Pathways to Leverage Transcompiler based Data Augmentation for Cross-Language Clone Detection

Subroto Nag Pinku
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
subroto.npi@usask.ca

Debajyoti Mondal
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
d.mondal@usask.ca

Chanchal K. Roy
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
chanchal.roy@usask.ca

Abstract—Software clones are often introduced when developers reuse code fragments to implement similar functionalities in the same or different software systems resulting in duplicated fragments or code clones in those systems. Due to the adverse effect of clones on software maintenance, a great many tools and techniques and techniques have appeared in the literature to detect clones. Many high-performing clone detection tools today are based on deep learning techniques and are mostly used for detecting clones written in the same programming language, whereas clone detection tools for detecting cross-language clones are also emerging rapidly. The popularity of deep learning-based clone detection tools creates an opportunity to investigate how known strategies that boost the performances of deep learning models could be further leveraged to improve the clone detection tools. In this paper, we investigate such a strategy, data augmentation, which has not yet been explored for cross-language clone detection as opposed to single language clone detection. We show how the existing knowledge on transcompilers (source-to-source translators) can be used for data augmentation to boost the performance of cross-language clone detection models, as well as to adapt single-language clone detection models to create cross-language clone detection pipelines. To demonstrate the performance boost for cross-language clone detection through data augmentation, we exploit Transcoder, which is a pre-trained source-to-source translator. To show how to extend single-language models for cross-language clone detection, we extend a popular single-language model, Graph Matching Network (GMN), in a combination with the transcompilers and code parsers (srcML). We evaluated our models on popular benchmark datasets. Our experimental results showed improvements in F1 scores (sometimes up to 3%) for the cutting-edge cross-language clone detection models. Even when extending GMN for cross-language clone detection, the models built leveraging data augmentation outperformed the baseline with scores of 0.90, 0.92, and 0.91 for precision, recall, and F1 score, respectively.

Index Terms—Code Clone Detection, Cross-Language Clones, Data Augmentation, Deep Learning, Graph Matching Networks

I. Introduction

Code clones are code fragments with similar functionalities in a software system. It appears when the developers reuse the existing source code knowledge base to annex the new features across the same or different platforms. Studies show that a software system may contain around 9% to 17% code clones [1], [2]. Such clones have adverse impacts on software systems as they often introduce redundant codes, require code changes to implement consistently, and make program comprehension harder for the developers [3], [4]. Moreover, reusing complex codes may generate intricate and sometimes incorrect programming logic. A rich body of research thus focuses on developing tools and techniques to detect and track clones across software systems so that they can be maintained through the complete software development life cycle [5], [6].

In this paper we focus on clones that appear across software systems written in different programming languages. Clone detection techniques in such a cross-language context are relatively less explored compared to the decades of research on single-language clone detection [7]. Cross-language clones are ubiquitous in software systems that are written in one language with certain functionalities and need to be translated into another language to add support for multiple platforms. Clones carry important domain knowledge and thus studying the clones in a system could potentially assist in understanding the system itself [8]. Cross-language clones can play a crucial role in program comprehension due to their potential for revealing code reuse patterns across different languages. Analysis of code reuse patterns in cross-language context could help researchers understand development practices, identify toxic code snippets, and build code searching tools [9] over diverse varieties of systems written in different languages. Furthermore, understanding the clones of a software system written in a certain programming language could potentially help understand a different software system written in a different programming language by tracking cross-language clones. Since multi-language software development (MLSD) appears to be common, at least in the open source world [10], cross-language clone detection could help understand any such clones and manage them accordingly in MLSD. Porting the same software systems to different platforms is natural and they could be written in different programming languages depending on the need. In such cases, cross-language clone detection tools will help detect, understand, and manage these clones across platforms.

Developers with existing knowledge and experience in one language make use of it to produce the same functionalities consciously or subconsciously. These codes are generally syntactically different but semantically the same [11]. Transforming systems written in one programming language to

another programming language is a tedious task. These can be considered as systems that are clones of each other whereas maintaining them over the years may take a lot of resources depending on their complexities.

Detecting cross-language clone is difficult due to the difference in syntax and textual nature of source codes [7]. Figure 1 shows an example of cross-language clones where three programs are written in two languages: the first one is in Java and the other two are in Python. All three programs solve the same problem of checking whether a given string is a palindrome. This example depicts the use of different structures and concepts such as library function, recursion, list slicing, etc. Programs that are written in different languages differ inherently due to the grammar behind them. As a result, token-based, text-based traditional approaches tend not to work well for cross-language clone detection [12]. Recently several deep learning models have shown good performances when detecting code clones in single-language settings [13], [14], [15]. These models appear to have good capabilities of leveraging the underlying structure and semantics of the code fragments to identify code clones. Among many techniques proposed for single-language clone detection, some are based on graph neural networks that consider both syntactic and semantic features of the code fragments. Consequently, such graph neural networks [14] arguably learn the representation better than other deep learning methods that consider only syntactical features [16].

A few techniques have been proposed for cross-language clone detection recently and deep learning models are shown to be effective in detecting clones [7], [17], [18]. One of the challenges in building deep learning models for cross-language clone detection techniques is the scarcity of data [17] as they typically require a large amount of data to well train the models [19]. In general, the performance of deep learning models depends on the quality and amount of data used to train them. A common technique to boost the performance of deep learning models is to use data augmentation that adds new data to the dataset by slightly modifying the existing data. Such data augmentation increases the number of data points and is shown to enhance the generalizability of deep learning models [20]. Although data augmentation techniques are well explored in computer vision and natural language processing research, only a few techniques are available that examine specifically the context of source code augmentation. These techniques predominantly focus on codes written in a single language and facilitate rule-based data augmentation [19]. While considering the problem of cross-language clone detection, we noticed that a rich body of literature on transcompilers has gone unnoticed [21]. A natural question that we thought of is whether they could be leveraged to extend existing single-language clone detection tools for cross-language settings and moreover, for the purpose of data augmentation. Are there reasons to use transcompiler based augmentations instead of augmentations based on simple mutations (e.g., line deletion, swap, or duplication)? Considering the constraints of cross-language clone detection, lack of sufficient data, and op-

```java
public static boolean checkPalindrome(String st){
    String r = "";
    boolean a = false;
    int i = st.length() - 1;
    while(i >= 0){
        r = r + st.charAt(i);
        i = i - 1;
    }
    if(st.equals(r)){
        a = true;
    }
    return a;
}
```

(a) Java code to check palindrome using library function

```python
def check_palindrome(s):
  def check_palindrome_helper(i, j):
    if j <= i:
      return True
    if s[i] != s[j]:
      return False
    return check_palindrome_helper(i + 1, j - 1)

  return check_palindrome_helper(0, len(s) - 1)
```

(b) Python code to check palindrome using recursion

```python
def check_palindrome(s):
    return s == s[::-1]
```

(c) Python code to check palindrome using list slicing

Fig. 1. Example of cross-language-clones: (a) is in Java, (b)–(c) are in Python.

portunities for leveraging transcompilers, we formulated the following research questions for this study.

RQ1. *To what extent does source-to-source translation based data augmentation influence cross-language clone detection models?*
In this research question, we explored the ability and extent of generalization for existing deep learning models through the lens of transcompiler-based data augmentation.

RQ2. *How does mutation based data augmentation perform compared to the source-to-source translation-based approach?*
To answer this research question, we studied the effect of feeding a deep-learning model with codes generated through random modification. We compared models trained with this approach with the ones trained with the transcompiler-based augmented data.

RQ3. *Can we use source-to-source translation to adapt single-language clone detection models to detect cross-language clones?*
To answer this research question, we investigated the opportunity to leverage existing single-language models for cross-language clone detection in a combination with transcompilers. We chose the state-of-the-art model [14] from the literature and extended it to cross-language settings.

Our contribution. In this paper we answer the research questions RQ1–RQ3, which results into the following contributions:

We introduce a data augmentation technique for cross-language clone detection using a transcompiler, which is a pre-trained deep learning model for source-to-source translation. We conduct controlled experiments on the widely used CLCDSA dataset [17] with state-of-the-art deep learning models for cross-language clone detection. Our experimental results show that the transcompiler-based data augmentation can boost the performances of these clone detection models when trained with the augmented dataset (e.g., we noticed 3% increase in F1 scores for some models).

Since the transcompiler we exploit uses a semantic-preserving translation, we examined whether the same level of performance boost could be achieved by augmented datasets that are created using simple mutation operations. By examining the abstract syntax trees (ASTs) of the augmented data we show that the ASTs for transcompiler augmented data are more diverse which provides some insights and justification for its use in the cross-language clone detection setting.

To examine whether transcompilers can aid in extending the single-language clone detection models for cross-laguage clone detection, we selected the Graph Matching Network (GMN), which is a widely used single-language clone detection technique and is known to show high performance on benchmark datasets [14]. Given a pair of code fragments written in different languages, the idea here is to first use transcompilers to transform one of these code fragments to match the language of the other code fragment, and then to use a single-language clone detection tool on the new pair. However, the transformed code obtained from transcompilers may not always be parsable, and hence, it cannot be directly fed as an input to GMN. We tackle this challenge by exploiting the srcML parser [22], which allows us to build XML representations to be used for GMN. Our experimental results show that the performances of such extended GMN models may not be the highest, but yet comparable to the cutting-edge cross-language clone detection models [7] that require high-end computing resources. This makes GMN an attractive option in a low-resource environment. Furthermore, this opens up the opportunity to explore whether the proposed framework for extending existing single-language clone detection models could be improved further or leveraged to build better cross-language clone detection models.

II. Background

A. Code Clone

Codes that share syntactic and semantic similarities are clones of each other. Code fragments that are modified or transformed through editing have the same functionalities and are termed syntactic clones. Semantic clones are code fragments that have major differences in their structure and have the same meaning or semantics [23], [24]. Clones can be broadly divided into four types [23], [25], [26]. **Type-I**: Identical code fragments with a varying number of comments and white spaces. **Type-II** : Code fragments that are equivalent in syntax with changes in identifier names, literals, types,

layout, and comments. **Type-III**: Along with Type-I and Type-II, this type has addition, removal, and/or modification of statements. **Type-IV**: Code fragments that have the same functional behavior but very different syntax.

Software systems that are maintained across different platforms are often developed in different languages. Therefore, these systems often have code fragments written in different languages but with the same functionalities. Such code fragments are known as cross-language clones [17]. These clones can be categorized as Type-IV clones.

B. Source-to-source translation

Transforming codes from one programming language to another language is defined as source-to-source translation. It is often referred to as transcompilation [27]. There are a few open source packages such as java2python[1], chsarp2python[2], cs2j[3], etc for source-to-source translation. These packages work only for the intended target language and are often unable to transform code the other way around. A few commercial tools are also available for high-level source-to-source translation[4]. These tools and techniques rely on hand-crafted rules that use the mapping of keywords and libraries from one language to another. There exist two well-known pre-trained deep learning models that can be used for code conversion. One is available through the Microsoft CodeXGLUE[5] project that can convert between C# and Java code, and the other one is Transcoder, which is available through Facebook research[6] that can convert among Java, Python, and C++.

C. Pre-trained Models

Pre-trained deep learning models have been successful in different natural language processing tasks. Among the available models, BERT [28] and GPT [29] achieved state-of-the-art results in different downstream tasks. With the inspiring results from these models, software researchers adopted them to build models such as CodeBERT [30] and CodeGPT [31]. These models have been in use for different tasks such as code completion, code search, code summarization, and so on [31].

Transcoder is a pre-trained neural machine translation based model for code conversion among programming languages released by Facebook [21]. It uses the sequence-to-sequence modeling approach [32] with unsupervised learning and exploits the transformer architecture with attention mechanism [33]. Transcoder achieved state-of-art accuracy in source-to-source code translation when compared with the existing commercial and non-commercial tools.

D. Data Augmentation

Deep learning models heavily rely on data to learn complex patterns from given data. With limited data, complex models

---

[1] https://github.com/natural/java2python
[2] https://github.Scom/shannoncruey/csharp-to-python
[3] https://github.com/twiglet/cs2j
[4] https://www.tangiblesoftwaresolutions.com/converters.html
[5] https://github.com/microsoft/CodeXGLUE
[6] https://github.com/facebookresearch/CodeGen

are known to suffer from issues such as over-fitting and the inability to generalize to other datasets. Providing more data to the model is one of the fundamental ways to overcome such challenges [20]. Creating synthetic data to increase dataset size by manipulating original data is known as data augmentation [19], which is widely used in computer vision and natural language processing domains.

E. Mutation Analysis in Code Clones

Mutation is the process of modifying a piece of code to get another slightly modified version of it. Mutation analysis is primarily used in software testing [34]. The primary goal is to introduce bugs in codes for testing. In code cloning, these operators are used to modify a code fragment to create another copy [35]. Mutation analysis consists of a set of operations. These operations include renaming identifiers, insertion or deletion of a statement, inter-changing loops from one to another, swapping statements, etc. The mutation operations are language independent unless replacing one type of control with another, which is language-specific. Any of these operations can be used in any sequence to generate any number of copies of a code fragment. Mutation analysis has been used to evaluate clone detection tools [36] and generate multiple copies of a code fragment [37].

F. Graph Matching Network (GMN)

Graph matching network is a graph neural network (GNN) that takes two homogeneous graph structures and finds the similarity between them [38]. The goal of GNN model is to achieve embedding for the nodes of the graph by learning about the surrounding structure and semantics. Code fragments inherently have a structure that can be represented as trees or graphs. In representation learning, GNN learns from its surrounding neighbors and finds an embedding for each node. The GMN model has proved to be very effective for clone detection tasks in single-language settings [14] with a proper embedding [16]. It takes the advantage of cross-graph attention to ensure that similar structure remains close in the embedding space and dissimilar structure spreads away while finding the global embedding.

III. RELATED WORK

1) Single-Language Clone Detection: Researchers have mostly been focused on single-language clone detection [39]. Traditional techniques primarily use structural features to detect single-language clones. Among them, a text-based approach such as NiCad [40] uses code normalization and text comparison to detect near-miss clones. Token-based approach SourcererCC [41] exploits tokens of code blocks to create an inverted index and compare them to find clones. However, it only captures information at the lexical level, which limits its performance. Deckerd [42] uses AST to create clusters of numeral vector representations of subtrees. These subtrees are created from the AST of code fragments, and the cluster is created using the Euclidean distance metric. It is dependent on pre-defined language-specific rules. These techniques have

been successful in detecting the first three types of clones. However, these conventional methods could not achieve the same performance for Type-IV clones [35] due to the inability to capture the code semantics among code fragments with different syntax.

The use of deep learning techniques has gained popularity because of its ability to learn the representation of the semantics of code fragments. White et al. [43] applied deep learning to reduce the gap between the syntactic and semantics of code fragments by using both lexical information of identifiers and structural information of AST. DeepSim [44] encodes Control Flow Graphs (CFG) to generate semantic metrics for deep neural networks. However, CFGs lack control and data flow information at different granular levels. Wang et al. [14] addressed this issue and proposed a modified AST structure by adding additional control flow edges to the AST. The authors combined the modified AST with graph neural network to successfully detect clones. Another work by Phan et al. [45] used a graph-based convolution network with CFGs, which learns semantic features to find software defects.

Ji et al. [46] adapted a graph convolution network with hierarchical active graphs with an attention mechanism to distinguish the importance among nodes in the AST. Zhang et al. [13] proposed an AST-based Neural Network (ASTNN), which splits ASTs into different sub-tree segments and uses bidirectional RNN. This method achieved success in both code classification and clone detection tasks. The recent success of encoder-decoder models in natural language processing tasks has paved the way for them in source code analysis applications. For instance, codeBERT [30] and graphCodeBERT [47] are built on top of Google's BERT model and have been used for multiple code tasks including clone detection.

A. Cross-Language Clone Detection

Only a few methods are known so far for cross-language clone detection due to the complexity and unavailability of proper datasets. Cheng et al. [48] proposed CLCMiner, which uses code revision histories to detect clones. Their method did not use any intermediate representation and only relied on data from the version control system. LICCA [49]is another well-known tool that uses tree-based intermediate representation and variant of the longest common subsequence algorithm. Nafi et al. [17] used hand-crafted features to train a siamese neural network for cross-language clone detection. The authors provided a cross-language clone dataset in their paper [17].

Mathew et al. [11] used dynamic analysis to detect cross-language clones. In another study, they studied clone detection as a special case of code search [50]. This study combines a generic AST and token-based approach with non-dominating sorting. Tao et al. [7] proposed C4 that leverages contrastive learning and exploits the pre-trained codeBERT model. They experimented with the CLCDSA datasets and achieved state-of-the-art performance in cross-language clone detection.

B. Source Code Augmentation

With the advent of the deep learning era and the availability of many big and/or benchmark datasets, deep learning models have gained a lot of interest from the research community. Cross-language clone detection is one of the few areas that lack a proper dataset [7]. Existing techniques rely on small datasets and are often hand-crafted by the authors [51].

There exists a number of studies that applied data augmentation in the context of code clones. Yu et al. [19] built a rule-based tool named SPAT for Java which has 18 transformation rules that can create semantically equivalent codes. The authors created these transformation rules by observing code patterns from clone pairs in the widely used BigCloneBench [52] and OJ datasets [53]. The significant similarity in clone pairs enabled the authors to extract patterns from them.

Transforming codes to generate adversarial examples is often used to test a model's robustness. Zhang et al. [54] used renaming techniques to attack code processing models. Models trained with these adversarial examples showed improvement in classification tasks and robustness of the model. Deepbugs [55] introduced a bug-inducing pattern in the code and proposed a name-based learning approach that detects bugs. It was very specific and limited to bug detections. Compton et al. [56] trained a model with obfuscated codes. This obfuscation of variable renaming showed a decline in the performance of the model in the method-name prediction task though the embedding showed better preservation of semantics.

All existing research on source code augmentation used rule-based transformations that require manual investigation and human intervention. Such manual procedure often introduces systematic bias [57], which jeopardizes the effectiveness of the process. Additionally, none of these transformation techniques focus on augmenting the dataset and are not built on top of any deep learning models. Our research is different from the studies mentioned in several aspects such as using the existing knowledge of transcompilers to augment data and extending single-language clone detection models for cross-language clone detection, which are both natural concepts that have not yet been explored in the literature.

IV. RESEARCH METHODOLOGY

A. Problem Formulation

We deal with two dimensions of cross-language clone detection: (a) the clone detection task and (b) improving the deep learning based cross-language clone detection models through data augmentation.

Clone detection is formulated as a binary classification problem. Given two code fragments F_i and F_j, we define a clone pair (F_i, F_j) and associate a label L_{ij}. A clone pair label is true when there are significant similarities between the fragments; otherwise, the label is false. A similarity score, S_{ij} is calculated on the pair (F_i, F_j), and the pair is true clone when S_{ij} is larger than a threshold value [58]. This similarity is based on the syntax or semantics of the code fragments. A pair of code fragments can only be called clones when we obtain a similarity score above the threshold.

Fig. 2. Overview of our approach to data augmentation and model training.

Improving a model's performance via data augmentation is formulated from the dataset perspective. Assume that D is a given dataset D, and a model M has an F1 score of s on D. If we can augment the dataset D by an amount X, train the model M on $D + X$, and get an F1 score of $s + t$, then the difference in F1 score is t, which we refer to as the improvement obtained by the augmentation. Note that in both cases the test dataset is the same, which is kept separate and thus not augmented.

Figure 2 illustrates the overall approach of our study while examining the impact of data augmentation on cross-language clone detection models. First, we convert each code fragment from the original dataset to get another version of it in another language (e.g., Java to Python and vice versa). Second, the original code fragments and the transcompiled fragments are stored in a database to be used for the assessment of the impact of augmentation. Third, the models are trained in two different ways, one that only uses the original dataset and the other using the augmented dataset. Here we compute the model accuracies and compare them to compute the impact of data augmentation.

B. Data Augmentation using Transcoder

In our study, we are particularly interested in generating data for cross-language clone detection models. We used source-to-source translation for this task. To create more data, we leveraged the pre-trained model, Transcoder, to convert codes from one language to another. In our dataset, we had Java and Python. Consequently, we converted all Java codes to Python and Python codes to Java. Since the Transcoder is a pre-trained deep learning model, it is very effective and efficient in generating fragments that share the same semantics as the original ones. Moreover, the pattern learned by the pre-trained models is not based on hand-crafted rules. As a result, the kind of codes it generates are diverse in nature. The details about semantic, syntactic, and computational correctness of the generated codes can be found in [21].

C. Mutation Based Augmentation

We created another dataset by transforming code fragments using mutation operations to see how the extent of its impact compares with the ones obtained using Transcoder based augmentation. The operation includes insertion, deletion, swap,

Fig. 3. Graph matching network combined with a transcompiler.

TABLE I
SUMMARIES OF ORIGINAL DATASET

Metric	AtCoder(AtC)		GoogleCodeJam(GCJ)	
	Java	Python	Java	Python
#Problems	1095	1028	261	223
#Average Lines	55	19	73	57
#Parsable Fragments	14838	14703	4341	1121
#Unparsable Fragments	620	11	5	2471

duplication of statements, comments, and change in operators [35], [36]. We chose these transformations based on how different types of clones are defined and categorized. For example, comments and white spaces are the only differences for Type I clones. Similarly, insertion, deletion, and modification of statements fall into Type III clones. We chose the operations with random order and frequency. These changes were made at random locations in the code fragment [36].

D. GMN with Transcompiler

To demonstrate how transcompiler could be used to extend single-language clone detection tools for cross-language settings, we leverage graph matching network (GMN). GMN is a single-language clone detection model that achieved superior single-language clone-detection performances for all four types of clones [14]. Furthermore, it exploits both the structure and semantics of code fragments which makes it more robust for source code analysis.

Figure 3 depicts the steps of using GMN with Transcoder. GMN can only match homogeneous structures, and the embedding space is based on the textual information of tokens for each node. As a result, different structures and texts would generate very different embedding in GMN. We chose Java as the target language for this reason. First, we pass the Python code fragment from a pair through Transcoder to get the converted code representation in Java. Then both of the fragments are passed through srcML to create the AST representation. We use this representation to train the model. We can test it against any fragment following the same conversion steps once the model is trained.

E. Models and Datasets

The cross-language clone detection models that we used in our experiments are CLCDSA and C4, and the single-language model that we chose to extend to the cross-language

TABLE II
SUMMARIES OF TRANSCOMPILED DATASET

Metric	AtCoder(AtC)		GoogleCodeJam(GCJ)	
	Java	Python	Java	Python
#Problems	1027	1086	221	258
#Average Lines	14	32	25	45
#Parsable Fragments	10024	5574	1989	1550
#Unparsable Fragments	4684	2620	1578	2620

setting is GMN. Figure 4 shows the structures of these models at a high level. We used the popular dataset from the paper that introduced the cross-language clone detection model called CLCDSA [17]. We will refer to this dataset as CLCDSA dataset for convenience. The dataset consists of code fragments from three programming contests that include AtCoder[7], Google CodeJam[8], and CoderByte[9]. AtCoder is a programming contest website that originated in Japan, and CodeJam is Google's programming competition. The data for CoderByte is not available in their replication package.

We chose code fragments of Java and Python programming languages for this study. The study is challenging because these two languages follow different paradigms, such as Java is a statically-typed language, and Python is a dynamically-typed language. There are about thirty-eight thousand Java and Python code fragments in the original dataset. We used Transcoder to convert code fragments from the original dataset to get another transcompiled version of it. We create the augmented dataset by combining this data with the original dataset. The summary statistics for original data are given in Table I, and the information regarding transcompiled data is shown in Table II. Both of the datasets have many unparsable code fragments.

We followed the pair creation procedure of CLCDSA [17]. Code fragments from the same problem are identified as clones, and code fragments from different problems are non-clones. We had 312,581 clone and non-clone pairs in the original dataset and 1,130,998 in the augmented dataset.

We created a smaller version of the dataset by filtering out fragments of less than six lines which is often considered as the minimum granularity for functional clone [40]. Then we randomly selected 60% of these code fragments to create clone pairs, which we will refer to as the 'sample dataset'. This sample dataset is around one-third of the original dataset and consists of 134,623 pairs. All code fragments from the sample dataset were converted using Transcoder. These converted fragments were combined with sample dataset to create the augmented sample dataset. The augmented sample dataset consists a total of 825,050 pairs.

To augment the dataset, clone/non-clone pairs were created after combining the original code fragments with the new TransCoder generated code fragments. We selected 20% random pairs from these new pairs and combined them with the pairs from the original dataset. As a result, this augmentation

---

[7]https://atcoder.jp/

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

[9]https://coderbyte.com/

Fig. 4. Deep learning models in our study. CLCDSA and C4 are existing models for cross-language clone detection. GMN is combined using a transcompiler.

process increased the datasets by 20% for both the original dataset and sample dataset.

We split the data into 8:1:1 ratio for train, validation, and test set [7], [14], [17]. Since binary classification can be biased with an imbalanced number of items for each class, we maintained a 1:1 ratio for clone and non-clone pairs across all models. We followed the standard procedure to select an equal number of clone and non-clone pairs randomly [7]. The models in this study have also used the same ratio and have been trained through this procedure to ensure data balance and reduce bias [7], [14], [17]. We followed the same procedure while creating the other augmented dataset with the mutation-based code transformations.

F. Evaluation Metrics

Precision, recall, and F1 score are the most widely used [7], [14] metrics for clone detection tasks, which are defined as follows.

$$Precision = \frac{T_P}{T_P + F_P} \tag{1}$$

$$Recall = \frac{T_P}{T_P + F_N} \tag{2}$$

$$F1score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

Here, T_P is the number of clones classified correctly, and T_N is the non-clones classified correctly. F_P stands for non-clones that were mistaken as clones, and F_N is clones that were classified as non-clones by the model. Since Transcompiler merely provides us with a code representation, the values

of these metrics depend on the clone detection models. We use F1 score as the ultimate measure for model evaluation.

G. Experimental Settings

We used the Transcoder model[10] specifically trained with deobfuscation objectives for Java and Python. We used the default settings for the Transcoder as stated in the original paper [21]. We followed the settings in CLCDSA [17] for filtering out clones and non-clones to prepare the dataset. We followed the settings mentioned in the respective papers to train the models [17], [7], [14].

In the case of C4, we choose a batch size of 4. We replicated the model with the original dataset provided in the replication package and this setting and found less than 1% disagreement between the results. We trained and tested the models following existing literature [7], [14] on a machine using RTX-3080ti. Our codes and datasets are available here.

V. RESULTS

A. Answering RQ1: Impact of data augmentation through source-to-source translation

To answer the first research question, we divided the clone detection techniques into two groups. The first three techniques in Table III are the deep learning techniques (DL models) used in this study. The last row shows the non-machine learning (non-ML) approaches. Among the deep learning techniques, CLCDSA has been considered as a reasonable standard baseline, and C4 is the state-of-the-art cross-language clone detection model [50], [7]. The GMN model is the

[10]https://github.com/facebookresearch/CodeGen

	Model	Precision	Recall	F1
	CLCDSA	0.49	**0.99**	0.66
DL Models	C4	**0.95**	0.96	**0.96**
	GMN	0.90	0.92	0.91
Non-ML	CLCMiner	0.36	0.57	0.44
Models	COSAL	0.55	0.89	0.68

Model	Precision	Recall	F1	Δ
CLCDSA	0.53	0.98	0.69	**3%**
C4	**0.97**	**0.99**	**0.98**	2%
GMN	0.93	0.95	0.94	**3%**

Model	Precision	Recall	F1
CLCDSA	0.50	**0.99**	0.66
C4	**0.92**	0.95	**0.93**
GMN	0.89	0.91	0.90

Model	Precision	Recall	F1	Δ
CLCDSA	0.54	0.93	0.68	**2%**
C4	**0.93**	**0.96**	**0.94**	1%
GMN	0.91	0.93	0.92	**2%**

augmentation. To answer this research question, we examined the performances [36] of the models on the mutation based augmented dataset. We followed the methodology described earlier (Section IV-E) to create the mutation based augmented dataset, and re-trained all of the models. We also created a smaller sample of this dataset and maintained the same number of pairs and the ratio between clone and non-clone pairs as we did for RQ1.

Dataset Size	Model	Precision	Recall	F1	Δ
	CLCDSA	0.52	0.97	0.68	-1%
Original	C4	**0.97**	**0.98**	**0.97**	-1%
	GMN	0.89	0.90	0.91	**-3%**
Small	CLCDSA	0.50	**0.98**	0.67	-1%
sample	C4	**0.930**	**0.944**	**0.936**	-0.4%
	GMN	0.88	0.90	0.89	**-3%**

The rightmost column shows the change in the F1 score for the mutated dataset compared to our transcompiler based approach. It is evident from Table that for all models, performance decreased from 1% to 3% in the case of a larger dataset, and it dropped 3% for GMN for the smaller sample dataset. C4 managed to perform almost similar for both the original dataset and the smaller sample dataset. In the case of the original dataset, it shows only a decline of 1% while maintaining high accuracies. We believe that the power of C4 lies in the CodeBERT model, as it is primarily dependent on the token sequences instead of the structure or semantics of code fragments.

We further investigate the reason why mutation based augmentation fails to provide a performance boost whereas the transcompiler based approach appears to work.

Why does the Transcompiler Based Approach Work?

To this end, we believe the quality of data impacts a model's performance to a large extent. Although mutation based modification of source codes does not hamper the token quality, it may create unparsable code, and alternate the syntax and sequence of execution. Consequently, it affects the AST tree structure. Therefore, it may initially appear that

adapted single-language clone detection model for cross-language clone detection.

We trained each model with original and augmented datasets. We also trained with the smaller version of each dataset to understand the degree of impact on the models over dataset size. As a result, this combination makes four sets of datasets in total. Table III shows the precision, recall, and F1 score for each of the DL models we trained with the original dataset. Among the models, C4 has the best precision and F1 score, and CLCDSA has the highest recall score. Table IV shows the scores for these models when trained with the augmented dataset. All of the models show an improvement of 2% to 3% in terms of F1 score. CLCDSA and GMN showed a higher increase compared to C4 in the F1 score.

Table V shows that for the smaller sample dataset (as described in Section IV-E). Model C4 again has the highest F1 score and CLCDSA still maintains its high recall value. Table VI shows the result on the augmented sample dataset. In this case, CLCDSA and GMN show a 2% increase in the F1 score. The overall F1 scores for all models and the performance boosts appear to be small when compared to the results on the original dataset, which is expected as the performances of deep learning models often suffer from the lack of data. However, in all cases, the results show that the Transcoder-based data augmentation can effectively improve the clone detection performances of the deep learning models.

> In summary, our experimental results show that the transcompiler-based data augmentation can substantially increase deep learning models' performance for cross-language clone detection (e.g., for some models the F1 score improves about 3% on benchmark dataset).

B. Answering RQ2: Comparing mutation based augmentation with transcompiler based augmentation

We found that augmentation with source-to-source translation worked well for both large and small datasets. However, at this point, a natural question is whether we could find the same level of performance boost using simple mutation based data

TABLE VIII
AST DISTANCE STATISTICS.

Dataset Size		Original	Transcompiled	Mutated
Large	Mean	17.93	16.53	17.85
	SD	0.244	0.384	0.277
Small	Mean	17.94	16.55	17.86
	SD	0.249	0.407	0.281

TABLE IX
MEAN ABSOLUTE DIFFERENCE BETWEEN THE AVERAGE ROOT-TO-LEAF
DISTANCES FOR THE ASTS CORRESPONDING TO THE ORIGINAL AND
NEWLY CREATED CODE FRAGMENTS

	Original vs. Transcompiled	Original vs. Mutated
Mean	1.3	0.09
SD	0.37	0.17

the new code fragments that are created by the mutation based augmentation are more diverse than the ones created by the transcompiler based approach. To examine this we computed the ASTs of all the code fragments and computed a similarity score between ASTs obtained from the original code and augmented code. Specifically, we compared the metric average root-to-leaf distance for the ASTs (i.e., the mean of the distances from the root to all leaf nodes for each ASTs), as shown in Table VIII.

Contrary to the initial assumption, we now can see that the variation appears to be more in the transcompiler based augmented data compared to the mutation based augmented dataset. We investigated the reason for this and found the code generated from Python to Java by Transcoder is shorter compared to the mutated fragments (Java to Java). The AST lengths are quite similar to the original code in case of random perturbation unless many statements were added and deleted. Figure 5 shows examples of a code fragment that can read a text file. When it is transcompiled from Java to Python using Transcoder, the number of lines is reduced. As evident from the last fragment in the example, the random application of the mutation operation modifies the code by deleting and swapping some lines of code. Lines 6 and 7 from (a) were swapped and lines 3 and 9 were deleted by the operations. This observation shows that the data produced by transcompilers are more diverse in nature. These code fragments have inherent variation in their structure since the pre-trained model was trained on a large corpus of real-world data. This tends to capture more variation from the data and use that in inference. This explains the success of transcompiler based approach to some extent as removing redundancies and increasing the diversity in datasets tends to improve the performance of machine learning models [59].

In summary, the mutation-based data augmentation may not boost the model performances and thus careful data augmentation such as Transcoder based ones are important. A potential reason for the success of Transcoder based data augmentation is their capability of increasing data diversity while preserving the code semantics.

TABLE X
RESULTS ON ONLY PARSABLE CODES VS. ALL.

Parser	Metrics	Original Dataset	Augmented Dataset
Only Parsable	Precision	0.90	0.92
	Recall	0.92	0.94
	F1	0.91	0.93
All	Precision	0.90	0.93
	Recall	0.92	0.95
	F1	0.91	**0.94**

C. Answering RQ3: Extending single-language model for cross-language clone detection

Here we examined GMN in more detail. Recall that although GMN is known to provide high accuracy for single-language settings, it has not previously been used for cross-language clone detection. Our experiment shows GMN outperformed the baseline model CLCDSA by a high margin. It showed a very robust result, for all the cases examined in Tables III–IV. Since GMN relies on the code structure more than the other two models, it showed a larger decline in performance when used with the mutation based augmentation method. The original implementation of GMN did not consider unparsable codes. We exploited srcML[11] parser to create a graph for all types of codes generated by transcompiler irrespective of whether they are parsable or not.

Table X shows the results on original and augmented datasets. The top rows show the result obtained using only the code fragments that were parsable and the bottom rows show the result for all code fragments (i.e., including unparsable ones). We can see for augmentation, taking all types of code fragments helped GMN to learn better representations.

GMN takes significantly less time than C4. The reason is the number of parameters and fine-tuning time required by CodeBERT [30]. The number of trainable parameters for GMN is 123,101, whereas C4 has 172,503,552, which is 1401 times higher. To train the models, GMN takes around one-fourth time of C4. Hence we believe that GMN can be an excellent choice in a resource-constraint environment and it now appears as a competitive candidate to be examined further for cross-language clone detection in future research.

In summary, our results show that single-language clone detection models may be used in a combination with transcompilers to create a pipeline for effective cross-language clone detection. However, such an adaptation may require additional steps depending on the specifications of the single-language clone detection models being extended.

VI. DISCUSSION

A. Generalizability of Our Approach

The quality of the data augmentation using source-to-source translation depends on the choice of transcompiler. This study considered only Java and Python programming languages as a use case. These two languages are way different from one

[11]https://www.srcml.org/about.html

```
1  public static readTextFile( ) {
2      try {
3          File f = new File("textfilename.txt");
4          Scanner sc = new Scanner(f);
5          while(sc.hasNextLine( )) {
6              String st = sc.nextLine( );
7              System.out.println(st);
8          }
9          sc.close( );
10     } catch(FileNotFoundException e) {
11         e.printStackTrace( );
12     }
13 }
```

(a) Java code for reading a text file

```
1  def read_textfile( ) :
2      with open('textfilename.txt', 'r') as f :
3          with f :
4              for st in f :
5                  print(st)
```

(b) Python code by Transcoder

```
1  public static readTextFile( ) {
2      try {
3          Scanner sc = new Scanner( );
4          while(sc.hasNextLine( )) {
5              System.out.println(st);
6              String st = sc.nextLine( );
7          }
8      } catch(FileNotFoundException e) {
9          e.printStackTrace( );
10     }
11 }
```

(c) Java code by mutation

Fig. 5. Example of codes modified differently. The code in (b) is transcompiled through Transcoder from (a), and code in (c) is generated by applying random mutation operations.

another compared to the languages such as C# and C++ which have more lexical features in common. The performance of the Transcoder depends on the agreement of the dynamic or static nature of the taken languages. The more common characteristics two different language shares, the better outcome is possible from the Transcoder. However, TransCoder is a pre-trained model and it can convert a code fragment with a fraction of a second and therefore, can be used for large-scale clone detection tasks.

We showed that our approach is efficient as the pre-trained model does not require any kind of fine-tuning and can be used for inference directly for code conversion. Moreover, the amount of time required for model inference is negligible. These characteristics make this approach ideal and generalized for cross-language data augmentations. Our experimental results show consistent improvements for all models even when they already achieved over 90% F1 score. This indicates that practitioners may want to use such data augmentation techniques to their fullest potential. Additionally, the experimental result also shows success in the adaption of a single-language clone detection model through the aid of a transcompiler. Although we chose a specific graph neural network for this task, the concept generalizes to other deep learning based single-language clone detection model.

B. Threats to validity

Our method relies on a pre-trained Transcoder model. We used it to convert code fragments from one language to another language. This may introduce the inherent bias in our study. Transcoder is trained on a large corpus of codes. It outputs varied levels of computationally correct codes depending on the choice of programming language. In our technique, the languages that could be supported are limited by the ones supported by the transcompilers. Nevertheless, the rapid development of pre-trained models and different fine-tuning approaches for transcompilers may overcome these limitations.

We used the dataset provided by CLCDSA paper [17], which is collected from programming competitions. As a consequence, they can be very different from real-world software systems. Moreover, many of these problems are complex, and the solutions require a well-thought approach. Therefore, the code fragments are likely to be diverse both syntactically and semantically. Hence it would be interesting to explore our proposed method in real-life software systems.

We considered only two languages for our study whereas the original dataset has code fragments from more languages. We observed that other studies considered different sets of programming languages from this dataset. For example, CLCDSA [17] used three languages in their study, C4 used four languages [7], and COSAL [50] used only Java and Python. Examining how the performances vary across different languages could be an interesting avenue for future research.

VII. CONCLUSION

Clone detection is a widely studied field in software engineering research. However, techniques for cross-language clone detection are not extensively explored compared to their single-language counterpart. Here we proposed a novel data augmentation technique that uses a transcompiler (a pre-trained deep learning model for source-to-source translation) for cross-language clone detection. Our experiment shows that such data augmentation improves the performances of cutting-edge cross-language clone detection models. We also exploited a single-language model for detecting cross-language clones through source-to-source translation. The performance of the single-language model surpassed the current baseline by a large margin. This opens up an opportunity to further examine single-language clone detection models to understand how the transcompilers could be combined more effectively to achieve even better cross-language clone detection performances. In the future, we envision applying explainable AI techniques to further explain the power of data augmentation and to develop techniques to select the augmented data appropriately to further enhance the effectiveness of our approach.

VIII. ACKNOWLEDGEMENT

REFERENCES

[1] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy, "Analyzing and forecasting near-miss clones in evolving software: An empirical study," in *2011 16th IEEE international conference on engineering of complex computer systems*, pp. 295–304, IEEE, 2011.

[2] I. D. Baxter and C. W. Pidgeon, "Software change through design maintenance," in *1997 Proceedings International Conference on Software Maintenance*, pp. 250–259, IEEE, 1997.

[3] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, 1998.

[4] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proceedings Eighth IEEE Symposium on Software Metrics*, pp. 87–94, 2002.

[5] A. Hanjalić, "Clonevol: Visualizing software evolution with code clones," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 1–4, IEEE, 2013.

[6] D. Mondal, M. Mondal, C. K. Roy, K. A. Schneider, Y. Li, and S. Wang, "Clone-world: A visual analytic system for large scale software clones," *Visual Informatics*, vol. 3, no. 1, pp. 18–26, 2019.

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

[8] J. H. Johnson, "Visualizing textual redundancy in legacy source.," in *CASCON*, vol. 94, pp. 9–18, Citeseer, 1994.

[9] T. Diamantopoulos and A. Symeonidis, "Employing source code information to improve question-answering in stack overflow," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 454–457, IEEE, 2015.

[10] P. Mayer, M. Kirsch, and M. A. Le, "On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers," *Journal of Software Engineering Research and Development*, vol. 5, pp. 1–33, 2017.

[11] G. Mathew, C. Parnin, and K. T. Stolee, "SLACC: Simion-based language agnostic code clones," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 210–221, 2020.

[12] L. Nichols, M. Emre, and B. Hardekopf, "Structural and nominal cross-language clone detection," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 247–263, Springer, 2019.

[13] 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)*, pp. 783–794, IEEE, 2019.

[14] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 261–271, IEEE, 2020.

[15] S. B. Ankali and L. Parthiban, "Detection and classification of cross-language code clone types by filtering the nodes of antlr-generated parse tree," *International Journal of Intelligent Systems and Applications*, vol. 13, no. 3, pp. 43–65, 2021.

[16] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[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)*, pp. 1026–1037, IEEE, 2019.

[18] D. Perez and S. Chiba, "Cross-language clone detection by learning over abstract syntax trees," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 518–528, IEEE, 2019.

[19] S. Yu, T. Wang, and J. Wang, "Data augmentation by program transformation," *Journal of Systems and Software*, vol. 190, p. 111304, 2022.

[20] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[21] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[22] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *2013 IEEE International Conference on Software Maintenance*, pp. 516–519, IEEE, 2013.

[23] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[24] F. Al-Omari, C. K. Roy, and T. Chen, "Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge," in *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pp. 57–63, IEEE, 2020.

[25] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[26] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 131–140, IEEE, 2015.

[27] W. S. El-Kassas, B. A. Abdullah, A. H. Yousef, and A. M. Wahba, "Enhanced code conversion approach for the integrated cross-platform mobile development (icpmd)," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1036–1053, 2016.

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, (Minneapolis, Minnesota), pp. 4171–4186, Association for Computational Linguistics, June 2019.

[29] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[30] 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), pp. 1536–1547, Association for Computational Linguistics, Nov. 2020.

[31] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. GONG, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. LIU, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[32] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[34] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Comparative assessment of testing and model checking using program mutation," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pp. 210–222, IEEE, 2007.

[35] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *2009 international conference on software testing, verification, and validation workshops*, pp. 157–166, IEEE, 2009.

[36] J. Svajlenko and C. K. Roy, "The mutation and injection framework: Evaluating clone detection tools with mutation analysis," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 1060–1087, 2019.

[37] Y. Fujiwara, N. Yoshida, E. Choi, and K. Inoue, "Code-to-code search based on deep neural network and code mutation," in *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pp. 1–7, IEEE, 2019.

[38] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*, pp. 3835–3845, PMLR, 2019.

[39] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim, "Deep learning application on code clone detection: A review of current knowledge," *Journal of Systems and Software*, vol. 184, p. 111141, 2022.

[40] 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*, pp. 172–181, IEEE, 2008.

[41] 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*, pp. 1157–1168, 2016.

[42] 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)*, pp. 96–105, IEEE, 2007.

[43] 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)*, pp. 87–98, IEEE, 2016.

[44] 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*, pp. 141–151, 2018.

[45] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 45–52, IEEE, 2017.

[46] X. Ji, L. Liu, and J. Zhu, "Code clone detection with hierarchical attentive graph embedding," *International Journal of Software Engineering and Knowledge Engineering*, vol. 31, no. 06, pp. 837–861, 2021.

[47] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[48] 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.

[49] 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)*, pp. 512–516, IEEE, 2018.

[50] G. Mathew and K. T. Stolee, "Cross-language code search using static and dynamic analyses," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 205–217, 2021.

[51] N. D. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

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

[53] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[54] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 1169–1176, 2020.

[55] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[56] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 243–253, 2020.

[57] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced? bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 121–130, 2009.

[58] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 476–480, IEEE, 2014.

[59] Z. Gong, P. Zhong, and W. Hu, "Diversity in machine learning," *IEEE Access*, vol. 7, pp. 64323–64350, 2019.