3-2022

Learning program semantics with code representations: An empirical study

Jing Kai SIOW

Shangqing LIU

Xiaofei XIE
Singapore Management University, xfxie@smu.edu.sg

Guozhu MENG

Yang LIU

Learning Program Semantics with Code Representations: An Empirical Study

Jing Kai Siow[1*] Shangqing Liu[1*] Xiaofei Xie[2] Guozhu Meng[3] Yang Liu[4,1†]
[1]Nanyang Technological University, Singapore
[2]Singapore Management University, Singapore
[3]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[4]School of Information Science and Technology, Zhejiang Sci-Tech University, China
{jingkai001,shangqin001}@e.ntu.edu.sg, xiaofei.xfxie@gmail.com, mengguozhu@iie.ac.cn, yangliu@ntu.edu.sg

***Abstract*—Program semantics learning is the core and fundamental for various code intelligent tasks e.g., vulnerability detection, clone detection. A considerable amount of existing works propose diverse approaches to learn the program semantics for different tasks and these works have achieved state-of-the-art performance. However, currently, a comprehensive and systematic study on evaluating different program representation techniques across diverse tasks is still missed.**

From this starting point, in this paper, we conduct an empirical study to evaluate different program representation techniques. Specifically, we categorize current mainstream code representation techniques into four categories i.e., Feature-based, Sequence-based, Tree-based, and Graph-based program representation technique and evaluate its performance on three diverse and popular code intelligent tasks i.e., Code Classification, Vulnerability Detection, and Clone Detection on the public released benchmark. We further design three research questions (RQs) and conduct a comprehensive analysis to investigate the performance. By the extensive experimental results, we conclude that (1) The graph-based representation is superior to the other selected techniques across these tasks. (2) Compared with the node type information used in tree-based and graph-based representations, the node textual information is more critical to learning the program semantics. (3) Different tasks require the task-specific semantics to achieve their highest performance, however combining various program semantics from different dimensions such as control dependency, data dependency can still produce promising results.

I. Introduction

With the booming development of open-source software, the amount of available code-related data has reached an unprecedented scale, which inspires researchers from both academia and the industry to explore employing data-driven approaches for diverse code-related problems such as type inference [1]–[3], clone detection [4]–[6], source code summarization [7]–[10], code search [11]–[13] and software vulnerability detection [14]–[16]. Most of the existing works attempt to understand the behavior of the program i.e., understand the program semantics by the well-designed approaches for different tasks and have achieved promising results. Typically, we can broadly categorize these data-driven code-related works into four major categories: Feature-based, Sequence-based, Tree-based, and Graph-based representation to learn the program semantics for different tasks.

Feature-based approach requires the domain knowledge extracted from the program by the experts to represent the program semantics for different tasks. For instance, FLUCSS [17] is designed for fault localization. It incorporated Spectrum Based Fault Localisation [18] metric and source code related metrics to identify the fault in the software. It further performed learning-to-rank with Support Vector Machine [19] and Genetic Programming [20]. Their approach achieved a performance of 50% Mean Average Precision (MAP) in ranking software defects. Bhel et al. [21] proposed mining security bugs from bug reports using TF-IDF [22] and Naive Bayes [23] algorithm. They achieved high precision e.g., 92.56% for detecting the bug report.

Sequence-based representation treats code as a flat sequence of tokens [24], [25] and converts them into numerical vectors with the distributed representation [26] and further employs these vectors for diverse tasks. For example, VulDeepecker [27] predicted if a program is vulnerable by learning on code gadgets. Code gadgets are generated by slicing programs through the function calls. These code gadgets are then input into a bi-directional LSTM [28] for learning whether the program is vulnerable. They achieved a score of 85.4% in F1-score across multiple vulnerability types. Source code summarization is another popular application of the sequence-based approach. Hu et al. [29] aimed to translate source code into their respective summary. They employed API sequences and code tokens as the input for an encoder-decoder model to generate the summary. They achieved 41.98 in BLEU-4 [30] and 18.81 in METEOR [31]. Since programs are highly structured data, which can be converted into different structural representations such as the abstract syntax tree (AST). Many works attempt to explore this information hidden in the text for different tasks [32]–[35]. For instance, code2vec [33] predicted method name through the path contexts extracted from the AST of the program and achieved an F1-score of 58.4%. CDLH [35] employed tree-based representation to detect code clones. It employed LSTM in learning an AST-based representation and hashed them to achieve a unique vector on each program. CDLH achieved promising results for Type-3 and Type-4 clone detection.

* Equal Contributions.
† Corresponding author.

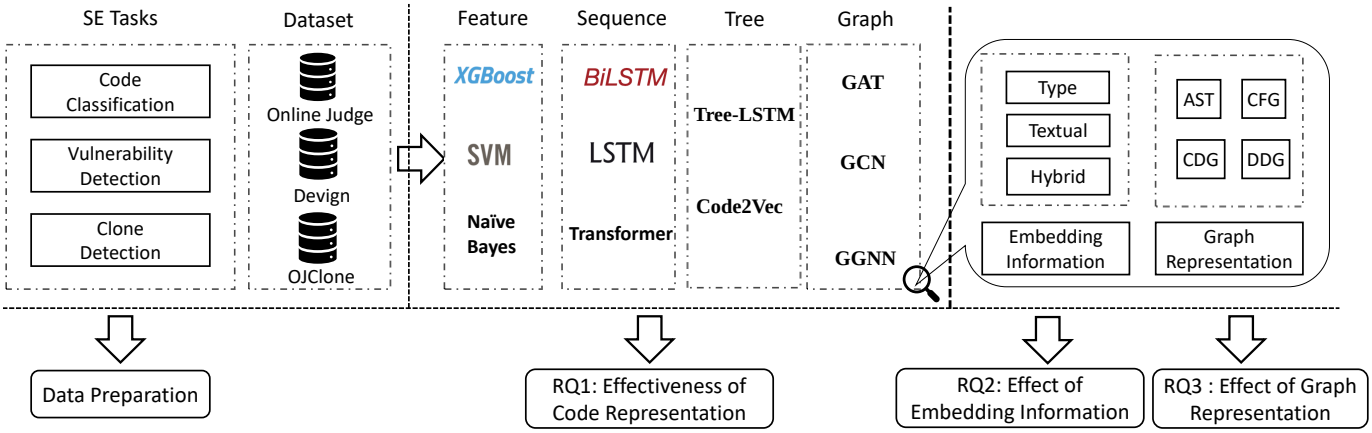


Fig. 1: Overview of our empirical study.

The graph-based approaches attempt to embed more structural information such as data-flow, control-flow rather than the pure AST into a graph [7], [8], [36], [37] to represent the program semantics. For instance, Allamanis et al. [36] targeted at detecting variables that are incorrectly used in a project and predicting the correct variable. They enhanced AST with different data flow information by constructing diverse types of edges among the nodes, such as connecting variable nodes where the variable was last written to, to achieve a graph-based representation on the program. Their approach achieved a high accuracy of 85.5% and 78.2% in finding wrong variable names on the seen and unseen projects respectively.

Despite different program representation techniques being widely utilized in learning program semantics for code-related tasks, currently, there is still a lack of a comprehensive study on evaluating and discussing the impact of different code representations across diverse tasks. Many challenges around code representation are still unsolved: (1) What type of the above code representation is optimal in the program scenario or in other words, whether there is an optimal code representation technique for different tasks? (2) The widely employed tree-based or graph-based approaches have shown superiority, however, these data usually contain complicated structures. For example, each node in AST usually contains node type and node textual information, whether they are both beneficial in learning program semantics? (3) The graph-based representation incorporates diverse program semantics such as data flow, control flow, data dependency, whether each component contributes equally to the final performance? Based on the questions, we aim to answer the following research questions (RQs) as follows:

- **RQ1: Comparison of Feature-based, Sequence-based, Tree-based and Graph-based Representation.**
- **RQ2: Comparison of Node Embedding Information.**
- **RQ3: Efficacy of Different Graph Representations.**

To answer these questions, we design our study on three popular and diverse code intelligent tasks: **Code Classification**, **Vulnerability Detection** and **Clone Detection** with typical approaches for these representations i.e., SVM, Naive Bayes, XGBoost for the feature-based representation; LSTM, BiLSTM, and Transformer for the sequence-based representation; Code2Vec, Tree-LSTM for the tree-based representation and Graph Convolution Neural Network (GCN), Graph Attention Network (GAT) and Gated Graph Neural Network (GGNN) for the graph-based representation. We further utilize the public released dataset for the evaluation and employ Joern [38] for the unification on code property graph (CPG) construction for the graph-based representation. Specifically, CPG contains abstract syntax tree (AST), control flow graph (CFG), control dependency graph (CDG), and data dependency graph (DDG) to facilitate investigating each component. By the extensive experiments (around **1000** GPU hours), we conclude that: (1) Since graph-based representation incorporates diverse program semantics, it outperforms other compared representation techniques by a significant margin among the selected tasks. (2) Both node type and node textual information are beneficial in capturing program semantics, however, the textual information is more critical. (3) Different task relies on the task-specific semantics to achieve the best performance, however, generally, a composite graph representation with the comprehensive program semantics can still produce promising results.

In summary, we make the following contributions:

- To the best of our knowledge, this is the first large-scale empirical study that evaluates different code representation techniques on diverse popular evaluation tasks.
- We conduct comprehensive experiments and analysis to investigate the effect of each component in the tree and graph representation in capturing the program semantics on different tasks.
- We provide extensive discussions from different aspects i.e., the learnt space by the different model, the semantic-preserving operation on the code snippet, and the statistical analysis of the program features on the predicted samples to illustrate the capacity and limitation of different representation techniques. We release our source code and data at https://github.com/jingkai92/

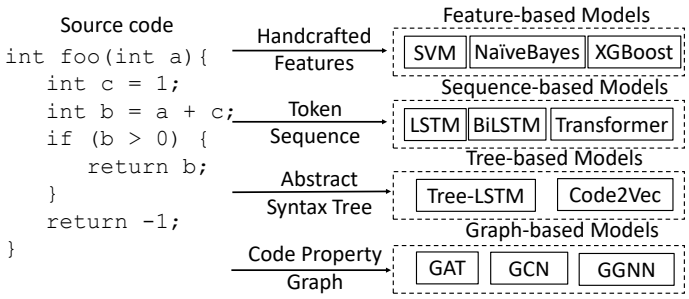


Fig. 2: Techniques on Code Representation.

learning-program-representation for the reproduction.

II. Overview

An overview of our study is shown in Fig. 1. In this study, we aim to answer the effectiveness of different code representations (Section II-B) for different tasks (Section II-A) on the public released benchmark. We categorize current state-of-the-art code representation techniques into feature-based, sequence-based, tree-based, and graph-based code representation and design RQ1-RQ3 (Section III) for the investigation.

A. Evaluation Tasks

We select three diverse tasks i.e., code classification [5], clone detection [5] and vulnerability detection [14] for our study. We selected these tasks based on two reasons: (1) These tasks can effectively evaluate the semantic, lexical, syntactic information that is on the comparing evaluations. For instance, control dependency and data dependency are crucial in detecting vulnerabilities as the data flow of variables can indicate wrongful usage of variables [27]. Lexical information is important for clone detection as programs that uses the same tokens might be more likely to be cloned. A code representation that performs well in all three evaluating tasks infers that the representation can learn better semantic, lexical, and syntactic information. (2) The three evaluating tasks are popular in the software engineering domains. We collected the number of publications, in recent years in top leading conferences, e.g., ASE, ICSE, NeurIPS. We discovered that there are at least two, five, and four publications that are relevant to source code classification, vulnerability detection, and clone detection. This shows that our evaluating tasks are popular and important.

Code Classification. Code classification aims to classify the code fragments by their functionalities. which is vital for program understanding. Given a program, code classification aims to identify which category it belongs to from a category set. We employ OJ dataset [39], a C Programming Language dataset, for code classification. The dataset contains 52,000 programs that are classified into 104 classes and each class performs a high-level operation, such as reversing an integer or finding minimum/maximum words from an array of words.

Clone Detection. Clone detection detects whether two code fragments achieve the same functionality with different implementations. We follow the same approach as Zhang et al. [5] and Wei et al. [35] to generate the code clone dataset based on OJ dataset [39]. Two programs in the same class can be considered as at least a functionality clone as they fulfill the same functionality [35]. Therefore, we gather the clones by pairing the programs that are in the same class and non-clones by pairing the programs that are in different classes. We randomly selected 25,000 programs and pair each of them with a clone and non-clones. We specifically ensure that a program that appears in the training set will not appear in the testing to ensure a fair evaluation. To differentiate from the dataset that we used in code classification, we refer to this dataset as OJClone.

TABLE I: Statistics of Dataset.

Dataset		# of Samples	# of Classes
Online Judge (OJ)	Training	28622	104
	Validation	3581	
	Testing	3628	
Devign	Training	38526	2
	Validation	4815	
	Testing	4817	
OJClone	Training	40000	2
	Validation	5000	
	Testing	5000	

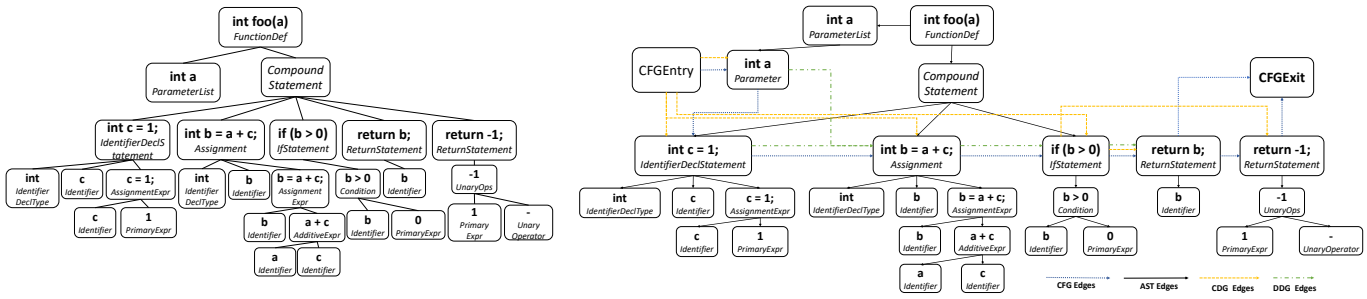
Vulnerability Detection. The goal of vulnerability detection is to detect the vulnerable code fragments that may be attacked for cyber security. We utilize the completed Devign dataset [14], which contains 66,067 labeled functions that are collected from four open-sourced C programming language projects, FFmpeg, Wireshark, Linux, and Qemu for the evaluation. These functions are classified as either vulnerable or non-vulnerable. Zhou. et al. verify security patches manually and extract the functions from these verified patches [14], hence, their reliability is ensured.

We remove the duplicate functions to ensure a fair comparison. Furthermore, we employ Joern [38] to extract the tree and the graph from the programs. However, due to some compilation errors, finally, there is a set of 35,831 programs in our OJ dataset and 48,158 programs in the Devign dataset and the statistics of the datasets are shown in Table I. We split all our datasets into 80% for the training set, 10% for the validation set, and 10% for the testing set.

B. Code Representation

As shown in Fig. 2, to evaluate the efficacy of different code representation, we categorise the code representation into four major categories: Feature-based, Sequence-based, Tree-based and Graph-based Representation.

Feature-based Representation. In our study, we used Term Frequency-Inverse Document Frequency (TF-IDF) to vectorize the code snippet. TF-IDF computes a vector for each program based on the term frequency and inverse document frequency, which reduces the importance for stopwords and increases the weightage for more relevant words. To evaluate feature-based representation, we selected three machine learning algorithms:



(a) Tree-based Representation (AST)

(b) Graph-based Representation (CPG)

Fig. 3: The Illustration of Tree and Graph Constructed by Joern where the original function is from Fig. 2.

Support-Vector Machine (SVM) [19], Naive-Bayes [23] and XGBoost [40]. We use TF-IDF as our feature vector and input these vectors to the models. SVM performs the classification by finding a hyperplane in the dimensional space that can distinctly identify samples in different classes, while Naive-Bayes computes the probability of a sample within a class with an assumption of independent features. XGBoost is an implementation of a gradient boosting machine that is known for scalability and performance. These models are commonly used in many SE tasks as the comparison baselines [5], [14].

Sequence-based Representation. Since code snippets can be treated as a flat sequence of tokens, many works utilize this sequential information directly to learn program semantics. Following the previous works [14], [36], [41], we also employ token sequence directly in evaluating the sequence-based representation of the programs. We further employ sub-word splitting for efficient learning, where words are split by their camelcase and underscore. For instance, a function name, "get_int", is split into two subword tokens, "get" and "int". The result of this sub-word splitting is a sequence of subwords tokens. Their purpose is to reduce the vocabulary size of the datasets. For evaluating sequence representation, we opt to use Long-Short Term Memory (LSTM) [42] and Bi-directional LSTM (BiLSTM) [28] network as they are suitable in training sequential data, i.e., sequence of tokens. For our experiment, we trained a LSTM and Bi-directional LSTM (BiLSTM) model to evaluate the performance on learning source code sequentially. We take the last hidden state of LSTM/BiLSTM as the learnt representation of the sequence representation. Furthermore, we added in transformer encoder [43] as part of our baseline to investigate the impact of the attention-based sequence approach. Transformer employs multi-head attention to learn a representation for a sequence and we utilize the contextual representation produced by the transformer encoder as the learnt program representation.

Tree-based Representation. The program is a highly structured data compared to the sequential data, hence many code-related works attempt to extract the structure information e.g., abstract syntax tree (AST) [5], [33], [35], [36] behind the text to capture the semantics. AST contains the syntactic information for a compiler to generate machine code, while removing unnecessary information, such as comments and whitespaces [44]. An example of AST produced by Joern can be seen in Fig. 3(a). We selected two approaches for evaluating tree-based representations, Tree-LSTM [45] and Code2Vec [33]. Tree-LSTM employs LSTM in learning the network topology of the input tree structure, or in our case, the AST. It computes the hidden states based on its successors. As opposed to the single forget gate used in LSTM, it uses one forget gate for each child to focus on important information and outperforms several sequence-based approaches. Furthermore, several works [32], [46] extend or employ Tree-LSTM as its baselines in software engineering. In our study, we employ the Child-Sum Tree-LSTM where the network learns the hidden states based on the summation of the children states. We employed max-pooling over all node representations to achieve the tree-based representation vector. Code2Vec [33] is a state-of-the-art technique in code representation. It represents programs in a bag of path context, where path context represents a path between terminal nodes across the AST. Code representation is learnt through focusing on these path contexts. We extracted the path contexts from the tool ASTMiner [47] and adapted the source code given by Alon et al. [33] for the evaluating tasks.

Graph-based Representation. Many types of graphs are associated with programs, such as control flow graph (CFG), control dependency graph (CDG), data dependency graph (DDG).Control flow graph depicts the execution flow of the statements in a program. Data dependency occurs when the value of a variable in a statement depends on the execution of the previous statement, whereas control dependency happens when the branching result of a predicate determines the execution of the immediate statements. Existing works utilized these information for various tasks [7]–[9], [14]. To investigate the impact of different types of graphs, we employ Code Property Graph (CPG) which is proposed by Yamaguchi et al. [48] by combining several graph representations, e.g., AST, CFG, CDG, and DDG, of source code into a graph structure. We did not consider other graph structures such as Allamanis et al [9], [36] to represent program semantics as we follow the previous works [7], [14]. We selected three popular and widely used GNN variants i.e., Graph Convolutional Network (GCN), Graph Attention Network (GAT), and Gated Graph Neural Network (GGNN) to be our evaluating GNNs. These networks

differ in their node message propagation, i.e., they aggregate and propagate information across the graph differently. GCN [49] uses first-order approximation of ChebNet [50]. The neighboring information of a node is aggregated using convolutional operation and layers of networks can be stacked to enhance the learning of node features. Velickovic et al. [51] propose GAT to use an attention mechanism in GNNs to attend over the neighborhood of a node to capture the local neighborhood information. GGNN [52] aggregates node information by Gated Recurrent Unit [53] at every iteration to learn the node representations. For above GNN variants, to obtain the graph-level representation, which can be considered as the program representation for different tasks, we employ the max-pooling operation over the learnt node representations.

Node Representation in Tree/Graph. To obtain the tree structure or the graph structure for a program, we use Joern [38], a tool that is widely used academically [48] and commercially [38], to transform a function. A simple example of the source code in Fig. 2 is presented in Fig. 3, where Fig. 3 (a) is the tree representation and Fig. 3 (b) is the graph representation. We can find that each node has its node type, which distinguishes the node from the others (the second line in each node), and code textual information, which is a small fragment code snippet from the original program (the first line in each node). To get the initial node representation of the graph and tree, which is a vector uniquely to represent each node for the model learning, we employ three different embedding methods: type embedding, textual embedding and hybrid embedding. In type embedding, we embed each node solely by its node type, i.e., embed the node by the unique vectors that represent each type of node. Each node type is input into a linear layer to learn a unique representation. Textual embedding learns an intermediate representation of the node by inputting the textual information of the node into a BiLSTM. We use the last hidden state of the BiLSTM as our initial node representation for the embedding. For the node without textual information such as the compound statement node, we use the empty string for the replacement and feed the empty string into BiLSTM to get the representation. Hybrid embedding employs a linear layer to learn the concatenated representation of textual embedding and type embedding. Then these embedded node representations are utilized with Tree-LSTM or GNNs respectively to learn the representations.

III. Empirical Study

In this section, we detail our experimental settings to ensure transparency in our experiments. We then answer the proposed RQ1-RQ3 with a comprehensive analysis.

A. Experiment Settings

Experimental Setup. We embed each function into TF-IDF feature vector and input these vectors into the SVM, Naive Bayes, and XGBoost, resulting in a list of class probabilities. The class with the highest probability will be our prediction for the model. We employ Multinomial Naive Bayes [23] and Radial Basis Function Kernel for SVM [19]. For XGBoost [40], we trained our model with 40 rounds with a tree-depth of 32. We used the following hyper-parameters for LSTM, BiLSTM, and Transformer network: 128 for word embedding dimension and LSTM hidden size, 4 layers of LSTM/BiLSTM and transformer encoder, 4 heads for transformer attention layer. We used DGL [54] as our implementation for Tree-based and Graph-based approaches. Similarly, we used a hidden dimension of 128. For GCN, GAT, and GGNN, the following hyperparameters are used: 128 for word embedding and initial node representation, 4 layers for GAT/GCN/GGNN, and 4 attention heads for GAT. We train all our baselines with training data and tune the models based on the validation data. We then report the performance of the models using the testing data. To ensure fairness in the evaluation, we fixed all the dimensions to be 128 and uses a learning rate between 0.01 to 0.0001 for different tasks, batch size of 128, and a dropout rate of 0.2. All models are trained until 50 epoch and early stopping of 10 epoch. We limit the number of tokens in the function to be 150 and restrict the size of the graph to be within 250 nodes. For all deep learning experiments, we employ an additional linear classifier to learn on classifying the learnt representation into their respective classes. All experiments are conducted on an Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz Linux 16.04 server with 128GB RAM and equipped with three Tesla V100-SXM2-32GB graphic cards.

Evaluation Metrics. We adopt accuracy and F1 as the evaluating metrics. Accuracy computes the correct prediction of each class and averages it with the total number of samples. F1 is commonly used in binary classification [5], [14], [35]. It is computed using a weighted combination of precision and recall. A high F1 implies the model has a low number of false positives and false negatives. The higher values of these metrics, the better performance the approach achieves.

B. RQ1: Comparison of Different Code Representations

Table II shows the comparison results of our experiments. For this RQ, we employed hybrid embedding for tree representation and graph representation to learn the programs. We observed that among all the feature-based models, XGBoost performs the best across all three tasks, in contrast, Naive Bayes has the worst performance since the dependencies among the tokens in the program cannot be captured.

From the third row of Table II, we can find that the sequence-based models have a better performance as compared to the feature-based approaches and BiLSTM performs the best across all three tasks, having accuracy in 83.82% in Code Classification, and F1-Score of 71.62% in Vulnerability Detection and accuracy of 85.44% in Clone Detection. This indicates that learning dependency among the tokens in the program is important for learning program semantics and these tokens have the rich semantic information to represent the programs, which yields better performance compared with the feature-based approaches.

The tree-based approaches such as Tree-LSTM has the better performance over these tasks compared with the sequence-based approaches. Specifically, Tree-LSTM has an improve-

TABLE II: Results of Code Classification, Vulnerability Detection and Clone Detection.

Models	Code Classification	Vulnerability Detection		Clone Detection	
	Accuracy	Accuracy	F1	Accuracy	F1
SVM	0.5413	0.5223	0.5144	0.6631	0.6780
Naive Bayes	0.2762	0.6934	0.6762	0.5493	0.6330
XGBoost	0.5929	0.7056	0.6951	0.7773	0.7979
LSTM	0.8094	0.7098	0.7135	0.8298	0.8414
BiLSTM	0.8382	0.7131	0.7162	0.8502	0.8544
Transformer Encoder	0.8193	0.4796	0.6482	0.5000	0.6660
Code2Vec	0.8973	0.7180	0.7192	0.6180	0.6719
Tree-LSTM	0.8600	0.7100	0.7209	0.9024	0.9055
GCN	0.8936	0.7015	0.7289	0.9166	0.9188
GAT	0.9042	**0.7278**	0.7306	0.8982	0.8997
GGNN	**0.9204**	0.7158	**0.7344**	**0.9350**	**0.9367**

ment of 2.18%, 0.47%, and 5.11% in accuracy and F1-Score over BiLSTM, which shows that the semantic and syntactic information on AST can be useful in learning the representation for the program, An interesting finding is that Code2Vec performs better than BiLSTM in both code classification and vulnerability detection, with an increase of 5.91% in code classification accuracy and 0.3% in F1-Score of vulnerability detection. However, the performance in clone detection is lacking. We attribute the low performance to the different purposes of the code representation. The original purpose of code2vec is to predict method names in Java programs [33]. Hence, path contexts might be better for predicting function names but lacks in detecting clones.

Finally, we can observe that the graph-based approaches perform the best on the evaluating tasks compared with other representations. GGNN outperforms all other non-graph representations by 2.31-64.42% in code classification accuracy, 1.35%-22.0% in F1-Score of vulnerability detection, and 3.12-30.37% in the accuracy of clone detection. Furthermore, the difference between the GNN variants such as GCN, GAT, and GGNN is not very obvious for example, GGNN and GAT have only a difference of 1.62% in code classification accuracy, 0.38% in vulnerability detection F1-Score, and 3.7% in clone detection F1-Score. We infer that since graph representation embeds more semantics of the programs compared with other baselines, hence it outperforms other baselines by a significant margin. However, the impact between different variants of GNNs is minor.

Answer to RQ1: Graph-based representations are best in representing program semantic among all our comparing representations. We achieve improvements up to 64.42% accuracy in code classification, 8.62% F1-Score in clone detection, and 30.37% F1-Score in vulnerability detection when graph-based representations are used.
Insights: Graph-based representation is superior to the sequence-based or tree-based representation for many tasks. However, the construction of the graph for the program is non-trivial which requires extra efforts and this limits the usage of graph-based representation. It is crucial to have better tools to facilitate the code property graph construction for other programming languages such as Java and Python.

C. RQ2: Comparison of Different Node Embedding Information

In this RQ, we want to investigate the impact of different information that is embedded into the node representation for tree-based or graph-based representation. Specifically, we ablate the performance that is embedded with type, textual, and hybrid embedding. Table III shows the results.

We observe that compared with the type embedding, textual embedding has a significant improvement. Specifically, the improvement in code classification accuracy ranges from 0.07% (GAT) to 7.23% (Tree-LSTM), in vulnerability detection F1-Score in a range of 5.85% (Tree-LSTM) to 8.84% (GCN), and in clone detection F1-Score in a range of 5.58% (GGNN) to 8.78% (GAT). We claim it is reasonable since the tokens tend to carry more semantic information of the program, e.g., a program of the function name *reverse_array*, which we can infer that it is to finish a reversal operation on the input array, and ignore this textual information will increase the difficulty of the model to capture the functionality. Furthermore, we also perform a simple statistic analysis on the OJ dataset for the code classification, we find that the average Jaccard Index (a metric to measure the text-similarity) for programs is 0.51, which indicates there are many overlap tokens between the programs. Hence, ignoring the token information in the node will harm the performance significantly.

Furthermore, we can see that combining textual and type embedding i.e., hybrid embedding, will further improve the performance over code classification and clone detection tasks. For instance, the performance of GGNN and Tree-LSTM increases by 3.65%, 2.20% when hybrid embedding is used on the code classification. We infer that incorporating both node type and node textual information can improve the model capacity and brings better performance. However, on the vulnerability detection, using type embedding has a negative impact. We conjecture that it is due to the way of combining embeddings. In the hybrid embedding, we just employ a linear

TABLE III: Results of Embedding Information.

Embedding	Code Classification	Vulnerability Detection		Clone Detection	
	Accuracy	Accuracy	F1	Accuracy	F1
Tree-LSTM (Type)	0.7657	0.6207	0.6507	0.7862	0.8085
GCN (Type)	0.8198	0.6101	0.6300	0.8410	0.8470
GAT (Type)	0.8860	0.5277	0.6567	0.7972	0.8067
GGNN (Type)	0.8787	0.5128	0.6602	0.8508	0.8558
Tree-LSTM (Textual)	0.8380	0.7162	0.7092	0.8606	0.8726
GCN (Textual)	0.8503	0.7183	0.7184	0.9036	0.9055
GAT (Textual)	0.8930	**0.7289**	0.7153	0.8906	0.8945
GGNN (Textual)	0.8839	0.7233	**0.7362**	0.9094	0.9116
Tree-LSTM (Hybrid)	0.8600	0.7100	0.7209	0.9024	0.9055
GCN (Hybrid)	0.8936	0.7015	0.7289	0.9166	0.9188
GAT (Hybrid)	0.9042	0.7278	0.7306	0.8982	0.8997
GGNN (Hybrid)	**0.9204**	0.7158	0.7344	**0.9350**	**0.9367**

TABLE IV: Results of Graph Representation Analysis with GGNN.

Models	Code Classification	Vulnerability Detection		Clone Detection	
	Accuracy	Accuracy	F1	Accuracy	F1
AST	0.8734	0.7033	0.7125	0.9172	0.9204
CFG	0.8890	0.7042	0.7085	0.9276	0.9300
CDG	0.8856	0.7160	0.7120	0.9144	0.9176
DDG	0.8339	**0.7235**	0.7133	0.9222	0.9251
CPG	**0.9204**	0.7158	**0.7344**	**0.9350**	**0.9367**

layer to concatenate the representation of the textual embedding and type embedding, which is simple and straightforward. Vulnerability detection is a much-complicated task, especially for the real vulnerabilities. Although hybrid embedding obtains sub-optimal performance on the vulnerability detection, however, on the other tasks, it still gets the best performance. We will explore a more effective combining way and leave it as our future work.

Answer to RQ2: Textual information is more critical to learning the program semantics for the tasks as compared to the node type for the tree-based and graph-based representation. Furthermore, combining both with a simple linear layer, we can obtain the optimal performance on code classification and clone detection and sub-optimal performance on vulnerability detection.
Insights: The combination way i.e., a single linear layer is simple and straightforward. Although it generates a promising performance on code classification and clone detection, more tasks need to evaluate its generalization. Furthermore, it is also valuable to explore some other combination ways.

D. RQ3: Efficacy of Different Graph Representations

In this RQ, we study the impact of different graph representations on the performance of the evaluating tasks. Specifically, we evaluate the performance of AST, CFG, CDG, and DDG across the three evaluating tasks. Among all comparing graph neural networks, GGNN has the highest performance as shown in RQ1 and RQ2. Hence, we employed GGNN for this evaluation.

Fig. 4: Vulnerable function that required data dependency for the detection.

```
1  static void fix_bitshift(ShortenContext
   ↪  *s, int32_t *buffer)
2  {
3      int i;
4      if (s->bitshift != 0)
5          for (i = 0; i < s->blocksize;
           ↪  i++)
6              buffer[s->nwrap + i] <<=
               ↪  s->bitshift;
7  }
```

The results are shown in Table IV. We observe that CFG performs better than other graph representations in code classification and clone detection, however on the vulnerability detection, DDG can achieve higher performance. We believe it is reasonable, as data dependency is easy to trigger the vulnerabilities since vulnerable functions tend to have complex dependencies among the statements, e.g., memory dereferencing, and buffer overflows. Furthermore, according to Fabian et al. [55], it is inherent that DDG has a bigger influence on finding vulnerable functions. We show an example for the illustration. Fig 4 shows an example of a function [1] from FFmpeg, which corresponds to a out-of-bound (OOB) bug. GGNN predicted the correct label for this function with the DDG representation, which is shown in Fig 5. Specifically, we can see that the "buffer" array keeps increasing by adding the variable "i" without OOB guard. This results in the indexing of the array possibly growing out of the array size. In DDG, the relationship between the increment of "i" and the "buffer" indexing can be directly observed and there is a dependency between the nodes with the types "ArrayIndexing" and "IncDecOp". However, this relationship is missed in other representations such as AST and CFG.

Furthermore, when all graph representations are combined

[1] A commit with its commit id (f42b31).

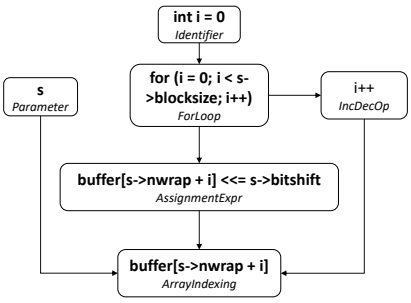


Fig. 5: Data Dependency Graph of the function in Fig 4.

i.e., CPG, it achieves better performance on code classification and clone detection and yields the sub-optimal results on vulnerability detection i.e., the accuracy of CPG (0.7158) is lower than DDG (0.7235), but F1-Score is still the highest (0.7344). It indicates that due to the different characteristics of the task, there might be some specific semantics that are particularly suitable for this task e.g., the data dependency for vulnerability detection and produce the best performance. However, a composite graph with immense semantic and syntactic information can still achieve promising results.

Answer to RQ3: Different task relies on the task-specific semantics to achieve the best performance, however, generally speaking, a composite graph with the comprehensive program semantics can yield the promising results.
Insights: Combining diverse dimensional code semantics, e.g., AST, CFG, CDG, DDG are beneficial for the neural networks to capture the program semantics, However, from one perspective, how to capture the task-specific semantics for a task to achieve the best results is still an open-question, from another perspective, CPG only contains syntactic information (AST), data flow information (CDG, DDG), control flow information (CFG), hence, is it sufficient to represent program semantics?

IV. Discussion

In this section, we first investigate the learnt space by the selected models, then perform an experiment on the semantic-preserving transformation to explore the capacity of different models and conduct a statistic analysis on the predicted results. Finally, we present the threats to the validity of this work.

A. Learnt Representation Space by Neural Network

To demonstrate the learning capability of GGNN over BiLSTM and Tree-LSTM, we employ t-SNE [56] to visualize the learnt representation space of code classification. Specifically, we randomly picked 7 classes in the testing dataset. The learnt space is shown in Figure 6, where the class is labeled along with the color in the plot. We can observe that the graph representation performs the best. As shown in Fig. 6(a) and Figure 6(b), the representations learnt by BiLSTM are more scattered across the plot than the learnt space by Tree-LSTM, which means that the distances of any samples from the same class are greater in BiLSTM. Hence, compared with BiLST, Tree-LSTM produces a better learning space. The learnt space of GGNN is shown in Figure 6(c), we can easily find that the boundary for each class is more clear and the aggregated cluster is more condensed compared with Tree-LSTM. This indicates that GGNN has a more powerful learning capacity compared with BiLSTM and Tree-LSTM.

B. Semantic-Preserving Transformation on Code Representation

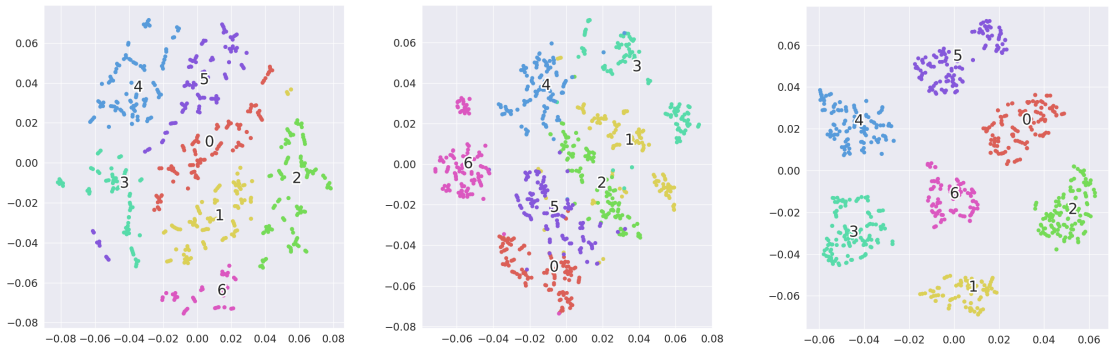
We further explore the capacity of these models on semantic-preserving operations. Here the semantic-preserving operation is defined to transform the code snippet with simple operations such as renaming the identifiers or swapping two independent statements while keeping the original program semantics. Specifically, we randomly select two vulnerable functions [2] that BiLSTM, Tree-LSTM and GGNN predict correctly from the Devign test set and both examples are shown in Fig. 7 and Fig. 8 respectively where the top section shows the original function, while the bottom section are the transformed version. The original function in Fig. 7 lacks a buffer overwrite protection, hence is exposed to buffer overflow vulnerability. For the function in Fig. 8, a deprecated variable, "dc->no_user" is used. This might cause a regression bug where a previously working version stops working. We conduct two kinds of simple transformation operations: (1) We randomly swap the location of two statements, that are independent with others. For instance, in Fig. 7, we swap Line 4 and Line 6 in the original function. This does not affect any dependency as they are independent assertion statements. (2) We randomly select some variables and replace all occurrences of the variable name into the meaningless placeholder. We select both operations since they are simple and easy for implementation.

We input the transformed functions with the trained BiLSTM, Tree-LSTM, and GGNN respectively to investigate the performance. Surprisingly, we find that TreeLSTM and GGNN can produce the correct prediction, while BiLSTM fails. We infer that since the sequence-based representations model the permutation and sequential information of the statements to capture the semantics, hence they are more susceptible to the "swap" and "renaming" operations while these operators do not destruct the original semantics. In contrast, due to Tree-LSTM and GGNN both employ the structure information to capture the semantics, they are more robust to these simple transformations and hence produce the correct predictions.

C. Analysis of Prediction Results

By Table II, we have proved that the graph-based representation has the best performance across three tasks, to further analyze its capacity, we conduct the statistical analysis on the specific complex features in the program that GGNN cannot learn well for the code classification. Specifically,

[2]Both functions are from FFmpeg(1ba08c) and Qemu(efec3d) respectively.



(a) t-SNE Plot of BiLSTM Learnt Representation

(b) t-SNE Plot of Tree-LSTM Learnt Representation

(c) t-SNE Plot of GGNN Learnt Representation

Fig. 6: t-SNE plot of Learnt Representation Space.

Fig. 7: Example 1 of Vulnerable Function with its transformed version.

```
/* Original Function */
static inline void
↪ put_codeword(PutBitContext *pb,
↪ vorbis_enc_codebook *cb, int entry)
{
    assert(entry >= 0);
    assert(entry < cb->nentries);
    assert(cb->lens[entry]);
    put_bits(pb, cb->lens[entry],
    ↪ cb->codewords[entry]);
}

/* Transformed Function */
static inline void f1(v2 *pb, v2 *v1,
↪ int entry)
{
    assert(v1->lens[entry]);
    assert(entry < v1->nentries);
    assert(entry >= 0);
    put_bits(pb, v1->lens[entry],
    ↪ v1->codewords[entry]);
}
```

Fig. 8: Example 2 of Vulnerable Function with its transformed version.

```
/* Original Function */
static void
↪ pic_common_class_init(ObjectClass
↪ *klass, void *data)
{
    DeviceClass *dc =
    ↪ DEVICE_CLASS(klass);
    dc->vmsd = &vmstate_pic_common;
    dc->no_user = 1;
    dc->props = pic_properties_common;
    dc->realize = pic_common_realize;
}


/* Transformed Function */
static void p1(ObjectClass *k1, void *d1)
{
    DeviceClass *d1 = DEVICE_CLASS(k1);
    d1->realize = pic_common_realize;
    d1->no_user = 1;
    d1->vmsd = &vmstate_pic_common;
    d1->props = pic_properties_common;
}
```

we manually examine the correct and incorrect predicted programs on the test set in code classification. We utilize the correct predicted programs (total 149) from the top-three best performing classes (Class 29, Class 66, Class 83) and incorrectly predicted programs (total 63) from the top-three worst-performing classes (Class 25, Class 52, Class 61) for investigation. We further summarize 5 complex features that may be existed in the program: Nested Loops, Multiple Loops, Pointer Operation, Do-While, and Others/Basic. Multiple loops refer to the loops that are initialized in different scopes and nested loops refer to the loops that are initialized within another loop in a program. We group programs under Pointer Operation whenever de-referencing of address occurs in the program. Do-while is an alternative type of looping mechanism. Lastly, if a program has no above-defined complex characteristics e.g., a program that only has single loops and assignment statements, we group it into the Others/Basic category.

TABLE V: Complex Features in Classified Programs.

Complex Features	Incorrect Predicted		Correct Predicted	
	Num	Ratio	Num	Ratio
Do-While	0	0%	3	1.84%
Multiple Loops	29	34.11%	58	35.58%
Nested Loops	25	29.41%	18	11.04%
Pointer Operation	17	20.00%	4	2.45%
Others/Basic	14	16.47%	80	49.07%
Total Features	85	-	163	-

The statistical results are shown in Table V. Note that a program can have multiple defined categories, for example, a program can have both multiple loops and nested loops associated with it. We can find that the values for Nested Loops and Pointer Operation in the incorrect samples are higher than these values in the correct samples, which proves that these program features are not learnt well. Furthermore, the

number of samples with the simple structure i.e., Others/Basic category in the incorrect samples are far less than the samples in the correct samples, which illustrates that complex program structure is still a challenge for the neural network to learn semantics. Lastly, we cannot claim the learning capacity of the neural network on Do-While and Multiple Loops, since the values of the correct and incorrect samples are near.

D. Threats to Validity

Evaluation Tasks. There are many other code-related tasks such as code translation [57], code summarization [7], code search [11], [58]. We only consider the classification tasks on C programming language since classification tasks are more suitable for quantitative analysis as compared to the generation tasks such as code summarization. Our study provide ideas on how to evaluate program semantics on complicated tasks.

State-of-the-art Results. We did not compare with the state-of-the-art results for each task, since these high-performing approaches tend to design more complicated architecture than these basic networks. For example, to achieve the best performance, Devign [14] utilized a convolution module to further improve the learnt representations by GGNN. To reduce the complexity of the analysis, we target basic models and conduct a systematic study to explore different categories of code representations, which we believe is fundamental and meaningful.

Hyper-parameter Tuning. Hyper-parameters affect the performance of each model. For fairness in our study, we tune the hyper-parameters such as embedding dimension, batch size to obtain the best result for each evaluating task. We tune code classification based on the best accuracy, vulnerability detection, and clone detection based on the best f1. We set the same seed for all experiments for the reproduction and eliminating the potential bias of randomness.

V. Related Works

Source Code Representations. Feature-based representations [21], [59] are commonly used in software engineering tasks as word frequency are important in identifying source code. Other feature-based representations, such as FLUCCS [17] require domain experts into identifying key features of source code. Sequence-based representation employs distributed representation in learning each token. Many works employ sequence-based representation in clone detection [6], vulnerability detection [60], auto-patch identification [61], code review [62]. These works have proven that sequence-based representations are capable of representing semantic and syntactic information of programs. Researchers investigate deeper representation by traversing the tree-based representation of the source code, such as AST [5], [32], [33], [63]. Code2vec [33] and Code2seq [32] find an efficient representation of source code by traversing AST and learning on the relevant paths. Many works [35] also employ AST in learning code clones as clones inherently have similar program constructs. While these representations contains the syntactic information of the source code, semantic information is lacking in them as control flow and data flow are not well-model into them. More works start to look into graph-based representation for programs and their application [7], [9], [36], [58], [64] on GNNs. Miltiadis et al. [36] combines the AST node with control flows and data dependency to learn source code graph representations. Wang et al. [37] added data-flow information to AST to improve the performance of clone detection. These works focus on embedding source code into a vector space without losing semantic and syntactic knowledge. Kang et al. [65] conduct an code2vec assessment on downstream tasks. It is similar to our work in nature, however, we employ a complicated graph representation of source code and extensively evaluate them.

Relevant Works to Selected Tasks. Classifying source code by functionality enables developers to locate functions. Many works [5], [39] focus on classifying source code. Clone detection is a popular topic in software engineering. Several works [5], [65]–[67] relate to clone detection. Vulnerable functions often are not straightforward and contain hidden semantics across multiple functions. Different ways are employed in finding vulnerabilities, such as fuzzing [68]–[72], malware detection [73], program metrics [74] and deep learning [14].

VI. Conclusion

We conduct a systematic study to investigate program representations i.e., Feature-based, Sequence-based, Tree-based, and Graph-based representation across three diverse and popular code-related tasks i.e., code classification, clone detection, and vulnerability detection on public benchmarks. We conclude our findings as follows: (1) Graph-based representation outperforms other techniques by a significant margin. (2) The node type and node textual information are both beneficial in the tree-based and graph-based representation to learn the program semantics, however, node textual information is more critical. (3) Different task relies on task-specific semantics to achieve the best performance, however, a composite graph with the comprehensive program semantics can still yield promising results. By our study, we hope to provide several insights and follow-up directions in the program representation field.

VII. Acknowledgments

This research is partially supported by the National Key R&D Programmes of China (No. 2019AAA0104301), National Natural Science Foundation of China (No. 61902395), National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001.

REFERENCES

[1] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," *CoRR*, vol. abs/2004.10657, 2020. [Online]. Available: https://arxiv.org/abs/2004.10657

[2] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 152–162. [Online]. Available: https://doi.org/10.1145/3236024.3236051

[3] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," *SIGPLAN Not.*, vol. 50, no. 1, p. 111–124, Jan. 2015. [Online]. Available: https://doi.org/10.1145/2775051.2677009

[4] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. USA: IEEE Computer Society, 2007, p. 96–105. [Online]. Available: https://doi.org/10.1109/ICSE.2007.30

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

[6] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2021.

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

[8] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," *arXiv preprint arXiv:2004.02843*, 2020.

[9] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," *arXiv preprint arXiv:1811.01824*, 2018.

[10] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

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

[12] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[13] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 196–207.

[14] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 10 197–10 207.

[15] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[16] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[17] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 273–283. [Online]. Available: https://doi.org/10.1145/3092703.3092717

[18] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[19] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[20] L. Vanneschi and R. Poli, *Genetic Programming — Introduction, Applications, Theory and Open Issues*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 709–739. [Online]. Available: https://doi.org/10.1007/978-3-540-92910-9_24

[21] D. Behl, S. Handa, and A. Arora, "A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf," in *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, 2014, pp. 294–299.

[22] C. Sammut and G. I. Webb, Eds., *TF–IDF*. Boston, MA: Springer US, 2010, pp. 986–987. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_832

[23] D. J. Hand and K. Yu, "Idiot's bayes: Not so stupid after all?" *International Statistical Review / Revue Internationale de Statistique*, vol. 69, no. 3, pp. 385–398, 2001. [Online]. Available: http://www.jstor.org/stable/1403452

[24] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. IEEE Press, 2015, p. 334–345.

[25] K. M. Hermann, T. Kociský, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom, "Teaching machines to read and comprehend," *CoRR*, vol. abs/1506.03340, 2015. [Online]. Available: http://arxiv.org/abs/1506.03340

[26] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119.

[27] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *CoRR*, vol. abs/1801.01681, 2018. [Online]. Available: http://arxiv.org/abs/1801.01681

[28] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *Signal Processing, IEEE Transactions on*, vol. 45, pp. 2673 – 2681, 12 1997.

[29] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 7 2018, pp. 2269–2275. [Online]. Available: https://doi.org/10.24963/ijcai.2018/314

[30] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://www.aclweb.org/anthology/P02-1040

[31] A. Lavie and A. Agarwal, "Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments," in *Proceedings of the Second Workshop on Statistical Machine Translation*, ser. StatMT '07. USA: Association for Computational Linguistics, 2007, p. 228–231.

[32] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *CoRR*, vol. abs/1808.01400, 2018. [Online]. Available: http://arxiv.org/abs/1808.01400

[33] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290353

[34] N. D. Bui, Y. Yu, and L. Jiang, "Treecaps: Tree-based capsule networks for source code processing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, 2021, pp. 30–38.

[35] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 3034–3040. [Online]. Available: https://doi.org/10.24963/ijcai.2017/423

[36] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740

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

[38] F. Yamaguchi. Welcome to joern's documentation! — joern 0.2.5 documentationg. [Online]. Available: https://joern.readthedocs.io/en/latest/index.html

[39] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 1287–1293.

[40] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," *CoRR*, vol. abs/1603.02754, 2016. [Online]. Available: http://arxiv.org/abs/1603.02754

[41] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," *CoRR*, vol. abs/1811.01824, 2018. [Online]. Available: http://arxiv.org/abs/1811.01824

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[44] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[45] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *CoRR*, vol. abs/1503.00075, 2015. [Online]. Available: http://arxiv.org/abs/1503.00075

[46] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, "Automatic source code summarization with extended tree-lstm," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8.

[47] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, "Pathminer: A library for mining of path-based representations of code," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 13–17.

[48] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

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

[50] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *CoRR*, vol. abs/1606.09375, 2016. [Online]. Available: http://arxiv.org/abs/1606.09375

[51] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *International Conference on Learning Representations*, 2018, accepted as poster. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[52] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks. arxiv, 2015," *arXiv preprint arXiv:1511.05493*, 2015.

[53] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. [Online]. Available: https://www.aclweb.org/anthology/D14-1179

[54] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. [Online]. Available: https://arxiv.org/abs/1909.01315

[55] F. Yamaguchi, "Pattern-based vulnerability discovery." Ph.D. dissertation, University of Göttingen, 2015.

[56] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[57] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[58] S. Liu, X. Xie, L. Ma, J. Siow, and Y. Liu, "Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search," *arXiv preprint arXiv:2111.02671*, 2021.

[59] G. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng, and X. Zhao, "Automatic classification method for software vulnerability based on deep neural network," *IEEE Access*, vol. 7, pp. 28 291–28 298, 2019.

[60] A. Xu, T. Dai, H. Chen, Z. Ming, and W. Li, "Vulnerability detection for source code using contextual lstm," in *2018 5th International Conference on Systems and Informatics (ICSAI)*, 2018, pp. 1225–1230.

[61] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *arXiv preprint arXiv:2105.14565*, 2021.

[62] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.

[63] S. Liu, C. Gao, S. Chen, L. Y. Nie, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *arXiv preprint arXiv:1912.02972*, 2019.

[64] S. Liu, "A unified framework to learn program semantics with graph neural networks," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1364–1366.

[65] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1–12.

[66] V. Saini, F. Farmahinifarahani, Y. Lu, D. Yang, P. Martins, H. Sajnani, P. Baldi, and C. V. Lopes, "Towards automating precision studies of clone detectors," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 49–59. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00023

[67] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: Cross language code clone detection using syntactical features and api documentation," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 1026–1037. [Online]. Available: https://doi.org/10.1109/ASE.2019.00099

[68] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "Fot: A versatile, configurable, extensible fuzzing framework," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 867–870. [Online]. Available: https://doi.org/10.1145/3236024.3264593

[69] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 627–637. [Online]. Available: https://doi.org/10.1145/3106237.3106295

[70] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 533–544. [Online]. Available: https://doi.org/10.1145/3338906.3338975

[71] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "APICraft: Fuzz driver generation for closed-source SDK libraries," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2811–2828. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-cen

[72] Y. Li, G. Meng, J. Xu, C. Zhang, H. Chen, and X. Xie, "Vall-nut: Principled anti-greybox-fuzzing," in *The 32nd International Symposium on Software Reliability Engineering*, ser. ISSRE 2021, 2021.

[73] G. Meng, M. Patrick, Y. Xue, Y. Liu, and J. Zhang, "Securing android app markets via modelling and predicting malware spread between markets," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 7, pp. 1944–1959, Jul 2019.

[74] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: Identifying vulnerable code for vulnerability assessment through program metrics," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 60–71. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00024