LibDB: An Effective and Efficient Framework for Detecting Third-Party Libraries in Binaries

Wei Tang*
tang-w17@mails.tsinghua.edu.cn
School of Software, Tsinghua University
Beijing, China

Yanlin Wang†
yanlwang@microsoft.com
Microsoft Research
Beijing, China

Hongyu Zhang
hongyu.zhang@newcastle.edu.au
The University of Newcastle
Newcastle, Australia

Shi Han
shihan@microsoft.com
Microsoft Research
Beijing, China

Ping Luo
luop@mail.tsinghua.edu.cn
School of Software, Tsinghua University
Beijing, China

Dongmei Zhang
dongmeiz@microsoft.com
Microsoft Research
Beijing, China

ABSTRACT

Third-party libraries (TPLs) are reused frequently in software applications for reducing development cost. However, they could introduce security risks as well. Many TPL detection methods have been proposed to detect TPL reuse in Android bytecode or in source code. This paper focuses on detecting TPL reuse in binary code, which is a more challenging task. For a detection target in binary form, libraries may be compiled and linked to separate dynamic-link files or built into a fused binary that contains multiple libraries and project-specific code. This could result in fewer available code features and lower the effectiveness of feature engineering. In this paper, we propose a binary TPL reuse detection framework, LibDB, which can effectively and efficiently detect imported TPLs even in stripped and fused binaries. In addition to the basic and coarse-grained features (string literals and exported function names), LibDB utilizes function contents as a new type of feature. It embeds all functions in a binary file to low-dimensional representations with a trained neural network. It further adopts a function call graph-based comparison method to improve the accuracy of the detection. LibDB is able to support version identification of TPLs contained in the detection target, which is not considered by existing detection methods. To evaluate the performance of LibDB, we construct three datasets for binary-based TPL reuse detection. Our experimental results show that LibDB is more accurate and efficient than state-of-the-art tools on the binary TPL detection task and the version identification task. Our datasets and source code used in this work are anonymously available at **https://github.com/DeepSoftwareAnalytics/LibDB**.

*Work is done during internship at Microsoft Research Asia.
†Corresponding author.

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528442

CCS CONCEPTS

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

KEYWORDS

Third-party libraries, Static binary analysis, Clone detection

ACM Reference Format:
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. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524842.3528442

1 INTRODUCTION

Third-party libraries (TPLs) are important components of modern software systems. They are reused frequently during software development [28, 38]. Open-source repository platforms and package management systems are the major sources of third-party libraries. However, security issues of the third-party code continue to arise. Vulnerabilities in well-known third-party libraries, such as the *Heartbleed* [11] bug, could bring security threats to millions of devices. In addition, non-compliant reuse, which is a violation of legal software licenses, could lead to costly commercial disputes. Unfortunately, many developers do not pay sufficient attention to the code that is imported from third-party libraries.

To overcome the emerging threats caused by the reuse of third-party libraries, a great number of research works [10, 20–22, 36, 38] and commercial products [26, 29] have been proposed. They detect TPL reuse based on a database of libraries, identify their versions that may contain potential vulnerabilities, and manage license violation risks. Most of them handle detection targets that are in the forms of source code [20, 21, 30] or Java bytecode [36]. TPL detection for binaries compiled from C/C++ sources is also important since many projects (especially the ones that require high efficiency) are written in C/C++. However, only a few previous works studied this problem [37]. It is urgent to fill the gap of TPL detection in binaries. Compared to source code or bytecode forms, it is more difficult to detect TPLs for binaries as features such as variable names and function names are stripped after compilation.

Most existing approaches for TPL detection in binaries, including BAT [19], OSSPolice [10] and LibDX [27], use two types of basic features, i.e., string literals and exported function names. Only B2SFinder [13] proposes to use extra features including `integer constants`, `switch/else`, and `if/else` to detect commercial off-the-shelf (COTS) software that may be stripped of strings and exported function names. However, B2SFinder still mainly relies on the basic features for detection. Basic features are highly efficient to search by indexing as key-value pairs, however, they have many disadvantages. As the code content is ignored and basic features may exist cross libraries, they are not sufficiently unique to represent a library in a large-scale database. They are also too coarse-grained to identify specific versions, since basic features may not change cross versions. Besides, not all basic features are necessarily retained in binaries after compilation with different settings.

A detailed discussion of these limitations can be found in Section 2.

To address the limitations of basic features used in existing work, function-level features can be utilized, as they contain more fine-grained information that can represent functions and their call relationships in binaries. Specifically, in our work, we incorporate function vector features to improve the recall of TPL detection, because they can be fully/partially preserved in binaries. We also design a novel filter module to filter away wrongly reported libraries to further improve the precision of TPL detection.

In this paper, we propose a framework **LibDB** for accurately and efficiently detecting third-party libraries in binaries using additional function-level features and function call graphs (FCGs). We adopt a binary-to-binary matching method and build the local TPL feature database containing features extracted from TPLs in binary form. In addition to the basic features (i.e., string literals and exported function names), we embed all functions in a binary to low-dimensional representation vectors via a graph embedding network, connect all functions through dependency, and obtain FCGs to represent the binary. Against the local feature database, LibDB uses two channels (the basic feature channel and the function vector channel) to quickly detect candidate TPLs that may be contained in the detection target. Then, it compares each candidate to the detection target using the FCG representation. Candidates from two channels, after using FCG filter, are integrated as the final detection results. Unlike existing binary-oriented TPL detection methods, LibDB can further provide version identification of the TPLs contained in the detection target.

In summary, our contributions are as follows:

- We propose a novel framework LibDB that utilizes function contents to detect binary TPL reuse instead of relying on basic features only.
- We design a comparison algorithm based on FCGs to calculate the similarity between two binary files. The algorithm greatly improves the precision of both LibDB and existing approaches.
- We use fined-grained function features to better identify the versions of the reused libraries, which is not supported by previous binary TPL detection methods.
- We have conducted extensive experiments to illustrate the effectiveness and efficiency of LibDB compared to state-of-the-art approaches.

2 BACKGROUND AND MOTIVATION

2.1 Existing Work on TPL Detection for Binaries

Existing techniques [10, 13, 27] for third-party library detection in binaries essentially have a similar detection scheme. They construct two sets of features from a detection target (BIN) and a library (LIB_{UNIT}). The task to detect library usage is to calculate a similarity score by comparing these two feature sets ($\frac{|BIN \cap LIB_{UNIT}|}{|LIB_{UNIT}|}$). Existing tools divide a library into multiple comparison units and a set of features is extracted from each unit. A similarity score is assigned to each unit and a positive unit means a library reuse. Weighting functions may be used to assign different scores to features. BAT [19] treats the entire library repository as a unit and extract string literals as features. It considers longer strings to have more uniqueness and assigns them larger weights. OSSPolice [10] indexes a repository by its structural layout (i.e., a tree of files and directories) and takes each node (a directory) as a comparison unit to calculate a score. LibDX [27] takes each binary in a library as a unit. It only considers the features of continuous matching, which means that it may ignore a large number of features, resulting in a low recall rate. Both OSSPolice and LibDX use TF-IDF to calculate a weighted score.

A state-of-the-art tool, B2SFinder [13], separates a library to multiple groups by parsing compilation commands, where all sources are organized as compilation dependency graphs and each graph is a comparison unit for similarity calculation. It assigns larger scores to special strings such as web links, function names, etc. B2SFinder introduces additional features including integer constants, constants in conditional statements ("`switch/case`" and "`if/else`"), and constant arrays. In essence, B2SFinder still heavily depends on basic features. It considers the order of features to construct array data and the structure of conditional statements. Besides, new types of features require B2SFinder to adopt an one-to-one comparison method for all libraries in database, since they cannot be implemented by searching. It limits the capability of B2SFinder when the database grows. In summary, all existing techniques heavily rely on basic features (string literals and exported function names).

2.2 Limitations of Existing Work

Existing works that compare two sets of basic features have the following deficiencies when faced with fused binaries:

Low precision when detecting against a large-scale database. Some features are popular and included in a large number of libraries. As the size of the library database increases, the number of possible occurrences of each feature increases, therefore the uniqueness and effectiveness of basic features decrease. Since multiple libraries are compiled into a fused binary, more popular features will be accumulated. When detecting against a larger database, more false positives could be reported [30].

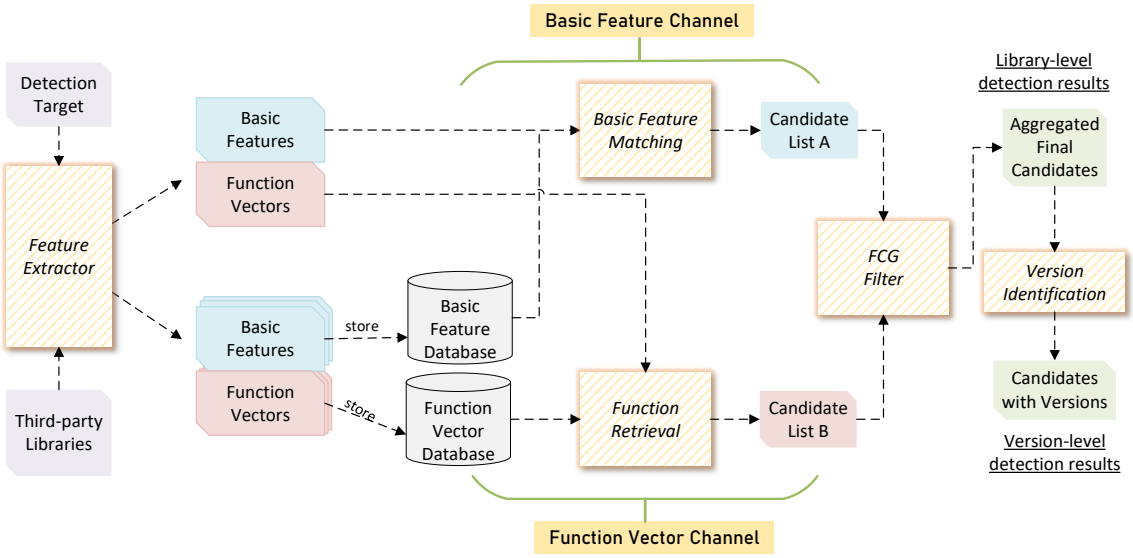


Figure 1: Overview of LibDB.

Low recall when few basic features exists. We observed that nearly all string literals are printout messages, such as log information including debug, warnings and errors. Developers can select the information they want and even remove all printouts by setting compilation macros. The compilation settings (i.e., "FLAGS") determine the code to be compiled in the compilation phase. For example, function "PNG_DEBUG" in LibPNG prints information when debugging. It contains message literals as arguments, however, they would not be compiled into the released files by undefining "PNG_DEBUG" in compilation parameters. In LibPNG, more than 95% of strings are printouts and most can be removed by macro settings. Besides, Quach et al. [25] shows that only 5% of libc functions are used on average by Ubuntu programs. Binary debloating techniques [1, 25] directly eliminate unneeded functions to improve software quality, meaning that only a small portion of functions from a source code file might be preserved. If the features are extracted at the file level, binary debloating would lead to a low similarity of the file, causing false negatives for approaches such as OSSPolice, B2SFinder, and LibDX, since they extract features and generate signatures at the file level.

Coarse-grained representation for version identification. TPLs evolve over time, with known vulnerabilities being patched and new (potentially vulnerable) code being added. Therefore, vulnerabilities are usually present in some successive versions. Each new version only makes minor changes to the previous one and different versions of a library are similar, especially for adjacent versions. The slightly changed code (e.g., a patch) has little effect on basic features. Therefore, previous approaches using coarse-grained model with basic features might not capture these subtle changes between versions. Only OSSPolice shows the results for version identification using unique strings to determine the best version, such as "1.2.6". However, the library information may not be stored in fused binaries. For example, the binary "libmain.so" in "Gomoku"[1] reuses the library "AGG", but there is no information about "AGG" in the version string. On the other hand, information about other libraries may provide noisy version strings. OSSPolice visioned that it can be empowered by fine-grained function-level features to improve the precision of version identification.

Our work addresses the limitations of existing work. To deal with the low precision problem, we design a FCG filter to filter out false positives. To deal with the low recall problem, in addition to the basic features, we identify function-level features to retrieve more results from the library database. Our fine-grained function features and FCG filter together can distinguish more correct versions than the coarse-grained basic features.

[1]https://f-droid.org/en/packages/com.traffar.gomoku/

3 PROPOSED APPROACH

3.1 Overview of LibDB

Figure 1 provides an overview of LibDB. LibDB contains five components: Feature Extractor (Section 3.2), Basic Feature Matching (Section 3.3), Function Retrieval (Section 3.4), Function-call Graph (FCG) Filter (Section 3.5), and Version Identification (Section 3.6). LibDB relies on the two feature databases, namely basic feature database and function vector database, which will be applied in two channels: basic feature channel and function vector channel, during detection. In the following, we briefly introduce each key component of LibDB to help readers acquire a high-level understanding of the workflow of LibDB before we step into its detailed designs:

Feature Extractor. Both TPLs and detection targets are in binary form and they share the feature extractor module. Binary inputs are disassembled and parsed by the feature extractor module. In the feature extractor, string literals and exported function names are extracted as basic features. Functions are embedded to representation vectors using neural networks and similar functions are mapped to representation vectors that have high cosine similarity scores.

Feature Databases. The output of the feature extractor contains two types of features: basic features and functions, which constitute the basic feature database and the function vector database, respectively. When building the local TPL database, each binary in a downloaded package is regarded as a comparison unit. Similarity will be calculated between each comparison unit and the detection target. In two feature databases, using inverted index, each feature is mapped to the comparison unit where the feature is extracted.

Channels. When matching features of detection targets, there are two channels corresponding to the basic features and function features. In the basic feature channel, basic features are applied to search directly in the basic feature database to obtain initial candidates rapidly. In the function vector channel, we retrieve the top-K nearest neighbors in the function vector database and locate related comparison units as initial candidates.

FCG Filter. Due to the lack of uniqueness of features against a large-scale database, there might be many false positives in the initial candidates output from the channels. We compare each candidate with the detection target using function-call graphs (FCG) in the FCG filter. FCG filter uses the same matching methods as the two channels but set different thresholds. The reasons and more descriptions are provided in Section 3.5. After using FCG to filter candidates, we get the final candidates. By locating libraries where the candidates are packaged, we can report library-level detection results.

Version Identification. LibDB further adopts a version ranking method to each positive library output by the FCG filter, then it pinpoints the rank #1 candidate as the version-level detection result.

3.2 Feature Extractor

Figure 2 illustrates the design of the feature extractor. Feature extractor extracts basic features and function vectors to construct fingerprints via three steps: disassembling binaries, extracting features, and function embedding.

3.2.1 Disassembling and Extracting. We build a Ghidra-based module to disassemble binaries, parse the assembly code, and extract features we need. Ghidra[2] is an open source reverse engineering framework developed by NSA (National Security Agency of the United States). It allows modular removal to accelerate the reverse process. After binaries are transformed to assembly code, we extract three parts: string literals, exported function names, and function features.

String literals and exported function names are basic features. String literals can be easily extracted from the data segment (`.rodata`, `.data` and `cstring`). We can obtain exported function names by reading their symbols in binaries. We further extract control flow graphs (CFGs) of functions as function features, which are used to generate function vectors in the subsequent embedding step.

3.2.2 Function embedding. Real-world applications have a wide variety of compilation conditions. Different compilation conditions may change binary code that are compiled from the same source code, resulting in difficulties in binary code matching.

[2]https://ghidra-sre.org

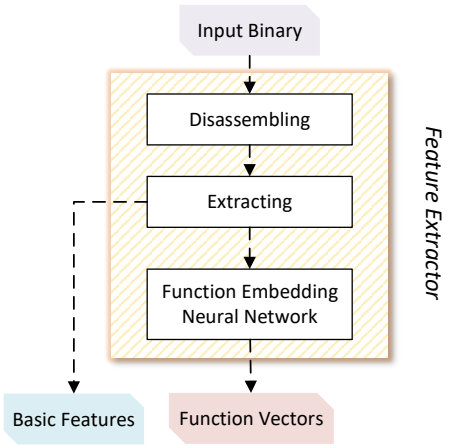


Figure 2: Feature extractor module.

Considering TPL reuse in real-world applications, we have summarized the following five main scenarios (5C scenarios) that could introduce changes to the binary code:

- *Cross-operating-system.* Open-source libraries may have different macro definitions depending on the target operating systems. For example, in the TPL `LibPNG`, there are conditional statements like "`#if defined(WIN32)`" and different "`Makefile`", which change the contents of compiled binaries on different operating systems. Besides, the format of binary files depends on the system they run on. Most common formats are PE (on Microsoft Windows), ELF (on Linux), and Mach-O (on MacOS).
- *Cross-architecture.* Content in compiled binaries varies with CPU architectures. For instance, there are several folders in `LibPNG` such as "`arm`", "`intel`", "`mips`" that contain unique optimized code for corresponding CPU architectures. Macro "`PNG_ARM_NEON_IMPLEMENTATION`" controls whether to use certain functions. It is also possible to set its value in compiler settings. In this case, the final constant features contained in binary files cannot be determined from the original code repository without compilation.
- *Cross-compiler.* In the open-source ecosystem, developers usually use GCC [17] as the main compiler. However, LLVM [18] has become more popular in recent years because of its excellent performance. Different compilers make differences when reusing TPLs.
- *Cross-compiler-version.* Compilers, like other software, evolve over time. After a library is released, developers may use another version of compiler to recompile their applications. Different versions of compilers also have impact on the binary code.
- *Cross-optimization-level.* Optimization is one of the challenges for binary code similarity detection [14]. Different optimization levels (O0-O3) have significant impact on the binary code. Developers can easily set the optimization level while compiling the reused libraries, resulting in differences in the binary.

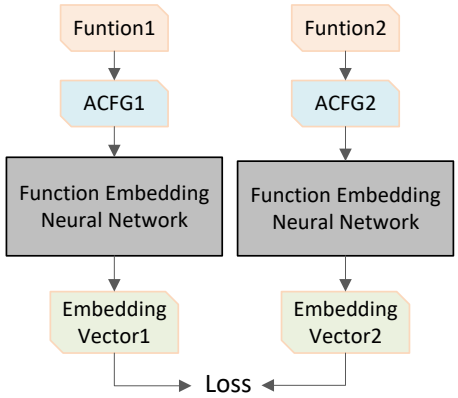


Figure 3: Embedding module.

Then, we construct a large dataset, the Cross-5C Dataset, to train the neural network model and calculate the similarity of binaries in 5C scenarios to cope with TPL reuses in real world. More details about the Cross-5C Dataset are provided in Section 4.1. From related works introduced in Section 7, we collect 13 open-source projects and compile each one using various compilation settings. For each scenario, we take several different values as possible conditions. For the cross-operating-system scenario, we consider three common platforms, Linux, MacOS and Windows. For the cross-architecture scenario, we select ARM and x86 as possible architectures. For the cross-compiler scenario, we choose two of the most well-known compilers, GCC and LLVM. They have many released versions up to now. We sample versions evenly through the whole version history. All versions are GCC: 4.8, 5.4, 6, 7.1, 8.1, 9, and LLVM: 3.5, 6, 12. For the cross-optimization-level scenario, we use all four levels (O0-O3) as available conditions. We obtain all possible combinations in 5C scenarios, finally we have 120 different kinds of compilation setting combinations.

Binary code similarity detection faces the challenge of 5C scenarios. However, it should be noted that LibDX is neither a system to find vulnerable function in binary code nor does it aim to report the similarity of two functions. We retrieve similar function pairs as features to find reused libraries using existing techniques for detecting similar function in binary code. A relatively large amount of false positive pairs would be retrieved against our large-scale database that contains 18.6 million functions. Accordingly, we designed a FCG-based filter to eliminate false pairs. More details are described in the Section 3.5.

For similar function retrieval in Channel 2, CFG based comparison is accurate but cannot handle an enormous amount of TPLs in database. Efficiently searching to retrieve similar functions is crucial, since we care about library-level detection instead of function-level or snippet-level detection. In recent years, researchers have used neural networks to map assembly code of functions to representation vectors [9, 16, 31]. Similar functions that are compiled from the same source code are mapped to vectors within a smaller distance than dissimilar functions. It is highly efficient to retrieve nearest neighbors as the detection result via vector similarity search.

Compared to other models such as ASM2VEC [9] that is mainly designed for a single assembly code language and not applicable for comparison across architectures, Gemini [31] is applicable to all 5C scenarios in the field of binary comparison. We use the Gemini model to embed each function to a vector, as depicted in Figure 3. Firstly, each function is transformed into an attributed control flow graph (ACFG) in which each block is represented using manually selected 7 types of features. Following the related work [31, 34], we ignore functions with fewer than 5 basic blocks. We then adopt Structure2vec [7] as the graph embedding network. It converts a graph (i.e., ACFG in LibDB) to a representation vector. A Siamese architecture [3] that contains one shared-parameter function embedding network (i.e., Structure2vec) is applied as shown in Figure 3. More technical details about Gemini can be found in its original paper [30].

We use the contrastive loss in Equation 1 for the optimization of the function embedding module. Optimizing Equation 1 leads similar function pairs (i.e., functions compiled from the same source code under different compilation conditions) to have cosine similarity close to 1, and dissimilar pairs (i.e., functions compiled from different source code) to have cosine similarity close to -1. In Equation 1, $Y_{i,j}$ represents the label of a function pair f_i and f_j and $\mathcal{F}$ is the set of functions. For each function, we randomly sample one similar function compiled from the same source code under different compilation conditions, and one dissimilar function compiled from different source code. If two functions are similar, their label is 1, otherwise -1. $S(f_i, f_j)$ is the cosine similarity of the two representation vectors of f_i and f_j. This way, we can efficiently search function representation vectors of detection targets against a large-scale database and retrieve similar functions according to the cosine similarity.

$$\mathcal{L} = \sum_{f_i, f_j \in \mathcal{F}} \frac{1}{2}\left(1+Y_{i,j}\right)\left(1-S(f_i, f_j)\right)^2 + \frac{1}{2}\left(1-Y_{i,j}\right)\left(1+S(f_i, f_j)\right)^2. \quad (1)$$

3.3 Basic Feature Matching

Basic feature channel searches basic features in the basic feature database and obtain a list of initial candidates (Candidate List A in Figure 1). For efficiency, we optimize the matching process using inverted index where string literals and exported function names are keys and comparison units the key exists in as values. We search basic features in database and get a list of corresponding comparison units as candidates. Each candidate has a set of common features that exist in both the comparison unit and detection target. We directly adopt the rules and thresholds used in B2SFinder [13] to filter candidates. A candidate is positive when it meets any of the following conditions: (1) the proportion of common strings against comparison unit is larger than 0.5; (2) the sum of weights is larger than 100 and the proportion of weights is larger than 0.1; (3) the number of common exported function names is larger than 20. More details can be found in B2SFinder repository.[3] Positive units are initial candidates and are sent to the FCG filter as inputs.

3.4 Function Retrieval

In addition to basic feature matching, in the function vector channel, LibDB can retrieve candidates using functions. This capability is very important especially when there are few or no basic features.

[3] https://github.com/1dayto0day/B2SFinder

We have collected a large-scale TPL database, namely the FedoraLib Database, which contains around 1,000 libraries with 25,000 versions from Fedora mirrors.[4] More details about the FedoraLib Database are provided in Section 4.1. All binaries as comparison units in TPLs are processed by the feature extractor introduced in Section 3.2. In total, 18.6 million functions are embedded to representation vectors and stored in the function vector database.

In order to detect the potential reuses of TPLs in detection targets, all functions of a detection target are extracted and converted to representation vectors using Structure2vec. Since all similar functions are mapped to closer vectors, we are able to search for similar functions using the nearest neighbor algorithm. We retrieve the top-K most similar functions for each function in the detection target and locate corresponding comparison units as initial candidates (Candidate List B in Figure 1). A larger K means that more functions are retrieved as similar functions. Further, more comparison units are obtained. On the contrary, a smaller K means fewer candidates but they are more likely to be similar to the function in the detection target. We implement the function retrieval module based on an efficient vector searching engine Milvus.[5] It supports using GPU and creates an index to speed up the searching phase. We choose the inner product distance between two embeddings as the similarity metric to search, since inner product can be easily converted to cosine similarity, which we use in the function embedding module (Section 3.2.2).

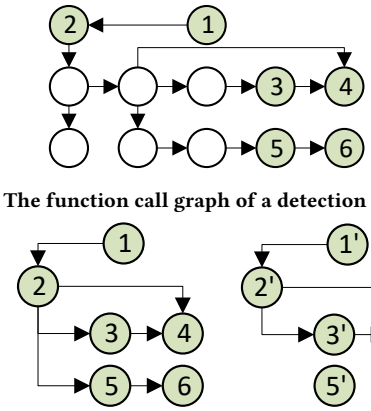
If the target has M functions, we can get $M * K$ similar functions and locate comparison units where these functions are extracted from. We sort all comparison units according to the number of similar functions they contain. Top-200 units will be passed to the subsequent FCG filter.

3.5 FCG Filter

The candidate list obtained from basic feature matching and function retrieval may contain lots of false positives. In order to filter out them, we propose a novel FCG based algorithm to compare binaries with the detection target and filter out those with low similarity. The algorithm contains two steps: function pairing and FCG comparison.

3.5.1 Function pairing. Given the FCG of a candidate and the FCG of the detection target, function pairing determines the node mapping between them. For each function f_i in the detection target, we match it with the most similar function $f_i^{'}$ that has the largest cosine similarity to f_i among all functions in the candidate. We set a threshold that requires the cosine similarity is larger than 0.8. It is a reasonable value according to our experiments. More details are discussed in Section 5.3.1. Finally, we can obtain a set of similar function pairs $\langle f, f^{'} \rangle$ indicating the similar node pairs between the candidate and the target. Note that for the function vector channel, the function pairing step can be omitted as similar function pairs are already retrieved in the function retrieval module.

3.5.2 FCG comparison. After the set of similar function pairs is obtained, we identify correct candidates using FCGs. Figure 4a illustrates the FCG of a detection target where a node represents a function and a directed edge represents function call dependency, with the caller pointing to the callee. Green nodes correspond to functions that exist in the similar function pairs and white nodes correspond to functions that do not exist in similar function pairs. We take the matched nodes (`Node 1-6`) as anchor nodes, since they are key points to show that two FCGs are similar. To reduce the size of graphs, we skip unmatched nodes (i.e., white nodes) to build a mini-FCG using anchor nodes only. In an FCG, unmatched nodes will be removed and dangling edges are connected from upstream nodes to downstream nodes. As shown in Figure 4, the corresponding mini-FCG of Figure 4a is shown on the left of Figure 4b. We remove all white nodes. If a matched node transitively depends on another, for example, node2 transitively depends on node3, we add a new edge from node2 to node3 in the mini-FCG. The original dependency relationships are retained. The mini-FCG formed by similar functions in the library is shown on the right of Figure 4b.



(a) The function call graph of a detection target.

(b) FCG comparison between library and detection target. Left is the mini-FCG for the target. Right is the mini-FCG for the library.

Figure 4: The FCG comparison method.

Our FCG comparison method calculates the number of common edges as the similarity score. Due to the existence of false similar function pairs, we leverage the function dependency to identify true function reuse. In mini-FCGs of the candidate and the target, two edges are common edges when their caller function pair and callee function pair exist in similar function pairs. It can be seen in Figure 4b that, the edge from node 1 to node 2 is a common edge, since both node 1 and node 2 form similar function pairs. Node 5 in the left graph is in the set of similar function pairs, but the corresponding matched function (node 5 apostrophe) does not call the function on node 6 apostrophe and node 2 apostrophe does not call node 5 apostrophe, either. Therefore, the similar function pair (node 5 and node 5 apostrophe) cannot contribute to the similarity score.

In the basic feature channel, we set a threshold on the number of common edges. Empirically, a candidate which has less than 3 common edges is recognized as a false candidate. Through comparative analysis, we find that similar function pairs in the function vector channel are of higher quality but low quantity compared to the basic feature channel. The reason is that similar function pairs in the function vector channel are retrieved against TPL function vector database. The top-K most similar functions from 18.6 million

[4] https://admin.fedoraproject.org/mirrormanager/
[5] https://milvus.io

functions are of high similarity. However, a comparison unit in our TPL database has 90 functions on average. In the basic feature channel, similar function pairs are matched by function pairing from a target to a unit. Not all pairs in the basic feature channel can surely exist in the top-K most similar functions in the function vector channel.

3.6 Library Report and Version Identification

Our TPL database contains four levels of information: library, version, comparison unit, and feature. A library has multiple versions and a version package may contain multiple binaries. Each of them is a comparison unit to be compared to the target file. FCG filter outputs merged final candidates from the basic feature channel and the function vector channel. LibDB reports all related libraries as library-level detection results where the final candidates exist. Based on those candidates, LibDB can further conduct version identification. It sums all similarity scores of candidates that exist in the same version package. The sum of scores is assigned to each version package. For a positive library, we report the version with the largest score as the version-level result.

4 EXPERIMENTAL DESIGN

4.1 Datasets

We collect 3 datasets for evaluation:

TPLBinary Dataset: This dataset is used as ground truth for binary TPL detection and version identification. There is no public test data for TPL detection in binaries. The TPLBinary Dataset is established with the reuse relationships of binary software. We contacted authors of related works and got the raw data of OSSPolice [10] and the test data of LibDX [27]. However, B2SFinder [13] can not extract features from the test data of LibDX successfully. Besides, most of the test data in LibDX are separate dynamic link libraries not fused binaries. The raw data of OSSPolice contains a set of Android applications from F-Droid.[6] To find the library reuse relationships in fused binaries, we first check their source repositories and get possible reuses like OSSPolice. Then, we analyze compilation commands in repositories and obtain the source code dependency like B2SFinder. Finally, we use the reverse engineering tool to disassemble the binary, and find specific evidence of reuse like LibDX. Since all applications are open-source, lots of information like string literals and function names are kept in binaries and can be used as evidence. Reuse relationships without evidence are ignored. Totally, we have 24 Android applications, 112 binaries and 172 reuse relationships.

Cross-5C Dataset: The Cross-5C Dataset contains manually compiled libraries for training and evaluating function comparison model. We collected 13 projects from related work [14, 31] covering areas of image processing, database, encryption, etc. They are manually compiled under the 5C scenarios illustrated in Section 3.2.

Each function is compiled to multiple instances under as many compilation conditions as possible. Sometimes, the compiled code may be the same in different compilation scenarios. In this case, we only keep one instance instead of multiple identical instances. Two binary functions are similar when they come from the same source code, even though they may have been changed in different compilation settings. We do not modify any other settings in compilation commands except for the 5C scenarios mentioned in Section 3.2. That means function inlining appears in our compilation processes. We use functions in 10 projects as train and validation sets, and the remaining 3 libraries as test set.

FedoraLib Database: We collect a large-scale TPL database from a Fedora mirror manager with all historical versions. All mirrors contain 400 thousand packages. It greatly exceeds the size of database used by existing tools. Based on the FedoraLib Database, we evaluate the recall of function retrieval module at large-scale, accuracy of library reuse detection and version identification. We select libraries that can be found in NVD[7] and obtain 997 libraries with about 25,000 versions and more than 200,000 comparison units. It is ten times larger than the TPL database used by B2SFinder [13]. In total, 18.6 million functions are extracted from the FedoraLib Database.

4.2 Compared Methods

We compare LibDB with several related methods and their variants:
Base: Base is a vanilla baseline we designed, which uses the simplest matching method with basic features. It obtains a list of candidates based on common features that exist in both the comparison unit and detection target. For each candidate, it will be reported as positive if: 1) the number of common features is larger than 15, or 2) the proportion of common features against candidate is greater than 0.2. Since the approach is simple and straightforward, Base method is very efficient.

LibDX [27]: LibDX is a tool for binary-to-binary comparison. It organizes constants as feature blocks to reduce false positives and take file names and requirement information as supplementary features. LibDX also relies on basic features.

B2SFinder [13]: B2SFinder is a state-of-the-art tool in the area of binary-source TPL detection. It uses seven types of features (`String`, `Export`, `Switch/case`, `If/else`, `String Array`, `Integer Array`, `Enum Array`) separately for detection. Results of each kind of features are simply aggregated together. For binary-to-binary comparison, only the first four types are applicable.

5 EVALUATION

In our evaluation, we aim to answer the following research questions:

5.1 RQ1: How does LibDB perform in detecting TPLs in binaries?

We evaluate the TPL detection performance of LibDB on the TPLBinary Dataset by comparing it to the recent three baselines introduced in Section 4.2: Base, LibDX, and B2SFinder. We adopt Precision (P), Recall (R) and F1 score as evaluation metrics:

$$P = \frac{\#(correct\ libraries)}{\#(reported\ libraries)}, \quad R = \frac{\#(correct\ libraries)}{\#(libraries\ in\ target)}$$

$$F1 = \frac{2 * P * R}{P + R}$$

[6] https://f-droid.org

[7] https://nvd.nist.gov

Table 1: Performance comparison with baselines on the TPLBinary Dataset.

Method	P (%)	R (%)	F1	VD	VP (%)
Base	29.1	86.5	0.44	1.32	40.2
LibDX	88.4	30.8	0.46	/	/
B2SFinder	16.3	94.2	0.28	/	/
LibDB	55.3	93.6	**0.70**	**0.58**	**60.4**

Experimental results are shown in Table 1. From the results in Table 1, we can see that LibDB outperforms all the baselines in terms of F1 score. Base method and LibDX have comparable F1 scores, but Base shows better recall and LibDX has the highest precision. B2SFinder achieves the highest recall rate of 94.2%.

After inspecting the results and comparing LibDB with baselines, we find that rules of basic features are similar among all methods and most false positives are retrieved due to common basic features across libraries. For LibDB, in basic feature channel before FCG filter, some similar functions exist in targets and candidates, especially standard library functions. They make our FCG filter recognize the false positive as a potential library reuse. Function vector channel uses contents of functions as features. It has the ability to report reused libraries that cannot reach the thresholds in rules with only few basic features. LibDX adopts feature block to eliminate false positives, however it significantly reduces the recall rate. B2SFinder has the lowest F1 score and precision, even though it has the highest recall. The new features (i.e., constants in "switch/case" and "if/else" statements) introduced by B2SFinder bring a slight improvement in recall, but more false positives. Besides, extra computation cost is caused by new features (Section 5.5).

5.2 RQ2: How does LibDB perform on the version identification task?

We use two metrics to evaluate version identification: VP (Version Precision) and VD (Version Distance). The VP metric calculates the rate of exact versions in all true positive libraries. It is used in previous works [10, 38]. We design an additional metric VD to measure the *distance* between the reported version and the correct version. Given two versions, $a1.b1.c1$, $a2.b2.c2$ in the format of Major.Minor.Patch, their distance is calculated by $10 * |a1 - a2| + b * |b1 - b2| + c * 0.1 * |c1 - c2|$. The reason for using VD is that vulnerabilities are usually associated with a series of consecutive versions. For example, `CVE-2019-7317`[8] exists in `LibPNG 1.6.x` ahead of `1.6.37`. That means, if the reported version is not exactly the same as the ground truth, it still makes sense to report a version close to the true version. Besides, reporting a close version compensates for the cases when the true version is not in the TPL database.

For related work, LibDX and B2SFinder do not support version identification. Only OSSPolice proposes a simple version identification method, which uses unique features of one version across all versions of a library. We integrate this version identification method with Base as a baseline. From Table 1, we can see that LibDB improves the version precision by 20% and has a version distance of 0.58, which are much better than the Base combined with the unique feature based version identification method. The experimental results demonstrate that our fine-grained function features can effectively improve the performance of version identification in terms of both VD and VP.

[8]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7317

Table 2: Subset1: function pairs with different operating systems. Subset2: different architectures. Subset3: different compilers. Subset4: different versions of the same compiler. Subset5: different optimization levels. Subset6: all five conditions are different, such as Linux-ARM-GCC-5.4-O3 and Win-x86-GCC-8.1-O0.

Subset	True Pairs	Recall (%)
Subset1 (OS)	65,120	98.63
Subset2 (ARM | x86)	30,979	96.99
Subset3 (GCC | LLVM)	69,783	91.29
Subset4 (Comp. Version)	64,077	98.17
Subset5 (O0 | O3)	106,420	70.73
Subset6 (Max Diff)	5,690	65.38

5.3 RQ3: How effective is the function retrieval component?

5.3.1 Accuracy of function embedding. The performance of function retrieval module depends on the function embedding network. We first evaluate the accuracy of the function embedding network and determine the threshold for cosine similarity between two function embeddings. The threshold is set to 0.8 according to the validation dataset in the Cross-5C Dataset.

If a compiled instance changes a lot compared to another instance of the same function, they would be judged as dissimilar functions. Since we retrieve similar functions as features to find reused libraries, it is crucial to recall true similar pairs. We are concerned about whether LibDB can tell that two functions are similar when the functions change due to compilation. In order to investigate the impact of different compilation conditions on code changes, we generate 8 subsets depending on the change conditions from test set. All datasets are described in Table 2. True pairs indicate all similar function pairs in that subset. Recall represents how many similar function pairs can be correctly judged when performing function comparison. We use the trained model to evaluate the impact of each condition on the recall of the similar functions, since recall is more important than precision in the retrieval phase.

In Table 2, We can see that compilation conditions, operating system, and compiler version, have less impact on binary code change. Even if code is changed by these two conditions, 98%-99% similar function pairs could still be correctly identified as similar pairs. Different architectures cause slightly greater impact on binary code and different compilers change the binary code more. The optimization level O0-O3 has the most significant impact on code changes and only 70% similar function pairs can be recalled. As most conditions are different, The subset6 (maximum differences) only achieves a recall of 65%.

5.3.2 Similar function retrieval. Because we retrieve similar functions as clues to find potential reused libraries, it is critical to recall enough similar functions between detection target and reused

Table 3: Recall of function retrieval on the Cross-5C Dataset against the FedoraLib Database.

Subset	top-10 (%)	top-20 (%)	top-50 (%)	top-100 (%)
Subset1-1 (Linux | Win)	59.13	63.43	67.66	71.07
Subset1-2 (Linux | Mac)	73.60	75.82	78.22	79.77
Subset1-3 (Win | Mac)	55.66	58.25	62.35	66.20
Subset2 (ARM | x86)	18.76	24.66	29.65	33.61
Subset3 (GCC | LLVM)	23.86	27.21	31.96	35.52
Subset4 (Comp. Version)	70.53	74.28	77.49	79.72
Subset5-1 (O0 | O3)	2.29	2.68	3.63	4.56
Subset5-2 (O2 | O0)	3.41	4.21	5.55	6.84
Subset5-3 (O2 | O1)	33.56	36.32	39.97	42.46
Subset5-4 (O2 | O3)	65.31	66.38	67.76	68.76
Subset6 (Max Diff)	0.54	0.76	1.09	1.48

library. Next, we evaluate the performance of function retrieval against the function vector database. Starting from similar function pairs of each subset, $\langle f_i, f_i^{'} \rangle$, we add f_i into function vector database and search $f_i^{'}$ against the database. We consider f_i to be recalled if it is in top-K nearest neighbors and calculate the recall rate for each subset. In Section 5.3.1, we know that different optimization levels (O0 | O3) result in quite different binary code. It may make the embedding model embed similar functions to vectors that have low cosine similarities. Thus, based on the Cross-5C Dataset, we generate more subsets with different optimization settings. Similarly, we further divide the subset1 into smaller subsets to explore the differences among Windows, Linux, and MacOS settings.

The recall values of different top-K on all subsets are shown in Table 3. In general, recall has dropped a lot compared to similar function identification in Section 5.3.1. As K increases, the growth of recall becomes slower. A K larger than 100 would result in little increase in recall, but retrieve more false positives. In Table 3, subset1, subset4 and subset5-4 (O2 | O3) have higher recall values than the other subsets. Subset2, subset3 and subset5-3 (O2 | O1) have lower recall values. The subset5-1 (O0 | O3) and subset5-1 (O2 | O0) have the lowest recall values among all compilation conditions. Most optimization settings are disabled at O0 and the higher the optimization level is, the more optimizations are enabled.[9] Many compilation optimizations like `-finline-functions-called-once` from O1, `-finline-functions` from O2 and `-fpeel-loops` at O3 can change the structure of CFG a lot and generate a different graph. It makes the function embedding network embed the function to a vector that has a small cosine similarity with similar functions compiled under other optimization levels. We find that closer levels, such as (O2 | O1) and (O2, O3), have higher recall values. As the most extreme case, the subset6 has the lowest recall value among all subsets. Even for the top-100 candidates, less than 2% of similar functions are recalled. The problem of low recall on subset5-1, subset5-2, and subset6 is left to our future work.

5.3.3 Effectiveness analysis of function retrieval module. We analyze the effectiveness of the function retrieval module in improving the recall when no enough basic features are available. As described in Section 2, most of string literals can be deleted by setting macros. Different levels of printouts (e.g., debug, warning, error) can be selected by developers. Therefore, different numbers of strings are kept in binaries. Exported function names, debloating techniques, and partial clones can delete unused functions in the library, leading to a reduction of the number of function name features. In order to simulate these scenarios, we randomly select basic features and keep 4 levels of proportions, 0%, 25%, 50% and 100%, and then conduct evaluations. Detection results on data with different proportions of basic features are provided in Table 4. It shows that our function retrieval module can bring different improvements to all methods. Improvements are more significant when there are fewer basic features. In extreme cases, for binaries without basic features, the function retrieval module can achieve a precision of 96.5% and a recall of 32.2% for Base. It is much better than B2SFinder that uses `switch/else` and `if/else`. We could find that when there is a sufficient quantity of basic features, libraries can be easily recalled. In the meantime, more false positives are reported, leading to a lower precision. From Table 4, we can see that the proportion of 50% achieves the highest F1 than other proportions. That is because random selection eliminates some popular features, and the false positives caused by the popular features are reduced. However, it does not mean that we can use only half of the basic features in detection. It cannot be assumed that there are always enough unique features in detection targets.

5.4 RQ4: How effective is the FCG filter component?

We evaluate the effectiveness of FCG filter by adding it to baselines (i.e., variants Base$_{+fcg}$ and B2SFinder$_{+fcg}$) and removing it from LibDB (i.e., variants LibDB$_{-fcg}$, LibDB$_{base-fcg}$ and LibDB$_{fr-fcg}$). LibDB$_{base}$ is LibDB with only basic feature channel. LibDB$_{fr}$ is LibDB with only function vector channel. As shown in Table 5, the use of FCG filter can significantly improve the precision of all evaluated approaches and their variants. Meanwhile, the recall rates generally remain the same. The recall rates of the baseline LibDB$_{fr-fcg}$ and Base dropped slightly because the FCG filter may filter out a small number of true positives. However, the F1 scores of all methods generally get improved. This study confirms the effectiveness of our FCG filter component for both LibDB and other methods. Also, we find that common functions, especially standard library functions, cause many false positives. In our future work, we will explore the IDA FLIRT technique,[10] which would be beneficial to recognize standard library functions generated by the supported compilers.

5.5 RQ5: How efficient is LibDB?

We use five servers to extract features. Each server has 8-core Intel i7 CPU and 64GB memory. For model training and evaluating LibDB, the system is deployed on a server with a 20-core Intel E5 CPU, 256GB memory and 4 Nvidia Tesla V100 GPUs. Total time consumption contains two parts: extracting features of TPLs from the FedoraLib Database and the detection time.

It is one-time cost to process FedoraLib Database and extract features of TPLs, although it is time-consuming and resource-consuming. This process took us about 43 hours on five servers.

[9] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[10] https://hex-rays.com/products/ida/tech/flirt/

Table 4: Impacts of the function retrieval module on data with different proportions of basic features. "+fr" indicates that our function retrieval module is used, and "-fr" denotes that our function retrieval module is removed.

String Percentage	0%			25%			50%			100%		
	P (%)	R (%)	F1	P (%)	R (%)	F1	P (%)	R (%)	F1	P (%)	R (%)	F1
Base	/	0	/	86.0	45.6	0.60	74.1	77.1	0.76	29.1	86.5	0.44
Base_{+fr}	96.5	32.2	0.48	84.9	57.9	0.69	73.6	83.0	0.78	29.2	88.3	0.44
B2SFinder	6.5	17.5	0.10	19.6	71.9	0.31	26.4	86.0	0.40	16.3	94.2	0.28
B2SFinder_{+fr}	12.8	35.7	0.19	19.8	73.1	0.31	26.4	86.5	0.41	16.3	94.2	0.28
LibDB_{-fr}	/	0	/	63.7	65.5	0.65	69.8	82.5	0.76	55.7	91.8	0.69
LibDB	96.5	32.2	0.48	65.5	72.5	0.69	70.3	86.0	0.77	55.3	93.6	0.70

Table 5: Impacts of the FCG filter component.

Method	P (%)	R (%)	F1
LibDB_{-fcg}	15.3	93.6	0.26
LibDB	55.3	93.6	0.70
$\text{LibDB}_{base-fcg}$	25.9	91.8	0.40
LibDB_{base}	55.0	91.8	0.69
LibDB_{fr-fcg}	12.2	38.0	0.18
LibDB_{fr}	96.5	32.2	0.48
Base	29.1	86.5	0.43
Base_{+fcg}	43.0	79	0.56
B2SFinder	16.3	94.2	0.28
B2SFinder_{+fcg}	41.4	94.2	0.58

More than 95% time spent on disassembling binary code, since reverse engineering is a time-consuming task, especially for large binaries. We set a timeout threshold to kill the task thread, if it cannot be finished within 30 minutes. According to our records, less than 1% binaries in FedoraLib Database encountered timeout. As for detection, Base method is most efficient and only takes 5 minutes. It uses the simplest matching method and can search features based on an inverted scheme. Compared to Base method, LibDX first performs the same feature searching task. Then it uses feature block method and takes 25 minutes to filter out false positives. Basic features in B2SFinder supports accelerating the matching process using an inverted index. But `switch/case` and `if/else` can only be applied by one-to-one comparison against FedoraLib Database. In total, B2SFinder requires 70 hours during the detection process. For LibDB, its detection time consists of three parts: extracting features from detection targets costs 35 minutes, basic feature channel costs 7.5 hours, and function vector channel costs 18 minutes. Basic feature channel requires more time, since there is a larger number of initial candidates to be passed into FCG filter component. FCG filter is the most time-consuming component in LibDB. In Basic feature channel, basic features matching can be finished within several minutes, like Base method. The rest of time (more than 7 hours) is devoted to FCG filter. In function vector channel, function retrieval is working on Milvus. It takes 1.1 seconds to retrieve the top-100 most similar function per 1,000 queries.

6 THREATS TO VALIDITY

There are three main threats to validity: **Binary obfuscation.**

Some libraries could utilize code obfuscation techniques to protect their intellectual property. The obfuscated code often has poor readability and maintainability, and could significantly change the features LibDB uses. Currently, LibDB does not consider code obfuscation. We will handle the commonly-used code obfuscation techniques (such as string encryption and CFG-flattening) in our future work.

New library version. LibDB detects TPLs based on a local database containing features of TPLs that are built in advance. It can only report possible reuses in the database. To mitigate this threat, we build a large-scale database containing 997 libraries with 25,000 version. The database could also be regularly updated with new libraries and new versions. Still, if the detection target is not in our database, LibDB will only report its closest version.

Establishment of TPL database. Currently, there is no public TPL database for binaries. Therefore, we have to build a new database from Fedora mirrors. We also search the library names in the NVD database. If there are vulnerabilities related to the named library, we will keep the library in our TPL database. This is a relatively simple method to identify which libraries are related to each of the vulnerabilities in NVD. Automated identification of libraries from vulnerability data is a challenging problem [5] and will be an important future work.

7 RELATED WORK

Third-party library detection. Most existing works for TPL detection are designed for Java libraries in Android applications [37]. Java is a cross-platform language. Java bytecode compiled in different compilation scenarios remain unchanged. In addition to string literals, there are more features such as class dependency [38, 39], CFG centroid [10], semantic features from program dependency graphs [6], and opcode of CFG [35] can be used to build fingerprints. Bytecode has different formats compared to binary code and these features is not effective for binaries. Therefore, techniques for Java library detection can not be directly applied for TPL detection in binaries. As discussed in Section 2, there are also several work on TPL detection for binaries, including BAT [19], OSSPolic [10], LibDX [27], and B2SFinder [13].

Binary code clone detection. Existing binary code clone detection approaches mostly focus on function-level [4, 32] or basic-block-level [23, 24] comparison such as vulnerable function detection and bug search. These approaches are based on CFG comparison or instruction analysis which are very slow and not scalable when analyzing a large-scale code base. DiscovRE [12] employs a pre-filter based on numeric features to retrieve a small set of candidates. Genius [15] is the first method to embed CFGs and apply function vector search to detect similar binary code. Recently, the development of graph embedding techniques using neural network has inspired researchers to embed functions to vectors [16, 31, 33] for binary code clone detection. It can efficiently retrieve similar functions from a large-scale database.

Similarity by composition. Inspired by image similarity [2], Yaniv et al. [8] illustrate the concept "binary code similarity by composition", which means that a binary file can be composed of parts of other binary files. It is similar to the concept of "fused binary" in our study. The difference is that Yaniv et al. consider instruction-level snippets and calculate the similarity of basic-block slices, while we focus on the similarity calculation at the file level.

8 CONCLUSION

In this paper, we propose a framework, LibDB, for binary-oriented TPL detection. LibDB uses contents of functions as features. It embeds functions in such a way that the similar functions have a higher cosine similarity score than the dissimilar functions. We obtain two lists of initial candidates via the basic feature channel and the function vector channel. Candidates from two channels, after FCG filtering, are combined as the final detection results. LibDB is able to further provide version identification of TPLs contained in the detection target. Extensive experiments have demonstrated the effectiveness and efficiency of LibDB.

Our datasets and source code are available at **https://github.com/DeepSoftwareAnalytics/LibDB**.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback.

REFERENCES

[1] Ioannis Agadakos, Di Jin, David Williams-King, V. P. Kemerlis, and G. Portokalidis. 2019. Nibbler: debloating binary shared libraries. *Proceedings of the 35th Annual Computer Security Applications Conference* (2019).

[2] Oren Boiman and Michal Irani. 2006. Similarity by Composition. In *Advances in Neural Information Processing Systems (NIPS 2006)*.

[3] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1993. Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems* 6 (1993), 737–744.

[4] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: cross-architecture cross-OS binary search. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016).

[5] Yang Chen, Andrew E Santosa, Asankhaya Sharma, and David Lo. 2020. Automated identification of libraries from vulnerability data. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 90–99.

[6] J. Crussell, Clint Gibler, and Hao Chen. 2015. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. *IEEE Transactions on Mobile Computing* 14 (2015), 2007–2019.

[7] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*. PMLR, 2702–2711.

[8] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.

[9] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489.

[10] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *ACM Sigsac Conference*. 2169–2185.

[11] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*. 475–488.

[12] Sebastian Eschweiler, Khaled Yakdan, and E. Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*.

[13] Muyue Feng, Zimu Yuan, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, Aihua Piao, Jingling Xue, and Wei Huo. 2019. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1038–1049. https://doi.org/10.1109/ASE.2019.00100

[14] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-Based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 480–491. https://doi.org/10.1145/2976749.2978370

[15] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).

[16] J. Gao, X. Yang, Ying Fu, Yu Jiang, and Jia-Guang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2018), 896–899.

[17] GNU. 2021. GCC, the GNU Compiler Collection. Retrieved Sep 9, 2021 from https://gcc.gnu.org/

[18] GNU. 2021. The LLVM Compiler Infrastructure. Retrieved May 9, 2021 from https://llvm.org/

[19] Armijn Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. 2011. Finding software license violations through binary code clone detection. In *MSR '11*.

[20] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. *2012 IEEE Symposium on Security and Privacy* (2012), 48–62.

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

[22] 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.

[23] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014).

[24] Jannik Pewny, Behrad Garmany, R. Gawlik, C. Rossow, and T. Holz. 2015. Cross-Architecture Bug Search in Binary Executables. *2015 IEEE Symposium on Security and Privacy* (2015), 709–724.

[25] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[26] synopsys. 2021. Black Duck Software Composition Analysis. Retrieved May 9, 2021 from https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html

[27] 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 *2019 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[28] Haoyu Wang, Yao Guo, Ziang Ma, and X. Chen. 2015. WuKong: a scalable and accurate two-phase approach to Android app clone detection. *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015).

[29] whitesource. 2021. Automate your open source security and compliance workflows. Retrieved May 9, 2021 from https://www.whitesourcesoftware.com/

[30] 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.

[31] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and D. Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity

Detection. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017).

[32] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Y. Liu. 2019. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Transactions on Software Engineering* 45 (2019), 1125–1149.

[33] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In *AAAI*.

[34] 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. In *NeurIPS*.

[35] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), 1695–1707.

[36] 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.

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

[38] Jiexin Zhang, A. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019).

[39] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, S. Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), 141–152.