applied sciences

MDPI

Article

BBDetector: A Precise and Scalable Third-Party Library Detection in Binary Executables with Fine-Grained Function-Level Features

Xiaoya Zhu [1], Junfeng Wang [2,*], Zhiyang Fang [3], Xiaokang Yin [1] and Shengli Liu [1,*]

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China
[2] College of Computer Science, Sichuan University, Chengdu 610065, China
[3] School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China
* Correspondence: wangjf@scu.edu.cn (J.W.); mr_shengliliu@163.com (S.L.)

Abstract: Third-party library (TPL) reuse may introduce vulnerable or malicious code and expose the software, which exposes them to potential risks. Thus, it is essential to identify third-party dependencies and take immediate corrective action to fix critical vulnerabilities when a damaged reusable component is found or reported. However, most of the existing methods only rely on syntactic features, which results in low recognition accuracy and significantly discounts the detection performance by obfuscation techniques. In addition, a few semantic-based approaches face the efficiency problem. To resolve these problems, we propose and implement a more precise and scalable TPL detection method BBDetector . In addition to syntactic features, we consider the rich function-level semantic features and form a feature vector for each function. Moreover, we design a scalable function vector similarity search method to identify anchor functions and the candidate libraries, based upon which we carry out TPL detection. The experiment results demonstrate that BBDetector outperforms B2SFinder and ModX in terms of effectiveness, efficiency, and obfuscation-resilient capability. For the nix binaries, the F1-score of BBDetector is 1.11% and 11.21% higher than that of ModX and B2SFinder, respectively. Moreover, for the Ubuntu binaries, the F1-score of BBDetector is 1.32% and 14.93% is higher than that of ModX and B2SFinder, respectively. And in terms of efficiency, the detection time of BBDetector is only 30.02% of ModX. Besides, for the obfuscation-resilient capability, BBDetector is much stronger than B2SFinder. BBDetector achieves a F1-score of 71%, slightly lower than the F1-score of 77% achieved with the non-obfuscated binary programs. However, B2SFinder only achieves an F1-score of 28%, much lower than that of 67% achieved with the non-obfuscated binary programs.

Keywords: third-party library detection; syntactic features; function-level semantic features; function vector similarity search; anchor functions

Citation: Zhu, X.; Wang, J.; Fang, Z.; Yin, X.; Liu, S. BBDetector: A Precise and Scalable Third-Party Library Detection in Binary Executables with Fine-Grained Function-Level Features. *Appl. Sci.* **2023**, *13*, 413. https://doi.org/10.3390/app13010413

Academic Editors: Arcangelo Castiglione and Christos Bouras

Received: 7 December 2022
Revised: 21 December 2022
Accepted: 23 December 2022
Published: 28 December 2022

1. Introduction

Third-party libraries (TPLs) have been widely adopted to boost production efficiency and reduce artificial costs during software development.

However, TPL reuse may introduce vulnerable or malicious code and expose the software, which reuses them to potential risks. According to the report of Sonatype [1], open source vulnerabilities are prevalent among popular projects, 29% of which contain at least one known vulnerability. For example, the remote code execution vulnerability (CVE-2021-30116) was found in the Kaseya VSA. The REvil hacker group used this vulnerability to deploy the ransomware in local clients and launch a large-scale extortion attack against related companies [2]. In addition, attackers no longer wait for the public disclosure of vulnerabilities to exploit but actively inject new vulnerabilities into open source projects that support the global supply chain. For example, the popular NPM UA-Parser-JS library was hijacked and planted with malicious code by hackers, resulting in numerous Windows

and Linux devices infected with cryptocurrency mining software and password-stealing trojans [3]. In addition, this library is part of the supply chain for many well-known software and enterprises. It will therefore be used in user-facing end-use applications, which will spread to millions of computers. Generally, researchers have no access to the program source code containing the TPLs and are limited to analyzing the binary executables. Therefore, there is an urgent need to discover the TLPs in binary executables.

Researchers have recently proposed different methods to detect TPLs [4–11]. For example, OSSPolice [5] extracts both string literals and exported function names to detect TPLs reused by the target binary program, and B2SFinder [7] extracts seven kinds of code features to carry out TPL detection. ModX [11] extracts string literals, function call graph (FCG), and functions accessing the same data to perform TPL detection.

However, we identify three major limitations of the existing TPL detection. First, at present, the third-party detection methods mainly rely on syntactic features (function names [5,6], strings [4], constants [7] etc.). On the one hand, some methods need function name information, which can be stripped away in binaries. On the other hand, with the increase of the third-party component database, the syntactic characters (e.g., strings and constants) increase, and the uniqueness and validity of the characters decrease, resulting in low accuracy of recognition. Moreover, obfuscation techniques (such as string encryption) can discount the detection performance for syntactic-based methods. Second, in addition to syntactic features, a few studies (such as ModX [11]) focus on the function call graph (FCG) and some functions' semantic features (accessing the same data) to detect third-party libraries, but the comparison of FCG causes a lot of time consumption and the functions selected for detection are based on syntactic features, causing performance degradation when the syntactic features are obfuscated. Third, most existing TPL detection methods [4–7,9,10] focus on the source code form of libraries. However, many third-party libraries have difficulty downloading their source code from the Internet (closed source libraries). In addition, some code in the source code will not be compiled into the binary executable file, such as the "text" part of the source code; thus, features extracted from the uncompiled source code will cause feature interference.

To tackle the above problems, we propose and implement a more precise and scalable TPL detection method, named BBDetector , which focuses on the binary form of third-party libraries. To solve the problem of TPL detection considering only syntactic features, we also consider the rich semantic features of functions contained in the libraries for more accurate matching. Moreover, in terms of efficiency, we do not consider the expensive FCG matching but consider all the function semantics. However, the function semantic matching may introduce the problem of large time consumption with the expansion of third-party libraries. To deal with fast and scalable function semantic matching, we first learn the rich semantic information of functions using the presentation learning model [12] and form a feature vector for each function. Then, we design a scalable function vector similarity search method, which can support fast vector similarity retrieval when facing a large-scale function vector database. Moreover, we use the function vector similarity search method to identify function pairs (i.e., anchor functions) that are highly similar and likely to have the same functionalities and the candidate libraries. Then, we carry out TPL detection based on these anchor functions and the candidate libraries.

To demonstrate the efficacy of BBDetector in TPL detection, we collect two types of datasets: Dataset I and Dataset II. Dataset I is a binary target program dataset with known TPL reused relationships, which includes programs built by the nix [13] package manager and a set of manually building binaries on Ubuntu 20.04. Dataset II is a TPL dataset, which is used to build third-party library databases. In addition, we use effectiveness (including precision, recall, and F1-score), efficiency, and code obfuscation-resilient capability as the evaluation metrics. We compare BBDetector with the state-of-the-art binary-to-binary TPL detection method ModX [11] and binary-to-source TPL detection method B2SFinder [7]. The experiment results show that BBDetector outperforms B2SFinder and ModX in terms of effectiveness, efficiency, and obfuscation-resilient capability. For the nix binaries, the

F1-score of BBDetector is 1.11% and 11.21% higher than that of ModX and B2SFinder, respectively. In addition, for the Ubuntu binaries, the F1-score of BBDetector is 1.32% and 14.93% higher than that of ModX and B2SFinder, respectively. Moreover, in terms of efficiency, the detection time of BBDetector is only 30.02% of ModX. Furthermore, for the obfuscation-resilient capability, BBDetector is much stronger than B2SFinder. BBDetector achieves an F1-score of 71%, slightly lower than that of 77% achieved with the non-obfuscated binary programs. However, B2SFinder only achieves an F1-score of 28%, much lower than that of 67% achieved with the non-obfuscated binary programs.

In summary, our contributions are summarized as follows: Current TPL detection methods mainly rely on syntactic features, which results in low recognition accuracy and significantly discounted detection performance by obfuscation techniques. A few semantic-based approaches (such as FCG graph) face the efficiency problem. We propose a more precise and scalable TPL detection method to resolve these problems. To improve accuracy, in addition to syntactic features, we also consider the rich function-level semantic features and form a feature vector for each function. Moreover, in terms of efficiency, we design a scalable function vector similarity search method to identify anchor functions and the candidate libraries. Then, we perform TPL detection based on these anchor functions and the candidate libraries. We implement and evaluate BBDetector with binary target programs with the known TPL reused relationships and TPL dataset. Moreover, the experiment results show that BBDetector outperforms B2SFinder and ModX in terms of three metrics: effectiveness, efficiency, and obfuscation-resilient capability.

2. Related Work

In this section, we introduce the related works of TPL detection. According to the forms of detection target programs and third-party libraries, current TPL detection methods can be divided into three categories: source-to-source comparison, binary-to-source comparison, and binary-to-binary comparison.

2.1. Source-To-Source Comparison

Source-to-source comparison methods detect reused libraries when the source code of detection target programs and third-party libraries are both available. Source-to-source comparison methods often perform syntactic analysis on the source code and carry out a matching based on tokens [14] and abstract syntax trees (AST) [15].

Baxter et al. [15] propose a tool that transforms source code into abstract syntax trees (AST) and detects copy-paste by finding identical subtrees. CCFinder [14] represents a source code as a token sequence and applies the rule-based transformation to the sequence. Then, it uses a suffix-tree matching algorithm to make a token-by-token comparison. Centris [10] is capable of detecting modified and nested libraries reuse by segmenting the source code and detecting the reuse of a unique part of the libraries only. For scalability, it adopts a redundancy elimination technique to reduce space complexity and stores hashed functions to accelerate the search.

2.2. Binary-To-Source Comparison

In the case of binary-to-source comparison detection, the target programs are binary files and the third-party libraries are source code files.

Binary Analysis Tool (BAT) [4] extracts the strings from both sources and binaries. It considers longer strings to have more uniqueness and assigns them larger weights. OSSPolice [5] extracts syntactical features, such as strings and exported function names when matching binaries against library sources. It introduces a novel hierarchical indexing scheme to achieve both high scalability and accuracy. BCFinder [6] selects string constants as features to generate a profile for sources and binaries. When extracting strings from sources, it ignores strings that are required by programming syntax such as “%s” , “%d”, which are required by `printf`. B2SMatcher [9] extracts five kinds of code features: string literals, exported function names, constants in assignments, constant parameters in function

calls, and in/out-degrees of a function on call graph. It categorizes these features into program-level features (string literals, exported function names) and function-level features (constants in assignments, constant parameters in function calls, and in/out-degrees of a function on call graph) and proposes a two-stage version identification approach based on the two levels of code features. B2SFinder [7] is a state-of-the-art method for binary-to-source TPL detection. It extracts seven kinds of code features (String literal, Switch/Case, If/else, Export function name, String Array, Integer Array, Enum Array) to make a separate analysis and detection.

2.3. Binary-To-Binary Comparison

When the target programs and third-party libraries are in binary format, we need to do a binary-to-binary comparison.

As far as we know, BinShape [16] is the first method to detection reused libraries in binary files. It produces a robust signature for each library function based on heterogeneous features covering CFGs, instruction-level characteristics, statistical characteristics, and function-call graphs. And then it desin a novel data structure to store these signatures and facilitate efficient matching against the target function. LibDX [8] extracts contents in the read-only DATA segment of binaries, which are mainly composed of string constants and sometimes contain function names and import libraries. Moreover, it takes a fuzzy filename and requires information as supplementary features. OSLDetector [17] detects third-party libraries for multi-platform software in binaries. It chooses constant strings extracted from `.rodata` section and `.data` section. And it takes measures such as filtering features and building an internal clone forest for all libraries to eliminate internal strings clones. LibDB [18] can effectively and efficiently detect imported TPLs even in stripped and fused binaries. It extracts basic and coarse-grained features and function contents features. It further adopts a function call graph-based comparison method to improve the accuracy of the detection. ModX [11] is a state-of-the-art method for binary-to-binary TPL detection. It extracts syntactic features (strings literals, constant numbers) and semantic features (function call graph, functions accessing common data) to make TPL detection.

In this paper, we design a binary-to-binary comparison to detect third-party libraries. The considerations that we collect libraries in binary format as our local feature database for two reasons. On the one hand, it is hard for many libraries downloaded from the Internet to look for their source code. Especially some libraries, such as Google-Mobile-Ads-SDK [19], are closed source libraries [8]. On the other hand, by analyzing binary files, we can get much higher-quality features to build a database. Open-source repositories have a great deal of code duplication, which is usually stored in the source code but not compiled in the binary file. Such as OpenCV and LibPNG, the source code of LibPNG is included in the OpenCV source repository but it is not compiled into OpenCV library binaries. The source code in `text` is not compiled into released binaries. Features extracted from these uncompiled source codes will increase the number of features, potentially leading to a low matching ratio and causing false negatives.

3. Methodology

3.1. Overview

In this paper, we design our TPL detection method named BBDetector , shown in Figure 1, and we present the details as follows. As illustrated in Figure 1, there are three components:

- **Feature extraction.** When given a target binary or a third-party library, we disassemble it using IDA Pro [20] and extract corresponding features, including string literal features and function semantic features, to represent it. We extract these features by writing an IDA python script.
- **Feature database construction.** In this process, we construct the local TPL feature database. The output of the TPL feature extraction process contains two types of features: string literal features and function semantic features, which are used to constitute the string feature database and the function feature database, respectively.

- **TPL detection.** BBDetector performs TPL detection by checking whether a target binary could be matched to any library in the local TPLs by calculating their similarity score.

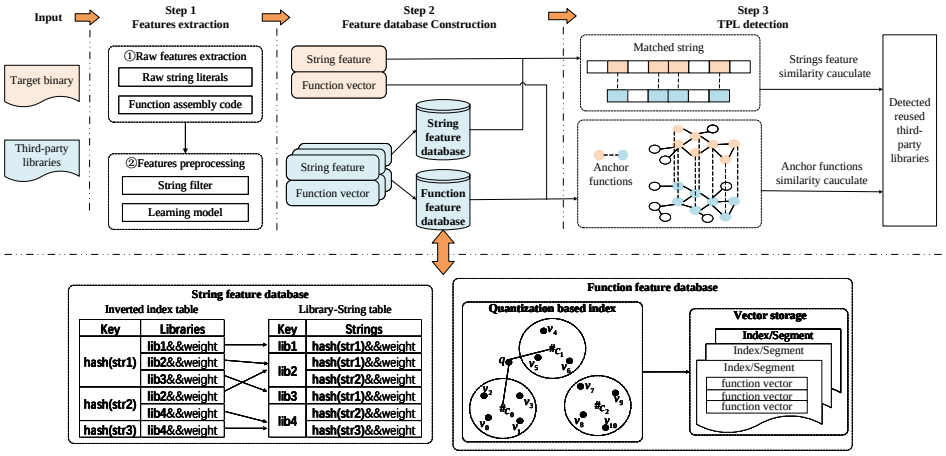

Figure 1. Overview of BBDetector.

3.2. Feature Extraction

In this process, we extract features from the target binaries and the third-party libraries, including two stages: raw feature extraction and feature generation.

3.2.1. Raw Feature Extraction

We first use IDA Pro to disassemble binary executables and identify function boundaries in this stage. Then, we extract raw string literals and function assembly code. String literals are basic syntactic features easily extracted from the data segment (.data, .rodata) and usually have unique values that can be easily distinguished. Suppose two functions from two binaries have identical string literals. In that case, they have a high chance of being the same function, and the two corresponding binaries maybe have a reuse relationship. We write the IDAPython script to extract raw string literals. In addition to syntactic string features, we extract function-level semantic features to represent the binaries more accurately. Given a binary file, we extract a list of assembly functions, their basic blocks, and control flow graphs.

3.2.2. Feature Generation

We preprocess the extracted raw features in this stage, including raw string feature filtering and function vector generation. String duplication is common among large-scale libraries, reduces detection efficiency, and results in possible false positives, so it is essential to filter some common and frequently occurring strings. OSLDetector [17] analyzes the distribution of string features length. It finds that the number of strings less than five in length (such as %s, #, etc.) accounted for 32% of their total strings, and they do not make a robust matching in the detecting process. Therefore, we select filter strings with a length of less than five, which can save storage space and improve detection efficiency. For function-level semantic feature, to facilitate subsequent searching and matching, we use an assembly code representation learning model Asm2Vec [12] to produce a numeric vector for each assembly function, as depicted in Figure 2. We use the assembly code of functions as training data without prior knowledge, and this model produces a vector for each function after training. The training process is as shown in the lower part of Figure 2: when given the current function and neighbor instructions, this model uses the current function's vector and the context provided by the neighborhood instructions to predict the current instruction and maximizes the log probability of seeing current instruction. When given a new target function that is not from training data, we use this representation learning model to predict its vector.

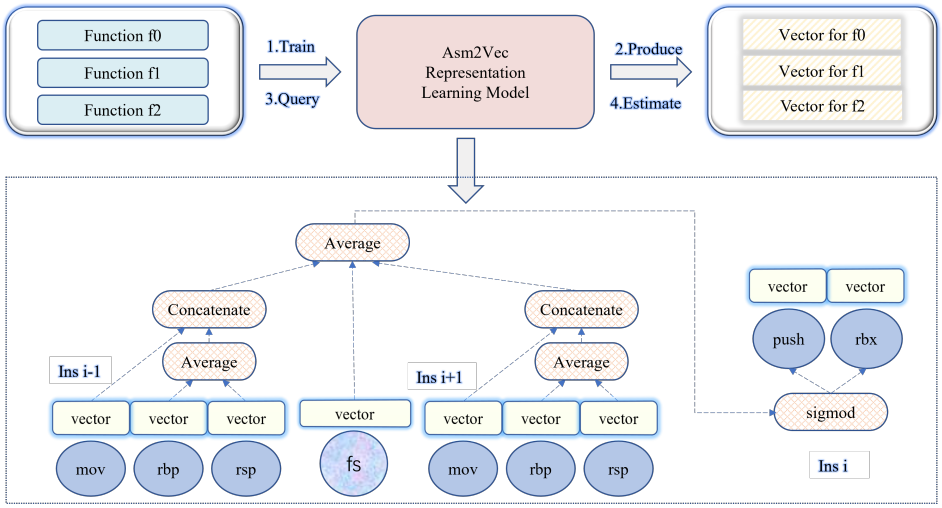

Figure 2. Function embedding module.

3.3. Feature Database Construction

After extracting the features of the third-party libraries, we need to store their features as a local TPL feature database for the subsequent TPL detection of the target binaries. The storage form of features is closely related to the detection efficiency. In this section, we introduce the building process of our TPL feature database, shown in Figure 1, including two components: string feature database and function feature database.

3.3.1. String Feature Database

Due to the challenge of string feature duplication, we assign a unique weight to each string feature to reduce the values of frequently-used strings. In BBDetector , we use `TF-IDF` algorithm, which is a popular algorithm to reflect the importance of words in a document in the corpus, to calculate the weight of each string literal. The weight w of a string i to a library j is shown as Equation (1).

$$w_{ij} = TF - IDF(i) = TF(i) * IDF(i) = \sum_{k} \frac{n_{ij}}{n_{kj}} * \frac{|L|}{|\{l \in L : i \in l\}|} \quad (1)$$

where n_{ij} is the frequency of string i in library j; $|L|$ is the total number of libraries and $|\{l \in L : i \in l\}|$ is the number of libraries which contain the string i.

After calculating the weight for each string, we build index tables to store the relationship between string features and libraries. We adopt hashing method to generate hash values for each string ($hash(str1)$) and use the same column width to store strings which can avoid space waste. Moreover, we adopt an inverted index technique [5,7] to create an inverted mapping of string literal features to the third-party libraries to speed up string literal feature searching and matching. The specific information is as shown in Figure 1, which includes two index tables: the left one uses the inverted index to store the information about the library where the string resides, and the right one stores the information about strings that the library contains to retrieve strings through a library name easily.

3.3.2. Function Feature Database

Next, we describe how to build a function vector database for scalable similarity search and matching. When detecting TPLs based on function features, we need to calculate the function vector similarity scores between the target binary and the libraries. In other words, we need to find similar matching function vectors for the target binary from a large database of TPL function vectors, which can be considered a function search problem. To search a similar vector facing a massive vector dataset efficiently, we can organize vector data by

building indexes on them. After the dataset is indexed, queries can be routed to clusters, or subsets of data, that are most likely to contain vectors similar to an input query. Then, we perform a one-to-one similarity matching in these clusters and reduce similarity comparison in unrelated clusters. Our method uses a quantization-based index [21,22], which has a high-speed query capability. We first perform vector quantization for each function vector and build a quantization-based index. Specifically, we apply a quantizer q to map a function vector v to a codeword $q(v)$ chosen from a codebook C, which is constructed by using K-means clustering algorithm and where each codeword is the clustering centroid and $q(v)$ is the closest centroid to function vector v. After building the index, we store these vectors in the segments, each of which stores an index. Figure 1 shows an example of 10 vectors (v_0 to v_9) of three clusters with centroids c_0, c_1 and c_2. And $q(v_0)$, $q(v_1)$, $q(v_2)$ and $q(v_3)$ is c_0 and so on.

3.4. TPL Detetction

We have selected two kinds of code features shown in Figure 1. For different code features, we design different matching algorithms.

3.4.1. TPL Detection Based on String Features

In TPL detection based on string features, we calculate the matching score between the target binary and TPLs. The matching score is calculated based on the matched weighted feature instances. As the number of matched feature instances increases, the matching score increases. Therefore, we set a threshold for the matching score, the same as B2SFinder. Specifically, when the matching score satisfies any of the following conditions: (1) the ratio of the matched strings between the target binary and a library is greater than 0.5, as shown in Equation (2); (2) the sum weight of matched string instances is larger than 100, and the ratio of this weight is larger than 0.1, as shown in Equation (3). We consider the target binary having a reuse relationship with the corresponding library.

$$\frac{|S_B \cap S_L|}{|S_L|} > 0.5 \tag{2}$$

$$W(S_B \cap S_L) > 100 \;\&\; \frac{W(S_B \cap S_L)}{W(S_L)} > 0.1 \tag{3}$$

where S_B represents the string instances including in the target binary B; S_L represents the string instances including in the library L to be compared; $|S_B \cap S_L|$ represents the number of common string instances including both B and L; $W(S_B \cap S_L)$ represents the weight sum of the common string instances; and $W(S_L)$ represents the weight sum of string instances including in L.

The specific algorithm of TPL detection based on string features is shown in Algorithm 1. The algorithm's input is the string features S_B of the target binary and the string index tables. Firstly, we obtain the candidate reused TPLs based on the strings included in the target binary and the string inverted index table (line 2–4). When a string of the target binary is in the index table, we take libraries that include this string as candidate reused TPL. Then, for each candidate library, we calculate the matching score and compare it with the predefined threshold to determine whether it is reused by the target binary (line 5–20). We obtain the matched string instances between the target binary and the candidate library (line 8–11). Then, we calculate the weight sum of matched string instances (line 12–16). Last, we calculate the matching scores to determine whether the values are greater than the threshold and whether this library is reused by the target binary (line 17–19).

Algorithm 1 TPL detection based on string features

```
function TPL_DETECTION_BASED_ON_STRING_FEATURES(S_B)
    candidate_matched_libs ← []
    for s in S_B do
        candidate_matched_libs.append(GET_LIBS(s))
    //feature match
    matched_libs ← []
    for lib in candidate_matched_libs do
        S_L ← GET_STRINGS(lib)
     // match string instances
        S_m ← []
        for s_b × s_l ∈ S_B × S_L do
            if s_b = s_l then
                S_m.append(s_l)
     //calculate weights
        for s in S_L do
            W ← GET_LIB_STRING_WEIGHT(s)
            W_L ← W_L+W
            if s in S_m then
                W_m ← W_m+W
     //calculate matching scores
        score ← W_m/W_L
        if score ≥ threshold then
            matched_libs.append(lib)
    return matched_libs
```

3.4.2. TPL Detection Based on Function Features

In TPL detection based on function semantic features, we measure semantic similarity between functions in the target binary B and the third-party libraries $L = L_1, L_2, \ldots, L_k$, where k is the number of libraries in the TPL database. Suppose we need to determine whether the target binary B reuses a library $L_i \in L$, since B and L_i consist of multiple functions, the matched function similarity score will be aggregated to measure the similarity score S_{BL_i} between B and L_i. To calculate S_{BL_i}, a problem we need to deal with is how to choose the matched functions (called anchor functions), which have a high similarity score and may have a high probability of reuse relationships, from B and L_i to compare with.

Due to B, which may reuse multiple libraries $L_i, L_j, \ldots$ and partially reuse L_i, many functions may be contained in only one. Therefore, we cannot directly conduct a one-to-many search to find the similar function $lf_j \in L_i$ of a function $bf_j \in B$ (i.e., anchor functions) which may be only contained in B and vice versa. Thus, we design a fast and scalable method to obtain the anchor functions for each library.

In the acquisition of anchor functions stage, we select the anchor functions. For each function $f \in B$, we find vectors that are most similar from the function vector database to obtain candidate reused TPL list and the anchor functions, respectively, i.e., $L_i : [(bf_1, lf_1), (bf_2, lf_2), \ldots, (bf_j, lf_j)]$, where bf_j is a function in the target binary B; lf_j is a function in the candidate matching library L_i; and (bf_j, lf_j) represent the matching anchor functions and we take their similarity scores as part of the similarity scores of B and L_i. Therefore, the problem of anchor function acquisition is expressed as the problem of vector similarity search. Specifically, it can be expressed as, given a collection of m function vectors in $V_B : q_1, q_2, \ldots, q_m$ and a collection of $n * k$ function vectors in $V_L : v_1, v_2, \ldots, v_{n*k}$, how to quickly find the top-k similar function vectors for each q_i? In BBDetector , we search similar function vectors based on quantization-based indexes.

Search processing of a query q_i over quantization-based indexes takes two steps: (1) Find the closest n_{probe} clusters based on the distance between q_i and the centroid of each clusters. (2) Search within each of the n_{probe} relevant clusters to find similar vectors. For

example, as shown in the lower right corner of Figure 1, assuming n_{probe} is 2, the closet clusters centroids of q_i are c_0 and c_1, and then we scan vectors $v_0, v_1, v_2, v_3, v_4, v_5, v_6$ to find similar vectors.

In the stage of the feature match, we measure the similarity score S_{BL_i} based on these anchor functions. We measure their similarity by computing their function vectors' cosine similarity scores for the anchor functions, which are often used to compare the similarity degree of the binary function vectors [12,23,24]. We obtain the similarity score S_{BL_i} by calculating the sum of the cosine similarity of these anchor function vectors and averaging it. When the similarity score is greater than the threshold (we set it to 0.8, which is determined empirically), we consider the target binary having a reuse relationship with the corresponding library. If the detected matching libraries have multiple versions in the TPL database, we use the version with the highest similarity score as the final matching version.

The specific algorithm of TPL detection based on function features is shown in Algorithm 2. The algorithm's inputs are the function vectors V_B of the target binary and the function vector index table. Firstly, we obtain the candidate reused TPLs and anchor functions based on the vectors including the target binary (line 2–9). For each vector $q \in V_B$ in the target binary B, we find its top-k similar function vectors (line 5). For each similar function vector lib_v, we consider the library including it as a candidate library (line 7–8) and take (q, lib_v) as an anchor function (line 9). Then, we calculate the matching score between B and a candidate library to judge whether the vector similarity values are greater than the threshold and determine whether this library is reused by the target binary (line 10–18). Then, we determine whether there are multiple versions of a library in the matching result. We first judge the number of a library (regardless of the specific version) exists in the matching result. If the number is greater than one (line 21), we select the version of this library with the highest similarity score (line 22) and take this version as the final matching result (line 23). If there is only one version in the matching result, we simply take the version as the final matching result (line 24–25). The similarity score between B and a candidate library is the average of the sum of the similarity scores of the anchor functions they contain.

Algorithm 2 TPL detection based on function features

```
1:  function TPL_DETECTION_BASED_ON_FUNCTION_FEATURES(V_B)
        // get candidate libraries and the anchor functions
2:      candidate_matched_libs ← []
3:      lib_anchor_functions ← {}
4:      for q in V_B do
5:          similar_lib_vs ← GET_TOP_K_SIMILAR_VECTOR(q, k)
6:          for lib_v in similar_lib_vs do
7:              lib ← GET_LIBS(lib_v)
8:              candidate_matched_libs.append(lib)
9:              lib_anchor_functions[lib].append([q,lib_v])
        //feature match
10:     matched_libs ← []
11:     for lib in candidate_matched_libs do
12:         anchor_functions ← GET_LIB_ANCHOR_FUNCTIONS(lib)
13:         for anchor_function in anchor_functions do
14:             similarity ← CALCULATE_SIMILARITY(anchor_function)
15:             similarity_scores ← similarity_scores + similarity
16:         ave_similarity_scores ← AVERAGE(similarity_scores)
17:         if ave_similarity_scores ≥ threshold then
18:             matched_libs.append(lib)
19:             function_matched_libs ← []
20:             for lib_without_version in matched_libs do
21:                 if LEN(lib_without_version) > 1 then
22:                     matched_version ← SELECT_MAX_SIMILARITY_SCORE_VERSION(lib_name)
23:                     function_matched_libs.append(matched_version)
24:                 if LEN(lib_without_version) == 1 then
25:                     function_matched_libs.append(lib_name_version)
26:     return function_matched_libs
```

4. Evaluation

In this section, we design several experiments to evaluate BBDetector in terms of effectiveness, efficiency, and code obfuscation-resilience capability and compare it with the state-of-the-art tool ModX [11] and B2SFinder [7].

4.1. Experimental Setup

4.1.1. Data Collection

To evaluate our TPL detection method, we collect two types of datasets from ModX: Dataset I and Dataset II. Dataset I is a binary target program dataset with a known TPL reused relationship, which includes programs built by the nix package manager and a set of manually building binaries on Ubuntu 20.04. The nix binary dataset is as shown in Table 1, which is classified into three classes according to binary program size. A brief description of the ubuntu dataset is as shown in Table 2, in which the first column presents the binary names, and the last two columns present the number of TPLs reused by these binaries and the specific information of TPLs. Dataset II is a TPL dataset, which is used to build a third-party library database.

Table 1. Nix binary dataset used for evaluating TPL detection.

Dataset	Set A	Set B	Set C	Total
File Size (KB)	0–100	100–1000	>1000	16.4–4413.5
Average Size (KB)	61.8	297.8	2210.2	724.8
Number of Binaries	15	66	25	106
Average Number of Functions	159.6	652.1	4224.8	1425

Table 2. Ubuntu datasets used for evaluating TPL detection.

Program	Libs Num	Libs Linked
ssldump	2	libc.so.6, libpcap.so.0.8
vim	4	libtinfo.so.6, libdl.so.2, libm.so.6, libc.so.6
busybox	3	libresolv.so.2, libm.so.6, libc.so.6
tcpdummp	3	libpcap.so.0.8, libc.so.6, libcap-ng.so.0
openvpn	5	liblzo2.so.2, libssl.so.47, libcrypto.so.45, libpthread.so.0, libc.so.6
sqlite3	3	libdl.so.2, libpthread.so.0, libc.so.6
openssl	5	libssl.so.1.1, libdl.so.2, libcrypto.so.1.1, libpthread.so.0, libc.so.6

4.1.2. Baseline Techniques

To evaluate the effect of TPL detection, the proposed method BBDetector is evaluated and compared with ModX algorithm [11] and B2SFinder algorithm [7]. ModX is a state-of-the-art method for binary-to-binary TPL detection. It extracts syntactic features (strings literal, constant numbers) and semantic features (function call graph, functions accessing common data) to make the TPL detection. Due to ModX is not open source for the time being, we only compare it with its results presented in the paper. B2SFinder is a state-of-the-art method for binary-to-source TPL detection. It extracts seven kinds of code features (String literal, Switch/Case, If/else, Export function name, String Array, Integer Array, Enum Array) to make a separate analysis and detection. In process of binary-to-binary TPL detection, only the first four types are applicable. In our experiment, we consider three kinds of features (String, Switch/Case, If/else) due to the binaries being stripped of function names.

4.1.3. Evaluate Metrics

We evaluate and compare BBDetector , B2SFinder and ModX based on three criteria: effectiveness, efficiency and code obfuscation-resilient capability. Specifically, we compare the effectiveness and code obfuscation-resilient capability of these TPL methods on the nix binary dataset or ubuntu binary dataset by using three metrics: precision (P), recall (R)

and F1-score (F1). The definitions of these metrics are shown as Equation (4), where TP is the number of TPLs detected by detection methods that are actually in the binaries; FP is the number of TPLs which do not belong to the binaries, but are wrongly recognized by detection methods; and FN is the number of TPLs which belong to the binaries, but are unrecognized by detection methods. For efficiency, we compare the detection time of each method.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \text{F1-score} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

4.2. Effectiveness

In this section, we use both the nix binary dataset and ubuntu binary dataset to measure the effectiveness of BBDetector from three metrics: precision (P), recall (R) and F1-score (F1). For nix binaries; there are a total of 323 real reuses. Based on these reuse labels, we compare BBDetector with B2SFinder and ModX and summarize the experiment results of the nix binary dataset in Table 3. Since ModX does not open its source code, we present the original experiment results in the paper [11].

Table 3. TPL detection on nix binary dataset.

Method	TP	FP	FN	Precision	Recall	F1-Score
BBDetector	162	25	161	86.6%	50.2%	63.5%
B2SFinder	138	22	185	86.3%	42.7%	57.1%
ModX	/	/	/	85.6%	49.6%	62.8%

In Table 3, we can observe that BBDetector outperforms B2SFinder and ModX in terms of these three metrics. B2SFinder only considers basic syntactic features. To compare with it, BBDetector obtains an F1-score of 63.5%, 11.21% higher than that of B2SFinder, demonstrating the necessity and usefulness for also taking into account fine-grained function-level semantic characteristics. ModX considers the FCG graph and features of functions accessing the same data, which leads to a huge time consumption when considering the expensive graph matching process and is also based on syntactic feature when considering function features. To compare with it, BBDetector obtains a 1.11% higher F1-score than that of ModX, proving the BBDetector validity of choosing anchor functions for semantic matching in BBDetector , even without considering the expensive graph matching.

In Table 4, we list the detailed TPL detection results for the ubuntu binary dataset. The first column presents the specific target binary file name and the second column presents the number of reused third-party libraries in each of them. The rest columns present the number of TP (true positives), FP (false positives) and FN for BBDetector , B2SFinder and ModX, respectively. As shown in Table 4, BBDetector achieves the highest F1-score of 77%, 14.93% higher than that of B2SFinder and 1.32% higher than that of ModX.

Moreover, we analyze and evaluate the effectiveness against some vulnerable version of libraries. In our TPL database, there are vulnerable vesion of libraries, some of which is as shown in Table 5. For example, library `libbsd.so.0` contains vulnerabilities `CVE-2019-20367` and `CVE-2016-2090`. Because the detection results of ModX do not give the contained specific library information, we only briefly analyze the result of BBDetector and B2SFinder on some vulnerable version of libraries. The target binary program `curl` reuses the vulnerable library `libcurl.so.4`. Since `curl` only partially reuses this library and the ratio of the number of reused functions is only 11.28% (number of common functions: 787, total number of functions in the library: 6975), the number rate of reused synactic features is small (the common string percent is only 4.3%) and B2SFinder does not identify this reuse relationship. However, BBDetector also considers function semantic features and measures the average similarity score (0.83, which is greater than the threshold) of the anchor functions between them; thus, BBDetector can correctly detect the reuse relationship between them.

Table 4. TPL Detection on ubuntu binary dataset.

Binary	Libs Linked	BBDetector			B2SFinder			ModX		
		TP	FP	FN	TP	FP	FN	TP	FP	FN
busybox	3	2	0	1	2	0	1	1	1	2
openssl	5	4	0	1	3	0	2	2	1	3
openvpn	5	3	1	2	1	2	4	4	0	1
sqlite3	3	1	0	2	1	0	2	3	1	0
ssldump	2	2	1	0	2	1	0	2	0	0
tcpdump	3	3	1	0	3	1	0	3	0	0
vim	4	3	1	1	3	1	1	2	0	2
Total	25	18	4	7	15	5	10	17	3	8
		BBDetector			B2SFinder			ModX		
		Precisicon	Recall	F1	Precisicon	Recall	F1	Precisicon	Recall	F1
		82%	72%	77%	75%	60%	67%	85%	68%	76%

Table 5. Some vulnerable version of libraries and the vulnerabilities they contain.

TPL	Vulnerabilities Contained in TPL
libc.so.6	CVE-2022-35023
libpcap.so.0.8	CVE-2019-15165
libbsd.so.0	CVE-2019-20367,CVE-2016-2090
libresolv.so.2	CVE-2015-7547,CVE-2015-5180
libcurl.so.4	CVE-2016-5421,CVE-2016-5420,CVE-2016-5419,CVE-2015-3153,CVE-2013-1944

4.3. Efficiency

In this section, we investigate the detection time of BBDetector and compare it with the B2SFinder and ModX in terms of the nix binary dataset. We compare the detection time of BBDetector with existing tools by employing the nix binary dataset. Note that the detection time does not include the database construction time (Since it is a one-time job, we will not consider it in the TPL detection process). Table 6 shows the comparison result of detection time. The first column represents the size (KB) of target binaries and the last three columns represent the average detection time for each target binary of the corresponding size in terms of BBDetector , B2SFinder and ModX, respectively. Besides, the last line represents the average detection time for all target binaries. Since B2SFinder only uses basic syntactic features, it has better performance than BBDetector and ModX. BBDetector considers function semantic features that need to perform function embedding, which will also cause time consumption. When both consider semantic features, the TPLs detection time-consuming of BBDetector is only 30.02% of ModX, which greatly improves the detection efficiency. ModX performs expensive graph matching and function semantic matching. When it measures similarities, which are mainly unstructured data, it must compare these features one by one in the detection procedure. However, we store these features using quantization-based indexes and search them efficiently, which greatly improves time efficiency.

Table 6. The average detection time of TPL detection for nix binary dataset(s).

Size	BBDetector	B2SFinder	ModX
Set (0 100 KB)	46.1	49.8	255
Set (100–1000 KB)	251.6	157.3	915.3
Set (>1000 KB)	1141.4	245.5	3538.8
Average	432.4	162.9	1440.6

Next, we analyze the time complexity of the third-party libraries detection based on function features. Assuming the target binary is represented as $B = vb_1, vb_2, \ldots, vb_m$, where m is the number of functions in B. The third-party libraries are represented as

$L = L_1, L_2, \ldots, L_k$, where k is the number of libraries in the TPL database, and a library L_i in L is represented as $L_i = vl_1, vl_2, \ldots, vl_n$, where n is the average number of functions in L_i. If we perform a one-to-one search and matching, the time complexity of the whole detection process based on function features is $O(m * k * n)$, which is very time-consuming because $k * n$ is the total number of functions in libraries. However, in BBDetector we use quantization-based indexes to divide the function vector into *nlist* clusters and we just need to compare the vector $vb_i \in B$ with *nlist* clustering centroids, whose time complexity is $O(nlist)$ and select the nearest n_{probe} (which is very little) clusters to compare with, whose time complexity is $O(n_{probe} * s_c)$, where s_c is the average size (i.e., $k * n/nlist$) of a cluster. Thus, the time complexity of BBDetector is $O(m * (nlist + n_{probe} * s_c))$, which is much smaller than $O(m * k * n)$, which greatly improves the search efficiency.

4.4. Obfuscation-Resilient Capability

In this section, we evaluate the obfuscation-resilient capability of TPL detection methods. The obfuscation-resilient capability is an important metric to measure the performance of a TPL detection tool since obfuscation techniques can discount the detection performance. We select the seven ubuntu binary program dataset and use a popular obfuscation tool, Armariris [25], to obfuscate these programs. Armariris is an obfuscator based on the LLVM project for multiple languages and platforms, currently supporting string obfuscation, control flow flatten and instruction substitutions.

Based on these obfuscated target binary problems, we only compare BBDetector with B2SFinder, since ModX has no experiment in designing obfuscations and is not open source for the time being, at present. The specific detection results are presented in Table 7. From the table, we can observe that B2SFinder only achieves an F1-score of 28%, much lower than that of 67% achieved by it with the non-obfuscated binary programs. This is because the obfuscation technique alters the syntactic information of a program, leading to the syntactic features being different from the original TPLs and without a doubt, affecting the detection performance of TPL detection methods based on syntactic features. However, we can see BBDetector achieves an F1-score of 71%, slightly lower than the F1-score of 77% achieved with the non-obfuscated binary programs, which indicates BBDetector is less affected by the code obfuscation technique due to we consider the function semantic features, demonstrating the obfuscation-resilient capability of BBDetector towards code obfuscation technique.

Table 7. TPL detection on obfuscation ubuntu binary dataset.

Binary	Libs Linked	BBDetector			B2SFinder		
		TP	**FP**	**FN**	**TP**	**FP**	**FN**
busybox	3	2	0	1	2	0	1
openssl	5	3	0	2	1	0	4
openvpn	5	3	1	2	0	0	5
sqlite3	3	1	0	2	0	0	3
ssldump	2	2	1	0	0	0	2
tcpdump	3	2	1	1	1	0	2
vim	4	3	1	1	0	0	4
Total	25	16	4	9	4	0	21

BBDetector			B2SFinder		
Precisicon	Recall	F1	Precisicon	Recall	F1
80%	64%	71%	100%	16%	28%

5. Conclusions

In this paper, we propose BBDetector , a more precise and scalable TPL detection method for target binary executables with fine-grained function-level features. There are a lot of works related to TPL detection. Most existing methods only consider syntactic

features, and a few rely on expensive graph matching and function semantic features that access the same data. Compared to these TPL detection methods, the innovation points of our work is embodied in two aspects: more precise and scalable. On one hand, in addition to syntactic features, we also consider fine-grained function-level semantic features, which can more accurately represent a target binary program or a TPL. On the other hand, to make TPL detection based on semantic features more scalable, we design a function vector similarity search method to seek anchors functions (each pair of anchor functions are considered as matched functions) and the candidate TPLs. Then, we carry out TPL detection based on these candidate libraries and anchor functions. The experiment results demonstrated that BBDetector outperforms the existing optimal TPL detection methods B2SFinder and ModX in effectiveness, efficiency and obfuscation-resilient capability.

Author Contributions: Conceptualization, X.Z. and J.W.; Data curation, Z.F. and X.Y.; Methodology, X.Z. and Z.F.; Investigation, X.Z. and J.W.; Writing—original draft preparation, X.Z.; Writing—review and editing, X.Z., X.Y., J.W. and S.L.; Supervision, J.W. and S.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Foundation Strengthening Key Project of the Science & Technology Commission 2019-JCJQ-ZD-113; in part by the National Key Research and Development Program under Grant 2022YFB3305203; in part by the National Natural Science Foundation of China under Grant U2133208 and Grant 62101368; and in part by the Basic Research Program of China under Grant 2021-JCJQ-ZQ-082; and in part by the Sichuan Youth Science and Technology Innovation Team under Grant 2022JDTD0014.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The calculated data presented in this work are available from the corresponding authors upon reasonable request.

Acknowledgments: The author would like to thank the anonymous reviewers for their valuable comments on our paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. 2021 State of the Software Supply Chain. Available online: https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021 (accessed on 6 October 2022).
2. Kaseya VSA Ransomware Attack. Available online: https://helpdesk.kaseya.com/hc/en-gb/articles/4403584098961-Incident-Overview-Technical-Details (accessed on 6 October 2022).
3. Popular NPM Library Hijacked to Install Password-Stealers, Miners. Available online: https://www.bleepingcomputer.com/news/security/popular-npm-library-hijacked-to-install-password-stealers-miners/ (accessed on 6 October 2022).
4. Hemel, A.; Kalleberg, K. T.; Vermaas, R.; Dolstra, E. Finding Software License Violations through Binary Code Clone Detection. In Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011, Waikiki, Honolulu, HI, USA, 21–28 May 2011.
5. Duan, R.; Bijlani, A.; Meng, X.; Kim, T.; Lee, W. 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, Dallas, TX, USA, 30 October–3 November 2017.
6. Tang, W.; Chen, D.; Luo, P. BCFinder: A Lightweight and Platform-Independent Tool to Find Third-Party Components in Binaries. In Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 4–7 December 2018.
7. Yuan, Z.; Xu, J.; Piao, A.; Xuey, J.; Yu, C. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019.
8. Tang, W.; Luo, P.; Fu, J.; Zhang, D. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), Los Alamitos, CA, USA, 18–21 February 2020.
9. Ban, G.; Xu, L.; Xiao, Y.; Li, X.; Yuan, Z.; Huo, W. B2SMatcher: Fine-Grained version identification of open-Source software in binary files. *Cybersecurity* **2021**, *4*, 1–21. [CrossRef]

10. Woo, S.; Park, S.; Kim, S.; Lee, H.; Oh, H. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021.
11. Yang, C.; Xu, Z.; Chen, H.; Liu, Y.; Gong, X.; Liu, B. ModX: Binary Level Partially Imported Third-Party Library Detection via Program Modularization and Semantic Matching. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 25–27 May 2022.
12. Ding, S.; Fung, B.; Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proceedings of 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019.
13. Dolstra, E.; Visser, E.; MD Jonge. Imposing a memory management discipline on software deployment. In Proceedings of the 26th International Conference on Software Engineering, Edinburgh, UK, 28 May 2004.
14. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [CrossRef]
15. Baxter, I.D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L. Clone detection using abstract syntax trees. In Proceedings of the International Conference on Software Maintenance, Bethesda, MD, USA, 20–20 November 1998.
16. Shirani, P.; Wang, L.; Debbabi, M. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. In Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment, Bonn, Germany, 6–7 July 2017.
17. Zhang, D.; Luo, P.; Tang, W.; Zhou, M. OSLDetector: identifying open-source libraries through binary analysis. In Proceedings of 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, VIC, Australia, 21–25 September 2020.
18. Tang, W.; Wang, Y.; Zhang, H.; Han, S.; Luo, P.; Zhang, D. LibDB: An Effective and Efficient Framework for Detecting Third-Party Libraries in Binaries. In Proceedings of the 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA, 23–24 May 2022.
19. Mobile Ads SDK. Available online: https://developers.google.cn/ad-manager/mobile-ads-sdk (accessed on 6 October 2022).
20. IDA Pro. Available online: https://www.hex-rays.com/products/ida/ (accessed on 6 October 2022).
21. Jegou, H.; Douze, M.; Schmid, C. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal.* **2010**, *33*, 117–128. [CrossRef] [PubMed]
22. Johnson, J.; Douze, M.; Jégou, H. Billion-scale similarity search with GPUs. *IEEE Trans. Big Data* **2017**, *7*, 535–547. [CrossRef]
23. Xu, X.; Chang, L.; Qian, F.; Yin, H.; Song, D. 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, Dallas, TX, USA, 30 October–3 November 2017.
24. Massarelli, L.; Di Luna, G.A.; Petroni, F.; Baldoni, R.; Querzoni, L. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In Proceedings of the 16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Gothenburg, Sweden, 19–20 June 2019.
25. Armariris. Available online: https://github.com/GoSSIP-SJTU/Armariris (accessed on 16 September 2022).