GraphBinMatch: Graph-based Similarity Learning for Cross-Language Binary and Source Code Matching

Ali TehraniJamsaz, Hanze Chen, Ali Jannesari
Iowa State University, Ames, Iowa, USA
{tehrani, hanzech, jannesar}@iastate.edu

***Abstract*—Matching binary to source code and vice versa has various applications in different fields, such as computer security, software engineering, and reverse engineering. Even though there exist methods that try to match source code with binary code to accelerate the reverse engineering process, most of them are designed to focus on one programming language. However, in real life, programs are developed using different programming languages depending on their requirements. Thus, cross-language binary-to-source code matching has recently gained more attention. Nonetheless, the existing approaches still struggle to have precise predictions due to the inherent difficulties when the problem of matching binary code and source code needs to be addressed across programming languages.**

In this paper, we address the problem of cross-language binary source code matching. We propose GraphBinMatch, an approach based on a graph neural network that learns the similarity between binary and source codes. We evaluate GraphBinMatch on several tasks, such as cross-language binary-to-source code matching and cross-language source-to-source matching. We also evaluate the performance of our approach on single-language binary-to-source code matching. Experimental results show that GraphBinMatch significantly outperforms state-of-the-art, with improvements as high as 15% over the F1 score.

***Index Terms*—cross-language, code similarity, binary-source matching**

I. Introduction

Binary code is a collection of instructions that can be executed by computing systems directly, whereas source code, which programmers write, is readable and understandable. Binary-to-source code matching is a technique to evaluate the likeliness of binary code and source code. This is an important aspect of many security software engineering tasks, such as vulnerability [1] and malware detection [2] and reverse engineering [3], [4].

Typically, when it comes to matching a binary code to a source code, we either want to find the match for the binary file or the source code file. For example, when we have a binary code fragment, retrieving its similar source code snippet would be helpful, which can be used in a reverse engineering task. The retired source code snippet enables researchers to understand what a binary code fragment does. From the other aspect, if we have a source code snippet with a vulnerability, matching it to a binary code form helps to identify whether the vulnerability exists in the binary file.

Existing approaches try to measure the semantic similarity between binary and source code. However, most works focus on matching binary to binary or source code to source code [5]–[7]. Binary-to-source code is a non-trivial task as two modalities are involved: binary and source codes. Recent works have been trying to measure the similarity between binary and source code; however, they fall short in matching binary-to-source code across different programming languages. Recently, Gui *et al.* [8] proposed a transformer-based neural network to learn the similarity of binary and source code across programming languages; they use LLVM Intermediate Representation (IR) as input data for their model. However, they treat IR as a sequence of tokens.

In this paper, we present GraphBinMatch. An approach based on a Graph Neural Network to learn the semantic similarities between binary and source code. Unlike previous approaches, binary and source code are treated as graphs by leveraging three types of flows: control flow, data flow, and call flow.

Experimental results show that GraphBinMatch outperforms state-of-the-art approaches by increasing F1 from 0.65 to 0.79, recall from 0.59 to 0.82, and precision from 0.73 to 0.76.

Overall, the major contributions of this paper are:

1) Formulating the problem as learning the similarities between graphs.
2) Developing a special graph neural network as the backbone of GraphBinMatch to learn the similarity of graphs.
3) Evaluation of GraphBinMatch on a comprehensive set of tasks.
4) Effectiveness of the approach not just for cross-language but also single-language.
5) Up to 15% improvement in comparison to state-of-the-art approach.

The rest of the paper is structured as follows: We first formulate the binary-source matching problem in Section II. Then, in Section III, our proposed approach is outlined and explained. Experimental setups are discussed and explained in Section IV. Section V presents the evaluation results, followed by Section VI in which we discuss some insights. Next, in Section VII, we provide an overview of related works, and lastly, Section VIII concludes the paper and explains the future works.

II. Formulating the Problem

We formulate cross-language binary code-matching detection as follows: Given two programs P_a as binary and P_b as source code written in two different programming languages, we aim to train a deep learning model to learn the function γ, which can predict whether the two input programs are a binary-code matching pair or non-binary-code matching pair.

The training set consists of triples (P_a, P_b, y_{ab}) where y_{ab} is the label. We consider all pairs of binary and source programs collected from the same coding task in the dataset as positive samples and label them as 1, indicating that they are binary-source matches. Conversely, we consider all pairs of binary and source programs generated from different tasks in the dataset as negative samples and label them as 0, indicating that they are non-binary-source matches.

$$\gamma(P_a, P_b) = \begin{cases} 1, & \text{if } P_a \text{ and } P_b \text{ are matching} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

III. Approach

This section will discuss our proposed approach for identifying binary-source matching pairs across programming languages. The overall workflow is shown in Figure 1. The input to GraphBinMatch consists of two files: a source code file and a binary file written in different programming languages. The first step is to compile the source file and decompile the binary file using the respective language-specific front-ends and tools to produce Intermediate Representations (IR) that are language-independent. Next, we create graphs capturing the programs' control, data, and call flow. We treat these graphs as heterogeneous graphs to better model the different types of nodes and edges. An edge in the graphs represents the relationships between two nodes. These heterogeneous graphs are inputs to our Graph Binary Matching Similarity Neural Network (GraphBinMatch).

A. Intermediate Representation

As mentioned, the first step in our proposed approach is to convert the files from the source language and binary to an intermediate representation. Intermediate representation (IR) is an intermediate form that modern compilers use to represent programs. IR provides a more straightforward abstraction. It allows multiple stages of transformation and analysis in the compiler to generate the target code more efficiently. Lowering programs to intermediate representation is a common compiler technique to simplify and optimize code. In GraphBinMatch, we first convert input files to `LLVM IR`. Using LLVM IR, our approach can represent input files in a language-independent format, allowing for easier comparison of code written in different programming languages.

We use two front-ends, `JLang` [1] and `Clang` [2], to convert Java and C++ programs to LLVM IR, respectively. `JLang` supports Java up to version 7, while Clang-5.0 converts C++ programs to LLVM IR. It is worth mentioning that different versions of LLVM can produce slightly different IRs, so using the same version of Clang as the one being used inherently by `JLang` helps to have more similarities between Java and C++ programs.

Our proposed approach uses RetDec[3] to generate LLVM IR from binary executables. RetDec is an open-source machine-code decompiler that can generate LLVM IR from binary executables. It supports various architectures, including x86, ARM, MIPS, and PowerPC. RetDec uses instruction parsing, data-flow analysis, and control-flow reconstruction techniques to reverse-engineer the binary code into LLVM IR.

B. Graph Generation

While there exist various approaches [9]–[11] to represent `LLVM IR` for deep learning models, recently, it has been shown that presenting programs as graphs can help deep learning models to learn the characteristics of programs more effectively [12]–[14]. Following the recent success in presenting programs as graphs, we also create graphs using LLVM IR files. In particular, we use `ProGraML` [14] to generate a graph for each of the LLVM IR files in our dataset. `ProGraML` extracts information from LLVM IR and constructs a graph consisting of different types of nodes (i.e., instruction, variable and constant) and edges (i.e., control flow, data flow, and call flow). These graphs capture programs' structural and semantic information and can be used as inputs to machine learning models for various program analysis tasks, including code similarity detection. These graphs must be encoded and passed to a graph neural network designed to learn the similarities among matching pairs and dissimilarities among non-matching pairs.

C. Node Feature Embedding

To generate node embeddings, we split the process into two parts: preprocessing of the source code and processing within the model itself. During source file preprocessing, we create a feature vector for each node using the `full_text` or `text` of the node's attributes in the graph generated by `ProGraML`. The `full_text` attribute represents the complete LLVM-IR instruction for the node, whereas the `text` attribute only contains the corresponding instruction type. For instance, in the instruction `%16 = load i32 i32 %15 align 8`, means the integer pointer %15 is loaded and stored in temporary variable %16. We can only obtain the instruction type from the `text` attribute. However, using the `full_text` attribute, we can determine that the load instruction is responsible for loading an integer pointer. This additional information can be used to better train the embedding layer and improve the model's accuracy.

We discovered that not all nodes have the `full_text` attribute during the preprocessing process, as some nodes only represent compiler configurations that only have `text` attribute. To address this issue, we use the `text` attribute as a fallback option when the `full_text` attribute is unavailable.

[1] https://polyglot-compiler.github.io/JLang

[2] https://clang.llvm.org

[3] https://github.com/avast/retdec

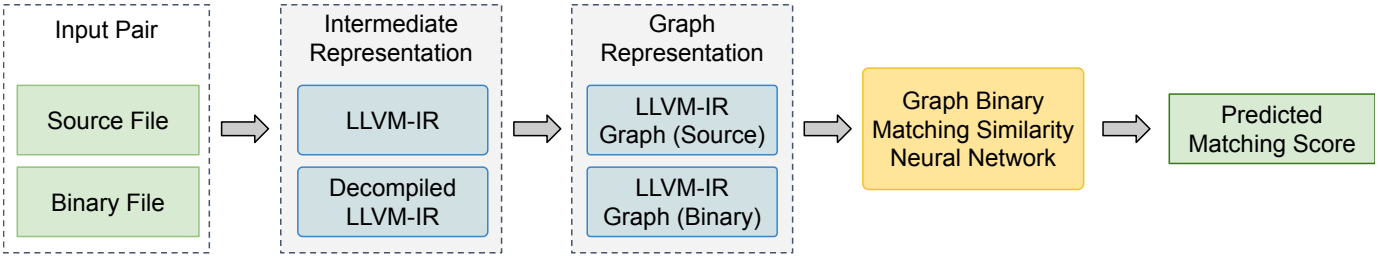

Fig. 1: The overview of our proposed approach.

The ProGraML paper itself uses `text` attribute of the nodes as the feature of the nodes; however, later in the experimental result section, we will see that using the `full_text` attribute of the nodes improves the prediction of the GraphBinMatch. Moreover, we use the edge `position` property provided by `ProGraML` as the edge feature. The `position` property in `ProGraML` contains the edge position information. For instance, for a valid LLVM-IR instruction such as `%result = add i32 %1, %2`, `ProGraML` generates an edge corresponding to `%1` and an edge corresponding to `%2`. In this example, the position of the first edge is 0, and the position of the second edge is 1.

Finally, we use the Byte-Pair Encoding (BPE) tokenizer to tokenize those LLVM-IR instructions to create the final node features, which we then add to the graph. Tokenizer is also able to map each token to a corresponding integer number. Therefore, we would have a sequence of integer numbers representing an LLVM-IR instruction for a node. This sequence of integer numbers is considered as the feature of the node. In the conversion process, we convert all LLVM-IR variables, such as `%12`, to a special token named `[VAR]`. By utilizing the tokenizer and carefully selecting the truncation and padding lengths, we ensure that our feature vectors are informative and can be effectively used in subsequent modeling steps.

LLVM-IR instructions have varying lengths; therefore, the feature set of each node in our graph will have its own length. To solve this problem and make sure that all nodes have the same length for their features, We use the average length of all nodes' feature vectors rounded up to the nearest power of 2 as the final intercept length. For example, if the average length of all LLVM-IR instructions in the dataset is 50 after tokenization, the final truncation length is 64. All feature vectors with a length of 64 will be truncated, and all feature vectors with a length less than 64 will be padded with `[PAD]` token.

D. Graph Binary Matching Similarity Neural Network

Once the preprocessing is finished, we will pass the graphs to our Graph Binary Matching Similarity Neural Network (GraphBinMatch) for further processing. GraphBinMatch is built upon SimGNN [15], but we have made several modifications to tailor it for heterogeneous graph similarity tasks, which will be discussed in the next sections.

The architecture of GraphBinMatch can be seen in Figure 2. To train GraphBinMatch, we create data points consisting of three items: **Source File A**, **Binary File B**, and **Label**. We use an embedding layer as the first layer of GraphBinMatch. This layer processes the feature vector of each node and tries to embed the nodes by learning the embedding space so that similar nodes would have similar embedding. The embedding layer increases the dimensionality of the entire feature vector to two dimensions. We utilize the max operation to reduce the two-dimensional feature vector to a single dimension. This one-dimensional feature vector serves as input to the graph convolution layer.

In the following subsections, we will provide a detailed explanation of each component of GraphBinMatch.

*1) **Heterogeneous Convolution**:*

GraphBinMatch is designed to take in the graph representation of two files and predict whether they match. The Convolution layer is constructed to support the heterogeneous graph using the `HeteroConv` wrapper from the `pytorch-geometric` library. As previously discussed, our Heterogeneous Graph representation has three types of relationships. This layer includes three separated `GATv2Conv` [16] layers to model each one of the relationships. The outputs of the `GATv2Conv` layers are stacked together, and then the element-wise maximum values are computed to have a latent representation of the nodes. After each `GATv2Conv`, we include additional `LayerNorm` to stabilize training and prevent overfitting.

*2) **Attention**:*

In GraphBinMatch, we incorporate an attention layer similar to the one introduced by Bai et al. [15]. The attention layer operates by taking node embeddings from the previous layer and passing them through an attention mechanism. We use a global graph embedding vector $c \in \mathbb{R}^D$ where D is the same dimension as the nodes' embedding dimension. c is created by averaging node embeddings passed through a non-linear transformation. This global embedding vector c contains the overall structure and information of the graph.

To calculate the attention for a given node n_i, we compute the inner product of c and the latent representation of n_i. The intuition behind this approach is that nodes that are more similar to the overall context of the graph should receive higher attention weights. Finally, the graph-level embedding is created by computing the weighted sum of all nodes. Here, the nodes' weights refer to the nodes' attention values.

*3) **Fully Connected Layer**:*

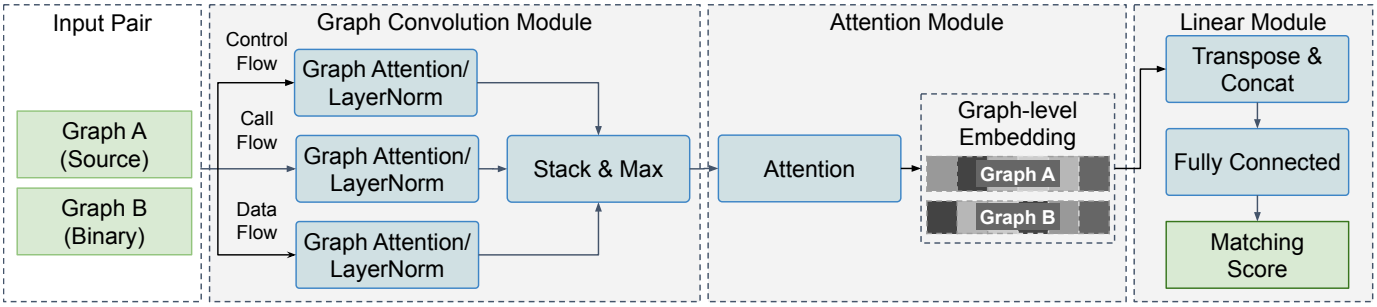

Fig. 2: Structure of Graph Binary Matching Similarity Neural Network (GraphBinMatch).

Once the graph-level embeddings on the two files are computed, the two vectors are concatenated and transposed to be fed to the fully connected layer to have the final prediction. In our study, we use two fully connected layers. After the first fully connected layer, a normalization layer is also applied to stabilize the learning process. Additionally, we introduce an extra dropout layer before the last linear layer to increase nonlinearity and regularization in the model.

In the next section, we evaluate GraphBinMatch and compare its results with the state-of-the-art approach.

IV. Experimental Setup

In this section, we evaluate GraphBinMatch, present the results, and address four research questions.

A. Tasks and Research Questions

We evaluate GraphBinMatch on cross-language and single-language tasks. For each one of these tasks, we use a specific dataset. We define the research questions as follows

- **RQ1:** Can graph-based representation and GNNs outperform transformer-based models in cross-language binary source code matching problems?
- **RQ2:** Does GraphBinMatch provide consistent results when applied in a single-language context with different optimization levels?
- **RQ3:** How is the performance of GraphBinMatch across different compilers?
- **RQ4:** Does GraphBinMatch perform well in terms of source-to-source matching?

B. Dataset Statistic

Cross Language Code Matching. To evaluate GraphBinMatch for the cross-language binary-source matching task, we use the CLCDSA dataset [17], which consists of source code files collected from two programming competition websites: AtCoder and Google CodeJam. These programming competition websites feature multiple tasks, and the dataset contains source code as solutions to the tasks in various programming languages. We have selected C, C++, and Java as the languages for our study to assess the effectiveness of our model in learning binary-source code similarities across different programming languages.

To ensure that our dataset has a balanced distribution of positive and negative samples, we consider valid solutions to the same task as matching pairs and valid solutions to different tasks as non-matching pairs. We discard any file that is not compilable. Following the baseline paper, we adopt the same train, validation, and test split ratio, which is a ratio of 6:2:2 to split the dataset.

As mentioned, in this study, we utilize JLang[4] and clang-5.0 as compilers for all Java source code files. JLang is an open-source compiler front-end that compiles Java source code into the corresponding LLVM-IR. This compiler is used to generate all corresponding LLVM-IR expressions for Java files. To convert LLVM-IR to binary executable files, `Clang-5.0` is used. Throughout the experiments, -Oz is set as the default optimization level of the compiler unless the optimization level is explicitly mentioned. To convert all binary executables to their LLVM-IR equivalent, we use an open-source RetDec decompiler.

Same Language Code Matching For the task of detecting matching within the same programming language, we use the POJ-104 dataset [18] [5]. This dataset comprises C++ solutions submitted by 500 students for 104 different programming problems from an Online Judge system (OJ) that serves an educational purpose. We compile the C++ source code files into LLVM-IR and binary executable using different optimization levels of `clang` and `gcc`. To decompile the binary executable to its corresponding LLVM-IR representation, we utilized RetDec similarly for all the binary executables.

TableI shows the statistics of the two datasets.

C. Baselines

As discussed, we evaluate the performance of GraphBinMatch for two different matching tasks: binary-source matching and source-source matching. To assess the effectiveness of GraphBinMatch, we compare it against BinPro [3], B2SFinder [7], and XLIR for binary-source matching.

BinPro is a tool that aims to tackle the challenge of identifying similarities between source and binary code even when the compiler or optimization level used is unknown. To this end, BinPro employs machine learning techniques to compute the best code properties for determining binary-to-source code similarity. These code properties are then

[4]https://github.com/polyglot-compiler/JLang

[5]https://drive.google.com/uc?id=0B2i-vWnOu7MxVlJwQXN6eVNONUU

TABLE I: Dataset Statistics

	Languages	# Sources	# LLVM-IR	# Binary Files	# Decompiled LLVM—IR
CLCDSA	C	15605	13929	14370	13929
	C++	16676	15375	15766	15589
	Java	19836	15124	17072	15124
POJ-104	C++	52000	38598	38598	37909

extracted and computed using static analysis tools to match binary and source codes with a bipartite matching algorithm.

B2SFinder detects binary code clones by inferring seven traceable features in binary and source code. It employs a weighted feature-matching algorithm capable of handling different features and calculating the weights of code feature instances based on their specificity and frequency of occurrence.

XLIR [8] is a transformer-based neural network model that is currently state-of-the-art for binary-source code matching. XLIR, as its name suggests, uses LLVM IR. To embed the tokens in LLVM IR, XLIR leverages a pre-trained BERT model. The model first pre-trains the neural network using a large external LLVM-IR corpus with masked language modeling (MLM) [19] as the pre-processing step. This step is aimed at learning meaningful representations of the LLVM-IR tokens. Once the tokens are embedded, XLIR maps them into a common space, and the LLVM-IR representations are learned jointly using a ternary loss function. This approach allows XLIR to match binary source code across different programming languages.

D. Experiment Setup

GraphBinMatch is built using `Pytorch-Geometric`[6]. For optimizing the learning parameters of GraphBinMatch, Adam [20] Optimizer is used with a learning rate of $6.6e^{-5}$, and Binary Cross Entropy is used as the loss function. We use GATv2 [16] as the graph convolution layer. The hyper-parameters of GraphBinMatch are tuned using RayTune[7].

GraphBinMatch comprises a PyTorch Embedding layer with a dimension of 128 to embed the tokens and five GATv2 graph convolution layers with a dimension of 256. In the GraphBinMatch model, we use `LeakyReLU` as our activation function, except the last linear layer is followed by a `Sigmoid` function. We train GraphBinMatch using four A100 NVIDIA GPUs with 80GB VRAM and Intel Xeon 6140 CPU with 128GB RAM.

E. Evaluation Metrics

Parameter	Prediction	Actual
True Positive(**TP**)	Matching	Matching
True Negative(**TN**)	Non-matching	Non-matching
False Positive(**FP**)	Matching	Non-matching
False Negative(**FN**)	Non-matching	Matching

TABLE II

[6]https://pytorch-geometric.readthedocs.io

[7]https://docs.ray.io/en/latest/tune/index.html

In the realm of code matching detection, precision (P), recall (R), and F1-scores (F1) are commonly used to evaluate the performance and accuracy of the models. These three criteria are derived from four measures: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). The definitions of these measures can be found in Table II.

Precision is often used to describe the accuracy of a model's positive prediction, that is, the accuracy for pairs identified as matching. It is defined as the proportion of true matching pairs out of all the matching pairs predicted by the model, as shown in Equation 2.

$$P = \frac{TP}{TP + FP} \tag{2}$$

Recall is defined as the percentage of matching pairs in the dataset that the model correctly predicts as matching pairs, which is the case described by Equation 3.

$$R = \frac{TP}{TP + FN} \tag{3}$$

F1-Score is commonly used to evaluate the performance of models. F1-Score is defined as the harmonic mean of the precision and recall values, which is the case described by Equation 4.

$$F1 = \frac{2PR}{P + R} \tag{4}$$

V. Experimental Results

In this section, we address the research questions.

*1) **RQ1: Can graph-based representation and GNNs outperform transformer-based models in cross-language binary source code matching problem?**:*

To answer the question of RQ1, we trained our model using the CLCDSA dataset. We evaluated the performance of GraphBinMatch in the binary-source matching task by comparing it against the aforementioned baselines, as shown in Table III. We used LLVM-IR from binary C/C++ programs and Java source code as GraphBinMatch's input in our experiments. The results show the effectiveness of GraphBinMatch with the precision, recall, and F1 scores achieving 0.76, 0.82, and 0.79, respectively. This represents over 20% improvement compared to the baseline paper. To further strengthen the robustness of our results, we conducted an additional experiment using LLVM-IR from Java binary and C/C++ source code. This yielded satisfactory results, with precision, recall, and F1 scores of 0.76, 0.77, and 0.77, respectively, which exceeded the baseline by 25%.

Table III reveals a performance gap between using binary Java source code and binary C/C++ code for the same task and dataset. This gap may be attributed to certain differences

TABLE III: Performance of cross-language binary-matching task (Threshold at 0.5).

	C/C++ binary code with Java source code			Java binary code with C/C++ source code		
	Precision	Recall	F1	Precision	Recall	F1
BinPro	-	-	-	0.36	0.37	0.36
B2SFinder	-	-	-	0.35	0.41	0.38
XLIR(LSTM)	0.62	0.53	0.57	0.55	0.51	0.53
XLIR(Transformer)	0.73	0.59	0.65	0.68	0.55	0.61
GraphBinMatch	0.75	0.73	0.74	0.75	**0.78**	0.77
GraphBinMatch(Tokenizer)	**0.76**	**0.82**	**0.79**	**0.76**	0.77	**0.77**

*Because of the limitations of BinPro and B2Sfinder in handling Java source code, we cannot provide test results for Java as source code

*In this table and the following ones, the results of the other tools are quoted from the baseline paper.

between the LLVM-IR obtained by decompiling and the one obtained from source code, as GraphBinMatch struggles to comprehend. The decompiled LLVM-IR is not always identical to the source code, and we attribute this difference to two primary factors. Firstly, the decompiled code may not always have the exact data type or the correct shape of arrays for array types. Secondly, the decompilation process often involves speculation and assumptions, resulting in variations in the control flow generated by decompiling binaries. Combining these factors within the binary file causes differences in the LLVM-IR.

*2) **RQ2: Does GraphBinMatch provide consistent results when applied in a single-language context with different optimization levels?**:*

TABLE IV: Performace of single language binary matching task (Threshold at 0.5).

	Precision	Recall	F1
BinPro	0.38	0.42	0.40
B2SFinder	0.43	0.46	0.44
XLIR(LSTM)	0.67	0.72	0.44
XLIR(Transformer)	0.85	0.86	0.85
GraphBinMatch	0.88	0.86	0.87

TABLE V: Same language binary matching result from different optimization level

	clang-10.0			gcc-9.4		
	Precision	Recall	F1	Precision	Recall	F1
O0	0.88	0.86	0.87	0.87	0.86	0.87
O1	0.87	0.88	0.88	0.89	0.85	0.85
O2	0.86	0.82	0.84	0.87	0.83	0.85
O3	0.86	0.83	0.85	0.84	0.81	0.83
Oz	0.90	0.85	0.87	0.87	0.87	0.87

The answer to this research question is affirmative. For this study, we used a different dataset than CLCDSA due to insufficient data for the same language. As shown in Table I, CLCDSA has a maximum of only 15589 C++ source codes available, whereas the POJ-104 dataset provides 37909 source codes. This replacement of the dataset provides 1.5 times more code. POJ-104 is also the dataset that has been used in the baseline paper as well. Table IV reveals that GraphBinMatch outperforms the baseline paper regarding precision, recall, and F1 scores, with scores of 0.88, 0.86, and 0.87, respectively, indicating that GraphBinMatch is more proficient at detecting binary-source matching of the same language. This table shows that the transformer-based model (XLIR) can perform quite well as the LLVM-IR from the same language since these IRs show more similarities. Despite the high scores of XLIR on a single language, GraphBinMatch still outperforms it.

To verify the robustness of GraphBinMatch in different optimization scenarios, we evaluated its performance using clang on the same dataset but with four different optimization levels (-O0, -O1, -O2, -O3, -Oz). TableV shows that GraphBinMatch achieves consistent performance across different optimization levels. This indicates that the model's performance is not dependent on a specific compiler or optimization level and can provide reliable results under varying optimizations.

By examining the performance of GraphBinMatch with different optimization levels of the same compiler, we can obtain some insights regarding RQ2. We suspect that higher optimization levels would result in more aggressive optimizations by the compiler, such as control flow tuning. These optimizations would provide the decompiler with additional assumptions and speculations, resulting in an increased difference between LLVM-IR from both the source code and the binary executable. As shown in Table V, we observe that the precision, recall, and F1 scores used to measure model performance gradually decrease as the optimization level increases. This suggests that more aggressive optimizations can slightly affect the model's performance in code matching detection.

*3) **RQ3: How is the performance of GraphBinMatch across different compilers?**:*

GraphBinMatch showed consistent performance across different optimization levels of the same compiler; in this subsection, we want to investigate further to see whether it could also provide similar performance across different compilers. This is an important consideration for real-world applications where the compiler used to generate a binary executable may be unknown. To test this, we used `clang` to generate LLVM-IR from the source code, but `gcc` to generate the binary executable of the POJ-104s dataset compared to RQ2. We kept the same settings as in RQ2 to convert the binary executable to the corresponding LLVM-IR using RetDec. Table V presents the performance of GraphBinMatch when generating binary executable using `gcc`. The results show that GraphBinMatch performs relatively the same when using different compilers.

We suspect the better performance figures of `gcc` than clang because the used decompilers tend to provide more information when compiling the C++ code generated by `gcc`. This is supported by our analysis, which shows that the average size of LLVM-IR generated by binary executables

compiled with `clang` is about $10,769.9$ bytes, while the average size of those compiled with `gcc` is about $18,525.2$ bytes. This means at the decompilation stage, the size of `gcc` compiled binary files is approximately 70% larger than those compiled with a `clang`. Such a significant difference in size will likely affect GraphBinMatch's ability to accurately detect binary matching.

*4) **RQ4: Does GraphBinMatch perform well in terms of source-to-source matching?**:*

In the source-source matching task, we evaluate the performance of GraphBinMatch using the CLCDSA dataset and compare it with the baseline paper. The experimental setup was the same as in RQ1, except that we used the LLVM-IR generated by `JLang` as input to GraphBinMatch. We test three language combinations: C/C++ vs. Java, C vs. Java, and C++ vs. Java. Table VI shows that GraphBinMatch outperformed the baseline paper by about 20% regarding precision, recall, and F1 scores. GraphBinMatch can also effectively detect source code matching across different programming languages.

A. Varying Threshold

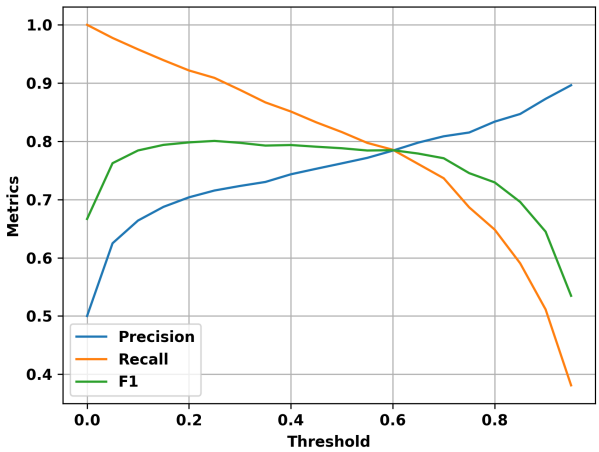

Fig. 3: [Higher is better] Varying the threshold results in different scores for precision, recall, and f1.

In the previous subsection, we mentioned that the threshold for GraphBinMatch was set to 0.5. We experimented with varying thresholds to investigate the effect of different thresholds on precision, recall, and F1 scores. Figure 3 shows the different scores that can be achieved by varying the threshold. Based on our findings, a threshold value of 0.2 would provide a slightly better F1 score. However, we also observed that using this threshold would significantly decrease accuracy to as high as 7%, making it impractical to use for optimal F1 score to find the best threshold. As a result, we chose to use a threshold value of 0.5, which we considered to be a more reasonable default threshold for our study. A smaller threshold yields higher recall since GraphBinMatch will predict all pairs as matching pairs. On the other hand, a larger threshold will result in higher precision, as GraphBinMatch predicts all pairs as non-matching pairs. Depending on the use case, a user of GraphBinMatch can manually set the threshold or let GraphBinMatch decide the best threshold based on the given metric (i.e., precision, recall, F1).

VI. Discussions

In this section, we discuss some challenges that can affect the performance of GraphBinMatch.

A. Investigating why GraphBinMatch may fail

Our experiments revealed that our model occasionally misidentifies pairs of matching code fragments. After reviewing the mispredicted samples individually, we found that most false positives occur because of a large gap in the sizes of the LLVM-IR fragments. We calculated and analyzed the number of nodes in the test set graphs and obtained Table VII. The table shows that the difference in the mean and the median number of nodes predicted by our model is much larger for the false positive set than for the true positive set. The median difference in the number of nodes between the two sets is nearly 50%.

After conducting an in-depth analysis, we found two main reasons for these false positives. The first reason is that the gap between LLVM-IR is larger than the tolerance of GraphBinMatch. For example, some code snippets may use sorting methods provided by the standard library, while others may implement their own sorting methods internally. GraphBinMatch may not effectively recognize that the method calls of the standard library are equivalent to the sorting methods the authors themselves have implemented in their code. Additionally, the template mechanism in C++ can impact GraphBinMatch since many of the C++ standard libraries are published as templates. This means that templates are also compiled as a part of LLVM-IR, which causes struggles for GraphBinMatch to recognize that the compiled template code is equivalent to standard library calls in Java.

The second scenario involves language differences between Java and C++ and different usage habits among their respective user communities, which can result in significant discrepancies in the LLVM-IR even when both are converted to compile as LLVM-IR. These discrepancies are rooted in the varying habits of different programming language users. For example, Figure 4 displays a matching pair that, despite their similarity, generate IR graphs with vastly different sizes: the Java-generated IR graph has 330 nodes and 660 edges, whereas the C++-generated IR graph has only 65 nodes and 115 edges. Such discrepancies can cause challenges for GraphBinMatch to recognize matching pairs, as the model may not account for the impact of language usage habits on the IR generation process, leading to incorrect results.

B. Extending GraphBinMatch to other programming languages

This study introduced a novel approach to detecting cross-language binary-code matching using graph neural networks

TABLE VI: Cross language source matching result

	GraphBinMatch			XLIR (LSTM)			XLIR (Transformer)		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
C vs Java	**0.77**	**0.80**	**0.78**	0.62	0.51	0.56	0.75	0.55	0.63
C++ vs Java	0.76	**0.82**	**0.79**	0.65	0.53	0.58	**0.77**	0.57	0.66
C/C++ vs Java	0.81	0.73	0.78	-	-	-	-	-	-

TABLE VII: Statistics for the number of nodes in test set

Type	Mean	Median
True Positive	1506	864
False Positive	2133	1303
True Negative	2573	1680
False Negative	2293	1289

(GNNs). A crucial component of our approach is using LLVM IR to measure the similarities between two programs written in different languages. By leveraging LLVM IR, we can capture high-level information about the code independent of the underlying programming languages.

While our approach is designed to be cross-language, it relies on the availability of compiler front-ends. One must use the corresponding compiler front-end to extend our approach to support additional programming languages to produce LLVM IR. Once the IR is generated, our approach remains the same. Our method is flexible enough to support any programming language compiled into LLVM IR.

C. How different token embedding influence the result

TABLE VIII: Performance of different LLVM-IR embedding techniques for same-language binary matching and cross-language binary matching

	Cpp vs Cpp			Cpp/C vs Java		
	Precision	Recall	F1	Precision	Recall	F1
text	0.86	0.83	0.85	0.75	0.73	0.74
full_text	0.89	0.87	0.88	0.84	0.75	0.79

In this section, we conduct an experimental study using the CLCDSA dataset with the same settings and preprocessing methods as in RQ1. Our objective is to investigate the impact of various token embedding techniques and tokenizers on the performance of GraphBinMatch. Specifically, we aim to explore the effects of different embedding methods and tokenization schemes on the ability of GraphBinMatch to match binary and source code pairs across multiple programming languages or the same languages. By examining these factors, we hope to better understand how to optimize the performance of GraphBinMatch and improve its ability to detect code clones in diverse language pairs.

First, we aim to evaluate the impact of using the `full_text` property versus `full_text` for the source-binary matching task. In the approach section of our paper, we mentioned that using the `full_text` property provided by `ProGraML` could lead to better performance than using `full_text` alone. As shown in Table VIII, we observe that using the `full_text` attribute improves the performance for both cross-language binary-source matching tasks and same-language binary-source matching tasks. Notably, for cross-language binary-source code tasks, using the `full_text` attribute provides more significant improvement than binary-source code matching tasks written in the same language. We believe this result is because by exposing more information to the model, we can better bridge the understanding gap of the model on cross-language tasks.

VII. Related Works

Code similarity detection has recently gained considerable attention with the advent of research tools and machine learning models. These advancements provide new opportunities for research in this field. In code similarity detection, the similarity of source code is generally classified into four levels, as outlined in [13]:

- **Type I**: This is also called *Exact Clone*. The source code can be identical, with only the indentation, comments, and code layout modified.
- **Type II**: This is also called *Parameterized clone*. The source code's structure and syntactic are similar except for some variable names, method names, and data types modified in addition to those mentioned in Type I.
- **Type III**: Compared to Type II, Type III code clone involves modification of statements, but the functional similarity is maintained.
- **Type IV**: The structure between two code fragments is syntactically and structurally different, but both code fragments perform similar behavior for the same input.

Research on code similarity detection is broadly divided into two research directions: algorithm-based and machine learning-based. Each of these two different directions will be described in detail below:

A. Algorithm based Approaches

The algorithm-based code clone detection is based on lexical or semantic analysis, so most do not have a good detection result for Type IV code clones, or even when Type III gaps are too large or too frequent. Most of the algorithms are only applicable to Type I, Type II, and part of Type III code clones. Also, their results for cross-language code clone detection are often unsatisfactory.

Research projects like CCFinderSW [21] and SourcerCC [22] use a token-based approach to detect code clones. Each of these tools uses its own implementation of a lexical analyzer to convert the source code into a token stream which serves as a basis for analysis and similarity score measurement using different algorithms. However, because of the lack of lexical

```java
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s = sc.next();
        int ans = 0;
        for (char c : s.toCharArray()) {
            ans += Integer.parseInt(""+c);
        }
        System.out.println(ans);
    }
}
```

(a) Java sample

```cpp
int main(){
    int X;
    cin >> X;
    int ans = 0;
    while(X > 0) {
        ans += X % 10;
        X /= 10;
    }
    cout << ans << endl;

    return 0;
}
```

(b) C++ sample

Fig. 4: An example of false negative case

information abstraction, it is difficult for these algorithms to obtain and understand the inherent semantics.

Semantic analysis-based code clone detection tools try to solve this problem. DECKARD [23] proposes a tree-based code clone detection scheme: it innovatively uses feature vectors to describe the structure of the syntax tree and clusters the clones by the Euclidean distance of the feature vectors.

Traditional algorithm-based code clone detection methods are often difficult to transfer to cross-language clone detection, resulting in very few studies addressing this problem. Even if the relevant studies propose corresponding solutions, they are limited by various restrictions and cannot address real-world requirements. Two of the most representative studies are LICCA [24] and CLCMiner [25]. LICCA detects code clones by attempting to convert different languages into uniform expressions, which can only cover cases where two pieces of code have a similar structure and syntactic elements. CLCMiner uses an NLP approach to view the source code but relies on the modification records of the source code.

B. Machine Learning Approaches

With the advancements in machine learning and graph neural networks, new ideas have emerged to solve the code clone detection problem. With the introduction of various code cloning datasets, it has become possible to use machine learning to solve such problems. Like CLCDSA [17] and BigCloneBench [26], they both contain code clones with Type IV similarity. Many recent studies have tried to replace traditional algorithms with machine learning models to achieve better accuracy and gain the ability to compare across different languages.

CDLH [27] first proposed a hash method to detect whether two pieces of code in the same language are clones. Their model accepts raw code fragments as input, encodes the abstract syntax tree using specific rules, and transforms it into a vector representation of the source code using an LSTM network. However, their encoding approach focuses more on the structure of the AST without considering other semantic information like node type.

In addition to tree-based methods, methods like CCLearner [28] and C4 [29] use token-based code clone detection: One of the first studies to use tokens as input to a code clone detection model is CCLearner. It classifies tokens into eight categories, calculates the similarity score for each category separately, and then uses the similarity of vectors to calculate the similarity between code fragments. This study concluded that more similarities in feature vectors imply a higher probability that a code pair is a code clone. This study only captures token and partial syntax level information but not the structure of the code. Their paper stated that CCLearner does not have good detection performance for Type IV/Type III clones due to this reason.

Another token-based research is C4 [29]. Their study used a pre-trained BERT model called CodeBERT [30] to embed the model's inputs. Since CodeBERT supports encoding multiple languages, C4 also supports cross-language code clone detection. Their study uses raw source code as input and first use CodeBERT to encode the raw source code to obtain a feature vector, and then their model learns and classifies the feature vector to obtain the final similarity score. They also implemented Contrastive Learning to increase the usage of the dataset. However, C4 only supports encoding the first 512 tokens because of the limitation of CodeBERT, affecting the final applicability of C4.

While great attention has been given to code clone detection, XLIR [8] is one of the recent works that has tried to tackle the problem of binary source code matching. It is common for software applications to be written in different programming languages to meet various requirements and computing platforms. Therefore, detecting binary source code clones across multiple programming languages can be beneficial in practical scenarios. For instance, when vulnerable binary code is detected, it becomes necessary to retrieve relevant source code fragments of all possible programming languages written for better vulnerability assessment. XLIR provides a transformers-based approach to this problem. We use XLIR as the baseline paper; more details the comparison results are provided in the Section V.

VIII. Conclusion and Future Works

This paper introduced GraphBinMatch, a novel cross-language binary-source matching model that uses LLVM IRs

and Heterogeneous graphs to learn the similarities. The model takes LLVM IR as input and can handle three types of code-matching detection tasks: cross-language binary matching detection, single-language binary matching detection, and cross-language source matching detection. It should be noted that GraphBinMatch is not limited to these specific languages. It can support any programming language with a compiler front end to generate LLVM IR.

Experimental results show that GraphBinMatch is effective and outperforms all three state-of-the-art tools. Specifically, our approach outperforms these tools regarding precision, recall, and F1 score.

In future work, we aim to extend GraphBinMatch to more programming languages, incorporating additional compiler front-ends to generate LLVM IR for these languages. Moreover, since our approach is data-driven, we are collecting more source code files to increase the size of the dataset, which will believe will improve the prediction of GraphBinMatch.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation (NSF) under grant number 2211982. we would also like to thank the Research IT team[8] of Iowa State University for their support in providing HPC clusters for conducting this research.

REFERENCES

[1] R. T. Yarlagadda, "Approach to computer security via binary analytics," *INTERNATIONAL JOURNAL OF INNOVATIONS IN ENGINEERING RESEARCH AND TECHNOLOGY [IJIERT]*, 2020.

[2] H. Yang, M. Fritzsche, C. Bartz, and C. Meinel, "Bmxnet: An open-source binary neural network implementation based on mxnet," in *Proceedings of the 25th ACM international conference on Multimedia*, 2017, pp. 1209–1212.

[3] D. Miyani, Z. Huang, and D. Lie, "Binpro: A tool for binary source code provenance," *arXiv preprint arXiv:1711.00830*, 2017.

[4] A. Shahkar, "On matching binary to source code," Ph.D. dissertation, Concordia University, 2016.

[5] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 141–151.

[6] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[7] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu *et al.*, "B2sfinder: Detecting open-source software reuse in cots software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1038–1049.

[8] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao, and H. Jin, "Cross-language binary-source code matching with intermediate representations," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2022, pp. 601–612. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER53432.2022.00077

[9] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[10] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–27, 2020.

[8]https://researchit.las.iastate.edu

[11] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[12] M. Allamanis, "Graph neural networks in program analysis," in *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer, 2022, pp. 483–497.

[13] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[14] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," *arXiv preprint arXiv:2003.10536*, 2020.

[15] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang, "Simgnn: A neural network approach to fast graph similarity computation," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, 2019, pp. 384–392.

[16] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=F72ximsx7C1

[17] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: cross language code clone detection using syntactical features and api documentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1026–1037.

[18] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[19] K. Sinha, R. Jia, D. Hupkes, J. Pineau, A. Williams, and D. Kiela, "Masked language modeling and the distributional hypothesis: Order word matters pre-training for little," *arXiv preprint arXiv:2104.06644*, 2021.

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

[21] Y. Semura, N. Yoshida, E. Choi, and K. Inoue, "Ccfindersw: Clone detection tool with flexible multilingual tokenization," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 654–659.

[22] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1157–1168. [Online]. Available: https://doi.org/10.1145/2884781.2884877

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 96–105.

[24] T. Vislavski, G. Rakić, N. Cardozo, and Z. Budimac, "Licca: A tool for cross-language clone detection," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 512–516.

[25] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu, and J. Zhao, "Clcminer: detecting cross-language clones without intermediates," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 2, pp. 273–284, 2017.

[26] J. Svajlenko and C. K. Roy, "Bigclonebench," in *Code Clone Analysis*. Springer, 2021, pp. 93–105.

[27] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 3034–3040. [Online]. Available: https://doi.org/10.24963/ijcai.2017/423

[28] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 249–260.

[29] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: Contrastive cross-language code clone detection," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 413–424.

[30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.