Cross-Language Binary-Source Code Matching with Intermediate Representations

Yi Gui[1], Yao Wan[1*], Hongyu Zhang[2], Huifang Huang[3], Yulei Sui[4], Guandong Xu[4], Zhiyuan Shao[1], Hai Jin[1]
[1]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[2]The University of Newcastle, Australia
[3]School of Mathematics and Statistics, Huazhong University of Science and Technology, Wuhan, China
[4]School of Computer Science, University of Technology Sydney, Australia
{guiyi, wanyao, hhf, zyshao, hjin}@hust.edu.cn, hongyu.zhang@newcastle.edu.au, {yulei.sui, guandong.xu}@uts.edu.au

***Abstract*—Binary-source code matching plays an important role in many security and software engineering related tasks such as malware detection, reverse engineering and vulnerability assessment. Currently, several approaches have been proposed for binary-source code matching by jointly learning the embeddings of binary code and source code in a common vector space. Despite much effort, existing approaches target on matching the binary code and source code written in a single programming language. However, in practice, software applications are often written in different programming languages to cater for different requirements and computing platforms. Matching binary and source code across programming languages introduces additional challenges when maintaining multi-language and multi-platform applications. To this end, this paper formulates the problem of cross-language binary-source code matching, and develops a new dataset for this new problem. We present a novel approach XLIR, which is a Transformer-based neural network by learning the intermediate representations for both binary and source code. To validate the effectiveness of XLIR, comprehensive experiments are conducted on two tasks of cross-language binary-source code matching, and cross-language source-source code matching, on top of our curated dataset. Experimental results and analysis show that our proposed XLIR with intermediate representations significantly outperforms other state-of-the-art models in both of the two tasks.**

***Index Terms*—Cross-language, clone detection, intermediate representation, binary code, code matching, deep learning.**

I. Introduction

Binary-source code matching, which aims to measure the similarity between binary code and source code, plays an important role in a variety of security software engineering related tasks, e.g., malware detection [1], vulnerability search [2], and reverse engineering [3], [4]. From one hand, given a binary code fragment, it is useful to retrieve similar source code snippets that can serve as references for reverse engineering. On the other hand, given a vulnerable source code, it is also helpful to check whether its corresponding binary form is included in a binary file, which is useful for vulnerability assessment and detection.

Existing Efforts and Limitations. The core technique for binary-source code matching is the calculation of semantic similarity across two modalities (i.e., binary and source code). To the best of our knowledge, most current methods are mainly concerning the matching within single modality, e.g., either source-to-source code matching [5], or binary-to-binary code matching [6]. Recently, several works have been proposed to investigate the binary-to-source matching problem. Yuan et al. [7] and Miyani et al [3] studied the binary-to-source matching for open-source software reuse detection and binary source code provenance, respectively. Yu et al. [8] studied the cross-modal matching of binary and source code at the function level. Both of these approaches extracted the semantic features of source code and binary code, and proposed two encoder networks to represent them as two hidden vectors. A similarity constraint (e.g., triplet loss function) is then designed to jointly learn these two encoder networks.

Despite much progress having been achieved on binary-source code matching, all current works are exclusively developed to detect binary-source clones from programs written in the same programming language. However, detecting binary-source code clones for programs written in different programming languages has made little progress in literature. In practice, software applications are often written in different programming languages to cater for different platforms. Therefore, detecting binary-source code clones across multiple programming languages is useful in real-world scenarios. For example, when we have a vulnerable binary code, it is necessary to retrieve all the relevant source code snippets for all possible programming languages they are written in, for better vulnerability assessment. To fill this gap, we, for the first time, formulate the problem of *cross-language binary-source code matching*.

Insights. The key challenge of binary-source code matching is to bridge the semantic gap between high-level programming language and low-level machine code even if they are in different textual appearance. Current approaches aim to align the semantic embeddings of binary code and source code in an end-to-end way. In compilers, intermediate representations are designed to support multiple front-end programming languages (e.g., C and Java) and multiple backend architectures (e.g., ARM and MIPS). That is, the intermediate representations

*Yao Wan is the corresponding author.

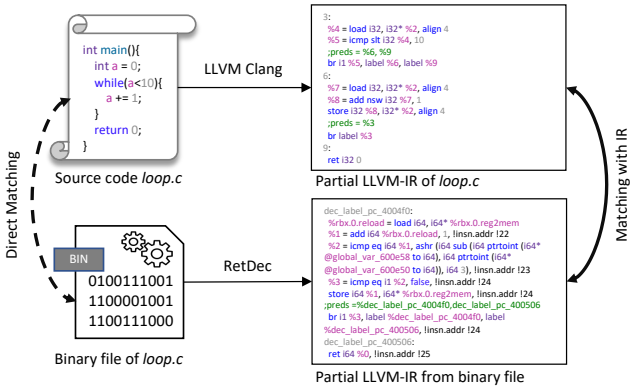

Figure 1: The insights of representing binary code and source code using LLVM-IR. A simple loop is implemented in both the source code file and the binary file, and the LLVM-IRs generated from both of them indicate the similar semantics.

are typically independent of programming languages and computer architectures, which can significantly reduce the gap between binary code and source code by sharing a similar word vocabulary and the syntax structure.

For better illustration, Figure 1 shows a source code fragment and a binary code fragment, together with their corresponding generated LLVM-IRs. We can transform the source code and binary code into LLVM-IR based on compiler tools, i.e., LLVM Clang[1] and RetDec[2], respectively. In this example, a `while` loop is implemented in both the source code file and the binary file. It is hard to find their semantic similarity from the textual appearance. However, we can find some clues from their LLVM-IRs. Figure 1 (right) shows the partial LLVM-IRs generated from source code and binary code, both of which represent the semantics of the `while` loop fragment. From other perspective, we can see that the intermediate representations can help unify the representations of source code and binary code of multiple programming languages and computer architectures.

Our Solutions and Contributions. Motivated by the aforementioned insights, this paper proposes XLIR, a novel approach based on Transformer, for the task of cross-language binary-source matching using intermediate representations (IRs). Specially, we parse both binary code and source code into intermediate representations. In this paper, we adopt the intermediate representation i.e., LLVM-IR which has been widely used in compiler optimization, program analysis, bug detection and verification. The LLVM-IR can be translated from multiple high-level programming languages (e.g., C/C++ and Java) as well as low-level machine code. To embed the intermediate representations, we adapt a Transformer-based neural network, which is first pre-trained by a masked language modeling on an external large-scale corpus of LLVM-IRs. We then map the LLVM-IR embeddings into a common space and jointly learn them using a triplet loss function.

[1]https://clang.llvm.org/
[2]https://retdec.com/

To the best of our knowledge, there is no dataset for cross-language binary-source code matching. For evaluation, we curated and contributed a new comprehensive dataset based on an existing dataset originally developed for cross-language source-source code matching. Experimental results and analysis show that XLIR significantly outperforms other state-of-the-art models. For the matching between Java binary code (compiled from corresponding LLVM-IR) and C source code, when comparing with the state-of-the-art tool B2SFinder, XLIR significantly improves the Precision, Recall and F1 from 0.35, 0.41 and 0.38 to 0.68, 0.55 and 0.61, respectively.

Overall, this paper makes the following major contributions:

- **New problem.** We, for the first time, formulate a new problem of cross-language binary-source code matching.
- **New insights.** Even though the source code and binary code are in different modalities, both of them can be transformed into IRs. In this paper, we provide an insight that it is feasible to mitigate the semantic gap between source code and binary code based on IRs (e.g., LLVM-IR). In addition to representing these LLVM-IRs, we propose an encoder network based on Transformer which is initialized by a pre-trained model for IR embedding.
- **Comprehensive experiments.** To validate the effectiveness of our proposed approach, we first curate a new dataset based on a public dataset that is used for cross-language source code clone detection. We then conduct comprehensive experiments on two tasks of cross-language source-source code matching, and cross-language binary-source code matching, on top of our curated dataset. Experimental results validate the effectiveness of our proposed approach when comparing with the state-of-the-art models.

II. Motivation

In this section, we first introduce the task of cross-language code clone detection, and then extend it to the cross-language binary-source code matching. We also present two practical scenarios of cross-language binary-source code matching.

A. Cross-Language Code Clone Detection

Detecting code clones is essential in software maintenance and refactoring. Existing efforts mainly focus on identifying code clones written in a single programming language. With the emerging of multiple-language platforms where applications are often written in different programming languages to cater for different requirements and computing platforms. It is useful to detect the code clones across different programming languages. For example, in the collaborative development of software, when a code fragment is modified by a Java developer, it is required to propagate the changes to the corresponding code fragment written by a C developer. Furthermore, when a developer finds a vulnerability in a code snippet written in one programming language, it is also useful to locate and identify the corresponding code fragment in other programming language. In cross-language code clone detection, the core insight is that although the source code fragments are written

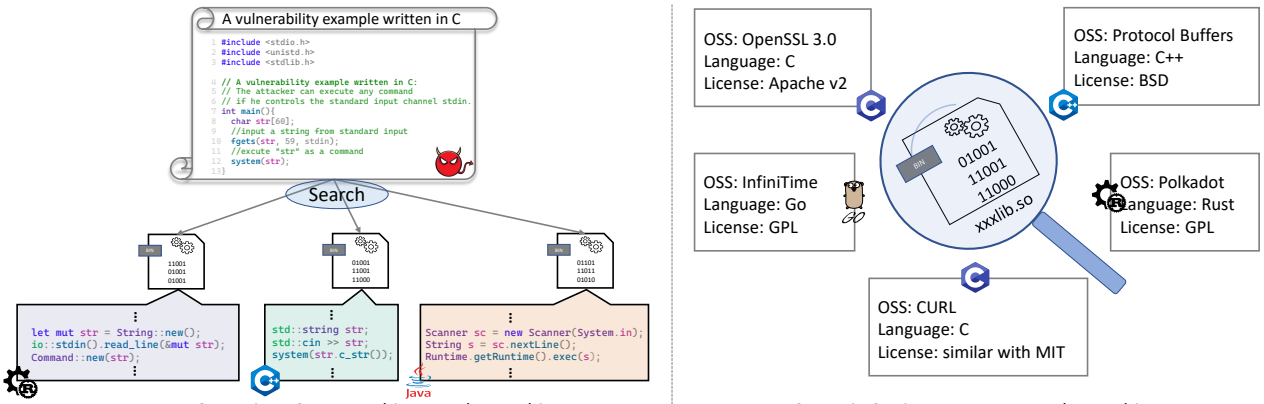

Figure 2: The motivating scenarios of cross-language binary-source code matching. (1) A real-world scenario of source-to-binary code matching in vulnerability detection. In this case, given a source code with vulnerability, using which the attacker can execute any command if he/she controls the standard input channel `stdin`. It is necessary for us to search for binary target files that may be written in other programming languages to check if there are similar backdoors in these binary target files. (2) A real-world scenario of binary-to-source code matching in copyright protection. Given a binary library file, we match it to a known open source software (OSS) library (or some of its components) to determine whether the binary file is derived from it, and further check whether the binary file follows the license in the original OSS.

in different programming languages with distinct textual appearance, they may share similar semantics.

B. Cross-Language Binary-Source Code Matching

In this paper, we extend the cross-language code clone detection (i.e., cross-language source-source code matching) to the cross-language binary-source code matching. We illustrate the motivation of this task using two real-world scenarios, as shown in Figure 2.

Scenario 1: source-to-binary code matching in vulnerability detection. As shown in Figure 2 (1), given a vulnerability source snippet written in C, in which the program accepts a string from the standard input stream and executes this string as a command. If the attacker has rights to write to the standard input stream, he/she can execute arbitrary commands on a target machine. The essence of this vulnerability is that the program directly uses the standard input stream as commands to execute, and this easy-to-implement pattern of backdoor may appear in existing binary files implemented by other programming languages. If we can match the vulnerable source code to binary code directly, we have a chance to find this vulnerability pattern in existing binary files.

Scenario 2: binary-to-source code matching in copyright protection. It is very common to introduce open source software (OSS) libraries written in different programming languages in software development, while different OSS libraries may use different open source licenses (e.g, GPL, Apache, BSD, MIT, etc.). Different licenses vary in details, for example, the GPL license requires that if a referrer modifies the original codes, it must also follow the GPL license. As shown in Figure 2 (2), given a binary library file, if we can find which OSS (or components of it) it may be derived from, we can check whether it follows the protocol in the original library, which is very meaningful for copyright protection. In this scenario, cross-language binary-to-source code matching can play an important role.

Directly matching a binary file composed of machine instructions with a source code file requires professional knowledge and skills and is usually very complicated, so an end-to-end method to solve this problem is desired.

III. Preliminaries

A. Intermediate Representation (IR)

The *Intermediate Representation* (IR) is a clearly-defined and well-formed representation of programs with generally simple syntax rules, used by a compiler while making transformation from source to target. Modern compilers first parse the source code, translate it into IR, and then generate target code from IR. This additional layer has a bi-directional independent property, i.e., the IR is independent from both the source code and the target machine, while keeping the semantics of a program. Hence, the IR forms the basis of our cross-language matching method. In this paper, we adopt the LLVM-IR, a specific type of IR first proposed by the LLVM infrastructure [9]. While containing the semantics of source code, the LLVM-IR is also in *Static Single Assignment* (SSA) form [10], in which any local variable is assigned exactly only once.

Originally implemented for C and C++, the language-agnostic design of LLVM has since spawned a wide variety of front ends (e.g., C#, and Rust). Source code written in common programming languages can be easily compiled to LLVM-IR, and binary files can also be decompiled to LLVM-IR through some tools. We convert both the source code and binary files into LLVM-IRs, and then process them through an encoder to obtain the latent vectors. After calculating the similarity between the vectors, we can detect source-source, source-binary,

or binary-source clones. In this way, we can achieve a universal end-to-end approach for code clone detection.

B. Code Embedding

Code embedding, also termed code representation learning, aims to preserve the semantics of programs into distributed vectors. It is essential in current deep learning-based program analysis. To the best of our knowledge, current code embeddings fall into four categorizes according to the code features they represent: token sequence, AST, IR, and code graphs. It is natural to represent the code based on its textual tokens, which reflect the lexical information of code. To represent the structural information of code, several works [11]–[13], [13]–[15] also propose to represent the AST and code graphs (such as control flow graph and data flow graph) using structural neural networks (e.g., TreeLSTM and GGNN). Recently, several works [16]–[18] propose to represent the low-level information of code using the IRs.

C. Problem Formulation

Here we formulate the problem of cross-language binary-source code matching formally. Note that, as an initial work, in this paper we limit our scope to matching the cross-language binary code and source code in files. Let $\mathcal{S}^{(l_u)} = \{s_1^{(l_u)}, s_2^{(l_u)}, \ldots, s_n^{(l_u)}\}$ denote a set of source code files that are written in programming language l_u, $\mathcal{B}^{(l_v)} = \{b_1^{(l_v)}, b_2^{(l_v)}, \ldots, b_n^{(l_v)}\}$ denote a set of binary code files that are built from the semantic-equivalent programs in $\mathcal{S}^{(l_u)}$ that are implemented in a different programming language l_v. When l_u and l_v denote the same programming language, the studied task will be reduced into the binary-source matching task that has been studied in [8]. Given the paired source code files with their corresponding binary code files, the goal of this paper is to respectively learn the embeddings of both of them, and then align these embeddings in a common space. The intuition is that the embeddings of paired source code and binary code in ground truth should be similar, while the embeddings of unpaired source code and binary code should be distinct as much as possible.

Formally, given a source code file $s_i^{(l_u)}$ and binary code file $b_j^{(l_v)}$, the embeddings of them are denoted as $\mathbf{s}_i^{(l_u)}$ and $\mathbf{b}_i^{(l_v)}$, respectively. We map the embeddings of source code and binary into a common feature space via ϕ and Φ, respectively.

$$\mathcal{S} \xrightarrow{\phi} V_{\mathcal{S}} \rightarrow J(V_{\mathcal{S}}, V_{\mathcal{B}}) \leftarrow V_{\mathcal{B}} \xleftarrow{\Phi} \mathcal{B}\,, \tag{1}$$

where $J(\cdot,\cdot)$ denotes the similarity function, e.g., cosine similarity, which is designed to measure the matching degree of $V_{\mathcal{S}}$ and $V_{\mathcal{B}}$, in order to learn the mapping functions.

In this paper, we argue that source code and binary code are in distinct feature space, we propose to first map them into a closer feature space of IRs. Generally, we can parse both source code and binary code into IRs, and then jointly learn their embeddings based on the IRs. Therefore, the Eq. 1 can be reformulated as follows:

$$\mathcal{S} \xrightarrow{\text{parser}} \mathcal{S}_r \xrightarrow{\phi} V_{\mathcal{S}} \rightarrow J(V_{\mathcal{S}}, V_{\mathcal{B}}) \leftarrow V_{\mathcal{B}} \xleftarrow{\Phi} \mathcal{B}_r \xleftarrow{\text{parser}} \mathcal{B}\,. \tag{2}$$

Eq. 1 and Eq. 2 show that we can transform the problem of matching source code and binary code from their original textual representation to the mid-level IRs.

IV. Cross-Language Binary-Source Code Matching

In this section, we present XLIR, which is designed for cross-language binary-source code matching with IRs.

A. An Overview

Figure 3 shows an overview of our proposed XLIR. The model training phase is composed of the following three steps: (1) *Transforming Source and Binary Code into IRs (cf. Sec. IV-B).* We first parse both the binary code and source code that are from different programming languages into IRs via several compiler tools (i.e., LLVM Clang[3] and JLang[4]). Currently, we can support the C, C++ and Java programming languages. (2) *Transformer-based IR embedding (cf. Sec. IV-C).* To represent the generated IRs, we feed them into a pre-trained Transformer-based language model (i.e., IR-BERT) for IR embedding. We pre-train a masked language model on a large-scale IR corpus, following the CodeBERT [19] and OSCAR [20], which are pre-trained models on code corpus and IR corpus, respectively. (3) *Model learning (cf. Sec. IV-D).* To correlate the embeddings of paired binary code and source code, we first map them into a common feature space, and jointly learn their correlations.

As the model trained, at the code matching phase, the cosine similarity is applied to measure the semantic similarity between binary code and source code. The matching score greater than a pre-defined threshold indicates that the binary code and source code from cross languages are matched.

B. Transforming Source and Binary Code into IRs

Without loss of generality, we choose C, C++ and Java as source programming languages in cross-language scenario, since the parser has greatly developed in these mature procedural languages. Additionally, we also choose the LLVM-IR as an intermediate representation, because (1) the LLVM-IR is source-independent, which yields different programming languages sharing the same semantics will keep similar IR structure; (2) the LLVM-IR is also target-independent and translation from LLVM-IR to any target-dependent assembly code is easy in practical; and (3) the LLVM-IR is widely recognized by the community, which can ease the code transformation, e.g., decompilation, optimization, and semantics extraction.

For the sake of rigor, in order to avoid leakage of string information such as function and variable names into the binary file during compilation, we pass the "`-s`" parameter to compiler to strip out all debug information when compiling.

At the stage of training data preparation, we exploit a variety of tools to transform different program representations into LLVM-IR. The procedure applies to two major representations of program: source code and binary code. For source code, we use LLVM Clang[1] to emit LLVM-IR from C and C++, which

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

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

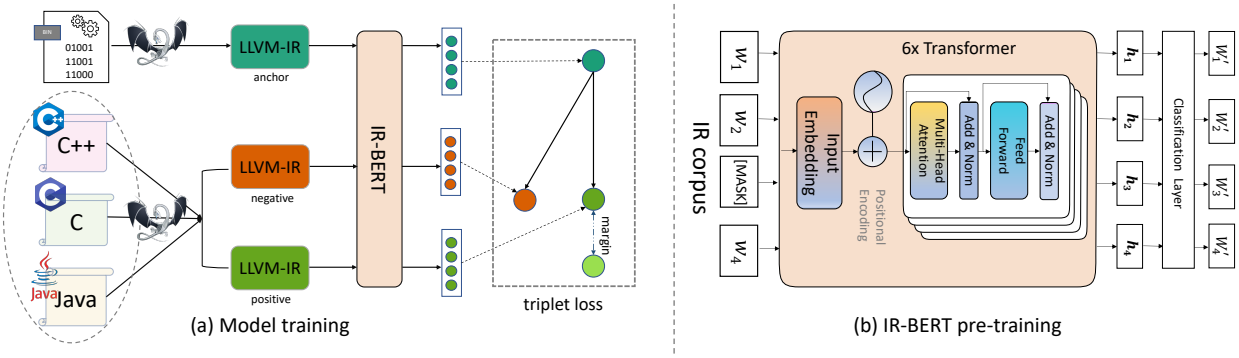

Figure 3: An overview of our proposed XLIR. In the model training phase (a), we first parse both the binary code and source code into IRs via several compiler tools. Then we feed the generated IRs into a pre-trained language model (i.e., IR-BERT based on Transformer for IR embedding (b). We learn the whole network by jointly mapping the embeddings of binary code and source code in a common space.

is officially supported by LLVM community. We use Jlang[4] and Polyglot[5] to translate Java into LLVM-IR. For binary code, we use the RetDec decompiler[2] to convert non-obfuscated binary files into LLVM-IR. Obfuscation is the deliberate act of creating source or binary code that is difficult to understand, the semantic information in obfuscated code is considered harder to extract. However, there is only a limited range of code in this form.

Note that, LLVM-IR has two different representations holding exactly the same semantic information. The *bitcode* format is intentionally designed for machine processing, e.g., code transformation and automatic optimization. The *machine IR* (MIR) format is human-readable and has been widely used in program debugging and analysis. For the sake of efficiency, we make use of the bitcode format as a universal representation of IR for code embedding.

C. Transformer-based LLVM-IR Embedding

We adapt Transformer [21] for LLVM-IR embedding. Transformer is based on the self-attention mechanism and has become a popular model for source code embedding. As shown in Figure 3(b), a Transformer model is composed of K layers of blocks, which can encode a sequence of instructions into contextual representation at different levels: $\mathbf{H}^k = [\mathbf{h}_1^k, \mathbf{h}_2^k, \ldots, \mathbf{h}_n^k]$, where k denotes the k-th layer. For each layer, the layer representation $\mathbf{H}^k$ is computed by the k-th layer Transformer block $\mathbf{H}^k = \text{Transformer}_k(\mathbf{H}^{k-1})$, $k \in \{1, 2, \ldots, K\}$.

In each Transformer block, multiple self-attention heads are used to aggregate the output vectors of the previous layer. A general attention mechanism can be formulated as the weighted sum of the value vector $\mathbf{V}$ using the query vector $\mathbf{Q}$ and the key vector $\mathbf{K}$:

$$\text{Att}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{model}}}}\right) \cdot \mathbf{V}\,, \tag{3}$$

where d_{model} represents the dimension of each hidden representation. For self-attention, $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are mappings of previous hidden representation by different linear functions, i.e., $\mathbf{Q} = \mathbf{H}^{l-1}\mathbf{W}_Q^l$, $\mathbf{K} = \mathbf{H}^{l-1}\mathbf{W}_K^l$, and $\mathbf{V} = \mathbf{H}^{l-1}\mathbf{W}_V^l$, respectively. At last, the encoder produces a final contextual representation $\mathbf{H}^L = [\mathbf{h}_1^L, \ldots, \mathbf{h}_n^L]$, which is obtained from the last Transformer block.

Pre-Training of IR-BERT. Following [20], we first pre-train a masked language model on a large-scale external LLVM-IR corpus, termed IR-BERT, and then transfer the parameters into our model. Given a LLVM-IR corpus, each LLVM-IR is first tokenized into a series of tokens, using Byte Pair Encoding (BPE [22]). Before pre-training, we first take the concatenation of two segments as the input, defined as $c_1 = \{w_1, w_2, \ldots, w_n\}$ and $c_2 = \{u_1, u_2, \ldots, u_m\}$, where n and m denote the lengths of two segments, respectively. The two segments are always connected by a special separator token `[SEP]`. The first and last tokens of each sequence are always padded with a special classification token `[CLS]` and an ending token `[EOS]`, respectively. The concatenated sentences are then fed into a Transformer encoder as input. In the masked language modeling, the tokens of an input sentence are randomly sampled and replaced with the special token `[MASK]`. In practice, we uniformly select 15% of the input tokens for possible replacement. Among the selected tokens, 80% are replaced with `[MASK]`, 10% are unchanged, and the left 10% are randomly replaced with the selected tokens from vocabulary [23]. Without loss of generality, we adopt the pre-trained model in [20], which can be seen as a variant of IR-BERT.

D. Model Learning

We learn XLIR by mapping the embeddings of binary code and source code into a common space, with a similarity constraint. The intuition is that if a binary code and a source code have similar semantics, their embeddings should be close to each other. Let triplet $\langle b, s^+, s^- \rangle$ denote a training instance, in which for binary code b, s^+ denotes the corresponding source code in compilation (also termed positive sample or anchor), s^- denotes a negative code snippet that is randomly chosen from the collection of all source code files. When training

[5]https://www.cs.cornell.edu/projects/polyglot/

on the set of $\langle b, s^+, s^- \rangle$ triplets, XLIR predicts the cosine similarities of both $\langle b, s^+ \rangle$ and $\langle b, s^- \rangle$ pairs and minimizes the ranking loss [24] as follows:

$$\mathcal{L} = \sum_{\langle b, s^+, s^- \rangle \in \mathcal{D}} \max(0, \alpha - \text{sim}(\mathbf{b}, \mathbf{s}^+) + \text{sim}(\mathbf{b}, \mathbf{s}^-)), \quad (4)$$

where $\mathcal{D}$ denotes the training dataset, sim denotes the similarity score between the binary code and source code, and α is a small constant margin. $\mathbf{b}$, $\mathbf{s}^+$ and $\mathbf{s}^-$ are the embeddings of b, s^+ and s^-, respectively. In this paper, we adopt the cosin similarity function and set α to 0.06 by default.

E. Code Matching

At the inference phase, given a binary code b, as well a set of source code files $\mathcal{S}$, For each source code file $s \in \mathcal{S}$, we first feed the binary code and source code files into our trained model and obtain their corresponding embeddings, denoted as $\mathbf{b}$ and $\mathbf{s}$. Then we calculate the matching score between $\mathbf{b}$ and $\mathbf{s}$ as follows:

$$\text{sim}(b, s) = \cos(\mathbf{b}, \mathbf{s}) = \frac{\mathbf{b}^T \mathbf{s}}{\|\mathbf{b}\| \|\mathbf{s}\|}, \quad (5)$$

where $\mathbf{b}$ and $\mathbf{s}$ are the vectors of binary code and source code, respectively. If the matching score is larger than a threshold, we consider the pair of binary code and source code as matched, otherwise unmached. Generally, 80% is used as the similarity threshold for code clone detection. In our experiments, unless otherwise specified, this value is used as default. We also evaluate the impact of threshold in Section VI.

V. Experimental Setup

A. Research Questions

We conduct experiments to answer the following research questions.

- **RQ1:** Is our proposed XLIR effective in cross-language binary-source code matching?
- **RQ2:** What is the effectiveness of our proposed XLIR in single-language binary-source code matching?
- **RQ3:** Can our approach with IRs be extended to detect source-source code functionality clones effectively?
- **RQ4:** What is the influence of major factors of XLIR?

B. Evaluated Tasks and Dataset

Cross-Language Source-Source Code Matching. This task aims to detect the source code clones across different programming languages, and has been studied previously in [25]. In [25], the authors introduced the CLCDSA dataset[6], which is composed of code snippets across four programming languages (i.e., C++, C#, Java and Python), collected from two online judge and contest platforms (i.e., AtCoder[7] and Google CodeJam[8]). In this dataset, each programming problem is affiliated with multiple solutions implemented in different programming languages. To validate the effectiveness our proposed XLIR on cross-language source-source code matching, we choose C, C++, and Java as the studied languages. Since our approach depends on IRs, all the source code files are supposed to be compiled, we filter out files that are unable to be successfully compiled, and divide the dataset into training, validation and test set according to a ratio of 6:2:2. Table I shows the statistics of filtered dataset used in this paper.

Table I: An overview of the CLCDSA [25] dataset for cross-language binary-source code matching.

		Train	Validation	Test
AtCoder	C	4,772	1,672	1,743
	C++	4,606	1,605	1,692
	Java	4,856	1,706	1,770
CodeJam	C	1,168	409	447
	C++	1,176	410	445
	Java	1,105	379	397
Total		17,683	6,181	6,494

Cross-Language Binary-Source Code Matching. Currently, there is no available dataset for the evaluation of cross-language binary-source code matching. To mitigate this gap, we curate a new dataset based on the CLCDSA dataset, by compiling the source code in one programming language into binary code, while keeping the source code in another programming language unchanged. In particular, we compile each source file into binary files using different compilers (i.e., GCC and LLVM Clang), with multiple optimization options (i.e., `-O0`, `-O1`, `-O2`, and `-O3`), across multiple platforms (i.e., `x86-32`, `x86-64`, `arm-32`, and `arm-64`). Consequently, each source code file is compiled into 32 different object files. Under the single-language setting, we conduct experiments mainly on POJ-104 [26], which is a public dataset composed of about 50,000 programs written in C and C++, we also collected data from a variety of online judge and contest platforms. In our dataset, each problem is affiliated with around 500 solutions.

Dataset for Pre-Training. We first pre-train an IR-BERT on a separate large-scale IR corpus. We adopt the dataset that has been used in [20], which is composed of eleven real world popular softwares (i.e., Linux-vmlinux, Linux-modules, GCC, MPlayer, OpenBLAS, PostgreSQL, Apache, Blender, ImageMagcick, Tensorflow, Firefox) from GitHub. We can compile these softwares into LLVM-IRs using LLVM Clang with `-O0` optimization level. Finally, 48,023,781 LLVM-IR instructions from 855,792 functions are obtained.

C. Baselines

We evaluate our proposed XLIR on two code matching tasks (i.e., cross-language binary-source code matching and cross-language source-source code matching). For each task, the performance of our model is compared XLIR with the following state-of-the-art baselines.

Cross-Language Binary-Source Code Matching. We extend the baselines for binary-source code matching, from single-language setting to cross-language setting.

- **BinPro** [3] extracts function call graphs (FCGs) for both binary and source code, and uses a bipartite matching algorithm (i.e., Hungarian algorithm [27]) to match them.

[6] https://github.com/Kawser-nerd/CLCDSA/

[7] https://atcoder.jp/

[8] https://codingcompetitions.withgoogle.com/codejam/

Table II: Performance of cross-language binary-source code matching (BinPro and B2SFinder do not support Java, and Java binary code refers to the binary code compiled from the corresponding LLVM-IR).

	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

Table III: Performance of single-language C++ binary code to C++ source code matching on POJ-104 dataset

	Precision	Recall	F1
BinPro	0.38	0.42	0.40
B2SFinder	0.43	0.46	0.44
XLIR (LSTM)	0.67	0.72	0.69
XLIR (Transformer)	0.85	0.86	0.85

- **B2SFinder** [7] extracts seven features from three perspectives (i.e., strings, integers and control-flow) for both binary and source code, and introduces a weighted matching algorithm to match them.
- **XLIR (LSTM)** is a variant of our proposed approach, in which LSTM [28] is used to encode the IRs.

Cross-Language Source-Source Code Matching. We first extend our proposed XLIR to the task of cross-language source-source code matching, and then compare XLIR with the following baselines.

- **LICCA** [29] is a tool for detecting source-to-source code clones across programming languages based on syntactic and semantic features of code. It has been verified in Java, C, JavaScript, Modula-2 and Scheme.
- **XLIR (LSTM)** is a variant of our proposed approach, in which the LSTM [28] is used to encode the IRs. This approach has been adapted to the task of cross-language source-source code matching.

D. Evaluation Metrics

We adopt recall, precision and F1-score for model evaluations, which have been widely used in text matching and information retrieval [30]. For a query code snippet (source or binary code), we call the corresponding cloned code snippet as positive, and the non-cloned code snippet as negative. Let T_p and F_p be the number of positive clones detected true and false, and F_n be the number of negative clones detected false, the precision P and recall R is calculated as follows:

$$P = \frac{T_p}{T_p + F_p},\ R = \frac{T_p}{T_p + F_n}. \tag{6}$$

We also use F1 for evaluation, which is defined as the harmonic mean of Precision and Recall. It can be interpreted as a trade-off between them. Formally, F1 is defined as follows:

$$F1 = 2 \cdot \frac{P \cdot R}{P + R}. \tag{7}$$

E. Implementation Details

We implement XLIR with PyTorch 1.9. As mentioned above, the encoder for LLVM-IR is based on the Transformer. For Transformer, the settings of each encoder are the same as BERT [23]. For the masking strategy, we take random 15% instructions from IR. For these selected instructions, we replace them with the `[MASK]` token and random characters with 80% and 10% probability, and keep them unchanged with 10% probability. We set the hidden size and word embedding size to 256. In pre-training phase, we generate a dictionary from the LLVM-IRs compiled from a large code corpus. When fine-tuning our model on the clone detection task, unrecognized characters will be replaced with $\langle$`UNK`$\rangle$.

We conducted experiments on a Linux server having four Tesla V100 GPU of 32GB memory. Our model training procedure is distributed on all four GPUs, and the parameters are updated via ADM optimizer, with the learning rate of $1e$-3 for training. To prevent over-fitting, we use a dropout of 0.4.

VI. Experimental Results and Analysis

A. RQ1: Effectiveness of IR for Cross-Language Binary-Source Code Matching

In order to verify the performance of our proposed model, we conduct experiments on our curated dataset based on CLCDSA. Table II shows the performance of binary-source code matching between binary files and source files written in C/C++ and Java. From this table, we can see that our algorithm is capable of detecting cross-language binary-source code clones. In the task of C/C++ binary to Java source code matching, precision, recall and F1 are as high as 0.73, 0.59 and 0.65, respectively, and in the task of matching Java binary and C source code, they are 0.68, 0.55 and 0.61, respectively. We have noticed that C/C++ and Java, source code files and binary files are quite different. Because of these differences, our results are quite encouraging. The results reveal that although source code and binary file are written in different programming languages and in different forms, functionality clone can be detected after they are converted into LLVM-IR by our approach. We think this may be attributed to the equivalent conversion of semantic information when source code files and binary files are transformed into LLVM-IR.

B. RQ2: Effectiveness of IR for Single-Language Binary-Source Code Matching

Since the CLCDSA dataset with a total of only about 30,000 source files after screening is relatively small, we extended the experiment of binary-source clone detection to the single-language dataset POJ-104 to further verify the performance of our approach in binary-source code matching. Table III shows the performance of C++ binary code to C++ source

Table IV: Performance of cross-language code clone detection.

	LICCA			XLIR (LSTM)			XLIR (Transformer)		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
C&C++	0.43	0.37	0.40	0.78	0.65	0.71	0.92	0.86	0.89
C&Java	0.31	0.29	0.30	0.62	0.51	0.56	0.75	0.55	0.63
C++&Java	0.33	0.29	0.31	0.65	0.53	0.58	0.77	0.57	0.66

Table V: Performance of XLIR on cross-language code clone detection without pre-training.

	Precision	Recall	F1
C&C++	0.88	0.84	0.86
C&Java	0.71	0.52	0.60
C++&Java	0.72	0.54	0.62

code matching. We can see that our XLIR outperforms all baselines by large margins, e.g., F1 score of XLIR is higher than BinPro and B2SFinder by 0.42 and 0.41, respectively. This shows the effectiveness of our method in binary-source code matching.

C. RQ3: Extended Evaluation on Cross-Language Source-Source Code Matching

We conduct code clone detection between C, C++ and Java, and the results are shown in Table IV. When the similarity threshold is 80%, no matter our XLIR uses Transformer or LSTM as the encoder, the performance exceeds LICCA by a large margin. Specially, XLIR achieves an average precision of 0.81, an average recall rate of 0.77, and an average F1 score of 0.73. This shows that our method can effectively detect cross-language functional clones. We can also see that the similarity between C and C++ is higher than that of between C/C++ and Java, because the basic syntax of C and C++ is very similar.

D. RQ4: Influence of Major Factors

We study the influence of the following major factors on our approach separately, i.e., pre-training, Transformer encoder, compilation options, and similarity threshold.

Impact of Pre-Training. To verify the impact of pre-training, we conduct the experiment of cross-language code clone detection without pre-training, and the results are shown in Table V. Compared with the performance with pre-training in Table IV, we can see the performance of our approach without pre-training deteriorates by a small but perceptible margin, i.e., Precision, Recall, F1 decreased by a an average of 0.04, 0.03 and 0.03, respectively. This shows the obvious effectiveness of pre-training on external large-scale datasets.

Contribution of the Transformer Encoder. To verify the effectiveness of Transformer encoder in our approach, we conduct experiments of cross-language binary-source clone detection and source-source clone detection using LSTM as encoder for LLVM-IR embedding instead of Transformer. The experimental results are shown in Tables II, III, and IV. The three tables show that our model with Transformer encoder outperforms that with LSTM encoder in terms of all evaluation metrics, on all the three tasks. We attribute it to the fact that LLVM-IRs are always long sequences, while the Transformer works better on representing them.

Table VI: Performance of code-binary clone detection on POJ-104.

		X86			ARM		
		Precision	Recall	F1	Precision	Recall	F1
gcc	-O0	0.87	0.84	0.85	0.84	0.87	0.85
	-O1	0.84	0.86	0.85	0.86	0.88	0.87
	-O2	0.89	0.85	0.87	0.83	0.84	0.87
	-O3	0.89	0.88	0.88	0.88	0.82	0.85
clang	-O0	0.85	0.86	0.85	0.83	0.87	0.85
	-O1	0.83	0.85	0.84	0.81	0.86	0.83
	-O2	0.84	0.83	0.83	0.84	0.87	0.85
	-O3	0.87	0.82	0.84	0.89	0.82	0.85

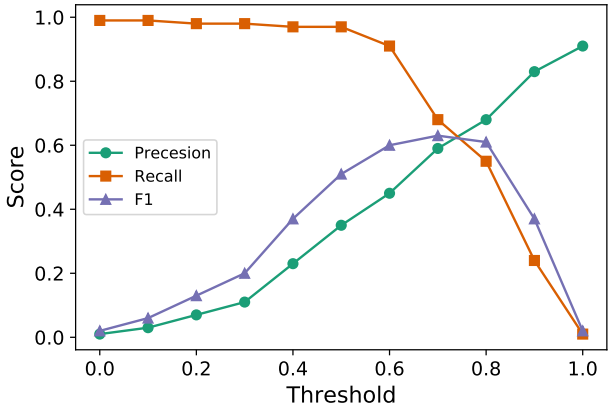

Figure 4: Influence of threshold on Java binary to C++ source code matching.

Influence of Compilation Options. In order to investigate the impact of different compilation options on binary-source clone detection, we carried out an experiment on dataset POJ-104: we use different compilation options of compilers, platforms and optimisations to compile POJ-104 into binary files (Linux elf files) , and then conduct clone detection between the source files and binary files. The results are shown in Table VI. We can see that our approach can consistently achieve good performance for various binary files. In particular, the average of precision, recall and F1 is about 0.85, and the variances are 0.0006, 0.0004, 0.0002, respectively. These results reveal that our approach is effective in matching source code and different binary files.

Influence of Threshold. In our experiment, we set 0.8 as the default threshold value, that is, when the similarity is greater than or equal to 0.8, the code pair is considered to be cloned. In order to explore the impact of threshold values on the final results, we adjust the threshold and carry out a series

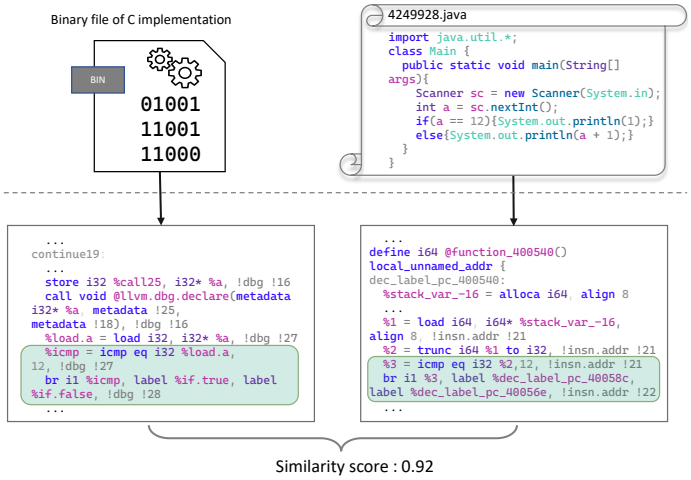

Figure 5: A case study to show how Java source code matchs with C binary code.

of experiments on Java binary to C++ source code matching. The experimental results are shown in Figure 4. From this figure, we can see that when the threshold rises from 0.5 to 0.98, the precision increases significantly while the recall drops quickly. This result is consistent with our intuition: the higher the similarity, the higher the precision; the lower the similarity, the higher the recall rate. When the threshold is between 0.7 and 0.8, a balanced point is achieved. In our work, we set the threshold as 0.8 by default.

VII. Discussion

A. Case Study

Figure 5 shows a real example of how a C binary file matches a Java source file. The two images in the upper side of the figure show a C binary file and a Java source file for the same problem *ABC011/A*[9] in the programming contest website AtCoder. The problem is: *given the input value n, if n equals to 12, outputs $n+1$, otherwise outputs 1*. The two code snippets in the bottom of Figure 5 are extracted from the LLVM-IR obtained by disassembling the binary object file and compiling the LLVM-IR of Java source code, respectively. We can see that the two LLVM-IR code snippets have equivalent semantics. Both of them contain a branch, conditioned on whether the input value n equals to 12 or not. The previous input instructions and the subsequent output-related instructions are omitted based on the space limitation. From this example we can see that an important reason why our method can work is that the semantic information is equivalently retained after converting to LLVM-IR.

B. Strength of XLIR

We have identified two advantages of XLIR: (1) Compared with some previous methods based on mainly extracting code literals in binary target file and specific characteristics in source code file, we transform source code file and binary target file into semantically equivalent LLVM-IRs for code matching task, which can greatly reduce the loss of program semantic information compared with extracting some manually selected features. (2) Our method is an end-to-end binary-source and source-source matching approach, so users do not need to grasp complicated reverse engineering skills to use this tool. This can greatly improve the efficiency of code clone detection and can be useful for software security related tasks.

C. Threats to Validity and Limitations

The first threat is that the source code for clone detection in our approach must be compilable, and the binary object file can be disassembled into LLVM-IR. In a few scenarios, there may be cases where incomplete or grammatically incorrect code snippets cannot be compiled but still need to be checked. Furthermore, parsing a binary object file that is seriously obfuscated and cannot be decompiled has always been a challenging problem in reverse engineering and is not the focus of this paper. The tool RetDec we use is capable of decompiling binary object files to LLVM-IR in most cases.

Another limitation of our method is that it only supports programming languages that have static LLVM compilers. Although many programming languages can be easily compiled by LLVM compilers, such as C/C++, Rust, Java, Ruby, CUDA, LUA, Objective-C, C#, OpenCL, etc., there are still some commonly-used programming languages that can not. For example, Python has only JIT's LLVM compiler due to the characteristic of being a dynamic programming language, that is, Python can only be translated into LLVM-IR during runtime. However, we believe that more and more programming languages will support static compilation to LLVM-IR, because LLVM has great potential in compiler optimization.

VIII. Related Work

A. Deep Learning for Source Code

The last few years have witnessed increasing interests on applying deep learning for source code modeling, so as to build intelligent tools to increase the productivity of software developers. One fundamental task is code embedding, which can support a variety of downstream tasks, including code search [31], [32], code summarization [33]–[35], code completion [36]–[39] and code clone detection [5], [40]–[42]. From our investigation, current approaches mainly represent source code in four perspectives, i.e., sequential code tokens, ASTs, code graphs, and IRs. It is natural to simply represent the semantics of program as a sequence of tokens, like that in NLP. Current approaches mainly tokenize the program into sequential tokens by several special separators, e.g., whitespace or Camel cases (for identifiers like `SortList` and `intArray`). To represent the structured syntactic information inside a program, one line of work use ASTs in the form of tree or graph for network feeding, this type of networks including Tree-CNN [11], Tree-LSTM [12] and GGNN [13]. Furthermore, another line of work represent the AST by serializing the structured AST into a list of instructions, so that sequential learning methods can be used [33], [43]. To add information

[9]http://atcoder.jp/contests/abc011/tasks/abc011_1/

from different perspective, a program can be translated into different representations. Augmented ASTs hold more detailed properties for a node [13], data-flow graphs [15] and control-flow graphs [14] tend to skeletonize the program by highlighting execution paths and changing of variables. Recently, some works resort to represent a program by IR, which is independent to programming languages and platforms [16]–[18]. Benefited from the pre-training techniques in processing natural language tasks [44], Feng et al. [19] pre-trained CodeBERT for the bimodal of programming language and natural language, which has shown promising results in various code-related tasks, such as code search and code summarization. Furthermore, Guo et al. [15] proposed GraphCodeBERT to advance CodeBERT by incorporating data-flow information into pre-training.

B. Code Clone Detection

It is a fundamental task to detect similar code (or clone code) in many software engineering tasks (e.g, code reuse, code summarization, and bug detection). Code clones can be roughly categorized into the following types: Type-1, Type-2, Type-3 and Type-4. CCFinder [45] extracted a series of tokens from code file and transform it according to several rules into a regular form for Type-1 and Type-2 clone detection. NICAD [46] introduced a two-stage approach which first finds and regulates potential clones to remove noise using pretty-printing and then enumerate potential clones using dynamic clustering. There are a variety of methods operation at different level to represent the syntax and semantic structure of a program. Jiang et al. [47] proposed Deckard, which incorporates ASTs into code representation learning use locality sensitive hashing for efficient clustering. To detect Type-3 clone, SourcererCC [48] was designed to capture the shared similarity of tokens among multiple approaches.

Recently, deep neural networks are applied to the task of code mathing. For example, White et al. [49] proposed DLC, which takes the lexical and syntactic information of code into account, and designs Recurrent Neural Networks to represent them. To better express structured syntactic information of code, Wei et al. [50] proposed to represent the syntactic information of code using TreeLSTM over ASTs. Furthermore, Zhang et al. [40] decomposed the AST into sentence-based abstract syntax subtrees, and proposed a two-way loop network for representation. This method has achieved good results in code matching. Zhao et al. [5] proposed to consider the data flow and control flow of source code, and proposed a deep learning framework for code representation. Based on the control-flow graph of code, Wu et al. [41] introduced a centrality analysis method from the perspective of social network analysis, which is efficient and effective in source code matching. Zhang et al. [40] proposed ASTNN, an AST-based model for code embedding, which decomposes a big AST into a series of limited-scale statement trees. ASTNN has achieved promising performance in code clone detection.

C. Cross-Language Source Code Analysis

With the recent progress in transfer learning, transferring knowledge across different programming languages has become a promising research direction. Xia et al. [51] studied the problem of cross-language bug localization based on language translation, which focuses on ranking source code files based on comments written in different natural languages. Chen et al. [12] proposed a tree-to-tree approach to transform programs from one language into another [12]. Bui et al. [52] proposed a bilateral model of two encoders, each of which is for encoding the abstract syntax of code in one programming language. Bui et al. [53] proposed to improve program translation via mining API mappings across programming languages based on adversarial learning. Nafi et al. [25] proposed an approach for cross-language source clone matching based on structured syntactic features and code API documentation. Gu et al. [54] proposed DeepAM, which can automatically mine API mappings between two languages from code corpus with single-language projects. Our paper is the first to study binary-source code matching across different programming languages.

IX. Conclusion and Future Work

In this research, we have formulated a new problem of cross-language binary-source code matching. We also propose a novel approach, termed XLIR, based on Transformer and intermediate representations of programs. Comprehensive experiments are conducted on two tasks of cross-language binary-source code matching, and cross-language source-source code matching, over a created cross-language dataset. Experimental results and analysis show that XLIR significantly outperforms other state-of-the-art models. For the matching of Java binary code to C source code, when comparing with B2SFinder, XLIR significantly improves the Recall, Precision, and F1 from 0.41, 0.35 and 0.38 to 0.55, 0.68 and 0.61, respectively.

Due to the challenging nature of the cross-language binary-source code matching problem, there is still ample room for enhancing the strength of our XLIR. In our future work, we will design more effective mechanisms to further improve the accuracy of code matching. We also plan to extend our approach to support smaller clone detection granularity, such as clone detection on a single function or small code snippet.

Artifacts. All the experiments in this paper are integrated into the open-source toolkit NaturalCC [55]. Our datasets and source code used in this work are available at **https://github.com/CGCL-codes/naturalcc**.

Acknowledgements

This work is supported by National Natural Science Foundation of China under grand No. 62102157, 61972444. This work is also partially sponsored by Tencent Rhino-Bird Focus Research Program of Basic Platform Technology. We would like to thank all the anonymous reviewers for their constructive comments on improving this paper.

REFERENCES

[1] R. T. Yarlagadda, "Aproach to computer security via binary analytics," *International Journal of Innovations in Engineering Research and Technology*, 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] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "Codecmr: Cross-modal retrieval for function-level binary source code matching," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[9] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[10] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 12–27. [Online]. Available: https://doi.org/10.1145/73560.73562

[11] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "Tbcnn: A tree-based convolutional neural network for programming language processing," *arXiv preprint arXiv:1409.5718*, 2014.

[12] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 2552–2562.

[13] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 184–195.

[14] 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.

[15] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[16] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 3589–3601.

[17] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *arXiv preprint arXiv:1909.06228*, 2019.

[18] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020, pp. 201–211.

[19] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[20] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T. Liu, "How could neural networks understand programs?" in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 2021, pp. 8476–8486.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[22] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[24] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[25] 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.

[26] 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.

[27] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 201–213.

[28] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.

[29] 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)*. IEEE, 2018, pp. 512–516.

[30] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," *arXiv preprint arXiv:2010.16061*, 2020.

[31] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 631–642.

[32] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. S. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 13–25.

[33] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[34] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[35] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[36] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, "Neural code completion," 2016.

[37] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2727–2735.

[38] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[39] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings*

of the 28th International Conference on Program Comprehension, 2020, pp. 37–47.

[40] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[41] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, "Scdetector: Software functional clone detection based on semantic tokens analysis," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 821–833.

[42] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2020.

[43] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[44] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[45] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[46] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th iEEE international conference on program comprehension*. IEEE, 2008, pp. 172–181.

[47] 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)*. IEEE, 2007, pp. 96–105.

[48] 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*, 2016, pp. 1157–1168.

[49] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[50] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[51] X. Xia, D. Lo, X. Wang, C. Zhang, and X. Wang, "Cross-language bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 275–278.

[52] N. D. Bui, Y. Yu, and L. Jiang, "Bilateral dependency neural networks for cross-language algorithm classification," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 422–433.

[53] N. D. Q. Bui, Y. Yu, and L. Jiang, "SAR: learning cross-language API mappings with little knowledge," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 796–806.

[54] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deepam: Migrate apis with multi-modal sequence to sequence learning," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, p. 3675–3681.

[55] Y. Wan, Y. He, J.-G. Zhang, Y. Sui, H. Jin, G. Xu, C. Xiong, and P. S. Yu, "Naturalcc: A toolkit to naturalize the source code corpus," *arXiv preprint arXiv:2012.03225*, 2020.