpl)
19         end
20     end
21     for tpl_s, tpl_r in tpl_dependencies.edges do
22         if in-degree(tpl_s) > η and tpl_r ∈ tpl_σ_set then
23             tpl_dependencies.delete(tpl_s, tpl_r)
24         end
25     end
26 return
```

where δ refers to a preset hyperparameter, R_φ refers to all the functions where their corresponding origin TPLs are actually tpl_r, and R refers to all the collected functions in tpl_r. The rationale behind Equation 1 is that we tend to include as many dependencies associated with the TPLs where a large proportion of their functions are reused (i.e., $|R_\varphi|/|R|$) as possible. To illustrate, since Finding 4 indicates the limitation of applying a fixed reuse ratio threshold, we determine to approach the optimal reuse ratios of different tpl_rs dynamically instead of applying a fixed reuse ratio threshold for all TPLs to include as many TPL dependencies as possible. Accordingly, our goal is to dynamically derive the optimal reused ratios for different TPLs in a coarse-grained manner by reducing $\delta \cdot |R|/|R_\varphi|$. Then we build a directed graph with the retained dependencies (lines 6-7) as input of *centrality-based filter* (introduced in Section 4.2.2), where each edge is assigned with the weight $\psi_s(r)$ computed as $|\omega|/|R|$ to reflect the function reuse ratio between TPLs as introduced in Section 3.5.

4.2.2 Centrality-based Filter. We propose *centrality-based filter* to identify and eliminate the invalid edges/dependencies to the TPL with high eigenvector centrality and low degree centrality, namely tpl_σ as in Algorithm 2 (lines 12-26). First, we calculate the in-degree centrality, i.e., the normalized summation of incoming edge weights for each TPL (line 14). Then we adopt the PageRank [46] algorithm (line 15) to calculate its eigenvector centrality (i.e., PageRank centrality) because PageRank is advanced in delivering the global contribution of each node, i.e., TPL in this paper. In particular, the

PageRank centrality of tpl_i is computed as Equation 2.

$$c_i = \alpha \sum_{j}^{n} A_{ij} \frac{c_j}{d_j^+} + \beta \tag{2}$$

where A denotes an adjacency matrix of the weights in TPL dependency with a total of n TPLs, c denotes the centrality vector, and d_j^+ denotes the out-degree of the tpl_j. Note that PageRank algorithm typically includes two parameters (i.e., damping factor $\alpha \in [0, 1]$ and β), and we strictly follow their default settings, i.e., $\alpha = 0.85$ and $\beta = (1 - \alpha)/n$, as suggested by previous studies [12, 39, 75]. To avoid the costly matrix inversion, we use an iterative computation method called the power method [19] to approximate the value of c. Based on the derived in-degree centrality and PageRank centrality for each TPL, we identify it as a tpl_σ when the ratio of its PageRank centrality to its in-degree centrality after normalization exceeds a preset hyperparameter ϵ (lines 16-19). Accordingly, we eliminate the corresponding TPL dependency (tpl_s, tpl_r) where for tpl_s, its in-degree exceeds a preset hyperparameter η and tpl_r is actually a tpl_σ. Eventually, the remaining TPL dependencies form a DAG as inspired by Section 3.5.

Figure 5 presents an example where the dependencies (*swift-clang*, *llvm*) and (*llvm*, *swift-clang*) are both recalled during *coarse-grained detection*, i.e., they both satisfy Equation 1. Nevertheless, the dependency (*llvm*, *swift-clang*) is essentially invalid via manual calibration. After applying the *centrality-based filter* for *swift-clang*, the ratio of its PageRank centrality (7.37e-03) to its in-degree centrality (3.62e-05) after normalization is much larger than the preset hyperparameter ϵ. We thus identify *swift-clang* as tpl_σ and further eliminate the invalid dependency (*llvm*, *swift-clang*).

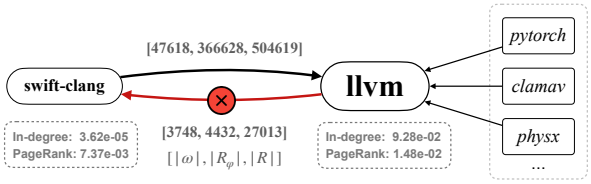


Figure 5: An example of *centrality-based filter*

5 EVALUATION

5.1 Research Questions

We evaluate the performance of *TPLite* with the following research questions. Specifically, we adopt the datasets and setups in our study for evaluation, following prior works [38, 63]:

- **RQ3:** *How do TPLite and its components impact on the accuracy of the TPL dependencies?*
- **RQ4:** *What impact does TPLite exert on the SCA results?*

5.2 Evaluation Setup

To evaluate the effectiveness of TPL reuse detection, we additionally include *CCScanner* [63] which combines *Centris* and SBOM file parsers for performance comparison. In particular, *CCScanner* analyzes and parses different types of SBOM files to generate the TPL dependencies and combines them with the dependencies generated by *Centris* as the final result. To evaluate the SCA results, in addition to *Centris* and *CCScanner*, we adopt the state-of-the-art binary-level SCA tools including two popular commercial products—Black Duck Binary Analysis (BDBA) [11] and Scantist [58] and one academic binary-to-source SCA tool B2SFinder [78]. Specifically, BDBA is one of the most reliable and expensive commercial binary-level SCA tools, which combines static and string-based analysis with fuzzy matching on the knowledge base. Scantist combines source-code-level and binary-level SCA into one platform which scans and remediates open-source security. B2SFinder employs a weighted feature matching algorithm to facilitate the code features extracted from both binary and source files.

To perform an ablation study on the components of *TPLite*, we also build technique variants of *Centris*, i.e., $Centris_{otd}$ (*Centris* with *function-level origin TPL detection*) and $Centris_{otd+cg}$ ($Centris_{otd}$ with *coarse-grained detection*). The hyperparameter δ for Equation 1 is set to 0.01. The hyperparameters ϵ and η in Section 4.2.2 are set to 1.5 and 5 respectively[1] which achieve the optimal accuracy of TPL dependency in terms of $F1$ score. All the evaluation results are averaged out of 10 runs to reduce the impact from randomness.

5.3 RQ3: Accuracy of TPL Reuse Detection

Table 5 presents the results of the derived TPL dependencies. We can observe that overall, *TPLite* achieves the highest $F1$ (73.31%) based on the 2,150 ground-truth TPL dependencies and significantly enhances the precision (35.71% to 88.33%), recall (49.44% to 62.65%), and $F1$ (41.47% to 73.31%) compared with *Centris*. Additionally, *TPLite* also improves over *CCScanner* in terms of precision (36.27% to 88.33%), recall (54.84% to 62.65%), and $F1$ (43.66% to 73.31%).

We further investigate the impact of *TPLite* components. Specifically, we observe that $Centris_{otd}$ reduces 1,522 false positives and increases the precision from 35.71% to 75.83% compared with *Centris* which indicates the effectiveness of *function-level origin TPL detection*. Moreover, $Centris_{otd+cg}$ recalls 206 more TPL dependencies with 122 more true positives compared with $Centris_{otd}$ by adopting the additional *coarse-grained detection*. We further manually calibrate the 122 true positives and find that all of these dependencies suggest partial reuse between TPLs with reused function ratio less than 10%. Eventually, we compare $Centris_{otd+cg}$ and *TPLite* to illustrate the effectiveness of *centrality-based filter*. We can observe that *TPLite* eliminates 303 dependencies with *centrality-based filter* compared with $Centris_{otd+cg}$ where 98.35% of them are actually false positives, increasing the precision from 73.96% to 88.33%. Moreover, *TPLite* and $Centris_{otd+cg}$ have quite close recall (62.65% vs. 62.88%) despite the deleted dependencies. Such results indicate each component developed in *TPLite* can effectively improve over *Centris* in terms of TPL reuse detection.

FP Analysis. We investigate all the 178 false positives caused by *TPLite* and 156 are caused when both tpl_s and tpl_r depend on a TPL that is not included in our adopted TPL dataset. For instance, both *xmrig* and *cpuminer-multi* reuse *crypto* which is not included in our TPL dataset. As a result, the origin TPL of the functions in *crypto* is identified as *cpuminer-multi*, leading to an invalid TPL

[1]We evaluate the impact of different hyperparameter setups and present the results in our GitHub repository [55] due to page limit. Note that applying different hyperparameter setups does not incur significant performance variations, indicating the effectiveness of *TPLite*.

Table 5: Accuracy of TPL dependency

Tool	Metrics of TPL dependency						
	Total	#TP	#FP	#FN	Precision(%)	Recall(%)	F1(%)
Centris	2,977	1,063	1,914	1,087	35.71	49.44	41.47
CCScanner	32,51	1,179	2,072	971	36.27	54.84	43.66
$Centris_{otd}$	1,622	1,230	392	920	75.83	57.21	65.22
$Centris_{otd+cg}$	1,828	1,352	476	798	73.96	62.88	67.97
TPLite	**1,525**	**1,347**	**178**	**803**	**88.33**	**62.65**	**73.31**

$Centris_{otd}$ = *Centris + function-level origin TPL detection*
$Centris_{otd+cg}$ = $Centris_{otd}$ *+ coarse-grained detection*

dependency from *xmrig* to *cpuminer-multi*. Typically, such an issue can be potentially alleviated when we adopt a larger TPL dataset.

FN Analysis. We investigate all the 803 false negatives and 615 are caused due to the black-box reuse [47] which refers to that the reusing TPL does not contain the external code of the reused TPL injected in the software artifact during the linking stage. For instance, *yara* depends on *openssl* in the ground-truth data while *yara* only includes the headers (e.g. #include <openssl/evp.h>) instead of containing the source code of *openssl*. In such a black-box reuse case, the project repository does not contain code of reused TPLs. Thus, it does not cause feature duplication. Eventually, the resulting missing TPL dependencies do not affect the SCA results. In this way, we do not need to handle the black-box reuse following prior works [47, 48].

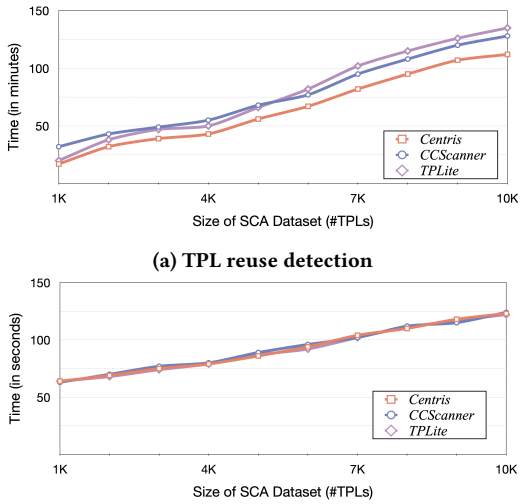
5.4 RQ4: Impact on SCA

We further evaluate the impact of TPL reuse detection by *TPLite* on the binary-level SCA. In particular, we first adopt the binary SCA engine BinaryAI [6] (introduced in Section 3.4.1). Then we remove the redundant features based on the TPL dependencies derived by *Centris*, *CCScanner*, and *TPLite* respectively and further construct their own TPL databases for performing SCA tasks. Note that we consider their performance comparison is fair because *Centris*, *CCScanner* and *TPLite* are equipped with the same binary SCA engine and code repository. Therefore, we derive that their performance gap is caused by the difference in their power to derive TPL dependencies and the resulting TPL databases.

Table 6: The SCA results

Tool	Metrics of Binary-level SCA		
	Precision(%)	Recall(%)	F1(%)
BinaryAI	25.76	56.34	35.35
Centris	56.12	53.28	54.66
CCScanner	60.32	56.19	58.18
TPLite	75.90	64.17	69.54
BDBA	72.46	58.55	64.77
Scantist	68.57	11.24	19.31
B2SFinder	21.08	57.62	30.87

Table 6 presents the results of component identification with different SCA tools. We can observe that in general, *TPLite* significantly outperforms all the other SCA tools. For instance, *TPLite* outperforms *Centris* (75.90% vs. 56.12% precision and 64.17% vs. 53.28% recall) and *CCScanner* (75.90% vs. 60.32% precision and 64.17% vs. 56.19% recall). Such results indicate the effectiveness of the TPL reuse detection of *TPLite* and its impact on SCA. Interestingly, *TPLite* dominates the performance of component identification among the state-of-the-art binary SCA tools. For instance, *TPLite* can even sightly outperform the widely-adopted and expensive BDBA (75.90% vs.72.46% precision, 64.17% vs. 58.55% recall, and 69.54% vs. 64.77% *F*1).



(a) TPL reuse detection

(b) SCA (online)

Figure 6: Time cost of *Centris*, *CCScanner*, and *TPLite*

We also measure the overhead of the TPL reuse detetcion and SCA. Figure 6a presents the time cost for deriving the TPL dependency in terms of different sizes of the TPL dataset. We can observe that in general, it costs *TPLite* more time over *Centris* and *CCScanner* in all the dataset sizes. For instance, it takes *Centris* and *TPLite* 112 and 135 minutes respectively to derive the TPL dependencies with the whole 10,241 TPLs. Considering that the TPL reuse detection is performed offline only once, such additional overhead is tolerable. Moreover, Figure 6b presents the time cost of the online component detection. Specifically, we measure the average time cost for detecting the components of our SCA testset with 128 binary files. We can find that there is no obvious difference of overhead between *Centris*, *CCScanner*, and *TPLite*.

6 THREATS TO VALIDITY

Threats to internal validity. The threat to internal validity mainly lies in the implementation of the studied subject. To reduce this threat, we reuse the source code of *Centris* to derive the TPL dependencies for the evaluation. Meanwhile, for the implementation of *TPLite*, the first two authors carefully review the code to ensure the correctness and consistency.

Threats to external validity. The threat to external validity mainly lies in the subjects and ground-truth dataset. To reduce this threat, we select the start-of-the-art technique *Centris* for studying its effectiveness of deriving TPL dependencies. We also compare *TPLite* with *Centris*, *CCScanner*, and three typical binary-level SCA tools

from both industry (BDBA and Scantist) and academia (B2SFinder). For the SCA dataset, we directly adopt the dataset from the original *Centris* paper with sufficient TPLs (10,241) for SCA tasks, similar as other SCA tools [13, 38, 78]. Considering the lack of publicly available ground-truth data for both TPL dependency and binary-level SCA, it also takes the authors excessive manual effort to carefully construct such two ground-truth datasets which are the largest datasets in their respective domain to our best knowledge. In particular, it takes over two months for the first two authors to carefully calibrate the ground-truth TPL dependencies. For uncertain or subjective cases, the first two authors perform more fine-grained analysis at the code level. If no agreement is reached, such cases are discarded to keep the consensus of deriving ground-truth data. For the ground-truth dataset of binary-level SCA, we adopt the largest number of projects (75) to construct a binary-level SCA dataset. Moreover, binary-level SCA can be even more labor-intensive compared with source-level SCA due to tremendous effort on binary analysis. Thus, we consider studying 75 projects for SCA in our paper to be rather sufficient.

Threats to construct validity. The threat to construct validity mainly lies in the adopted metrics in our study. To reduce this threat, we follow prior works [13, 68, 71, 78, 80] to evaluate multiple widely-used metrics, i.e., TP, FP, FN, *Precision*, *Recall* and *F1* score.

7 RELATED WORK

7.1 Software composition analysis

As many binary-analysis-based research works [23, 25, 36, 69, 70], many SCA techniques identify the third-party components for binary files (i.e., binary-level SCA) via alleviating feature duplication caused by internal code clones. Du et al. [13] propose the concept of internal code clones and design a hierarchical indexing scheme to identify such cases. OSLDetector [82] constructs an internal clone forest to reduce the impact of feature duplication between TPLs. B2SFinder [78] employs a weighted feature-matching algorithm to enhance the reliability of TPL detection and specific rules to recognize false positives caused by feature duplication. LibDX [61] proposes the logic feature block concept representing logical characteristics of code to deal with feature duplication. *Centris* [68] eliminates the redundant features with the TPL dependency based on function birth time and then performs the SCA component identification in the source code level. Some binary SCA techniques improve feature extraction to increase the accuracy of component identification. Xu et al. [74] propose ISRD which uses a multi-level birthmark model to address feature obfuscation. Tang et al. [62] adopt function contents as features and convert functions in binary to vector representation with neural network embedding. Modx [76] decomposes the program into fine-grained modules with program modularization techniques and extracts both semantic and syntactic features. Multiple SCA technologies are specifically designed to identify components of Android applications. WuKong [64] consists of two phases—coarse-grained detection by static semantic features and fine-grained clone detection algorithm. Libradar [40] extends WuKong with an improved clustering algorithm, and LibD [35] additionally adopts the feature hashing algorithm to enhance scalability. LibScout [10] omits unused code and employs normalized class to reduce the impact of code obfuscation. Zhang et al. [83] leverage modularized structures to formulate the SCA as a binary integer programming model. Zhan et al. [81] compare and study some TPL detection tools on four criteria and later propose ATVHUNTER [80], which uses the program control flow graph as a coarse-grained feature and the opcode as a fine-grained feature for two-stage detection. In this paper, we propose *TPLite* to derive the TPL dependencies and adapt it for binary-level SCA, where *TPLite* outperforms multiple existing state-of-the-art binary-level SCA tools [11, 58].

7.2 Code clone detection

Code clone detection evaluates the code similarity between software projects. SourcererCC [57] is a token-based code clone detector with different levels of granularity in source code level. Based on SourcererCC, Lopes et al. [38] build a duplicate code map called DéjàVu for the code repositories on Github. Semura et al. [59] propose CCFinderSW to provide a flexible way of supporting multilingual code detection. Zou et al. [86] utilize graph kernel to improve the efficiency of code clone detection based on the program dependency graph. Wu et al. [71] improve the scalability of semantic clone detection by analyzing the centrality of tokens in the control flow graph. CodeCMR [77] adopts DPCNN and GNN for feature extraction of source code and binary code respectively to perform function-level binary—source code matching. Movery [67] identifies the vulnerable code clones with internal and external modification of OSS. *Centris* [68] adopts local sensitive hash for function-level code clone detection. Gemini [73] applies Structure2vec to generate function-level embeddings and evaluates the binary code similarity by measuring the distance between embeddings. Kim et al. [30] formulate binary code clone detection as a graph alignment problem and propose XBA to utilize graph convolutional networks to learn the semantic features of binary code. Zeng et al. [79] perform an extensive study on the existing techniques of code clone detection based on pre-trained models and find that encoder-based models can in general outperform encoder-decoder-based models in code clone detection.

8 CONCLUSION

In this paper, we have extensively investigated the state-of-the-art SCA technique of the C/C++ ecosystem *Centris*. Specifically, we first find that the accuracy of TPL dependencies derived by *Centris* and the impact on the SCA component identification may not well generalize to our evaluation dataset. We further find that inaccurate function birth time and threshold-based recall can be the key factors that compromise the effectiveness of *Centris*. Inspired by our findings, we propose *TPLite* for TPL reuse detection and adapt the derived TPL dependencies to binary-level SCA. The evaluation results demonstrate that *TPLite* can significantly improve over *Centris* in terms of the accuracy of deriving TPL dependencies with 88.33% precision and 62.65% recall and the SCA component identification with 75.90% precision and 64.17% recall.

9 ACKNOWLEDGEMENT

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), and Shenzhen Peacock Plan (Grant No. KQTD2016112514355531).

REFERENCES

[1] 2010. E2fsprogs: Ext2/3/4 Filesystem Utilities. https://e2fsprogs.sourceforge.net.
[2] 2012. The DWARF Debugging Standard. https://dwarfstd.org.
[3] 2022. Apache Maven. https://maven.apache.org.
[4] 2022. GCC, the GNU Compiler Collection. https://gcc.gnu.org.
[5] 2022. The GNU C Library (glibc). https://www.gnu.org/software/libc.
[6] 2023. BinaryAI: binary file security analysis platform. https://binaryai.net.
[7] Filip Agneessens, Stephen P Borgatti, and Martin G Everett. 2017. Geodesic based centrality: Unifying the local and the global. *Social Networks* 49 (2017), 12–26.
[8] Sultan S Alqahtani, Ellis E Eghan, and Juergen Rilling. 2017. Recovering semantic traceability links between APIs and security vulnerabilities: An ontological modeling approach. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 80–91.
[9] Amrita Pathak. 2022. Software Composition Analysis (SCA): Everything You Need to Know in 2022. https://geekflare.com/software-composition-analysis.
[10] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 356–367.
[11] Black Duck Hub. 2022. Synopsys Black Duck Binary Analysis. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis/binary-analysis.html.
[12] Ying Ding, Erjia Yan, Arthur Frazho, and James Caverlee. 2009. PageRank for ranking authors in co-citation networks. *Journal of the American Society for Information Science and Technology* 60, 11 (2009), 2229–2243.
[13] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2169–2185.
[14] Elasticsearch. 2022. Path hierarchy tokenizer. https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-pathhierarchy-tokenizer.html.
[15] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 516–527.
[16] Muyue Feng, Weixuan Mao, Zimu Yuan, Yang Xiao, Gu Ban, Wei Wang, Shiyang Wang, Qian Tang, Jiahuan Xu, He Su, et al. 2019. Open-source license violations of binary software at large scale. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 564–568.
[17] Santo Fortunato, Marián Boguñá, Alessandro Flammini, and Filippo Menczer. 2006. Approximating PageRank from in-degree. In *International workshop on algorithms and models for the web-graph*. Springer, 59–71.
[18] Linton C Freeman. 1978. Centrality in social networks conceptual clarification. *Social networks* 1, 3 (1978), 215–239.
[19] Gene H Golub and Charles F Van Loan. 2013. *Matrix computations*. JHU press.
[20] GrammaTech. 2021. Binary Software Composition Analysis, Securing the Modern Software Stack. https://www.grammatech.com/binary-software-composition-analysis-sca.
[21] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 63–72.
[22] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2020. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability* 70, 1 (2020), 304–318.
[23] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627.
[24] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.
[25] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and Improving Hybrid Fuzzing. In *Proceedings of the 45th International Conference on Software Engineering*.
[26] Rie Johnson and Tong Zhang. 2017. Deep pyramid convolutional neural networks for text categorization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 562–570.
[27] Julie Peterson. 2021. Software Composition Analysis Explained. https://www.mend.io/resources/blog/software-composition-analysis.
[28] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.
[29] Dongjin Kim, Seong-je Cho, Sangchul Han, Minkyu Park, and Ilsun You. 2014. Open Source Software Detection using Function-level Static Software Birthmark. *J. Internet Serv. Inf. Secur.* 4, 4 (2014), 25–37.
[30] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 2022. Improving cross-platform binary analysis using representation learning via graph alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 151–163.
[31] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614.
[32] Amanda Lee and Travis Atkison. 2017. A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions. In *Proceedings of the SouthEast Conference*. 18–25.
[33] Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, and Dae-Kyoo Kim. 2022. Deep learning application on code clone detection: A review of current knowledge. *Journal of Systems and Software* 184 (2022), 111141.
[34] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 249–260.
[35] 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 *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 335–346.
[36] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 627–637.
[37] Nelly Litvak, Werner RW Scheinhardt, and Yana Volkovich. 2006. Probabilistic relation between in-degree and pagerank. In *International Workshop on Algorithms and Models for the Web-Graph*. Springer, 72–83.
[38] 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* 1, OOPSLA (2017), 1–28.
[39] Nan Ma, Jiancheng Guan, and Yi Zhao. 2008. Bringing PageRank to the citation analysis. *Information Processing & Management* 44, 2 (2008), 800–810.
[40] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th international conference on software engineering companion*. 653–656.
[41] Audris Mockus. 2007. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 7–7.
[42] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. 2019. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1026–1037.
[43] Mark E. J. Newman. 2018. Mathematics of networks. *Oxford Scholarship Online* (2018).
[44] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, 7–13.
[45] OWASP. 2022. OWASP Dependency-Check. https://owasp.org/www-project-dependency-check.
[46] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.
[47] Thiagarajan Ravichandran and Marcus A Rothenberger. 2003. Software reuse strategies and component markets. *Commun. ACM* 46, 8 (2003), 109–114.
[48] David Reid, Mahmoud Jahanshahi, and Audris Mockus. 2022. The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software. (2022).
[49] Github Repository. 2011. zlib-v0.8 file of deflation algorithm. https://github.com/madler/zlib/blob/v0.8/deflate.c.
[50] Github Repository. 2015. Independent JPEG Group's JPEG software. https://github.com/LuaDist/libjpeg.git.
[51] Github Repository. 2022. ClickHouse: open-source column-oriented database management system. https://github.com/ClickHouse/ClickHouse.
[52] Github Repository. 2022. SQLiteC++ file of sqlite3. https://github.com/SRombauts/SQLiteCpp/blob/master/sqlite3/sqlite3.c.
[53] Github Repository. 2022. SQLiteC++ with native C APIs of SQLite. https://github.com/SRombauts/SQLiteCpp/tree/master/sqlite3.
[54] Github Repository. 2022. ZLIB DATA COMPRESSION LIBRARY. https://github.com/madler/zlib.
[55] Github Repository. 2023. TPLite: TPL dependency scanner with origin detection and centrality analysis. https://github.com/Tricker-z/TPLite.
[56] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 117–128.
[57] 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 38th International Conference on Software Engineering*. 1157–1168.
[58] Scantist. 2022. Scantist, an open source management platform. https://scantist.io.
[59] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. 2017. CCFinderSW: Clone detection tool with flexible multilingual tokenization. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 654–659.
[60] Manuel Sojer and Joachim Henkel. 2010. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems* 11, 12 (2010), 868–901.

[61] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 104–115.

[62] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. LibDB: An Effective and Efficient Framework for Detecting Third-Party Libraries in Binaries. *arXiv preprint arXiv:2204.10232* (2022).

[63] 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 *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[64] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 71–82.

[65] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1267–1293.

[66] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[67] 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 *31st USENIX Security Symposium (USENIX Security 22)*. 3037–3053.

[68] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 860–872.

[69] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*. 1634–1645.

[70] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. 2022. Evaluating and improving neural program-smoothing-based fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. 847–858.

[71] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: software functional clone detection based on semantic tokens analysis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 821–833.

[72] Stefan Wuchty and Peter F Stadler. 2003. Centers of complex networks. *Journal of theoretical biology* 223, 1 (2003), 45–53.

[73] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 363–376.

[74] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-enabled software reuse detection based on a multi-level birthmark model. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 873–884.

[75] Erjia Yan and Ying Ding. 2011. Discovering author impact: A PageRank perspective. *Information processing & management* 47, 1 (2011), 125–134.

[76] 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*. 1393–1405.

[77] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883.

[78] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1038–1049.

[79] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 39–51.

[80] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. 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, 1695–1707.

[81] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. 2020. Automated third-party library detection for android applications: Are we there yet?. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 919–930.

[82] Dan Zhang, Ping Luo, Wei Tang, and Min Zhou. 2020. OSLDetector: identifying open-source libraries through binary analysis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1312–1315.

[83] 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*. 55–65.

[84] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[85] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 141–151.

[86] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 931–942.