Fine-grained Code Clone Detection by Keywords-based Connection of Program Dependency Graph

Yueming Wu, Wenqi Suo, Siyue Feng, Cong Wu, Deqing Zou, and Hai Jin, *Fellow, IEEE*

***Abstract*—Code clone detection is intended to identify functionally similar code fragments, a matter of escalating significance in contemporary software engineering. Numerous methodologies have been proffered for the detection of code clones, among which graph-based approaches exhibit efficacy in addressing semantic code clones. However, they all only consider the feature extraction of a single sample and ignore the semantic connection between different samples, resulting in the detection effect being unsatisfactory. Simultaneously, the majority of existing methods can only ascertain the presence of clones, lacking the capability to provide nuanced insights into which lines of code exhibit greater similarity. In this paper, we advocate a novel PDG-based semantic clone detection method namely *Keybor* which can locate specific cloned lines of code by providing a fine-grained analysis of clone pairs. The highlight of the approach is to consider keywords as a bridge to connect PDG nodes of the target program to retain more semantic information about the functional code. To examine the effectiveness of *Keybor*, we assess it on a widely used *BigCloneBench* dataset. Experimental results indicate that *Keybor* is superior to 14 advanced code clone detection tools (*i.e., CCAligner*, *SourcererCC*, *Siamese*, *NIL*, *NiCad*, *LVMapper*, *CCFinder*, *CloneWorks*, *Oreo*, *Deckard*, *CCGraph*, *Code2Img*, *GPT-3.5-turbo*, and *GPT-4*).**

***Index Terms*—Code Clones, Program Dependency Graph, Keywords, Fine-grained.**

I. INTRODUCTION

CODE clone refers to the phenomenon of the duplication of entire code or code fragments. It is commonly classified into syntactic clone and semantic clone according to the degree of code duplication. Syntactic clones refer to code segments with textual similarity, categorized into three subtypes based on the level of similarity: textual similarity (Type-1), lexical similarity (Type-2), and syntactic similarity (Type-3). Semantic clones, also known as Type-4, pertain to code segments with similar functionality, employing different code syntaxes. The rapid evolution of software engineering increases exponentially the demand for code. The prevalence of code cloning in practical applications is on the rise due to its significant advantages in alleviating the time and effort burdens on developers. However, code cloning also introduces some disadvantages, such as increased maintenance costs due to the propagation of vulnerabilities. Therefore, the detection of code clones becomes very important.

Some techniques for code clone detection also exist currently. For example, methods based on tokens offer the highest scalability and efficiency, *Toma* [1] extracts token types and calculates six similarity scores to construct feature vectors. By utilizing the new electron mismatch index and the asymmetric similarity coefficient, *CCAligner* [2] can detect Type-1 and Type-2 clones, while also showing the potential to identify Type-3 clones. However, since they only extract tokens for textual information and lack the exploration of semantic information, they cannot detect semantic clones. Detecting semantic clones poses the greatest challenge. To detect Type-4 clones, graph-based detection methods [3]–[8] and tree-based methods [9]–[15] preserve the program details by extracting different intermediate representations of the program. However, these approaches either solely extract singular syntactic information or disregard the correlation between code structures, resulting in suboptimal performance when detecting complex semantic clones. Learning-based code clone detection methods [6], [10], [11], [16]–[18] have been developed to identify complex semantic clones. However, the study [19] shows that the studied advanced deep learning models have demonstrated poor generalization performance, and are typically time-consuming to train. These two drawbacks make learning-based clone detection techniques less suitable for real-world clone detection scenarios. Furthermore, these methodologies primarily emphasize confirming the existence of clone relationships between code pairs, yet they lack precision in pinpointing the specific lines of code that have been cloned. Hence, there is a pressing demand for a traditional code similarity-based clone detection approach capable of conducting a nuanced analysis of semantic clones at a fine-grained level.

In this paper, we develop a novel *program dependency graph* (PDG)-based semantic clone detection method, *Keybor*. To overcome the problem that traditional clone detection methods often fail to capture the semantic similarity between cloned codes and cannot accurately locate statements with high similarity, we explicitly solve two main challenges:

- *Challenge 1: How to retain more program details to detect more semantic clones with high precision?*
- *Challenge 2: How to achieve finer-grained clone detection that can report statements with high similarity?*

Y. Wu, W. Suo, S. Feng (corresponding author) and D. Zou are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China. E-mail: fengsiyue@hust.edu.cn.

Y. Wu and D. Zou are also with Jinyinhu Laboratory, Wuhan, China.

C. Wu is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore.

H. Jin is with 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.

To address the first challenge, we propose to bridge the PDG nodes of the target program using keywords and utilize the graph embedding algorithms *Struc2vec* [20] and *Sent2vec* [21] to extract structural features and code features respectively. Specifically, we first distill the program details of each method into a PDG, where each node represents a code line of the method. Subsequently, we connect two PDGs using keyword nodes as hubs, forming a new graph, namely *PairPDG*. By incorporating edges based on keyword connections, we can more accurately capture the semantic associations between the code segments. This enhancement results in a more pronounced manifestation of semantic similarity among clone pairs.

To overcome the second challenge, we perform line-level similarity matching. Since each node in the PDG represents a line of code in the source code. By calculating the similarity between nodes in two target source codes, we can identify pairs of nodes with high similarity. This approach enables the precise localization of similar statement pairs within the clone pairs based on line numbers. In detail, we use *Sent2vec* [21] to extract the semantic features of each line of code and achieve fine-grained clone detection by analyzing the textual similarity of each line of code in clone pairs.

We propose *Keybor* and subject it to evaluation using the extensively employed dataset *BigCloneBench* [22], [23]. *Keybor* enables fine-grained detection of complex semantic clones by connecting keywords to enrich semantic information. Our experiments reveal that *Keybor* significantly outperforms 12 state-of-the-art traditional code clone detection systems and two large language models. For example, the recall of *Keybor* when detecting moderately Type-3 is 83% while our 14 comparative tools (*i.e., CCAligner* [2], *SourcererCC* [24], *Siamese* [25], *NIL* [26], *NiCad* [27], *LVMapper* [28], *CCFinder* [29], *CloneWorks* [30], *Oreo* [18], *Deckard* [9], *CCGraph* [7], *Code2Img* [31], *GPT-3.5-turbo* [32], and *GPT-4* [33]) can only achieve 14%, 1%, 14%, 19%, 2%, 19%, 1%, 15%, 30%, 12%, 29%, 25%, 59%, and 77%, respectively. For fine-grained analysis, *Keybor* not only reports whether two methods are clones but also pinpoints the pairs of statements with high similarity to help researchers conduct subsequent in-depth analysis.

In general, the primary contributions made in this paper are as follows:

- We use keywords to connect PDG nodes of two methods and perform semantic feature extraction using a graph embedding algorithm to preserve more semantic information about the code.
- We design a PDG-based fine-grained semantic clone detection method, *Keybor*[1], which not only reports whether two methods are clones but also can precisely locate pairs of statements with high similarity.
- We conduct comparative evaluations with 14 systems on *BigCloneBench* [22], [23] dataset. Experimental findings confirm that *Keybor* has the optimal detection performance over *CCAligner* [2], *SourcererCC* [24], *Siamese* [25], *NIL* [26], *NiCad* [27], *LVMapper* [28], *CCFinder* [29], *CloneWorks* [30], *Oreo* [18], *Deckard* [9], *CCGraph* [7], *Code2Img* [31], *GPT-3.5-turbo* [32], and *GPT-4* [33].

[1]https://github.com/KeyborCode2024/keybor.

II. BACKGROUND AND MOTIVATION

Before presenting our proposed system, we will commence by establishing essential terminology that will be employed consistently throughout this manuscript.

A. Clone Type

Code cloning pertains to the presence of identical or similar source code fragments within a code base. Classifications of code clones are typically delineated based on varying levels of similarity, resulting in the general categorization into the following four types [34], [35]:

- **Type-1 (textual similarity)**: Code fragments that are identical except for differences in white spaces, layouts, and comments.
- **Type-2 (lexical similarity)**: Code fragments that are identical except for variations in identifier names and lexical values, along with the differences found in Type-1 clones.
- **Type-3 (syntactic similarity)**: Code fragments that are syntactically similar but differ at the statement level. In addition to the variations seen in Type-1 and Type-2 clones, these fragments may have statements that are added, modified, or removed.
- **Type-4 (semantically similarity)**: Code fragments that are syntactically dissimilar but perform the same functionality.

```
//Original
private int sum(int n){
    int m = 0;
    int i = 1;
    while(i <= n){
        m += i;
        i++;
    }
    return m;
}
```

```
//Type-1
private int sum(int n){
    int m = 0;
    int i = 1;
    while(i <= n){
        m += i;
        i++;
    }
    return m;
}
```

```
//Type-3
private long sum(long n){
    long m = 0;
    long i = 1;
    while(i <= n){
        m += i;
        i++;
        mymtem.out.println(m);
    }
    return m;
}
```

```
//Type-2
private int sum(int a){
    int b = 0;
    int t = 1;
    while(t <= a){
        b += t;
        t++;
    }
    return b;
}
```

```
//Type-4
private int sum(int n){
    int m = 0;
    for (int i = 1; i <= n; i++) {
        m += i;
    }
    return m;
}
```

Fig. 1. Examples of different clone types

In order to visually illustrate four types of code cloning, Figure 1 provides examples of clones at various degrees

based on the original code. The original method implements a cumulative summation function. Type-1 to Type-4 code clones undergo varying degrees of modification based on the original. Type-1 code clone alters the comments, except that the two code fragments are otherwise identical. Type-2 code clone makes no changes to the code snippet, except that the variable names of the original code have been replaced (*i.e.,* a, b, t instead of n, m, i). Type-3 code clone extends the original fragment by one line of output while maintaining its original syntax. Type-4 code clone substitutes the while loop with a for loop, accomplishing equivalent functionality through an entirely divergent syntax structure. This variant of code clone, termed a semantic code clone, poses heightened challenges in detection due to the potential significant alterations in the code structure. This classification of code clones reflects the varying degrees of similarity between code fragments, with code clones becoming progressively less similar and more difficult to detect from Type-1 to Type-4.

B. Program Dependency Graph

A *Program Dependence Graph* (PDG) is an intermediate representation used to illustrate the dependencies between various statements and expressions in a program. It unifies the representation of both control and data dependencies, helping developers understand the logical structure of a program. PDGs are commonly used in optimization, debugging, code analysis, and clone detection scenarios.

A PDG is a graphical structure where each node represents a statement or expression, and the edges represent the dependencies between these statements or expressions. These dependencies are categorized as follows:

- Data Dependence: This describes the relationship where one statement depends on the data generated by another statement. For example, if statement A assigns a value to variable x, and statement B uses that value, statement B is considered data-dependent on statement A.
- Control Dependence: This represents the control flow of the program, indicating that the execution of a statement depends on the result of a previous control structure's evaluation. For example, if the execution of statement B depends on the condition of statement A (such as an if or while statement), then statement B is control-dependent on statement A.

C. Graph Embedding Algorithm

A graph is a structure consisting of nodes and edges, utilized to represent entities and the relationships between them. A graph can be directed or undirected, and edges can be weighted or unweighted. In many real-world scenarios, such as social networks, knowledge graphs, and communication networks, data naturally exists in the form of graphs.

Graph embedding algorithms convert graph-structured data (*e.g.,* social network, molecular structure, knowledge graph) into low-dimensional vector representations. This vectorized representation makes it easier to perform machine learning and data mining tasks, such as node classification, link prediction, and community detection, while retaining the graph's structural information [36], [37]. The main goal of graph embedding algorithms is to encode the nodes or subgraphs of a graph as low-dimensional real-valued vectors, while preserving as much of the original structural information and attributes as possible. This transformation of complex graph structures into numerical feature vectors allows machine learning algorithms to process them more effectively. After embedding, tasks like similarity calculation, link prediction, and node classification become simpler and more efficient.

Graph embedding algorithms can be divided into the following categories:

- Matrix decomposition-based methods: Examples include Graph Laplacian Eigenmaps and Spectral Embedding.
- Random walk-based methods: Examples include DeepWalk and node2vec. These methods capture both local and global information in the graph through the use of random walks and embed this information into the vector space.
- Deep learning-based methods: Examples include *Graph Convolutional Networks* (GCN) and GraphSAGE. These methods use neural networks, particularly convolutional neural networks, to learn node representations in graphs.

D. Motivation

Keywords [38] are defined and reserved for usage within the programming language and are generally used to form the overall framework of a program, to express key values and complex semantics with structure, etc. As predefined words with a specific purpose, keywords can define the structure and flow of a program and specify the actions that the program should perform. For example, keywords related to the data type definitions (*e.g.,* int and char) determine the data types used in a program and thus affect the data flow. The keywords related to the program control (*e.g.,* do and while) can determine the direction of program execution, thereby influencing the control flow. The majority of syntactic information in the context of a program is made up of keywords and certain symbols, the meaning of which effectively symbolizes the usage of that syntactic format.

In addition, it is prohibited for programmers to define the same identifiers as a keyword while programming. Because of the restrictions on the naming of identifiers, keywords become fixed, unambiguous lexical elements in the code, which significantly facilitates recognition and understanding. Consequently, identifying a limited and fixed number of keywords in a program enables us to get a glimpse of the overall general flow, behavior, and structural framework of the code. In general, keywords provide a clear and concise way to express program logic, which somewhat reflects the construction of the program structure framework and the expression of semantics. Can connecting the two codes in a clone pair using keywords as pivotal elements result in a more accurate capture of the semantic associations between the codes, thus enhancing the conspicuousness of semantic similarity between the clone pairs? We investigate existing related work on PDG-based semantic clone detection, all of which deal with individual PDG and have not been connected intrinsically. Therefore, a preliminary study is conducted to investigate whether adding keyword connections in feature

extraction could more accurately depict the program's feature information.

To verify our proposed idea, 10,000 Type-4 clone pairs are randomly chosen from the dataset *BigCloneBench* (BCB) [22] to implement a controlled experiment that compared the effects of Keyword-based connection with Non-keyword connection. First, we extract the PDGs for each pair of clone methods to conduct a comparative experiment on keyword connectivity in graph form. In the Keyword-based connection experiment, we analyze the nodes in each PDG and connect them based on the code statements contained within the nodes and the relevant keywords. For example, if a statement contains the keyword "return 0;", then that node is connected to the "return" keyword node. If both PDGs contain the "return" keyword, a connection is established between the two PDGs via the "return" keyword. After completing the keyword connectivity, we employ the widely used node embedding method *Struc2vec* [20] to obtain the feature vectors corresponding to the target code and compute the similarity. In the Non-keyword connection experiments, we do not establish any connections between the two PDGs; instead, they were treated as two independent graphs for feature vector extraction and similarity calculation. Figure 2 shows the similarity between Non-keyword connection and Keyword-based connection experiments.

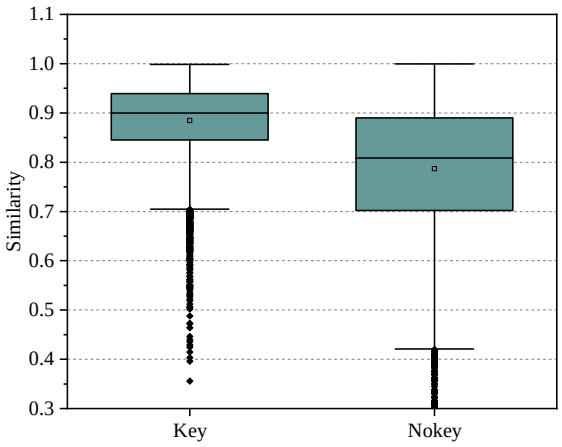

Fig. 2. The similarity between Keyword-based connection and Non-keyword-based connection experiments

The results present in Figure 2 indicate that the similarity calculated in the Keyword-based connection is significantly higher than that in the Non-keyword connection experiments, which indicates that bridging the two functions with keywords provides more information to facilitate the identification of similarities in the code. However, we discover that the average similarity of the clone pairs was high whether or not the keywords connection was introduced. This is because our experiment focuses solely on the structural similarity of the graphs and overlooks the similarity of the code. We plan to address this limitation in our subsequent experimental designs.

Therefore, based on the observations, we propose the method *Keybor*, which enriches the semantic information by connecting keywords, thus improving the accuracy of the detection of complex semantic clones.

III. SYSTEM

We propose a fine-grained clone detection method *Keybor* for complex semantic clones based on keyword connection. To enhance the preservation of semantic information in functional code, keywords are utilized as connectors between PDG nodes in the target program. After that, we employ graph embedding methods to capture semantic features and evaluate their similarity by *Cosine Similarity* of two vectors. This section provides a comprehensive explanation of the method's architecture and the implementation of each component.

Algorithm 1 Keybor workflow

Input: Two methods M_1 and M_2 to be detected as clones or not, and the keywords list $keywords\_list$.
Output: Detection result $result$.

// PDG Generation
$PDG_1 \leftarrow$ PDG_GENERATE (M_1)
$PDG_2 \leftarrow$ PDG_GENERATE (M_2)
// PDG Mergence
for each $keyword$ in $keywords\_list$ **do**
    $keyword\_node \leftarrow$ NEW_NODE ($keyword$)
end for
for each $node$ in PDG_1 **do**
    if $node$ has $keyword$ in $keywords\_list$ **then**
        Connect the $node$ with $keyword\_node$
    end if
end for
Do the same thing for PDG_2 to get merged graph $PDG_1\_PDG_2$
// Feature Extraction
// Code Feature Extraction
$vector\_code_1 \leftarrow []$
for each $node$ in PDG_1 **do**
    $vector \leftarrow$ SENT2VEC ($node$)
    $vector\_code_1 += vector$
end for
Do the same thing for PDG_2 to get $vector\_code_2$
// Structural Feature Extraction
$vectorslist\_structure_1$, $vectorslist\_structure_2 \leftarrow$ STRUC2VEC ($PDG_1\_PDG_2$)
$vector\_structure_1 \leftarrow []$
for each $vector$ in $vectorslist\_structure_1$ **do**
    $vector\_structure_1 += vector$
end for
Do the same thing for $vectorslist\_structure_2$ to get $vector\_structure_2$
$vector_1 \leftarrow$ MEAN ($vector\_code_1$, $vector\_structure_1$)
$vector_2 \leftarrow$ MEAN ($vector\_code_2$, $vector\_structure_2$)
// Verifying
$sim \leftarrow$ SIMI_COS ($vector_1$, $vector_2$)
if $sim >= 0.9$ **then**
    $result \leftarrow 1$
else
    $result \leftarrow 0$
end if

A. Overview

As depicted in Figure 3 and Algorithms 1, *Keybor* consists of four main phases: *PDG Generation*, *PDG Mergence*, *Feature Extraction*, and *Verifying*.

- ***PDG Generation***: The objective of this stage is to generate the corresponding PDG for each method. The methods are provided as input, and the resulting PDGs are produced as output.
- ***PDG Mergence***: This stage combines two PDGs into a unified graph by incorporating keyword nodes. It receives two PDGs as input and outputs the integrated PDG.

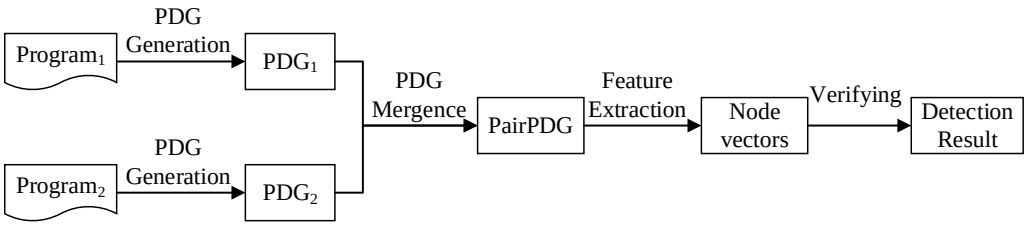

Fig. 3. System architecture of *Keybor*

TABLE I
JAVA KEYWORD TOPIC DIVISION IN DETAIL

Topic	Keywords	Topic	Keywords	Topic	Keywords
Accsess	public, private, protected	Enum	enum	Instanceof	instanceof
Jump	break, continue	Class	class	Assert	assert
Loop	do, for, while	Interface	interface	Char	char
Condition	switch, case, default, if, else	Extends	extends	Goto	Goto
Exception	try, catch, finally	Boolean	boolean	Super	super
Throw	throw, throws	Strictfp	strictfp	This	this
Package	package, import	Abstract	abstract	Const	const
Integer	byte, short, int, long	Native	native	Final	final
Float	float,double	New	new	Void	void
Synchronized	synchronized	Static	static	Transient	transient
Implements	implements	Return	return	Volatile	volatile

- ***Feature Extraction***: In this stage, we extract feature by incorporating both code and structural information. The merged PDG serves as the input, and the output consists of vectors that represent the features of each node in the integrated PDG.
- ***Verifying***: The goal of this stage is to assess whether the code pairs are semantically similar. The stage takes the feature vectors as input and outputs a decision.

B. PDG Generation and Mergence

We utilized the open-source code parsing tool *Joern* [39] to extract the PDG [40] of a function. The PDG provides a detailed and structured graphical representation, facilitating in-depth program analysis and understanding.

Keywords, as a class of syntactic structure in programming languages, can not only label primitive data types but also be used to identify various program structures such as loops, statement blocks, conditions, branches, and so on. Furthermore, the uniqueness of keywords allows them to serve as identifiers in code statements, reflecting the semantic framework of the program to some extent. Therefore, within the PDG merging procedure, we designate keyword nodes as pivotal elements connecting the PDGs associated with the two code segments in a clone pair. This approach aims to more comprehensively depict the inherent relationship between the two code segments.

Moreover, semantic clones manifest as heterogeneous code structures that achieve identical functionality to the original code by employing distinct identifiers, keywords, types, and layouts. For instance, as illustrated in Figure 1, the Type-4 clone employs *for* loop syntax instead of *while* loop syntax, resulting in dissimilarity in text or syntax between the cloned and original code, while maintaining semantic similarity. In this context, if keyword nodes are utilized as pivotal elements in PDG Mergence, the "while" loop statement in the original code would be connected to the keyword node "WHILE", while the "for" loop statement in the Type-4 clone would be connected to the keyword node "FOR". This discrepancy in connecting different keyword nodes poses a challenge in effectively capturing the semantic consistency between the two loop structures.

To alleviate this situation of identical semantics but no connection due to absolute matching and to maintain additional program semantics, we categorize the 50 keywords into 33 distinct topics based on the official usage guidelines for Java keywords [41]. The specific topic categorization is outlined in Table I, the topic "access" encompasses keywords representing access control, namely "public," "private," and "protected"; the topic "loop" includes keywords associated with loop control, such as "for," "while," and "do"; and the topic "integer" encompasses keywords representing integer data types, including "byte," "int," "short," "long". Subsequently, we adopt topic nodes to replace keyword nodes as pivotal elements in connecting PDGs, aiming for a more effective reflection of similarities in code structure and functionality.

To provide a more comprehensive explanation of the detailed steps involved in the proposed method, we elucidate the PDG Mergence process using the original code snippet and Type-4 code clone pair from Figure 1 as an example. Figure 4 depicts the *PairPDG* for this clone pair following step PDG Mergence. Nodes G_i and H_i in Figure 4 respectively represent the statement nodes for the original code snippet and the Type-4 code clone. Each node represents a line of code within the method, and the subscript i denotes the line number of the statement, facilitating precise identification in fine-grained clone detection. Simultaneously, we introduce topic nodes as pivotal elements to connect the two PDGs. Solid edges depict the control/data dependencies within the target program's PDG. Dashed edges connect statements containing keywords with their corresponding topic nodes, facilitating the

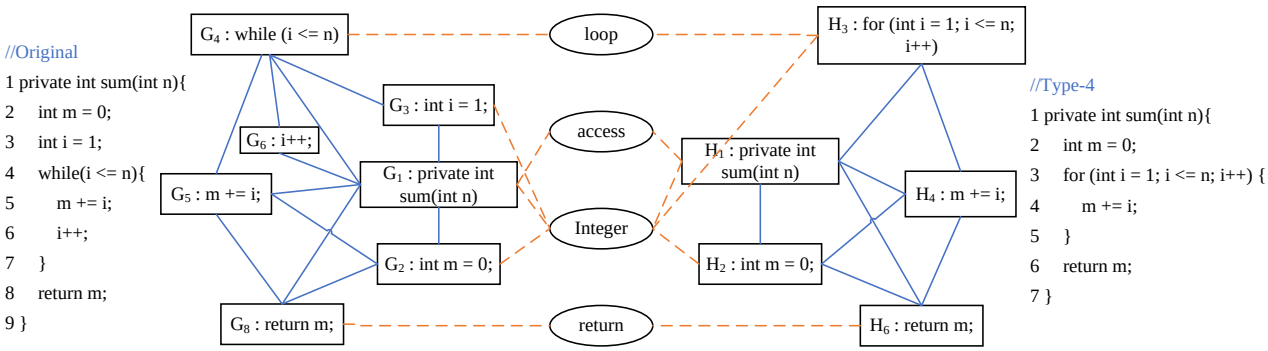

Fig. 4. *PairPDG* generated after the phase PDG Mergence

merging of two separate PDGs into the *PairPDG*. For instance, node G_3 contains the keyword *int*, belonging to the *Integer* category, hence G_3 is connected to the *Integer* topic node.

C. Feature Extraction

For the feature extraction of *PairPDG*, we select a variety of advanced graph embedding methods to map the graph into low-dimensional dense vectors to facilitate the subsequent classification of clone pairs. To enhance the precision in extracting both semantic and syntactic information from the PDG, our proposed feature extraction strategy is divided into two main parts: code information extraction and structural information extraction. The extraction of code information targets the codes corresponding to the nodes in the PDG for word embedding, converting the corresponding codes in the nodes into fixed vectors; The extraction of structural information captures the connections between nodes in the graph by analyzing the edge information within the PDG. Finally, it obtains the structural feature vector of each node by the node embedding algorithm.

1) Code Feature Extraction

The extraction of code features focuses on the extraction of code information from the statement nodes. We treat the code of a statement as text and utilize *Sent2vec* [21] to extract features. *Sent2vec* is an alternative embedding representation that contains the sentiment semantics of a sentence in its embedding vector. Based on *FastText* [42] and *Word2vec* [43], the sentence embedding is computed as the mean of the embeddings of the individual source words that constitute it, and the distributed representation of the sentence is trained by using a simple but effective unsupervised target. The algorithm surpasses the performance of the leading unsupervised model across the majority of benchmark tasks and even outperforms the supervised model on many tasks, highlighting the robustness of the generated sentence embeddings [44], [45].

In contrast to predicting target words from character sequences, *Sent2vec* predicts target words from source word sequences. A wide variety of open-source libraries, including the Linux kernel, are chosen as the training set to train the model to minimize the data leakage problem caused by the overlap of the training and test sets. Subsequently, the code corresponding to each node is fed into the trained model to obtain the code vectors.

2) Structural Feature Extraction

A variety of advanced node embedding models for unsupervised learning of graph-structured data are selected for the extraction of structural information from the synthesized *PairPDG*. Commonly used graph embedding algorithms typically balance between the homogeneity and structural equivalence of the graph. Specifically, the "homogeneity" of the network denotes that the embedding of nodes adjacent to each other should have the closest resemblance to one another (*i.e.,* local neighborhood similarity). As shown in Figure 4, the embedding expressions of node H_3 and its connected nodes H_1, H_4 should be close to each other, which is a reflection of "homogeneity". The term "structural" refers to the embedding of structurally similar nodes that should be as close as feasible. For instance, nodes G_1 and H_1 in the *PairPDG* are connected to the same topic nodes (access and integer), and they exhibit similar data and control dependencies in their respective PDGs. Hence, nodes G_1 and H_1 exhibit analogous structures and architectures. Consequently, their embedding representations should be similar, reflecting "structural similarity". In a straightforward sense, two nodes with the same degree are considered structurally similar, and this similarity is strengthened if their neighboring nodes also exhibit the same degree. In terms of clone detection, nodes with high similarity in the two target programs in the synthesized *PairPDG* are more likely to contain the same keywords (*i.e.,* connect the same keyword nodes) and have similar dependencies with other nodes. As illustrated in Figure 4, the structural similarity of nodes (*e.g.,* G_1 and H_1) offers a clearer indication of the degree of cloning corresponding to each node in the target program. In other words, nodes with a high degree of similarity have a more comparable structure in the graph. Due to their high structural similarity, we can consider that these two nodes are likely to be highly similar in clone detection. Based on this situation, we choose the node embedding models, which are advanced in current research and biased towards structural similarity, to obtain a more accurate vector representation and enhance the precision of clone detection.

In the specific implementation, we adopt two types of node embedding models for feature extraction, one of which is the simple Structural Node Level Embedding Model: *Struc2vec* [20] and *Role2vec* [46]. This class of models merely reads the graph's node connection information and maps it to a vector without accounting for the code information stored in

the nodes, which subsequently needs to be synthesized with the code feature vectors generated by *Sent2vec*. *Struc2vec* is a framework for generating graph node vector representations that retain structural identity. *Struc2vec* employs a hierarchical structure to quantify structural similarities among distinct nodes, constructing a multilayered graph. By traversing this multilayered graph through weighted random walks, it generates random contexts for each node, where nodes frequently appearing in similar contexts may share similar structures. Subsequently, applying *word2vec* [47] to the sampled random walk sequences facilitates the learning of embedding vectors for each node. *Role2vec* proposes role-based graph embedding, utilizing attributed random walks that are independent of vertex identity. Rather than learning separate embeddings for each individual node,*Role2vec* learns each role's embedding by mapping feature vectors to roles through functions that capture the node's role behavior, such that if they are structurally similar, the two vertices belong to the same type.

The other is the Attributed Node Level Embedding Model: *AE* [48] and *MUSAE* [48], which captures information about a node from the local distribution of node attributes around the node. These two algorithms are respectively based on pooling (*AE*) and multiscale (*MUSAE*) learning of local feature information for attribute node embedding. The algorithms consider node "features" as standard basis vectors and use attribute information to complement the local network structure, outperforming similar methods in predicting node attributes, computational scalability, and migration learning. We use the vector generated by *Sent2vec*'s mapping of the node code as an attribute of the node and feed it into the model to get the vector of the node.

D. Verifying

After graph embedding feature extraction, each code line gets a corresponding feature vector. Calculating the similarity between corresponding rows yields fine-grained similarity results. The function-level similarity involves summing the vectors of all rows in the target function to obtain the function feature vector, and then calculating the similarity of the two function vectors. To quantify the similarity, we employ the *Cosine Similarity* as our computational approach. Cosine similarity is a common method for measuring the similarity between two vectors, particularly in text and high-dimensional data analysis. It is commonly applied in areas like text mining, recommendation systems, and clustering analysis, making it especially suitable for handling high-dimensional features. The value of cosine similarity ranges from [-1, 1], where 1 represents perfect similarity, 0 signifies no similarity, and -1 signifies complete opposition, making it easy to understand and interpret. This method assesses their similarity by computing the cosine of the angle formed between the two vectors. The formula is given by:

$$Simi_{cos}(A,B) = cos(\theta) = \frac{\Sigma_{i=1}^{n} A_i B_i}{\sqrt{\Sigma_{i=1}^{n} A_i^2}\sqrt{\Sigma_{i=1}^{n} B_i^2}} \quad (1)$$

When the similarity between two code fragments exceeds the predefined threshold, this pair is identified as a clone pair; otherwise, it is categorized as a non-clone pair.

IV. EXPERIMENTS

In this section, we address the following research inquiries:

- *RQ1: How does the performance of Keybor vary in clone detection under different factors?*
- *RQ2: Can Keybor surpass other clone detectors in performance?*
- *RQ3: How effective is Keybor in fine-grained detection and localization?*
- *RQ4: What is the runtime overhead of Keybor when detecting code clones?*

A. Experimental Settings

1) Dataset.

We choose BCB [22] as the dataset for our experiments. This dataset is extracted from the *IJaDataset* [49] and has undergone manual validation by three experts. *IJaDataset* contains 25,000 items with 365 million lines of code. BCB contains 10 types of questions and more than 8 million labeled clone pairs and 270,000 non-clone pairs. Given the ambiguous boundary between Type-3 and Type-4 clones, BCB classifies Type-3 into four subcategories of weak, moderate, strong, and very strong based on similarity scores assessed through line-level and token-level code normalization, as shown below: i) *Very Strongly Type-3* (VST3), representing a similarity of 90-100%, ii) *Strongly Type-3* (ST3), representing a similarity of 70-90%, iii) *Moderately Type-3* (MT3), representing a similarity of 50-70%, and iv) *Weakly Type-3/Type-4* (WT3/T4), representing a similarity of 0-50%. To experiment, we randomly select 250,000 cloned pairs and 250,000 non-cloned pairs from BCB. The clone pairs we select include 48,116 T1 pairs, 4,234 T2 pairs, 4,577 VST3 pairs, 16,818 ST3 pairs, 76,341 MT3, and 99,914 WT3/T4 pairs.

2) Implementation

We conduct experiments on a server equipped with the Ubuntu 16.04 operating system, 62GB of RAM, and a 32-core Intel Xeon processor. Note that to ensure fairness, like previous studies [2], [24], the scalability evaluation is run with a restricted quad-core CPU and 12GB of RAM. To leverage the potential of the PDG-based method for identifying code clones based on both syntactic and semantic similarity, we incorporate *Joern* [39] to facilitate the generation of PDGs corresponding to the target code. In addition, since the dataset chosen for the experiment uses the JAVA programming language, we choose a Python library *Javalang* [50] to perform lexical analysis to obtain the corresponding token sequences to achieve keyword identification.

3) Comparison

The body of academic work dedicated to clone detection is extensive, making it infeasible to conduct a comprehensive comparison of our approach with every existing method. Since our method is a traditional code similarity-based clone detection approach, rather than one that uses machine learning or deep learning models for training and prediction, we chose not to compare our method with these detectors. Prior research [19] has highlighted two major limitations of learning-based clone detection methods: 1) The studied state-of-the-art deep learning models have been shown to generalize poorly. These methods perform well only on the dataset they are trained

on, but their performance degrades significantly when applied to other datasets. 2) Training them is often time-consuming. These two drawbacks make learning-based clone detection techniques less suitable for real-world clone detection tasks. As a result, we do not include these detectors in our comparison.

Therefore, we conduct a comparative analysis between *Keybor* and the following advanced, widely-used, and non-learning-based code clone detectors: ***CCAligner*** [2]: An advanced detector employing code window extraction and editing distance metrics for code clone identification. ***SourcererCC*** [24]: An advanced code clone detector computing the overlapping similarities of tokens between methods. ***Siamese*** [25]: An advanced detector converting token sequences from source code into diverse manifestations for code clone identification. ***NiCad*** [27]: An advanced code clone detector employing the TXL parser for calculating method similarity. ***NIL*** [26]: An advanced code clone detector calculating the *longest common subsequence* of tokens of methods. ***LVMapper*** [28]: An advanced code clone detector that computes the count of shared tokens and dynamic threshold. ***CCFinder*** [29]: A tool leveraging token-based techniques specifically tailored for detecting clones in large-scale codebases, achieved through comparing the similarity of code token sequences. ***CloneWorks*** [30]: An efficient clone detection tool achieved by computing modified Jaccard similarity. ***Oreo*** [18]: A metric-based code clone detection tool that combines information retrieval and machine learning. ***Deckard*** [9]: A clone detection tool utilizing tree to achieve similarity by computing Euclidean distance between vectors. ***CCGraph*** [7]: A clone detection method based on PDG that employs an approximate graph-matching algorithm. ***Code2Img*** [31]: A scalable code clone detection algorithm utilizing image transformation techniques. ***GPT-3.5-turbo*** [32]: An advanced language model for various applications, from coding assistance to creative writing. ***GPT-4*** [33]: An advanced language model building upon the foundation set by GPT-3.5-turbo. It has improved instruction adherence and is better able to handle complex tasks. When running these tools for performance comparison, we select the parameter configurations that are reported in their respective papers to yield the best results.

4) Metrics

We measure *Keybor*'s effectiveness using the generally accepted metrics as follows: *Precision* (P), *Recall* (R), and *F-measure* (F1). $P = \frac{TP}{TP+FP}$, $R = \frac{TP}{TP+FN}$, $F1 = \frac{2*P*R}{P+R}$. Among them, *true positive* (TP) refers to the samples that are correctly identified as clone pairs, *false positive* (FP) refers to the samples mistakenly identified as clone pairs, and *false negative* (FN) refers to the samples wrongly identified as non-clone pairs.

B. RQ1: Different Factor Setting

To illustrate the effectiveness of different methods and different parameters in *Keybor* clone detection, we set up comparative experiments in this subsection. We choose 250,000 pairs of clones and 250,000 pairs of non-clones mentioned in the experimental data to complete the experiment. The *Keybor* experiment contains three variables: different PDG synthesis methods, different graph embedding tools, and different thresholds.

For the purposes of our experiments, we employ three alternative PDG synthesis methods: Non-keyword connection, Keyword-based connection, and Topic-based connection. The two target code PDGs are combined in step *PDG Mergence* to create the image PairPDG. Non-keyword connection experiment does not process PairPDG. Keyword-based connection experiment adds keyword nodes to PairPDG and connects statement nodes containing keywords to the matching keyword nodes. Topic-based connection experiment improves on the Keyword-based connection experiment by categorizing keywords into 33 different topics according to their usage, adding topic nodes to the PDG, and connecting code nodes to the corresponding topic nodes. Furthermore, we employ four graph embedding tools *Struc2vec*, *Role2vec*, *MUSAE*, and *AE* for feature extraction, and select multiple verification thresholds in the clone detection stage to obtain the optimal value. According to the experimental results obtained by *Keybor* under different variables shown in Figure 5, we derive three conclusions.

First of all, in the case of an ideal overall precision value, the Topic-based connection experiment is superior to the Keyword-based connection experiment and the Keyword-based connection experiment is superior to the Non-keyword connection experiment for the PDG synthesis process. The reason why the Keyword-based connection experiment outperforms the Non-keyword connection experiment lies in the fact that keyword connection provides more information to assist in identifying similarities in the code. The statement nodes containing the same keywords in two target code PDGs can be connected by keyword nodes to capture the common features between them. At the same time, this new PDG retains more semantic information because the keywords can help us identify code with similar semantics in different code blocks. For example, if the keyword is "for", then blocks related to "for" may be linked together to form a larger PDG, so that we can more accurately detect blocks with similar semantics, even if they are in different methods.

Keyword-based connection experiments can help us identify code that has similar semantics in different code blocks. However, it does a poor job of capturing the deeper semantic parallels in the code. In contrast, the Topic-based connection experiment classifies the keywords using 33 different topics as shown in Table I. These topics include data structures, Accsess, Jump, etc. Each topic represents a specific semantic concept. Then, we connect each statement node with the corresponding topic to form a new PDG. This approach not only considers the keywords in the code but also the relationship between these keywords, which can detect the code blocks with similar semantics more accurately, thus improving the accuracy of detection. Note that the Topic-based connection experiment enhances the detection of similarity in heterogeneous code (*i.e.,* utilizing "for" syntax instead of "while"), helping us to better capture similarities in heterogeneous code. However, we found that in some cases (*e.g.,* threshold=0.4), the results without keyword concatenation may be better. This is due to the improper setting of the model threshold at this

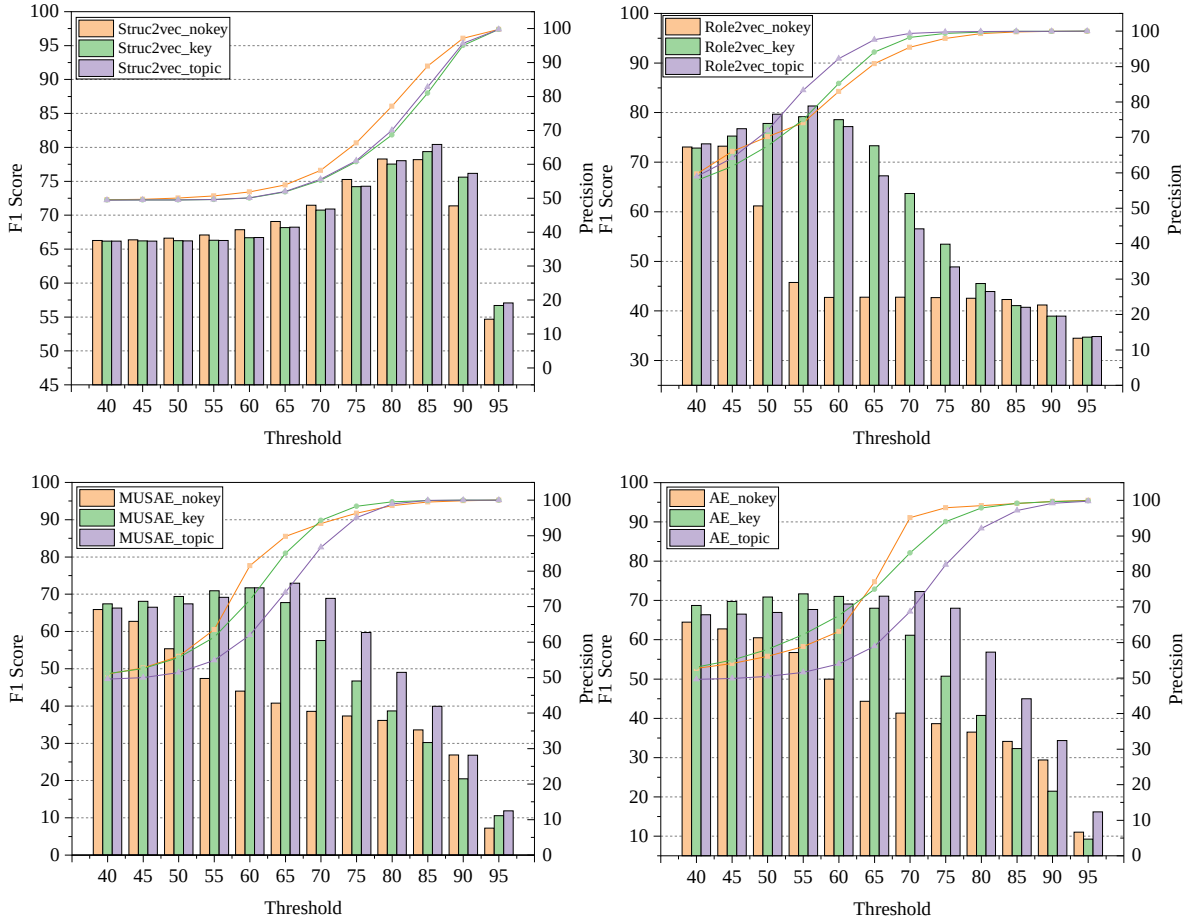

Fig. 5. F1 score and Precision of *Keybor* using different methods and parameters

point, *i.e.,* the precision is too low and the recall is too high, resulting in a low F-score. To solve this problem, we need to readjust the thresholds to ensure that the model works better.

Secondly, we discover that *Struc2vec* outperforms all other graph embedding techniques. Instead of evaluating the structural resemblance between nodes according to their properties and edges or their positional relationships in the network, *Struc2vec* leverages a hierarchical approach to assess node similarity across multiple scales, building a multilayer graph that encodes structural similarities and creates contextual representations for the nodes. Compared with the other three algorithms, *Struc2vec* has a more rigorous notion of what structural similarity means and is more effective at capturing the structural information, thus performing better in clone detection tasks. *Struc2vec*'s accurate mapping of code structure information enables us to have a more comprehensive understanding of the code's structure and the relationship between nodes and increases the accuracy of clone detection and heterogeneous code detection.

TABLE II
DETECTION PERFORMANCE OF *Sent2vec* AND *CodeBERT*

	Recall						P
	T1	T2	VST3	ST3	MT3	WT3/T4	
Kerbor_Sent2vec	100	100	100	96	83	17	96
Kerbor_CodeBERT	100	100	100	95	82	18	96

Finally, the experimental results also show that different thresholds may produce different effects for different methods. Therefore, the selection of thresholds needs to be tailored to each specific case to obtain the optimal detection results. We determine the optimal value for the validation threshold through analysis of the experimental results. As illustrated in Figure 5, we calculate the F1 score and precision for threshold values from 0.4 to 0.95 with a step size of 0.05. We select *Struc2vec* as the graph embedding tool for analyzing experimental results based on the topic connection in light of the preceding conclusion. We observe that precision improves as the threshold increases, while the F1 score also rises, peaking at 0.85, where it reaches 80.43%. Once the threshold surpasses 0.85, the F1 score starts to decline noticeably. When the threshold value is 0.85, the precision is only 82.32%, which is not satisfactory. As a result, the threshold value of 0.9 is eventually selected as the optimal parameter, at which point the F1 score is 76.17% and the precision is 95.69%.

Moreover, to demonstrate the effectiveness of *Sent2vec*, we replace it with the code embedding method *CodeBERT* and record the results in Table II. They indicate that the use of *CodeBERT* and the use of *Sent2vec* demonstrates comparable performance in both recall and precision when detecting various types of clones. However, regarding running speed, the embedding efficiency of *CodeBERT* is quite slow,

TABLE III
DETECTION PERFORMANCE OF *Keybor*, *CCAligner*, *SourcererCC*, *Siamese*, *NIL*, *NiCad*, *LVMapper*, *CCFinder*, *CloneWorks*, *Oreo*, *Deckard*, *CCGraph*, AND *Code2Img* ON BCB (DUE TO THE LIMITED SPACE, WE USE THE ABBREVIATIONS OF THEIR NAMES TO REPRESENT THEM)

Tools		Keybor	CCA	Sou	Sia	NIL	NiCad	LVM	CCF	Clon	Oreo	Dec	CCG	Code
Recall	T1	**100**	100	94	100	99	98	99	100	100	100	60	100	100
	T2	**100**	100	78	96	97	84	99	93	98	99	52	100	100
	VST3	**100**	99	54	85	88	97	98	62	88	100	62	80	99
	ST3	**96**	65	12	59	66	52	81	15	64	89	31	35	93
	MT3	**83**	14	1	14	19	2	19	1	15	30	12	29	25
	WT3/T4	**17**	0	0	0	0	0	0	0	0	0.7	1	11	4
Precision		96	61	100	98	86	99	59	72	96	90	35	95	98

taking an average of 2.12 seconds per function, while *Sent2vec* takes only 0.00049 seconds. Therefore, we chose *Sent2vec* for code embedding.

Summary: *When Struc2vec is chosen as the graph embedding tool, Keybor based on topic connection performs better when the threshold is set to 0.9. Keybor can achieve better performance when selecting Sent2vec as the code embedding method.*

C. RQ2: Comparative Performance

According to the experiments in RQ1, when *Struc2vec* is chosen as the feature extraction tool and connected based on the topic, *Keybor* achieves the most optimal detection efficiency when the threshold is set to 0.9. As a result, we employ this option for subsequent overall effectiveness experiments. We fix the parameters for each tool based on what they claimed in their papers to yield the best results. As with previous methods [18], [26], [27], we calculate the recall for different clone types on the BCB dataset as well as the overall precision across all types and compare these values of *Keybor* with the selected 12 non-LLM tools.

From Table III, we notice that the recall of *Keybor* in each type is much higher than the other experiments while ensuring higher accuracy. With a precision of 96%, our recall in types T1, T2, and VST3 all approximated 100%. The recall in type ST3 is 96%, which is 3% better than the best-performing tool among the remaining tools, *Code2Img*. The recall in type MT3 is 83%, which is 53% better than the best-performing tool, *Oreo*. The recall in type WT3/T4 is 17%, which is 6% better than the best performing tool, *CCGraph*. It indicates that *Keybor* demonstrates a notable improvement in clone detection performance compared to other tools.

We discover that these nine token-based clone detectors are ineffective in detecting Type-3 clones. This is due to the token-based representation form just performing lexical analysis, neglecting the code's structural and semantic information, resulting in a low level of code abstraction. As a result, some complex clones that preserve identical semantic information but differ at the syntactic level cannot be identified. *Deckard* detects semantic clones using a tree-based intermediate representation, which is slightly superior to the token-based clone detection method but performs poorly in terms of overall recall. Since *Deckard* clusters the feature vectors of subtrees based on rules predefined by functions, it is difficult to divide subtrees once they do not match the rules, leading to a decrease in both recall and precision. Another tree-based code clone detection tool, *Code2Img*, performs well in detecting simpler clones, but its effectiveness significantly decreases when dealing with complex syntactic and semantic clones. Although *Code2Img* constructs an adjacency image by normalizing the AST and preserving the code's structural features, its dependence on ASTs, which primarily represent syntax rather than semantics, hinders its effectiveness in detecting semantic clones.

The metric-based clone detection method *Oreo* performs well on types T1, T2, and VST3, but performs poorly for more complex semantic clone detection. Although *Oreo* integrates information retrieval, machine learning, and metric methods and employs deep neural networks to handle the symmetry of the input vectors, its feature extraction approach is still syntactically and textually limited to capturing the semantic information of Type-4 clones. In contrast, a graph-based clone detector is able to more accurately identify semantic similarity between codes since it can consider dependencies between code blocks, not just their syntactic structure. Both *CCGraph* and *Keybor* are graph-based clone detection tools, but *CCGraph*'s recall is still lower than *Keybor*'s. Specifically, *CCGraph* uses a two-stage filtering method along with an approximate graph-matching algorithm based on the Weisfeiler-Lehman (WL) graph kernel for detecting clones. Compared with *Keybor*, *CCGraph* provides relatively less semantic information and a simpler feature extraction method that does not consider the connection between keywords and PDG nodes. Consequently, *CCGraph* may encounter some difficulties in dealing with certain types of clones, while *Keybor* detects them more precisely.

As for *GPT-3.5-turbo* and *GPT-4*, known for their exceptional performance in natural language processing and programming language tasks, to perform code clone detection via their APIs. We use the prompt "Please analyze the following two code snippets and determine if they are code clones. Respond with 'yes' if the code snippets are clones or 'no' if not." to detect clones. Since both large language models (LLMs) require payment for usage, it is not feasible to evaluate all 540,000 code pairs for clone detection. Therefore, we randomly select 500 clone pairs from each clone type, along with 3,000 non-clone pairs, to conduct a small comparative experiment. In order to show that our sampling results are representative of the overall results, we use the two-sample *Kolmogorov-Smirnov Test* (ks-test) [51] to compare whether a substantial difference exists between the sample distribution and the overall distribution. The two-

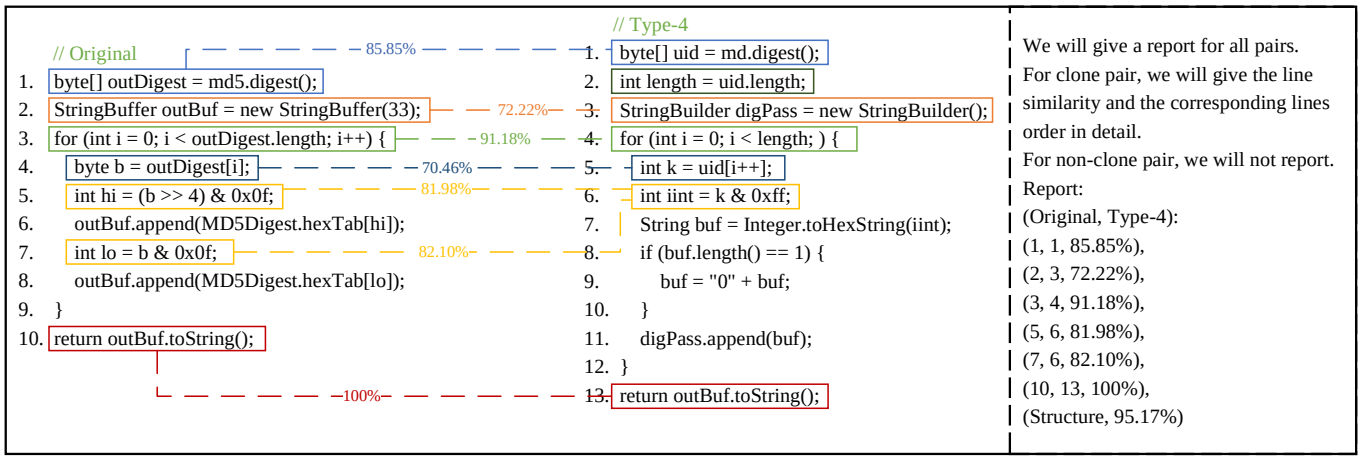

Fig. 6. Fine-grained Analysis Report

sample ks-test is a hypothesis-free distributional, flexible, and efficient statistical method that detects significant differences between two sample distributions and is applicable to data of all distribution types. The BCB dataset assigns token similarity to each code pair. We perform the ks-test with all similarity scores for each clone type (or pairs) as the overall data and the similarity scores of a randomly selected sample of 500 (or 3000) as the sample data. If the p-value of the ks-test is below 0.05, it indicates a significant difference between the sample and overall distributions, and if the p-value exceeds 0.05, it suggests that no significant difference between the two distributions. Our experimental results show that the p-values of the KS test for T1, T2, VST3, ST3, MT3, WT3/T4, and nonclone are 0.89, 0.73, 0.92, 0.75, 0.73, 0.97, and 0.87, respectively. This indicates that our randomly selected samples are representative of the overall situation.

TABLE IV
DETECTION PERFORMANCE OF LARGE LANGUAGE MODELS *GPT-3.5-turbo* AND *GPT-4*

	Recall						P
	T1	T2	VST3	ST3	MT3	WT3/T4	
Keybor	100	100	100	96	81	21	97
GPT-4	100	98	99	94	77	15	96
GPT-3.5	100	57	85	78	59	9	95

The results, as recorded in Table IV, show that *GPT-4* nearly matches the detection performance of *Keybor*. This is because *GPT-4* is able to consider code structure and semantics, recognizing clones even when function names and other details change by identifying structural and functional similarities between code snippets. In contrast, *GPT-3.5-turbo* produces relatively poorer results due to its weaker understanding of code semantics.

Summary: *Keybor achieves superior detection performance and equivalent accuracy compared to existing tools. The results confirm the effectiveness of our detection method. As for LLMs, Keybor can achieve slightly better detection performance than GPT-4.*

D. RQ3: Fine-grained Clone Analysis

Following graph embedding feature extraction in *Keybor*, the nodes for each line of code will receive the matching feature vector. Considering the fine-grained analysis focuses primarily on comparing the similarity of each line of statement code, *Sent2vec* is utilized to extract semantic feature vectors for similarity calculation. In the fine-grained analysis experiment, we compare each statement node to all statement nodes in another target program in turn and evaluate the inter-node similarity by calculating the *Cosine Similarity* between the two node vectors. When the maximum value in the calculated cosine value exceeds the line-level threshold we set, we consider the statements corresponding to these two nodes to be similar and report the corresponding position and similarity of the statements.

We provide detection findings as well as a fine-grained analysis report for cloned pairs. The fine-grained analysis report provides similarity scores between sentences, which serve as a foundation for overall clone vulnerability detection. This allows users to clearly understand where the clones occur and the basis on which the tool identifies them. For non-cloned pairs, we will not report them. To illustrate the *Keybor* fine-grained clone analysis problem more comprehensively and clearly, we randomly select 100 WT3/T4 type clone pairs to evaluate the effectiveness of fine-grained clone analysis. We manually verify the accuracy of the similarity evaluations and find that 98.12% of the results align with the ground truth. A correctly analyzed fragment is shown as an example in Figure 6. According to the report, the overall similarity of the clone pair (Original, Type-4) is 95.17%, which is higher than the set threshold of 90%, so we determine it as a clone pair and give a line-level fine-grained analysis report. In this report, we set the line-level similarity threshold to 70%, and users can adjust it to meet their own needs.

The report shows that variable definition and assignment statements (*e.g.,* (1, 1), (2, 3), (5, 6), (7,6) with similarity of 85.85%, 72.22%, 81.98%, 82.10%, respectively) and for loops (*e.g.,* (3, 4) with 91.18% similarity) achieve high similarity and are accurately identified. These statement pairs present similar syntactic structures to each other and perform localized

TABLE V
THE RUNTIME OF *Keybor* AND 12 OTHER CLONE DETECTION TOOLS

Method	**Keybor**	CCA	Sou	Sia	NIL	NiCad	LVM	Dec	CCF	Clon	Oreo	CCG	Code
1M	**1m2s**	52s	37s	45m1s	10s	1m48s	29s	27m12s	39s	43s	4m22s	15m10s	35s
10M	**47m11s**	26m3s	12m37s	14h11m	1m38s	2h10m	13m38s	-	6m30s	10m35s	2h56m12s	10h21m10s	4m57s
100M	**10h4m49s**	-	2h38m5s	-	1h38m29s	-	17h23m39s	-	-	16h13m34s	5d3h16m	17d37h46m	2h7m2s
250M	**4d9h24s**	-	5d6h55m1s	-	7h40m7s	-	3d13h47m	-	-	2d14h3m	-	-	8h15m37s

additions, modifications, and deletions. For example, the clone statement pair (1, 1) changes the names of the variables *outDigest* and *md5* in the original code to *uid* and *md*, respectively, without altering the semantics of the program. The statement pair (10, 13) has the same code and the calculated similarity is 100%. The report reveals that Keybor captures the contextual and emotional information of an utterance accurately, enabling line-level clone detection.

However, there are instances where fine-grained clone judgments fail. For example, in a pair of clones with complex semantic similarity, one method implemented cloud storage file downloads to the local system, while the other copied files within the local file system. *Keybor* correctly identifies these as clone pairs. However, when analyzing fine-grained similarity, sentences handling data transfers from an input stream to an output stream in both functions (*e.g.,* "IOUtils.copyStream(is, fout);" and "inChannel.transferTo(0, inChannel.size(), outChannel);") are assigned low similarity scores (30.54%). *Keybor* struggles to provide correct indications for sentences where significant changes are made to implement functions.

Summary: *Our line-level similarity analysis technique provides an accurate reflection of the similarity relationships in the corresponding parts of the source code.*

E. RQ4: Scalability Evaluation

This section compares the runtime of *Keybor* with that of 12 other clone detection tools. The LLMs do not support large-scale code clone detection because they cost a lot of money, so we do not compare the time overhead with two LLMs. We use the CLOC [52] tool to divide the entire BCB dataset into four subsets of 1M-, 10M-, 100M-, and 250M-LOC. We then record the time consumption for each dataset.

Table V shows the time performance of each detection tool across different dataset sizes. The "-" indicates that the tool failed to detect the corresponding dataset. As can be seen, for smaller datasets, seven clone detection tools exhibit faster detection speeds. However, as the dataset size increases, some tools either fail to complete the detection or have a longer runtime than *Keybor*. For example, *LVMapper* takes more than 17 hours to detect 100M LOC, and *SourcererCC* takes over five days to process 250M LOC, which takes longer than *Keybor*. *CCAligner* and *Nicad* trigger out-of-memory exceptions when detecting 100M LOC and 250M LOC codebases, indicating their limited scalability. Although *Keybor* is not the fastest, it can scale to datasets as large as 250M LOC, demonstrating better scalability than most other detection tools. Moreover, compared to the graph-based tool *CCGraph*, *Keybor*'s runtime is significantly shorter. In future work, *Keybor* aims to achieve more efficient clone detection by adopting more advanced graph embedding techniques.

Summary: *Although Keybor is not the fastest, it can still scale to 250M LOC and is more efficient than traditional graph-based method.*

V. DISCUSSION

A. Threats to Validity

The different parameters and method settings of clone detection methods have a significant impact on their performance and execution time, and it is difficult to find the most appropriate parameters due to the diversity of threshold choices. To mitigate the threat, we take a series of experiments to calculate F1 scores and precision for different methods with thresholds ranging from 0.4 to 0.95 and steps of 0.05, whereby we evaluate the performance and determine the optimal parameters. In addition, the calculation of time overhead is prone to errors owing to varying machine states. To minimize the risk, we take the average value by repeating the measurement several times to ensure the accuracy and validity of the results. Another potential threat lies in the accuracy of the measurements. We cannot guarantee that the code pairs labeled as "non-clones" are truly non-clones, as there may be instances of oversight during manual annotation. To mitigate this threat, we randomly selected 400 reported results for cross-validation by three independent reviewers, similar to previous works [24]. In addition, the choice of hardware does significantly affect the performance of different methods. In order to ensure fairness and consistency among different methods, the experiments of all methods are run in the same hardware environment. Make sure to use the same operating system, driver version, dependent libraries, and frameworks (*e.g.,* TensorFlow, PyTorch) versions to reduce the performance bias due to the difference in the software environment. Also, the CPU utilization, memory occupation, and other information are recorded in the experiments to confirm whether the method makes full use of the hardware resources. Finally, the inconsistency in the results produced by baseline tools could undermine the reliability of the research findings. To mitigate this threat, we use the parameter settings that are claimed to give the best results in the paper of the baseline. However, the results of the baseline tool were still inconsistent with its paper. We identify two possible reasons for this phenomenon: 1) Differences in hardware, software versions, or configuration parameters may affect experimental results. 2) Subtle differences in the versions of tools or libraries used in the baseline may lead to inconsistent results.

B. Why does Keybor perform better?

First, *Keybor* uses the topic words to connect PDG nodes and establish associations between the nodes of statements containing the same keywords in the PDG of two target codes, so as to capture the common features and the semantic similarities between them at a deeper level. In addition, for the characteristic that two target code similar nodes in synthetic PDG have similar topology, *Keybor* implements the graph embedding algorithm *Struc2vec*, which focuses on structural similarity, for semantic feature extraction. *Struc2vec* constructs a multi-layered graph to capture structural similarity, employing a hierarchical approach to assess node similarity at various levels and generate structural contexts for the nodes. This algorithm can handle complex code structures better, which improves the algorithm's performance and accuracy.

C. Limitation and Future Work

Analyzing 500,000 pairs of code clones requires 2.9 hours, posing a challenge for its application in large-scale clone detection. Therefore, we propose integrating *Keybor* with other advanced rapid clone detection tools, positioning it as the second step in the large-scale clone detection process to complement granularity and accuracy. Through this integration, we aim to maintain efficiency while leveraging *Keybor*'s strengths in meticulous and precise clone detection. This collaborative strategy holds the potential to achieve more desirable performance and outcomes in large-scale clone detection tasks.

Notably, throughout the entire process of clone detection, the graph embedding algorithm *Struc2vec* incurs the majority of the time expenditure, accounting for 90% of the overall detection time. This observation highlights the potential space for optimizing detection efficiency. In future work, we aim to explore lightweight graph embedding algorithms to balance performance and time costs, reducing computational load while preserving accuracy for large-scale software systems.

In addition, this paper focuses on detecting clones of Java code. However, with only slight adjustments, *Keybor* can be expanded to encompass additional programming languages. For instance, we can apply *Joern* [39] to generate PDG and utilize *pycparser* [53] to extract keywords from C source code. Then, we can employ the same embedding algorithm and similarity calculation technique to detect clones of C code.

VI. RELATED WORK

This section reviews existing methods for clone code detection, mainly including text-based, token-based, tree-based, graph-based, and metric-based techniques.

Text-based clone detection methods [27], [54]–[60] focus solely on comparing source code as sequences of lines or strings, ignoring the semantic meaning behind the code structure. Johnson [54] extracts the fingerprints of the functions separately and detects the clones via the fingerprint-matching method. Therefore, it can find almost only textual duplicates and does not identify functionally similar code. [27] also treats the code as text and utilizes the longest common subsequence to detect hidden clones. These methods focus solely on text information, using simple string matching to detect clones, without considering program semantics or logic. As a result, they can only identify complete clones or those with significant textual similarity, failing to detect Type-3 or semantic clones.

The clone detection techniques based on token [2], [16], [24], [26], [29], [48], [61], [62] extract the target program's token sequences by lexical analysis, and after that detecting code clones by analyzing the repeated token subsequences. *CCFinder* [29] detects clones by extracting token sequences and applying transformation rules. *SourcererCC* [24] detects clones by computing the overlap similarity and setting a threshold, which has a simple algorithm principle and good scalability. *CCAligner* [2] excels at detecting clones with large differences. *NIL* [26] uses N-gram token sequences, and the introduced inverted indexing method can effectively identify clone candidates, and the longest common subsequence to verify clone candidates. It is perhaps within the range of capability when detecting the first three clones. Yet, these approaches cannot excavate Type-4 clones.

Tree-based clone detection techniques [9]–[15], [63], [64] parse the code as a tree and detect clones rely on a match between two trees *Deckard* [9] makes use of an algorithm named *Locality Sensitive Hashing* (LSH) to complete code clone detection through clustering similar vectors obtained from ASTs and applies to multiple languages. *CDLH* [10] uses the Tree-LSTM [65] to represent the normalized binary tree to form a vector. Instead of analyzing the whole abstract syntax tree directly, *ASTNN* [11] divides the AST into sub-trees by their own rules, then encodes the sub-trees separately, and finally integrates the encoded vectors from sub-trees into the complete vector representation with a bidirectional recurrent neural network model. [12] is the first approach to analyze tree-paths and thus perform clone detection. The vector representation of tree-paths is learned by a compare-aggregate model that computes the similarity of two vectors to detect does these two codes are cloned or not.

Graph-based clone detection techniques [3]–[8], [66], [67] extract graph representations of programs. Traditional graph-based methods perform clone detection through subgraph matching, *e.g.,* [3] and [4]. However, this approach usually takes a lot of time, so, *CCGraph* [7] first reduces the size of the PDG, then performs a first rough filter using some numerical features, and finally for each category obtained by first filter, a second filter is performed using the similarity of strings. *DeepSim* [6] treat the code similarity issue as a binary classification issue to detect clones. It can effectively detect semantic clones with completely different syntax. *SCDetector* [8] regards graphs as a social network to extract semantic tokens for code blocks, and detects Type-4 clones by combining the rapid processing capability with the semantic richness. [66] captures semantic information at multiple levels and speeds up semantic clone detection by generating uniform graph embeddings for parallel processing. [67] uses the FC-PDG and R-GCN to capture code context and detect clones.

Metric-based clone detection techniques [18], [68]–[75] leverage code attributes to measure the similarity between two code snippets. These metrics can be derived from the source code or the details of the tree or graph. For example, [69] and [68] extract semantic features in the AST to identify code clones. [70] uses the metrics retrieved from the source

code to detect clones. [18] combines software metrics with machine learning and information retrieval to detect clones, thus achieving high precision and recall.

The approaches based on text and token are the quickest, but they neglect program structure and semantics, and thus cannot discover semantic code clones. Tree-based and graph-based methods offer superior accuracy by capturing both syntactic and semantic information more effectively, but they either extract only single syntactic information or neglect the association between code structures, resulting in poor performance in recognizing complicated semantic codes. In our method, we construct a bipartite graph framework that links the PDGs of two methods using keywords. Following that, for semantic feature extraction, a graph embedding approach concentrating on structural similarity is employed, and measure their similarity through *Cosine Similarity*. The addition of keywords and graph embedding algorithms enable *Keybor* to achieve accurate and effective detection of complex semantic clones.

VII. CONCLUSION

This paper introduces an innovative PDG-based approach for detecting complex semantic clones. In our approach, we utilize keywords as pivotal links connecting the two PDGs corresponding to the clone pair, thereby enhancing the semantic associations between similar statements. Furthermore, we conduct line-by-line similarity matching for statements within clone pairs, yielding more fine-grained results in clone detection. We perform comparative evaluations on *BigCloneBench* [22], [23] dataset. Experimental results indicate that *Keybor* significantly outperforms the 14 state-of-the-art code clone detection systems *CCAligner* [2], *SourcererCC* [24], *Siamese* [25], *NIL* [26], *NiCad* [27], *LVMapper* [28], *CCFinder* [29], *CloneWorks* [30], *Oreo* [18], *Deckard* [9], *CCGraph* [7], *Code2Img* [31], *GPT-3.5-turbo* [32], and *GPT-4* [33].

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the Key Program of National Science Foundation of China under Grant No. 62172168.

REFERENCES

[1] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, "Machine learning is all you need: A simple token-based approach for effective code clone detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[2] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: A token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018, pp. 1066–1077.

[3] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, 2001, pp. 301–309.

[4] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 2001 International Static Analysis Symposium (ISAS'01)*, 2001, pp. 40–56.

[5] M. Wang, P. Wang, and Y. Xu, "Ccsharp: An efficient three-phase code clone detector using modified pdgs," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*, 2017, pp. 100–109.

[6] G. Zhao and J. Huang, "Deepsim: Deep learning code functional similarity," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*, 2018, pp. 141–151.

[7] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "Ccgraph: A pdg-based code clone detector with approximate graph matching," in *Proceedings of the 35th International Conference on Automated Software Engineering (ASE'20)*, 2020, pp. 931–942.

[8] 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 *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*, 2020, pp. 1000–1012.

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

[10] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the 2017 International Joint Conferences on Artificial Intelligence (IJCAI'17)*, 2017, pp. 3034–3040.

[11] 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 *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*, 2019, pp. 783–794.

[12] H. Liang and L. Ai, "Ast-path based compare-aggregate network for code clone detection," in *Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN'21)*, 2021, pp. 1–8.

[13] Y.-B. Jo, J. Lee, and C.-J. Yoo, "Two-pass technique for clone detection and type classification using tree-based convolution neural network," *Applied Sciences*, vol. 11, no. 14, pp. 1–18, 2021.

[14] J. Pati, B. Kumar, D. Manjhi, and K. K. Shukla, "A comparison among arima, bp-nn, and moga-nn for software clone evolution prediction," *IEEE ACCESS*, vol. 5, no. 1, pp. 11 841–11 851, 2017.

[15] S. Chodarev, E. Pietrikova, and J. Kollar, "Haskell clone detection using pattern comparing algorithm," in *Proceedings of the 13th International Conference on Engineering of Modern Electric Systems (EMES'15)*, 2015, pp. 1–4.

[16] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Proceedings of the 2017 International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 249–260.

[17] 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, 2021.

[18] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*, 2018, pp. 354–365.

[19] E. Choi, N. Fuke, Y. Fujiwara, N. Yoshida, and K. Inoue, "Investigating the generalizability of deep learning-based clone detectors," in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 2023, pp. 181–185.

[20] L. F. Ribeiro, P. H. Saverese, and D. R. Figueiredo, "struc2vec: Learning node representations from structural identity," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 385–394.

[21] M. N. Moghadasi and Y. Zhuang, "Sent2vec: A new sentence embedding representation with sentimental semantic," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 4672–4680.

[22] "Bigclonebench," https://github.com/clonebench/BigCloneBench, 2022.

[23] 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 *Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 476–480.

[24] 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 (ICSE'16)*, 2016, pp. 1157–1168.

[25] C. Ragkhitwetsagul and J. Krinke, "Siamese: scalable and incremental code clone search via multiple code representations," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2236–2284, 2019.

[26] T. Nakagawa, Y. Higo, and S. Kusumoto, "Nil: large-scale detection of large-variance clones," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 830–841.

[27] C. K. Roy and J. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 2008 International Conference on Program Comprehension (ICPC'08)*, 2008, pp. 172–181.

[28] 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 30th AAAI Conference on Artificial Intelligence (AAAI'16)*, 2016, pp. 1287–1293.

[29] 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.

[30] J. Svajlenko and C. K. Roy, "Cloneworks: a fast and flexible large-scale near-miss clone detection tool." in *ICSE (Companion Volume)*, 2017, pp. 177–179.

[31] Y. Hu, Y. Fang, Y. Sun, Y. Jia, Y. Wu, D. Zou, and H. Jin, "Code2img: Tree-based image transformation for scalable code clone detection," *IEEE Transactions on Software Engineering*, 2023.

[32] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[33] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

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

[35] 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.

[36] M. Belkin and P. Niyogi, "Laplacian eigenmaps for dimensionality reduction and data representation," *Neural computation*, vol. 15, no. 6, pp. 1373–1396, 2003.

[37] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[38] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*. Addison-Wesley Professional, 2000.

[39] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, 2014, pp. 590–604.

[40] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[41] "Java keyword reference," https://docs.oracle.com/cd/E13226_01/workshop/docs81/pdf/files/workshop/JavaKeywordReference.pdf, 2022.

[42] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[43] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[44] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: an image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2365–2376.

[45] A. Kerrigan, K. Duarte, Y. Rawat, and M. Shah, "Reformulating zero-shot action recognition for multi-label actions," *Advances in Neural Information Processing Systems*, vol. 34, pp. 25 566–25 577, 2021.

[46] J. B. L. T. L. W. R. Z. X. K. H. E. Nesreen K. Ahmed, Ryan Rossi, "Learning role-based graph embeddings," *International Joint Conference on Artificial Intelligence(IJCAI) StarAI*, pp. 1–8, 2018.

[47] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[48] Y. Golubev, V. Poletansky, N. Povarov, and T. Bryksin, "Multi-threshold token-based code clone detection," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*, 2021, pp. 496–500.

[49] "Ijadataset2.0," http://secold.org/projects/seclone, 2013.

[50] "A pure python library for working with java source code, provies a lexer and parser targeting java 8. (javalang)," https://pypi.org/project/javalang/, 2022.

[51] F. J. Massey, "The kolmogorov-smirnov test for goodness of fit," *American Statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[52] "Count lines of code," https://github.com/AlDanial/cloc, 2024.

[53] "pycparser is a complete parser of the c language," https://pypi.python.org/pypi/pycparser/, 2022.

[54] J. H. Johnson, "Substring matching for clone detection and change tracking," in *Proceedings of the 1994 International Conference on Software Maintenance (ICSM'94)*, 1994, pp. 120–126.

[55] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*, 1999, pp. 109–118.

[56] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *Proceedings of the 11th IEEE International Workshop on Software Clones (IWSC'17)*, 2017, pp. 1–7.

[57] S. Kim and H. Lee, "Software systems at risk: An empirical study of cloned vulnerabilities in practice," *Computers Security*, vol. 77, no. 1, pp. 720–736, 2018.

[58] S. Jadon, "Code clones detection using machine learning technique: support vector machine," in *Proceedings of the 2016 IEEE International Conference on Computing, Communication and Automation (ICCCA'16)*, 2016, pp. 299–303.

[59] D. Yu, J. Wang, Q. Wu, J. Yang, J. Wang, W. Yang, and W. Yan, "Detecting java code clones with multi-granularities based on bytecode," in *Proceedings of the 41st IEEE Annual Computer Software and Applications Conference (COMPSAC'17)*, 2017, pp. 317–326.

[60] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP'17)*, 2017, pp. 595–614.

[61] N. Göde and R. Koschke, "Incremental clone detection," in *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (ECSMR'09)*, 2009, pp. 219–228.

[62] Y.-L. Hung and S. Takada, "Cppcd: A token-based approach to detecting potential clones," in *Proceedings of the 14th IEEE International Workshop on Software Clones (IWSC'20)*, 2020, pp. 26–32.

[63] Y. Yu, Z. Huang, G. Shen, W. Li, and Y. Shao, "Astsdl: predicting the functionality of incomplete programming code via an ast-sequence-based deep learning model," *Science China Information Sciences*, vol. 67, no. 1, p. 112105, 2024.

[64] K. Z. Y. L. Y. C. Xiaobing SUN, Xin PENG, "How security bugs are fixed and what can be improved: an empirical study with mozilla," *SCIENCE CHINA Information Sciences*, vol. 62, no. 1, pp. 019 102–, 2019.

[65] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[66] D. Yu, Q. Yang, X. Chen, J. Chen, and Y. Xu, "Graph-based code semantics learning for efficient semantic code clone detection," *Information and Software Technology*, vol. 156, p. 107130, 2023.

[67] B. Hu, Y. Wu, X. Peng, C. Sha, X. Wang, B. Fu, and W. Zhao, "Predicting change propagation between code clone instances by graph-based deep learning," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 425–436.

[68] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of the 1996 International Conference on Software Maintenance (ICSM'96)*, 1996, pp. 244–253.

[69] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Proceedings of the 6th International Software Metrics Symposium (ISMS'99)*, 1999, pp. 292–303.

[70] J. F. Patenaude, E. Merlo, M. Dagenais, and B. Laguë, "Extending software quality assessment techniques to java systems," in *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99)*, 1999, pp. 49–56.

[71] C. Ragkhitwetsagul, J. Krinke, and B. Marnette, "A picture is worth a thousand words: Code clone detection based on image similarity," in *Proceedings of the 12th IEEE International Workshop on Software Clones (IWSC'18)*, 2018, pp. 44–50.

[72] S. M. F. Haque, V. Srikanth, and E. S. Reddy, "Generic code cloning method for detection of clone code in software development," in *Proceedings of the 2016 International Conference on Data Mining and Advanced Computing (SAPIENCE'16)*, 2016, pp. 340–344.

[73] M. Sudhamani and L. Rangarajan, "Code clone detection based on order and content of control statements," in *Proceedings of the 2nd IEEE International Conference on Contemporary Computing and Informatics (ICCCI'16)*, 2016, pp. 59–64.

[74] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE'17)*, 2017, pp. 27–30.

[75] M. Tsunoda, Y. Kamei, and A. Sawada, "Assessing the differences of clone detection methods used in the fault-prone module prediction," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, 2016, pp. 15–16.