Learning Graph-based Code Representations for Source-level Functional Similarity Detection

***Abstract*—Detecting code functional similarity forms the basis of various software engineering tasks. However, the detection is challenging as functionally similar code fragments can be implemented differently, e.g., with irrelevant syntax. Recent studies incorporate program dependencies as semantics to identify syntactically different yet semantically similar programs, but they often focus only on local neighborhoods (e.g., one-hop dependencies), limiting the expressiveness of program semantics in modeling functionalities. In this paper, we present TAILOR that explicitly exploits graph-structured code features (e.g., multi-hop neighbors) for functional similarity detection. Given source-level programs, TAILOR first represents them into code property graphs (CPGs) — which combine abstract syntax trees, control flow graphs, and data flow graphs — to collectively reason about program syntax and semantics. Then, TAILOR learns representations of CPGs by applying a CPG-based neural network (CPGNN) to iteratively propagate information on them. It improves over prior work on code representation learning through a new graph neural network (GNN) tailored to CPG structures instead of the off-the-shelf GNNs used previously. We systematically evaluate TAILOR on C and Java programs using two public benchmarks. Experimental results show that TAILOR outperforms the state-of-the-art approaches, achieving 99.8% and 99.9% F-scores in code clone detection and 98.3% accuracy in source code classification.**

I. INTRODUCTION

Detecting code functional similarity serves as the cornerstone of various software engineering tasks, such as code clone detection [1], [2], source code classification [3], [4], and vulnerability discovery [5], [6]. Recently, intensive research efforts have been invested in comprehending source-level program functionalities [7]. A common paradigm is to extract latent features (a.k.a. embeddings) to represent code fragments and calculate distances among embedding vectors to measure functional similarities. Based on how code fragments are used, existing solutions generally fall into three categories: token-based, tree-based, and graph-based methods.

Token-based methods treat source code as natural language texts and model code fragments by parsing them into token sequences [8]–[10]. However, due to the lack of program structures, these solutions fail to recognize textually different yet structurally similar code fragments. For example, if only comparing token sequences, $a = b + c$ and $a = c + b$ are not considered equivalent, even though they share identical syntactical structure. To integrate structural knowledge embedded in code fragments, *tree-based methods* detect program similarities in the form of abstract syntax tree (AST) [11]–[15]. Nevertheless, AST is agnostic to program semantics (e.g., control flow), limiting its ability to identify semantically similar programs with different syntax. As a remedy, *graph-based methods* build program dependency graphs (PDGs) to incorporate control/data dependencies, predicting semantically similar functionalities by discovering isomorphic subgraphs [16]. While achieving higher performances, they are bounded by the low efficiency of graph isomorphism detection [17]. Alternatively, recent work [18], [19] transforms the problem of finding isomorphic subgraphs into matching graph patterns. However, these techniques often focus on local neighborhoods (e.g., one-hop neighbors in PDGs), limiting the expressiveness of program semantics (e.g., multi-hop control dependencies). For example, DeepSim [18] encodes control/data flow into an adjacency matrix, but this matrix only describes first-order dependency between two variables.

To overcome the limitations above, we aim to develop a new approach that can exploit graph-structured code features explicitly, effectively, and efficiently. Towards this end, we take inspiration from the recent advances of graph neural networks (GNNs) in many domains such as social networks [20], recommendations [21], and security [22]. The key of GNNs is to learn graph structures by propagating node representations along graph paths. This leads to the expressive modeling of multi-hop neighbors, injecting structural knowledge into graph representation learning. Intuitively, considering the graph nature of code features (e.g., control flow graph), GNNs are beneficial to the reasoning of program semantics toward more effective functional similarity detection. Moreover, GNNs also excel at efficiency, with a runtime complexity linear to the size of input graphs.

Various approaches have been proposed to explore the potential of GNNs in program analysis. For example, Allamanis et al. [23] adopt the gated graph neural network (GGNN) [24] to predict the name of a variable given its usage. Similarly, FA-AST [25] also leverages the GGNN but for the task of code clone detection. While promising, these approaches share a major drawback — *their learning models are directly inherited from off-the-shelf GNNs without any justification or customization*. Unfortunately, existing GNNs, originally not designed for program analysis, likely include operations not necessarily useful for modeling program functionality, which not only negatively increases the training difficulty but degrades GNN's effectiveness. For example, GGNN applies a recurrent neural network (RNN) to all nodes in a graph, requiring storing intermediate states of nodes in the memory, making it not scale well to large programs [26]. More importantly, we empirically find that replacing the RNN with a much simpler weighted summation [27] brings an improvement in code clone detection. Following this guidance, we strive to design a new

neural network architecture that includes only the essential mechanism of GNNs (e.g., neighborhood propagation) to capture graph-structured code features. A GNN-based model designed this way can be not only easy to train but boost the performance of existing GNNs.

In this paper, we propose TAILOR that detects code functional similarity through a neural network architecture tailored to learn graph-based code representations. First, we need to decide what kind of code representations includes the key feature to describe functionalities. As different code representations are designed for unique program characteristics, we choose to combine them into a joint graph structure named *code property graph* (CPG) [28], providing a comprehensive view of code functionalities. In particular, we integrate ASTs, control flow graphs, and data flow graphs into the CPG as they provide fundamental syntactical and semantic features for program analysis [29]. Thereafter, we design a CPG-based neural network (CPGNN) to distill useful features in the CPG for functional similarity detection. More specifically, CPGNN iteratively propagates program embeddings over the CPG to refine them. By further stacking multiple propagation iterations, it enforces the program embeddings to integrate graph-structured CPG patterns to predict similar functionalities.

We evaluate TAILOR on two code functional similarity detection tasks (code clone detection and source code classification) using two public benchmarks (OJClone [4] and BigCloneBench [30]). Experimental results show that our approach outperforms the state-of-the-art solutions: token-based methods [8], [10], tree-based methods [12], [14], [15], [31], and graph-based methods [19], [25], [32]. Besides, our CPGNN is superior to four widely-used off-the-shelf GNNs (GCN [27], LightGNN [33], GGNN [24], and KGAT [21]). Specifically, for code clone detection, TAILOR achieves F-scores of 99.9% and 99.8% on the OJClone and BigCloneBench datasets, respectively. For source code classification, TAILOR achieves an accuracy of 98.3%.

In summary, we make the following contributions:

- We present TAILOR that explicitly learns graph-based code representations for two functional similarity detection tasks, i.e., code clone detection and source code classification.
- We design a novel neural network architecture (CPGNN) tailored to exploit graph-structured features from code property graphs to generate high-quality code representations.
- We conduct extensive experiments on two public datasets. The results show that TAILOR achieves state-of-the-art performances on both code clone detection and source code classification. All the artifacts (code, data, and logs) are available at https://anonymous.4open.science/r/ICSE_Tailor.

II. PRELIMINARIES

A. Learning Code Representations

A code fragment is a contiguous segment of source code [34] that can be specified as $c = (f, s, e)$, including the source file f, and the lines where c starts from s and ends at e. To automate software engineering tasks, a code fragment is typically encoded as a machine-ingestible representation, i.e., an embedding vector z_c. However, the major challenge is how to obtain high-quality code representations that preserve both program syntactical and semantic features.

Traditional signature-based approaches rely on hand-crafted features (e.g., variable type) to represent code fragments [18], [35]. However, it is difficult to guarantee that manually engineered features are beneficial for specific tasks (e.g., code clone detection). To overcome this limitation, researchers have recently applied deep learning (DL) to transform code fragments into embeddings automatically. Instead of requiring expert knowledge to define code features, DL employs neural networks — optimized for software engineering objectives — to predict the most beneficial code representations. In this way, the DL-based approaches achieve state-of-the-art performances in various tasks [6], [12], outperforming heuristics-based approaches. Before being fed into a neural network, a code fragment is usually converted into an intermediate representation (e.g., token streams, statement sequences, or abstract syntax trees) to abstract useful features of the underlying program. A neural network suitable to process the intermediate representation is then adopted to learn the corresponding features. For example, ASTNN [14] treats a code fragment as a sequence of statements so that it leverages a recurrent neural network (RNN) to parameterize code fragments. InferCode [15] takes an abstract syntax tree as input and thus employs a tree-based convolutional neural network (TBCNN) to generate its neural representation.

In this paper, we represent a code fragment into a graph structure, code property graph (CPG). Afterward, a CPG-based graph neural network (CPGNN) is tailored to suit the need for learning the graph-based code representation.

B. Detecting Code Functional Similarity

Given a code fragment of interest c^*, we aim to examine a corpus of code fragments $\mathcal{C} = \{c_1, c_2, ...\}$ and identify candidates that are functionally equivalent or similar to c^*. This problem is referred to as code functional similarity detection. At its core is to quantify the similarity between two code fragments in the form of embedding vectors, $sim(z_{c^*}, z_{c_j}), c_j \in \mathcal{C}$. A straightforward solution is adapting certain (e.g., Euclidean and Cosine) distance metrics. Especially, the shorter the distance is, the more functionally similar the two code fragments are to each other. However, distance metrics assume that each dimension of an embedding equally contributes to the measurement of code similarity, which does not necessarily hold in practice [18]. To address this issue, recent studies [14], [15], [19] integrate the distance metric as an objective function into the pipeline of code representation learning. This design enables a neural network to automatically weigh the importance of different dimensions in z_c through backward propagating supervision signals of code similarity.

III. OVERVIEW

Figure 1 presents an overview of the TAILOR architecture. It receives a corpus of code fragments and detects functional

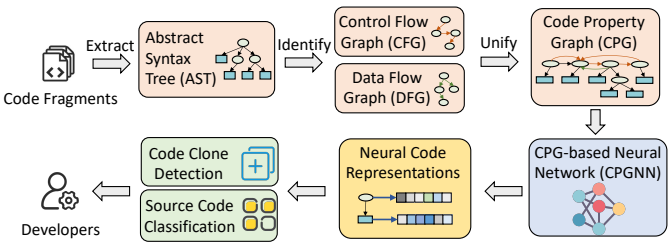


Fig. 1: Overview of TAILOR's Architecture.

similarity among them, which enables code clone detection and source code classification. TAILOR consists of three main steps: (1) parsing code fragments into code property graphs; (2) learning code representations using a tailored graph neural network; (3) predicting code functional similarities.

To reason about code functionality from different aspects collectively, we combine three classic code representations — abstract syntax tree (AST), control flow graph (CFG), and data flow graph (DFG) — into a joint data structure called code property graph (CPG). Specifically, we first extract the AST from a code fragment and identify control and data dependencies in the AST to construct the CFG and DFG. Then, the AST, CFG, and DFG are combined into a CPG, an example of which is illustrated in Figure 3, which encodes both program syntax and semantics.

Once the construction of CPGs from code fragments is done, we develop a CPG-based neural network called CPGNN to learn the corresponding representations (i.e., embeddings). The key idea behind our CPGNN is to *iteratively propagate node embeddings along the CPG structure, explicitly incorporating graph-structured program features into the code representation learning process.* After CPGNN refines node embeddings, we aggregate them by pooling to form the vector representations of code fragments. Finally, all vectorized code representations go through a supervised classifier (specifically, a binary classifier for code clone detection and a multi-class classifier for source code classification) to predict code functional similarity scores, which are later compared with the ground truth to optimize the CPGNN.

IV. METHODOLOGY

A. Code Property Graph (CPG)

Various code representations have been developed in program analysis to characterize different program properties. However, individual representations commonly exhibit only particular aspects (e.g., control flow) of code fragments and thus are insufficient to reveal their functionalities.

To provide a comprehensive view of program functionalities, we combine multiple code representations into a joint data structure, namely code property graph (CPG) [28]. In particular, our CPG is designed to integrate three classic code representations, namely an abstract syntax tree (AST), a control flow graph (CFG), and a data flow graph (DFG), because they present essential syntactical and semantic features of the underlying software programs and are not limited to a particular programming language.

In the following subsections, we use a running example in Figure 2 to demonstrate the steps to build a CPG and how the CPG facilitates program analysis (e.g., explaining what the variables *a* and *b* at Line 3 represent).

```
1 bool isfactor(int a, int b) {
2     bool res = false;
3     if (a % b == 0) {
4         res = true;
5     }
6     return res;
7 }
```

Fig. 2: Exemplary Code Fragment.

*1) **Extracting AST**:* AST provides a tree representation to encode the abstract syntactic structure of source code [36]. Formally, we define an AST as $\mathcal{G}_{ast} = (\mathcal{V}_{ast}, \mathcal{E}_{ast})$, where $\mathcal{V}_{ast}$ and $\mathcal{E}_{ast}$ are sets of AST nodes and edges, respectively. Leaf nodes of an AST denote operands (e.g., identifiers), and inner nodes indicate the corresponding operators (e.g., assignments). More specifically, each AST node is composed of a type (e.g., *identifier*) and a token (e.g., *isfactor*) that capture its syntactical and lexical information. The edges in the AST describe how code statements are nested to produce a program. Compared with plain source code, an AST abstracts away irrelevant syntax (e.g., punctuation and white space) for program analysis. In fact, AST is typically the first intermediate representation produced by code parsers (e.g., compilers) and forms the basis for generating other code representations (e.g., CFG) [28]. However, a limitation of the AST is to include only the syntax of code fragments rather than their semantics (e.g., program dependencies), which can be critical for modeling functionalities.

*2) **Identifying CFG**:* CFG enumerates all possible orders in which code statements may be executed. Each order is determined by a set of conditional statements, e.g., *if* and *while*. Towards this end, nodes in a CFG represent statements, and edges indicate transfers of control. To provide program-wide control flows, we consider both intra-procedural and inter-procedural dependencies, which describe control flows within a function (e.g., branches) and across functions (invocations). Although a CFG can be built upon an assembly and/or intermediate language for a particular programming language (e.g., bytecode for Java), we choose to construct a CFG atop an AST to guarantee the generality of our approach.

Technically, we extract intra-procedural control dependencies with the following three steps: (1) identifying code statements sharing the same parent AST node; (2) sequentially connecting these statements by their order (i.e., line number) in a code fragment; (3) if the parent AST node indicates a transfer of control[1], updating statement relationships according to the semantics of the transfer. To extract inter-procedural control dependencies, we also perform a three-step procedure: (1) enumerating all caller functions by locating the AST nodes whose type is *Invocation*; (2) identifying their callee functions by searching for the declared functions with the

[1]In our implementation, we include nine types of control transfers, namely, *if*, *switch*, *while*, *do_while*, *for*, *try_catch*, *break*, *continue*, and *goto*.

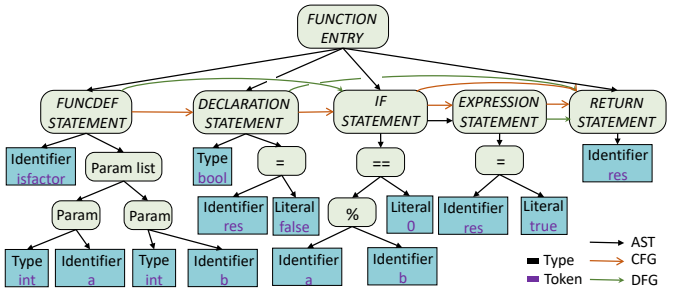


Fig. 3: Code Property Graph built upon Figure 2.

same function name, parameter number, and parameter type; (3) connecting the caller and callee functions by creating invocation and return edges. Formally, our CFG is formulated as $\mathcal{G}_{cfg} = (\mathcal{V}_{cfg}, \mathcal{E}_{cfg})$, where $\mathcal{V}_{cfg}$ are AST nodes of code statements (e.g., *if statement*), and $\mathcal{E}_{cfg}$ denote intra-procedural and inter-procedural control dependencies.

*3) **Identifying DFG**:* DFG describes how variables are defined and used by code statements in an AST. Similar to a CFG, nodes in the DFG denote statements, but edges reflect the influences of statements on variable values. As the DFG is variable-oriented, it is often adopted to support program analysis that requires tracking the life cycle of variables, e.g., detecting and diagnosing information leakage [37]. To identify data dependencies in an AST, we first determine the set of variables defined and used by each statement. Then, we calculate *reaching definitions* to discover connections between variable definitions and uses among statements. Thereafter, we identify *use-def chains* of variables to form data-flow edges in a DFG. Formally, we define the DFG as $\mathcal{G}_{dfg} = (\mathcal{V}_{dfg}, \mathcal{E}_{dfg})$, where $\mathcal{V}_{dfg}$ and $\mathcal{E}_{dfg}$ are AST nodes of code statements and data dependencies, respectively.

*4) **Unifying Representations**:* After extracting the AST from a code fragment and identifying the CFG and DFG, we unify these code representations into a joint CPG. Figure 3 illustrates an example of the CPG built upon the code fragment in Figure 2. With the CPG as the backbone, we capture both syntactical and semantic program features, providing a comprehensive view of code fragments for modeling their functionalities. For example, by collectively tracking the AST and data dependencies between *FUNCDEF STATEMENT* and *IF STATEMENT* in Figure 3, we identify that the variables *a* and *b* used at Line 3 in Figure 2 are, in effect, the arguments of the function *isfactor* defined at Line 1.

A CPG is formally defined as $\mathcal{G}_{cpg} = (\mathcal{V}_{cpg}, \mathcal{E}_{cpg})$, where $\mathcal{V}_{cpg} = \mathcal{V}_{ast}$ and $\mathcal{E}_{cpg} = \mathcal{E}_{ast} \cup \mathcal{E}_{cfg} \cup \mathcal{E}_{dfg}$. CPG nodes denote AST nodes attributed with their types and tokens, and edges reflect AST edges, control dependencies, or data dependencies.

B. CPG-based Neural Network (CPGNN)

We now tailor a new neural network architecture for CPGs (CPGNN) to learn their representations, the workflow of which is presented in Figure 4. CPGNN consists of three key components: (1) embedding initialization, which parameterizes each CPG node as an embedding (i.e., numeric vector) using its type and token; (2) embedding propagation, which updates

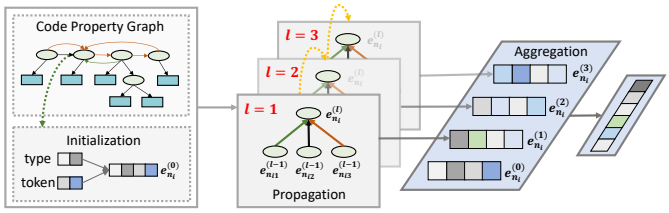


Fig. 4: Illustration of CPG-based Neural Network.

node embeddings by iteratively propagating information from neighbors; (3) embedding aggregation, which combines embeddings from all propagation iterations as the final representation of CPG nodes.

*1) **Embedding Initialization**:* As our CPG is built upon an AST, a CFG, and a DFG, its nodes and edges naturally preserve the syntax and semantics of code fragments. To inject syntactical features into node representations, we first traverse the CPG by following AST edges in pre-order to obtain node type/token symbols as the training corpus. Then, on the corpus, we employ word2vec [38] to project all symbols into a d-dimensional space and collect the corresponding symbol embeddings as $\mathbf{W}_v \in \mathbb{R}^{S \times d}$, where S is the number of unique types/tokens. Afterward, we can retrieve *initial* embeddings of type s_1 and token s_2 by querying $\mathbf{W}_v$ as follows:

$$\mathbf{h}_{s_1} = x_{s_1}\mathbf{W}_v, \quad \mathbf{h}_{s_2} = x_{s_2}\mathbf{W}_v,$$

where $\mathbf{h}_{s_1} \in \mathbb{R}^d$ and $\mathbf{h}_{s_2} \in \mathbb{R}^d$ are the initial embeddings of s_1 and s_2, respectively; $\mathbf{x}_{s_1} \in \{0,1\}^S$ and $\mathbf{x}_{s_2} \in \{0,1\}^S$ separately denote the one-hot encodings of s_1 and s_2.

As each CPG node $n \in \mathcal{V}_{cpg}$ is characterized by both a type s_1 (e.g., *identifier*) and a token s_2 (e.g., *isfactor*), we use the fusion of token and type embeddings to describe n's representation as follows:

$$\mathbf{e}_n = \mathbf{h}_{s_1} \| \mathbf{h}_{s_2},$$

where $\|$ denotes the concatenation operator. By doing so, the node representation $\mathbf{e}_n$ includes both syntactical and lexical information. Note that for inner CPG nodes without token symbols, we pad an empty token embedding for concatenation.

For prior graph-based methods [18], [19], *initial* node representations are directly used to represent a code fragment, which are later fed into a classifier to model its functionality. In contrast, we further refine these node representations by propagating information from their neighbors, so as to latch on to the graph-structured program characteristics.

*2) **Embedding Propagation**:* Beyond the type and token attributes, contexts (i.e., neighbors) also play an important role in depicting CPG nodes. For example, probing into the variable *a* in Figure 3, its node attributes (i.e., type *identifier* and token *a*) only describe its name, which hardly reflects its functionality holistically; whereas, the multi-hop path *FUNCDEF STATEMENT* → *Parameter list* → *Parameter* → *Identifier a* not only presents the neighbors of variable *a*, but also emphasizes that the variable is a function argument. To capture the information carried in such high-order connectivity, we get inspiration from the embedding propagation mechanism

of GNNs [39], [40]. The key idea is to iteratively propagate node representations along with CPG edges to enrich the representations of ego nodes.

Here, we begin with the design of one-iteration propagation and then generalize it to multiple iterations.

First-iteration Propagation. Given a CPG node $n \in \mathcal{V}_{cpg}$, we define its one-hop neighbors as $\mathcal{N}_n = \{m|(m,n) \in \mathcal{E}_{cpg}\}$, where m denote the node directly connecting to n. To further characterize the local neighborhood of n, we compute the neural information propagated from $\mathcal{N}_n$ to n as:

$$\mathbf{e}_{\mathcal{N}_n} = \sum_{m \in \mathcal{N}_n} \alpha(m,n)\mathbf{e}_m,$$

where $\alpha(m,n)$ controls the decay factor on each propagation along with CPG edge (m,n). Intuitively, $\alpha(m,n)$ acts as an attention function that specifies how much information is passed from $\mathbf{e}_m$ to $\mathbf{e}_n$, which normalizes the propagation so that the scale of node embeddings is not linearly increased with propagation paths. We implement it as:

$$\alpha(m,n) = \frac{1}{\sqrt{|\mathcal{N}_m|\,|\mathcal{N}_n|}},$$

where $|\mathcal{N}_m|$ and $|\mathcal{N}_n|$ denote the number of one-hop neighbors for CPG nodes m and n, respectively.

Having obtained the information propagated from neighbors, we next exploit it to update the representation of the CPG node n, formally designed as:

$$\mathbf{e}_n^{(1)} = f(\mathbf{e}_n, \mathbf{e}_{\mathcal{N}_n}),$$

where $f(\cdot)$ is the aggregation function; $\mathbf{e}_n^{(1)} \in \mathbb{R}^{2d}$ is the node representation after the first propagation iteration.

Clearly, the design of function $f(\cdot)$ is at the core of information aggregation. Recent studies [6], [24], [25] have adopted the aggregation function from the off-the-shelf GGNN [24] that is built atop a gated recurrent unit (GRU) as follows:

$$f_{\text{GGNN}} = \text{GRU}\left(\mathbf{e}_n, \mathbf{e}_{\mathcal{N}_n}\right).$$

GRU is originally designed to solve the vanishing gradient problem in neural machine translation [41]. In our case, the gradient could vanish if the propagation iterations grow to a huge number. However, we empirically find that four iterations have been sufficient to capture code features for modeling functionalities. Therefore, GRU is not necessary but heavy to learn code representations, which even negatively increases the difficulty of training (See Table VI for evidence). As such, we choose to develop $f(\cdot)$ as follows:

$$f = \text{LeakyReLU}\left(\left(\mathbf{e}_n \| \mathbf{e}_{\mathcal{N}_n}\right)\mathbf{W_g}\right),$$

where $\mathbf{W_g} \in \mathbb{R}^{4d \times 2d}$ is a trainable transformation matrix to distill useful information from the concatenation; LeakyReLU is a non-linear activation function. In this way, our GNN takes $\mathbf{W_g}$ as the only trainable parameter for one-iteration propagation, making it practically easy to train and revise.

Multi-iteration Propagation. To further leverage code features from multi-hop neighbors, we stack multiple propagation iterations to update CPG node representations. The number of hops in integrating neighbors is determined by the number of propagation iterations L. Formally, in the l-th iteration, the representation of CPG node n is updated as:

$$\mathbf{e}_n^{(l)} = f(\mathbf{e}_n^{(l-1)}, \mathbf{e}_{\mathcal{N}_n}^{(l-1)}),$$

where $\mathbf{e}_n^{(l)}$ is the updated representation after aggregating the information propagated from l-hop neighbors, while $\mathbf{e}_n^{(0)} = \mathbf{e}_n$ is the type/token embedding initialized by word2vec; $\mathbf{e}_{\mathcal{N}_n}^{(l-1)} = \sum_{m \in \mathcal{N}_n} \alpha(m,n)\mathbf{e}_m^{(l-1)}$. The aggregation function $f(\cdot)$ is defined as:

$$f = \text{LeakyReLU}\left(\left(\mathbf{e}_n^{(l-1)} \| \mathbf{e}_{\mathcal{N}_n}^{(l-1)}\right)\mathbf{W_g}^{(l)}\right),$$

where $\mathbf{W_g}^{(l)} \in \mathbb{R}^{4d \times 2d}$ is the trainable transformation matrix at l-th iteration. Notice that the representations of CPG nodes are updated based on the CPG topology in a synchronous fashion. That is, we can compute new representations of all CPG nodes in parallel to guarantee system efficiency.

*3) **Embedding Aggregation**:* After L iterations of information propagation, we establish a series of representations for each single CPG node n, namely $\{\mathbf{e}_n^{(0)}, \mathbf{e}_n^{(1)}, ..., \mathbf{e}_n^{(L)}\}$. The outputs in different iterations highlight neighboring nodes at certain hops. Upon these representations, we apply the concatenation operator to combine them together:

$$\mathbf{e}_n^* = \mathbf{e}_n^{(0)} \| \mathbf{e}_n^{(1)} \| ... \| \mathbf{e}_n^{(L)}.$$

Compared to more complicated combination operators (e.g., GRU, LSTM), the concatenation benefits from its simplicity: involving no additional parameters to train but preserving explicit information pertinent to different hops of CPG neighbors (e.g., the multi-hop path from *FUNCDEF STATEMENT* to *Identifier a* in Figure 3).

In summary, iteratively propagating and aggregating information over a CPG allows us to incorporate graph-structured code features explicitly — covering syntax, semantics, and context — into the representations of CPG nodes.

*4) **Time Complexity Analysis**:* The time cost of CPGNN comes mainly from the embedding initialization, propagation, and aggregation on CPG nodes. By adopting word2vec in skip-gram, the computational complexity to initialize CPG nodes is $O(|P|d(1+log_2(S))$ [42], where $|P|$, d, and S denote the sizes of training corpus, type/token embedding, and type/token symbols, respectively. Note that this initialization is a one-time effort, no matter how many training epochs are performed. More specifically, while training CPGNN, we do not feed the training loss (of code clone detection and code classification) backward to fine-tune word2vec embeddings. In the propagation over CPG, the time complexity of updating node embeddings is $O(L|\mathcal{E}_{cpg}|d^2)$, where L and $|\mathcal{E}_{cpg}|$ denote the numbers of propagation iterations and CPG edges, respectively, and d^2 comes from the matrix multiplication that CPGNN applies for feature transformation. As for the embedding aggregation, only the concatenation is conducted, for which the time cost is $O(\sum_{l=1}^{L}|\mathcal{V}_{cpg}|d)$, where $|\mathcal{V}_{cpg}|$ denotes the number of nodes in a CPG. The overall complexity for training N epochs is

$O(|P|d(1+log_2(S)) + N(L|\mathcal{E}_{cpg}|d^2 + \sum_{l=1}^{L}|\mathcal{V}_{cpg}|d))$ that is linear to the size of a CPG, $\mathcal{G}_{cpg} = (\mathcal{V}_{cpg}, \mathcal{E}_{cpg})$.

C. Neural Code Representation

Having obtained the final embeddings of CPG nodes, we would like to generate a holistic representation of a code fragment. To do so, we hire a pooling function to combine CPG node embeddings as a neural code representation:

$$\mathbf{z}_c = \rho([\mathbf{e}^*_{n_1}, \mathbf{e}^*_{n_2}, ..., \mathbf{e}^*_{n_V}]),$$

where ρ is designed as the average pooling to delineate important features from CPG nodes; $\{\mathbf{e}^*_{n_1}, \mathbf{e}^*_{n_2}, ..., \mathbf{e}^*_{n_v}\}$ collect the representations of CPG nodes $\{n_1, n_2, ..., n_v\}$ derived from CPGNN in Section IV-B3. Alternatively, ρ can also be set as the max pooling, which empirically achieves similar performances in modeling code functionalities.

D. Functional Similarity Detection

In this section, we describe how neural code representations are adopted to solve two code functional similarity detection tasks: code clone detection and source code classification.

*1) **Code Clone Detection**:* This task aims to predict whether a pair of code fragments share similar functionalities. Towards this end, given two code fragments c_i and c_j, we measure their functional similarity by calculating the Euclidean distance between their neural representations $\mathbf{z}_{c_i}$ and $\mathbf{z}_{c_j}$ as:

$$\hat{\mathbf{y}}_{ij} = \text{sigmoid}\left(||\mathbf{z}_{c_i} - \mathbf{z}_{c_j}||\mathbf{W_{ccd}}\right),$$

where $\mathbf{W_{ccd}} \in \mathbb{R}^{2dL\times 1}$ is a trainable transformation matrix to distill important code features for code clone detection; sigmoid converts the predictive score into the probability. If $\hat{\mathbf{y}}_{ij}$ is larger than a pre-defined threshold δ, the code fragment pair is predicted as a clone. Otherwise, the pair is a non-clone.

To optimize our model for the objective of code clone detection, we resort to a binary cross-entropy loss as follows:

$$\mathcal{L}_{ccd} = \sum_{(c_i,c_j)\in\mathcal{O}_{ccd}} (-(\mathbf{y}_{ij} \cdot ln(\hat{\mathbf{y}}_{ij}) + (1-\mathbf{y}_{ij}) \cdot ln(1-\hat{\mathbf{y}}_{ij}))),$$

where $\mathcal{O}_{ccd} = \{(c_i, c_j)|c_i, c_j \in \mathcal{C}\}$ denotes a training set of clone and non-clone pairs; $\mathbf{y}_{ij} \in \{0, 1\}$ provides the ground-truth clone label for the code fragment pair (c_i, c_j).

*2) **Source Code Classification**:* This task intends to classify a corpus of code fragments by their functionalities. Given the neural representation of a code fragment $\mathbf{z}_{c_i}$, we apply a fully-connected layer to classify it into M functionality categories:

$$\hat{\mathbf{y}}_i = \text{softmax}(\mathbf{z}_{c_i}\mathbf{W_{scc}}),$$

where $\mathbf{W_{scc}} \in \mathbb{R}^{2dL\times M}$ is a trainable transformation matrix; softmax converts the predictive score into the probability distribution over M functionality categories. The category that c_i belongs to is determined by $\arg\max(\hat{\mathbf{y}}_i)$.

To optimize our model for source code classification, we opt for a categorical cross-entropy loss as follows:

$$\mathcal{L}_{scc} = \sum_{c_i\in\mathcal{O}_{scc}} (-\mathbf{y}_i \cdot ln(\hat{\mathbf{y}}_i)),$$

where $\mathcal{O}_{scc} = \{c_i|c_i \in \mathcal{C}\}$ denotes a training set of code fragments for classification; $\mathbf{y}_i$ provides the one-hot encodings of ground-truth classification labels.

V. Evaluation

In this section, we focus on evaluating the performance of TAILOR by answering the following research questions (RQs):

- **RQ1:** How does TAILOR perform in the code clone detection and source code classification, compared with the state-of-the-art?
- **RQ2:** How does CPGNN contribute to TAILOR, compared with off-the-shelf GNNs?
- **RQ3:** To what extent do different design choices of CPGNN affect TAILOR's performance?

A. Implementation and Setup

*1) **Implementation**:* We use tree-sitter[2] to extract ASTs from code fragments written in C and Java languages. The rest of the CPG building (e.g., CFG and DFG identification) is developed in 7,020 lines of Python code. We implement our CPGNN using TensorFlow [43] in 3,023 lines of code. The model is optimized by Adam optimizer [44] with a learning rate of 0.1. We train the model for 30 epochs on code clone detection and 250 epochs on source code classification. To mitigate the over-fitting problem, we employ a dropout technique with a dropping ratio of 0.1. All model parameters are initialized with Xavier [45]. For hyper-parameters, we apply a grid search. In light of the best performance, we report experimental results in a setting with the embedding size d of type/token symbols as 16 and the numbers of propagation iterations L as five and four on the OJClone [4] and BigCloneBench [30] datasets. The threshold δ is set to 0.5 for code clone detection. We randomly sample software programs in an experimental dataset to constitute the *disjoint* training, validation, and testing sets, of which the proportions are 80%, 10%, 10%. It is worth mentioning that we strictly follow the recommendations by [46] to avoid common pitfalls in evaluating learning-based approaches. For example, we tune hyper-parameters based solely on the validation set to avoid biased parameter selection.

All experiments are performed on a server with Intel Xeon Gold 6248 CPUs @ 2.50GHz, 188GB physical memory, and two NVIDIA Tesla V100 GPUs with 32GB memory.

*2) **Baselines**:* To demonstrate TAILOR's effectiveness, we perform a thorough literature review to include as many state-of-the-art approaches as possible for empirical comparison, including two token-based (SourcererCC and NIL), four tree-based (RtvNN, Code2Vec, ASTNN, and InferCode), and three graph-based (FCDetector, FA-AST, and Mocktail) approaches. We perform the comparison by detecting functionally similar code fragments at the level of programs. Note that similar programs may consist of different numbers of methods.

- **RtvNN** [12]: This model uses recurrent/recursive neural networks to detect clones using code structures and identifiers.

[2] https://tree-sitter.github.io/tree-sitter

TABLE I: Dataset Statistics.

		OJClone	BigCloneBench
Program		52,000	22,723
LOC	**Sum**	2,021,272	683,745
	Min.	4	4
	Max.	1,305	1,060
	Avg.	39	30
CPG node/edge	**Sum**	12,689,188/18,971,201	5,187,310/6,425,799
	Min.	19/22	8/10
	Max.	4,201/13,112	10,604/12,913
	Avg.	244/365	228/283

TABLE II: Results of Code Clone Detection on OJClone.

Approach	Precision	Recall	F-score	AUC	Time(s)
RtvNN	56.0	91.2	69.4	68.9	**<1**
SourcererCC	9.0	92.4	16.4	54.1	225
Code2Vec	97.6	75.6	85.2	98.6	**<1**
Mocktail	94.8	94.2	94.5	98.9	**<1**
ASTNN	98.3	92.1	95.1	96.0	175
FCDetector	96.3	86.9	91.8	91.8	**<1**
NIL	71.8	43.7	54.3	63.2	98
TAILOR	**100**	**99.7**	**99.9**	**100**	16

The time reported in the table is the time spent on prediction (or inference).

- **SourcererCC** [8]: This is a token-based method, which exploits a bag-of-tokens strategy to detect near-miss clones from large codebases and inter-project repositories.
- **Code2Vec** [31]: Such method encodes code fragments into fixed-length vectors via representing source code as a collection of AST paths and learning their semantic properties. Here we further input code vectors to a fully-connected layer for code clone detection and classification.
- **ASTNN** [14]: This is the state-of-the-art tree-based model to learn code representations, which decomposes ASTs into statement trees to avoid the gradient vanishing problem.
- **FA-AST** [25]: This work is the first to apply GNNs (e.g., GGNN) to the domain of code clone detection.
- **FCDetector** [19]: This graph-based method detects code clones with syntactical and semantic program representations derived from word2vec [38] and graph2vec [47].
- **Mocktail** [32]: This approach extends Code2Vec by integrating additional CFG and PDG paths.
- **InferCode** [15]: This model learns code representations by predicting subtrees identified from the contexts of ASTs. Similar to Code2Vec, we further input the code representations to a fully-connected layer to classify functionalities.
- **NIL** [10]: This is the state-of-the-art token-based method, which verifies clone candidates by calculating the longest common sub-sequences among token sequences.

We have reproduced all the baseline approaches from their authors' open-source implementations except FA-AST. For a fair comparison, we use the same settings while evaluating TAILOR and these baselines. Specifically, while generating training/validation/testing sets for TAILOR, we have labeled to which set a program belongs and followed the same labels to generate training/validation/testing sets for the baselines.

B. RQ1: Performance on Code Clone Detection

*1) **Settings**:* To evaluate how TAILOR performs on code clone detection, we utilize two widely-adopted datasets: OJClone [4] and BigCloneBench (BCB) [30]. Their statistics (e.g., the maximum number of CPG nodes/edges per program) are summarized in Table I.

The OJClone dataset is collected from a pedagogical online judge system, containing 52,000 C programs belonging to 104 programming tasks. We treat two programs solving the same task as a clone pair since they achieve similar functionalities. Following the previous work [13], we generate clone and non-clone pairs using the first 15 programming tasks. For a fair comparison, we further randomly sample clone pairs to guarantee a similar distribution of clone and non-clone pairs to [14]. Finally, we collect 19,800 clone pairs as positive samples and 300,000 non-clone pairs as negative samples.

The BCB dataset is a popular code clone benchmark collected from 25,000 Java software systems, containing 6,000,000 clone pairs and 260,000 non-clone pairs. Each clone pair is manually assigned a clone type (i.e., Type-1, Type-2, Type-3, and Type-4) depending on the line-level and token-level similarities of its constituent code fragments. Specifically, for Type-1 and Type-2 clones, their similarities are 1. As the boundary between Type-3 and Type-4 is often ambiguous, these two types are divided into strongly Type-3 with a similarity between [0.7, 1.0), moderately Type-3 with a similarity between [0.5, 0.7), and weakly Type-3/Type-4 with a similarity between [0.0, 0.5). Following recent literature [14], we sample 20,000 pairs from each clone type as positive samples and 20,000 non-clone pairs as negative samples. Particularly, we fetch all the clone pairs from Type-1, Type-2, and strongly Type-3 as none of them includes more than 20,000 pairs. In total, we collect 71,677 clone pairs.

Note that we evaluate FCDetector only on OJClone since it requires caller-callee relations, but BCB does not contain inter-procedural programs. As Mocktail does not support Java programs, we also evaluate it only on OJClone. Additionally, FA-AST is not publicly available, so we cite the results reported in its paper as we share the same experimental dataset. We have attempted to fine-tune pre-trained code representations from InferCode to detect code clones. However, since InferCode does not perform well and its original paper also leaves the supervised code clone detection as future work, we do not include it in our evaluation.

*2) **Metrics**:* We use Precision, Recall, F-score, and AUC as the evaluation metrics in code clone detection. Specifically, Precision and Recall measure correctly detected clones against all predicted clones and all ground-truth clones, respectively. F-score calculates the harmonic mean of Precision and Recall. AUC, ranging between [0, 1], shows the capability of distinguishing between clone and non-clone pairs. The higher the AUC is, the better TAILOR is at code clone detection.

*3) **Results**:* Experimental results on the OJClone and BCB datasets are listed in Table II and Table III. TAILOR consistently outperforms all the state-of-the-art solutions regarding Precision, Recall, F-score, and AUC in the OJClone dataset. Specifically, TAILOR achieves a relative improvement of 4.8% (99.9% vs. 95.1% from ASTNN) and 1.1% (100% vs. 98.9% from Mocktail) in terms of F-score and AUC, respectively.

From Table III, we also see that TAILOR achieves the best

TABLE III: F-scores of Code Clone Detection on BCB.

Approach	T1	T2	ST3	MT3	WT3/T4	Avg.
RtvNN	95.3	89.3	86.6	79.0	68.5	83.7
SourcererCC	**100**	99.3	80.9	9.2	0.0	57.9
Code2Vec	98.9	97.0	94.2	90.1	84.7	93.0
ASTNN	**100**	99.9	96.7	95.6	93.6	97.2
FA-AST	**100**	**100**	99.8	98.2	94.6	98.5
NIL	99.9	99.5	92.7	35.6	2.9	66.1
TAILOR	**100**	99.9	**99.8**	**99.6**	**99.8**	**99.8**

TABLE IV: Results of Code Clone Detection on Individual Clone Types in BCB.

Clone Type	Precision	Recall	F-score	AUC	Time(s)
T1	100	100	100	100	8
T2	100	99.7	99.9	100	5
ST3	99.7	99.9	99.8	99.9	7
MT3	99.3	100	99.6	99.9	9
WT3/T4	99.6	100	99.8	99.9	9

The time reported in the table is the time spent on prediction (or inference).

performance (F1-score) in the BCB dataset. It is worth noticing that all of the evaluated approaches perform well in detecting the Type-1 (T1) and Type-2 (T2) clones. This is expected as T1 indicates code copies without modification, and T2 includes code clones with generally identical syntax. On the contrary, only TAILOR is effective in detecting the rest of the clone types. Even for the most challenging weakly Type-3/Type-4 (WT3/T4) clones, TAILOR still achieves nearly a 1.0 F-score. However, the best existing solution (FA-AST with GNN) only gets a 0.946 F-score. This experiment demonstrates the advantage of GNNs in code clone detection and the necessity of designing GNNs tailored to learn code representations. Another interesting observation is that the tree-based and graph-based approaches perform much better than token-based approaches in moderately Type-3 (MT3) and WT3/T4 clones. This matches our understanding that token-based methods lack the knowledge of program structures, which can be captured by other methods in the form of ASTs or PDGs.

In Table IV, we also show TAILOR's detailed performance on the BCB dataset regarding Precision, Recall, and AUC. Upon closer investigation, we find that TAILOR gains very high AUC (>0.999) on every type of code clone pairs, demonstrating that our CPGNN excels at distinguishing program functionalities.

In terms of efficiency, we report the time cost of predicting code clones in Table II and Table IV. Results show that RtvNN, Code2Vec, Mocktail, and FCDetector spend the least time (<1s). The reason is that they directly input code fragments embedded as vectors into a binary classifier. In contrast, TAILOR performs additional embedding propagation and aggregation. Compared with ASTNN, TAILOR is ten times faster because ASTNN requires dynamic batch size adjustment during the prediction procedure.

C. RQ1: Performance on Source Code Classification

*1) **Settings**:* To evaluate how TAILOR performs on source code classification, we again use the OJClone dataset [4], containing 52,000 C programs from 104 programming tasks. As our goal is to classify code fragments by their functionalities, we treat programs that solve the same task belonging to the same class. In total, we collect 104 well-balanced classes from the OJClone dataset, each class containing 500 programs.

TABLE V: Results of Source Code Classification on OJClone.

Approach	Precision	Recall	F-score	Accuracy	Time(s)
Code2Vec	65.2	64.2	64.1	64.2	**<1**
Mocktail	85.2	85.5	85.1	85.5	4
ASTNN	98.0	97.9	97.9	97.9	15
InferCode	93.1	92.9	92.8	93.0	5
TAILOR	**98.4**	**98.3**	**98.3**	**98.3**	6

The time reported in the table is the time spent on prediction (or inference).

For fairness, we only compare TAILOR with existing solutions that have been evaluated [14], [15], [32] or claim to be useful [31] in classifying code functionalities.

*2) **Metrics**:* Considering that this problem is a balanced multi-class classification problem, we adopt Accuracy and macro-averages of Precision, Recall, and F-score as the evaluation metrics. Technically, a macro-average metric first computes the metric for each class independently and then reports the average. Take Macro-Precision as an example, we calculate its value as: $\textit{Macro-Precision} = \frac{1}{N}\sum_{i}^{N} \textit{Precision}_i$, where N is the number of classes, and $Precision_i$ denotes the Precision of the *i-th* class. Accuracy measures correctly classified programs against all ground-truth programs. For short, we call Macro-Precision as Precision, Macro-Recall as Recall, and Macro-Fscore as F-score henceforth.

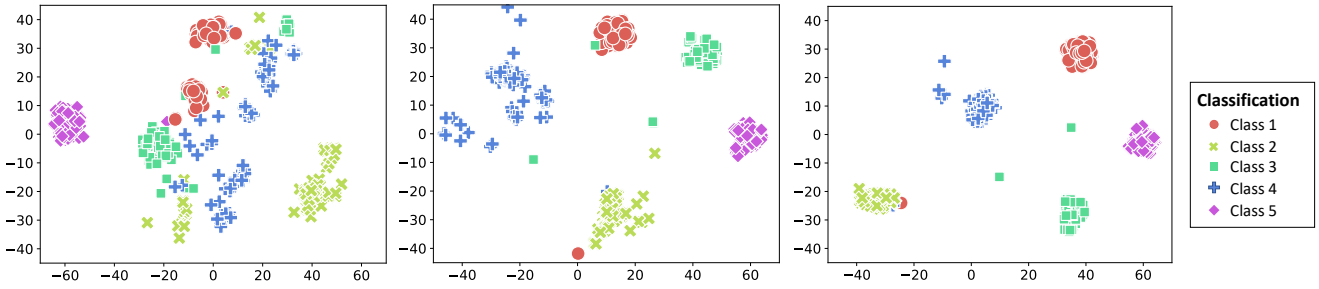
*3) **Results**:* Table V summarizes the results of TAILOR and the state-of-the-art approaches on source code classification. TAILOR achieves the highest Precision (98.4%), Recall (98.3%), F-score (98.3%), and Accuracy (98.3%). To understand the internals of our classification results, we visualize code representations of the first five classes in the OJClone dataset using the t-SNE [48] in Figure 5c. Clearly, the programs solving the same task are clustered together, while those solving different tasks are separated with clear boundaries. This visualization can be viewed as evidence that TAILOR successfully learns code representations beneficial for program functionality classification.

Similar to the code clone detection, we compare TAILOR's time cost with that of the state-of-the-art in Table V. TAILOR is nearly three times faster than ASTNN, while Code2Vec performs the best efficiency as it only needs to input the given code representations into a multi-class classifier.

D. RQ2: Comparison of Different GNNs

As discussed in Section IV-B2, off-the-shelf GNNs may introduce operations not necessarily helpful in modeling code functionalities. To investigate how different GNNs affect TAILOR's performance, we compare CPGNN with five popular GNN variants as follows:

- **CPGNN-NP** represents CPGNN with the embedding propagation disabled, which directly aggregates initial node embeddings to formulate program embeddings.
- **GGNN** [24] adopts the gated recurrent units (GRUs) to aggregate information from neighboring nodes.



(a) CPGNN-NP (b) GCN (c) CPGNN

Fig. 5: Classification of the first five programming tasks in OJClone, where each color represents a class. Best view in color.

- **GCN** [27] denotes the graph convolutional network, which aggregates node embeddings propagated from neighboring nodes with a weighted summation.
- **KGAT** [21] aggregates information propagated from neighbors with a weighted summation and a weighted product.
- **LightGCN** [33] is a lightweight GCN, which discards the feature transformation and nonlinear activation operations in the embedding aggregation.

Table VI reports the performances of all GNNs in code clone detection and source code classification on the OJClone dataset. CPGNN performs the best regarding all the evaluation metrics, while CPGNN-NP performs the worst. This matches our expectation as CPGNN-NP fails to inject CPG structures into code representation learning. LightGCN also does not perform well compared to the rest of the GNNs. The reason may be that LightGCN cannot infer the nonlinear relationships among CPG nodes. More importantly, it cannot distinguish the importance of information propagated from neighboring nodes due to the removal of feature transformation. Additionally, we find that GCN, GGNN, and KGAT underperform CPGNN, which validates our claim that directly adopting off-the-shelf GNNs may include unnecessary operations and even degrade performances. By achieving nearly 1.0 F-scores in both code clone detection and source code classification, our proposed CPGNN proves to be the most effective GNN in detecting similar code functionalities. Compared with the OJClone dataset, we observe a similar phenomenon on the BCB dataset. For example, CPGNN-NP also performs the worst on BCB (with an F1-score of 70.4% on WT3/T4) while CPGNN performs the best (with an F1-score of 99.8% on WT3/T4).

To gain further insight, similar to CPGNN, we visualize code representations produced by CPGNN-NP and GCN — which perform the worst and best among the GNN variants — in Figure 5a and Figure 5b, respectively. As can be seen, TAILOR clusters similar code fragments with clear boundaries. In particular, dissimilar code fragments are clustered far away from each other, and similar code fragments stay very close together. However, the clustering boundaries produced by CPGNN-NP and GCN are much less clear, even though GCN achieves reasonably high accuracy in source code classification. This result suggests the necessity of designing GNNs tailored to software engineering tasks.

We also would like to point out that considering the large codebases of modern software systems, improving the performance of code representation learning for similarity detection becomes increasingly important. For example, classifying code fragments can assist developers in understanding and maintaining large codebases. With this functionality, developers can (semi-)automatically identify commonly repetitive tasks in software systems and replace them with standard APIs [49]. As such, improving the precision of source code classification by even 1% could result in thousands of fewer false-positive programs to be manually investigated. This research has advanced the state-of-the-art in the area of source-level functional similarity detection. We hope that, through the study of TAILOR, we can re-establish the usage of GNNs in the code representation learning landscape.

TABLE VI: Effect of Different GNNs on OJClone.

Task	Metric	GNNs					
		CPGNN-NP	LightGCN	GCN	GGNN	KGAT	CPGNN
CCD	Precision	100	75.5	99.0	98.8	99.4	**100**
	Recall	0.0	7.2	98.4	97.6	92.9	**99.7**
	F-score	0.0	13.1	98.7	98.2	96.0	**99.9**
	AUC	82.5	89.8	99.9	99.9	99.9	**100**
	Time(s)	**81**	125	190	211	199	193
SCC	Precision	72.3	79.6	97.6	96.8	95.2	**98.4**
	Recall	72.4	79.8	97.6	96.7	95.2	**98.3**
	F-score	72.0	79.3	97.5	96.7	95.2	**98.3**
	Accuracy	72.0	79.6	97.6	96.7	95.2	**98.3**
	Time(s)	**13**	38	98	N/A	110	107

The time reported in the table is the time spent on training for one epoch.

Besides the effectiveness, we also compare the efficiency of GNN variants by measuring the training time per epoch using GPUs. Table VI shows that CPGNN-NP spends the least time by removing the embedding propagation. CPGNN is comparable to GCN and KGAT since they have similar time complexities for embedding propagation and aggregation. GGNN takes the longest time due to its complicated embedding aggregator (GRU). Note that by running the GRU, GGNN needs to store all the intermediate states of CPG nodes in the memory. However, since our GPU does not have enough memory to hold all the node states for source code classification, we have to run GGNN using CPUs instead of GPUs and thus report N/A in Table VI.

E. RQ3: Evaluating the Design of CPGNN

As the proposed CPGNN plays an important role in TAILOR, we investigate the impacts of its different design choices on TAILOR's performance (e.g., F-score). Specifically, we

TABLE VII: Effect of Embedding Initialization in Code Clone Detection (CCD) and Source Code Classification (SCC).

Task	Dataset	Metric	Embedding		
			Type	Token	Comb
CCD	OJClone	F-Score	99.6	99.6	**99.9**
CCD	BCB	F-Score	99.5	99.7	**99.8**
SCC	OJClone	Accuracy	97.7	97.8	**98.3**

TABLE VIII: Effect of Code Representations in CPGs.

Task	Dataset	Metric	Code Representations			
			A	A+C	A+D	A+C+D
CCD	OJClone	F-Score	99.4	99.8	99.8	**99.9**
CCD	BCB	F-Score	99.4	99.6	99.6	**99.8**
SCC	OJClone	Accuracy	97.9	98.1	98.0	**98.3**

empirically evaluate the effect of embedding initialization, CPG representations, and propagation iteration numbers.

*1) **Effect of Embedding Initialization**:* TAILOR initializes CPG nodes with a fusion of type and token embeddings. To understand their separate contributions, we conduct an ablation study by initializing CPG nodes with only types or tokens and evaluating the influences on code clone detection and source code classification. Table VII shows the results of different embedding initialization schemas, where *Type*, *Token*, and *Comb* specify the initialization with types, tokens, and a combination of types and tokens, respectively. We find that using only types or tokens consistently results in lower performance, which well validates our design of fusing type and token embeddings to initialize CPG nodes.

*2) **Effect of CPG Representations**:* TAILOR represents a CPG with an AST, a CFG, and a DFG. Here, we study the performances of alternative CPG representations, namely AST, AST and CFG, and AST and DFG. Experimental results are summarized in Table VIII, where *A*, *C*, *D* denote the AST, CFG, and DFG, respectively. TAILOR (i.e., *A+C+D*) yields the best performance. This is expected because it provides a comprehensive view of program characteristics, which suits the need to understand functionalities at different levels of code representations. Surprisingly, AST has also achieved very high F-scores on both code clone detection and source code classification. This phenomenon indicates that program syntactical structures significantly contribute to modeling program functionalities, which is also consistent with the findings in recent studies [6], [12], [14].

*3) **Effect of Propagation Iteration Numbers**:* CPGNN's propagation iteration number L decides how many hops of neighbors are integrated into code representations. Here, we change the iteration number, from one to five, to understand the advantage of the usage of multi-hop neighbors. We use TAILOR-1 to indicate the CPGNN with one propagation iteration and similar notations for others. Table IX summarizes the results with different numbers of propagation iterations. In general, increasing the number of propagation iterations (i.e., including neighborhood at higher hops) boosts TAILOR's performance. For example, TAILOR-4 and TAILOR-5 consistently achieve higher performances (i.e., F-score or Accuracy) than TAILOR-1, TAILOR-2, and TAILOR-3. This result quantitatively verifies that CPGNN can enhance its expressiveness in distinguishing code functionalities by incorporating higher-hop neighbors, which also confirms that CPGNN explicitly injects graph-structured program features into code representation learning. Moreover, we discover that the performance improvement becomes marginal by stacking one more layer over TAILOR-4. This discovery suggests that four-hop or five-hop neighbors around CPG nodes provide sufficient information for learning code representations.

TABLE IX: Effect of CPGNN Propagation Iteration Number.

Task	Dataset	Metric	CPGNN Layer				
			1	2	3	4	5
CCD	OJClone	F-Score	98.9	99.4	99.5	99.8	**99.9**
CCD	BCB	F-Score	99.5	99.7	99.6	**99.8**	99.8
SCC	OJClone	Accuracy	96.4	97.3	97.6	98.1	**98.3**

VI. THREATS TO VALIDITY

There are four main threats to the validity. First, we conduct experiments using only C and Java programs, where the results may not generalize to software systems implemented in other programming languages (PLs). That being said, TAILOR's design and implementation are not limited to a particular PL. For example, our CPG is built upon the AST, CFG, and DFG, all of which are general code abstractions. We leave it as future work to investigate TAILOR's effectiveness on different PLs.

Second, the quality of our experimental datasets (OJClone and BCB) cannot be guaranteed even though they are widely used in the literature [14], [15], [19], [25]. Specifically, the OJClone dataset may not be representative as it is not collected from a real production environment. Besides, due to the lack of ground truth, we assume that two programs in the OJClone are functionally similar if they solve the same programming task. However, this assumption may not hold in practice. To mitigate this threat, we also evaluate TAILOR using the BCB dataset that provides manually validated code clone pairs. Unfortunately, such ground truth can be biased based on the benchmark authors' intuition, potentially affecting our findings in the evaluation. In addition, OJClone and BCB datasets do not include obfuscated software systems where developers deliberately change code to conceal functionalities and bypass analysis. However, we note that software obfuscation is an open problem orthogonal to the study of TAILOR. One should refer to existing work [50], [51] for solutions. At last, whether TAILOR is effective to program functionalities (e.g., multithreading) beyond these datasets remains unknown, and we leave it as future work.

Third, hyper-parameters used in the CPGNN will affect TAILOR's performance. As a common practice, we apply a grid search to tune and find the hyper-parameters (e.g., the number of propagation iterations) that perform the best in validation sets. To reproduce our experimental results, we set the same random seeds, present all the hyper-parameter settings in Section V-A1 and publicly release our artifacts (including training logs).

Finally, if available, we leverage open-source implementations while reproducing the baseline approaches. However, as we have further fine-tuned their configurations (e.g., hyper-parameters) to achieve the best performances in our exper-

imental datasets, they may not be consistent with those in the original paper. We cannot reproduce FA-AST but cite the reported results due to missing details in its paper. Still, we believe this comparison is reasonable as we share the same experimental dataset. A supplementary comparison will be made when FA-AST is available to us.

VII. Related Work

Program Similarity. Measuring program similarity is a fundamental capability in software engineering with a broad spectrum of applications, such as bug detection [5], [52], [53], refactoring [54], [55], and code reuse [56]–[58]. DyCLINK [59] provides a dynamic approach that identifies similar code fragments based on their execution traces. However, due to the nature of the dynamic analysis, such an approach suffers from poor scalability and incomplete code coverage. Elsewhere in the literature, static approaches have also been widely used to quantify program similarity based on source code analysis [8], [10], [11], [14], [18], [19]. For example, SourcererCC [8] tokenizes code fragments and adopts a bag-of-tokens model (similar to bag-of-words models in information retrieval) to compare their similarities. Deckard [11] abstracts code fragments into syntax trees and leverages the locality-sensitive hashing to cluster similar ones. ASTNN [14] transforms source code into statement sequences and applies the recurrent neural network to detect functionally similar programs. To benefit from transfer learning, WySiWiM [60] exploits a pre-trained image classification neural network to learn visual representations of source code. By doing so, WySiWiM is computationally more efficient than traditional models that require training from scratch. Inspired by this, one potential approach to further speed up training Tailor is to leverage transfer learning by integrating self-supervised learning tasks. We note that software source code may not always be available due to security considerations or license restrictions. As such, researchers propose to measure program similarity based on binaries [61]–[66]. For example, Tracelet [67] and BinSequence [68] compare binary similarity by calculating edit distances among instruction sequences. Genius [69], Gemini [70], and DeepBinDiff [65] feed binaries into deep neural networks to learn their representations for similarity detection. To further boost the effectiveness, TREX [64] hires a hierarchical transformer model to encode execution semantics into binary representations. Additionally, the potential of intermediate representation (IR) has also been explored to compare program similarity [71]–[73]. For example, LLNiCad [71] performs NiCad [74] on the C-like IR to detect code clones.

Tailor detects program similarity at the source code level. It has achieved state-of-the-art performances in both code clone detection and source code classification. We attribute such advancement to the effective modeling of a code property graph using a tailored graph neural network.

Graph Neural Networks in Software Engineering. Graph neural networks (GNNs) excel at learning graph-structured representations [27], [39], [75]–[77]. As code features (e.g., control/data flow) are often represented as graphs, researchers have recently started to exploit GNNs in software engineering tasks, such as program comprehension [78], [79], code summarization [80]–[82], code clone detection [25], variable name prediction [23], vulnerabilities discovery [6], [83], and log level suggestion [84]. For example, Wang et al. [79] propose a graph internal neural network that learns program intervals (i.e., subgraphs that represent looping constructs) to facilitate method name prediction. LeClair et al. [80] adopt a convolutional GNN to generate natural language descriptions of source code. Allamanis et al. [23] utilize GGNN [24] for variable naming and misuse detection. Hoppity [83] leverages a GNN [85] as external memory to detect and fix bugs in Javascript programs.

To our best knowledge, FA-AST [25] is the first to deploy GNNs in code clone detection. While also using a GNN to compare program similarity, Tailor significantly differs from FA-AST in the design of the GNN: FA-AST directly employs off-the-shelf GNN models (e.g., GGNN [24]) to learn code representations. However, our evaluation shows that the recurrent neural network (specifically, the gated recurrent cell) in GGNN complicates a clone detector unnecessarily and even degrades the detection performance. We empirically find that it is of great importance to study the impacts of each operation in a GNN when applied for program analysis. Following this guidance, we design a CPG-based GNN with only operations beneficial for learning code features from a CPG.

VIII. Conclusion

In this paper, we present Tailor, a graph-neural-network-based approach tailored to detect functionally similar code fragments. Tailor summarizes program syntactic and semantic features into a code property graph (CPG). By propagating information over the CPG, Tailor explicitly exploits graph-structured program features to identify similar functionalities. We evaluate Tailor on two public datasets and compare it with nine state-of-the-art approaches. Experimental results show that Tailor outperforms existing solutions on both code clone detection and source code classification tasks.

Data Availability: All the artifacts are available online at https://anonymous.4open.science/r/ICSE_Tailor.

REFERENCES

[1] 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*, 2002.

[2] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, 2009.

[3] G. Frantzeskou, S. MacDonell, E. Stamatatos, and S. Gritzalis, "Examining the significance of high-level programming features in source code author classification," *Journal of Systems and Software*, 2008.

[4] 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 (AAAI)*, 2016.

[5] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, 2007.

[6] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, 2019.

[7] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, "Learning program semantics with code representations: An empirical study," in *SANER*, 2022.

[8] 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)*, 2016.

[9] 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)*, 2018.

[10] 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 (ESEC/FSE)*, 2021.

[11] 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)*. IEEE, 2007.

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

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

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

[15] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021.

[16] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2006.

[17] M. Wang, P. Wang, and Y. Xu, "Ccsharp: An efficient three-phase code clone detector using modified pdgs," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017.

[18] 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 (ESEC/FSE)*, 2018.

[19] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[20] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference (WWW)*, 2019.

[21] X. Wang, X. He, Y. Cao, M. Liu, and T.-S. Chua, "Kgat: Knowledge graph attention network for recommendation," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2019.

[22] J. Zeng, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "Shadewatcher: Recommendation-guided cyber threat analysis using system audit records," in *IEEE Symposium on Security and Privacy*, 2022.

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

[24] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," in *International Conference on Learning Representations (ICLR)*, 2016.

[25] 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)*. IEEE, 2020.

[26] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, 2020.

[27] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[28] 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 (S&P)*. IEEE, 2014.

[29] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, 1986.

[30] 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 (ICSME)*. IEEE, 2014.

[31] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, no. POPL, 2019.

[32] D. Vagavolu, K. C. Swarna, and S. Chimalakonda, "A mocktail of source code representations," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1296–1300.

[33] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang, "Lightgcn: Simplifying and powering graph convolution network for recommendation," in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval (SIGIR)*, 2020.

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

[35] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[36] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance*. IEEE, 1998.

[37] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[38] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[39] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, 2017.

[40] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *stat*, 2017.

[41] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, 2014.

[42] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *Computer Science*, 2013.

[43] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI)*, 2016.

[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[45] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics (AISTATS)*. JMLR Workshop and Conference Proceedings, 2010.

[46] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *USENIX Security Symposium*, 2022.

[47] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[48] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[49] D. Kawrykow and M. P. Robillard, "Improving api usage through automatic detection of redundant code," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2009.

[50] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications." in *NDSS*, 2014.

[51] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: toward extracting hidden code from packed android applications," in *European Symposium on Research in Computer Security*. Springer, 2015.

[52] M. Ahmadi, R. M. Farkhani, R. Williams, and L. Lu, "Finding bugs using your own code: detecting functionally-similar yet inconsistent code," in *30th USENIX Security Symposium (USENIX Security)*, 2021.

[53] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *2008 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2008.

[54] M. F. Zibran and C. K. Roy, "Towards flexible code clone detection, management, and refactoring in ide," in *Proceedings of the 5th International Workshop on Software Clones (IWSC)*, 2011.

[55] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017.

[56] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005.

[57] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[58] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," *Journal of Systems and Software*, 2018.

[59] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: detecting similarly behaving software," in *Proceedings of the the 24th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, 2016.

[60] P. Keller, A. K. Kaboré, L. Plein, J. Klein, Y. Le Traon, and T. F. Bissyandé, "What you see is what it means! semantic representation learning of code based on visualization and transfer learning," 2021.

[61] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017.

[62] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017.

[63] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[64] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020.

[65] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[66] X. Li, Y. Qu, and H. Yin, "Palmtree: learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[67] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm Sigplan Notices*, 2014.

[68] H. Huang, A. M. Youssef, and M. Debbabi, "Binsequence: Fast, accurate and scalable binary code reuse detection," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[69] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[70] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[71] P. M. Caldeira, K. Sakamoto, H. Washizaki, Y. Fukazawa, and T. Shimada, "Improving syntactical clone detection methods through the use of an intermediate representation," in *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE, 2020.

[72] O. Kononenko, C. Zhang, and M. W. Godfrey, "Compiling clones: What happens?" in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014.

[73] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[74] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*. IEEE, 2011.

[75] J. Li, Y. Rong, H. Cheng, H. Meng, W. Huang, and J. Huang, "Semi-supervised graph classification: A hierarchical graph perspective," in *The World Wide Web Conference (WWW)*, 2019.

[76] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua, "Neural graph collaborative filtering," in *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval (SIGIR)*, 2019.

[77] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations (ICLR)*, 2019.

[78] S. Liu, "A unified framework to learn program semantics with graph neural networks," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020.

[79] Y. Wang, K. Wang, F. Gao, and L. Wang, "Learning semantic program embeddings with graph interval neural network," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.

[80] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension (ICPC)*, 2020.

[81] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," in *International Conference on Learning Representations (ICLR)*, 2020.

[82] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *International Conference on Learning Representations (ICLR)*, 2019.

[83] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[84] J. Liu, J. Zeng, X. Wang, K. Ji, and Z. Liang, "Tell: Log level suggestions via modeling multi-level code block information," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[85] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, 2008.