Decompilation Based Deep Binary-Source Function Matching

Xiaowei Wang[1,2,3,4][0009−0006−9824−2543], Zimu Yuan[1,2,3,4][0000−0002−9494−7478], Yang Xiao[1,2,3,4][0009−0005−8009−2252], Liyan Wang[1,2,3,4], Yican Yao[1,2,3,4][0009−0001−2194−3230], Haiming Chen[5][0000−0002−2659−4148], and Wei Huo[1,2,3,4][0009−0000−7121−1196]

[1] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
{wangxiaowei, yuanzimu, xiaoyang, wangliyan, yaoyican, huowei}@iie.ac.cn
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[3] Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences
[4] Beijing Key Laboratory of Network Security and Protection Technology
[5] Institute of Software, Chinese Academy of Sciences, Beijing, China chm@ios.ac.cn

Abstract. Binary and source matching is vital for vulnerability detection or program comprehension. Most existing works focus on library matching (coarse-grained) by utilizing some simple features. However, they are so coarse-grained that high false positives occur since developers tend to reuse source code library partly. These shortcomings drive us to perform fine-grained matching (i.e., binary and source function matching). At the same time, due to the enormous differences between the form of binary and source functions, function matching (fine-grained) meets huge challenges. In this work, inspired by the decompilation technique and advanced neural networks, we propose tool, a **D**ecompilation based deep **B**inary-**S**ource function **M**atching framework. Specifically, we take the triplet features from both *binary pseudo-code* and *source code* functions as input, which are extracted from code property graph and can represent both the syntactic and semantic information. In this way, the binary and source functions are represented in the same feature space so to ease the matching model to learn function similarity. For the matching model, we adopt a self-attention based siamese network with contrastive loss. Experiments on two datasets, $R0$ and $R3$, show that our tool achieves consistent improvements than other methods, which demonstrate the effectiveness of our self-attention based matching model, and our triplets features can well capture the two kinds of code functions. Our work improves the accuracy of binary and source code matching, which in turn enables us to better address security issues such as vulnerability detection and program comprehension.

Keywords: Function Matching · Binary-Source Function Matching · Deep Learning.

1 Introduction

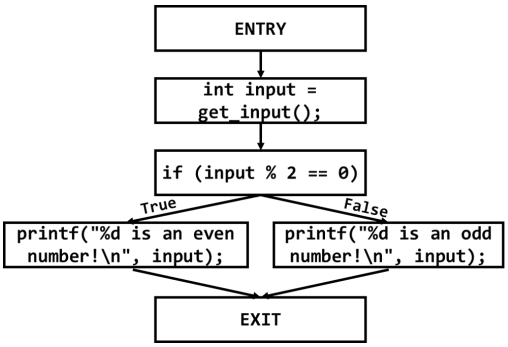
Binary-source matching is an important task in computer science, which can be applied in various applications such as vulnerability detection [9], plagiarism detection [29], malware detection [30] and reverse engineering [32].

Existing works mainly focus on binary-source library matching [9,14,16]. These works extract a part of constant features which are not changed in the process of compiling from both source and binary projects. These features include strings, integers, export functions, string arrays, integer arrays and so on. Then, various matching algorithms (such as TF-IDF [33] based) are applied on these features to match binary and source projects. However, there are two main disadvantages of binary-source library matching works. First, these features may not exist in binary projects with removing features intentionally, which lead to false positives and false negatives. Second, they are too coarse-grained and these features are not adequate for fine-grained (function level) matching, while it is important.

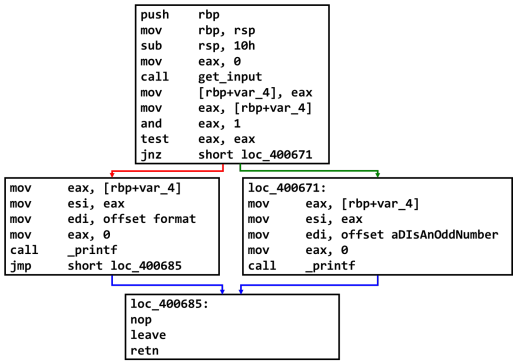
Listing 1.1. Example source code function

```
void even_or_odd()
{
    int input = get_input();
    if (input % 2 == 0)
        printf("%d is an even number!\n", input);
    else
        printf("%d is an odd number!\n", input);
}
```

Therefore, we try to focus on more fine-grained matching, i.e., binary-source function matching. However, binary-source function matching meets huge challenges since the form of binary and source code functions are totally different. On the one hand, binary function is represented by machine code, assembly code or control flow graph (CFG), while source code function is represented by statements, abstract syntax tree (AST), CFG or program dependency graph (PDG). On the other hand, even both functions are represented by CFGs, the nodes in the graph are different. Figure 1(a) and Figure 1(b) show CFGs of source code function and corresponding binary function in Listing 1.1. Node in Figure 1(a) is a statement while node in Figure 1(b) is composed by a sequence of assembly code. It is not easy to match a statement to a sequence of assembly code without debug information. Besides, the number of nodes in two functions' CFG are not same, which means we do not know how to map nodes in source to nodes in binary, since one node in source can map to two or more nodes in binary, and vice versa. For example, the second and the third node in Figure 1(a) are equivalent on semantic of the first node in Figure 1(b). At the same time, if the conditional check statement in source function is composed by two or more conditions, there would be more than one node in the CFG of binary function while one node in the CFG of source code function. In details, the number of conditions in one conditional check of source function corresponds to the number of nodes in binary's CFG. Therefore, how to select the features to present the



(a) CFG of example source code function.



(b) CFG of example binary function.

Fig. 1. An example of CFG for source and binary function.

source and binary functions is important and not easy, which makes it to be a huge challenge to perform the function matching between source and binary.

Listing 1.2. Example pseudo-code function

```
int even_or_odd()
{
  int result; // eax
  unsigned int v1; // [rsp+Ch] [rbp-4h]

  v1 = get_input();
  if ( v1 & 1 )
    result = printf("%d is an odd number!\n", v1);
  else
    result = printf("%d is an even number!\n", v1);
  return result;
}
```

In this paper, to solve above issues, we propose a Decompilation based deep Binary-Source function Matching (DBSM) framework, which is built upon the function-level matching and needs no compilation from source code projects into binaries. Specifically, our approach consists of the following steps. *Step-1:* Binary functions are lifted to pseudo-code with the aid of IDA-Pro[6] toolkit. For example, Listing 1.2 is the resulted pseudo-code form of the binary function in Figure 1(b). Obviously, pseudo-code is now more similar to the source code (shown in Listing 1.1) compared with the original assembly code. *Step-2:* Given the source code and pseudo-code function as the input, we apply a robust parser Joern [13] to parse the codes and generate a code property graph which merges AST, CFG and PDG into a joint data structure for each code. *Step-3:* Triplets are extracted from code property graph of source code and pseudo-code function, which can represent the syntax and semantic information. Triplets are represented as $(src, path, tgt)$, where src and tgt are nodes in the code property graph while $path$ is the relationship between src and tgt. *Step-4:* Above triplet features of source code and pseudo-code function are finally fed into a siamese network for function similarity matching. Specifically, we design a self-attention mechanism based siamese neural network, and we adopt the advanced contrastive loss for the matching model training.

We conduct experiments on two datasets, $R0$ and $R3$, for the task evaluation. These datasets are collected from fifteen real world source code projects with compiling them manually in two different settings, which can be useful and typical resources to evaluate our DBSM. The results show that DBSM achieves strong performances and significantly outperforms other baseline models. Specially, we obtain a high 89.6 Recall@1 on $R0$ and 82.2 on $R3$.

In a summary, our contributions are as follows:

- We propose a binary and source function matching framework DBSM by leveraging triplets of source code and pseudo-code, which is lifted from binary assembly.
- We extract triplets from AST, CFG and DDG that can represent the syntax and semantic information for each code function.

[6] https://www.hex-rays.com/products/ida/

- We build a self-attention based siamese neural network, and apply the contrastive loss on the triplets for the function similarity matching.
- Experiments on two datasets show that our `DBSM` method can obtain superior performance. Especially, `DBSM` achieves a high Recall@1 score of 89.6 and 82.2 on $R0$ and $R3$.

2 Related Work

Our work is related to the binary-source matching on both library level and function level. In this section, we briefly introduce these related works.

2.1 Library Matching

It is very common for software developers to integrate source code projects into target software, which can accelerate the development. Developers must comply with the license of source code projects, and vulnerabilities in these projects may affect target software. At the same time, most target software are binaries (close sourced). Therefore, there are many works trying to detect source code projects in binaries.

Three techniques (detection using strings, detection using compression and detection using binary deltas) are proposed in BAT [16] to detect source code projects in target binaries. OSSPolice [14] selects string literal and exported function as features and applies hierarchical matching algorithm to detect source code projects in android app binaries. B2SFinder [9] chooses seven features which are not changed drastically during the compilation and employs a weighted feature matching algorithm to achieve the task. LibID [15] converts library identification problem into a Binary Integer Programming (BIP) problem.

These works can detect source code projects used in target binaries precisely and efficiently. However, these methods are too coarse-grained since software developers may reuse part of source code projects or even some functions only. Moreover, these features are not suitable for function-level matching since there are not these features in many functions. Therefore, we need to extract more fine-grained features which can represent function.

2.2 Function matching

There are three types of function matching: binary-binary, source-source and binary-source matching.

There are tens of works proposed for solving binary-binary function matching problems. INNEREYE-CC [17] computes similarity beyond function pairs inspired by neural machine translation. DEEPBINDIFF [18] proposes an unsupervised program-wide code representation learning technique for binary diffing with extracting semantic information by leveraging NLP techniques and performing TADW algorithm to generate basic block embeddings. α-diff [19] employs

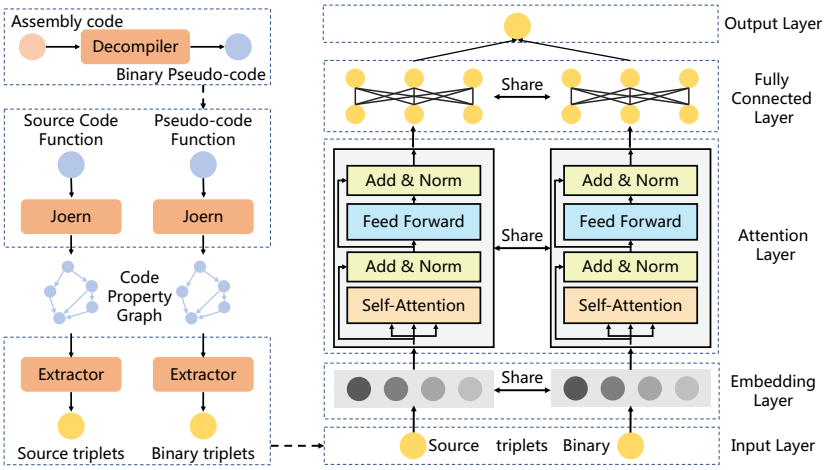


Fig. 2. The overview of our DBSM framework.

three semantic features (intra-function features, in/out degree in call graph features and imported function features) to address function similarity problem with DNN model [37]. These works require users to compile all source code projects into binaries first, which is very time-consuming or even impossible. For one hand, only approximately quarter can be compiled automatically. [9] For the other hand, it is impossible to compile old or complex source code projects manually since it relies on compiling environment deeply.

For source-source function matching, Tree-based convolutional neural network [21] is used for classifying programs according to functionality. CCD [22] introduces a joint code representation that applies fusion embedding technique to learn hidden syntactic and semantic features and takes DNN as a classifier to detect code clone. TreeCaps [23] fuses capsule networks with tree-based convolutional neural networks together for code clone detection task. Different from these code clone works on source code only, we focus on function matching between binary and source code. There is not rich syntactic and semantic information in binary function.

As for binary-source function matching, to the best of our knowledge, there is only one work focus on the task. CodeCMR [20] takes function matching between binary and source code as an end-to-end cross-modal retrieval task. It adopts DPCNN (resp. GNN) for source code (resp. binary) feature extraction and exploits LSTM to capture two kinds of code literals. In a word, CodeCMR utilizes four different neural networks for the task, bug it is very complex.

3 Methodology

Figure. 2 shows the overview of DBSM, which contains four modules. The *Decompilation* module (§ 3.2) lifts assembly code to pseudo-code. The *Code Property*

Graph Generation module (§ 3.3) takes source and binary pseudo-code functions as inputs and generates code property graph with the aid of robust parser Joern. The *Feature Extraction* module (§ 3.4) takes code property graph of binary and source functions as input, and generates a list of triplets as features for each function. The *Self-attention based Matching Model* module (§ 3.5) takes features of binary and source code function as inputs, and trains a model to match binary and source code functions.

3.1 Problem formulation

Existing works focus on binary to source library matching (coarse-grained) or function matching at binary level only. Instead, we perform binary to source function matching, which is more fine-grained. The binary and source function matching task is defined as a matching problem. We first describe the necessary notations used in this paper for our `DBSM` approach.

We define the input dataset $D = \{(f_i^s, f_i^b; y_i)\}$, $i \in \{1, 2, ..., N\}$, where the $f_i^s \in \mathcal{F}^s$ and $f_i^b \in \mathcal{F}^b$ represents the i-th sample of (source feature, binary feature) pair from the source function feature space $\mathcal{F}^s$ and binary function feature space $\mathcal{F}^b$, and N is the number of samples in the dataset D. Here $y_i \in \{0, 1\}$ is the label for the feature pair (f_i^s, f_i^b), where 1 means the two function features are a matched pair while 0 is not. As we will introduce later, each function feature f_i (i.e., f_i^s or f_i^b) is represented by a list of triplets, that is, $f_i = [t_j]_{j=1}^M$, $t_j = (src, path, tgt)$, where $src, path, tgt$ is the source, path and target node in each triplet t_j, and M is the total number of triplet list for the function feature f_i. With above definitions, the classification of `DBSM` is to learn a function mapping from $\mathcal{F}$ to label $\{0, 1\}$. We denote the mapping (matching) function as g, and the corresponding learning objective is to minimize the loss function over g as:

$$\min \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(g(f_i^s, f_i^b; y_i)), \tag{1}$$

where $\mathcal{L}(\cdot)$ can be cross-entropy loss function or other loss functions. In our paper, we use the contrastive loss [38] that will be introduced in Section 3.5.

3.2 Decompilation

Binary functions are usually represented as machine code, assembly code or CFG. However, much high-level (semantic) information is lost when compiling source code to binary. In other words, there are enormous differences between binary and source code functions, such as the form. Fortunately, binary analysis tool such as IDA-Pro[7] or Ghidra[8] can decompile assembly code to pseudo-code (decompilation), which means the high-level information can be recovered in some degree. For example, Listing 1.2 is the result of decompilation from Figure 1(b)

[7] `https://www.hex-rays.com/products/ida/`
[8] `https://ghidra-sre.org/`

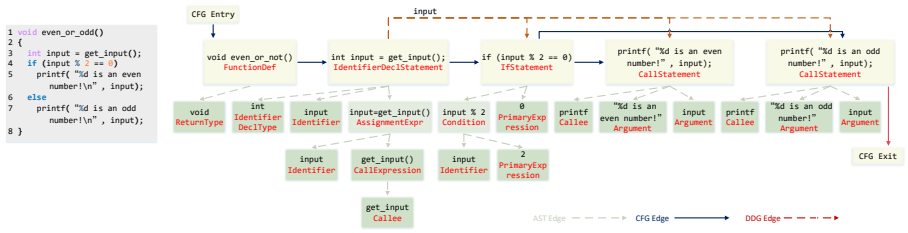


Fig. 3. An example of the Code Property Graph for the function.

by IDA-Pro. Intuitively, it is now much easier to match source function and pseudo-code function since the latter is more similar to the former than other binary function forms.

3.3 Code Property Graph Generation

Existing works perform program analysis on source code with various levels, such as token-level, statement-level, function-level, or different representations, such as AST, CFG, PDG. The code property graph (a joint data structure includes AST, CFG and PDG) is proved to be useful in many applications such as vulnerability detection [24,13,25], because the syntactic and semantic relationships can be captured by the code property graph. In details, AST represents the relationship of tokens, which shows the syntactic information, while CFG and PDG represents the relationships among statements, which show the semantic information.

Given the source or pseudo-code function as the input, we apply a robust parser Joern [13] to parse the code and generate the corresponding code property graph. Figure 3 is the code property graph of Listing 1.1. The nodes with color light yellow are the statement (CFG and DDG nodes), and nodes with color light green are AST nodes, where the darker is leaf nodes and the brighter is non-leaf nodes. The solid lines are CFG edges, and the dotted lines with color brown (resp. light green) are DDG (resp. AST) edges.

3.4 Feature Extraction

From the code property graph, we are now able to extract a list of triplets i.e., $(src, path, tgt)$, where src and tgt are nodes and $path$ is the edge in the graph. For nodes in CFG and DDG, each edge and its corresponding nodes are extracted as triplets. The value of nodes is the *attributes* (highlighted with the red color in the node). For non-leaf nodes in AST, only the *attributes* are taken into consideration, too. While for leaf nodes in AST, both the *attributes* and *values* (highlighted with the black color in the node) are considered. The value of $path$ depends on what the edge belongs to, which can be ast_x, cfg_x and ddg_x, where x represents the count of same triplet $(src, path, tgt)$. For example,

if there are two same triplet $(s1, ast, t1)$, then we aggregate them as $(s1, ast2, t1)$. With proposed strategy, syntactic information and semantic information are all covered. In addition, features used in library matching (such as strings and integers) are also extracted.

We take Figure 3 as an example to clearly illustrate the feature extraction procedure. The triplet `(IdentifierDeclStatement, ddg2, CallStatement)` is extracted since there are two same triplets `(IdentifierDeclStatement, ddg, CallStatement)` in DDG, which means that two `CallStatements` use the same value assigned at `IdentifierDeclStatement`. It depicts the flow of data from one statement to the others, which represent the semantic information. In AST, the triplet `(FunctionDef, ast1, ReturnType)` is extracted from the most left two nodes. Since these two nodes are non-leaf nodes, only the value of attributes are taken into consideration. For leaf nodes in AST, the triplet `(Argument, ast1, "%d is an odd number!")` is extracted, which represents that there is a string literal named `"%d is an odd number!"` in function. The string feature is used in binary-source library matching, which contributes a lot.

3.5 Self-attention based Siamese Network

We utilize the siamese network structure [40] to capture the relation between the two functions. Siamese network has been proven to be effective to model the similarity between the two inputs from close modalities. It usually contains two shared left and right branches to process the inputs. In our work, specifically, the triplet features of the source code and the pseudo (binary) code will be processed by a shared model (as shown in the right part of Figure 2) for the final matching learning.

To process the triplet features, we carefully choose outstanding deep learning models. Inspired from the outstanding performances achieved by the self-attention [34] based networks, such as Transformer [26] in natural language processing and computer vision, we adopt the self-attention mechanism to build a siamese network so as to model the triplet features for our matching model. We first feed the list of triplets as input and map them into low-dimensional embedding vectors. Then they will be processed by the self-attention layer to learn the representations. Finally the representations of the two code functions are put into the last layer for matching. The detailed descriptions of each module are in the follows.

Embedding Layer. The embedding layer is designed to transform the input elements into fixed low-dimensional vectors. As we introduced before, the triplet feature is consisted of three elements, i.e., $(src, path, tgt)$, and the src and tgt elements are mapped into a same embedding space $E_{st} \in \mathcal{R}^{V_{st} \times d}$, where V_{st} is the vocabulary size of the src and tgt elements, d is the size of the embedding vector. Similarly, the $path$ element is mapped into embedding space $E_p \in \mathcal{R}^{V_p \times d}$, where V_p is the vocabulary size of all $path$ elements. For each triplet feature t_j, we denote the mapped embedding vectors as $(e^j_{src}, e^j_{path}, e^j_{tgt})$, then we simply concatenate the three embeddings as the embedding vector for

the triple feature, $e_{t_j} = [e^j_{src}; e^j_{path}, ; e^j_{tgt}] \in \mathcal{R}^{3d}$. Therefore, for one function f_i (see Section 3.1), the list of number of M triplet features are embedded by $E_i = [e_{t_1}, e_{t_2}, ..., e_{t_M}] \in \mathcal{R}^{M \times 3d}$.

Following [26], we also add positional embedding to capture the relative order information of triplets in the function feature. Specifically, each position has its own position embedding $E_p = [p_1; p_2; ...; p_N] \in \mathcal{R}^{N \times d}$, and the position embeddings are learnt with other model parameters. The position embedding is summed up with the triplet feature embedding E_i to formulate the final input representation of the model.

Attention Layer. The attention layer contains a self-attention sub-layer and a feed-forward sub-layer.

The self-attention [34], or called intra-attention, is an attention mechanism designed to relate different positions of a single sequence in order to compute the representation of the sequence. In our scenario, we regard the list of the triplet features as a sequence. We follow the introduction in [26] to build the scaled dot-product self-attention layer. Specifically, it is computed by:

$$\texttt{attn}(Q, K, V) = softmax(\frac{(QW_Q)(KW_K)^T}{\sqrt{d}})(VW_V), \tag{2}$$

where Q, K, V represent the Query, Key and Value matrices, which are the list of triplet feature representations outputted by the above introduced embedding layer. The W_Q, W_K, W_V are the three learnable parameter matrices. The scaled item $\sqrt{d}$ is used to prevent the dot-products growing large in magnitude.

After the self-attention, we add a position-wise feed-forward sub-layer to further process non-linear activation:

$$\texttt{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2, \tag{3}$$

where W_1, W_2 and b_1, b_2 are the corresponding weight matrices and bias vectors. The input feature x is the feature output from the attention sub-layer.

Note that for each sub-layer, we also apply a residual connection [35] and a layer normalization [36] operation. Therefore, to make a summary, suppose the input feature to the attention layer is x and the output feature is $\hat{x}$, the whole computation is:

$$\begin{aligned} \hat{x} &= \texttt{LN}(\texttt{attn}(x) + x), \\ x &= \texttt{LN}(\texttt{FFN}(\hat{x}) + \hat{x}), \end{aligned} \tag{4}$$

where LN is the layer normalization operation.

Output & Loss After obtaining the representations for each function features from the attention layer, we now build one last fully-connected layer to transform them into hidden states h^s (source code) and h^b (binary code), and finally we perform the similarity matching.

To train this similarity matching network, we can choose different loss functions, for example, the cross-entropy loss function [39], the contrastive loss function [38]. Here we adopt the contrasitve loss function built on the vector distance.

Table 1. The statistics about the open source projects we collected for our dataset $R0$ and $R3$. "LoC" stands for the lines of code in each function, "Avg. Source" means the average number for the source code statistics. The symbols "△" and "♢" indicate the average number for the binary $R0$ and $R3$, respectively.

Projects	Domain	# Functions	LoC (Avg. Source)	LoC (△)	LoC (♢)
Zlib	Compression	13	49.3	61.9	50.3
UnRAR	Compression	120	98.2	103.0	101.4
Expat	XML Parser	371	70.0	76.1	76.3
Jasper	Image Processing	760	66.1	68.7	69.3
LibPNG	Image Processing	867	87.0	94.6	93.9
TCPdump	Packet Analyser	901	94.6	96.1	108.2
OpenJPEG	Image Processing	1615	79.8	82.5	85.1
LibTIFF	Image Processing	1923	90.5	92.2	93.5
FreeType2	Font Processing	4289	116.9	125.0	128.4
OpenSSL	Encryption	4620	81.2	84.3	87.2
VLC	Video Processing	5139	61.5	63.4	65.0
Sqlite	Database	6489	167.0	174.6	197.6
CURL	Network	7047	177.7	183.3	202.0
ImageMagick	Image Processing	10557	255.5	242.3	275.8
Radare2	Binary Analyser	19201	90.6	90.6	90.6
Total	-	63912	-	-	-

Essentially, contrastive loss is evaluating how good a job the siamese network is distinguishing between the function pairs, which is acknowledged to be a better learning objective. Specifically, the loss function is:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(\frac{1}{2}(y_i * d_i^2 + (1-y_i) * \max\{(m-d_i), 0\}^2)), \tag{5}$$

where $y_i \in \{0, 1\}$ is the label of for i-th function pair, d_i is the Euclidean distance between the corresponding two features h_i^s and h_i^b, and m is the margin value used to control the learning distance.

4 Experiments

4.1 Datasets Preparation

To verify the effectiveness of `DBSM` in matching source code and binary functions, we conduct experiments on two dataset $R0$ and $R3$ collected by ourselves. Concretely, the datasets are compiled from real popular open source projects. We chose open source projects that satisfies the following criteria. First, they are implemented by C/C++ language since `DBSM` is designed to match functions between C/C++ source code projects and corresponding binaries. Second, they need to cover diverse application domains so that the generality of `DBSM` can

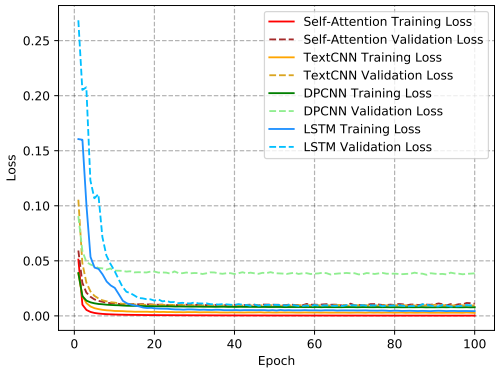
Table 2. Experiments results on two datasets $R0$ and $R3$.

Model	GCC-x64-O0 ($R0$)			GCC-x64-O3 ($R3$)		
	Recall@1	Recall@3	Recall@5	Recall@1	Recall@3	Recall@5
Raw code + TextCNN	56.3	71.4	75.3	44.7	55.9	60.0
Raw code + DPCNN	51.1	70.6	76.5	32.5	50.1	57.8
Raw code + Attention	63.7	79.9	84.3	42.1	57.6	64.3
Triplets + LSTM	83.2	94.0	95.4	77.3	90.7	93.1
Triplets + TextCNN	87.5	96.8	97.7	81.2	93.9	95.5
Triplets + DPCNN	78.2	95.1	96.9	69.6	89.9	93.2
Our model	**89.6**	**97.4**	**98.0**	**82.2**	**94.2**	**95.9**

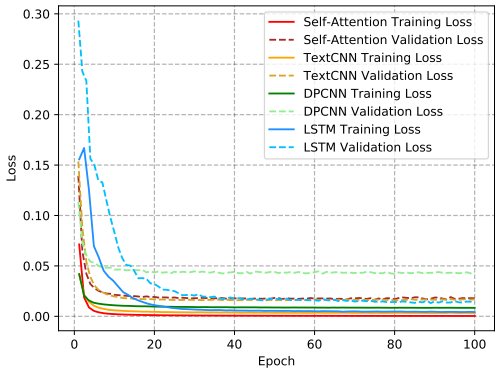
be evaluated. With above two criteria, we chose fifteen open source projects. Table 1 summarizes their statistics. The application domains we cover include compression, xml parser, image processing, packet analyser, font processing, encryption, video processing, database, network, binary analyser, which should be diverse enough to show the generalization ability of our `DBSM`. After collecting above open source projects, we compile them to binaries manually with different optimization levels, and we remove duplicated functions to save the unique function instance. Then, with the aid of IDA-Pro toolkit, each pseudo-code function for the binary code is extracted and mapped to the corresponding source code function. In total, we finally collect 63,912 pairs of functions. The average LoC in the source code functions range from 49 to 255, while the number of functions ranges from 13 to 19,201. We take binaries compiled with compiler GCC, architecture x64 (64 bits) and optimization level O0 as dataset $R0$, similarly for the optimization level O3 as dataset $R3$. After processing the data, the vocabulary size of the V_{st}/V_p for $R0$ is about $315k/218k$, while for R_3 is $296k/248k$. We randomly split each dataset to be training set, validation set, and test set, and there are about 80% of pairs of functions as training data (51,130 pairs), 10% as validation data (6,391 pairs) and the rest 10% as testing data (6,391 pairs).

4.2 Baseline Methods

To show the effectiveness of our proposed method `DBSM`, we compare with several different baseline methods. As we introduced before, since there are almost no works studying the function level matching problem, we design baseline methods from different aspects, in order to fully test the effectiveness of each component (i.e., triplet features and attention model) in our framework. Specifically, the compared baselines including the following types: (1) The binary transformed pseudo code function and source code function are both fed into the network with their raw code representations, without any preprocessing and feature extractions. (2) The binary transformed pseudo code function and the source code function are both processed by our `DBSM` method. Besides, we also conduct experiments on various model architectures, including the LSTM [41], TextCNN [42],



(a) Training/validation loss curves on $R0$.



(b) Training/validation loss curve on $R3$.

Fig. 4. Training and validation loss curves of different models on $R0$ and $R3$.

DPCNN [43] models, which are also used in other similar applications [20], for a better comparison with our attention based models.

4.3 Metric

We introduce the `Recall` value as our evaluation metric. Recall is for the proportion of actual positives that are identified correctly. Specifically, we use Recall@n (i.e., Recall@1, Recall@3, Recall@5) score to measure the Recall score among the top-n predicted pairs.

4.4 Implementation

For the siamese network parameters, the size of the embedding layer d is 128, and the corresponding triplet emebdding size $3d$ is 384. The hidden dimension for `FFN` layer is 512, and for the last fully-connected layer, it is also 384. For the model training, we use initial learning rate 0.001 and the Adam optimizer [44] with default setting to perform the model optimization. Dropout is used with value 0.1. During training, the batch size is set to be 1024, and we set the max number of triplets for each function to be 200. For training hardware, we train out model for total 100 epochs on single GTX 1080 GPU card.

4.5 Main Results

In Table 2, we show the results of our method. Besides, we also show the compared baseline models. From the table, we can see that our method achieves a strong performance on both $R0$ and $R3$ datasets, and over all evaluation metrics, Recall@1/3/5. Specifically, we obtain 89.6 Recall@1 on $R0$ and 82.2 Recall@1 on $R3$ dataset. Compared with different baseline models, our `DBSM` clearly outperform all the other methods, no matter with what kind of feature inputs or model networks.

4.6 Training Analysis

To give a better understanding of the training process of our method, we plot the loss curves on training and validation sets of our model and other methods. Here we mainly compare with the different model structures on both $R0$ and $R3$ datasets. The results are visualized in Figure 4(a) and Figure 4(b). From the figures, we can observe that our self-attention based model has a fast training convergence rate than other models, and the validation loss curve also proves that our model can achieve better performances.

4.7 Components Analysis

In this section, we compare `DBSM` with two types of baseline models (introduced in Section 4.2) to illustrate the contribution of triplet feature extraction and attention model.

Contribution of Triplet Features. To evaluate the effect of triplet features, we feed raw code (source and pseudo-code function) into TextCNN, DPCNN and attention models on character level. [20] mentioned that using character code as input is more robust and faster than parsing the code into an AST. We conduct experiments on our dataset and show the results in the first group of lines in Table 2. We can see that `DBSM` outperform above the raw code based methods on all Recall@n (n=1,3,5) metrics. Specifically, it improved Recall@1 by 33.3, 38.5 and 25.9 on $R0$, while improving 37.5, 49.7 and 40.1 on $R3$. Therefore, these results can clearly demonstrate that triplet features can better capture source and binary function information at both syntactic and semantic level, while the character level features only contains low-level information.

Contribution of Attention Model. To evaluate the superiority of our attention model, we simply replace the attention model with other model structures (LSTM, TextCNN, DPCNN), while keep other components in our framework `DBSM` unchanged. The experimental results are shown in the second group of lines in Table 2. From these numbers, we can observe that `DBSM` outperforms LSTM, TextCNN and DPCNN with respect to Recall@1 by 6.4, 2.1 and 11.4 on $R0$, while improving 4.9, 1.0 and 12.6 on $R3$. These improvements clearly demonstrate that our self-attention based model has strong ability to process the triplet features and capture the relationship between these triplet features.

5 Conclusion

In this paper, we propose a framework `DBSM` to solve binary and source function matching problem. Our approach takes advantage of decompilation technique to represent binary functions with high level (semantic) information. Besides, we extract triplets from code property graph, where the features are at syntactical and semantic level. Furthermore, we apply a self-attention based siamese network to process the triple features and perform matching learning task. Our evaluation results on real world datasets have demonstrated that, `DBSM` can match binary and source function well compared with other approaches.

Acknowledgments. The authors would like to thank the anonymous reviewers for their helpful feedback on an earlier version of this paper. This work is partly supported by National Key R&D Program of China under Grant #2022YFB3103900, Strategic Priority Research Program of the CAS under Grant #XDC02030200 and Chinese National Natural Science Foundation (Grants #62032010, #62202462).

References

1. P. Langley, "Crafting papers on machine learning," in *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, P. Langley, Ed. Stanford, CA: Morgan Kaufmann, 2000, pp. 1207–1216.
2. T. M. Mitchell, "The need for biases in learning generalizations," Computer Science Department, Rutgers University, New Brunswick, MA, Tech. Rep., 1980.
3. M. J. Kearns, "Computational complexity of machine learning," Ph.D. dissertation, Department of Computer Science, Harvard University, 1989.
4. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds., *Machine Learning: An Artificial Intelligence Approach, Vol. I.* Palo Alto, CA: Tioga, 1983.
5. R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*, 2nd ed. John Wiley and Sons, 2000.
6. N. N. Author, "Suppressed for anonymity," 2021.
7. A. Newell and P. S. Rosenbloom, "Mechanisms of skill acquisition and the law of practice," in *Cognitive Skills and Their Acquisition*, J. R. Anderson, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc., 1981, ch. 1, pp. 1–51.

8. A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 211–229, 1959.
9. M. Feng, Z. Yuan, F. Li, G. Ban, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu, A. Piao, J. Xue, and W. Huo, "B2sfinder: Detecting open-source software reuse in cots software," *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1038–1049, 2019.
10. S. H. H. Ding, B. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 472–489, 2019.
11. Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
12. "Ida pro," `https://www.hex-rays.com/products/ida/`, 2020.
13. F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.
14. R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
15. J. Zhang, A. Beresford, and S. A. Kollmann, "Libid: reliable identification of obfuscated third-party android libraries," *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.
16. A. Hemel, K. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *MSR '11*, 2011.
17. F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *ArXiv*, vol. abs/1808.04706, 2019.
18. X. Li, "Learning program-wide code representations for binary diffing," in *NDSS*, 2020.
19. B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "α diff: Cross-version binary code similarity detection with dnn," *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 667–678, 2018.
20. Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "Codecmr: Cross-modal retrieval for function-level binary source code matching," in *NeurIPS*, 2020.
21. L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *AAAI*, 2016.
22. C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.
23. N. D. Q. Bui, Y. Yu, and L. Jiang, "Treecaps: Tree-based capsule networks for source code processing," *ArXiv*, vol. abs/2009.09777, 2020.
24. Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *NeurIPS*, 2019.
25. Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "{MVP}: Detecting vulnerabilities using patch-enhanced vulnerability signatures," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1165–1182.
26. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017.

27. X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "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*, 2017, pp. 363–376.
28. Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.
29. L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017.
30. D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 921–937.
31. F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *arXiv preprint arXiv:1808.04706*, 2018.
32. Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 79–94.
33. J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.
34. Z. Lin, M. Feng, C. N. d. Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," in *International Conference on Learning Representations*, 2017.
35. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
36. J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.
37. H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring strategies for training deep neural networks." *Journal of machine learning research*, vol. 10, no. 1, 2009.
38. R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2. IEEE, 2006, pp. 1735–1742.
39. P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, no. 1, pp. 19–67, 2005.
40. J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, 1993, pp. 737–744.
41. S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
42. Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1746–1751.
43. R. Johnson and T. Zhang, "Deep pyramid convolutional neural networks for text categorization," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 562–570.

44. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.