Toward Interpretable Graph Tensor Convolution Neural Network for Code Semantics Embedding

JIA YANG and CAI FU, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, China
FENGYANG DENG and MING WEN, Huazhong University of Science and Technology, China
XIAOWEI GUO and CHUANHAO WAN, Huazhong University of Science and Technology

Intelligent deep learning-based models have made significant progress for automated source code semantics embedding, and current research works mainly leverage natural language-based methods and graph-based methods. However, natural language-based methods do not capture the rich semantic structural information of source code, and graph-based methods do not utilize rich distant information of source code due to the high cost of message-passing steps.

In this article, we propose a novel interpretable model, called graph tensor convolution neural network (GTCN), to generate accurate code embedding, which is capable of comprehensively capturing the distant information of code sequences and rich code semantics structural information. First, we propose to utilize a high-dimensional tensor to integrate various heterogeneous code graphs with node sequence features, such as control flow, data flow. Second, inspired by the current advantages of graph-based deep learning and efficient tensor computations, we propose a novel interpretable graph tensor convolution neural network for learning accurate code semantic embedding from the code graph tensor. Finally, we evaluate three popular applications on the GTCN model: variable misuse detection, source code prediction, and vulnerability detection. Compared with current state-of-the-art methods, our model achieves higher scores with respect to the top-1 accuracy while costing less training time.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering → Software safety**; **Feature interaction**; **Software verification and validation**; **Language features;**

Additional Key Words and Phrases: Tensor computation, code embedding, graph neural network

Cai Fu is supported by China NSF (62072200), Jia Yang is supported by China NSF (62202146).

Authors' addresses: J. Yang and C. Fu (corresponding author), Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan, Hubei Province, 430074, China; emails: jia yang@hbut.edu.cn, fucai@hust.edu.cn; F. Deng, M. Wen, X. Guo, and C. Wan, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan, Hubei Province, 430074, China; emails: fengyang_deng@hust.edu.cn, mwenaa@hust.edu.cn, d202280650@hust.edu.cn, wanchuanhao@hust.edu.cn.

1049-331X/2023/07-ART115 $15.00
https://doi.org/10.1145/3582574

ACM Reference format:
Jia Yang, Cai Fu, Fengyang Deng, Ming Wen, Xiaowei Guo, and Chuanhao Wan. 2023. Toward Interpretable Graph Tensor Convolution Neural Network for Code Semantics Embedding. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 115 (July 2023), 40 pages.
https://doi.org/10.1145/3582574

1 INTRODUCTION

Source code semantics embedding is used to understand the behavior of the program and represent latent semantic properties of the program [51, 52], which is the most common and useful functionality in Integrated Development Environments [38]. Code embedding technologies can be used to speed up software engineering processes, such as code recommendation [62], bug repair [73], and code summarization [20]. We combine the sequence information of code tokens and the structure information of various heterogeneous code graphs to achieve accurate code embedding. Then, we perform three specific popular applications using the learned code embedding: source code prediction, variable misuse detection, and vulnerability detection. However, some problems of code embedding need to be solved: inaccurate text-based and token-based code embedding, large program structure, the long-range dependencies of codes, and scalable code corpus.

Existing works mainly performed source code embedding from two parts. First, some existing works applied various language models, such as various n-gram models [28, 30, 46], which focused on the repeatability of source code to obtain code embedding. Some methods treated source code as token sequences and input them into the standard language models [9, 25, 52, 58, 64, 69]. However, treating codes as text can not capture the structured semantic information of source codes [37]. Besides, the code token sequences that are only extracted from AST are much longer than normal natural language sentences. They failed to consider the long-range dependencies of codes that are induced by the same token at distant locations. These methods also need to train a large code corpus to ensure the accuracy of code embedding, which may cause efficiency problems. Second, some existing works used the graph-based learning model to obtain code embedding [49, 51, 68]. GINN [74] generalized from a curated graph neural network using a hierarchy of intervals for scaling the learning process to large graphs. However, these methods miss out on rich distant information and node order information of source code sequences, and they lose some context information when each code token is associated with a single representation.

In this article, we propose a toward interpretable **graph tensor convolution neural network (GTCN)** to generate accurate code semantic embedding. The critical design of GTCN is to combine tensor circulate computation and graph convolution neural network for learning high-dimensional code features. Our model focuses on various code contexts' construction and ultimately captures the essence of precise code semantics.

First, we extract the syntax information and hierarchical structural information from the source code by encoding the program as various graphs with token features. Different code graphs represent different code semantic information, such as control flow, data flow, operation order, and operand values. We construct a high-dimensional code graph tensor using various heterogeneous code graphs. Second, a key innovation of our model is to design the tensor-based graph neural network, which combines graph convolution computation and tensor circulation computation. We input the code graph tensor in this model to achieve code embedding, which obtains fine code features via leveraging the traditional graph neural network and tensor computation. There are three key advantages of using tensor technology:

- Tensor as a high-dimensional data structure [41] can be used to represent the various properties of code, which can combine the space information of different code semantic graphs and the sequential information of code token sequences.

- Tensor computation can better learn the data correlation of source code from three dimensions [63]. The tensor computation not only can learn the row data relationship and the column data relationship, but also can learn the third-dimensional data relationship.
- The deep learning algorithm with tensor computation can alleviate the overfitting problem to a certain extent [76, 79]. Since the tensor product is a bilinear form, they use second-degree polynomial interactions to improve the expressiveness of the model, and the standard matrix dot product only uses first-degree polynomial interactions. The second-degree polynomial interactions can learn more code feature correlation than the first-degree polynomial interactions in the deep learning algorithm [70].

Finally, to evaluate the potential of the proposed GTCN model in three popular code tasks—**variable misuse detection**, **source code prediction**, and **vulnerability detection**—we design the composited output layer that combines the candidate-based attention output layer and vocabulary-based output layer. The composited output layer can solve the **out-of-vocabulary (OoV)** problem to a certain extent by expanding the vocabulary using local candidates. Variable misuse detection aims to detect whether the variable name is correct at the given location. Source code prediction intelligently predicts a source code token given its left code contexts (that is, derivable from preceding tokens). Our model encodes the leftward code contexts and itself to obtain the representation of each code node. Vulnerability detection focuses on accurately detecting vulnerable source codes. Our model achieves better accuracy for both three tasks and needs less training regime than current state-of-the-art methods.

Contributions are summarized as follows:

(1) We build a code graph tensor to represent various heterogeneous code graphs. We use AST as the backbone and then explicitly encode the data dependency, program control, and code token sequences into a code graph tensor. Specifically, we extract four code graphs: AST, **control flow graph (CFG), dataflow dependency graph (DDG)**, and **natural code sequence (NCS)**, and construct these four graphs as a code graph tensor.

AST can represent all syntax information between code nodes, while there may be a long distance between two data nodes with relevant semantic information in AST. Two data nodes in DDG can be directly connected to reduce the distance between correlated long-distance nodes further. CFG represents control information and jumps information, this information can not be well represented in AST. NCS retains the sequence order information between nodes. These four graphs can fully represent the semantic information of the code, which can be used to learn accurate code semantics.

(2) We propose a toward interpretable graph tensor convolution neural network. It is the first work of combining the toward interpretable tensor computation and graph convolution neural network to generate code embedding as far as we know, which can hierarchically capture the higher-dimensional linear relationships from the code graph tensor.

(3) We evaluate our model on the C# dataset. Our model achieves 94.6% top-1 recall and 97.1% F1 on the variable misuse detection task, beating other methods. Results show that our model achieves about 18% higher top-1 recall than other methods, on average.

We evaluate our model on the Python dataset, showing that our best model achieves 76.1% top-1 recall and 74.9% F1 on the source code prediction task. Results show that our model, on average, achieves about 13% higher top-1 recall than other methods.

We evaluate our model on the C dataset, showing that our best model achieves 93.8% top-1 recall on the vulnerability detection task. Results show that our model achieves about 6% higher top-1 recall than other methods at least.

2 PRELIMINARY

We first introduce some notations that are used in our model. The fraktur letter denotes the tensor, the uppercase bold letter denotes the matrix, and the lowercase bold letter denotes the vector. $\mathbf{X}_v \in \mathbb{R}^{d\times 4}$ represents the initial node feature matrix of the node v. $\mathbf{H}_v \in \mathbb{R}^{m\times 4}$ represents the node embedding matrix of the node v in the hidden layer, and $m > d$. $\mathbf{A}^+$ represents the transpose of adjacent matrix $\mathbf{A}$. $\mathcal{H}^{(i)}$ denotes the ith frontal slice matrix of tensor $\mathcal{H}$, and $\mathcal{H}^{(i)} = \mathcal{H}(:,:,i)$. $\mathcal{H}(i,:,:)$ denotes the ith horizontal slice matrix of tensor $\mathcal{H}$. Then, we briefly introduce tensor computation and **graph convolution neural network (GCN)** and discuss the tensor advantage.

2.1 Tensor Computation

Tensor transpose. Let the size of $n_2\times n_3\times n_1$ tensor $\mathcal{T}^\dagger$ denote the conjugate transpose of a tensor $\mathcal{T} \in \mathbb{R}^{n_1\times n_2\times n_3}$. We obtain $\mathcal{T}^\dagger$ by transposing each frontal slice $\mathcal{T}^{(k)} \in \mathbb{R}^{n_1\times n_2}$ and then make the order of transposed frontal slices from 2 to n_3 reversed [41].

Tensor-matrix product [18]. Let tensor $\mathcal{X}$ be $n_1 \times n_2 \times n_3$ and matrix $\mathbf{U}$ be $n_4 \times n_2$, then the tensor-matrix product of $\mathcal{X} \times_3 \mathbf{U}$ is the tensor of size $n_1 \times n_4 \times n_3$:

$$\mathcal{X} \times_3 \mathbf{U}(i_1, j_4, i_3) = \sum_{i_2=1}^{n_2} \mathcal{X}(i_1, i_2, i_3)\mathbf{U}(j_4, i_2). \tag{1}$$

Tensor product. If $\mathcal{X} \in \mathbb{R}^{n_1\times n_2\times n_3}$ and $\mathcal{Y} \in \mathbb{R}^{n_2\times n_4\times n_3}$, then the tensor product $\mathcal{X}*\mathcal{Y}$ is a $n_1\times n_4\times n_3$ tensor, having

$$\mathcal{X} * \mathcal{Y} = \text{fold}(\text{Circ}(\mathcal{X}) \cdot \text{Matvec}(\mathcal{Y})), \tag{2}$$

where Matvec($\mathcal{Y}$) takes the tensor $\mathcal{Y}$ into a block $n_2n_3 \times n_4$ matrix,

$$\text{Matvec}(\mathcal{Y}) = \begin{bmatrix} \mathcal{Y}^{(1)} \\ \mathcal{Y}^{(2)} \\ \vdots \\ \mathcal{Y}^{(n_3)} \end{bmatrix}. \tag{3}$$

And the operator fold($\cdot$) makes the matrix into the tensor. The Circ($\mathcal{X}$) unfolds the tensor $\mathcal{X}$ as a circulant matrix.

$$\text{Circ}(\mathcal{X}) = \begin{bmatrix} \mathcal{X}^{(1)} & \mathcal{X}^{(n_3)} & \mathcal{X}^{(n_3-1)} & \dots & \mathcal{X}^{(2)} \\ \mathcal{X}^{(2)} & \mathcal{X}^{(1)} & \mathcal{X}^{(n_3)} & \dots & \mathcal{X}^{(3)} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \mathcal{X}^{(n_3)} & \mathcal{X}^{(n_3-1)} & \dots & \mathcal{X}^{(2)} & \mathcal{X}^{(1)} \end{bmatrix}. \tag{4}$$

Graph Convolution Network (GCN). The layer-wise propagation rule of the GCN [43]:

$$H_{l+1} = f_{GCN}(A,\ H_l) = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H_lW_l), \tag{5}$$

where $\tilde{A} = A + I_N$ is the adjacency matrix of the graph with added self-connections. I_N is the identity matrix, $\tilde{D}_{ii} = \sum_i \tilde{A}_{ij}$, and W_l is a layer-specific trainable weight matrix. $\sigma(\cdot)$ denotes an activation function. H_{l+1} is the matrix of node embedding at the l + 1-th layer, and $H_0 = X$, X is the matrix of initial node features. The form of this propagation rule (Equation (5)) can be motivated via the first-order approximation of localized spectral filters on graphs. Equation (5) uses the eigenvalues and eigenvectors of the Laplacian matrix of the graph to analyze the properties of graphs via spectral graph theory.

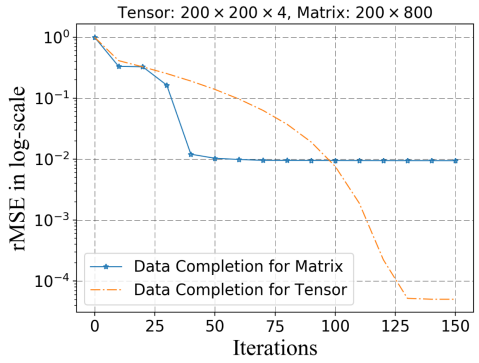

Fig. 1. Results for matrix completion and tensor completion.

The GCN model can relax certain assumptions typically made in graph-based semi-supervised learning by conditioning both on the data X and on the adjacency matrix A of the underlying graph structure. The GCN model is especially powerful in scenarios where the adjacency matrix contains information not present in the data X.

2.2 Tensor Advantage

The tensor structure is multilinear and can handle more data correlation [31, 36]. We add a case study to show the reason that we can learn more data correlations from the code graph tensor.

We assume that there are four graph matrices with the size $n \times n$, which are generated randomly. We construct these four random graph matrices as a 3D graph tensor of size $n \times n \times 4$ and a matrix of size $n \times 4n$. Then, we perform similar data completion methods using the matrix or the tensor as input. Data completion can be used to show the ability to learn data correlation [33, 36]. Tensor completion is proposed for filling the missing elements of high-dimensional partially observed tensor data, and matrix completion is proposed for filling the missing elements of the partially observed matrix [36].

We set $n = 200$, and four graph matrices are of size 200×200, then we construct a three-order tensor of size $200 \times 200 \times 4$ using these four graph matrices, as well as construct a large matrix of size 200×800 using the same four graph matrices. The tensor and matrix have the same data elements. The sample rate is set as 0.3.

We use the Frank-Wolfe iteration algorithm to apply data completion. The Frank-Wolfe algorithm is a popular conditional gradient algorithm, commonly used in machine learning [32]. For tensor completion, we use the approximate Frank-Wolfe version of Reference [24]. For matrix completion, we use the Frank-Wolfe algorithm with a linear sub-problem as References [21, 32].

We compare the recovery errors when models are converged. Results are shown in Figure 1, the recovery error of tensor completion approximates 10^{-4}, and the recovery error of matrix completion is 10^{-2} when they are converged.

Given the same data elements and the same sample rate, we can see that the data completion based on the tensor structure performs better than the data completion based on a matrix structure. Compared with a large recombined graph matrix, the tensor structure keeps every original graph information, rather than rudely connecting four matrices. The tensor structure is multilinear and can keep more data correlation. For example, if some semantic information of the code node can not be obtained by CFG, then it can learn from the AST, DDG, and NCS. The node connectivity in other graphs also can reflect in this graph. In the tensor structure, they are influenced by each

other, and these four graph matrices are complementary to each other. The matrix only can learn the row data relationship or the column data relationship, but the tensor can learn not only the row data relationship and the column data relationship, but also the third-dimension data relationship. Therefore, the tensor can better learn data correlation and handle data structure, and our model can actively and effectively learn high-dimensional code features via the tensor.

The tensor-based algorithm can alleviate the overfitting problem in deep learning to some degree [15, 53, 76, 79]. In the deep learning models, the overfitting problem is a serious problem. If the complexity of the predictive function formed by the training model to predict the outcome, is much higher than the problem in real world. In this condition, it leads to overfitting problems [26]. The high-dimensional data features, complex hypothetical model, a large of model parameters, and small training samples, all lead to the overfitting problem of the training model, which decreases the generalization ability of the training model [26, 59].

As we know, the model parameters are learned from the training samples. For example, for a quadratic classifier, user classes must be modeled by a set of subclasses, and the mean vector and covariance matrix of each subclass are the parameters that must be estimated from training samples. If the size of model parameters is much larger than the number of training samples, then this model will have overfitting problems [29]. For example, for the common quadratic maximum-likelihood estimator, when the dimensionality of data exceeds the number of training samples, the maximum-likelihood covariance estimate is singular and cannot be used, even in cases where the number of training samples is only two or three times the number of dimensions, estimation error can be a significant problem [44].

The number of training samples required to train a classifier for high-dimensional data is much greater than that required for conventional data. In practice, most models need to vectorize the high-dimensional data and input it into the model. The vectorization of high-dimensional data is to connect vectors of each row or column in the high-dimensional data end to end. This operation will generate a very long vector. Vectorizing the high-dimensional data makes the size of data features larger, and thus we need to learn a model with more parameters to fit a large of data features. If the training samples are insufficient, then it will cause the overfitting problem of the training model. Specifically for our variable misuse detection and vulnerability detection tasks, it is difficult to collect a large of training samples in the real world, and we need to design a model with better generalization ability to solve the overfitting problem.

Mathematically, if the model has a small model complexity penalty and a small train error, then it can guarantee that its test error is also very small [17]. The overfitting problem is related to model complexity. Therefore, to improve the generalization ability of the model, we need to control the training error and the model complexity penalty to be relatively small [17]. The model complexity penalty is $\sqrt{\frac{h(\log(2N/h)+1)-\log(\eta/4)}{N}}$ [34], where N represents the number of train samples, h is the VC dimension of the model, VC dimension represents the assumed freedom degree, which is the assumed number of feature vectors.[1] The η represents a probability value ($0 \leq \eta \leq 1$), and $1 - \eta$ represents the probability that test error is small than a value; this value is calculated by the training error adding the model complexity penalty. By observing the model complexity penalty, we can see that both the large h and the large N can decrease the model complexity penalty.

The tensor-based model can decrease the model complexity penalty via decreasing the VC dimension h (freedom degree) [40]. In recent years, tensors as a kind of high-dimensional data structure have been widely adopted to analyze multilinear relations, such as hyperspectral images [10], deep learning [54], and video processing [63]. The property of tensor is "good

[1] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_dimension.

enough" to be linear, so we can use it to deal with almost all linear problems, which is the main value of tensor structure [71]. The tensor does not depend on a specific reference frame or coordinate system. A tensor can be used to represent high-dimensional data. The tensor structure can keep the original data structure, such as spatial position information and internal information between features in the original data.

The tensor-based model supports inputting high-dimensional data into the model directly. The most significant advantage of the tensor-based model is that it projects the data into three subspaces at least, namely, the row subspace, the column subspace, and the third-order subspace. The freedom degree of input data will be significantly reduced [79]. **The tensor-based model reduces model complexity to make interpretable model space smaller, which improves the model interpretability. We can better capture the data features from the simpler model space.**

Moreover, the tensor computation can reduce the number of variables in the optimization problem, which can decrease the overfitting phenomenon in the learning process [15]. The tensor-based model computes fewer parameters than a vector-based model [35]. For example, if we unfold a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ as a vector, then the model needs to calculate $I_1 \times I_2 \times I_3$ parameters. If we use the multilinear subspace algorithm of tensor, then the model only needs to calculate $I_1 + I_2 + I_3$ parameters [35].

3 METHODOLOGY AND MODEL

Code is a kind of special language with important structural features. In other words, code can be represented as specific graph data with node text information. For code data, we need to effectively extract the text sequence information of code nodes and structural information of code graphs to obtain accurate code semantic features.

As we all know, GCN as an efficient variant of convolution neural networks is widely used in graph-based deep learning applications. GCN scales linearly in graph edges and learns node representations in the hidden layer. The convolution box not only learns the node features in a graph, but also extracts the graph features. GCN can encode both local graph structure and node features, which is widely used in node classification, graph classification, link prediction, and graph embedding [43].

Motivating by the tensor advantage (see Section 2.2) and GCN, as well as considering the generality and capacity of the model, we combine tensor computation and GCN as our basic learning framework. A natural idea is to abstract programs into high-dimensional graph representations for a tensor-based and graph-based neural network model to learn. We propose a new toward interpretable **graph tensor convolution neural network model (GTCN)**. As shown in Figure 2, our scheme follows three steps:

(1) **Graph Tensor of Composite Code Semantics.** Our model encodes the raw source code of a function into a code graph tensor with comprehensive program semantics. We leverage two pieces of information from the source code: code sequence text information and code semantic structure information. Our model uses four code graphs to represent these two pieces of information: **Abstract Syntax Tree (AST), Control Flow Graph (CFG), Dataflow Dependency Graph (DDG)**, and **Natural Code Sequence (NCS)**. We give specific discussions of these four graphs in Section 3.1.1. We expose these structured data as a tensor input into a tensor-based graph neural network model.

(2) **Graph Tensor Convolution Layer.** This graph tensor convolution module combines tensor circulate computation and graph convolution operation, which learns the node embedding of the hidden layer from the code graph tensor.

(3) **Candidate-based Attention or Vocabulary-based Output Layer.** We use the candidate-based attention output layer to achieve variable misuse detection and use the composited

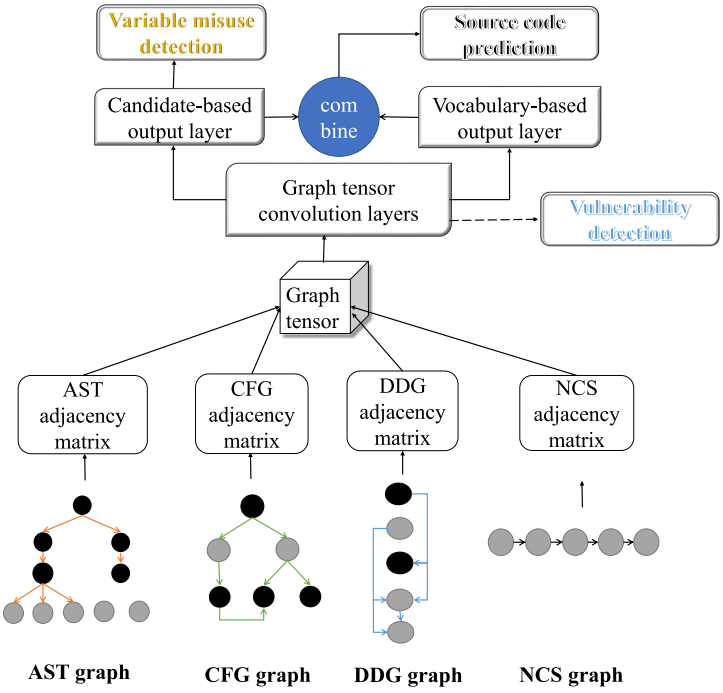

Fig. 2. Overview of our model.

output layer to achieve source code prediction. The composited output layer combines candidate-based attention and vocabulary-based output layer, which can solve the OoV problem to some degree. We use a **multi-layer perceptron model (MLP)** as the classification layer to perform vulnerability detection through a sigmoid function.

3.1 Methodology

Note that program codes are specific languages with various structural information. Compilers normally use multiple graphs to represent complex code relationships. Our model wants to handle directly various enhanced code graphs. These graphs contain both code text and semantic structure.

3.1.1 Code Graph Representation. Normally, **there are three common code contexts [52, 57, 72]**: (1) Enclosing context, such as class name in codes. (2) Implementation context represents the implemented process of a method and includes the used variable names, accessed field names, invoked method names, and so on. (3) Interface context represents the input or output of the method, such as the class of passed parameters, the class of return value, and so on. These contexts contain different semantic structure information between tokens. To better represent different code contexts, we extract various graphs that represent these contexts. We give a simulated presentation for the information propagation process of code nodes in the code graph tensor.

Specifically, we use the following four graphs to represent these contexts; these four graphs are proven effectively learning code embedding for vulnerability detection [77, 81]. **Abstract Syntax Tree (AST)**. AST is the backbone of a program graph, which is represented as an abstract syntax structure tree of source codes. **Control Flow Graph (CFG)**. CFG is an abstract representation of programs, which represents all the execution paths that a program might traverse. **Data Flow Dependency Graph (DDG)**. DDG represents the data dependency between instructions in the simplest form. **Natural Code Sequence (NCS)**. NCS represents the natural sequence order of

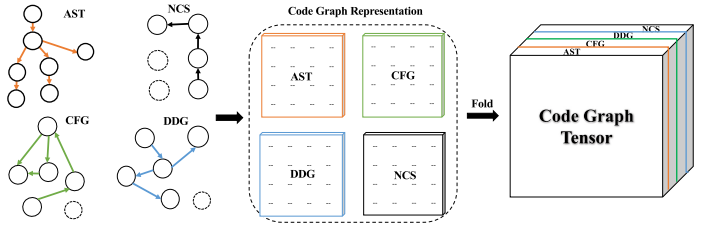

Fig. 3. Code graph tensor.

syntax children nodes in AST, which can not be represented by a normal AST edge. The graph of the natural code sequence adds edges (NextToken edges) between each syntax token and the successor of this token.

We use these four heterogeneous code graphs to represent code contexts fully. AST can represent the enclosing context. CFG mainly represents the implementation context. DDG mainly represents the interface context. NCS supplements the natural sequence order of tokens in source codes [81]. We need to design a data structure to integrate these four heterogeneous graphs and input them into the model to achieve code embedding. In our model, the node connectivity of every relation graph (e.g., AST, CFG, DDG, NCS) is represented as an adjacency matrix with node features.

3.1.2 Code Graph Tensor Construction. Motivating by tensor advantage, we consider constructing a code graph tensor to represent the richer code relations. Code graph tensor is utilized to manifest deeper semantics and high-dimensional code structure behind the textual code, and we focus on abstract syntax trees, control flow graphs, dataflow dependence graphs, and natural code sequences in a high-dimensional joint data structure, which captures the syntactic and semantic relationships among the different code tokens.

As shown in Figure 3, each adjacency matrix with the initial node features is a frontal slice of tensor, which denotes the corresponding representation of a kind of connection type. These graphs share the same node identity number and initial node embedding. We use all AST nodes to construct the node set, since nodes in other graphs are included in AST nodes. Therefore, a function c_i can be represented by a code graph tensor $\mathcal{A}$. These four graph representations share the same node set, V.

Our approach highlights that code semantic features can be adequately learned using a novel representation of source codes, which is denoted as the code graph tensor. We show an example of code fragment for "integer additive computation" function in Figure 4. The left part is the source codes of this function written by C programming language, and the right part is the corresponding joint graph representation for showing the code semantics. The relationships between nodes in this joint graph representation can be represented in the proposed code graph tensor. We can see that the propagation of node information is from three parts: one is to aggregate information from neighbors of each node within a graph, another is to exchange node information between different graphs in the tensor, and the node also can learn features from node-self.

For example, if some semantic information of the code node can not be obtained by CFG, then it can learn from the AST, DDG, and NCS. The node connectivity in other graphs also can reflect in this graph. In the tensor structure, they are influenced by each other, and these four graph matrices are complementary to each other. The matrix only can learn the row data relationship or the column data relationship, but the tensor can learn not only the row data relationship and the column data relationship, but also the third-dimension data relationship. Therefore, a tensor can better learn data correlation and handle data structure. A tensor can be used to spread and assemble code information across different token-relation graphs.

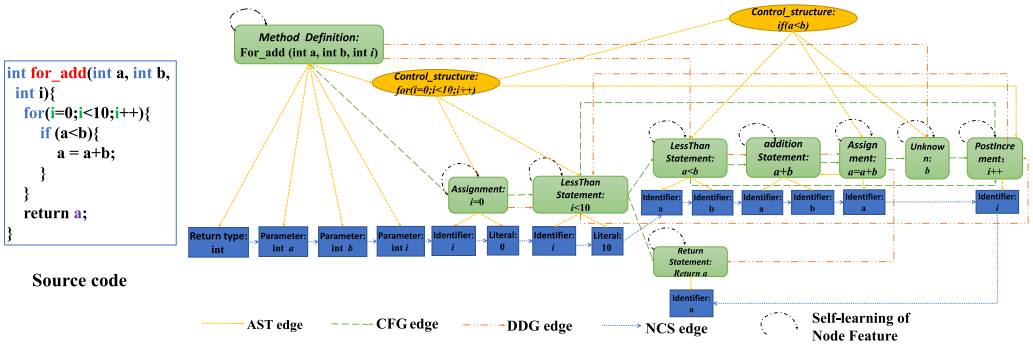

Fig. 4. Joint graph representation in code graph tensor.

3.2 Model

We propose a graph tensor convolution neural network to learn code node features from the proposed code graph tensor.

3.2.1 Initial Node Embedding Generation. First, we generate initial node embedding for each node. We combine the information from the value and type of code token to compute the initial code token embedding. Motivated by the word2vec model [22], we use this algorithm to encode its initial node embedding $\mathbf{x}_v \in \mathbb{R}^d$, where d is the feature dimension of the initial node feature vector. We copy four initial node feature vectors and form a matrix. We obtain the initial node representation $\mathbf{X}_v \in \mathbb{R}^{d\times 4}$ and then pad it with zeros to match the size of the hidden layer (that is, m).

3.2.2 Graph Tensor Convolution Neural Network. We design a novel graph tensor convolution layer. Specifically, we input the code graph tensor with initial node embedding into this layer, which can directly learn on the code graph tensor in a straightforward but effective way.

First, we define the node feature tensor $\mathcal{H}_l \in \mathbb{R}^{n\times m\times 4}$ in the hidden layer, m denotes feature dimension of node in the lth layer, and n denotes the number of nodes. The frontal slice $\mathcal{H}_l^{(i)} \in \mathbb{R}^{n\times m}$ is the node feature matrix of the ith graph in the code graph tensor at the lth layer. We assume that the initial node feature tensor $\mathcal{H}_0 = \mathcal{X}$.

We propose a novel graph tensor convolution algorithm to compute node embedding in each hidden layer. In our code graph tensor, different frontal slices represent different graphs, and they are AST, CFG, DDG, and NCS, respectively. The graph tensor convolution layer can harmonize heterogeneous information and structures of different graphs during the learning process, which can directly perform convolution learning on the code graph tensor.

In each layer, we apply the tensor circulate computation on the code graph tensor $\mathcal{A}$ and the node feature tensor $\mathcal{H}_l$ of the current hidden layer and then we propose a GTCN layer to compute the node feature tensor $\mathcal{H}_{l+1}$ of the next hidden layer based on standard GCN [43]. The model calculates the weight parameters. As shown in Figure 5, the process is formulated as follows:

$$\begin{aligned}\mathcal{H}_{l+1} &= f_{GTCN}(\mathcal{A}, \mathcal{H}_l) = \sigma(\widetilde{\mathcal{A}} * \mathcal{H}_l \times_3 \mathbf{W}_l) \\ &= \sigma(\text{fold}(\text{Circ}(\widetilde{\mathcal{A}}) \cdot \text{Matvec}(\mathcal{H}_l)) \times_3 \mathbf{W}_l).\end{aligned} \tag{6}$$

For each frontal slice of $\widetilde{\mathcal{A}}$, $\widetilde{\mathcal{A}}(:,:,i) = \widetilde{\mathcal{D}}(:,:,i)^{-\frac{1}{2}}\mathcal{A}'(:,:,i)\widetilde{\mathcal{D}}(:,:,i)^{-\frac{1}{2}}$. $\mathcal{A}'$ is the code graph tensor with added self-connections, and $\mathcal{A}' = \mathcal{A} + \mathcal{I}$. $\mathcal{I} \in \mathbb{R}^{n\times n\times 4}$ is the identity tensor. The first frontal slice of $\mathcal{I}$ is an identity matrix of size $n\times n$, and the other frontal slices are all zero matrices.

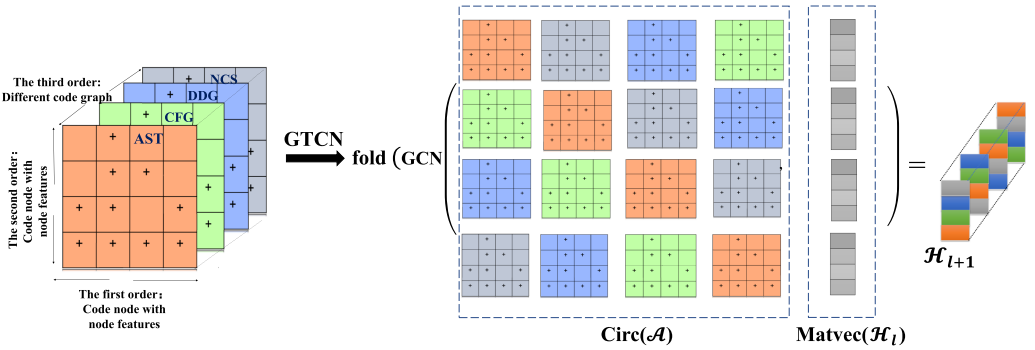

Fig. 5. The formulation of GTCN.

Let the $\widetilde{\mathcal{D}}(j,j,i) = \sum_{\kappa=1}^{n} \mathcal{A}'(j,\kappa,i)$ and $\mathbf{W}_l \in \mathbb{R}^{m\times m}$ is a layer-specific trainable weight matrix. $\sigma(\cdot)$ denotes a non-linear activation function such as ReLU or Leaky ReLU. Matvec($\mathcal{H}_l$) takes the tensor $\mathcal{H}_l$ into a block $4n \times m$ matrix and fold($\cdot$) makes the matrix into the tensor. Matvec($\mathcal{H}_l$) = $[\mathcal{H}_l^{(1)}, \mathcal{H}_l^{(2)}, \mathcal{H}_l^{(3)}, \mathcal{H}_l^{(4)}]$.

For the normalized symmetric graph adjacency tensor $\widetilde{\mathcal{A}} \in \mathbb{R}^{n\times n\times 4}$, it has four frontal slices of size $n \times n$, $\widetilde{\mathcal{A}}^{(1)}, \ldots, \widetilde{\mathcal{A}}^{(4)}$, and the Circ($\widetilde{\mathcal{A}}$) is defined as follows:

$$\text{Circ}(\widetilde{\mathcal{A}}) = \begin{bmatrix} \widetilde{\mathcal{A}}^{(1)} & \widetilde{\mathcal{A}}^{(4)} & \widetilde{\mathcal{A}}^{(3)} & \widetilde{\mathcal{A}}^{(2)} \\ \widetilde{\mathcal{A}}^{(2)} & \widetilde{\mathcal{A}}^{(1)} & \widetilde{\mathcal{A}}^{(4)} & \widetilde{\mathcal{A}}^{(3)} \\ \widetilde{\mathcal{A}}^{(3)} & \widetilde{\mathcal{A}}^{(2)} & \widetilde{\mathcal{A}}^{(1)} & \widetilde{\mathcal{A}}^{(4)} \\ \widetilde{\mathcal{A}}^{(4)} & \widetilde{\mathcal{A}}^{(3)} & \widetilde{\mathcal{A}}^{(2)} & \widetilde{\mathcal{A}}^{(1)} \end{bmatrix}. \tag{7}$$

We can learn code node features via the above graph tensor convolution layer. As we know, the graph structure is generally very irregular, and the graph data can be considered infinite-dimensional data, so it has no translation invariance [55]. The graph-based model can not directly learn the node order like CNN and RNN (the sequence-based neural network models). However, the order of the nodes plays an important role in representing code functions, since the code is also a kind of text, while most existing graph-based approaches did not extract the node order information [14, 16, 72, 74].

Our GTCN model directly learns the time sequence information of codes from the NCS of the code graph tensor. The time sequence information of codes represents the code node order, which is the natural sequential order of the code when the code is produced by the programmer. The graph of NCS adds edges (NextToken edges) between each syntax token and the successor of this token, which can not be represented by a normal AST edge. The order of operands and opcode in statements is represented by the NCS edges. NCS can complement the classical code representations (AST, CFG), because its unique flat structure captures the order relationships of code tokens in a "human-readable" fashion.

Moreover, motivated by tensor product [41], we design an algorithm that combines the circulate computation on the code graph tensor and the standard GCN [43], which can capture the misaligned correlation between node embeddings from the different code graphs. Our model can directly perform information propagation of code nodes from the third dimension of the code graph tensor with considering all orders of graphs in the code graph tensor, since we use the circulate computation of tensor product to compute the node embedding (see Equations (6) and (7)). We give an example to explain the influence of graph orders in our GTCN model.

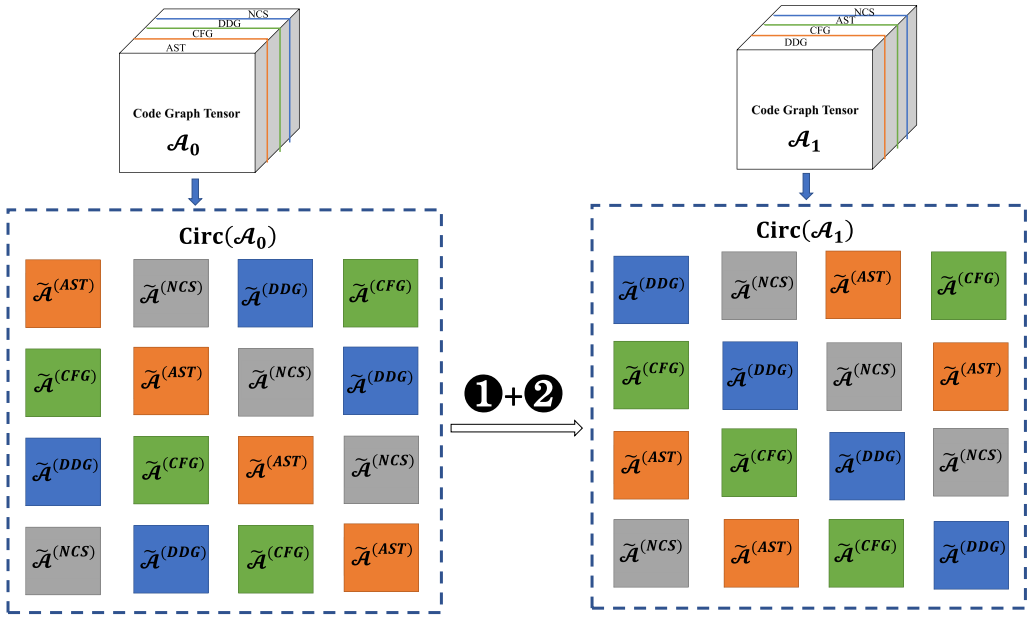

Fig. 6. An example of graph order in our GTCN model.

As shown in Figure 6, there are two different tensors, and they have different graph orders in the frontal slices of tensor, that is, in the tensor $\mathcal{A}_0$, the first frontal slice matrix $\mathcal{A}_0^{(1)}$ represents AST graph, the second frontal slice matrix $\mathcal{A}_0^{(2)}$ represents CFG graph, the third frontal slice matrix $\mathcal{A}_0^{(3)}$ represents DDG graph, the fourth frontal slice matrix $\mathcal{A}_0^{(4)}$ represents NCS graph, and in the tensor $\mathcal{A}_1$, the first frontal slice matrix $\mathcal{A}_1^{(1)}$ represents DDG graph, the second frontal slice matrix $\mathcal{A}_1^{(2)}$ represents CFG graph, the third frontal slice matrix $\mathcal{A}_1^{(3)}$ represents AST graph, the fourth frontal slice matrix $\mathcal{A}_1^{(4)}$ represents NCS graph.

The column transformation and the row transformation between two matrices are primary transformation, which does not change the rank and the linear correlation of row (column) vectors of $Circ(\widetilde{\mathcal{A}}_0)$. The primary transformation of circulant matrix $Circ(\widetilde{\mathcal{A}}_0)$ does not influence the subsequent $f_{GTCN}()$ computation. Therefore, the order of AST, CFG, DDG, and NCS has no influence on the performance of the proposed model, although the different orders of graphs lead to different tensors.

After training L graph tensor convolution layers, we get the final node embedding tensor $\mathcal{H}_L$ at the last layer. The final node embedding for each node v, $\mathbf{H}_v = \mathcal{H}_L(v, :, :)$. In practice, we use eight graph tensor convolution layers to compute node features.

3.3 Model Advantage

As we know, the key innovation of GNN is that nodes pass the message (coordinate information with each other) and update node features by propagating information among other nodes within the neighborhood. Motivated by message-passing steps of GNN, the single GCN can be used to

concisely propagate node information in code graphs, which can directly perform convolutional learning on code graphs.

However, the preliminary GCN model gathers all different code graphs in the same representation space in a "rude" manner and squeezes them into one graph. This operation destroys the tensor structure to a certain degree [81]. Different code graphs are used to maintain different properties of the given code data. Therefore, it is important to release some degree of freedom to learn the heterogeneous information of different code graphs in the tensor. We need to design a mechanism to harmonize various heterogeneous code information and different code graph structures during the learning process.

Equation (7) shows that tensor product operation is a closed operation that maintains element order. Through cyclic calculation in the tensor product, we can better learn the data correlation of code token features from three dimensions (various heterogeneous code graphs). The data of matrix structure can only learn row data relationship or column data relationship, while the data of tensor structure can learn not only row data relationship and column data relationship but also the third-dimensional code data correlation.

Moreover, as we know, the **graph convolution neural network (GCN)** can automatically learn both itself features of code nodes and the connection information between nodes. There are two main types of graph convolution neural networks: One is based on the spatial domain or vertex domain, and the other is based on the frequency domain or spectral domain. For example, the spatial domain can be analogous to convolution directly on nodes of a graph, while the spectral domain can be analogous to convolution after the Fourier transform of the graph.

The GCN shares the filter parameters over the whole graph. The form of the GCN propagation rule can be motivated via the first-order approximation of localized spectral filters on graphs. Successive filtering operations or convolution layers can effectively convolve the L-order neighborhood of a node, where L is the number of successive filtering operations or convolution layers in the neural network model.

In practice, it can be beneficial to constrain the number of parameters further to address the overfitting problem and to minimize the number of computation operations per layer (such as matrix multiplications).

Our model uses a high-dimensional tensor structure and the circulation operations of tensor product to combine the spatial domain and the spectral domain of code nodes sequence and code graphs. In our toward interpretable graph tensor convolution neural network model, the tensor as an amplifier can enhance the fast and scalable semi-supervised classification ability of the GCN model. Our tensor-based GCN model can improve the ability of accurate code semantic embedding for the neural network-based code semantic embedding approaches.

In the computation process of our GTCN model (Equation (6)), the tensor product of f_{GTCN} can reduce the number of parameters in the optimization problem, which can solve the overfitting phenomenon in the learning process to some degree [15]. We use the multilinear tensor subspace computation in GTCN, and our model only needs to calculate at most $m \times m$ parameters [50] per each hidden layer. If we unfold the code graph tensor $\mathcal{A}$ as four separate graph matrices to input into the standard GCN model, then the standard GCN model needs to calculate $4 \times m \times m$ parameters per each hidden layer. If the model has a total of L layers, then our model saves $4 \times L$ times parameters compared with the standard GCN.

Mathematically, the tensor product is a bilinear form, they use second-degree polynomial interactions to improve the expressiveness of the model, and the standard matrix dot product only uses first-degree polynomial interactions [70]. The second-degree polynomial interactions can learn more code feature correlation than the first-degree polynomial interactions in the deep learning algorithm [70]. We empirically prove that the tensor-based graph neural network performs much

better than the sequence neural networks and the standard graph neural networks for code embedding in Section 6.

4 TASKS

Our code-embedding model focuses on solving three tasks: variable name misuse, source code prediction, and vulnerability detection. We propose a candidate-based attention output layer, a composited output layer, and a simple classification layer to achieve these three tasks. We consider the attention mechanism, which combines the initial node representation $\mathbf{X}_v$ and node representation $\mathbf{H}_v$ of the final hidden layer to compute the node output vector $\mathbf{o}_v$. We assume that a variable node v is to be predicted. To compute the representation $\mathbf{H}_v$ in the final hidden layer, we generate a new node at the position of v and call this new node as v_{slot}, which uses all types of edges to connect to node v's neighbors, i.e., $\mathbf{H}_v = \mathbf{H}_{v_{\text{slot}}}$. We replace $\mathbf{H}_v$ with $\mathbf{H}_{v_{\text{slot}}}$, which allows variable nodes to better learn node features without relying on node v itself. After obtaining $\mathbf{o}_v$ through the output layer, we use the hierarchical softmax function to predict or locate variable names.

4.1 Variable Misuse Detection

Candidate-based Attention Output Layer. Besides the node representation in the hidden layer, we consider the candidate-based attention for solving the variable misuse detection task. In this task, we use all type-correct suggestions that can be used at a predicted location. Let $\mathbb{V}_i$ represent the collection of all type-correct variable nodes at i location in scope. That is, those variables can not raise a compiler error at the location i. This location is denoted as a slot. We need to correctly select variable v_i from $\mathbb{V}_i$. The information of the candidates can help our model to predict the token more accurately. When predicting the token at the slot i, we depend on not only the node feature of $\mathbf{H}_i$ in the hidden layer but also candidate feature $\mathbf{U}_{i,v}$.

We use hierarchical softmax classification function to obtain the output y_i^o of node i to be predicted by concatenating the initial node embedding of node i (X_i), the node embedding of node i at the hidden layer (H_i), and node embeddings of candidates ($U_{i,v}$): We use hierarchical softmax classification function to obtain the output $\mathbf{y}_i^o$ of node i to be predicted:

$$\begin{aligned} \mathbf{o}_i &= sigmoid(\mathbf{W}^o(\mathbf{X}_i; \mathbf{H}_i; \mathbf{U}_{i,v}; \mathbf{H}_i\mathbf{U}_{i,v}^+)), \\ \mathbf{y}_i^o &= h_{softmax}(\mathbf{W}^y\mathbf{o}_i + \mathbf{b}^y), \end{aligned} \tag{8}$$

in which $\mathbf{W}^o \in \mathbb{R}^{m\times(d+m+1)}$, $\mathbf{W}^y \in \mathbb{R}^{c\times m}$, $\mathbf{b}^y \in \mathbb{R}^c$ are trainable parameters, $\mathbf{o}_i$ is a vector of size $m \times 1$ that combines all candidate information, where c is the candidate size, and m is the size of node representation in the hidden layer, and ";" represents to connect the left term and the right term. Neural networks are trained iteratively, which update the model parameters until the loss of the model is converged to a minimum over an entire dataset.

4.2 Source Code Prediction

Vocabulary-based Output Layer. We consider the token to be predicted that is from the prelearned global vocabulary rather than candidates. The source code prediction model is used to predict the next source code based on the leftward context tokens. This vocabulary-based output layer projects the final token embedding into the vocabulary, and the final token embedding is learned from previous graph tensor convolution layers. Then, we use the hierarchical softmax classification function to obtain the output $\mathbf{y}_t^v \in \mathbb{R}^{|C|}$.

$$\begin{aligned} \mathbf{o}_i &= sigmoid(\mathbf{W}^o(\mathbf{X}_i; \mathbf{H}_i)), \\ \mathbf{y}_i^v &= h_{softmax}(\mathbf{W}^v\mathbf{o}_i + \mathbf{b}^v), \end{aligned} \tag{9}$$

where $\mathbf{W}^o \in \mathbb{R}^{m \times m}$, and $\mathbf{W}^v \in \mathbb{R}^{|C| \times m}$ are trainable parameters. $|C|$ is the size of the vocabulary, m is the size of node representation in the hidden layer, and ";" represents to the connection between the left term and the right term.

The Composited Output Layer. The composited output layer combines the candidate-based attention output and vocabulary-based output. Only considering the vocabulary-based model for the source code prediction task may cause invalid predictions for the previously unseen or rare variable name (OoV problem). Moreover, vocabulary-based models are slower and larger. If a vocabulary-based model achieves good prediction accuracy, then its vocabulary must be sufficiently large to contain a number of suggestions. A large vocabulary implies a large node embedding matrix, which substantially occupies memory space. The computation speed of this model is relatively slow, since it needs to compute Equation (9) over the whole vocabulary V. If we combine the candidate-based attention output with the vocabulary-based output, then the candidate providers that come from the leftward code tokens are used to supply the candidate providers.

Specifically, we use a switcher $s_i \in [0, 1]$ to indicate the probability of a token to be predicted from the global vocabulary, and then $1 - s_i$ indicates the probability of a token to be predicted from the local candidates. Comparing s_i and $1 - s_i$, we choose one output layer with a higher probability value. Formally, the vocabulary-based produces a probability distribution $\mathbf{y}_i^v \in \mathbb{R}_V$ for the next token $\mathbf{X}_i$ within the vocabulary. We set $s_i = sigmoid(\mathbf{W}^s \mathbf{H}_i + \mathbf{b}^s)$, where $\mathbf{W}^s \in \mathbb{R}^m$ and $\mathbf{b}^s \in \mathbb{R}^1$ are trainable weight and bias. $s_i \in [0, 1]$ is used to control the influence of $\mathbf{y}_i^o$ and $\mathbf{y}_i^v$. Finally, we have the final prediction by concatenating these two probability distributions: $\mathbf{y}_i = [s_i \mathbf{y}_i^o; (1 - s_i)\mathbf{y}_i^v]$.

We combine much more precise and informative candidates with vocabulary providers. Such candidate providers are preferred, since they usually make valid suggestions at the location to be predicted. Besides, vocabulary providers can provide in partial or incomplete code contexts, while local candidates can not yield informative results.

4.3 Vulnerability Detection

We use a **multi-layer perceptron model (MLP)** as the classification layer to classify vulnerable source codes through a sigmoid function.

5 INTERPRETABILITY OF GTCN MODEL

The intrinsic interpretability of graph neural networks is to find a small subset of the input graph's features, which guides the rationale model prediction [75]. Reference [80] proposed a novel taxonomy on the interpretability of neural networks with three dimensions: "Passive" vs. "Active" approaches; type of explanations ("Examples," "Attribution," "Hidden semantics," "Rules"); "Local" vs. "Global" interpretability. The detailed introductions are shown in the Reference [80].

Our model aims to design a toward interpretable graph tensor convolution neural network to learn code semantics better. Motivated by the survey on the interpretability of neural network [80], we explain the interpretability of our model by discussing the explanation pattern of "Passive," "Attribution," as Explanation ("Global"). Specifically, we have the following discussions in three parts:

(1) In our model, we consider various heterogeneous code graphs to represent different attributions of code data. We define the relaxed code sub-graph to analyze the influence of each attribution. Specifically, we use four heterogeneous AST, CFG, DDG, and NCS graphs to represent four main relaxed sub-graph for the input code data.

AST can represent all syntax information between code nodes, while there may be a long distance between two nodes with relevant semantic information in AST. Two nodes in DDG are directly connected to reduce the distance between correlated nodes further. CFG represents control information and jumps information, this information can not be well represented in AST. NCS

retains the sequence information between nodes. NCS can complement the classical representations, because its unique flat structure captures the relationships of code tokens in a "human-readable" fashion.

The rationale for choosing various code graphs as an interpretation is that vulnerabilities or misuse variables often involve the data sequence text and control dependencies [60]. Each graph as one attribution represents one concept to learn code hidden semantics. AST, CFG, and DDG keep the space structure of codes. NCS keeps the time sequence information of code nodes. Most existing works omit the time sequence information of code nodes. Our model combines the time sequence information of code nodes with the structure information of other heterogeneous code graphs. These four attributions are manually picked. A natural way of our model to get global attribution is to design a high-dimensional tensor structure to combine these four individual attributions (AST, CFG, DDG, NCS).

(2) We use gradient-related and backpropagation methods to explain our model interpretability. The goal is to find interpretable information propagation of heterogeneous code graphs. We analyze the backpropagation of our GTCN model to show mathematical interpretability.

We explain how global attribution influences the weight in our proposed GTCN model. Generally, we use backpropagation to train the graph-based neural network model. In our three tasks, we need to perform prediction or classification. In practice, the softmax function is used in tandem with the negative log-likelihood loss to perform classification. Assume that the input x_i has an output $\mathbf{y}_i^o$, and the $\mathbf{y}_i^o$ is the output of hierarchical softmax classification function that is calculated in the output layer of our model, then the negative log-likelihood loss is defined as the Equation (10):

$$E(\theta) = -\log \mathbf{y}_i^o \tag{10}$$

and our objective is to minimize the negative log-likelihood w.r.t. all weight parameters θ. To optimize $\mathbf{W}_l$ weight parameters, we need to find derivative $E(\theta)$ w.r.t. $\mathbf{W}_l$. For applying backpropagation through time, we need to unfold our GTCN and backpropagate the error from $E(\theta)$ to all previously hidden layers $\mathbf{H}_j$ to calculate the $\mathbf{W}_l$ where $j \in [1, \ldots, l]$.

$$\begin{aligned}\frac{\partial E(\theta)}{\partial \mathbf{W}_l} &= \sum_{j=1}^{l} \frac{\partial E(\theta)}{\partial \mathcal{H}_j} \frac{\partial \mathcal{H}_j}{\partial \mathbf{W}_l} = \sum_{j=1}^{l} \frac{\partial E(\theta)}{\partial \mathcal{H}_j} \frac{\partial \mathcal{H}_j}{\partial \mathcal{B}_j} \frac{\partial \mathcal{B}_j}{\partial \mathbf{W}_l} \\ &= \sum_{j=1}^{l} \frac{\partial E(\theta)}{\partial \mathcal{H}_j} f'(\mathcal{B}_j)(\text{fold}(\text{Circ}(\widetilde{\mathcal{A}}) \cdot \text{Matvec}(\mathcal{H}_l))),\end{aligned} \tag{11}$$

where $\mathcal{B}_l = \text{fold}(\text{Circ}(\widetilde{\mathcal{A}}) \cdot \text{Matvec}(\mathcal{H}_l)) \times_3 \mathbf{W}_l$. Since $f(\cdot)$ is a non-linear activation function, such as ReLU or Leaky ReLU, the derivatives of $f(\cdot)$ are:

$$f'(\mathcal{B}_j) = \begin{cases} 1, \text{if } \mathcal{B}_j > 0 \\ 0, \text{if } \mathcal{B}_j < 0 \end{cases}. \tag{12}$$

In our task, according to the outputs of Equations (8) and (9), the y_i is obtained by $\mathbf{o}_i$ and hierarchical softmax function, the $\mathbf{o}_i$ is obtained by a sigmoid function. We can obtain $\frac{\partial E(\theta)}{\partial \mathcal{H}_j}$ by seeking the derivatives of the sigmoid function and hierarchical softmax function.

In the above partial derivative $\frac{\partial E(\theta)}{\partial \mathbf{W}_l}$, we can see that the derivatives of the weight are multiplied by the pre-activated hidden unit error in the neural network with tensor products. According to the derivations in the tensor-based backpropagation process [70], we know that the tensor computation can directly learn more information from the input ($\mathcal{A}$, the high-dimensional code relationship graphs) and hidden units ($\mathcal{H}$, the code node feature maps of a filter calculated on

all graphs) than the standard dot product and addition operations of a typical neural network. The tensor computation improves the performance of the deep learning model and enhances the expressiveness of the neural network.

Neural networks with tensor computation can mediate the interaction between entity vectors, and the tensor structure can extend the data relationship even without external code context information. Our model uses arbitrary-order proximity to generate an interpretable feature space for the code nodes.

These heterogeneous code graphs (AST, CFG, DDG, NCS) are entangled with each other in the proposed code graph tensor. In other words, a code graph tensor as a high-layer filter may contain a mixture of different code semantic patterns. The circulate operation of the tensor product (a kind of tensor computation used in our model) in the GTCN model can encourage high-layer filter (code graph tensor) to represent integral code semantics (see Equations (6) and (7)).

Tensor as a high-dimensional data structure [41] can strengthen various properties of code, which can handle the topology information of different code semantic graphs and the sequential information of code token sequences into the same computation space. The tensor structure can handle more data correlation [31], which is better for learning the hidden semantics of code as a global attribution.

Tensor computation can better learn the data correlation of source code from three dimensions [63]. We use a three-order tensor to represent code data shown in Figure 3, the first dimension (the row data relationship) is code nodes, the second dimension (the column data relationship) is code nodes, and the third dimension (the front data relationship) is different code graphs.

Vividly, our model can simultaneously perform intra-graph propagation and inter-graph propagation from the code graph tensor. The intra-graph propagation aggregates information from neighbors of each node per graph. Our model parallelly performs intra-graph propagation in four code graphs (that is, AST, CFG, DDG, NCS) from the first dimension and the second dimension of the code graph tensor.

Our model directly performs the inter-graph propagation of code nodes from the third dimension of the code graph tensor without considering the order of graphs, since we use the circulate computation of tensor product to compute the node embedding at the hidden layer (see Equations (6) and (7)). The inter-graph propagation is used for harmonizing heterogeneous information between graphs, which is used to exchange information between different code graphs. The node also can learn features from node-self via inter-graph propagation.

Figure 7 shows the tensor propagation process of code node information in the code graph tensor. The tensor propagation is from three parts: one is to aggregate information from neighbors of each node within a graph (intra-graph propagation), another is to exchange information between different graphs in the tensor (inter-graph propagation), and the node also can learn features from node-self (inter-graph propagation).

For example, we can not learn the semantic information of code execution and data transfer from the AST, but we can learn from other CFG, DDG, and NCS via intra-graph propagation. The node connectivity in other graphs also can reflect in this graph. In the tensor structure, they are influenced by each other, and these four graph matrices are complementary to each other. The matrix only can learn the row data relationship or the column data relationship, but the tensor can learn not only the row data relationship and the column data relationship but also the third-dimension data relationship. Therefore, a tensor can better learn data correlation and handle data structure. A tensor can be used to spread and assemble code information across different token-relation graphs.

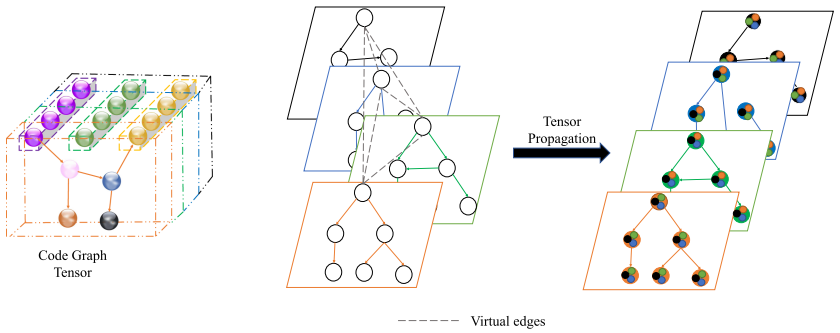

Fig. 7. The tensor propagation process of code nodes in the code graph tensor.

Especially for our three tasks, the variable misuse detection and the vulnerable detection tasks focus more on structure information of code semantics. For example, insecure argument problems can be detected by ASTs alone [65]. Resource leaks and some use-after-free vulnerabilities can be detected by ASTs with control flow graphs [81]. It has been demonstrated effective for the variable name misuse task to use heterogeneous code graphs with type information of the nodes and the edges [73, 74]. The recent advance in deep learning-based source code prediction task has demonstrated the effectiveness of the **natural sequence of source code (NCS)** [9, 51], since the source code prediction task depends more on the node sequence information of code contexts. NCS can complement the classical representations, because its unique flat structure captures the relationships of code tokens in a "human-readable" fashion.

Moreover, to evaluate how well our model produces interpretable tensor propagation of heterogeneous code graphs, we carry out the experiments on three tasks to analyze the model interpretability in Section 6.

6 EVALUATION

We evaluate our code-embedding model concerning its accuracy and efficiency for three tasks: variable misuse detection [8], source code prediction [74], and vulnerability detection. We, respectively, discuss these three tasks in the following subsections. We show how we perform experiments and report the evaluation results, compared with other state-of-the-art methods. We release the source code of our model and datasets on our Gitee [5] (or GitHub [6]).

6.1 Variable Misuse Detection

We evaluate our code semantic embedding method on the variable misuse detection task.

6.1.1 Dataset. Primarily, we download large open-source C# repositories on Github as datasets. We used the same 29 projects as Allamanis [8] and add other latest 20 top-star projects. The full dataset description of Allamanis 29 projects is shown in Appendix D of reference [8]. Other latest 20 top-star projects are "shadowsocks-windows," "roslyn," "Ryujinx," "MaterialDesignlnXAML," "OpenRA," "mono," "UnityCsReference," "csharplang," "ArchiSteamFarm," "uno," and so on. They are distinct corpora from diverse domains. This dataset contains about 4.6 million of non-empty lines of code, 17,942,163 code nodes, and 24,144 large composited graphs.

To better evaluate and compare with other state-of-the-art methods, we use the same way to collect the VISIBLETEST dataset and the INVISIBLETEST dataset as Allamanis [8]. We mix the first 31 projects to train the model and split these projects into the proportion $train$ 6 : $validation$ 1 : $test$ 3 for the VISIBLETEST dataset. Besides, we use the last eight projects to form the INVISIBLETEST

dataset ("scriptcs," "ServiceStack," "ShareX," "SignalR," "Wox," "csharplang," "ArchiSteamFarm," "uno"), which are used to test on completely unknown projects.

Moreover, we add a public corpus of GitHub Python files extensively [45], ETH-Py150. After deleting duplicate files, we further extract all top-level functions from the Python dataset. It consists of 394K training functions, 42K validation, and 214K test unique functions. For each function and slot pair, we generate a buggy example by replacing the original variable at the slot location, and we randomly chose an incorrect variable in this location. There is also one bug-free example without any modification. ETH-Py150 contains Python files with a function-level scope, slots in Python function are variables, and candidates are variables in the scope of the slot. Since Python is dynamically typed, there is no type information to be used.

6.1.2 Compared Method. We evaluate our model and the following state-of-the-art variable misuse detection methods for performance comparison.

The baseline of our model: We set 8 graph tensor convolution layers as one hyperparameter of the proposed model. In a batch size, the number of graph nodes is 500,00. The initial learning rate is set as 0.001, and we decay it by multiplying 0.1 after about 30 epochs. We train our model for a maximum of 300 epochs using Adam and early stopping with a window size of 10. After 150 epochs, we stop training our model. We set the embedding size of a code node value as 128 for the C# dataset. The number of candidates is set as 10.

Allamanis [8]: They used scale-gated graph neural networks to learn code embedding by constructing graphs from source codes. This model used two message-passing layers, in which the first layer had four node message passes per block. We then sweep over GGNN architectures that repeat these blocks 1 to 4 times (that is, 4 to 16 message passes). We include an ablation with 256-dimensional node messages.

GINN [74]: GINN focused on intervals of program graphs for mining a program's feature representation. Intervals generally manifested in looping construct. Furthermore, GINN operated on a hierarchy of intervals for scaling the learning process to large graphs. They published the source codes in GitHub [2]. We implement one abstraction cycle within GINN. We keep all GGNN's default parameters (e.g., size of node embeddings, number of layers) in GINN.

GraphCodeBERT [23]: GraphCodeBERT considered the inherent structure of code. This work improved CodeBert [19] and published the source codes in GitHub [4]. GraphCodeBERT includes 12 layers Transformer with 768-dimensional hidden states and 12 attention heads. In the fine-turning step, we set the learning rate as 2e-5, the batch size as 32, the max sequence length of queries and codes as 128 and 256, and the max number of nodes as 64. We use the Adam optimizer to update model parameters and perform early stopping on the training set.

RNN Sandwich [27]: They wrapped traditional (gated) graph message-passing layers in sequential message-passing layers. They published the source codes in GitHub [3]. In the RNN layer, the size of node embedding is a 128-dimension vector, and the number of layers is 2. The number of message-passing blocks is 3 to span a similar parameter domain as the GGNN module.

RGCN [66]: This paper proposed a relational graph convolution neural network. They performed this model for two tasks: link prediction and entity classification. For RGCN, we used a 2-layer model with 16 hidden units and trained with Adam for 50 epochs using a learning rate of 0.01.

GTN [78]: This paper proposed graph transformer networks that could generate new graph structures of source codes, which learned the composited relations of the selected edge types for developing useful multi-hop connections. We use three graph transformer layers and use the 1×1 convolution layer to initialize the parameters of the graph transformer layer.

TensorGCN [54]: Existing TensorGCN work first performs intra-graph propagation and then performs inter-graph propagation. For each layer of existing TensorGCN, first, it performs

standard GCN on each horizontal slice matrices of the tensor ($\hat{\mathcal{A}}(i,:,:)$), respectively, then, it performs standard GCN on each frontal slice matrices of the tensor ($\hat{\mathcal{A}}(:,:,i)$), respectively. We train a two-layers TensorGCN model, and the node embedding is set as 200-dimension. In the training process, the dropout rate is 0.5 and the training epochs are 200. The model uses the Adam optimizer with a learning rate of 0.002, and early stopping is performed when validation loss does not decrease for 10 consecutive epochs.

6.1.3 Experiment Setup. In our GTCN model, we use mini-batch SGD with the Adam optimizer [39] and the categorical cross-entropy loss function to train. We report the average result after running each experiment three times. Our model is implemented by PyTorch [61], and we run our experiments on a Linux server with the NVIDIA QUADRO RTX 5000. This server has 16*GB* GPU memory.

Label method. In the variable misuse detection task, there is at least one type-compatible replacement variable token in the candidate scope. The type-compatible replacement variable tokens do not raise a compiled error during the type-check process. Each of these candidate nodes represents the speculative placement of the variable within the scope. We label the originally existing correct variable token at the predicted location as the right prediction.

Initialization. Initialization is responsible for mapping each code token into a numerical vector. We use two one-dimensional convolution layers and one one-dimensional maximum pooling layer to obtain initial node embedding. For the C# dataset, we set the embedding size of a node as 256, in which the embedding size of the type is 128 and the embedding size of the value is 128. For the Python dataset, we set the embedding size of value as 256. Since Python is dynamically typed, we do not consider the type information.

Metrics. Our evaluation metric contains top-1 recall (Top-1), top-5 recall (Top-5), accuracy, and F1. Top-1 recall of variable misuse detection points to how often the model correctly localizes the wrong code token. Accuracy means how often the model correctly predicts labels. We use the same method to measure F1 as Reference [20].

6.1.4 Results. Results of the variable misuse detection task using large VISIBLETEST C# dataset are shown in Table 1. We report top-1 recall, top-5 recall, accuracy, and F1. We construct source codes as a code graph tensor and then input the code graph tensor into our model. Our model outputs the fixing code token at the location to be detected. Our model performs better than other methods regarding the top-1 recall, top-5 recall, accuracy, and F1. The top-1 recall of our method is higher than that of Allamanis by 18%, than that of traditional deep GCN by 37%, than that of GINN by 10%, than that of RNN sandwich by 11%, than that of RGCN by 25%, than that of GTN by 31%, and than that of TensorGCN by 13% for the VISIBLETEST dataset. Figure 8 shows the ROC curve for variable misuse detection.

Generalizing across various code projects from different domains is a key challenge for deep learning. To better evaluate the generalization ability of our tensor-based model, we do experiments using the INVISIBLETEST dataset, which has no same files in the training set. Results of the variable misuse detection task using large C# INVISIBLETEST dataset are shown in Table 2. Our model outperforms other state-of-the-art methods in terms of top-1 recall, top-5 recall, accuracy, and F1; although it is a little lower than the performance with the VISIBLETEST dataset. Because some hierarchy types are unknown, the unused code variables, method, and class names, and so on, can differ substantially in the INVISIBLETEST dataset (C#).

The top-1 recall of our method is higher than that of Allamanis by 30%, than that of traditional deep GCN by 36%, than that of GINN by 13%, than that of RNN sandwich by 13%, than that of RGCN by 42%, than that of GTN by 25%, than that of GraphCodeBert by 1%, than that of TensorGCN by 11% for the INVISIBLETEST dataset (C#).

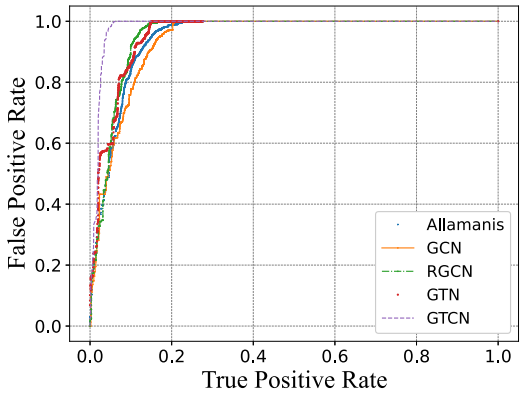

Fig. 8. ROC curve for variable misuse detection.

Table 1. Evaluation for the Variable Misuse Detection Task Using VISIBLETEST C# Dataset

Method	VISIBLETEST dataset (C#)			
	Top-1	Top-5	Accuracy	F1
Allamanis	0.769	0.962	0.909	0.862
GCN	0.572	0.928	0.920	0.756
GINN	0.849	0.992	0.934	0.889
GraphCodeBert	0.937	0.995	0.942	0.940
RNN sandwich	0.835	0.991	0.918	0.864
RGCN	0.701	0.995	0.939	0.830
GTN	0.641	0.994	0.933	0.793
TensorGCN	0.818	0.996	0.924	0.876
GTCN	**0.946**	**0.998**	**0.953**	**0.971**

Table 2. Evaluation for the Variable Misuse Detection Task Using INVISIBLETEST C# Dataset

Method	INVISIBLETEST dataset (C#)			
	Top-1	Top-5	Accuracy	F1
Allamanis	0.580	0.943	0.888	0.758
GCN	0.515	0.931	0.865	0.724
GINN	0.746	0.965	0.898	0.790
GraphCodeBert	0.861	0.966	0.919	0.845
RNN sandwich	0.742	0.950	0.871	0.783
RGCN	0.459	0.920	0.853	0.689
GTN	0.622	0.945	0.898	0.785
TensorGCN	0.765	0.938	0.870	0.805
GTCN	**0.875**	**0.974**	**0.940**	**0.869**

Results of the variable misuse detection task using the Python dataset are shown in Table 3. Our model performs better than other methods in terms of top-1 recall, top-5 recall, accuracy, and F1. The top-1 recall of our method is higher than that of Allamanis by 18%, than that of traditional deep GCN by 25%, than that of GINN by 7%, than that of RNN sandwich by 4%, than that of GraphCodeBert by 4%, than that of RGCN by 29%, than that of GTN by 19%, than that of TensorGCN by 7% for Python dataset.

6.1.5 Hyperparameters. Hyperparameters of GTCN are tuned using four important features: the embedding size of node features, the number of candidates, batch sizes, and the influence of different code graph types in the code graph tensor. We perform hyperparameter experiments using the VISIBLETEST C# dataset and report the corresponding results after running experiments three times.

(1) Impact of embedding sizes. We measure the impact of the embedding size on accuracy using the VISIBLETEST C# dataset. For our model, we consider different embedding sizes as 64, 128, 256, 512, and 1,024. Table 4 reports Top-1, Top-5, accuracy, and F1 among different embedding sizes for the variable misuse detection task. Results show that the code node value with the 128 embedding size performs much better than other embedding sizes. Our model can not learn enough information from the code graph tensor with the 64 embedding size, while the large 1,024 embedding sizes lead to the overfitting problem in the deep learning model.

Table 3. Evaluation for the Variable Misuse Detection Task Using Python Dataset

Method	Python dataset			
	Top-1	Top-5	Accuracy	F1
Allamanis	0.612	0.905	0.659	0.643
GCN	0.538	0.894	0.636	0.595
GINN	0.721	0.918	0.764	0.745
GraphCodeBert	0.752	0.925	0.802	0.782
RNN sandwich	0.747	0.924	0.758	0.754
RGCN	0.501	0.895	0.738	0.602
GTN	0.601	0.905	0.755	0.720
TensorGCN	0.717	0.920	0.756	0.746
GTCN	**0.789**	**0.948**	**0.812**	**0.805**

Table 4. Evaluation for Variable Misuse Detection among Different Embedding Sizes

Embedding size (values)	VISIBLETEST dataset (C#)			
	Top-1	Top-5	Accuracy	F1
64	0.913	0.991	0.938	0.921
128	**0.946**	**0.998**	**0.953**	**0.971**
256	0.917	0.991	0.933	0.924
512	0.883	0.988	0.926	0.892
1024	0.784	0.976	0.907	0.805

(2) Impact of each graph type in code graph tensor. To evaluate the influence of different code graphs (AST, CFG, DDG, NCS), we remove any graph data from the code graph tensor and construct novel composited code graph representations. For example, "AST+CFG+DDG" means that we remove the NCS data in the code graph tensor and construct the AST, CFG, and DDG as a new tensor. Thus, the input data of the model becomes a new tensor of size $n \times n \times 3$.

Results of Table 5 show how different types of code graphs influence the performance of variable misuse detection. We use the VISIBLETEST C# dataset and set the embedding size of the code node value as 128. We find that the accuracy using full code graph tensor outperforms that using other composited code graph representations. Surprisingly, we also find that results learned from any combination of three code graphs are quite encouraging. In terms of top-1 recall, the full code graph tensor outperforms other combinations with three code graphs from 3% to 17%. The DDG and CFG have the greatest influence on learning the code semantic embedding for the variable misuse detection task.

For long-distance dependence of codes, the code graph tensor can shorten the distance between long-distance related nodes. These four graphs can fully represent the semantic information of the code, which can be used to learn accurate code semantics. Therefore, the full code graph tensor helps the GTCN model to learn better detection models than other graph combinations.

(3) Impact of the number of candidates. Table 6 shows the performance of the GTCN model on variable misuse detection per number of type-correct, in-scope candidate variables. Here, we compute the performance of the GTCN model that uses different candidates: $\{5, 10, 15, 20, 25\}$.

(4) Impact of batch sizes. We train our model with batch sizes of $\{2, 4, 6, 8\}$ graphs. Table 7 shows the performance of the GTCN model on variable misuse detection among different batch

Table 5. Evaluation for Variable Misuse Detection among Different Graph Combinations

Graph Combination	VISIBLETEST dataset			
	Top-1	Top-5	Accuracy	F1
Full Code Tensor	**0.946**	**0.998**	**0.953**	**0.971**
AST+CFG+DDG	0.917	0.991	0.918	0.905
AST+DDG+NCS	0.866	0.986	0.883	0.878
AST+CFG+NCS	0.776	0.971	0.868	0.791
CFG+DDG+NCS	0.865	0.986	0.872	0.886

Table 6. Evaluation for Variable Misuse Detection

The number of candidates	VISIBLETEST dataset (C#)			
	Top-1	Top-5	Accuracy	F1
5	0.970	0.998	0.906	0.958
10	0.946	0.998	0.953	0.971
15	0.944	0.996	0.942	0.938
20	0.906	0.971	0.934	0.881
25	0.888	0.968	0.921	0.886

Table 7. Evaluation for Variable Misuse Detection among Different Batch Sizes

Batch size	VISIBLETEST dataset (C#)			
	Top-1	Top-5	Accuracy	F1
2 graphs	0.765	0.919	0.903	0.846
4 graphs	0.904	0.983	0.939	0.926
6 graphs	0.946	0.998	0.953	0.971
8 graphs	0.904	0.983	0.939	0.926

Table 8. Evaluation for Variable Misuse Detection among Different Graph Orders

Graph Orders	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩
Top-1	0.946	0.948	0.948	0.950	0.948	0.944	0.948	0.948	0.948	0.944
Top-5	0.998	0.997	0.993	0.994	0.989	0.992	0.989	0.991	0.994	0.993
Accuracy	0.953	0.946	0.946	0.949	0.946	0.943	0.945	0.946	0.956	0.958
F1	0.971	0.964	0.972	0.975	0.969	0.969	0.969	0.974	0.976	0.973

sizes. The batch size with 8 graphs is too large for models, and we select the batch size with 6 graphs.

The batch size with large graphs usually are very sparse, and we represent edges of code graphs as an adjacency, which could reduce memory consumption. Our model can be easily implemented using a high-dimensional sparse tensor using GTCN for sets of large, diverse graphs, allowing large batch sizes that exploit the parallelism of modern GPUs efficiently.

(5) Impact of code graph tensors with different graph orders. To evaluate the influence of the order of graphs (AST, CFG, DDG, NCS) in code graph tensor, we evaluate experimental performance among different code graph tensors with different graph orders. We use the VISIBLETEST dataset (C#) to perform variable misuse detection. Vividly, we show different code graph tensors with different graph orders in Figure 9. Figure 9 contains most graph orders in code graph tensor. The former ①–④ are the clockwise transformation of different graphs in code graph tensors, and the latter ⑤–⑩ are the two exchange of different graphs in code graph tensors. Results of Table 8 show that different orders of code graphs have no great influence in the performance of variable misuse detection task, in which ① is used in our model.

6.1.6 Case Study. As shown in Figure 10, we use three different code snippets to show the result of variable misuse detection, which not only locates but also fixes the misused variable (i.e., the wrong code tokens are highlighted within the shadow box). Our model detects the right codes with high probability.

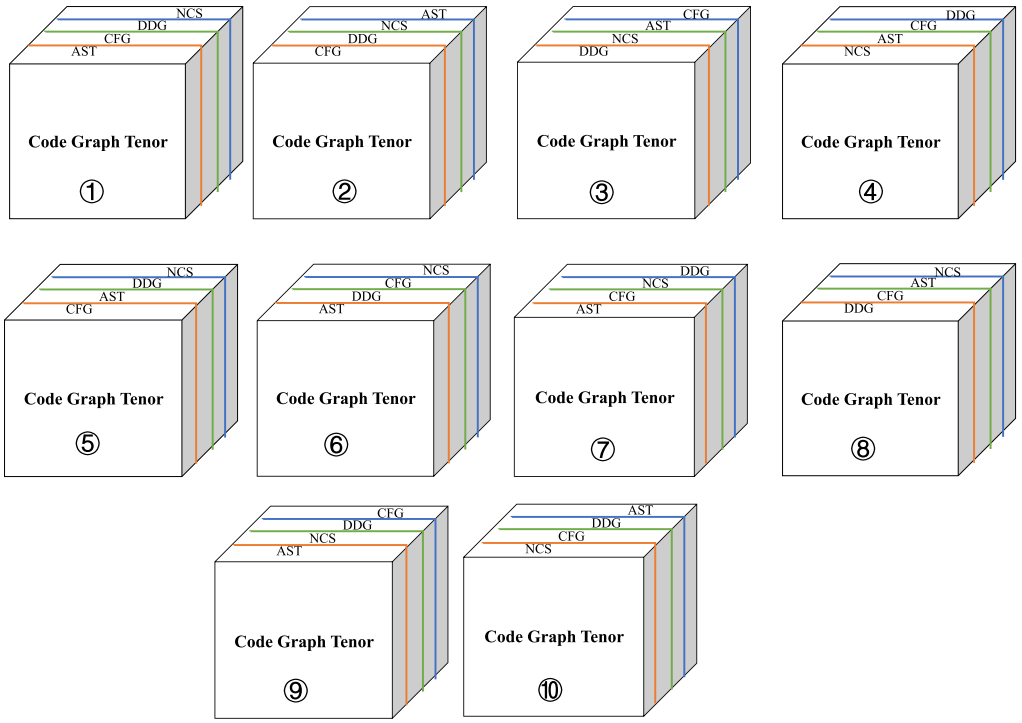

Fig. 9. Different code graph tensors with different graph orders.

The first top box shows the source codes of a "Get file Directories" algorithm. As shown in the shadow box of orange color, the *searchPath* is misused. In this location, it needs to create a new *baseDirectory* set if the *searchPath* is empty, rather than create *searchPath*, since this function is used to return the *baseDirectory*. Our model correctly predicts the code in this location.

In the second middle box of Figure 10, it shows the source codes of a "Joint set search" algorithm, which is used to check the connectivity of graphs or to judge whether the elements belong to the same set. As shown in the shadow box of blue color, the $Parent[qRoot] = qRoot$ in the *unionelements*() function is misused. The location of *qRoot* represents the parent node of *qRoot* as *qRoot*. That is, it takes itself as the parent node. Obviously, there is an error. The fixed results of our model show that the correct variable of this location should be *pRoot*, and the parent node of the *qRoot* node should be set to *pRoot*.

In the third bottom box of Figure 10, it shows the source codes of a "Array partition" algorithm. As shown in the shadow box of orange color, the $i++$ in the $Patition()$ function is misused. The location of $i++$ should be $ii++$, since the additive iteration of i is written in the for statement. Our model correctly detects $i++$ and fixes it as $ii++$.

GTCN is capable of capturing fine-grained semantic features of source codes. GTCN produces the correct prediction with high probability, demonstrating its higher precision in reasoning the semantics of a program.

6.2 Source Code Prediction

We evaluate our code semantic embedding method on the source code prediction task.

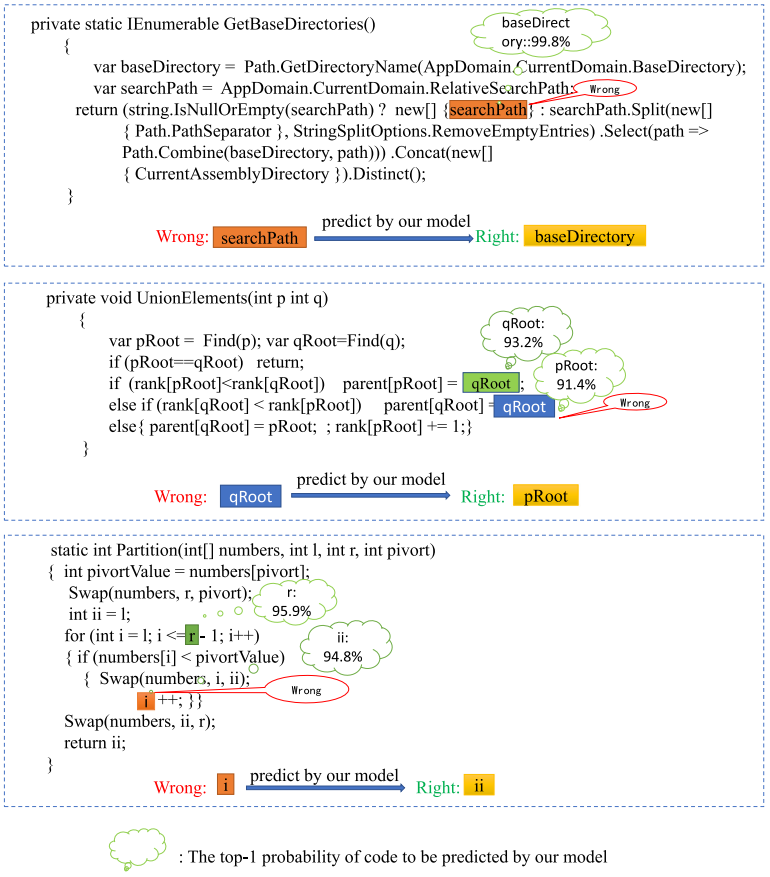

Fig. 10. Variable misuse detection on three different code snippets from the VISIBLETEST dataset.

6.2.1 Dataset. We used the traditional py150 dataset [45] that contains parsed ASTs for training and evaluating the source code prediction task. This dataset deletes duplicate files and the copy of another existing repository. We retain programs that resolve and have up to 30,000 nodes in AST. In addition, we only use repositories with licensed and non-virus licenses. We divide py150 dataset into three parts: 100,000 functions for training, 50,000 functions for verification, and 10,000 functions for testing. This dataset is the VISIBLETEST dataset (Python). Then, we generate the test set of the INVISIBLETEST dataset (Python) by randomly selecting 10,000 functions from other unseen Python datasets [23].

Specifically, we predict the code token depends on its leftward contexts, where all left tokens of this token can participate in its prediction. We input an average of 477.81 code tokens in the model, and the model outputs 6.61 tokens.

For parsing, we used the Python AST parser, Tree-sitter, to obtain the AST of the Python dataset. We extract other CFG, DDG, and NCS by our-self codes. To simulate the code running process in the real world, we truncate the extracted full graphs and keep the left part of the token to be predicted in four graphs. We use the partial AST, CFG, DDG, and NCS to perform the source code prediction.

Table 9. Evaluation for the Source Code Prediction Task Using VISIBLETEST Dataset (Python)

Method	VISIBLETEST dataset (Python)			
	Top-1	Top-5	Accuracy	F1
RNN	0.559	0.604	0.563	0.573
GGNN	0.532	0.602	0.553	0.558
Pointer	0.698	0.731	0.692	0.702
Transformer-XL	0.701	0.750	0.697	0.711
GINN	0.689	0.738	0.698	0.709
TensorGCN	0.601	0.664	0.670	0.695
GTCN	**0.761**	**0.873**	**0.743**	**0.749**

6.2.2 Compared Method. We use some same compared methods in the variable misuse detection task: GINN [74], GraphCodeBert [23], TensorGCN [54]. Moreover, we add the following other compared methods:

The baseline of our model: We set 10 graph tensor convolution layers for the source code prediction task. The initial learning rate is set as 0.001, and we decay it by multiplying 0.1 after about 80 epochs. After 230 epochs, we stop training our model. We set the embedding size of a node as 256.

Pointer [45]: Pointer proposed a pointer mixture network for achieving code completion. The pointer network learned the code token representation from the sequential code contexts via RNN. This model used a single-layer LSTM network with unrolling length of 50 and a hidden unit size of 1,500. We set the initial learning rate as 0.001 and decay it by multiplying 0.6 after every epoch. We clip the gradients' norm to 5 to prevent gradients from exploding. The size of the attention window is 50. The batch size is 128 and this model trains 8 epochs.

Transformer-XL [42]: This paper proposed a self-attention neural network model, which was used to generate code semantics embedding for source code prediction. The embedding sizes for the type and value of the code token are 300 and 1,200, respectively. The vocabulary is set as 50,000. We use UNK to represent values outside the vocabulary. We use a 6-layer Transformer-XL network as the partial AST encoder. We set the length of hidden units of node embedding as 256.

6.2.3 Experiment Setup.

Label method. In the source code prediction task, our vocabulary consist of the number of 100,00 most frequent tokens from all datasets. If tokens to be predicted are in vocabulary, then we use the corresponding IDs in vocabulary to label these tokens. If the tokens to be predicted are not in the vocabulary, then we check whether tokens are from candidates within the input graph, and then we label these tokens as the corresponding candidates. Otherwise, the OoV value is labeled as "unkn." We treat the predictions of "unkn" targets as wrong predictions, which is the same as Li [45].

Initialization. We use the same initialization way as the variable misuse detection task. For the Python dataset, we set the embedding size of value as 256. Since Python is dynamically typed, we do not consider the type information.

6.2.4 Results. As shown in Table 9, results of source code prediction show that GTCN performs better than other methods. The top-1 recall of GTCN is higher than that of Pointer by 6%, than that of Transformer-XL by 6%, than that of GINN by 7%, than that of TensorGCN by 16%, than that of traditional RNN, GGNN model by approximate 21% for VISIBLETEST dataset (Python). Compared to the results of TensorGCN [54], traditional GGNN [47], GCN [43], and our GTCN model, we can see that tensor computation has a tremendous and positive influence on prediction accuracy.

Table 10. Evaluation for the Source Code Prediction Task Using INVISIBLETEST Dataset (Python)

Method	INVISIBLETEST dataset (Python)			
	Top-1	Top-5	Accuracy	F1
RNN	0.361	0.437	0.463	0.399
GGNN	0.372	0.473	0.414	0.423
Pointer	0.648	0.702	0.691	0.635
Transformer-XL	0.662	0.708	0.658	0.645
GINN	0.612	0.628	0.608	0.609
TensorGCN	0.589	0.615	0.586	0.595
GTCN	**0.706**	**0.825**	**0.701**	**0.705**

Table 11. Evaluation for the Source Code Prediction Task among Different Embedding Sizes

Embedding Size	VISIBLETEST dataset (Python)			
	Top-1	Top-5	Accuracy	F1
64	0.592	0.714	0.569	0.603
128	0.644	0.650	0.697	0.668
256	0.761	0.873	0.743	0.749
512	0.758	0.895	0.738	0.739

For the INVISIBLETEST dataset (Python), results of the source code prediction task are shown in Table 10. The F1 of GTCN is higher than that of Pointer by 6%, than that of Transformer-XL by 5%, than that of GINN by 10%, than that of TensorGCN by 13%, and than that of traditional RNN, GGNN model by approximately 33%. GTCN uses candidate-based attention to solve the OoV problem and outperforms other works. Existing Pointer, GraphCodeBert, and Transformer-XL works are based on sequence approaches. These approaches need to train a large code corpus, which costs a lot of training time and computation space.

6.2.5 Hyperparameters.

(1) Impact of embedding sizes. We measure the impact of the embedding sizes on accuracy using the VISIBLETEST dataset (Python). For our model, we set different embedding sizes as 64, 128, 256, and 512. Table 11 reports Top-1, Top-5, accuracy, and F1 among different embedding sizes for the source code prediction task. Results show that both 256 and 512 embedding size performs much better than 64 and 128 embedding sizes. The source code prediction task needs more node features.

(2) Impact of each graph type in code graph tensor. To evaluate the influence of different code graphs (AST, CFG, DDG, NCS), we remove any graph data from the code graph tensor and construct novel composited code graph representations. For example, "AST+CFG+DDG" means that we remove the NCS data in the code graph tensor and construct the AST, CFG, and DDG as a new tensor. Thus, the input data of the model becomes a new tensor of size $n \times n \times 3$.

Results of Table 12 show how different types of code graphs influence the performance of source code prediction. We set the embedding size to 256 and use the VISIBLETEST Python dataset. We find that the accuracy of using a full code graph tensor outperforms that of using other composited code graph representations. In terms of top-1 recall, the full code graph tensor outperforms other combinations with three code graphs from 4% to 11%. The DDG and NCS have the greatest influence on the source code prediction task.

Table 12. Evaluation for Source Code Prediction among Different Graph Combinations

Graph Combination	VISIBLETEST dataset (Python)			
	Top-1	Top-5	Accuracy	F1
Full Code Tensor	**0.761**	**0.873**	**0.743**	**0.749**
AST+CFG+DDG	0.703	0.793	0.696	0.700
AST+DDG+NCS	0.721	0.821	0.691	0.715
AST+CFG+NCS	0.652	0.773	0.671	0.665
CFG+DDG+NCS	0.714	0.789	0.726	0.766

Table 13. Evaluation for Source Code Prediction among Different Graph Orders

Graph Orders	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩
Top-1	0.761	0.758	0.760	0.758	0.758	0.754	0.758	0.758	0.758	0.764
Top-5	0.873	0.877	0.873	0.875	0.875	0.874	0.869	0.872	0.873	0.874
Accuracy	0.743	0.745	0.744	0.749	0.746	0.733	0.745	0.746	0.745	0.741
F1	0.749	0.743	0.742	0.745	0.749	0.749	0.746	0.748	0.751	0.743

(3) Impact of code graph tensors with different graph orders. To evaluate the influence of the order of graphs (AST, CFG, DDG, NCS) in code graph tensor, we use the VISIBLETEST dataset (Python) to perform source code prediction among different code graph tensors with different graph orders. Different code graph tensors with different graph orders are shown in Figure 9, which is introduced in the variable misuse detection task. Results of Table 13 show that different orders of code graphs have no great influence in the performance of source code prediction task, in which ① is used in our model.

6.2.6 Case Study. As shown in Figure 11, we present source code prediction examples to analyze the GTCN performance using the Python dataset. We use several code snippets to test the performance of our model GTCN and Allamanis [8]. Figure 11 shows the top three predicted code tokens.

In the first box example, the target token to be predicted *baseDirectory* is the first new variable in *GetBaseDirectories* function. Neither our GTCN model nor Allamanis can correctly predict the code token *baseDirectory*. We can not obtain this code token from the candidates of left contexts and code vocabulary. We correctly predict this token at the top-2 rank in our model and is ranked third in Allamanis.

In the second box example, the target token to be predicted *AppDomain* means a class name. The corresponding node type of *AppDomain* is $FunctionDef$. GTCN is able to make the correct prediction using the information contained in the code graph tensor.

In the third box example, the target token to be predicted *searchPath* has not been used in the previous code contexts. Our model correctly predicts the code with 67.2% probability, while the baseline model fails. The Allamainis model can not correctly predict the object variable "searchPath". The last box example is also in the same way. Moreover, we add another test snippet sample to show the source code prediction performance in the Appendix.

6.3 Vulnerability Detection

We use the datasets that are from Devign [81] for vulnerability identification of C source codes. The samples contain four kinds of vulnerabilities, including 33,119 vulnerability functions and 126,882 normal functions. Table 14 shows the statistics of specific CWEs in all datasets. For detecting

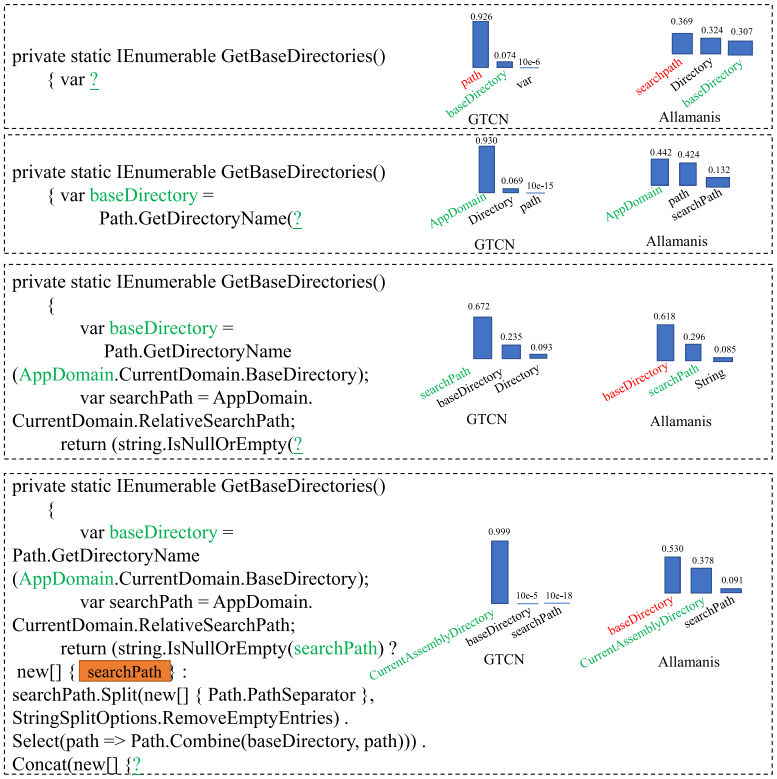

Fig. 11. Source code predictions on slots within a snippet code from Ninject project.

Table 14. CWE Statistics of Vulnerabilities

CWE ID	Vulnerabilities	Normal	Total
CWE-119	6,455	47,276	53,731
CWE-120	1,724	10,767	12,491
CWE-469	1,335	10,767	12,102
CWE-476	2,044	10,767	12,811
Composite	21,561	47,305	68,866

source code vulnerabilities at the function level, we label a vulnerable function with 1 and label a normal function as 0. We use the word2vec model to generate initial node embedding, and we set the embedding size as 512. The initial learning rate is set as 0.001, and we decay it by multiplying 0.1 after about 30 epochs. After 210 epochs, we stop training our model.

Graph Generation. After collecting source codes, we need to generate graph representations for programs and obtain initial representations for each node token. We use Joern [77] to extract ASTs and CFGs for all vulnerability functions or normal functions in datasets. Joern is a popular and easy tool to extract AST and CFG for C programs. We remove some functions that are compiled with errors in the extraction process of ASTs and CFGs, We collect 160,001 functions in total and remove 315 functions that can not be compiled via Joern. Since the original DDGs edges are labeled with the variables involved, to represent the DDG of source codes better and concisely, we use the same way as Devign [81] to handle the DDGs.

6.3.1 Compared Method. The baseline of our model: We use the word2vec model to generate initial node embedding, and we set the embedding size as 512. After 150 epochs, we stop training our model.

Devign [81]: Devign combined a gated graph neural network with a convolution layer for graph-level code semantic embedding, and then they evaluated the model for vulnerability detection. The source codes are published in Github [1]. In the embedding layer, initial node embedding is set as 100-dimension. We use the word2vec algorithm to generate initial node embedding. In the gated graph recurrent layer, the dimension of hidden states is set as 200, and the number of timesteps is set as 6. We use the Adam optimizer with a learning rate of 0.0001. The batch size is set as 128. The model trains 100 epochs using early stopping.

IVDetect [48]: IVDetect is an interpretable vulnerability detector using Intelligence Assistant to detect vulnerabilities. This model separately considered the vulnerable statements and their surrounding contexts via data dependencies and control flow. We use the dataset and source codes provided by IVDetect. The dataset and the source code of IVDetect were published in Reference [7]. Both our model and IVDetect use the same dataset provided by Devign. We report the average result after running this experiment three times. We use one ChildSumTreeLSTM layer, five GRU layers, and three GCNconv layers to train the IVDetect model. The batch size of graph nodes is 10. The learning rate is set as 0.0001, and this model sets the dropout rate as 0.3. The embedding size of a node is set as 128.

MVD [14]: MVD is a statement-level memory-related vulnerability detection approach based on **flow-sensitive graph neural networks (FS-GNN)**, which only focused on capturing implicit memory-related vulnerability patterns. However, they did not provide the source codes of the MVD model in the published GitHub URL. It is difficult to reproduce the MVD model fully. To compare the performance between our GTCN model and the MVD model, we use three steps to perform MVD experiments. First, we use the data process method provided by MVD to obtain PDG with additional call relations and return values. Second, we use Doc2Vec to embed the code statements. Third, we use GGNN + convolution + softmax activation function to perform vulnerability detection. The learning rate is 0.0001, and this model sets the dropout rate as 0.3. The embedding size of a node is set as 128.

DeepWukong [16]: DeepWukong designed the conv-pool block to learn code embedding for vulnerability detection. The conv-pool block included a graph convolutional layer (GCN, GAT, k-GNN) and a graph pooling layer (top-k). This model extracts XFG (a subgraph of the PDG) edges and code tokens to input into the model, which preserves control-flow and data-flow information together with the natural language information of a program. The batch size is 64. The learning rate is set as 0.002, and this model sets the decay rate as 0.95. The embedding size of a node is set as 128.

6.3.2 Results. Table 15 shows results of the vulnerability identification. GTCN performs better than Devign, TensorGCN, IVDetect, MVD, and DeepWukong for five different datasets. For different CWE datasets, the accuracy of our model is higher than that of Devign by 23%, than that of TensorGCN by 6%, than that of IVDetect by 39%, than that of MVD by 30%, and than that of DeepWukong by 2%, on average. The F1 of our model is more than that of Devign by 24%, than that of TensorGCN by 8%, than that of IVDetect by 36%, than that of MVD by 20%, and than that of DeepWukong by 4%, on average. IVDetect performs worse. DeepWukong has a close performance of vulnerability detection with our model. IVDetect is a too-complex model for vulnerability detection, which does not perform well using our datasets. MVD model is a type-specific vulnerability detection method.

Table 15. Evaluation for Vulnerability Detection (Embedding Size = 512)

Method / Dataset	IVDetect				MVD			
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1
CWE-119	0.555	0.216	0.512	0.299	0.451	0.766	0.309	0.472
CWE-120	0.549	0.341	0.639	0.533	0.542	0.741	0.469	0.637
CWE-476	0.549	0.188	0.500	0.273	0.751	0.798	0.419	0.585
CWE-469	0.553	0.418	0.507	0.467	0.775	0.438	0.458	0.448
Composite	0.556	0.576	0.554	0.565	0.683	0.967	0.672	0.803
Average	0.552	0.348	0.542	0.427	0.640	0.742	0.465	0.589
Method / Dataset	DeepWukong				Devign			
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1
CWE-119	0.915	0.752	0.712	0.728	0.785	0.655	0.441	0.527
CWE-120	0.912	0.727	0.434	0.584	0.753	0.434	0.533	0.481
CWE-476	0.904	0.638	0.714	0.663	0.719	0.710	0.426	0.533
CWE-469	0.941	0.838	0.815	0.826	0.779	0.488	0.570	0.526
Composite	0.938	0.902	0.913	0.908	0.525	0.776	0.533	0.664
Average	0.922	0.771	0.717	0.742	0.712	0.613	0.501	0.546
Method / Dataset	TensorGCN				**GTCN**			
	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1
CWE-119	0.860	0.678	0.735	0.706	0.905	0.775	0.691	0.731
CWE-120	0.848	0.800	0.819	0.810	0.925	0.748	0.730	0.739
CWE-476	0.914	0.678	0.765	0.761	0.927	0.583	0.714	0.642
CWE-469	0.886	0.242	0.492	0.324	0.964	0.896	0.882	0.889
Composite	0.887	0.893	0.904	0.898	0.968	0.908	0.937	0.922
Average	0.879	0.674	0.743	0.700	**0.938**	**0.782**	**0.791**	**0.785**

Table 16. Evaluation for the Vulnerability Detection Task among Different Embedding Sizes

Embedding Size	CWE-469 datasets			
	Accuracy	Recall	Precision	F1
64	0.812	0.767	0.731	0.749
128	0.835	0.783	0. 764	0.773
256	**0.968**	0.839	0.835	0.837
512	0.964	**0.896**	**0.882**	**0.889**

(1) Impact of the embedding sizes. We measure the impact of the embedding sizes on accuracy using the CWE-469. For our model, we consider different embedding sizes as 64, 128, 256, and 512. Table 16 reports Top-1, Top-5, accuracy, and F1 among different embedding sizes for the vulnerability detection task. Results show that both the 256 embedding size and the 512 embedding size perform much better than the 64 embedding size and the 128 embedding size. The vulnerability detection task needs more node information.

(2) Impact of each graph type in code graph tensor. Table 17 compares performances of full code graph tensor and other graph combinations. Our full code graph tensor outperforms the other three graph combinations. The accuracy with full code graph tensor is more than others by

Table 17. Evaluation for Vulnerability Detection among Different Graph Combinations

Graph Combination	CWE-469 datasets			
	Accuracy	Recall	Precision	F1
Full Code Tensor	**0.964**	**0.896**	**0.882**	**0.889**
AST+CFG+DDG	0.918	0.673	0.793	0.728
CFG+DDG+NCS	0.917	0.781	0.732	0.755
AST+DDG+NCS	0.896	0.621	0.705	0.659
AST+CFG+NCS	0.905	0.609	0.761	0.675

Table 18. Evaluation for Vulnerability Detection among Different Graph Orders

Graph Orders	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩
Accuracy	0.964	0.970	0.964	0.968	0.968	0.968	0.970	0.969	0.968	0.964
Recall	0.782	0.787	0.782	0.796	0.783	0.784	0.789	0.781	0.784	0.783
Precision	0.791	0.796	0.794	0.791	0.794	0.793	0.798	0.796	0.795	0.791
F1	0.785	0.789	0.787	0.785	0.787	0.789	0.786	0.783	0.785	0.783

5.4%, on average. Results show that CFG and DDG have the greatest influence. Both code token sequence and code logical topology structure are significantly important in learning code semantic embedding and performing accurate vulnerability detection.

(3) Impact of code graph tensors with different graph orders. To evaluate the influence of the order of graphs (AST, CFG, DDG, NCS) in code graph tensor, we use the CWE-469 dataset to perform vulnerability detection among different code graph tensors with different graph orders. Different code graph tensors with different graph orders are shown in Figure 9, which is introduced in the variable misuse detection task. Results of Table 18 show different orders of code graphs have no great influence in the performance of vulnerability detection task, in which ① is used in our model.

6.4 Result Analysis

In this section, we analyze the learning process of GTCN and other state-of-the-art models to discuss the reasons that GTCN performs better. Specifically, there are three reasons:

- Our model captures four types of code graphs, AST, CFG, DDG, and NCS. They represent different semantic structure information of codes. Other state-of-the-art methods only use the AST to extract the code semantic information. Our model exploits more structural information from source codes.
- By utilizing the hierarchical high-dimensional structural information of graph tensor and the code text information contained in an amount of code dataset, our model is able to learn more accurate embeddings from codes and thus can achieve better performance via the interpretable tensor propagation;
- Our model uses the tensor-based neural network. The bilinear form of the tensor product simultaneously increases more powerful connections between the input units and hidden units. This operation improves the performance and expressiveness of our model. Because the tensor product in neural networks are the second-degree polynomial interactions, and the second-degree polynomial interactions can learn more code feature correlation in the deep learning algorithm [70]. The common GGNN architecture and RNN architecture only use first-degree polynomial interactions in the standard dot product. The code graph tensor

Table 19. Model Efficiency

Model	Parameters	Time (s)	Epochs	Model	Parameters	Time (s)	Epochs
Allamanis	275,952	11,928	200	Devign	4,421,192	65,441	90
GCN	4,354,160	12,614	200	Pointer	3,363,401	210,492	8
RGCN	1,340,016	12,198	200	GINN	313,928	11,968	200
GraphCodeBert	124,647,470	370,081	50	Transformer-XL	1,489,406	101,036	8
MVD	390,158	26,256	90	DeepWukong	3,280,387	55,794	50
IVDetect	426,624	39,584	100	TensorGCN	6,563,881	20,537	200
GTN	622,704	12,330	200	GTCN	309,232	12,686	200

helps our model capture more information from the code semantic graph and thus improves the model's performance.

6.5 Model Efficiency

For analyzing the model efficiency, we count the number of parameters and compare the training time of GTCN with other methods. We train all tasks for a maximum of 300 epochs using Adam with a learning rate of 0.001 and early stopping with a window size of 10, i.e., we stop training if the validation loss does not decrease for 10 consecutive epochs.

Table 19 shows the training time and the number of parameters for all baseline models. Specifically, we train about 230 epochs for the source code prediction task, about 150 epochs for the variable misuse detection task, and about 210 epochs for the vulnerability detection task. The 200 epochs are average training epochs for three tasks. As shown in Table 19, we can see that "Pointer" trains 8 epochs when it converges, "GraphCodebert" trains 50 epochs when it converges, TensorGCN trains 200 epochs when it converges, and "Transformer-XL" trained 8 epochs when it converges.

Results show that GTCN uses less trainable parameters but with better performance. GraphCodeBert is a large complex model that needs to cost a lot of training time to train a large of parameters. The number of parameters in GraphCodebert is more than 400 times that of our model, and the total training time is 29 times that of our model.

The number of parameters in Transformer-XL and Pointer is about $5-10$ times that of our model, and the training time is $9-18$ times that of our model. Pointer and Transformer-XL perform many recurrent computations during the updating process of node representation in the hidden layer; while in our model, we use the code graph tensor structure, the representations of each function are computed relying on the graph tensor convolution layers, and the recurrence only happens in graphs. The tensor operation allows for substantially parallelized multiply operation, which decreases training time.

6.6 Interpretability Experiments

We use the INVISIBLETEST C# dataset with misused variables and the same experimental setting to perform the evaluation of the interpretation model. The idea is that if a sub-graph is removed from our code graph tensor, and the model prediction is affected, then this sub-graph is crucial and must be included in the interpretation for the detection result. We use the relaxed graph mask method to evaluate the interpretability performance. Thus, the minimal sub-graph in our code graph tensor is a kind of entire logical code graph, i.e., the crucial token sequences, data dependencies, and control dependencies of a function, that are most decisive/relevant to the detected variables when the variable is misused. Our model improves the graph convolution neural network architecture by integrating highly efficient tensor computation for better learning code semantic embedding.

Table 20. Evaluation for the Impact of Attributions in Variable Misuse Detection Task Using INVISIBLETEST C# Dataset

	AST	CFG	DDG	NCS	AST +CFG	AST +DDG	AST +NCS	CFG +NCS
Top-1	0.757	0.739	0.801	0.696	0.784	0.807	0.696	0.696
Top-5	0.898	0.898	0.898	0.898	0.913	0.926	0.913	0.913
Accuracy	0.772	0.759	0.817	0.758	0.795	0.852	0.801	0.804
F1	0.768	0.748	0.809	0.748	0.770	0.839	0.769	0.771
	CFG +DDG	DDG +NCS	AST +CFG +NCS	AST +CFG +DDG	AST +DDG +NCS	CFG +DDG +NCS	full code graph tensor	
Top-1	0.750	0.754	0.696	0.867	0.766	0.765	0.875	
Top-5	0.924	0.928	0.926	0.964	0.923	0.923	0.974	
Accuracy	0.871	0.888	0.838	0.904	0.872	0.856	0.940	
F1	0.838	0.857	0.784	0.852	0.817	0.815	0.869	

We first build a base model with only a single attribution, such as **abstract syntax structure (AST)**. We then build other variants of our model by gradually adding one more attribution in other heterogeneous code graphs to the base model including **data dependencies (DDG), control flow dependencies (CFG)**, and the sequence of code tokens (NCS). We measure the accuracy for each variant in the variable misuse detection task. If one sub-graph (a single attribution) is added to the tensor to improve the accuracy of variable misuse detection, then we consider the interpretable information propagation of this attribution effective. The rationale is that if the code graph tensor contains some attributions relevant to the added attributions to fix the misused variables, then that interpretation is useful in pointing out the code relevant to the misused variable detection.

Table 20 shows the changes to the metrics as we incrementally add each attribution (AST, CFG, DDG, NCS) into our model. Generally, the interpretable tensor propagation between attributions contributes positively to the better performance of GTCN. We use the top-1 recall, top-5 recall, accuracy, and F1 to measure the performance. The global attribution (full code graph tensor) performs best on the variable misuse detection task. When GTCN considers only single attribution, we can see that NCS has a small impact, and DDG has a great impact. When we use a basic code syntax tree, the top-1 recall of variable misuse detection is 75.7%. Top-1 relatively improves 2.7% when CFG is additionally considered in AST. The model can distinguish control dependencies of codes. Top-1 relatively improves 6% when DDG is additionally considered in AST+CFG. The model can distinguish data dependencies of codes, because AST and CFG do not help much discriminate the information of data transmission. Top-1 relatively improves 0.8% when NCS is additionally considered in AST+CFG+DDG, and this feature allows the model to detect similar incorrect variables in code sequences.

Table 21 gives some misused variables in specific functions that are correctly detected and fixed using global attribution (full code graph tensor) and can not be detected by partial graph tensor (two attributions or three attributions).

As shown in Tables 5, 12, and 17, results show that the information propagation into the code graph tensor is effective, because it finds the misuse code token and gives respective fixes for specific functions. We compare the performance on three tasks using the entire tensor and that of

Table 21. Some Tokens Are Correctly Detected and Fixed by Global Attribution in the Variable Misuse Detection Task

Filename	slot_location	token	Filename	slot_location	token
TypeMapPlanBuilder.cs	1,657	Source	Client.cs	1,136	connection
Internationalization.cs	226	location	ServiceBusConnection.cs	1,298	TopicIndex
Client.cs	1,123	connection	Cache.cs	783	PollStatus
Cache.cs	1,456	LastPoll	TypeMapPlanBuilder.cs	1,009	_destination
TestItem.cs	176	i	Cache.cs	194	force
MappingExpression.cs	3,746	reverseTypeMap	Cache.cs	630	sourceFilePath
MappingExpression.cs	3,388	x	TypeMapPlanBuilder.cs	2,941	nullCheckedExpression
QueryMapperVisitor.cs	398	left	ServiceBusConnection.cs	163	_trace
TypeMapPlanBuilder.cs	321	customExpression	Wmi.cs	346	_remoteOptions
Mapper.cs	150	Configuration	ArraySegmentTextReader.cs	117	bytesCount
TypeMapPlanBuilder.cs	2,357	destValueExpr	Mapper.cs	1,424	sourceType

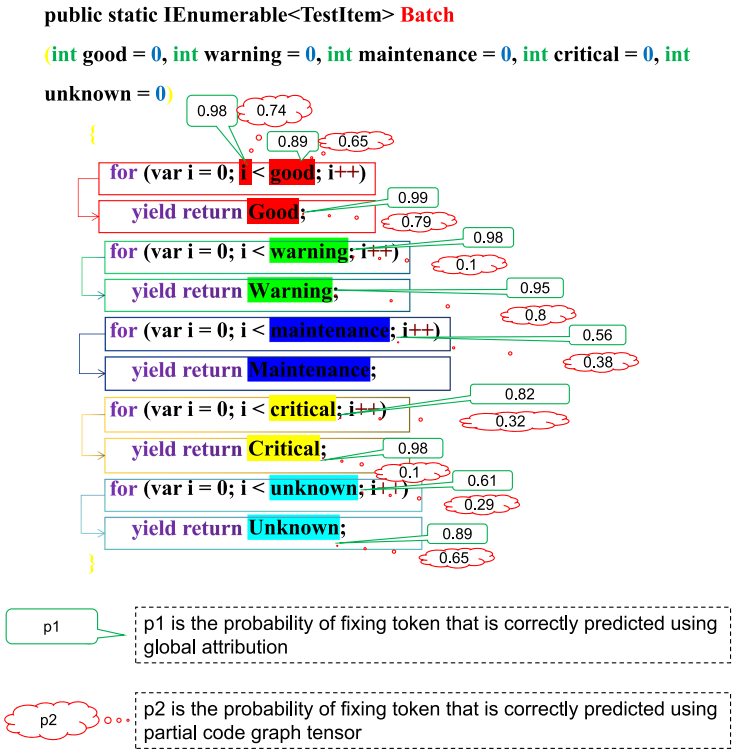

Fig. 12. The top-1 probabilities of some fixing tokens that are correctly predicted in "Batch(...)" function.

using a partial tensor. Our model encourages filters to learn consistent and exclusive patterns (e.g., heterogeneous code graphs and features of code nodes).

Figure 12 shows a function in variable misuse detection task: *Batch*(...), in which "boxed" variables are fixing variables that are correctly predicted. We use two models for observing the top-1 probability of the fixing token that is correctly predicted in the candidate list. One model uses a full code graph tensor and another uses a partial code graph tensor with two attributions. The tensor propagation in our code graph tensor includes the syntax structure of codes, the control/data dependencies, and code sequences between contexts of boxed variables to help detect and fix this variable with high probability. Generally, the improvement in accuracy shows the positive contributions of tensor propagation between all code graphs. Our model with a full code graph tensor

Table 22. Tensor Performance of Source Code Prediction

Model	standard GCN	tensorGCN	GN	GTCN
Top-1	0.571	0.601	0.595	0.761

gives the correct prediction with high probability. The code graph tensor with two attributions can provide the correct fixing with low probability.

Moreover, to better show the impact on interpretable tensor propagation of our GTCN model, we compare GTCN with other GCN-related methods. We combine our GTCN model with the standard GCN model, existing tensorGCN method [54], and **graph normalization (GN)**. The existing tensorGCN method uses the first-dimension and the second-dimension tensor slices to learn the node embedding. GN takes the features of all nodes in a graph into account [13], which normalizes the hidden representations across nodes. We use the Python dataset with large code graphs to perform the experiments.

From Table 22, we can see that our GTCN has a great effect on source code prediction empirically. Tensor computation and global attribution learn more node context information from codes to predict the next token. Existing tensorGCN directly performs standard GCN for each frontal slice of the tensor and does not consider the relationship between all frontal slices (thus all graphs). They fail to utilize the tensor structure effectively.

Table 15 compares the performance of vulnerability detection between our model and the existing interpretable IVDetect model. Our model improves over IVDetect by 39% in accuracy, respectively. Higher accuracy indicates that GTCN can provide better vulnerability detection interpretation. In our model, the code graph tensor provides more than one propagation path from one node to another. The weight for a node is the weight of the tensor computation through multiple propagation paths. The computation time of IVDetect is three times that of our GTCN model (see Table 19).

7 RELATED WORKS

Most recent fields of source code embedding use sequence-based and graph-based deep learning.

Sequence-based deep learning methods for code embedding. Bichsel et al. [11] modeled various relationships between variables using different conditional random fields, elements, and types of AST. Nguyen [58] used an abstract summarization method to generate code names based on the code tokens, which were collected from the program entity names from method parameter types. Karampatsis et al. [37] proposed a subword model to obtain code embedding based on the machine translation method of Sennrich et al. [67]. Rahman [62] used a sequence neural network (LSTM) and added an attention mechanism in LSTM model for code completion. This method was also used to detect code errors at the predicted locations. However, the disadvantage of this method was that it only captured the source code text information and did not capitalize on the well-defined program structure information.

Graph-based deep learning methods for code embedding. Most existing code embedding methods adopted various graph neural networks to predict the code token [49, 68]. Nguyen [56] developed a graph-based statistical model, which used a much more complex subgraph statistical algorithm to perform the API suggestion. This method needed to generate a corpus of code graphs to compute the appearance probability of a target graph given a set of graph contexts. This operation cost too much time. Allamanis et al. [8] learned distributed representations of variables to predict variable names and detect variable misuse using all usages of the variable. Raychev et al. [12] proposed a probabilistic model using the decision tree algorithm and domain-specific grammar detection. Liu [51] proposed a method to represent the hierarchical program structure

information and used the Transformer-XL language model to obtain the long-range dependencies of the sequential code data. Li et al. [46] proposed a prediction model of the source code defect based on DP-CNN model. They treated AST as a series of token vectors and used CNN to train the model. However, they left out the semantic information of other code graphs, such as CFG, DDG.

8 CONCLUSION

In this article, we have proposed a tensor-based graph neural network for achieving code embedding via constructing a code graph tensor from various code contexts. To solve the OoV problem, we have designed a composited output layer to predict the next codes from the pre-trained vocabulary or local candidates using code embedding, which is learned through the proposed graph tensor convolution neural network model. This model efficiently combines the tensor computation and graph convolution neural network. We evaluate model effectiveness in both the source code prediction task, the variable misuse detection task, and vulnerability detection. Results show that the top-1 recall of variable misuse detection is 94.6%, which achieves 18% higher than other state-of-the-art methods, on average. The top-1 recall of source code prediction is 76.1%, and our model achieves at least 13% higher scores with respect to F1 than other state-of-the-art methods. The top-1 accuracy of vulnerable detection is 93.8%, on average.

REFERENCES

[1] Yaqin Zhou. 2019. Source codes of the paper: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. https://github.com/epicosy/devign.

[2] Yu Wang. 2020. Source codes of the paper: Learning semantic program embeddings with graph interval neural network. https://github.com/GINN-Imp/GINN.

[3] Vincent J. Hellendoorn. 2020. Source codes of the paper: Global relational models of source code. https://github.com/VHellendoorn/ICLR20-Great.

[4] Zhangyin Feng. 2021. Source codes of the paper: CodeBERT: A pre-trained model for programming and natural languages. https://github.com/microsoft/CodeBERT.

[5] Jia Yang. 2022. Source codes of this paper. https://gitee.com/cse-sss/GTCN.

[6] Jia Yang. 2022. Source codes of this paper. https://github.com/SmileResearch/GTCN.

[7] Yi Li. 2021. Source codes of the paper: Vulnerability detection with fine-grained interpretations. https://github.com/vulnerabilitydetection/VulnerabilityDetectionResearch.

[8] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *6th International Conference on Learning Representations*.

[9] Gareth Ari Aye and Gail E. Kaiser. 2020. Sequence model design for code completion in the modern IDE. *CoRR* abs/2004.05249 (2020).

[10] Richard G. Baraniuk. 2011. More is less: Signal processing and the data deluge. *Science* 331(6018) (2011), 717–719.

[11] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. 2016. Statistical deobfuscation of Android applications. In *ACM SIGSAC Conference on Computer and Communications Security*. 343–355.

[12] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic model for code. In *33rd International Conference on Machine Learning*. 2933–2942.

[13] Tianle Cai, Shengjie Luo, and Keyulu Xu. 2021. GraphNorm: A principled approach to accelerating graph neural network training. In *38th International Conference on Machine Learning*. 1204–1215.

[14] Sicong Cao, Xiaobing Sun, Lili Bo et al. 2022. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In *44th IEEE/ACM 44th International Conference on Software Engineering*. ACM, 1456–1468.

[15] Lei Cheng and Qingjiang Shi. 2021. Towards overfitting avoidance: Tuning-free tensor-aided multi-user channel estimation for 3D massive MIMO communications. *IEEE J. Sel. Top. Sig. Process.* 15, 3 (2021), 832–846.

[16] Xiao Cheng, Haoyu Wang, Jiayi Hua et al. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (2021), 38:1–38:33.

[17] Vladimir Cherkassky. 1997. The nature of statistical learning theory. *IEEE Trans. Neural Netw.* 8, 6 (1997), 1564.

[18] Andrzej Cichocki and Danilo P. Mandic. 2015. Tensor decompositions for signal processing applications: From two-way to multiway component analysis. *IEEE Sig. Process. Mag.* 32, 2 (2015), 145–163.

[19] Zhangyin Feng, Daya Guo, Duyu Tang et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16–20 November 2020 (Findings of ACL)*, Vol. EMNLP 2020. Association for Computational Linguistics, 1536–1547.

[20] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *7th International Conference on Learning Representations*. OpenReview.net.
[21] Gauthier Gidel, Tony Jebara, and Simon Lacoste-Julien. 2017. Frank-Wolfe algorithms for Saddle point problems. In *20th International Conference on Artificial Intelligence and Statistics AISTATS*. 362–371.
[22] Martin Grohe. 2020. word2vec, node2vec, graph2vec, X2vec: Towards a theory of vector embeddings of structured data. In *39th ACM Symposium on Principles of Database Systems*. 1–16.
[23] Daya Guo, Shuo Ren, and Shuai Lu. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *9th International Conference on Learning Representations*. OpenReview.net.
[24] Xiawei Guo, Quanming Yao, and James Tin-Yau Kwok. 2017. Efficient Sparse low-rank tensor completion using the Frank-Wolfe algorithm. In *31st AAAI Conference on Artificial Intelligence*. 1948–1954.
[25] Kaisei Hanayama, Shinsuke Matsumoto, and Shinji Kusumoto. 2020. Humpback: Code completion system for Dockerfile based on language models (short paper). In *Joint Proceedings of SEED & NLPaSE co-located with 27th Asia Pacific Software Engineering Conference*. 67–73.
[26] Douglas M. Hawkins. 2004. The problem of overfitting. *J. Chem. Inf. Model.* 44, 1 (2004), 1–12.
[27] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *8th International Conference on Learning Representations*. OpenReview.net.
[28] Abram Hindle, Earl T. Barr, Zhendong Su et al. 2012. On the naturalness of software. In *34th International Conference on Software Engineering*. 837–847.
[29] Illia Horenko. 2020. On a scalable entropic breaching of the overfitting barrier for small data problems in machine learning. *Neural Comput.* 32, 8 (2020), 1563–1579.
[30] Ruizhe Huang, Ke Li, and Ashish Arora. 2020. Efficient MDI adaptation for n-gram language models. In *21st Annual Conference of the International Speech Communication Association*. 4916–4920.
[31] Zhichao Huang, Xutao Li, Yunming Ye, and Michael K. Ng. 2020. MR-GCN: Multi-relational graph convolutional networks based on generalized tensor product. In *29th International Joint Conference on Artificial Intelligence*. 1258–1264.
[32] Martin Jaggi. 2013. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *30th International Conference on Machine Learning*. 427–435.
[33] Prateek Jain, Om Thakkar, and Abhradeep Thakurta. 2017. Differentially private matrix completion, revisited. *CoRR* abs/1712.09765 (2017).
[34] Jayadeva, Mayank Sharma, Sumit Soman, and Himanshu Pant. 2018. Ultra-sparse classifiers through minimizing the VC dimension in the empirical feature space—submitted to the special issue on "off the mainstream: Advances in neural networks and machine learning for pattern recognition." *Neural Process. Lett.* 48, 2 (2018), 881–913.
[35] Ante Jukic and Marko Filipovic. 2013. Supervised feature extraction for tensor objects based on maximization of mutual information. *Pattern Recognit. Lett.* 34, 13 (2013), 1476–1484.
[36] Ibrahim Kajo, Nidal S. Kamel, and Yassine Ruichek. 2020. Self-motion-assisted tensor completion method for background initialization in complex video sequences. *IEEE Trans. Image Process.* 29 (2020), 1915–1928.
[37] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code != big vocabulary: Open-vocabulary models for source code. In *42nd International Conference on Software Engineering*. 1073–1085.
[38] Dominic Kempf, René Heß, Steffen Müthing, and Peter Bastian. 2021. Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. *ACM Trans. Math. Softw.* 47, 1 (2021), 6:1–6:31.
[39] Utkarsh Mahadeo Khaire and R. Dhanalakshmi. 2020. High-dimensional microarray dataset classification using an improved adam optimizer (iAdam). *J. Amb. Intell. Humaniz. Comput.* 11, 11 (2020), 5187–5204.
[40] Behnoush Khavari and Guillaume Rabusseau. 2021. Lower and upper bounds on the VC-dimension of tensor network models. *CoRR* abs/2106.11827 (2021).
[41] Misha E. Kilmer and Carla D. Martin. 2011. Factorization strategies for third-order tensors. *Linear Algeb. Applic.* 435, 3 (2011), 641–658.
[42] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *43rd IEEE/ACM International Conference on Software Engineering*. 150–162.
[43] Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations*. OpenReview.net.
[44] Bor-Chen Kuo and David A. Landgrebe. 2002. A covariance estimator for small sample size classification problems and its application to feature extraction. *IEEE Trans. Geosci. Rem. Sensor* 40, 4 (2002), 814–819.
[45] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *27th International Joint Conference on Artificial Intelligence*. 4159–4165.

[46] Xiang Li, Kefan Qiu, Cheng Qian, and Gang Zhao. 2020. An adversarial machine learning method based on OpCode N-grams feature in malware detection. In *5th IEEE International Conference on Data Science in Cyberspace*. 380–387.
[47] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated graph sequence neural networks. In *4th International Conference on Learning Representations*.
[48] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 292–303.
[49] Zhao Li, Yuying Xing, Jiaming Huang et al. 2021. Large-scale online multi-view graph neural network and applications. *Fut. Gen. Comput. Syst.* 116 (2021), 145–155.
[50] Zhang Li-mei, Qiao Li-shan, and Chen Song-can. 2009. A survey of feature extraction and classifier design based on tensor pattern. *J. Shandong Univ. (Eng. Sci.)* 39, 1 (2009), 6–14.
[51] Fang Liu, Ge Li, and Bolin Wei. 2020. A self-attentional neural architecture for code completion with multi-task learning. In *28th International Conference on Program Comprehension*. 37–47.
[52] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.
[53] Risheng Liu, Zhouchen Lin, Zhixun Su, and Kewei Tang. 2010. Feature extraction by learning Lorentzian metric tensor and its extensions. *Pattern Recognit.* 43, 10 (2010), 3298–3306.
[54] Xien Liu, Xinxin You, and Xiao Zhang. 2020. Tensor graph convolutional networks for text classification. In *34th AAAI Conference on Artificial Intelligence*. 8409–8416.
[55] Johannes C. Myburgh, Coenraad Mouton, and Marelie H. Davel. 2021. Tracking translation invariance in CNNs. *CoRR* abs/2104.05997 (2021).
[56] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based statistical language model for code. In *37th IEEE/ACM International Conference on Software Engineering*. IEEE Computer Society, 858–868.
[57] Anh Tuan Nguyen, Tung Thanh Nguyen, and Hoan Anh Nguyen. 2012. Graph-based pattern-oriented, context-sensitive source code completion. In *34th International Conference on Software Engineering*. IEEE Computer Society, 69–79.
[58] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *42nd International Conference on Software Engineering*. 1372–1384.
[59] Vijay Pandey. 2020. Overcoming overfitting and large weight update problem in linear rectifiers: Thresholded exponential rectified linear units. *CoRR* abs/2006.02797 (2020).
[60] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2010. Detection of recurring software vulnerabilities. In *25th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 447–456.
[61] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. 2021. secureTF: A secure TensorFlow framework. *CoRR* abs/2101.08204 (2021).
[62] Md. Mostafizer Rahman, Yutaka Watanobe, and Keita Nakamura. 2020. A neural network based intelligent support model for program code completion. *Sci. Program.* 2020, 7426461 (2020), 1–18.
[63] Avinash Ratre and Vinod Pankajakshan. 2018. Tucker tensor decomposition-based tracking and Gaussian mixture model for anomaly localisation and detection in surveillance videos. *IET Comput. Vis.* 12, 6 (2018), 933–940.
[64] Myroslava Romaniuk. 2020. N-gram models for code completion in Pharo. In *4th International Conference on the Art, Science, and Engineering of Programming*. 227–228.
[65] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton et al. 2018. Automated vulnerability detection in source code using deep representation learning. In *17th IEEE International Conference on Machine Learning and Applications*. IEEE, 757–762.
[66] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem et al. 2018. Modeling relational data with graph convolutional networks. In *The Semantic Web—15th International Conference, ESWC 2018, Heraklion, Crete, Greece (Lecture Notes in Computer Science)*, Vol. 10843. Springer, 593–607.
[67] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *54th Annual Meeting of the Association for Computational Linguistics, ACL*.
[68] Penghao Sun, Julong Lan, Junfei Li et al. 2021. Combining deep reinforcement learning with graph neural networks for optimal VNF placement. *IEEE Commun. Lett.* 25, 1 (2021), 176–180.
[69] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. In *25th ACM International Conference on Knowledge Discovery & Data Mining*. 2727–2735.
[70] Andros Tjandra, Sakriani Sakti, and Ruli Manurung. 2016. Gated recurrent neural tensor network. In *International Joint Conference on Neural Networks*. 448–455.
[71] S. Waner. 1986. Introduction to differential geometry and general relativity. Lecture Notes by Stefan Waner, with a Special Guest Lecture by Gregory C. Levine, Department of Mathematics, Hofstra University, https://medusa.teodesian.net/docs/mathematics/Intro%20to%20Differential%20Geometry%20and%20General%20Relativity%20-%20S.%20Warner%20(2002)%20WW.pdf.

[72] Huanting Wang, Guixin Ye, and Zhanyong Tang. 2021. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Trans. Inf. Forens. Secur.* 16 (2021), 1943–1958.
[73] Wenhan Wang, Kechi Zhang, Ge Li, and Zhi Jin. 2020. Learning to Represent Programs with Heterogeneous Graphs. (2020). arXiv:cs.SE/2012.04188
[74] Yu Wang, Ke Wang, Fengjuan Gao et al. 2020. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 137:1–137:27.
[75] Yingxin Wu, Xiang Wang, An Zhang, Xiangnan He, and Tat-Seng Chua. 2022. Discovering invariant rationales for graph neural networks. In *10th International Conference on Learning Representations*. OpenReview.net.
[76] Le Xu, Lei Cheng, Ngai Wong, and Yik-Chung Wu. 2021. Overfitting avoidance in tensor train factorization and completion: Prior analysis and inference. In *IEEE International Conference on Data Mining*. IEEE, 1439–1444.
[77] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy*. 590–604.
[78] Seongjun Yun, Minbyul Jeong, Raehyun Kim et al. 2019. Graph transformer networks. In *Annual Conference on Neural Information Processing Systems*. 11960–11970.
[79] Wei Zhang, Zhouchen Lin, and Xiaoou Tang. 2009. Tensor linear Laplacian discrimination (TLLD) for feature extraction. *Pattern Recognit.* 42, 9 (2009), 1941–1948.
[80] Yu Zhang, Peter Tiňo, Ales Leonardis, and Ke Tang. 2021. A survey on neural network interpretability. *IEEE Trans. Emerg. Top. Comput. Intell.* 5, 5 (2021), 726–742.
[81] Yaqin Zhou, Shangqing Liu, Jing Kai Siow et al. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Annual Conference on Neural Information Processing Systems*. 10197–10207.

Received 18 July 2022; revised 17 November 2022; accepted 4 January 2023