BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching

Ling Jiang
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology
Shenzhen, China
11711906@mail.sustech.edu.cn

Junwen An
Southern University of Science and Technology
Shenzhen, China
12012109@mail.sustech.edu.cn

Huihui Huang
Southern University of Science and Technology
Shenzhen, China
12010336@mail.sustech.edu.cn

Qiyi Tang, Sen Nie, Shi Wu
Tencent Security Keen Lab
Shanghai, China
{dodgetang,snie,shiwu}@tencent.com

Yuqun Zhang*
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

ABSTRACT

While third-party libraries (TPLs) are extensively reused to enhance productivity during software development, they can also introduce potential security risks such as vulnerability propagation. Software composition analysis (SCA), proposed to identify reused TPLs for reducing such risks, has become an essential procedure within modern DevSecOps. As one of the mainstream SCA techniques, binary-to-source SCA identifies the third-party source projects contained in binary files via binary source code matching, which is a major challenge in reverse engineering since binary and source code exhibit substantial disparities after compilation. The existing binary-to-source SCA techniques leverage basic syntactic features that suffer from redundancy and lack robustness in the large-scale TPL dataset, leading to inevitable false positives and compromised recall. To mitigate these limitations, we introduce BinaryAI, a novel binary-to-source SCA technique with two-phase binary source code matching to capture both syntactic and semantic code features. First, BinaryAI trains a transformer-based model to produce function-level embeddings and obtain similar source functions for each binary function accordingly. Then by applying the link-time locality to facilitate function matching, BinaryAI detects the reused TPLs based on the ratio of matched source functions. Our experimental results demonstrate the superior performance of BinaryAI in terms of binary source code matching and the downstream SCA task. Specifically, our embedding model outperforms the state-of-the-art model CodeCMR, i.e., achieving 22.54% *recall@1* and 0.34 *MRR* compared with 10.75% and 0.17 respectively. Additionally, BinaryAI outperforms all existing binary-to-source SCA tools in TPL detection, increasing the precision from 73.36% to 85.84% and recall from 59.81% to 64.98% compared with the well-recognized commercial SCA product.
https://www.binaryai.net

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; • **Software and its engineering** → *Software libraries and repositories.*

KEYWORDS

Software Composition Analysis, Static Binary Analysis

ACM Reference Format:
Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639100

1 INTRODUCTION

Software composition analysis (SCA) [24, 53, 71] refers to identifying the open-source components (reused third-party libraries, i.e., TPLs) contained in the software artifacts for cost-effective development. Based on the SCA result, developers can easily track the security threats introduced to the software artifact by TPLs, such as vulnerability propagation and license violation [4, 27, 36, 52]. Considering diverse forms of target software project and identified TPLs, the existing SCA techniques are divided into multiple categories (e.g., binary-to-binary SCA [54, 55, 69] and binary-to-source SCA [13, 41, 71]). In particular, as one of the mainstream SCA techniques, binary-to-source SCA techniques identify the source code projects as reused TPLs contained in the target binary file by measuring the similarity between the binary file and a large-scale collected TPL dataset based on their extracted code features

*Yuqun Zhang is the corresponding author. He is also affiliated with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, China.

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

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639100

(i.e., binary source code matching). During the modern DevSecOps CI/CD pipeline, binary-to-source SCA is normally integrated into the build or deploy phase to automatically scan the components along with security risks within binary files. Typically, binary-to-source SCA tends to be scalable and practical in real-world software development scenarios [13, 44, 53] by conveniently incorporating new open source repositories into the TPL dataset.

The existing binary-to-source SCA techniques utilize basic syntactic features (e.g., string literals) that remain consistent after compilation to perform binary source code matching. For instance, B2SFinder [71], the state-of-the-art academic SCA technique, leverages a weighted matching algorithm to combine three matching techniques for the corresponding basic syntactic features. Although basic features can be used to build the correspondence between binary and source code, the existing binary-to-source SCA techniques relying on basic features still have two limitations. First, these basic features typically exhibit a significant degree of redundancy in the large-scale TPL dataset. Such an issue causes feature duplication across collected TPLs that can further limit the precision of SCA, i.e., incurring inevitable false positives during feature matching. Moreover, it is commonly observed that few or even no common basic syntactic features exist between reused TPLs and target binary files, especially the binary files which are stripped of certain string literals such as function names [71], compromising the recall of SCA techniques [13] that rely on such features. In fact, binary code differs significantly from the source code, and few features remain consistent after compilation. Such disparity can even increase false negatives when applying traditional techniques to match basic features. Therefore, it is essential to employ finer-grained features, such as function-level features which typically contain more high-level syntactic and semantic information compared with basic syntactic features, to advance the accuracy of binary source code matching and further strengthen the downstream binary-to-source SCA task.

In this paper, we propose BinaryAI, a binary-to-source SCA technique with intelligent function-level binary source code matching. Given the disparities introduced by compilation, we adopt a transformer-based model to capture the token-based syntactic features and generate function embeddings for computing the similarity between binary and source functions. Specifically, BinaryAI uses the large language model from the suite of *Pythia* [8] as the starting point, followed by pre-training the model in a supervised manner using contrastive learning [43]. Based on the trained model, the embeddings for all source functions from a large-scale TPL dataset are generated offline and stored in a corpus (i.e., vector database). For the online SCA detection of the target binary file, BinaryAI performs decompilation to extract binary functions, which are further encoded into embeddings as queries to retrieve *top-k* similar source functions from the corpus. Furthermore, BinaryAI adopts *locality-driven matching* at the second phase of binary source function matching. Specifically, we leverage link-time locality and function call graph as additional structured information to capture the semantic features and identify the exactly matched source function from the *top-k* similar functions. Eventually, BinaryAI leverages the matched source functions to calculate the ratio of reused functions as the similarity score between collected TPLs and the target binary file, further identifying the components along with potential security risks whose similarity exceeds a pre-defined threshold as in many previous works [13, 54, 55, 71].

In this paper, we evaluate the effectiveness of BinaryAI in terms of binary source code matching and TPL detection (i.e., SCA). Specifically, we first construct three datasets: 1) training set for the model containing around 10M function pairs as positive samples, 2) large-scale TPL dataset to construct the SCA database and corpus for retrieving similar source functions, and 3) SCA test set with manually labeled components and binary-to-source function mappings. The evaluation results reveal that the binary source code matching model in BinaryAI outperforms the state-of-the-art model CodeCMR by increasing *recall@1* from 11.92% to 22.73% for the SCA test set. Moreover, the *locality-driven matching* can effectively identify the correct source function from *top-k* retrieved results, further increasing *recall@1* from 22.73% to 66.90% that is close to the upper bound 70.45% (i.e., *recall@100*) restricted by the model capability. Based on the matched source functions, we evaluate the accuracy of BinaryAI regarding TPL detection. The evaluation results demonstrate that BinaryAI dominates the performance among all the existing binary-to-source SCA tools, e.g., outperforming the start-of-the-art academic SCA tool B2SFinder by increasing the precision from 31.78% to 85.84% and the recall from 54.93% to 64.98%. It even outperforms the well-recognized commercial SCA product by increasing the precision from 73.36% to 85.84% and the recall from 59.81% to 64.98%.

To summarize, our paper makes the following contributions:

- To our best knowledge, we are the first to adapt function-level binary source code matching to binary-to-source SCA and train a transformer-based model to retrieve similar source functions.
- We propose a two-phase binary source function matching in BinaryAI by leveraging link-time locality to enhance the accuracy of function matching with the *top-k* retrieved results.
- We evaluate BinaryAI, where the results suggest that the model of BinaryAI significantly outperforms CodeCMR in binary source code matching. In addition, BinaryAI dominates the performance among the existing binary-to-source SCA tools.

2 BACKGROUND AND MOTIVATION

2.1 Software Composition Analysis

Software composition analysis (SCA) typically refers to identifying third-party libraries (TPLs) in the target software project to track security threats and license violations introduced by these open-source components. Given the potential risks to the software supply chain associated with accessing the source code (e.g., privacy policy), binary SCA has emerged as the predominant technique, which can be easily integrated into the build or deploy phase during DevSecOps to automatically scan the components in binary files [13, 53, 71]. Existing binary SCA techniques [13, 31, 54, 69, 71] extract software features from a large-scale TPL dataset to construct the SCA database and then utilize code clone detection to identify similar features between TPLs and the binary file. Subsequently, they recognize the TPLs as the reused components if the number of similar features exceeds a pre-defined threshold. Based on different forms of TPLs in the database, binary SCA can be divided into two categories: binary-to-source SCA [13, 53, 71] and binary-to-binary SCA [54, 55, 69].

2.1.1 Binary-to-Source SCA. The TPL dataset in binary-to-source SCA consists of large-scale crawled open-source C/C++ projects, the majority of which are GitHub repositories and source packages from the GNU/Linux community. By matching source code features extracted from the C/C++ repositories, binary-to-source SCA identifies the reused source-code-level TPLs in the target binary file. Specifically, the fundamental step in binary-to-source SCA is binary source code matching, which maps binary code to the corresponding source code. B2SFinder [71], the start-of-the-art tool, selects basic syntactic features (e.g., string literals) that still remain consistent after compilation to match the source code and open-source components. In addition to binary SCA, binary source code matching is crucial in other scenarios of software security, such as reverse engineering [41] and malware analysis [20]. To our best knowledge, the effectiveness of existing binary source code matching is generally compromised due to substantial disparities between binary and source code [70].

2.1.2 Binary-to-Binary SCA. In the binary-to-binary SCA task, the TPLs in the SCA database are stored in the binary format built from source packages. By leveraging publicly available package managers (e.g., Nix [12]), source packages can be compiled automatically across different versions, architectures, and optimization levels. Many existing binary-to-binary SCA techniques [32, 55, 69] integrate advanced embedding-based approaches to detect code similarity between binaries and further identify the reused libraries based on the SCA database. Specifically, they leverage deep neural network models to embed binary functions into the representation of vectors and perform binary code clone detection by measuring the similarity between function embeddings [11, 40, 57, 67]. Apart from basic syntactic features, these techniques typically capture semantic features such as the control flow graph (CFG) for each binary function to strengthen their accuracy of code clone detection and the downstream SCA task.

2.2 Motivation

In this section, we intend to discuss the respective limitations of binary-to-binary and binary-to-source SCA to motivate our approach. Notably, binary-to-binary SCA can be compromised by the poor scalability of the TPL dataset. In particular, due to the intricacies associated with automatic compilation, only a limited subset of source packages maintained by package managers can be compiled automatically into multiple versions of binary files and incorporated into the SCA database. Extensive open-source C/C++ projects, such as GitHub repositories, can hardly be included in the TPL dataset, which is hindered by the substantial overhead of manual compilation. For instance, ModX [69], the state-of-the-art technique, selects 100 most frequently reused TPLs from a total of 795 maintained by Nix [12] to build the binaries as the database. However, there are around 10K TPLs (~100X compared with ModX) in the existing largest dataset for binary-to-source SCA [24]. Note that the limited scale of the TPL dataset can significantly inhibit the practicality of SCA due to the likelihood that the contained TPLs and the corresponding vulnerabilities cannot be identified. Therefore, we select binary-to-source SCA as the primary subject of our investigation.

Subsequently, we deliberate the constraints of binary-to-source SCA. Existing binary-to-source SCA tools leverage basic syntactic features, such as string literals, to establish a correspondence between binary code and source code of TPLs, which may not well generalize to all the scenarios. Firstly, these basic features tend to exhibit a significant degree of redundancy in the large-scale TPL dataset. For instance, the string *"407 Proxy Authentication Required"*, which indicates a common HTTP error, duplicates across more than 50 TPLs within our collected dataset. The presence of redundant syntactic features decreases their uniqueness and effectiveness, incurring inevitable false positives to decrease the precision of SCA. Furthermore, it is commonly observed that few or even no common syntactic features exist between reused TPLs and target binary files, especially the binary files which are stripped of distinctive features such as string literals and exported function names [71]. Meanwhile, existing techniques for extracting strings from C/C++ source code are not inherently robust, e.g., missing strings generated by concatenating macro-defined and constant strings to mismatch the string literals extracted from the binary files in the corresponding TPLs, such that the recall of binary-to-source SCA can also be compromised. Therefore, it is essential to employ fine-grained features (e.g. function-level features) in binary-to-source SCA such that high-level semantic information can be processed to mitigate the issue of redundancy and unreliability with basic features.

In this paper, considering the substantial disparities between binary and source functions introduced by compilation, we attempt to enhance binary-to-source SCA by adopting a transformer-based model to produce function-level embeddings and conducting binary source code matching accordingly.

3 APPROACH

We propose BinaryAI, a binary-to-source SCA technique with intelligent binary source code matching. Figure 1 presents the workflow of BinaryAI, which consists of four phases: *feature extraction* (Section 3.1), *embedding-based function retrieval* (Section 3.2), *locality-driven matching* (Section 3.3) and *third-party library detection* (Section 3.4). Specifically, BinaryAI is initialized by extracting C/C++ source-code functions from extensive repositories in the TPL dataset and C-like pseudo-code functions from the target binary file via decompilation (marked as ❶). Accordingly, BinaryAI adopts the large language model to generate the embeddings for source and binary functions. Note that binary source code matching in BinaryAI is not an end-to-end process but is divided into two distinct stages. Specifically, BinaryAI first trains a transformer-based model to learn the token-based syntactic features and retrieves *top-k* most similar source functions from the corpus for each query binary function (❷). Next, BinaryAI utilizes additional structured representations (e.g., link-time locality) to capture semantic features and match the exact source function from the *top-k* candidates (❸). Eventually, BinaryAI identifies the reused TPL components when the corresponding ratio of common source functions exceeds a pre-defined threshold with the target binary file (❹).

3.1 Feature Extraction

We extract functions with other meta information from both the TPL dataset and the target binary file respectively, in preparation

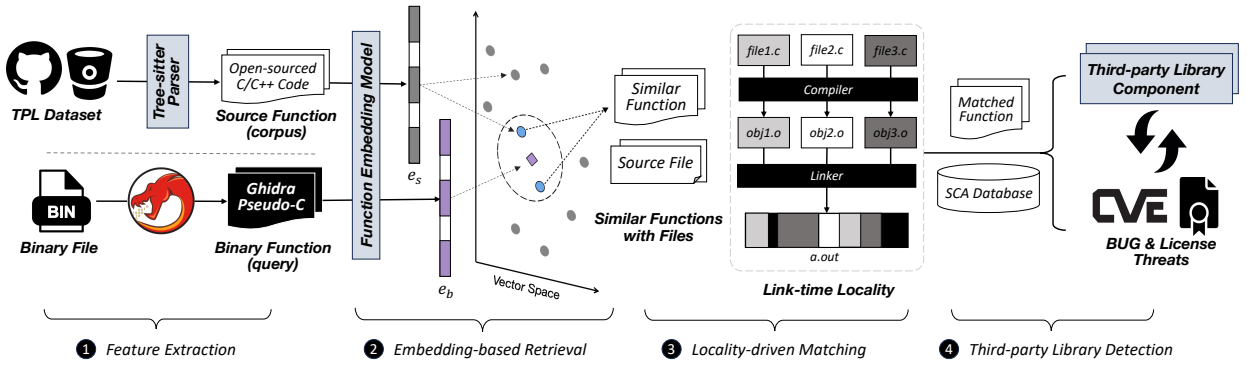


Figure 1: The workflow of BinaryAI

for subsequent phases of BinaryAI. Specifically, we characterize the features in terms of source function and binary function.

Source Function. All the open-source projects in the TPL dataset employ `git` as their version control system. For each project, we collect the C/C++ source files across all the versions (i.e., git tags) and distinguish each file with the hash value of its content. Then we apply tree-sitter [46], an open-source source code parser, to construct the file's abstract syntax tree (AST) with its built-in C/C++ language parsers and extract all the unique source functions. Meanwhile, we maintain two inverted indexes to store the correspondences of extracted source functions into the SCA database, where one index maps each source function to all the files containing it, and the other maps each source function to all the TPLs containing it.

Binary Function. We leverage Ghidra [2], an open-source reverse engineering framework developed by National Security Agency (NSA), to analyze the binary file, which involves disassembling the binary code and identifying functions, data structures, and other relevant information. Subsequently, Ghidra performs decompilation to generate the C-like pseudo code representation of the functions (i.e., binary functions). Additionally, we leverage Ghidra to extract the relative virtual address (denoted as *bin_rva*) as the ordinal number denoting the link-time locality in the binary file along with the function call graph as the inter-function communication. Note that we design BinaryAI with the assumption that the input binary file has been stripped, i.e., all the debugging and and symbol information are eliminated, which is common in real-world scenarios [71].

3.2 Embedding-based Function Retrieval

The core insight of BinaryAI is to perform function-level binary source code matching based on function embeddings. In particular, our objective is to train a model that learns meaningful vector representations for both binary and source functions in a single vector space, where similar binary-to-source function pairs are expected to stay close while dissimilar ones are far apart. In this way, their similarity can be calculated using their corresponding embeddings. Typical code representation learning allows only one single code format of the matched objects, i.e., either source-to-source [16, 37, 38, 49, 60] or binary-to-binary [28, 35, 39, 57, 66] code matching. However, for binary source code matching, C/C++ language features (e.g., function inlining [23]) and compiler optimization (e.g., code motion [30]) can lead to substantial differences between binary code and source code, and such disparity can be rather challenging when designing BinaryAI. To fill this gap, an ideal model needs to accurately capture subtle syntactic features to generate code embeddings for measuring similarity. Notably, existing large language models are extremely effective at learning the syntax of natural language [48, 51], and this ability extends to code languages as well [3, 17, 18]. In particular, a model trained in multiple programming languages can potentially identify similar token-based features across different code formats [72]. This can help detect code clones, even when the code has been translated into different languages. To this end, we use an existing large language model as the base model and further pre-train the model with a corpus of labeled binary source function pairs to build our model. Specifically, we apply a contrastive learning approach in a supervised manner to train the model acting as the function encoder to generate embeddings. This allows us to learn the code representations for both binary and source functions that minimize the distance between similar positive samples while distancing dissimilar negative samples. In this paper, we adopt *Pythia* [8], which is widely adopted by the research community, as the base model to train our model[1]. As mentioned before, we initialize the original model with 410M parameters from the suite of *Pythia* (i.e., `pythia-410m` [14]) and then further perform pre-training using contrastive learning.

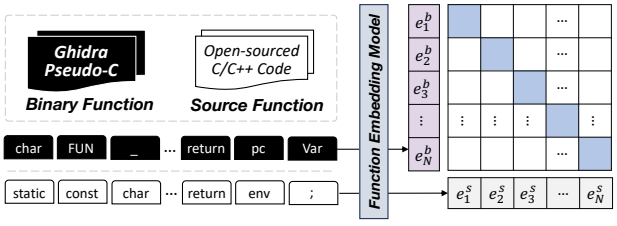


Figure 2: CLIP over binary and source function pairs

[1]The base model for BinaryAI has evolved through multiple iterations. Previously, we have also employed OPT [75] and BLOOM [61] as the based models. By the time of publication, we use *Pythia* because it has delivered optimal performance after training.

Note that as one of the key ingredients in contrastive learning, enlarging in-batch negatives can effectively help the model to learn more discriminative representations as it needs to distinguish between a larger number of positive and negative samples in each batch [22], leading to better representation learning and improved performance on downstream tasks (i.e., binary SCA). To this end, we leverage the loss function of CLIP (Contrastive Language-Image Pre-training) [43], which is originally designed to align the representations of images and text captions, as our contrastive training objective. Figure 2 presents the training process based on CLIP contrastive learning method. We first perform tokenization that converts the functions into a sequence of tokens. Subsequently, we pass the tokenized input through the *Pythia* model to obtain the function embeddings by extracting the output of the last hidden layer, where we denote (e_i^b, e_i^s) as the binary and source function embeddings of the i_{th} positive sample. Accordingly, (e_i^b, e_j^b) represents one negative sample if i is not equal to j. For the process of contrastive training, one batch consists of N binary-to-source pairs and CLIP calculates the cosine similarity matrix between all the possible pairs. The training objective is to maximize the similarity between N positive samples while minimizing the similarity for the rest $N*(N-1)$ negative samples via a symmetric cross-entropy loss over the matrix [58]. Equation 1 presents the binary-to-source loss function L_{bin} and the source-to-binary loss function L_{src} where τ is a learnable parameter to scale the logits. Note that the two loss functions are differed by swapping binary and source function embeddings when computing the similarity. Therefore, the overall loss function L_{CLIP} is the average value of L_{bin} and L_{src} denoted as Equation 2. Moreover, we extend the Momentum Contrast (MoCo) methodology [19] to our contrastive pre-training, which further increases the number of negative samples and enables more effective contrastive learning by building dynamic dictionaries for CLIP.

$$L_{bin(src)} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp\left(\text{sim}(e_i^{b(s)}, e_i^{s(b)})/\tau\right)}{\sum_{j=1}^{N}\exp\left(\text{sim}(e_i^{b(s)}, e_j^{s(b)})/\tau\right)} \quad (1)$$

$$L_{CLIP} = (L_{bin} + L_{src})/2 \quad (2)$$

We deploy the trained model in BinaryAI by initially deriving function embeddings offline for all the source functions in the SCA database (with a total of 56,342,179 unique functions from 12,013 TPLs) and store the source function embeddings to the vector database as corpus. Then for the online binary-to-source SCA task, we extract binary functions from the target binary file and perform real-time derivation of binary function embeddings. These derived embeddings serve as queries to retrieve similar source functions from the corpus for a given binary function. Figure 3 presents the retrieved *top-1* most similar source function with the query of a binary function. This is actually a positive sample and has a similarity of 0.982. Eventually, we apply the relative virtual address (denoted as *bin_rva*) as the identifier for binary functions and obtain their *top-k* most similar source functions as the output of *embedding-based function retrieval*. Note that we attach all the source files containing the corresponding functions (i.e., *src_func* ⇒ *src_files*) by accessing the inverted index described in Section 3.1.

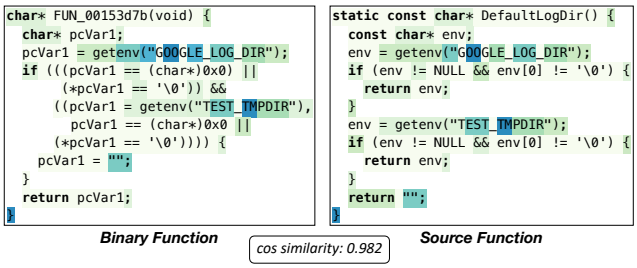


Figure 3: Retrieved binary and source function pair

3.3 Locality-driven Matching

Ideally, we can directly select the source function with the highest similarity to the binary function in terms of function embeddings (i.e., *top-1 of embedding-based function retrieval*) as our matching result. Nevertheless, due to the subtle modifications in source functions across different versions, there is a significant presence of similar functions within the large-scale TPL source repositories. Consequently, relying solely on language model-generated function embeddings to capture token-based syntactic features is insufficient for accurately matching the source functions, since the retrieved *top-k* source functions can be quite similar. To tackle this issue, we attempt to leverage link-time locality [15] (i.e., relative virtual address as described in Section 3.1) and function call graph as supplementary inter-function communication representing structured semantic features, which can help further identify the positive sample from the *top-k* similar source functions in the second phase of binary source code matching. In this section, we present the fundamental rationale and the workflow of *locality-driven matching*.

For the conventional C/C++ toolchain used to build binary files, the source code files (*file.c*) are initially compiled into object files (*obj.o*) by the compiler. Subsequently, the linker resolves symbol references between object files and combines them to produce the binary file, where the code sections of each object file are merged. By analyzing the process of compilation, we can derive several basic findings. 1) All the source functions in the same source file are compiled into a single object file although their relative locality to the source file might change. 2) The object files are continuously linked into the binary file, and all the functions (i.e., binary functions in the machine code format) within the code section of the object file preserve their relative locality. 3) Due to C/C++ template functions and conditional compilation, one source function in a source file can correspond to multiple binary functions in the object files (i.e., "1-to-n" mapping from source to binary functions). Inspired by these findings, we can further derive that the link-time localities of the binary functions compiled from the same source file are rendered continuous in the binary file. Correspondingly, given the address space of the binary file, we can perform reverse engineering by cutting intervals containing continuous binary functions to recover the boundaries of the object files [15] and further identify the corresponding source files compiled into the binary file, thus accurately matching the source functions. To this end, we extract the continuous function pairs by link-time locality for each source file as the function intervals that can be mapped back to the address space of the binary file. Note that we consider isolated

Algorithm 1: Locality-driven Matching

```
   Input: bin2src_topk          ▷ Retrieved topk similar source functions
   Result: bin2src_match        ▷ Matched binary source function pairs
1  Function MatchFuncPairs:
2      file2pairs, intervals, bin2src_match ← ∅
3      for (bin_rva, similar_funcs) ∈ bin2src_topk do
4          for (src_func, src_files) ∈ similar_funcs do
5              file2pairs[file].add(bin_rva ⇒ src_func) for file in src_files
6          bin2src_match[bin_rva] = top1_similar_src_func
7      for (file, bin2src_pairs) ∈ file2pairs do
8          intervals.add(MaxFileInterval(bin2src_pairs, file.func_count)
9      intervals.sort(key=lambda x : (x.start, -x.end, x.max_hit))
10     for interval in intervals do
11         if interval.start > last_select_interval.end then
12             for (bin_rva, src_func) ∈ RestrictByFCG(interval.func_pairs) do
13                 bin2src_match[bin_rva] = src_func
14             last_select_interval ← interval
15     return bin2src_match
16
17 Function MaxFileInterval(bin2src, file_func_count, i=0, j=0):
18     address ← sort(bin2src.keys)
19     interval ← initInterval(max_hit=0)
20     while j < len(address) do
21         func_slice ← { bin_rva : bin2src[bin_rva] for bin_rva in address[i : j] }
22         src_hit ← len(func_slice.values)
23         if src_hit <= file_func_count then
24             if src_hit > interval.max_hit then
25                 interval.max_hit ← src_hit
26                 interval.start, interval.end ← i, j
27                 interval.func_pairs ← func_slice
28             j++
29         else i++
30     return interval
```

function pairs in the file as invalid matches and eliminate them while selecting the continuous interval. Compared to other files, the files compiled into the binary file should have a longer continuous interval of functions. Therefore, we form the file selection as an interval covering problem within the address space of the binary file and further utilize function call graph to facilitate the binary source function matching within the selected files.

Algorithm 1 presents the overall workflow of *locality-driven matching*. First, we obtain all the included files of each source function from retrieved *top-k* candidates and build the index *file2pairs* mapping each source file to all its retrieved binary source function pairs (lines 3-5). Meanwhile, we initialize the matching result with the *top-1* most similar source function (line 6). Next, we extract the continuous function pairs for each source file (lines 7-8). Specifically, we sort the function pairs by *bin_rva* (i.e., link-time locality) and leverage a sliding window with two separate pointers (*i*&*j*, both are initialized to 0) to slice the file and generate the corresponding function interval (lines 17-30) that simultaneously satisfies two conditions. 1) The number of the source functions within the interval does not exceed the total number of functions in the file (line 23). 2) The sliced interval has the maximum number of function pairs (lines 24-25). Furthermore, we map the continuous function intervals extracted from each source file back to the address space of the binary file based on *bin_rva*. As mentioned before, we attempt to select longer intervals to cover as many functions within the binary file as possible. To this end, we transform the file selection into an interval covering problem and tackle the problem greedily. Specifically, we sort the intervals according to a particular set of priorities (line 9), i.e., interval with lower start point (*x.start*), higher end point (*-x.end*), and more contained binary source function pairs (*x.max_hit*), which allows us to prioritize longer intervals that can also cover more binary functions compared with other equally long intervals. Before selecting the interval corresponding to the source file, we ensure that its start point should be higher than the end point of the previously selected interval to avoid potential overlaps (lines 10–11).

As in Finding 3, while there can be "1-to-n" correspondence from source to binary functions in a single source file, the retrieved function pairs in the file tend to be more complicated and elusive (e.g., "n-to-n" mappings caused by similar source functions). Correspondingly, incorrect function matching might still occur even when we identify the correct source file. To alleviate the issue, we leverage function call graph to restrict the function pairs for each selected file before generating the matching results (Algorithm 1, line 12). Suppose there are two function pairs (bin_1, src_1) and (bin_2, src_2) in the source file. If there are function calls between both the binary and source functions (e.g., bin_1 calls bin_2 and src_1 calls src_2), these two function pairs can be considered correct. In this case, we can filter out the mapping from these binary functions to other source functions, e.g., (bin_1, src_3). Figure 4 presents an example of function matching between source and binary files with the longest function interval $[25697, 287b3]$. We can observe from the function call graph that both the binary function `FUN_00028061` and source function `cJSON_AddTrueToObject` in the function pair have two callees, which are also matched in the same file. Therefore, we can derive three correct function matches via function call graph. Eventually, we assign all remaining function pairs in the selected files to update the matching results of binary-to-source functions (line 13) as the output of *locality-driven matching*.

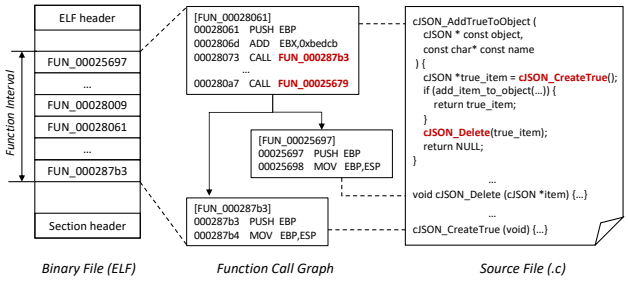


Figure 4: Function matching between source and binary files

3.4 Third-party Library Detection

BinaryAI acquires the matched source functions and further performs TPL detection (i.e., SCA task) for the target binary file as presented in Algorithm 2. The baseline technique is to retain all the included TPLs of each matched source function by referring to the SCA database (lines 3-4), which contains the inverted index of the

Algorithm 2: Third-party Library Detection

```
   Input: bin2src_match, tpl_dependency
   Result: components
1  Function DetectComponents:
2      tpl2func_match, components ← ∅
3      for (bin_rva, src_func) ∈ bin2src_match do
4          src_tpls ← retrieved TPLs containing src_func in SCA database
5          filtered_tpls ← FilterByDependency(src_tpls, tpl_dependency)
6          for tpl ∈ filtered_tpls do
7              tpl2func_match[tpl].add(bin_rva)
8      for (tpl, matched_funcs) ∈ tpl2func_match do
9          if len(matched_funcs) / tpl.total_func_count > θ then
10             components.add(tpl)
11     return components
12
13 Function FilterByDependency(src_tpls, tpl_dependency):
14     filtered_tpls ← src_tpls
15     for tpl ∈ src_tpls do
16         reused_tpls ← tpl_dependency[tpl]
17         if reused_tpls and src_tpls have intersection then
18             filtered_tpls.remove(tpl)
19     return filtered_tpls
```

correspondence from the source functions to TPLs as described in Section 3.1. However, noticing that in general source functions are extensively cloned across various TPLs [38, 49, 56, 59] in the large-scale dataset, such internal code clones [13] can lead to inevitable false positives if we retain all their included TPLs. For instance, assuming a binary file only contains the TPL *zlib* as the component, other TPLs reusing *zlib* are also identified as the components owing to the common functions cloned from *zlib*. To alleviate the issue, we follow previous works [24, 60, 62] to filter TPLs based on the TPL dependency which exhibits the reuse relations across TPLs and only retain the reused ones. Specifically, we leverage TPLite [24], the state-of-the-art technique based on function birth time (i.e., the earliest release time) and hierarchical path information, to generate the TPL dependency in advance, which works as the additional input of SCA to help identify the reused TPLs.

Algorithm 2 presents the workflow of TPL detection in BinaryAI. First, we extract all the included TPLs for each matched source function from the SCA database (lines 3-4). Subsequently, we filter TPLs based on the TPL dependency and count the matched functions for all the retained TPLs (lines 5-7). Specifically, we filter the TPLs whose reused ones are also included with the matched source function (lines 14-18), which should indicate the internal code clones between TPLs. For instance, the matched source function `deflateInit` is both included in TPLs *zlib* [47] (deflate.c) and *llvm* [45] (llvm/runtime/zlib/deflate.c). The TPL dependency indicates *llvm* reuses *zlib*, and we thus only select *zlib* by filtering out *llvm*. Eventually, we calculate the ratio of matched functions to the total number of source functions for each selected TPL, indicating the similarity between the binary file and the source code repository. If the ratio exceeds a pre-defined threshold θ, BinaryAI identifies the corresponding TPL as the contained component in the target binary file (lines 8-10). Meanwhile, BinaryAI detects whether security threats are introduced by these components by retrieving the official vulnerability repository, e.g., the NVD database [9].

4 EVALUATION

In the evaluation, we attempt to investigate the performance of BinaryAI by answering the following research questions:

- ***RQ1:*** How effective is the embedding model in measuring the similarity between the binary and source functions?
- ***RQ2:*** How does BinaryAI perform in terms of binary source code matching with the two separate phases?
- ***RQ3:*** What is the accuracy of BinaryAI in detecting TPLs in binary files compared to state-of-the-art techniques?

4.1 Dataset

To extensively evaluate the performance of BinaryAI in terms of different mechanisms, we first construct three datasets following existing works [13, 55, 60, 69, 71] for model training and the evaluation of the downstream binary-to-source SCA task.

4.1.1 Training Dataset. To obtain a large number of matched binary-to-source function pairs as positive samples for training the model, we construct the automatic compilation pipeline based on official ArchLinux packages [5] and Arch User Repository (AUR) [6] following the insight from BinaryCorp in jTrans [57]. Specifically, we apply the command `makepkg` to compile all the ArchLinux packages and AUR automatically. Meanwhile, we hook the compiler to generate the debugging information in the format of DWARF [1]. On one hand, we decompile the output binary file with Ghidra [2] to acquire the mapping from the virtual address to the binary function. On the other hand, we parse the DWARF debugging information and extract the mapping from the virtual address to the source file with line number. We further leverage tree-sitter [46] to slice out the corresponding source function in the file. By merging the mapping from both sides and filtering out mismatched functions due to runtime errors, we obtain around 10M matched function pairs with an average of about 500 tokens per function as the training set.

4.1.2 TPL Dataset & Corpus. Following previous works [56, 60, 62], we collect a large number of C/C++ open-source projects by crawling from GitHub repositories and source packages of the GNU/Linux community, and we obtain the dataset consisting of 12,013 TPLs, which is adequate for the SCA task [60]. Next, we extract 56,342,179 unique source functions[2] and derive the corresponding function embeddings based on the trained model which are stored persistently in the FAISS [26] database as the corpus. To our best knowledge, this corpus is the largest in the domain of binary source code matching, where the state-of-the-art technique CodeCMR [70] retrieves close embeddings within the corpus of 10,000 functions. A larger corpus is more practical as it includes more source functions that are similar to each other. This significantly increases the difficulty of embedding-based function retrieval and further validates the generality of our mechanism.

4.1.3 SCA Test Set. To evaluate the performance of TPL detection for BinaryAI, we construct our binary SCA test set compiled by 85 open-source software projects and obtain 150 binary files as the test cases, along with manually labeled components. Specifically, we collect highly prominent projects with more than 1K

[2] As of the time of publication, the magnitude is around 56M. Note that we deploy this module in industry, enabling continuous supplementation of new TPLs and source functions to improve the practicality of BinaryAI.

stars from GitHub. Furthermore, we select projects with over 10 sub-modules indicating that their compiled binaries are more likely to have multiple components, facilitating the evaluation of SCA. Next, we compile the source code of 85 projects into 150 stripped binary files across multiple architectures and compiler configurations. Meanwhile, we follow previous works [54, 55, 60] to manually label the reused TPLs as the components by rigorously analyzing all file paths, included header files, and other meta-information from SBOM files (e.g., CMakeLists), README, copyright, and license. As a result, a total of 1,045 components are labeled as the ground-truth SCA results, forming the largest dataset in binary-to-source SCA.

To further investigate the accuracy of *locality-driven matching* and its contribution to binary source function matching, we need to label the ground-truth correspondence between binary and source functions at a fine granularity within real-world binary files. Given the high expense of manual analysis, we label the binary-to-source function mappings for 15 (10%) most commonly used binary files out of 150 binary files. In particular, we perform reverse engineering manually to determine which function within the source files is compiled to the binary function in the object files. As a result, we obtain 23,529 matched function pairs within these 15 binary files.

4.2 Experiment Setup

To evaluate the effectiveness of *embedding-based function retrieval*, we include CodeCMR [70], the state-of-the-art binary source function matching model, for performance comparison with the model of BinaryAI. Note that CodeCMR utilizes separate function encoders (DPCNN for source function and GNN for binary function) and triplet loss as the contrastive learning objective. To ensure a fair comparison, we adopt the same training set described in Section 4.1.1. For the training process, we follow the training setup in the original paper for CodeCMR. As for BinaryAI, the maximum length of the embedding model is 2048, the training epoch is 196, the batch size is 512, and the learning ratio is 0.001. Furthermore, we follow CodeCMR to include BinPro [41] and B2SFinder [71] as traditional techniques in comparison with the neural network-based techniques for binary source code matching. Both BinPro and B2SFinder match code with basic syntactic features (e.g., string and integer constants), and we use Hungarian algorithm [42] based on the weights in their original papers to match source functions from the corpus. Note that we adopt 32,296 and 23,529 function pairs respectively from the validation set of the model and the 15 manually labeled SCA test cases as the query sets to validate whether the model can generalize to different datasets.

Given binary functions as queries, we adopt multiple metrics to evaluate the performance of retrieving similar source functions from the corpus. Specifically, we adopt *MRR* (Mean Reciprocal Rank) computed by averaging the reciprocal ranks across all queries as denoted in Equation 3. To verify the upper bound of model capability and the effectiveness of *locality-driven matching*, we also adopt the count of positive samples that can be detected within retrieved *top-k* similar functions and the corresponding recall by dividing it to the total number of queries (denoted as *Count/Recall@k*).

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{3}$$

For *locality-driven matching*, we evaluate the accuracy by identifying positive binary source function pairs, as well as its contribution to refining the *top-1* result retrieved by the model, where we adopt the 15 binary files with manually labeled function mappings as the ground truth. Eventually, we utilize all the 150 binary files with 1,045 labeled reused TPLs for the SCA task. We compare BinaryAI with the existing binary-to-source SCA tools including two academic tools: OSSPolice [13] and B2SFinder [71], and two well-established commercial products: bsca-A and bsca-B[3]. For a fair comparison, we utilize the same TPL dataset (12,013 projects) to build their corresponding SCA database.

Note that we adopt the same metrics for investigating these two tasks that include *Precision* (i.e., the ratio of true positives to all the derived results), *Recall* (i.e., the ratio of true positives to all the ground-truth data), and *F1* score (i.e., the measure of accuracy by considering both precision and recall). Considering the trade-off between precision and recall, we set the threshold θ to 0.01 that achieves the maximum *F1* score.

4.3 Results and Analysis

4.3.1 RQ1: Effectiveness of Function Embedding. We first compare BinaryAI with CodeCMR in terms of *embedding-based function retrieval*. Table 1 demonstrates the evaluation results of retrieving source functions with two query sets as described in Section 4.2. We can observe that BinaryAI outperforms CodeCMR in both query sets in terms of *MRR* (0.3006 vs. 0.1431 and 0.3958 vs. 0.2232). Moreover, BinaryAI can retrieve more positive samples for all *top-k* setups (i.e., *Count/Recall@k*) compared with CodeCMR. For instance, within the *top-10* retrieved results with queries from the binary SCA test set, BinaryAI detects 13,493 matched source functions for all the 23,529 queries with 57.35% recall while CodeCMR detects 7,873 with 33.46% recall. By combining two query sets, BinaryAI achieves 0.3407 *MRR* in contrast to 0.1769 of CodeCMR, indicating that the positive samples retrieved by BinaryAI tend to have a higher average rank (around the 3rd, 1/0.34). Additionally, BinaryAI effectively increases the *recall@1* from 10.75% to 22.54% and *recall@100* from 33.87% to 56.60% compared with CodeCMR.

Note that BinaryAI and CodeCMR respectively employ a large language model (i.e., *Pythia*) and a combination of DPCNN and GNN as the base model, along with CLIP loss and triplet loss as the training objective. To perform an ablation study on the model and training objective respectively, we further train two new models by reassembling the base models and training objectives from BinaryAI and CodeCMR. Table 1 demonstrates that CodeCMR increases *MRR* from 0.1431 to 0.2319 and *recall@1* from 9.89% to 16.89% in the validation set by updating the loss function from triplet loss to CLIP. For BinaryAI, modifying the training objective from CLIP to triplet loss degrades the performance (0.3006 vs. 0.2774 for *MRR*), indicating the effectiveness of the loss function from CLIP. Then we investigate the impact of the base model, we can observe that even though CodeCMR improves the performance by using CLIP, this effect is still inferior to BinaryAI trained with triplet loss (0.2319 vs. 0.2774 for *MRR*). Therefore, we can demonstrate the advantage

[3]The specific commercial tools are not disclosed due to the constraints of confidentiality agreements and the proprietary nature of the software, thus the names of the commercial tools used in this paper have been anonymized.

Table 1: Result of retrieving similar source functions

Model	Objective	Validation Set of Model (query=32,296)					Binary SCA Test Set (query=23,529)				
		MRR	Count/Recall@1	Count/Recall@10	Count/Recall@50	Count/Recall@100	MRR	Count/Recall@1	Count/Recall@10	Count/Recall@50	Count/Recall@100
BinPro	N/A	0.0027	771 / 2.39	1,165 / 3.61	1,593 / 4.93	1,845 / 5.71	0.0036	612 / 2.60	944 / 4.01	1,262 / 5.36	1,507 / 6.40
B2SFinder	N/A	0.0042	945 / 2.93	1,717 / 5.32	2,108 / 6.53	2,436 / 7.54	0.0048	864 / 3.67	1,305 / 5.55	1,740 / 7.40	2,082 / 8.85
CodeCMR	Triplet	0.1431	3,195 / 9.89	6,543 / 20.26	7,827 / 24.24	8,347 / 25.85	0.2232	2,805 / 11.92	7,873 / 33.46	9,875 / 41.97	1,0561 / 44.89
CodeCMR	CLIP	0.2319	5,456 / 16.89	10,589 / 32.79	12,256 / 37.95	12,801 / 39.64	0.2820	3,638 / 15.46	9,889 / 42.03	12,510 / 53.17	13,319 / 56.61
BinaryAI	Triplet	0.2774	6,552 / 20.29	12,627 / 39.10	14,009 / 43.38	14,460 / 44.77	0.3539	4,692 / 19.94	12,113 / 51.48	14,650 / 62.26	15,395 / 65.43
BinaryAI	**CLIP**	**0.3006**	**7,235 / 22.40**	**13,465 / 41.69**	**14,682 / 45.46**	**15,020 / 46.51**	**0.3958**	**5,348 / 22.73**	**13,493 / 57.35**	**15,873 / 67.46**	**16,576 / 70.45**

of utilizing a large language model in the domain of binary source function matching.

> ***Finding 1:*** *BinaryAI can be more effective than CodeCMR in terms of the embedding-based function retrieval with the usage of LLM and CLIP as the training objective.*

We further investigate the difference between neural network-based techniques (i.e., BinaryAI and CodeCMR) and the existing feature-matching-based techniques (i.e., BinPro and B2SFinder). We can observe that the performance of the traditional techniques is rather limited in retrieving matched source functions. Specifically, *MRR* for BinPro and B2SFinder in both query sets is less than 0.005, indicating that the matched source function has a rank of over 200 on average for each query. Moreover, both BinPro and B2SFinder recall less than 10% positive samples within *top-100* and less than 5% at *top-1* with the two query sets. Next, we investigate the reason and find several factors leading to decreased performance. Firstly, many source functions in the corpus share similar basic features, making it challenging to distinguish them effectively. Secondly, some binary functions as queries lack meaningful basic features, further compromising the retrieved results.

> ***Finding 2:*** *The existing feature-matching-based techniques incur limited performance in matching source functions from large-scale corpus, further indicating the effectiveness of embedding-based function retrieval.*

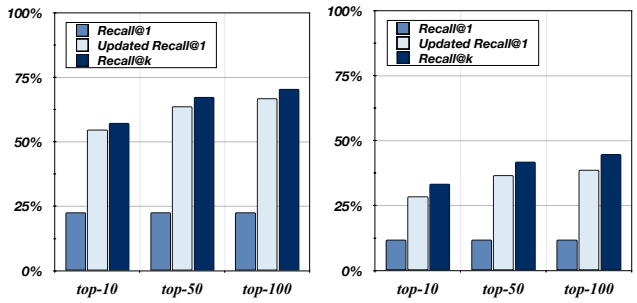
4.3.2 RQ2: Accuracy of Binary Source Code Matching. Previous findings indicate that BinaryAI achieves distinct improvement in retrieving source functions from large-scale corpus compared with the state-of-the-art techniques. However, BinaryAI is still limited to binary source code matching by directly applying *recall@1* (22.73% for 23,529 queries from the binary SCA test set in Table 1), which is insufficient for the downstream SCA task. In this RQ, we investigate the accuracy of *locality-driven matching* along with its contribution to binary source code matching based on the SCA test set with 15 binary files. Table 2 presents the matching results with the input of retrieved *top-10* similar source functions. Note that in addition to the results that match the ground truth (denoted as "Exact Match"), we also follow previous works [29, 60] to include the results that are identical to ground truth after normalization (denoted as "Fuzzy Match") since such results are applicable for other downstream tasks that do not require high accuracy, such as reverse engineering [10]. Overall, we can observe that the precision for the exact match is 81.63% on average. More specifically, the precision exceeds 75% in all binary files ranging from 75.19% (*turbobench*) to 94.92% (*hyriseSystem*). Such results illustrate that the accuracy of function matching based on link-time locality and function call graph is high and can generalize to all the binary files in the SCA test set. Moreover, the precision for the fuzzy match is 95.86% on average and exceeds 90% in all binary files ranging from 90.00% (*turbobench*) to 98.30% (*nano_node*), i.e., a large amount of false positives can match the ground truth after normalization.

> ***Finding 3:*** *Locality-driven matching can effectively identify the exact source function from top-k retrieved results and such results generalize to different binary files.*

Table 2: Result of locality-driven matching (k=10)

Binary	#Label	BinaryAI	Exact Match			Fuzzy Match		
			#TP	P (%)	R (%)	#TP	P (%)	R (%)
controlblock	185	107	86	80.37	46.49	99	92.52	53.51
db_bench	359	253	209	82.61	58.22	239	94.47	66.57
dosbox_core	2,804	2,042	1,854	90.79	66.12	1,974	96.67	70.40
eth_sc	267	232	190	81.90	71.16	221	95.26	82.77
hyriseSystem	318	197	187	94.92	58.81	193	97.97	60.69
kvrocks	2,240	1,452	1,190	81.96	53.13	1,415	97.45	63.17
nano_node	1,604	939	752	80.09	46.88	923	98.30	57.54
pagespeed	6,430	3,442	2,683	77.95	41.73	3,305	96.02	51.40
prometheus	204	157	138	87.90	67.65	146	92.99	71.57
replay-sorcery	770	454	367	80.84	47.66	437	96.26	56.75
st-device-sdk	801	582	486	83.51	60.67	536	92.10	66.92
tendisplus	2,197	1,541	1,265	82.09	57.58	1,498	97.21	68.18
tic80	832	695	573	82.45	68.87	668	96.12	80.29
turbobench	762	270	203	75.19	26.64	243	90.00	31.89
yuzu-cmd	3,756	1,795	1,374	76.55	36.58	1,675	93.31	44.60
Total	**23,529**	**14,158**	**11,557**	**81.63**	**49.12**	**13,572**	**95.86**	**57.68**

Then we further investigate the contribution of *locality-driven matching* to binary source code matching. Specifically, we apply each newly matched source function from the phase of *locality-driven matching* to update the corresponding *top-1* similar function from *embedding-based function retrieval*. Figure 5 presents the original *recall@1* based on *embedding-based function retrieval*, the updated *recall@1* by adding the newly matched source functions with different *top-k* retrieved results as input of *locality-driven matching*, and the corresponding *recall@k* which means the upper bound of recall restricted by the capability of the model. Overall, the newly matched results by *locality-driven matching* significantly improve the original *recall@1* that almost reaches the upper bound for both BinaryAI and CodeCMR. For BinaryAI, *locality-driven matching* increases the *recall@1* from 22.73% to 54.70% with the upper bound as



(a) BinaryAI (b) CodeCMR

Figure 5: Contribution of locality-driven matching

57.35% for *top-10*, and further increase the *recall@1* to 66.90% with the upper bound as 70.45% for *top-100*. Similarly, *locality-driven matching* increases the *recall@1* from 11.92% to 28.61% with the upper bound as 33.46% for *top-10* in CodeCMR, indicating that as long as the model retrieves the exactly matched source function within the *top-k* results, *locality-driven matching* can effectively identify and update the matched source function as the new *top-1* result. Such a result is promising for improving binary source code matching, indicating that we can focus on enhancing the model capability of *top-k* retrieval in the future and leverage *locality-driven matching* to further identify the matched results. Overall, we can demonstrate the effectiveness of the two-phase design of binary source code matching to capture both syntactic and semantic code features in BinaryAI.

> ***Finding 4:*** *Locality-driven matching significantly increases the overall accuracy of binary source code matching, facilitating the downstream binary SCA.*

4.3.3 RQ3: Accuracy of TPL Detection. Lastly, we compare the performance of BinaryAI with the existing tools in terms of binary-to-source SCA. Table 3 demonstrates the overall result of TPL detection for 1,045 labeled components within 150 binary files. We can observe that, in general, BinaryAI significantly outperforms all the other SCA tools. For instance, BinaryAI significantly outperforms typical academic binary-to-source SCA techniques OSSPolice and B2SFinder. Furthermore, BinaryAI can even outperform well-recognized commercial binary SCA product bsca-B (85.84% vs. 73.36% precision, 64.98% vs. 59.81% recall, and 73.97% vs. 65.90% *F1*). Figure 6 presents the distribution of precision and recall for TPL detection across the 150 binary files. We observe that BinaryAI can dominate the precision and recall of component identification in most binary files, followed by the bsca-B. On the contrary, OSSPolice, B2SFinder, and bsca-A cannot generalize well to our SCA test cases with compromised precision or recall.

False Positive Analysis. We investigate 112 false positives from all 791 identified components which is rather limited in the domain of binary-to-source SCA, and find that all of them are related to the limitation of the TPL dependency. Specifically, there are overlapped function features between false positives and the correct TPLs, while we fail to filter out the false positives based on the TPL dependency generated by TPLite.

Table 3: Result of binary-to-source SCA

Tool	Verification of TPL Detection					
	#TP	#FP	#FN	P (%)	R (%)	F1 (%)
OSSPolice	348	191	697	64.56	33.30	43.94
B2SFinder	574	1232	471	31.78	54.93	40.26
bsca-A	232	108	813	68.24	22.20	33.50
bsca-B	625	227	420	73.36	59.81	65.90
BinaryAI	**679**	**112**	**366**	**85.84**	**64.98**	**73.97**

False Negative Analysis. We investigate all the false negatives, where most of them (312 out of 366) are caused by the partial reuse of the third-party components. In particular, the binary file only reuses a small fraction of functions from the labeled TPL, leading to a lower ratio than the pre-defined threshold θ. For instance, *nano_node* only reuses 8 functions from *leveldb* that causes the false positive. Note that the partial TPL reuse is generally the challenge of SCA [24, 60, 62], and other reasons for false negatives include missing the corresponding source functions due to decompilation errors and the capability of the model to retrieve similar functions.

> ***Finding 5:*** *BinaryAI dominates the performance of TPL detection among the state-of-the-art binary SCA tools.*

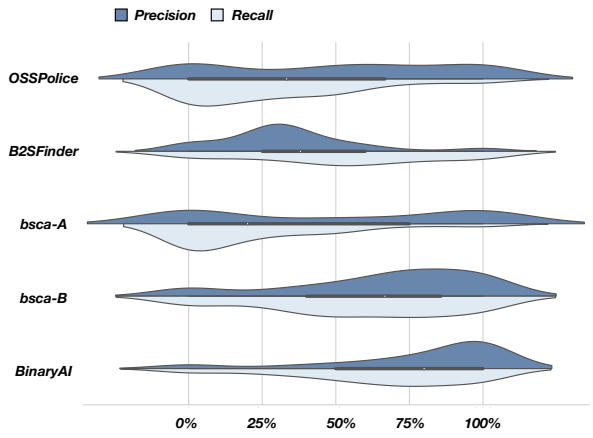


Figure 6: Distribution of binary SCA results

5 THREATS TO VALIDITY

Threats to internal validity. The threat to internal validity mainly lies in the design of BinaryAI. To reduce this threat, we have spent over two years exploring technical solutions, including training the model to directly learn structured code features (e.g., AST, control, or data flow information) as part of the embedding. However, their performance is far inferior to the current design of BinaryAI. Additionally, to reduce the threat of implementation, we invite three senior engineers in the relevant domain to review the code and ensure its correctness and consistency carefully.

Threats to external validity. The threat to external validity mainly lies in the subjects and dataset. To reduce this threat, we compare the model in BinaryAI with the state-of-the-art model CodeCMR in the domain of binary source code matching. For the downstream SCA task, we compare BinaryAI with four typical binary SCA tools from both industry and academia (OSSPolice, B2SFinder). For the dataset, we follow previous works [56, 57, 60] to construct substantial binary source function pairs as the training dataset by building an automatic compilation pipeline for ArchLinux packages. Meanwhile, we collect a large-scale TPL dataset and create the largest corpus of source functions respectively in binary-to-source SCA and binary source code matching. Considering the lack of publicly available ground-truth data, it also takes the authors excessive manual effort to calibrate the ground-truth components and label the correspondence of functions.

Threats to construct validity. The threat to construct validity mainly lies in the adopted metrics in our evaluation. To reduce this threat, we strictly follow prior works [24, 55, 57, 60, 70] to evaluate BinaryAI with multiple widely-used metrics, i.e., *MRR*, *Recall@k*, *Precision*, *Recall*, and *F1* score.

6 RELATED WORK

6.1 Software Composition Analysis

Many existing SCA techniques based on binary analysis [21, 25, 34, 50, 63–65] employ various feature extraction approaches and matching algorithms to improve the accuracy of TPL detection. B2SFinder [71] extracts control-flow structures to capture the target program's semantic information and allocates weight to different features. ModX [69] takes a modularization approach that clusters functions into semantically-based modules. Xu et al. [68] propose a multi-level birthmark model that extracts program features on three levels to deal with obfuscation. Tang et al. [55] propose LibDB, which utilizes syntactic and function embedding features. It also filters out duplicated features with the assistance of function call graphs. OSSPolice [13] adopts a hierarchical indexing scheme to locate true matches. Multiple SCA works are designed to operate in a source-to-source setting. SourcererCC [49] utilizes an optimized partial index and filtering heuristics to detect open-source code clones. Lopes et al. [38] further adopt SourcererCC to construct a duplicate code map called DéjàVu for the code repositories on GitHub. Centris [60] takes function signature as the basic feature and derives TPL dependencies based on function birth time to alleviate internal code clones. TPLite [24] utilizes hierarchical path information to identify the origin TPL and centrality analysis to filter out false positives.

In addition to the C/C++ ecosystem, many SCA techniques are specifically designed to identify components of Android applications. ATVHunter [73] takes a two-phase approach which uses control-flow graphs as features in the first stage and the opcode of control-flow graphs in the second stage. LibScout [7] employs class hierarchy information that does not rely on concrete code to improve resilience against code obfuscation. Zhang et al. [74] propose LibID to cope with a wider range of obfuscation scenarios in Android. It leverages class signatures as features and a three-stage matching scheme to ensure that a library exists. In this paper, we propose BinaryAI, which is the first to adopt a transformer-based model in the domain of binary-to-source SCA. Additionally, BinaryAI leverages link-time locality to enhance the accuracy of binary source code matching and the downstream SCA task.

6.2 Code Clone Detection

Code clone detection is widely used to evaluate code similarities among software projects. It can be divided into two categories: binary-to-source code matching and binary-to-binary code matching. In the binary-to-source matching level, CodeCMR [70] adopts DPCNN for source code feature extraction and GNN for binary code feature extraction. Many existing works are in the binary-to-binary matching level. Asm2Vec [11] is an assembly code representation learning model, which produces a vector representation for each assembly function in the repository and uses cosine similarity to retrieve the top-k ranked candidates as results. InnerEye [76] proposes a cross-lingual deep learning approach at the basic-block level. jTrans [57] presents a transformer-based approach that uses a jump-aware representation of the analyzed binary code and a newly-designed pre-training task to embed the control flow information into the language model. Based on graph representation learning, Xu et al. [67] propose Gemini, a neural network-based approach, which applies Structure2vec to embed the attributed control flow graph (ACFG) and then measures the distance between the embeddings for two functions. Kim et al. [28] present XBA, a deep learning-based technique, which first abstracts binary disassembly graphs (BDGs), and then formulates the binary code representation learning as a graph alignment problem. DeepBinDiff [33] relies on both the code semantic information distilled by NLP techniques and program-wide control flow information to generate embeddings at the basic block level. VulHawk [39] proposes an intermediate representation function model with NLP techniques and graph convolutional networks to generate function embeddings and an entropy-based adaptor to alleviate the differences caused by various file environments in function embeddings.

7 CONCLUSION

In this paper, we propose a novel binary-to-source SCA technique, BinaryAI, to alleviate the problem that existing binary-to-source SCA techniques suffer from redundancy in the large-scale TPL dataset and few syntactic features between reused TPLs and target binary files. BinaryAI trains a transformer-based model to generate function embeddings by learning the token-based syntactic feature of the code language and leverages locality-driven matching to enrich semantic features for further identifying the positive samples. Based on the matched source functions, BinaryAI performs SCA by detecting the reused TPL components in the target binary file. The evaluation results indicate that the embedding model significantly outperforms CodeCMR with 22.54% recall@1 and 0.34 *MRR*. Additionally, BinaryAI with *locality-driven matching* can further improve the binary source code matching and its TPL detection result exceeds all state-of-the-art binary-to-source SCA tools.

ACKNOWLEDGEMENT

This work is partially supported by the National Natural Science Foundation of China (Grant No. 62372220). Ling Jiang would like to dedicate this paper to the love of his fiancée.

REFERENCES

[1] 2012. The DWARF Debugging Standard. https://dwarfstd.org.

[2] National Security Agency. 2023. Ghidra Software Reverse Engineering (SRE) Framework. https://ghidra-sre.org.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*. Association for Computational Linguistics, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[4] Amrita Pathak. 2022. Software Composition Analysis (SCA): Everything You Need to Know in 2022. https://geekflare.com/software-composition-analysis.

[5] Archlinux. 2021. Arch linux. https://archlinux.org/packages/.

[6] Archlinux. 2021. Arch User Repository. https://aur.archlinux.org/.

[7] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 356–367. https://doi.org/10.1145/2976749.2978333

[8] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*. PMLR, 2397–2430.

[9] Harold Booth, Doug Rike, and Gregory Witte. 2013. The National Vulnerability Database (NVD): Overview. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=915172

[10] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. https://doi.org/10.1145/3428293

[11] 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. https://doi.org/10.1109/SP.2019.00003

[12] Eelco Dolstra, Eelco Visser, and Merijn De Jonge. 2004. Imposing a memory management discipline on software deployment. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 583–592. https://doi.org/10.1109/ICSE.2004.1317480

[13] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2169–2185. https://doi.org/10.1145/3133956.3134048

[14] EleutherAI. 2023. Pythia-410M. https://huggingface.co/EleutherAI/pythia-410m.

[15] EVM. 2018. A Code Pirate's Cutlass, Recovering Software Architecture from Embedded Binaries. REcon 2018.

[16] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 516–527. https://doi.org/10.1145/3395363.3397362

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*. 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[19] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 9729–9738.

[20] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 63–72. https://doi.org/10.1145/3468744.3468752

[21] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627.

[22] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive Code Representation Learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 5954–5971. https://doi.org/10.18653/v1/2021.emnlp-main.482

[23] Ang Jia, Ming Fan, Wuxia Jin, Xi Xu, Zhaohui Zhou, Qiyi Tang, Sen Nie, Shi Wu, and Ting Liu. 2023. 1-to-1 or 1-to-n? Investigating the Effect of Function Inlining on Binary Similarity Analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–26.

[24] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-Party Library Dependency for Large-Scale SCA in the C/C++ Ecosystem: How Far Are We?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3551349.3560432

[25] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and improving hybrid fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 410–422.

[26] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[27] Julie Peterson. 2021. Software Composition Analysis Explained. https://www.mend.io/resources/blog/software-composition-analysis.

[28] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 2022. Improving cross-platform binary analysis using representation learning via graph alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 151–163. https://doi.org/10.1145/3533767.3534383

[29] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614. https://doi.org/10.1109/SP.2017.62

[30] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 4 (1994), 1117–1155.

[31] 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. https://doi.org/10.1109/ICSE.2017.38

[32] Siyuan Li, Yongpan Wang, Chaopeng Dong, Shouguo Yang, Hong Li, Hao Sun, Zhe Lang, Zuxin Chen, Weijie Wang, Hongsong Zhu, and Limin Sun. 2023. LibAM: An Area Matching Framework for Detecting Third-party Libraries in Binaries. *ArXiv* abs/2305.04026 (2023). https://api.semanticscholar.org/CorpusID:258557875

[33] Xuezixiang Li. 2019. Learning Program-Wide Code Representations for Binary Diffing. *Proceedings 2020 Network and Distributed System Security Symposium* (2019).

[34] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 627–637.

[35] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-Version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. 667–678. https://doi.org/10.1145/3238147.3238199

[36] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 672–684. https://doi.org/10.1145/3510003.3510142

[37] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning Graph-based Code Representations for Source-level Functional Similarity Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 345–357. https://doi.org/10.1109/ICSE48619.2023.00040

[38] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28. https://doi.org/10.1145/3133908

[39] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In *Proceedings of the 2023 Network and Distributed Systems Security Symposium (NDSS)*.

[40] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium (USENIX Security 22)*. 2099–2116.

[41] Dhaval Miyani, Zhen Huang, and David Lie. 2017. Binpro: A tool for binary source code provenance. *arXiv preprint arXiv:1711.00830* (2017).

[42] James Munkres. 1957. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics* 5, 1 (1957), 32–38.

[43] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PMLR, 8748–8763.

[44] Red Hat. 2022. What is software supply chain security? https://www.redhat.com/en/topics/security/what-is-software-supply-chain-security.

[45] Github Repository. 2023. The LLVM Compiler Infrastructure. https://github.com/llvm/llvm-project.

[46] Github Repository. 2023. Tree-sitter, a parser generator tool and incremental parsing library. https://github.com/tree-sitter/tree-sitter.

[47] Github Repository. 2023. Zlib data compression library. https://github.com/madler/zlib.

[48] Sascha Rothe, Jonathan Mallinson, Eric Malmi, Sebastian Krause, and Aliaksei Severyn. 2021. A Simple Recipe for Multilingual Grammatical Error Correction. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. Association for Computational Linguistics, 702–707. https://doi.org/10.18653/v1/2021.acl-short.89

[49] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168. https://doi.org/10.1145/2884781.2884877

[50] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[51] Xin Sun, Tao Ge, Shuming Ma, Jingjing Li, Furu Wei, and Houfeng Wang. 2022. A Unified Strategy for Multilingual Grammatical Error Correction with Pre-trained Cross-Lingual Language Model. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*. 4367–4374. https://doi.org/10.24963/ijcai.2022/606

[52] Synopsys. [n. d.]. What is software composition analysis. https://www.synopsys.com/glossary/what-is-software-composition-analysis.html.

[53] Synopsys. 2023. Black Duck Binary Analysis (BDBA). https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis/binary-analysis.html.

[54] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 104–115. https://doi.org/10.1109/SANER48275.2020.9054845

[55] 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 *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 423–434. https://doi.org/10.1145/3524842.3528442

[56] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. 2022. Towards Understanding Third-party Library Dependency in C/C++ Ecosystem. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12. https://doi.org/10.1145/3551349.3560432

[57] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. JTrans: Jump-Aware Transformer for Binary Code Similarity Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13. https://doi.org/10.1145/3533767.3534367

[58] Lilian Weng. 2021. Contrastive Representation Learning. *lilianweng.github.io* (May 2021). https://lilianweng.github.io/posts/2021-05-31-contrastive/

[59] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3037–3053. https://www.usenix.org/conference/usenixsecurity22/presentation/woo

[60] 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. https://doi.org/10.1109/ICSE43902.2021.00083

[61] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).

[62] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. 2023. OSSFP: Precise and Scalable C/C++ Third-Party Library Detection using Fingerprinting Functions. In *Proceedings of the 45th International Conference on Software Engineering*.

[63] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. 2023. Enhancing Coverage-Guided Fuzzing via Phantom Program. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1037–1049.

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

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

[66] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. 1106–1118. https://doi.org/10.1145/3597926.3598121

[67] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 363–376. https://doi.org/10.1145/3133956.3134018

[68] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-enabled software reuse detection based on a multi-level birthmark model. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 873–884. https://doi.org/10.1109/ICSE43902.2021.00084

[69] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering*. 1393–1405. https://doi.org/10.1145/3510003.3510627

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

[71] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. 2019. B2sfinder: detecting open-source software reuse in cots software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1038–1049. https://doi.org/10.1109/ASE.2019.00100

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

[73] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1695–1707. https://doi.org/10.1109/ICSE43902.2021.00150

[74] Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. 2019. Libid: reliable identification of obfuscated third-party android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 55–65. https://doi.org/10.1145/3293882.3330563

[75] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[76] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*.