NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

LEARNING PROGRAM SEMANTICS VIA EXPLORING PROGRAM STRUCTURES WITH DEEP LEARNING

SHANGQING LIU

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

2022

LEARNING PROGRAM SEMANTICS VIA EXPLORING PROGRAM STRUCTURES WITH DEEP LEARNING

SHANGQING LIU

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2022

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

04/01/2023

. . . . . .

Date

Liu Shangqing

. . . . . .

LIU SHANGQING

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

04/01/2023

. . . . . .

Date

. . . . . .

LIU YANG

Authorship Attribution Statement

This thesis contains material from 2 papers published in the following peer-reviewed journal and 3 papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as Shangqing Liu, Cuiyun Gao, Sen Chen, Nie Lun Yiu, Yang Liu, "ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking," IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2020.3038681.

The contributions of the co-authors are as follows:

- I was the lead author. I was responsible for writing the code, drafting the manuscript and conducting all experiments.
- Prof. Gao and Prof. Chen helped to polish the manuscript drafts.
- Mr Yiu helped to draft the related work section in the paper.
- Prof. Liu discussed the idea and provided support for the research.

Chapter 4 is published as Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, Yang Liu. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.

The contributions of the co-authors are as follows:

- I was the corresponding author. I was responsible for co-designing the methodology, writing most of the code, drafting the manuscript and conducting the majority of the experiments.

- Dr Zhou lead the project, proposed the methodology and revised the manuscript drafts.
- Mr Siow helped to write some code.
- Prof. Du helped to conduct some baseline experiments.
- Prof. Liu discussed the idea and provided support for the research.

Chapter 5 is published as Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, Yang Liu, "Retrieval-Augmented Generation for Code Summarization via Hybrid GNN," 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.

The contributions of the co-authors are as follows:

- I was the lead author. I was responsible for writing the code, drafting the manuscript and conducting all experiments.
- Dr Chen helped to design the methodology, checked the implementation and revised the manuscript drafts.
- Prof. Xie and Mr Siow helped to polish the manuscript drafts.
- Prof. Liu discussed the idea and provided support for the research.

Chapter 6 is published as Shangqing Liu, Xiaofei Xie, Jing Kai Siow, Lei Ma, Guozhu Meng, Yang Liu, "GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search," IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2022.3233901.

The contributions of the co-authors are as follows:

- I was the lead author. I was responsible for writing the code, drafting the manuscript and conducting all experiments.

- Mr Siow, Prof. Xie, Prof. Ma and Prof. Meng helped to polish the manuscript drafts.
- Prof. Liu discussed the idea and provided support for the research.

Chapter 7 is published as Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, Yang Liu, "ContraBERT: Enhancing Code Pre-trained Models via Contrastive Learning," The 45th International Conference on Software Engineering.

The contributions of the co-authors are as follows:

- I was the lead author. I was responsible for writing the majority of the code, drafting the manuscript and conducting all experiments.
- Mr Wu, Prof. Xie and Prof. Meng helped to polish the manuscript drafts.
- Prof. Liu discussed the idea and provided support for the research.

04/01/2023 Liu Shangqing

. . . . . .

Date

. . . . . .

LIU SHANGQING

Acknowledgements

At the time of ending my Ph.D study, I would like to express my sincere gratitude and appreciation to anyone who has helped me during my Ph.D study. I have gained a lot in this period: knowledge, skills, friendship, etc. The study journey at NTU is a precious treasure in my life.

First, I would like to extend my sincere gratitude to my Ph.D supervisor Prof. Liu Yang for his valuable understanding, guidance and support. Prof. Liu Yang gave me free time to learn the basics of deep learning and software engineering. His detailed guidance helped me to strengthen my research thinking and research skills greatly. His valuable support during these years has helped me to conduct my research. Without his help, I will not be able to finish this thesis.

Second, I would like to thank Prof. Chen Bing, who is my supervisor at Nanjing University of Aeronautics and Astronautics when I pursued my master's degree. Without his recommendation, I cannot have such a meaningful journey at NTU.

Third, I would like to thank my colleagues and friends in Cyber Security Lab (CSL) for their friendship and help in both my life and study, especially Prof. Xie Xiaofei, Dr Zhou Yaqin, Dr Feng Ruitao, Dr Zhou Yuan, Dr Li Yuekang, Dr Chen Hongxu, Prof. Chen Sen, Prof. Gao Cuiyun, Dr Siow Jing Kai and many others. I also want to thank CSL lab technician Mr Tan Suan Hai for the kind IT support. In addition, I want to express my sincere appreciation to Dr Chen Yu from Meta AI. I learned a lot of knowledge and skills from him.

Last but also important, I would like to express my sincere appreciation to my parents, who provided their unconditional support to me. Without their support, I cannot finish my study. I am in deep gratitude for their love and sacrifice.

Contents

List of Figures

List of Tables

Summary

The ubiquitousness of software in modern society and the boom in open-source software have made software engineering into the "big code" era. The availability of code-related data is massive (e.g., billions of code, millions of code changes, bug fixes and code documentation), which yields a hot topic in both academia and industry, which is how to adopt the data-driven approach (e.g., deep learning) to solve conventional software engineering (SE) problems. Many works analogize programs to sequential text in natural language with some sequential-based models such as RNNs, LSTMs or the Transformer to learn the program semantics. However, though the programs can be considered as a flat sequence roughly, they tend to consist of different structures in a program such as abstract syntax tree, control flow, and data flow. In this thesis, we explore how to encode program structures beyond the program text with deep learning techniques to learn program semantics for various applications.

First, we aim at exploring the way to inject the program structures into sequential-based models such as LSTMs to improve the model on learning program semantics. Because of the rapidly increasing number of open-source software, currently, millions of projects are hosted on the software hosting platform (e.g., GitHub), describing the code changes among different versions of the software in the natural language (i.e., commit message) can greatly facilitate the users and developers to understand the version evolution rapidly. Hence, we target commit message generation. Specifically, since abstract syntax tree (AST) is the first-step representation used by the program parser to understand the semantics of the program, we extract a set of AST paths behind the code changes and feed them with the LSTM-based encoder-decoder framework for the message generation. We evaluate our approach by comparing it with some sequential-based techniques, which leverage the text of the changed code with LSTMs or its variants for the generation. The results demonstrate that extracting structures hidden in the program is able to facilitate the neural model to learn the program semantics and yield more accurate results.

Second, besides ASTs, a program can also be represented in other structures such as a control flow graph and data flow graph. We propose a composite programming representation technique Devign, which combines code property graph (CPG), consisting of abstract syntax tree (AST), control flow graph (CFG), and data flow graph (DFG), with natural code sequence (NCS) to represent a program and further leverage graph neural networks (GNNs) for vulnerability identification. Software vulnerability identification is a complicated task and it requires the model to learn the comprehensive program semantics for accurate identification. To address the limitation of currently no public large-scale real vulnerability dataset, we design an algorithm with a cross-validation process by four security experts for reliable data collection. Furthermore, we have made our partial data public to benefit academia and industry. In addition, we propose a convolution module followed by a gated graph neural network (GGNN) to learn the fine-grained vulnerability features. The extensive experimental results in terms of accuracy and F1 confirm that when combing all kinds of program structures, our approach exceeds any one of the program structures and achieves the best performance for vulnerability identification.

Third, source code summarization targets to describe the functionality of a code snippet in the natural language and an accurate source code summarization system can help the developers understand the functional logic without reading the code. However, a reliable code summarization system requires the model to have the capacity to learn program semantics to generate an accurate description. Inspired by our previous work Devign, we utilize CPG, which incorporates AST, CFG and DFG to represent the code snippet. We further propose a graph-based encoder-decoder model (i.e., HGNN) for code summarization, where the input is the code property graph and the output is the code summary. To improve the learning capacity of the model, we propose global attention on any pair of nodes over a graph to mitigate conventional GNN models, which only capture the local neighbourhood information. In addition, we also innovate a retrieval-based augmentation mechanism to retrieve the most similar source code with the current program from the retrieval database and combine the retrieved code as well as the corresponding summary to further improve the quality of the generated summary. The extensive evaluation has demonstrated that HGNN is more powerful than

conventional GNNs such as the graph convolution network (GCN) and the graph attention network (GAT). Furthermore, our proposed source code summarization system can generate more high-quality summaries.

Fourth, code search aims at retrieving the semantic-equivalent code snippets from a large code corpus when providing a query in the natural language. It is a critical issue especially in the "big code" era since some studies have confirmed that more than 90% of the efforts from software developers aim to reuse the existing code. Besides constructing the program graph with various types of edges on AST to represent a program, we further propose to convert the natural language query to a dependency graph to explicitly describe the token relations in the query and design our neural network architecture GraphSearchNet, which considers both the program and the query as the graphs and incorporates the bidirectional gated graph neural network (BiGGNN) with the multi-head attention mechanism for semantic code search. An extensive experiment compared with 11 baseline approaches has demonstrated that by jointly learning the structure behind the program and the natural language query, the semantic mapping relations can be easily captured.

Finally, the aforementioned models are all trained in a supervised manner, however, recently, unsupervised techniques have demonstrated their superiority against the supervised techniques and the large-scale pre-trained models in the code scenario such as CodeBERT, GraphCodeBERT has provided significant improvements over various downstream software engineering tasks such as clone detection, code translation. Specifically, GraphCodeBERT incorporates data flow information of the program to neural networks at the pre-training phase to learn general code representations. It has achieved state-of-the-art performance for different downstream tasks. We want to explore the other questions in these large-scale pre-trained code models such as robustness. From our preliminary findings, we observe that these pre-trained models are not robust to the label-preserving program mutations and the simple variable renaming operation will degrade the performance significantly. It proves that program semantics are not captured well by these pre-trained models. To address this limitation, we propose our framework ContraBERT to enhance the robustness of existing pre-trained models. We design a

set of data augmentation operators to construct different variants. Then we combine these variants with the original data and further train the existing pre-trained models with masked language modeling (MLM) and InfoNCE objective to enhance the model robustness. The experiments from the open-source benchmark CodeXGLUE have confirmed the robustness of the pre-trained models is improved. In addition, we also confirm that these robustness-enhanced models could provide significant improvements on different tasks compared with the original model.

Chapter 1

Introduction

The concept of "Deep Learning" was proposed by Hinton et al. [1] in 2006. They believed that an artificial neural network (ANN) [2] with multiple hidden layers is powerful to learn data features automatically. In 2012, AlexNet [3], which was trained by a deep convolutional neural network (CNN) for the image classification on the ImageNet dataset, outperformed state-of-the-art techniques by a significant margin and it had attracted widespread attention. Afterwards, with the rapid development of deep learning framework, algorithms and hardware support, we have entered the era of deep learning and it has been widely used in various fields such as computer vision [4], natural language processing [5] and game playing [6].

Inspired by the great success of deep learning techniques in various fields, researchers from both academia and industry have shown great enthusiasm for exploring and applying deep learning techniques in software engineering (SE). Deep learning techniques tend to rely on massive data to achieve promising results. However, thanks to the booming development of open-source software, the availability of code-related data is massive. According to the official report [7] from GitHub, it has reached 100 million hosted repositories by 2018. Hence, the massive amounts of code-related data break this fundamental bottleneck. We can observe that various SE-related tasks such as software vulnerability detection [8, 9], code clone detection [10, 11], test case generation [12], etc. have explored deep learning techniques to solve practical problems.

Though deep learning techniques have been widely adopted for different SE tasks, in essence, these tasks rely on the neural model to learn program semantics for different scenarios. Many existing works [8, 12–14] consider the program as a flat sequence and feed them directly into the sequential-based neural models such as LSTMs [15] or the Transformer [16] for learning, however, a program tends to consist of complex structures such as abstract syntax tree, control flow graph and data flow graph. Hence, in this thesis, we try to explore the way by incorporating program structures into deep learning techniques to learn program semantics for different software engineering scenarios.

1.1 Motivations and Challenges

The program differs from the natural language text in that a program is executable and has formal syntax and semantics. Program is executable, however, text often cannot. Hence, the program is often semantically brittle (i.e., small changes can lead to great changes in program semantics), whereas the natural language text is more robust for readers to understand its meaning even if it contains some mistakes. Furthermore, the programming language is a formal language, which follows a strict mathematical model. Programming must strictly follow the syntax specification to ensure the code can be executed in a proper run-time environment. As a result, the structure hidden in a program text tends to reveal its semantics. We present a simple example, which is shown in Figure 1.1 for better illustration. This function causes the out-of-bound (OOB) issue, that was introduced by the commit [1] in open-source project FFmpeg. Specifically, we can observe that the "buffer" array keeps increasing by adding the variable "i" without the OOB guard. The increasing index possibly grows out of the array size, which leads to out-of-bound error. In addition, if we only consider the sequential-based neural networks and take this function as a flat sequence (i.e., static void ... i++ ... buffer [... + i] ...) as the input for the model, the relations between "buffer" and "i" cannot be easily captured intuitively. In contrast, the data dependency graph of this function is presented in Figure 1.2, we can observe that there is a direct link from the node "i++" to "buffer" to explicitly describe the relations of both variables. Hence, we can find that

[1] The commit id (f42b31).

FIGURE 1.1: A real vulnerable function extracted from FFmpeg.

```
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  }
```

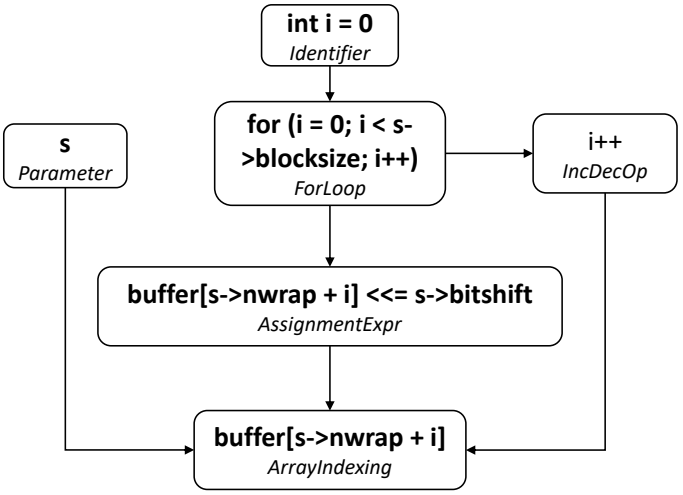


FIGURE 1.2: The data dependency graph of the function in Figure 1.1.

the structure of the program tends to easily reveal more program logic, which is beneficial for neural networks to capture program semantics. This inspires our study on the exploration of incorporating different program structures with deep learning techniques to learn program semantics.

The conventional neural network architectures such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), long short-term memory networks (LSTMs) and the Transformer are not designed for structured data, the way to incorporate program structures with the sequential-based models for program learning is less touched and the performance is also unknown. Furthermore, focusing on the structured data, whether we can use the structured-based neural models to learn the program semantics for different software engineering tasks and how to design the model architecture based on the characteristics of different tasks to achieve the best results are also

unknown. In addition, the recent widely concerned large-scale pre-trained models in the code scenario such as CodeBERT [17], GraphCodeBERT [18] have attracted wide attention. Especially for GraphCodeBERT, it also incorporates program structures for the pre-training and achieves state-of-the-art performance for different software engineering tasks. We want to explore the other questions in these large-scale pre-trained code models such as robustness and this problem is less explored.

Totally, we sum up some of the key problems in a systematic way.

- While maintaining the unchanged sequential-based network architecture, how to embed structures of a program to the sequential-based models?
- How to incorporate different structures to represent the semantics of a program comprehensively?
- Based on different software engineering tasks, how to design the specific neural network architecture to achieve the best performance?
- How to qualify the robustness of the pre-trained models in the code scenario?

We highlight the key challenges for solving the above problems.

1. The sequential-based neural networks such as RNNs or LSTMs usually take the sequential text as input, however, program structures tend to non-sequential format. For example, an abstract syntax tree (AST) is tree-structured data. Hence, how to convert this tree to a sequence format while maintaining the program syntax for the sequential-based models to learn program semantics is unknown.

2. Different kinds of structure can be extracted from a program and each of them may focus on a specific aspect, for example, abstract syntax tree (AST), which is the tree-structured data, that is used for the compilation, data flow graph (DFG), which records the usage of variables in a program and control flow graph (CFG), which describes all paths that might be executed during program execution. Thus, a unified representation of the program which consists of different program semantics such as control flow, and data flow can help the model comprehensively learn program

semantics from different dimensions. However, the way to incorporate different structures into a unified representation is a challenge.

3. Different software engineering tasks tend to have some characteristics, such as source code summarization, which have been confirmed that some information retrieval-based techniques could help the model provide significant improvements [19], thus aiming at different tasks, how to design the network architecture considering different characteristics of the task to achieve the best performance is a huge challenge.

4. The pre-trained models are usually trained in an unsupervised manner. Specifically, they are first trained on the large code corpus where the data are unlabeled to learn the general token representations, then based on different downstream tasks, we fine-tuned them with the task-specific data. Hence, the way to select a proper downstream task to evaluate the robustness of the pre-trained models is worth exploring. Furthermore, how to enhance the model's robustness is also unknown.

1.2 Main Work

Figure 6.3 presents the overview of this thesis. First, we seek to explore the way to embed the program structure into the sequential-based models to confirm the effectiveness of the program structure is beneficial for the neural networks. Then inspired by recent advanced works on graph neural networks (GNNs) [20–22], we incorporate different kinds of program structures with GNNs for program semantics learning. Specifically, we investigate the semantic learning capacity of GNNs on three diverse tasks (i.e., vulnerability detection, source code summarization and code search). We design different GNN variants to fully maximize the learning capacity of the model based on the characteristic of each task. In addition, we also conduct the robustness analysis on the recent popular pre-trained models in the code scenario.

To solve the first problem, we propose ATOM, which is based on the abstract syntax tree (AST) of the program for commit message generation to investigate the performance of program structure to the sequential-based models. We denote a set of AST paths to

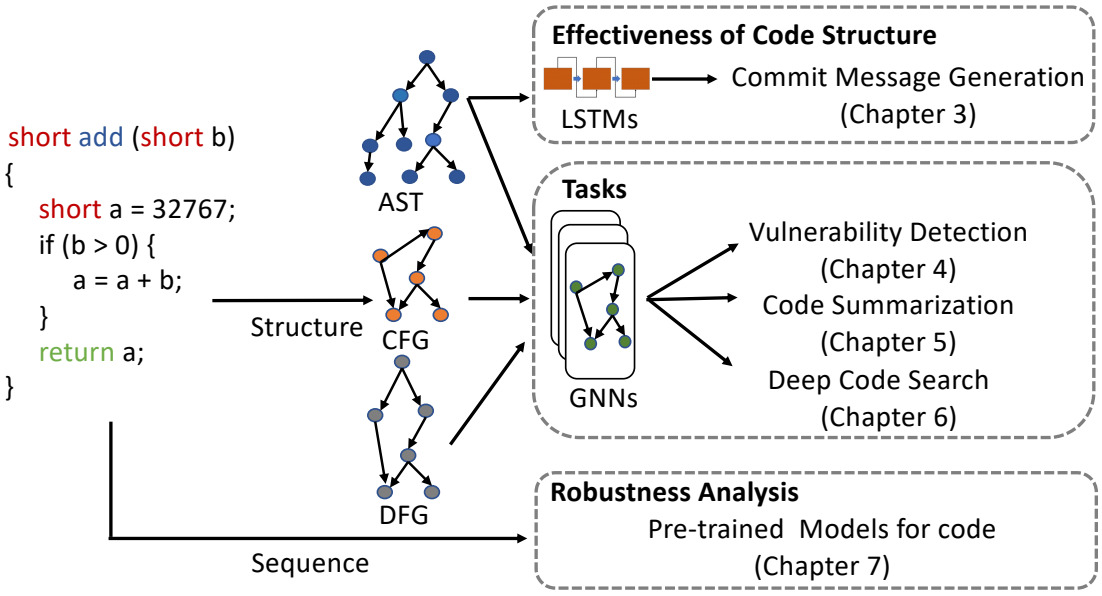


FIGURE 1.3: Research contributions of the thesis.

maintain the syntax of a program as the input for the model. Specifically, each path is a set of nodes that go through the shortest distance between the start node and the end node. The start node and end node are the leaf nodes in AST, which contain node values, whereas the other nodes in a path are non-terminal nodes, which have node types. In the learning phase, to encode each path, we divide this process into two steps. For the values in the start node and end node, we split them into subtokens and then sum them up with their embedding vectors as the leaf features. Furthermore, the node types in a path are encoded by the bidirectional LSTM and the final states of LSTM are used as the syntax features. Finally, we concatenate leaf features and syntax features to represent each path. In this way, we can convert AST to the sequence format while keeping the syntax information with the sequential-based models for learning.

The second problem and the third problem are closely related. Generally, in this thesis, we consider the AST as the basis and combine it with other structures such as CFG and DFG for GNNs to learn the program semantics for different tasks. Specifically, for vulnerability detection, since it requires learning the comprehensive program semantics for accurate identification, we combine a code property graph (CPG), which consists of AST, CFG, DFG, with Natural Code Sequence (NCS) to represent a function. We

utilize a gated graph neural network (GGNN) to learn the program semantics. Furthermore, we propose a fine-grained feature classifier (i.e., Convolution module) followed after GGNN to learn vulnerability-related features over the node features for classification. The experiments demonstrate the effectiveness compared with the multi-layer perceptron (i.e., MLP), which is usually used after GGNN. For source code summarization, we also utilize CPG to represent the program. Based on CPG, we propose a graph-based encoder-decoder neural network architecture (i.e., HGNN), where the encoder is designed by combining the conventional message passing used in GGNN and the proposed global attention over any pair of nodes to enhance the model in learning global interactions over any pair of nodes. In addition, we innovate a retrieval-based augmentation mechanism to retrieve the most similar code-summary pair for the model to further improve the quality of the generated summaries. In terms of code search, we also take AST as the cornerstone and combine it with syntax edge (i.e., NextToken, Subtoken) and data flow edge (i.e., LastUse, LastWrite and ComputedFrom) to represent the program. Aiming at the specificity of this task, which involves the query as the input, we construct the dependency graph for a query and further propose GraphSearchNet for semantic code search. It consists of two separate encoders (i.e., program encoder and query encoder), each of them utilising the bidirectional GGNN [23] with a multi-head attention module to learn semantic relations.

For the last problem, the pre-trained models are usually trained on the large unlabeled corpus and then fine-tuned on different kinds of downstream tasks. Some generation tasks such as code translation [24, 25], source code summarization [26, 27] are unsuitable for measuring the model robustness, due to the complicity of tasks. In contrast, we select clone detection, which is to identify the semantic-equivalent program from a set of distractors. We further evaluate the model robustness in a zero-shot manner, which means that it does not involve fine-tuning and we directly utilize the pre-trained model for evaluation. Specifically, we select those samples that are predicted correctly by the pre-trained models. Then we randomly rename the variables within these programs from 1 to 8 edits. We utilize these newly generated data to evaluate the predicted accuracy. We find that the accuracy drops significantly, which confirms these pre-trained

models are not robust to adversarial examples. We further propose our framework ContraBERT by designing a set of data augmentation operators with contrastive learning to improve the robustness of the existing pre-trained models. Furthermore, we also find that these robustness-enhanced pre-trained models provide significant improvements on various kinds of downstream tasks compared with the initialized models.

1.3 Contributions of the Thesis

The main contributions of this thesis are as follows:

First, we demonstrate the hidden structures beyond the program text are beneficial for the sequential-based models to learn program semantics. Furthermore, we design an AST-based commit message generation system, which takes AST paths as the input for an LSTM-based encoder-decoder framework to generate commit messages.

Second, focusing on different SE tasks (i.e., vulnerability detection, source code summarization, code search), we fully take the task characteristic into consideration and further design the graph-based neural network architectures with different GNN variants to improve the model learning capacity and achieve the state-of-the-art performance.

Last but not the least, we observe that the current widely used pre-trained models in the code scenario are not robust to adversarial attacks and we further propose a framework based on the well-designed data augmentation operators with contrastive learning to improve the robustness of existing pre-trained models.

1.4 List of Materials Related to the Thesis

The thesis mainly contains the materials from the following papers.

1. **Shangqing Liu**, Cuiyun Gao, Sen Chen, Nie Lun Yiu, Yang Liu, ”ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking,” IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2020.3038681.

2. Yaqin Zhou, **Shangqing Liu**, Jing Kai Siow, Xiaoning Du, Yang Liu. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.

3. **Shangqing Liu**, Yu Chen, Xiaofei Xie, Jing Kai Siow, Yang Liu, "Retrieval-Augmented Generation for Code Summarization via Hybrid GNN," 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.

4. **Shangqing Liu**, Xiaofei Xie, Jing Kai Siow, Lei Ma, Guozhu Meng, Yang Liu, "GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search," IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2022.3233901.

5. **Shangqing Liu**, Bozhi Wu, Xiaofei Xie, Guozhu Meng, Yang Liu, "ContraBERT: Enhancing Code Pre-trained Models via Contrastive Learning," The 45th International Conference on Software Engineering.

1.5 Outline of the Thesis

The rest of this thesis is organized as follows:

Chapter 2 introduces the background on learning program semantics with deep learning techniques for different SE tasks.

Chapter 3 presents the work from Paper 1. In this chapter, we explore the way to incorporate the program syntax with bidirectional LSTM for commit message generation. This chapter consists of model design and the comparison with other sequential-based models.

Chapter 4 presents the work of Paper 2. In this chapter, we aim at exploring the way to extract different kinds of structures with graph neural networks to learn program

semantics for vulnerability detection. This chapter consists of the details of the model design, the released large-scale vulnerability dataset and the evaluation.

Chapter 5 presents the work of Paper 3. In this chapter, we focus on the way to improve the learning capacity of GNNs for source code summarization. We further incorporate the retrieval mechanism for the model to produce high-quality results. This chapter consists of the details of the model design and the comparison with other baselines.

Chapter 6 presents the work of Paper 4. In this chapter, we target utilizing the program and query structure with GNNs for semantic code search. In addition, to transform the program into a graph, we propose to convert the natural language query to a graph to capture the semantic relations between the program and the query. This chapter consists of the details of the model design and the comparison with other baselines.

Chapter 7 presents the work of Paper 5. In this chapter, we aim the analysis of the widely used pre-trained models in the code scenario and further propose a framework to enhance the robustness of the pre-trained models. This chapter contains the details of the model design, the results of the analysis and the comparison with the original models.

Chapter 8 concludes this thesis and provides some future research directions in the area of employing deep learning techniques to learn program semantics.

Chapter 2

Related Work

Learning program semantics with deep learning techniques for software engineering is a hot topic, in this chapter, we first briefly introduce the related works on four diverse SE applications (i.e., commit message generation, vulnerability identification, source code summarization, code search) that are related to this thesis. Then we introduce the widely concerned pre-trained models in the code scenario. Lastly, we introduce the basics of graph neural networks (GNNs), which we will use for some applications.

2.1 Commit Message Generation

The previous works of commit message generation can be roughly grouped into three categories: rule-based, retrieval-based, and learning-based techniques. Early works [28–31] depend on pre-defined rules or templates to generate commit messages based on the changed code. For instance, Buse et al. [28] utilized the pre-defined templates to generate commit messages. Shen et al. [31] generated the commit message by an analysis of the changed code to extract what information and why information with the defined template for the generation. ChangeSribe [29, 30] further considered the impact set of a commit in addition to the commit stereotype and type of changes, then the extracted information was filled to a pre-defined template to generate the commit message.

The retrieval-based techniques [32, 33] mainly retrieve the most similar code changes from the database and reuse its corresponding commit message as the generated message for current commit. For instance, Huang et al. [32] measured the code changes based on the syntactic similarity and semantic similarity to return existing commit messages. NNGen [33] reused the commit message based on computing the cosine similarity between the changed code vectors, which were produced by a bag-of-words model.

The learning-based techniques [34–37, 37–39] employ the encoder-decoder framework, where takes the changed code as the input for the encoder model and generates the commit message by the decoder model. For example, Jiang et al. [34] and Loyola et al. [35] utilized the sequential-based encoder-decoder model to generate messages. Liu et al. [37] further improved the sequence-to-sequence model by incorporating the pointer mechanism to mitigate out-of-vocabulary (OOV) problems. In addition, the latest work by Wang et al. [38] was proposed to combine retrieval-based and learning-based techniques to generate messages. Furthermore, the decay sampling mechanism [40, 41] was utilized to improve the performance on the testset.

2.2 Vulnerability Identification

Automated identification of software security vulnerabilities is a fundamental problem in security. Conventional vulnerability identification techniques can be generally categorized into three categories: static analysis [42–45], dynamic analysis [46–48] and symbolic analysis [49, 50]. The static analysis-based techniques rely on different structures of the code (e.g., AST, control flow, data dependency of a program) for the detection. Though the static analysers are usually lightweight, they tend to cause high false positives [51]. In contrast, dynamic-based techniques detect vulnerabilities by generating a massive amount of program inputs to execute the program for test. Dynamic analysers tend to have high accuracy, but they are computationally intensive. The symbolic analyzers detect the vulnerability based on the analysis of the program execution path [52].

There are also some conventional machine-learning based techniques (except neural networks) for vulnerability identification [53–55]. In these techniques, features hand-crafted by human experts are fed to machine learning algorithms for vulnerability detection. However, due to the complexity of different types of vulnerability, it is impractical to characterize all vulnerabilities by hand-crafted features.

Inspired by the power of automated feature extraction by deep learning techniques, more advanced works shift to deep learning techniques for vulnerability detection. The early works for vulnerability detection with deep learning approaches tend to take the program text as the input with CNNs [56] or LSTMs [8, 57–59] or combined both [58] to learn features automatically. However, these techniques just consider the program as the text, while ignoring their structures. Furthermore, some practitioners attempt to embed program structures to neural networks for the detection [60, 61]. For example, Lin et al. [60] proposed to traverse ASTs by the depth-first traversal to convert them into sequence with bidirectional LSTM for learning. Li et al. [61] encoded the statements that are related to the program dependency graph (PDG) to the bidirectional gated recurrent unit (BGRU) [62] to capture the dependency relations in a program for detection.

2.3 Source Code Summarization

Source code summarization, which targets generating high-quality comments in the natural language automatically, is a hot topic in software engineering. The hand-crafted template [63–65] is an early technique used in source code summarization. For example, Moreno et al. [65] generated the summary for the Java class by selecting the information from the stereotype information in conjunction with the predefined heuristics and combining it with a set of pre-defined rules. However, the hand-crafted template usually extracts the keywords to represent the program for generation, sometimes, the generated summary is meaningless.

In addition to the hand-crafted template, information retrieval techniques are widely used for source code summarization [66–71]. Generally, it retrieves a similar code snippet from a dataset and returns the corresponding summary of the code that matches the

given code. The common techniques used for code summarization are vector space model (VSM) [72], latent dirichlet allocation (LDA) [73], latent semantic indexing (LSI) [74]. The performance of these retrieval-based techniques depends on the dataset heavily. If a dataset does not have a similar code snippet compared with the given code, the generated summary is incorrect.

Recently, deep learning techniques have made breakthroughs for source code summarization [13, 75–81]. The early exploration in deep learning utilizes RNNs/LSTMs for source code summarization [13, 14, 75]. With the increasing popularity of the transformer, some practitioners also employ it in a supervised/unsupervised manner to generate code summary [17, 26, 82–84]. However, the majority of the above works ignore program structures for learning. To incorporate the program structure, LeClair et al. [78] proposed to employ AST with graph neural networks to generate the summary. Wan et al. [76] incorporated AST and the sequential text of the program into a deep reinforcement learning algorithm for source code summarization. In addition, Rencos [19] proposed to combine the neural networks with retrieval techniques for code summarization.

2.4 Code Search

Code search targets produce the semantic-equivalent program given the natural language query, which is meaningful for software developers. Similar to source code summarization, the early works [85–90] in code search utilized information retrieval to query the code by extracting both query and code characteristics. For example, Lu et al. [85] extended a query with some synonyms generated from WordNet to improve the hit ratio. McMillan et al. [91] proposed Portfolio which combines keyword matching to return the queried functions. CodeHow [86] extended the query with some APIs and utilized them with a boolean model to retrieve the matched programs. Despite these works could produce some accurate results, the performance of proposed techniques mainly relies on the searched code base and the semantic gap between the program and query is not addressed.

To learn semantics for the program and the query, some deep learning-based approaches were proposed [92–100] and these works attempted to use the deep learning techniques *e.g.*, LSTMs, CNNs, Transformer to learn high-dimensional representations for programs and queries. For example, DeepCS [101] encoded API sequence, tokens, and function names with multiple LSTMs to get the vector representations, and further encoded the queries into another vector with an extra LSTM for code search. CodeSearchNet [93] explored some basic encoders such as LSTM, CNN, and SelfAtt Encoder for code search. Furthermore, UNIF [94] proposed that using a simple bag-of-word model with an attention mechanism can achieve state-of-the-art performance. However, these DL-based techniques in code search are mostly based on sequential models and ignore the rich structural information behind the programs and queries. OCoR [102] proposed a neural network code search. Specifically, it represented each word by combing the vector representations of its characters to capture the overlap between the names. Furthermore, it designed an overlap matrix to represent the overlap degree between words in query and identifiers in code. Based on the character embedding and the overlap matrix, it further utilized a neural network for code search and utilized MRR to evaluate the model performance. Another work MMAN [103] encoded the program sequence, AST and CFG on a manually collected C dataset with LSTM, Tree-LSTM and GGNN to learn the representation of a program. It further encoded the query with another LSTM for multi-modal learning. The latest work G2SC [104] proposed to convert the code graphs into lossless sequences by a specific graph traversal strategy so that it can be flexibly integrated with existing models such as CodeBERT to better use the code graphs. Compared with this work, we propose to use graphs for both code and the natural language query for code search.

2.5 Pre-trained Models in Code Scenario

With the development of computational power, substantial works have demonstrated the pre-trained models are beneficial for various natural language processing (NLP) tasks and the transformer-based [16] pre-trained models such as BERT [105], RoBERTa [106], GPT [107], T5 [108] have attracted widespread attention. Generally, the pre-trained techniques usually train a model on a large unlabeled corpus with some

unsupervised learning objectives such as masked language modeling (MLM) [105], then this model is further fine-tuned on some specific downstream tasks with the labeled data to achieve the best performance.

Inspired by the advanced pre-trained models in NLP community, some practitioners [17, 18, 82, 83, 109–113] also employ the pre-trained techniques to learn a general model for different software engineering tasks. For example, Kanade et al. [109] pre-trained CuBERT with a massive amount of Python programs and then fine-tuned this model for some classification tasks such as variable misuse classification. Feng et al. [17] proposed CodeBERT, a bimodal pre-trained model for the programming language (PL) and natural language (NL) that learns the program representation to support code search and source code summarization. GraphcodeBERT [18] combined the variable data-flow graph of the program with the code sequence and natural language sequence to further improve CodeBERT. After that, CodeXGLUE [82] released a benchmark consisting of various software engineering tasks based on the pre-trained CodeBERT and CodeGPT [114]. In addition, Guo et al. [115] proposed UniXcoder, a unified cross-modal pre-trained model based on a multi-layer Transformer for "code intelligence". Specifically, it utilized mask attention matrices with prefix adapters for code understanding and generation. Liu et al. [116] proposed a CommitBART to support commit-related downstream tasks. Compared with existing pre-trained models, we illustrate they are not robust and further propose ContraBERT to enhance model robustness. Though pre-trained models for the program have confirmed the superior performance compared with supervised models, the analysis of the robustness for these pre-trained is less torched.

2.6 Graph Neural Networks

Graph Neural Networks (GNNs) [20,21,117,118] have attracted wide attention over the past years since GNNs can handle complex structured data, which contains the elements (nodes) with the relations (edges) between them. Hence, a variety of scenarios such as social networks [119], programs [120, 121], chemical and biological systems [122] leverage GNNs to model graph-structured data. Generally, a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

contains a node list $\mathcal{V}$ and the edge list $\mathcal{E}$, where $(u, v) \in \mathcal{E}$ denotes an edge from the node u to the node v. The learning for GNNs is to propagate neural messages (node features) in the neighboring nodes and this process is namely neural message passing. It consists of multiple hop computation. Specifically, at each hop, the node aggregates the received messages from its neighbors and updates the representation by combining the aggregated messages with its own previous features. Formally, for a node v has an initial representation $\boldsymbol{h}_v^0 \in \mathbb{R}^d$, where d is the dimensional length. The initial representation is usually derived from its label or the given features. Then, a hop updates its representation by the message passing and obtains the new representation $\boldsymbol{h}_v^1 \in \mathbb{R}^d$. This process can be expressed as follows:

$$\boldsymbol{h}_v^{(k)} = f(\boldsymbol{h}_v^{(k-1)}, \{\boldsymbol{h}_u^{(k-1)} | u \in N_v\}; \theta_k) \tag{2.1}$$

where N_v is a set of nodes that have edges to v. k denotes k-th hop and the total number of hops K is determined empirically as a hyper-parameter and $1 \leq k \leq K$. The function f usually distinguishes different GNN variants. For example, graph convolution networks (GCN) [21] can be expressed as:

$$\boldsymbol{h}_v^{(k)} = \sigma(\sum_{u \in N_v \cup \{v\}} \frac{1}{c_{v,u}} \boldsymbol{W} \boldsymbol{h}_u^{(k-1)}) \tag{2.2}$$

where $c_{v,u}$ is a normalization factor and it is usually set to $\sqrt{|N_v| \cdot |N_u|}$ or $|N_v|$, $\boldsymbol{W}$ is the learnable weight and σ is a non-linearity such as rectified linear unit (ReLU) [123]. Another widely used GNN variant in the program scenario is gated graph neural network (GGNN) [20], which uses a recurrent unit r (e.g., GRU [62] or LSTM [15]) for the update. The message passing can be calculated as follows:

$$\boldsymbol{h}_v^{(k)} = r(\boldsymbol{h}_v^{(k-1)}, \sum_{u \in N_v} \boldsymbol{W} \boldsymbol{h}_u^{(k-1)}; \boldsymbol{\theta}_r) \tag{2.3}$$

where $\boldsymbol{\theta}_r$ is the recurrent cell parameters and $\boldsymbol{W}$ is the learnable weight.

Chapter 3

ATOM: Commit Message Generation Based on AST

In this chapter, we demonstrate the detailed design of embedding program structures to the sequential-based models (i.e., LSTMs) for commit message generation to illustrate that program structures are effective for neural networks to learn program semantics.

3.1 Introduction

The program hosting platforms (e.g., GitHub) have been widely used in software development for software developers to reduce maintenance cost and time cost. However, software developers need to submit a commit message to record the changed code during the software update. High-quality commit messages are vital for developers to understand the software version evolution rapidly since these messages summarize the reason for the code changes in natural language, which is easy for developers to capture the high-level intuition without auditing implementation details.

However, writing commit messages is time-consuming for software developers. First, so far, the specification about the writing format of commit messages has not existed and software developers usually follow their own writing style. Second, there are a massive amount of commits without the corresponding commit messages. For instance,

according to the official report [124] released by SourceForge, there are nearly 14% of commits without the commit messages in more than 23,000 open-source Java projects. Furthermore, in our collected dataset, which has top-ranked ∼60 projects in terms of star numbers on GitHub, the meaningless commit messages account for around 10% of the entire collected commits. Hence, automated commit message generation to describe code changes in natural language is necessitated and meaningful in software engineering.

Automated commit message generation is a challenging task and several techniques have been proposed. The early works tend to utilize the rule-based techniques for solving (e.g., DeltaDoc [28] and ChangeScribe [29]). These techniques could summarize the code changes depending on the customized rules, however, these manually defined rules are limited and they cannot cover all cases. Furthermore, the generated messages tend to be verbose and meaningless. To address this limitation, Jiang et al. [34] proposed a sequential-based model, which adopted LSTMs for translating code changes into commit messages. However, the sequential-based models just consider the code as a flat sequence while ignoring the information of program syntax and semantics. In addition, Liu et al. [32, 33] attempted to directly reuse the existing commit messages in the collect dataset by some information retrieval (IR) techniques. The retrieval-based techniques could achieve promising results on similar programs but may generate poor on dissimilar programs.

To address the aforementioned challenges, we design a novel commit message generation model, namely ATOM (**A**bstract syntax **T**ree-based c**O**mmit **M**essage generation) for accurate commit message generation. ATOM extracts the syntax information on the abstract syntax tree (AST) of a program and takes it as the input for the model to learn program semantics to generate commit messages. In terms of evaluation, since the previous benchmark [34] cannot provide the completed function to construct ASTs, we build a new benchmark, which includes ∼160k commits and evaluate our approach on it. Extensive experimental results demonstrate that ATOM can significantly outperform the sequential-based models. The contributions of this work can be summarized as follows:

- We propose to encode the program syntax of the code changes based on ASTs to learn program semantics for commit message generation.
- We provide a new and well-cleaned benchmark, including complete function-level code snippets of ∼160k commits. We clean the benchmark by filtering out meaningless commits and make our benchmark [125] public.

3.2 Motivation and Background

In this section, we first discuss the motivation of our work, then introduce the background of a commit and some deep learning techniques that will be used.

3.2.1 Motivation

Existing works on commit message generation [34, 35, 126] generally consider code changes as a flat sequence, while ignoring the rich structural information for learning. We believe that ignoring program structures for the neural networks limits the model to learning program semantics. Hence, in this work, we target exploring abstract syntax tree (AST), which is usually the first-step representation of the program for better representing code changes. Furthermore, to better incorporate the syntax information with the sequential-based models such as LSTMs, inspired by the previous works [127, 128], we utilize AST paths to encode program syntax with LSTMs to generate commit messages.

3.2.2 Commit

The commit is used in Git to record the code changes between different software versions. A commit usually consists of the commit message and the code changes. The commit message is written by software developers in natural language to help developers understand the code changes. The code changes are usually called `diff`, recording the changed code between two software versions. A `diff` usually has one or multiple

FIGURE 3.1: A simple Java function.

```
1  public void printString()
2  {
3      String str = "ATOM"
4      for(int i = 0; i < 10; i++){
5          print(str);
6      }
7  }
```

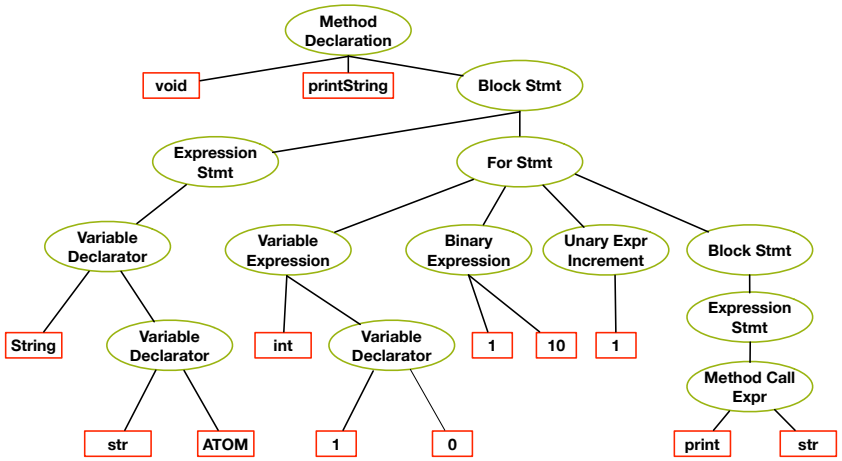


FIGURE 3.2: The AST of the function in Figure 3.1.

chunks in files. The modified code is wrapped by "@@" in a chunk with the negative sign "-" or positive sign "+" with a line number to denote the deleted or added statements.

3.2.3 Abstract Syntax Tree

An abstract syntax tree (AST) is the intermediate representation of a program, which usually consists of leaf nodes and non-leaf nodes. The leaf nodes refer to values, which represent identifiers and names from the code, while the non-leaf nodes represent some syntactic structure. Specifically, we present a simple code snippet in Figure 3.1 with its AST in Figure 3.2. We can observe that "str", "ATOM" are identifiers, which are leaf nodes in AST, while "ExpressionStmt", "ForStmt" are syntactic structures, which are non-leaf nodes. A total number of 106 different non-leaf nodes can be obtained by *JavaParser* [129], which is an AST parser for parsing Java programming language.

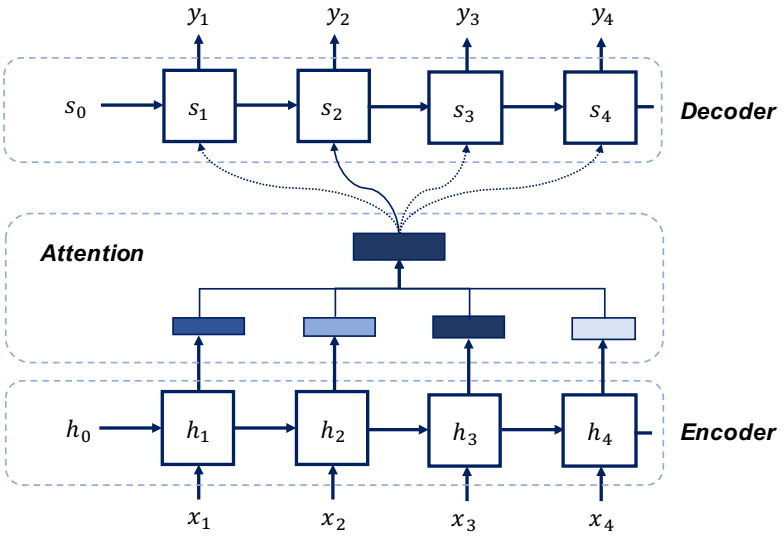


FIGURE 3.3: The sequential-based encoder-decoder model.

3.2.4 Sequential-based Encoder-Decoder Model

The architecture of the sequential-based encoder-decoder model is shown in Figure 3.3 and it takes the source sequence as the input to generate the target sequence. Usually, it consists of two RNNs [62] with the built-in LSTMs [15] for the generation. The attention mechanism [5, 130] is often used to further improve performance.

3.2.4.1 Recurrent Neural Network

RNNs are able to capture the sequential dependencies, hence they are widely used to model sequential data. The loop computation in RNNs allows information propagation from the previous step to the next step. At each step, the unit in RNNs takes current input as well as the hidden states, produced by the previous step to predict the current output. The chain-like structure enables RNNs to learn from the past, however, they also suffer from long-term dependencies. To mitigate the limitation of RNNs in learning long-distance information and handling the long sequence, some advanced variants are proposed such as Long Short-Term Memory (LSTM) [15] and Gated Recurrent Unit (GRU) [131].

3.2.4.2 Long Short-Term Memory

To capture long-distance dependencies that RNNs fail to learn, Hochreiter et al. [15] designed LSTMs. Specifically, LSTMs contain a memory cell with a gated mechanism to

help "forget" unimportant information. By the gated operation, the memory cell could preserve more long-distance dependencies compared with the vanilla RNNs. Hence, RNNs built with LSTMs are a common practice to model sequential data.

3.2.4.3 Attention Mechanism

The attention mechanism could further improve the generation performance of the encoder-decoder model. Compared with only utilizing the final hidden state as the context vector for the input to a decoder, it further considers all hidden states of the input sequence for the generation. Specifically, it computes a mapping matrix at each step of the decoder output to the encoder's hidden states. The attention weights are trained by a forward neural network to align the attention scores. Hence, at each step of the output, it has access to the entire input sequence and dynamically selects a specific element from the input for generation, which allows the decoder to place more attention on the relevant input. Two different variants of attention mechanism [5, 130] are widely used for the encoder-decoder model.

3.3 Approach

In this section, we first introduce the overview of ATOM and then detail each module that consists in ATOM.

3.3.1 Overview

ATOM consists of the following modules for commit message generation:

- *Preprocessing Module.* A commit usually consists of the commit message and code changes and we process them separately. For the code changes, we extract AST paths corresponding to the code changes by retrieving the completed functions in a repository. For the commit message, we take the first sentence with lemmatization operation of the commit message as the target to represent the entire commit message.

- *Generation Module.* We encode the extracted AST paths with bidirectional LSTMs to represent code changes and further utilize a LSTMs-based decoder with attention for generation.

3.3.2 Preprocessing Module

The code changes and the commit messages are preprocessed separately to prepare the input for ATOM.

3.3.2.1 Code Changes

Based on the corresponding sign "+" and "-" of `diffs`, which is shown in Figure 3.4, we can obtain *added* and *deleted* statements. We further tokenize these statements by pygments [132] and remove meaningless tokens such as punctuations. In this way, we can get a list of tokens for the *added* statement and *deleted* statement, which is defined as $W^{+/-}$, where $W^{+/-} = \{w_1, w_2, ..., w_i\}$ and i is the i-th token in the changed code. However, to obtain AST paths, we need to retrieve the completed functions from `diffs`. The reason to obtain the completed function is that AST parsing requires a basic compilation unit [133], which includes a single class definition and the wrapped functions. Specifically, we utilize Ctags [134] with the file path and modified line numbers in `diffs` to retrieve the completed functions and define the retrieved function as the *added* function and *deleted* function. We further parse these functions by JavaParser [129] to get ASTs. Once we obtain ASTs, to get `diffs`-related AST paths, we search the shortest path on AST for any two tokens (for example "indexSettings", "onOrAfter" in Figure 3.4) belonging to $W^{+/-}$ and denote this path as $x = \{w_i, n_1, ..., n_l, w_j\}$, where n_l is the l-th non-leaf node and w_i and w_j are the leaf nodes that values are in $W^{+/-}$. Following this process, we can obtain a set of `diffs`-related AST paths, denoting as $X^{+/-} = \{x_1^{+/-}, ..., x_{p/k}^{+/-}\}$, where p, k are the total number of AST paths for the *added*/*deleted* code respectively.

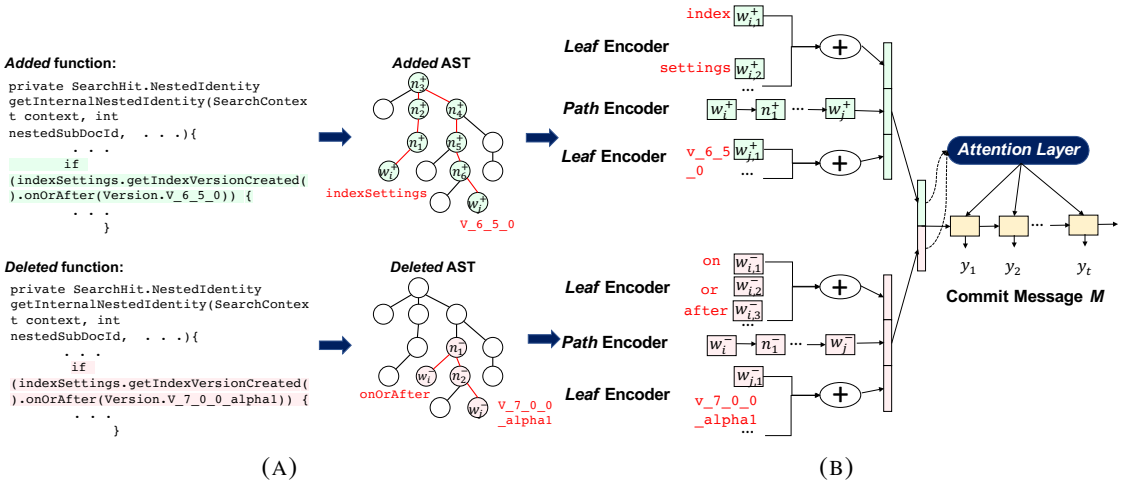


FIGURE 3.4: The network architecture of ATOM, where *added* and *deleted* function denote the completed functions retrieved from the `diff`. The highlighted path in *added* or *deleted* AST is one of the paths extracted from `diffs`.

3.3.2.2 Commit Message

We take the first sentence as the target to represent the entire commit message because the first sentence is often considered as the high summary of the entire commit message [34]. We further split these tokens with the underline (i.e., "_") and replace file names and digits with unique placeholders "<FILE>" and "<NUMBER>". To reduce the vocabulary set, we lemmatize each token to its base form by NLTK toolkit [135]. After this process, the obtained message is denoted as $M = \{y_1, y_2, ..., y_n\}$ where n is the length of a commit message.

3.3.3 Generation Module

The existing works for commit message generation usually consider `diffs` as a flat sequence, which ignores the program syntax information (e.g., AST) to generate the commit message. Inspired by the previous works [127, 128], we propose to encode AST paths about the changed code to generate the commit message. The network architecture of ATOM is illustrated in Figure 3.4, which involves three components: *AST Encoder* to encode an AST path to its vector representation; *Attention* to focus on the relevant AST paths dynamically; and *Message Decoder* to generate a commit message.

3.3.3.1 AST Encoder

Given a set of `diffs`-related AST paths (i.e., $X = \{x_1^{+/-}, x_2^{+/-}, ..., x_{p/k}^{+/-}\}$), for each path $x \in X$, we have $x = \{w_i, n_1, ..., n_l, w_j\}$. We encode x with bidirectional LSTM to obtain its representation.

- **Path Representation.** We encode the node type in an AST path by a bidirectional LSTM and take the final states of BiLSTM as the vector representation. Specifically, for a path $x = \{w_i^-, n_1^-, n_2^-, w_j^-\}$ in Figure 3.4, it can be expressed as follows:

$$\boldsymbol{h}_{w_i^+}, ..., \boldsymbol{h}_{w_j^+} = \text{BiLSTM}(\boldsymbol{E}^{\text{nodes}}_{w_i^+}, ..., \boldsymbol{E}^{\text{nodes}}_{w_j^+}) \tag{3.1}$$

$$\boldsymbol{path\_feat}^+ = [\boldsymbol{h}^{\leftarrow}_{w_i^+}; \boldsymbol{h}^{\rightarrow}_{w_j^+}] \tag{3.2}$$

$$\boldsymbol{h}_{w_i^-}, ..., \boldsymbol{h}_{w_j^-} = \text{BiLSTM}(\boldsymbol{E}^{\text{nodes}}_{w_i^-}, ..., \boldsymbol{E}^{\text{nodes}}_{w_j^-}) \tag{3.3}$$

$$\boldsymbol{path\_feat}^- = [\boldsymbol{h}^{\leftarrow}_{\boldsymbol{w}_i^-}; \boldsymbol{h}^{\rightarrow}_{w_j^-}] \tag{3.4}$$

  where $\boldsymbol{E}^{\text{nodes}}$ is the node type embedding matrix.

- **Leaf Representation.** Since the values of the start leaf node w_i and end leaf node w_j are also in `diff` text, we utilize another embedding matrix $\boldsymbol{E}^{\text{subtokens}}$ for the representation. Specifically, we split the values into subtokens, for example, for the token "onOrAfter" in Figure 3.4, we split it into "on", "or" and "after" and then sum the embeddings of these subtokens to represent a leaf node:

$$\boldsymbol{leaf\_feat}_{w^+} = \sum_{s \in \text{split}(w^+)} \boldsymbol{E}^{\text{subtokens}}_{w^+}[s] \tag{3.5}$$

$$\boldsymbol{leaf\_feat}_{w^-} = \sum_{s \in \text{split}(w^-)} \boldsymbol{E}^{\text{subtokens}}_{w^-}[s] \tag{3.6}$$

To represent a completed path (i.e.,x), we concatenate the path representation as well as the leaf representation and further use a fully connected layer to obtain the representation, which can be expressed as follows:

$$\boldsymbol{z}^{+/-} = \text{layer}([\boldsymbol{leaf\_feat}_{w_i^{+/-}}; \boldsymbol{path\_feat}^{+/-}; \boldsymbol{leaf\_feat}_{w_j^{+/-}}]) \tag{3.7}$$

Finally, we concatenate a number of *added* paths and *deleted* paths to represent the changed code:

$$\boldsymbol{Z} = [\boldsymbol{z}_1^+; ...; \boldsymbol{z}_p^+; \boldsymbol{z}_1^-; ...; \boldsymbol{z}_k^-] \tag{3.8}$$

3.3.3.2 *Attention*

By AST encoder, we obtain a set of path representations (i.e., $\boldsymbol{Z} = \{\boldsymbol{z}_1, \boldsymbol{z}_2, ..., \boldsymbol{z}_{p+k}\}$), where $p + k$ is the total number of paths, we use Luong attention [130] to focus on the important part. At each step of decoding, the attention mechanism will learn the weight distribution over the AST paths to dynamically capture the important parts.

3.3.3.3 Message Decoder

We average the vector representations of *added* and *deleted* paths (i.e., $\boldsymbol{Z} = \{\boldsymbol{z}_1, \boldsymbol{z}_2, ..., \boldsymbol{z}_{p+k}\}$) as the initial hidden states for the decoder, which can be expressed as follows:

$$\boldsymbol{h}_0 = \frac{1}{p+k} \sum_{i=1}^{p+k} \boldsymbol{z}_i. \tag{3.9}$$

At each decoding step t, a context vector $\boldsymbol{c}_t$ is calculated by the matrix $\boldsymbol{Z}$ and current hidden state $\boldsymbol{h}_t$ in the decoder:

$$\boldsymbol{\alpha}^t = \text{softmax}(\boldsymbol{h}_t \boldsymbol{W}_\alpha \boldsymbol{Z}), \qquad \boldsymbol{c}_t = \sum_i^{p+k} \boldsymbol{\alpha}_i^t \boldsymbol{z}_i. \tag{3.10}$$

where $\boldsymbol{\alpha}^t$ is the variable-length alignment vector whose size equals $p + k$ and $\boldsymbol{W}_\alpha$ is the learnable weight matrix. Then $\boldsymbol{h}_t$ and $\boldsymbol{c}_t$ are concatenated together for predicting target token y_t [130]:

$$p(y_t | y < t, z_1, ..., z_{p+k}) = \text{softmax}(\boldsymbol{W}_s \tanh(\boldsymbol{W}_c[\boldsymbol{c}_t; \boldsymbol{h}_t])) \tag{3.11}$$

where $\boldsymbol{W}_s$ and $\boldsymbol{W}_c$ are the learnable weight matrices.

During the learning process, we utilize softmax cross entropy to minimize the loss:

$$\text{loss} = y \log(\frac{e^{\text{logits}}}{\sum_r e^{\text{logits}}}). \tag{3.12}$$

where y is the label in the vocabulary set r and logits is the output value of the decoder.

3.4 Experimental Evaluation

In this section, we evaluate the performance of ATOM against some state-of-the-art baselines.

3.4.1 Setup

3.4.1.1 Experimental Benchmark

The previously used dataset [33,34,36] does not have commit ids or complete functions to extract AST. Hence, we crawled a new dataset. Specifically, we collected 56 popular projects from GitHub based on the "project stars". The raw messages of these commits are noisy and they may be empty or contain non-ASCII characters. Furthermore, the rollback or merge commits may have too many chunks, which are unsuitable for deep learning-based techniques. Hence, we remove them and keep 628,887 commits. In addition, we set a threshold of chunks as 5 to filter the project initialization and fundamental functionality updating commits and leave with 438,665 commits. We keep the commits, which are modified in the *.java* file for parsing AST. Finally, after removing the commits, whose message lengths are larger than 20 and the same contents, we keep $\sim$160k samples. Similar to Jiang et al. [34], we randomly select 10% for validation and testing and the remaining 80% of the dataset for training.

3.4.1.2 Experimental Settings

We set the max number of paths for *added* and *deleted* code to 80 to represent the changed code. The encoder consists of a bidirectional LSTM and a LSTM as a decoder. The dimension size is fixed to 256. To avoid overfitting, we set the dropout to 0.4. The maximum number of epochs is 3,000 with a batch size equals to 256 and the early stop is 20 to stop the training process. The learning rate is equal to 0.0001. We choose Adam [136] for optimization. The beam search with a beam width of 5 is selected at the testing phase. Hyper-parameters (e.g., embedding size, learning rate, encoder and decoder layers) are tuned on the validation set. The other hyper-parameters (e.g., batch

size and beam search width) are configured following Code2seq [127]. All experiments were conducted on servers with 36 cores and 4 Nvidia Graphics Tesla P40 and M40.

3.4.1.3 Evaluation Metrics

The automatic metrics such as BLEU-N [137], ROUGE-L [138], and Meteor [139] are selected as the evaluation metrics. These metrics have proved the effectiveness in measuring the text similarity between the generated sequences and ground truths.

- BLEU-*N*. BLEU-*N* calculates the n-gram precision of a generated sequence to the ground truth. Furthermore, it adds a penalty for short sentences. *N* usually chooses 1/2/3/4 to represent the scores of unigram, 2-grams, 3-grams and 4-grams and BLEU-*N* can be expressed as follows:

$$\text{BLEU-}N = \text{BP} * exp(\sum_{n=1}^{N} w_n \log p_n) \tag{3.13}$$

  where $N = 1, 2, 3, 4$, the uniform weights $w_n = 1/N$, and

$$\text{BP} = \begin{cases} 1, & \text{if } c > r \\ e^{1-r/c}, & \text{if } c \leq r \end{cases} \tag{3.14}$$

  where r is the length of the ground truth and c is the length of the generated sequence.

- ROUGE-L. ROUGE-L calculates F-score on the longest common subsequence (LCS). It is usually used in automatic summarization evaluation by computing the text similarity between two given sequences.

- Meteor. Meteor calculates the score based on a weighted F-score and a penalty function for incorrect word order, where F_{mean} is computed by unigram precision (P) and unigram recall (R) and Penalty is the ratio of the number of the matched chunks to the matched unigrams. This process can be expressed as follows:

$$\text{Meteor} = F_{\text{mean}}(1 - \text{Penalty}) \tag{3.15}$$

$$F_{\text{mean}} = \frac{10PR}{R + 9P} \tag{3.16}$$

$$\text{Penalty} = 0.5 * (\frac{\#\text{chunks}}{\#\text{unigrams\_matched}})^3 \tag{3.17}$$

3.4.2 Comparison Methods

The sequential-based encoder-decoder models NMT [34, 35, 126], CODISUM [36] are selected for comparison. We further add Commit2Vec [140] as another baseline. The details of these baselines are presented as follows:

- NMT [34, 35]. NMT adopts sequential-based encoder-decoder models (described in Section 3.2.4) for automated commit message generation. Specifically, Bahdanau and Luong attention are separately selected in Jiang et al. [34] and Loyola et al. [35] for the commit message generation. We also compare both, which are denoted as $\text{NMT}_{\text{(Luong)}}$ and $\text{NMT}_{\text{(Bahdanau)}}$.

- CODISUM [36]. CODISUM utilized the normalized code changes where the identifiers were unified with corresponding placeholders to learn the representation for code changes. Furthermore, it combined with the pointer network [141] to mitigate the Out-of-Vocabulary (OOV) issue to improve the performance.

- Commit2Vec [140]. Commit2Vec fed `diffs`-related AST paths to a fully-connected neural network to encode code changes for security-related commit identification. Though Commit2Vec aims at a binary classification problem, we can also add a decoder for commit message generation. Hence, we take it as a baseline by adding a LSTM-based decoder.

3.4.3 Experimental Results

In this section, we present the experimental results and provide an analysis by the following research questions.

TABLE 3.1: Comparison results with baseline approaches.

Methods	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor
$NMT_{(Luong)}$	13.12	8.01	6.11	5.23	12.73	10.37
$NMT_{(Bahdanau)}$	12.78	7.66	5.72	4.81	11.95	9.87
CODISUM	7.82	3.61	2.22	1.75	9.87	8.35
Commit2Vec	12.72	7.78	6.09	5.38	13.54	10.43
ATOM	**15.97**	**10.70**	**8.83**	**7.35**	**14.80**	**11.82**

3.4.3.1 What is the Performance of ATOM Comparing with Baseline Approaches?

Table 3.1 presents the results of our approach in line with the baseline approaches. Generally, we can observe that ATOM outperforms the baseline approaches significantly in terms of BLEU-N, ROUGE-L and Meteor. Specifically, we observe that ATOM outperforms the sequential-based encoder-decoder models such as $NMT_{(Luong)}$ or $NMT_{(Bahdanau)}$ significantly, which demonstrates that incorporating the structures of the program to LSTMs is effective for the model to learn program semantics and thus could provide the significant improvement. In contrast, these sequential-based encoder-decoder models only consider the program as a flat sequence, which is limited the model to learning program semantics. Furthermore, we can find that the performance of different attention mechanisms (i.e., Luong or Bahdanau) is close, which confirms that both of them are effective. In addition, we can also observe that the performance of Commit2Vec is lower than ATOM, which demonstrates that the fully-connected layer used in Commit2Vec is unable to learn the sequential dependencies in the AST path compared with the bidirectional LSTM.

Answer to RQ1: ATOM outperforms the sequential-based encoder-decoder models, which proves that incorporating the program structures to the conventional sequential-based models such as LSTMs is beneficial for the model to learn program semantics.

3.4.3.2 What is the Performance of ATOM with a Different Number of Paths?

ATOM encodes a set of AST paths extracted from the changed code to represent `diffs`, however, the total number of paths relies on the length of `diffs`. In ATOM, we set the maximum number of paths to 80 for the *added* and *deleted* code to represent the

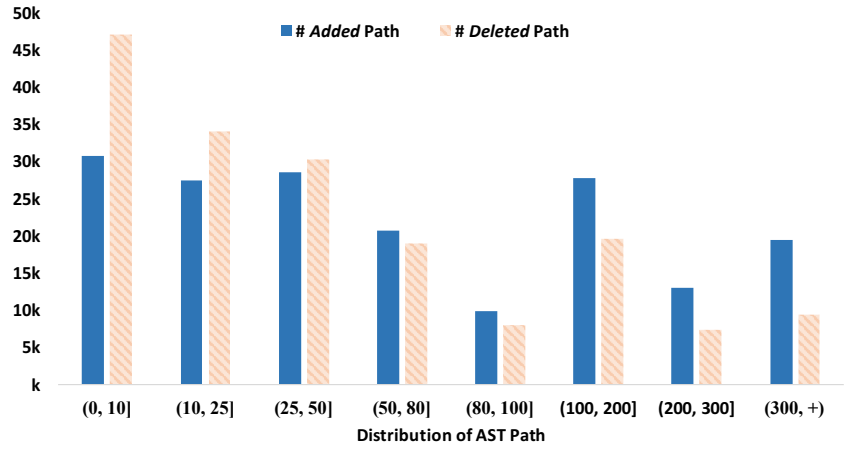


FIGURE 3.5: The statistical distribution of AST paths in our dataset. Each bar is the number of commits that has the number of AST paths in a specific interval.

changed code. We also want to investigate the performance of ATOM under a different number of paths. Specifically, we first statistically analyse the path distribution on our dataset. As shown in Figure 3.5, nearly 80% of commits have fewer than 80 AST paths in our dataset. Hence, for this experiment, we truncate AST paths to a specific number. For example, to validate the performance when setting the number to 30, we randomly select 30 paths from the set of AST paths, which number may be greater than 30.

The experimental results are presented in Table 3.2. We find that when setting the path number to 80, the values of BLEU-4, ROUGE-L and Meteor are the highest, which indicates that 80 is optimal. Furthermore, we also find that fewer paths usually have worse results (e.g., when the number of AST paths is set to 30, BLEU-4 drops dramatically). We believe that it is caused by the fewer paths having the limited capacity to represent code changes. Furthermore, with the increasing number of paths, the performance is not continuously improved. For example, when setting the path to 100 and furthermore, the scores even have a slight decrease when the number increased from 200 to 300. In addition, the larger number of paths will also increase the burden of model training. Hence, we summarize that 80 is an optimal value in our dataset.

Answer to RQ2: The optimal number of AST paths to represent `diffs` is 80 on our dataset. Fewer paths have the limited capacity to represent the changed code and more paths also cannot produce continuous improvement.

TABLE 3.2: The performance of different numbers of paths.

# Path	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor
30	11.53	6.57	4.80	4.27	13.66	10.22
50	13.67	8.70	6.83	5.96	13.92	10.89
80	**15.97**	**10.70**	**8.83**	**7.35**	**14.80**	**11.82**
100	15.11	10.01	8.19	7.09	14.34	11.42
200	13.89	8.83	6.88	6.07	14.01	11.01
300	13.72	8.77	6.85	6.04	13.95	10.99

3.5 Discussion

In this work, we just select the first sentence of the commit message as the target for the model to generate. Compared with the whole commit message, the first sentence usually summarises the entire commit message [34], but it is not absolute. We use this setting followed by other works [34, 93] and it can accelerate the training process. We leave the experiments about using the entire commit message as the target sequence as our future work.

3.6 Threats to Validity

One of the threats is the collected dataset. Our dataset contains more information than Jiang's [34], but more data is always beneficial to deep learning models. Another threat is that we only compare ATOM on our benchmark. As Jiang's [34] dataset does not provide commit ids, we cannot extract the *Added* and *Deleted* ASTs for encoding. Hence, we cannot verify the effectiveness of ATOM on Jiang's dataset.

3.7 Conclusion

In this chapter, we propose ATOM, an automated commit message generation system based on an abstract syntax tree. We propose to extract AST paths of the changed code to encode the syntax information of programs. We further utilize the bidirectional LSTM for encoding, followed by another LSTM as the decoder to generate commit messages. With our extensive experiments, we have confirmed that incorporating program structures to the sequential-based model such as LSTMs helps the model to learn program semantics better and thus produce promising results.

Chapter 4

Devign: Vulnerability Identification by GNNs

From Chapter 3, we have confirmed that incorporating ASTs of programs to the sequential-based models is beneficial for the model to learn program semantics, however, besides AST, a program tends to have other structures. Hence, in this chapter, we explore the way to combine diverse program structures with graph neural networks to learn the comprehensive program semantics for software vulnerability identification.

4.1 Introduction

Recently, the number of software vulnerabilities has increased rapidly, either reported publicly by CVE (i.e., Common Vulnerabilities and Exposures) or discovered internally. Furthermore, on one hand, the popularity of open-source libraries contributes to the increment, on the other hand, it also propagates the impact. These vulnerabilities caused by the insecure code can be exploited to attack the software system, which causes substantial damage.

Vulnerability detection is an important yet far-from-settled issue. Including some conventional techniques such as static analysis [42–45], dynamic analysis [46–48,142–147] and symbolic execution [49, 50], many advanced works have been made by applying

machine learning-based approaches. In the early works [148–150], security experts usually hand-crafted features and took them as the input for machine learning algorithms for detection. However, it is impractical to describe all vulnerability types with the hand-crafted features since the root causes of vulnerabilities change by the types of weaknesses [151] and libraries.

To avoid the efforts on feature extraction by human experts, recent works explore utilizing deep neural networks on vulnerability identification [8, 56, 59] to learn features automatically. However, these works fail to learn comprehensive program semantics to describe vulnerabilities of high complexity and diversity in the real program. On one hand, in terms of approaches, they only consider the program as a flat sequence, which analogizes to natural languages. However, source code is more formal and strict than natural languages and consists of various structures such as abstract syntax tree (AST), control flow, data flow and etc. Furthermore, some vulnerabilities are subtle, which requires investigating the comprehensive program semantics from different dimensions. Therefore, the weakness of the existing works limits to cover vulnerabilities. On the other hand, in terms of training data, Russell et al. [56] labeled the data by static analyzers, which caused high false positives. The simple artificial code used in Vuldeepecker [8] has the symbol "good" or "bad" inside the code to distinguish vulnerability or non-vulnerability, which is far less the complexity of the real code [152].

To address the above challenges, we propose a graph neural network (GNN) based neural network with composite programming representation for vulnerability identification. It ensures to encode a full set of program semantics to capture various vulnerability characteristics. We further innovate a *Conv* module, which takes the learnt node features from the graph neural network as the input to choose more coarse features by using the conventional convolutional and dense layers for graph-level classification. Furthermore, we manually labeled a dataset from 4 open-source projects in the C programming language to testify the capacity of the composite programming embedding for the program and the proposed model for vulnerability identification. We name this model *Devign* (Deep Vulnerability Identification via Graph Neural Networks).

- We take ASTs as the backbone and further encode the control flow and data dependency into a joint graph of various types of edges with each type denoting the connection regarding the corresponding representation. The comprehensive representation helps to capture as types of vulnerabilities as possible and ensures learning better node features by GNNs.

- We innovate a *Conv* module after the gated graph neural network for the graph-level classification. The *Conv* module enables learning hierarchically from the node features to capture the higher level of representations for graph-level classification tasks.

- We implement *Devign*, and evaluate its effectiveness through *manually* labeled data sets (*cost around 600 man-hours*) collected from the 4 popular C open-source projects. We release two datasets at our official website (https://sites.google.com/view/devign). The extensive experimental results demonstrate that *Devign* achieves an average 10.51% higher accuracy and 8.68% F1 score than baselines. Meanwhile, the *Conv* module brings an average 4.66% accuracy and 6.37% F1 gain. We apply *Devign* to 40 latest CVEs collected from 4 projects and get 74.11% accuracy, which indicates its usability of detecting new vulnerabilities.

4.2 Model Design

Fabian et al. [153] have confirmed that manually crafted vulnerability patterns with code property graphs by integrating the syntax and dependency semantics is effective for software vulnerability detection. Inspired by this, we design *Devign* to learn vulnerable features on code property graphs with graph neural networks [20]. Specifically, the architecture of *Devign* is shown in Figure 4.1, which includes the three components: (1) *Graph Embedding Layer of Composite Code Semantics*, which converts the raw source code of a function into a joint graph structure with comprehensive program semantics. (2) *Gated Graph Recurrent Layers*, which learn the node features by aggregating and passing information on neighboring nodes in graphs. (3) *The Conv module*, which extracts meaningful node features after the gated graph recurrent layers for graph-level prediction.

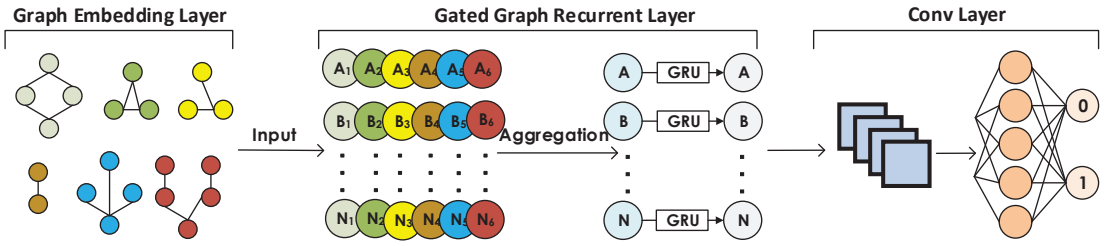


FIGURE 4.1: The Architecture of *Devign*

4.2.1 Problem Formulation

Compared with the majority of machine learning-based approaches detect vulnerability at the granularity level of a source file or an application (i.e., whether a source file or an application is potentially vulnerable [8, 59, 149, 153]), in this work, we target at analysing the vulnerable code at the *function-level*, which is the fine level of granularity for vulnerability detection.

We formalize this problem as a binary classification problem (i.e., learning to detect whether a given function is vulnerable or not). Let a sample of data be defined as $((c_i, y_i)|c_i \in \mathcal{C}, y_i \in \mathcal{Y}), i \in \{1, 2, \ldots, n\}$, where $\mathcal{C}$ denotes the set of functions in code, $\mathcal{Y} = \{0, 1\}^n$ represents the label set with 1 for vulnerable and 0 otherwise, and n is the number of instances. Since c_i is a function, we assume it is encoded as a multi-edged graph $g_i(V, A, \boldsymbol{X}) \in \mathcal{G}$ (See Section 4.2.2 for the embedding details). Let m be the total number of nodes in V, $A \in \{0, 1\}^{t \times m \times m}$ is the adjacency matrix, where t is the total number of edge types. An element $e^p_{s,t} \in A$ equal to 1 indicates that node v_s, v_t is connected via an edge of type p, and 0 otherwise. $\boldsymbol{X} \in \mathbb{R}^{m \times d}$ is the initial node feature matrix where each vertex v_j in V is represented by a d-dimensional real-valued vector $x_j \in \mathbb{R}^d$. The goal is to learn a mapping from $\mathcal{G}$ to $\mathcal{Y}$, $f : \mathcal{G} \mapsto \mathcal{Y}$ to predict whether a function is vulnerable or not. The prediction function f can be learned by minimizing the loss function:

$$\min \sum_{i=1}^{n} \mathcal{L}(f(g_i(V, A, \boldsymbol{X})), y_i|c_i)) + \lambda\omega(f) \tag{4.1}$$

where $\mathcal{L}(\cdot)$ is the cross entropy loss function, λ is an adjustable weight, and $\omega(\cdot)$ is a regularization.

4.2.2 Graph Embedding Layer of Composite Code Semantics

As shown in Figure 4.1, the graph embedding layer EMB is a mapping function from the code c_i to a graph data structure as the input for the model, which can be expressed as follows:

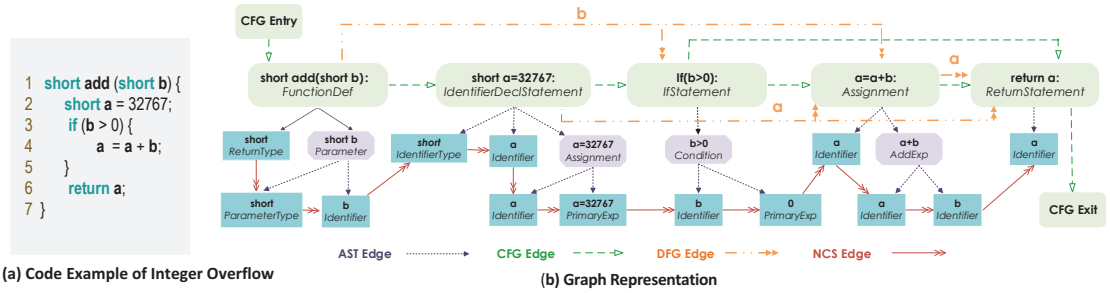
$$g_i(V, A, \boldsymbol{X}) = \mathrm{EMB}(c_i), \forall i = \{1, \ldots, n\} \tag{4.2}$$

We present the reason and approach to utilize the classical code representations to embed the code into a composite graph for feature learning as follows.

4.2.2.1 Classical Code Graph Representation and Vulnerability Identification

Traditional program analysis utilizes the various representations of the program to describe the program semantics behind the textual sequence and the classic representations include abstract syntax tree, control flow, and data flow that reveal the syntax and semantics of the source code. In addition, the most of vulnerabilities, for example, memory leaks are difficult to discover without considering the composite program semantics [153]. For instance, Fabian et al. [153] reported that utilizing ASTs alone is able to discover insecure arguments [153] while combining with control flow graphs, it enables to discover two more types of vulnerabilities (i.e., resource leaks and some use-after-free vulnerabilities). Furthermore, by integrating three types of code graphs, it is likely to cover the majority of types of vulnerabilities except two that require extra external information (i.e., race condition depending on runtime properties and design errors that are hard to model without details on the intention of a program).

Though *manually* crafted the vulnerability templates in the form of graph traversals [153], it provided the key insight and proved the feasibility to learn a broader range of vulnerability patterns through integrating properties of ASTs, data flow graphs and control flow graphs into a joint data structure. In addition to the above three classical code structures, we also consider the natural sequence of source code, since it is also confirmed the effectiveness [8,56] by the recent advanced works on deep learning-based vulnerability detection systems. Furthermore, it also complements the classic representations because its unique flat structure could capture the dependencies of code tokens in a 'human-readable' fashion.



(a) Code Example of Integer Overflow

(b) Graph Representation

FIGURE 4.2: Graph Representation of Code Snippet with Integer Overflow

4.2.2.2 Graph Embedding of Code

In this section, we briefly introduce each type of code representations and how we incorporate different subgraphs into one joint graph. We provide a code example of integer overflow as in Figure 4.2(a) with its graph representation as shown in Figure 4.2(b) for better illustration.

- Abstract Syntax Tree (AST). AST is a tree structure of source code. Usually, it is the first-step representation used by code parsers and it forms the basis for many other code representations and the node set of AST V^{ast} consists of all the nodes of the rest three code representations used in this paper. The major AST nodes are shown in Figure 4.2. All the boxes are AST nodes, with specific codes in the first line and node type annotated. The blue boxes are leaf nodes of AST and purple arrows represent the child-parent AST relations.

- Control Flow Graph (CFG). CFG describes all paths that might be traversed through a program during its execution. The executed path is determined by conditional statements e.g., *for*, *if*, and *switch* statements. In CFGs, nodes denote statements and conditions, and they are connected by directed edges to indicate the transfer of control. The CFG edges are highlighted with green dashed arrows in Figure 4.2. In particular, the flow starts from the entry and ends at the exit, and two different paths derive at the *if* statements.

- Data Flow Graph (DFG). DFG tracks the usage of variables throughout the CFG. Data flow is variable-oriented and any data flow involves the access or modification of certain variables. A DFG edge represents the subsequent access or modification of

the same variables. It is illustrated by orange double arrows in Figure 4.2 and with the involved variables annotated over the edge. For example, the parameter b is used in both the *if* condition and the assignment statement.

- Natural Code Sequence (NCS). In order to encode the natural sequential order of the source code, we use NCS edges to connect neighboring code tokens in ASTs. The reason to introduce this type of edge is to reserve the programming logic reflected by the sequence of source code. The NCS edges are denoted by red arrows in Figure 4.2, connecting all the leaf nodes of the AST.

Thus, a function c_i can be denoted by a joint graph g with four types of subgraphs (i.e., 4 types of edges) sharing the same set of nodes $V = V^{ast}$. As shown in Figure 4.2, each node $v \in V$ has two attributes (i.e., *Type* and *Code*), where *Type* denotes the node type attribute and *Code* contains the source code. We encode *Code* by a pre-trained word2vec model, which is trained on the code corpus built on the whole source code files in the projects, and encode *Type* by label encoding. Then, we concatenate the two encodings as the initial node representation $\boldsymbol{x}_v$.

4.2.3 Gated Graph Recurrent Layers

The main idea of graph neural networks (GNNs) is to interact with the node representation from local neighborhoods by the message passing. Based on different kinds of message passing, there are a variety of GNN variants such as gated graph recurrent networks [20], GraphSAGE [22] and graph convolutional networks [21]. Following the previous work [120], we select a gated graph recurrent network (GGNN) for the node embedding.

Given an initialized graph $g_i(V, A, \boldsymbol{X})$, for each node $v_j \in V$, we initialize the node state vector $\boldsymbol{h}_j^{(1)} \in \mathbb{R}^z, z \geq d$ using the initial annotation by copying $\boldsymbol{x}_j$ into the first dimensions and padding extra 0's to allow hidden states that are larger than the annotation size (i.e., $\boldsymbol{h}_j^1 = [\boldsymbol{x}_j^\top, \boldsymbol{0}]^\top$). Let K be the total number of hops for message passing. To propagate information throughout graphs, at each hop $k \leq K$, all nodes communicate with each other by passing information via edges dependent on the edge type, which is

denoted as p^{th} adjacent matrix A_p of A and this process can be expressed as follows:

$$\boldsymbol{a}_{j,p}^{(k-1)} = A_p^{\top}\left(\boldsymbol{W}_p\left[\boldsymbol{h}_1^{(k-1)\top}, \ldots, \boldsymbol{h}_m^{(k-1)\top}\right] + \boldsymbol{b}\right) \tag{4.3}$$

where $\boldsymbol{W}_p \in \mathbb{R}^{z\times z}$ is the learnable weight and $\boldsymbol{b}$ is the bias. In particular, a new state $\boldsymbol{a}_{j,p}$ of node v_j is calculated by aggregating information of all neighboring nodes defined on the adjacent matrix A_p on edge type p. The remaining steps are a gated recurrent unit (GRU) that incorporates information from all types with node v at the previous hop to get the current node's hidden state $\boldsymbol{h}_{i,v}^{(k)}$, i.e.,

$$\boldsymbol{h}_j^{(k)} = \mathrm{GRU}(\boldsymbol{h}_j^{(k-1)}, \mathrm{AGG}(\{\boldsymbol{a}_{j,p}^{(k-1)}\}_{p=1}^{t})) \tag{4.4}$$

where t denotes the total edge types and $\mathrm{AGG}(\cdot)$ denotes an aggregation function that could be one of the functions $\{\mathrm{MEAN}, \mathrm{MAX}, \mathrm{SUM}, \mathrm{CONCAT}\}$ to aggregate the information from different edge types to compute the next hop node embedding $\boldsymbol{h}^{(k)}$. We use the SUM function in the implementation. The above propagation procedure iterates over K steps, and the state vectors at the last hop $\boldsymbol{H}_i^{(K)} = \{\boldsymbol{h}_j^{(K)}\}_{j=1}^{m}$ is the final node representation matrix over the node set V.

4.2.4 The Conv Layer

The generated node features from the gated graph recurrent layers can be used as input to any prediction layer (e.g., for node or link or graph-level prediction), and then the whole model can be trained in an end-to-end fashion. In our problem, we require to perform the task of graph-level classification to determine whether a function c_i is vulnerable or not. The standard approach to graph classification is gathering all these generated node embeddings globally e.g., using a linear weighted summation to flatly add up all the embeddings [20, 154] as shown in Eq (4.5),

$$\tilde{y}_i = \mathrm{Sigmoid}\left(\sum \mathrm{MLP}([\boldsymbol{H}_i^{(K)}; \boldsymbol{x}_i])\right) \tag{4.5}$$

where $\mathrm{sigmoid}$ function is used for classification and MLP denotes a Multilayer Perceptron (MLP) that maps the concatenation of $\boldsymbol{H}_i^{(K)}$ and $\boldsymbol{x}_i$. This kind of approach hinders effective classification over entire graphs [155, 156].

Thus, we design the *Conv* module to select nodes that are relevant to the current graph-level task. Previous works in [156] proposed to use a SortPooling layer after the graph convolution layers to sort the node features in a consistent node order for graphs without fixed ordering so that traditional neural networks can be added after it and trained to extract useful features characterizing the rich information encoded in graph. In our problem, each code representation graph has its own predefined order and connection of nodes encoded in the adjacent matrix, and the node features are learned through gated recurrent graph layers instead of graph convolution networks that require sorting the node features from different channels. Therefore, we directly apply 1-D convolution and dense neural networks to learn features relevant to the graph-level task for more effective prediction[1]. We define $\sigma(\cdot)$ as a 1-D convolutional layer with maxpooling, then

$$\sigma(\cdot) = \mathrm{MAXPOOL}\big(\mathrm{Relu}\big(\mathrm{CONV}(\cdot)\big)\big) \tag{4.6}$$

Let l be the number of convolutional layers applied, then the *Conv* module, can be expressed as

$$\boldsymbol{Z}_i^{(1)} = \sigma\big([\boldsymbol{H}_i^{(K)}, \boldsymbol{x}_i]\big), \ldots, \boldsymbol{Z}_i^{(l)} = \sigma\big(\boldsymbol{Z}_i^{(l-1)}\big) \tag{4.7}$$

$$\boldsymbol{Y}_i^{(1)} = \sigma\big(\boldsymbol{H}_i^{(K)}\big), \ldots, \boldsymbol{Y}_i^{(l)} = \sigma\big(\boldsymbol{Y}_i^{(l-1)}\big) \tag{4.8}$$

$$\tilde{y}_i = \mathrm{Sigmoid}\big(\mathrm{AVG}(\mathrm{MLP}(\boldsymbol{Z}_i^{(l)}) \odot \mathrm{MLP}(\boldsymbol{Y}_i^{(l)}))\big) \tag{4.9}$$

where we firstly apply traditional 1-D convolutional layers and dense layers respectively on the concatenation $[\boldsymbol{H}_i^{(K)}; \boldsymbol{x}_i]$ and the final node features $\boldsymbol{H}_i^{(K)}$, followed by a pairwise multiplication on the two outputs, then an average aggregation on the resulted vector is selected, and at last the sigmoid function to make a prediction.

4.3 Evaluation

We compare *Devign* with a number of state-of-the-art vulnerability identification techniques, with the goal of understanding the following questions:

[1]We also tried LSTMs and BiLSTMs (with and without attention mechanisms) on the sorted nodes in AST order, however, the convolution networks work best overall.

- **RQ1:** How does our *Devign* compare to the other learning-based vulnerability identification approaches?
- **RQ2:** How does our *Conv* module powered *Devign* compare to the *Ggrn* with the flat summation in Eq 4.5 for the graph-level classification task?
- **RQ3:** Can *Devign* learn from each type of the code representations (e.g., a single-edged graph with one type of information)? And how do the *Devign* models with the composite graphs (e.g., all types of code representations) compare to each of the single-edged graphs?
- **RQ4:** How does *Devign* perform on the latest vulnerabilities reported publicly through CVEs?

4.3.1 Data Preparation

It is nontrivial to get high-quality data sets of vulnerable functions due to the collection requires qualified expertise. We noticed that though the released data sets of vulnerable functions [59], the labels are generated by statistic analyzers which are not reliable. Other potential datasets used in [157, 158] are not available. In this work, with the support of our industrial partners, we invested a team of security experts to collect and label the data from scratch. In addition to the raw function collection, we also need to generate graph representations for each function. We describe the details below.

Raw Data Gathering To test the capability of *Devign*, we evaluate on manually-labeled functions collected from 4 large open-source projects in C programming language that are popular among developers and diversified in functionality (i.e., Linux, QEMU, Wireshark, and FFmpeg).

To ensure the quality of data labelling, we started by collecting security-related commits, which we would label as vulnerability-fix commits or non-vulnerability fix commits, and then extracted vulnerable or non-vulnerable functions directly from the labeled commits. The vulnerability-fix commits (VFCs) aim at fixing potential vulnerabilities, from which we can extract vulnerable functions from the source code of versions previous to the revision made in these commits.

The non-vulnerability-fix commits (non-VFCs) are commits that do not fix any vulnerability, similarly from which we can extract non-vulnerable functions from the source code before the modification. We follow the similar approach proposed in [158] to collect the commits. It consists of the following two steps. (1) *Commits Filtering*. Since only a tiny part of commits are vulnerability-related, we exclude the security-unrelated commits whose messages are not matched by a set of security-related keywords such as DoS and injection. The rest are left for manual labelling, which is more likely security-related. (2) *Manual Labelling*. Four professional security researchers spent totally *600 man-hours* to finish a two-round data labelling and cross-verification. Given a VFC or non-CFC, we extract the functions and assign the corresponding labels.

Graph Generation We utilize Joern [153], an open-source code analysis tool based on code property graphs, to extract ASTs and CFGs for functions. However, due to some compile errors and exceptions of Joern, we can only construct ASTs and CFGs for part of functions. We filter out these functions that cannot have ASTs or CFGs or have obvious errors in them. We substitute DFGs with three relations (i.e., *LastRead (DFG_R)*, *LastWrite (DFG_W)*, and *ComputedFrom (DFG_C)* [120], since the original DFGs edges are labeled with the used variables, which greatly increases the number of edges and complicates the graphs. Specifically, *DFG_R* represents the immediate last read of each occurrence of the variable. Each occurrence can be directly recognized from the leaf nodes of ASTs. *DFG_W* represents the immediately last write of each occurrence of variables. Similarly, we make these annotations to the leaf node variables. *DFG_C* determines the sources of a variable. In an assignment statement, the left-hand-side variable is assigned a new value by the right-hand-side expression. Furthermore, we filter out functions whose node size larger than 500 for efficiency and these samples

TABLE 4.1: The statistics of the dataset.

Project	Sec. Rel. Commits	VFCs	Non-VFCs	Graphs	Vul Graphs	Non-Vul Graphs
Linux	12811	8647	4164	16583	11198	5385
QEMU	11910	4932	6978	15645	6648	8997
Wireshark	10004	3814	6190	20021	6386	13635
FFmpeg	13962	5962	8000	6716	3420	3296
Total	48687	23355	25332	58965	27652	31313

accounts for 15%. Table 4.1 shows the statistics of the dataset.

4.3.2 Baseline Methods

We compare *Devign* with the state-of-the-art vulnerability detection approaches, as well as the gated graph recurrent network (*Ggrn*) that used the linearly weighted summation for classification.

- Metrics + Xgboost [157]: We collected 11 vulnerability metrics and 4 complexity metrics for each function by Joern and further utilized Xgboost for classification. Here we did not use the proposed binning and ranking method in the original paper since it was not a learning-based approach. We search the best parameters via Bayes Optimization [159].

- 3-layer BiLSTM [8]: It considered the source code as a flat sequence and fed the tokenized code into bidirectional LSTMs with initial embeddings trained by Word2vec for the detection. We implemented a 3-layer bidirectional for the best performance.

- 3-layer BiLSTM + Att: It is an improved version of Li et al. [8] with the attention mechanism [160] to improve the detection accuracy.

- CNN [56]: Similar to Li et al. [8], it took source code as natural languages and utilized the bag of words to the token embedding and then fed them to CNNs to learn.

4.3.3 Performance Evaluation

***Devign* Configuration** We set the dimension of word2vec for the initial node representation to 100 in the embedding layer. The dimension of hidden states in the gated graph recurrent layer is 200 and the number of hops is 6. For the parameters of *Conv* in

TABLE 4.2: Classification accuracies and F1 scores in percentages: The two far-right columns give the maximum and average relative difference in accuracy/F1 compared to *Devign* model with the composite code representations (i.e., *Devign* (Composite)).

Method	Linux		QEMU		Wireshark		FFmpeg		Combined		Max Diff		Avg Diff	
	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1
Metrics + Xgboost	67.17	79.14	59.49	61.27	70.39	61.31	67.17	63.76	61.36	63.76	14.84	11.80	10.30	8.71
3-layer BiLSTM	67.25	80.41	57.85	57.75	69.08	55.61	53.27	69.51	59.40	65.62	16.48	15.32	14.04	8.78
3-layer BiLSTM + Att	75.63	82.66	65.79	59.92	74.50	58.52	61.71	66.01	69.57	68.65	8.54	13.15	5.97	7.41
CNN	70.72	79.55	60.47	59.29	70.48	58.15	53.42	66.58	63.36	60.13	16.16	13.78	11.72	9.82
Ggrn (AST)	72.65	81.28	70.08	66.84	79.62	64.56	63.54	70.43	67.74	64.67	6.93	8.59	4.69	5.01
Ggrn (CFG)	78.79	82.35	71.42	67.74	79.36	65.40	65.00	71.79	70.62	70.86	4.58	5.33	2.38	2.93
Ggrn (NCS)	78.68	81.84	72.99	69.98	78.13	59.80	65.63	69.09	70.43	69.86	3.95	8.16	2.24	4.45
Ggrn (DFG_C)	70.53	81.03	69.30	56.06	73.17	50.83	63.75	69.44	65.52	64.57	9.05	17.13	6.96	10.18
Ggrn (DFG_R)	72.43	80.39	68.63	56.35	74.15	52.25	63.75	71.49	66.74	62.91	7.17	16.72	6.27	9.88
Ggrn (DFG_W)	71.09	81.27	71.65	65.88	72.72	51.04	64.37	70.52	63.05	63.26	9.21	16.92	6.84	8.17
Ggrn (Composite)	74.55	79.93	72.77	66.25	78.79	67.32	64.46	70.33	70.35	69.37	5.12	6.82	3.23	3.92
Devign (AST)	**80.24**	84.57	71.31	65.19	79.04	64.37	65.63	71.83	69.21	69.99	3.95	7.88	2.33	3.37
Devign (CFG)	80.03	82.91	74.22	70.73	79.62	66.05	66.89	70.22	71.32	71.27	2.69	3.33	1.00	2.33
Devign (NCS)	79.58	81.41	72.32	68.98	79.75	65.88	67.29	68.89	70.82	68.45	2.29	4.81	1.46	3.84
Devign (DFG_C)	78.81	83.87	72.30	70.62	79.95	66.47	65.83	70.12	69.88	70.21	3.75	3.43	2.06	2.30
Devign (DFG_R)	78.25	80.33	73.77	70.60	80.66	66.17	66.46	72.12	71.49	70.92	3.12	4.64	1.29	2.53
Devign (DFG_W)	78.70	84.21	72.54	71.08	80.59	66.68	67.50	70.86	71.41	71.14	2.08	2.69	1.27	1.77
Devign (Composite)	79.58	**84.97**	**74.33**	**73.07**	**81.32**	**67.96**	**69.58**	**73.55**	**72.26**	**73.26**	-	-	-	-

Devign, we apply (1, 3) filter with ReLU function for the first convolution layer and followed by a max pooling layer with (1, 3) filter and (1, 2) stride. The second convolution layer also has (1, 1) filter with a max pooling layer with (2, 2) filter and (1, 2) stride. We utilize Adam optimizer with a learning rate of 0.0001 and batch size of 128 for the learning process. We further use $L2$ regularization to avoid overfitting and 100-epoch for an early stop. We randomly shuffle the dataset and split it with the ratio of 75% for the training and the rest 25% for validation. We train *Devign* on a server with Nvidia Graphics Tesla M40 and P40.

Results Analysis The evaluation metrics are textitaccuracy and *F1 score*. Table 4.2 shows all experimental results. First, we analyze the results regarding **RQ1**. By comparing the results between the baseline approaches, *Ggrn* and *Devign* with composite code representations, we can find that both *Ggrn* and *Devign* outperform the baselines in all data sets significantly. Especially, compared to all the baseline approaches, the relative accuracy gain by *Devign* is averagely 10.51%, at least 8.54% on the QEMU dataset. *Devign* (Composite) also outperforms 4 baseline methods in F1 score, i.e., the relative gain of F1 score is 8.68% on the average and the minimum relative gains on each dataset (Linux, QEMU, Wirshark, FFmpeg and Combined) are 2.31%, 11.80%, 6.65%, 4.04% and 4.61% respectively. *Devign* achieves the highest F1 score (i.e., 84.97%)

among all datasets, we believe that it is caused by Linux following the best practices of coding style. *In summary, Devign with comprehensive semantics encoded in graphs outperforms existing state-of-the-art vulnerability identification approaches.*

Next, we investigate the performance gain of *Devign* against *Ggrn* (The answer to **RQ2**). First, we observe the values with the composite code representation and we find that *Devign* reaches higher accuracy (an average of 3.23%) than *Ggrn* in all datasets and the highest accuracy gain is 5.12% on FFmpeg dataset. Furthermore, *Devign* gets a higher F1, an average of 3.92% than *Ggrn*, where the highest F1 gain is 6.82 % on the QEMU dataset. In addition, we check the value with each single code representation and we can obtain a similar conclusion that generally *Devign* outperforms *Ggrn* by a significant margin, where the maximum accuracy gain is 9.21% for DFG_W edge and the maximum F1 gain is 17.13% for DFG_C. *Overall, the average accuracy and F1 gain by Devign compared with Ggrn are 4.66%, 6.37%, which demonstrates that the Conv module is more effective in learning related features for the graph-level vulnerability identification.*

Then we check the results for **RQ3**. It is surprising that the results by the single-edged graphs are quite encouraging in both *Ggrn* and *Devign*. We find that the accuracy of *Ggrn* in some specific types of edges is slightly higher than that in the composite graph for example, CFG and NCS graphs have better results on the FFmpeg and combined dataset. When it turns to *Devign*, except the Linux, the accuracy of the composite graph representation is superior to any single-edged graph with the gain ranging from 0.11% to 3.75%. In addition, the improvement F1 score brought by the composite graph is averagely 2.69%, ranging from 0.4% to 7.88%, compared with the single-edged graphs in *Devign*. *To sum up, composite graphs help Devign to learn more comprehensive program semantics for vulnerability identification than the single-edged graphs.*

Finally, to answer **RQ4**, we collect the latest 10 CVEs of each project to investigate whether *Devign* can be applied to identify zero-day vulnerabilities. We obtain 112 vulnerable functions in total based on the fixed commits of these 40 CVEs. We take these vulnerable functions as the input to the trained Devign model and achieve an average

accuracy of 74.11%. *The results demonstrate the capacity of Devign in discovering new vulnerabilities in the real scenario.*

4.4 Threats to Validity

One of the threats lies in the compared baselines. In this work, the implementation of these baselines is finished by ourselves. To reduce this threat, the co-authors carefully check the correctness of our implementation. Another threat is the used open-source tool Joern for graph construction. It may contain some parsing errors, especially for some complex functions. To reduce this threat, we filter out those graphs that have obvious parsing errors by some self-defined rules such as removing the graph which has empty nodes.

4.5 Conclusion

In this chapter, we propose a vulnerability identification approach *Devign*, which encodes a variety of program structures into a joint graph with graph neural networks to learn program semantics for discovering vulnerable functions. With extensive experiments, we have confirmed that *Devign* achieved a new state-of-the-art performance on the real collected vulnerability dataset. In the next chapter, we explore the way to incorporate program structures for automatic source code summarization.

Chapter 5

Retrieval-augmented Generation for Code Summarization

From Chapter 4, we encode the comprehensive program semantics with graph neural networks (GNNs) for software vulnerability identification and achieve state-of-the-art performance. In this chapter, we aim to incorporate the program structures for the automatic source code summarization, which summarizes the functionality of a code snippet in the natural language.

5.1 Introduction

With software systems growing in complexity and size, nearly 90% [76] efforts are cost by software developers on software maintenance (e.g., bug fix and software version iteration) in software development. Source code summary, which is in a form of natural language, plays a vital role in the comprehension and maintenance process. A high-quality summary could greatly reduce the effort of the developer to read and understand the program. However, manually writing code summaries is time-consuming and tedious. With the acceleration of software iteration, it has become a heavy burden for software developers. Thus, source code summarization which automates concise descriptions of programs is important and meaningful.

Automatic source code summarization is a crucial yet far-from-the-settled problem. There are the following key challenges: (1) the source code and the natural language summary are heterogeneous, they may not share common lexical tokens, synonyms, or language structures (2) the source code is complex with complicated logic and variable grammatical structure, making it difficult to learn program semantics. Traditionally, information retrieval (IR) techniques have been widely used in code summarization [66–69]. As code duplication [161, 162] is common in the "big code" era [163], early works summarize the new programs by retrieving the similar code snippet in the existing code database and use its summary directly. In essence, the retrieval-based approaches convert the code summarization to the code similarity calculation task, which may achieve promising results on similar programs, but are limited in generalization (i.e., they have poorer performance on programs that are very different from the code database). To improve the generalization performance, recent works focus on generation-based approaches. Some works explore Seq2Seq architectures [5, 130] to generate summaries from the given source code. The Seq2Seq-based approaches [13, 75, 127] usually consider the source code or abstract syntax tree parsed from the source code as a flat sequence and follow a paradigm of encoder-decoder with the attention mechanism for generating a summary. However, these works only rely on sequential-based models, which are struggling to capture the rich semantics of source code (e.g., control dependencies and data dependencies). Furthermore, generation-based approaches typically cannot take advantage of similar examples from the retrieval database, as retrieval-based approaches do.

To better learn the semantics of the source code, Allamanis et al. [120] lighted up this field by representing programs as graphs. Some follow-up works [164] attempted to encode more code structures (e.g, control flow, program dependencies) into code graphs with graph neural networks (GNNs), and achieved more promising results than the sequence-based approaches. Existing works [120, 164] usually convert code into graph-structured input during preprocessing and directly consume it via modern neural networks (e.g., GNNs) for computing node and graph embeddings. However, most GNN-based encoders only allow message passing among nodes within a K-hop neighborhood (where K is usually set to a small number such as 4) to avoid over-smoothing [165, 166],

thus capture only local neighborhood information and ignore global interactions among nodes. Even there are some works [167] that try to address this challenge with deep GCNs (i.e., 56 layers) [21] by the residual connection [168], however, the computation cost cannot endure in the program, especially for a large and complex program.

To address these challenges, we propose a framework for automatic code summarization, namely Hybrid GNN *(HGNN)*. Specifically, from the source code, we first construct a code property graph (CPG) based on the abstract syntax tree (AST) with different types of edges (i.e., Flow To, Reach). In order to combine the benefits of both retrieval-based and generation-based methods, we propose a *retrieval-based augmentation mechanism* to retrieve the source code that is most similar to the current program from the retrieval database (excluding the current program itself) and add the retrieved code as well as the corresponding summary as auxiliary information for training the model. In order to go beyond local graph neighborhood information and capture global interactions in the program, we further propose an attention-based dynamic graph by learning global attention scores (i.e., edge weights) in the augmented static CPG. Then, a hybrid message passing (HMP) is performed on both static and dynamic graphs. We also release a new code summarization benchmark by crawling data from popular and diversified projects containing **95k+** functions in *C* programming language and make it public [1]. We highlight our main contributions as follows:

- We propose a general-purpose framework for automatic code summarization, which combines the benefits of both retrieval-based and generation-based methods via a retrieval-based augmentation mechanism.
- We innovate a Hybrid GNN by fusing the static graph (based on code property graph) and dynamic graph (via structure-aware global attention mechanism) to mitigate the limitation of the GNN on capturing global graph information.
- We release a new challenging *C* benchmark for the task of source code summarization.

[1] https://github.com/shangqing-liu/CCSD-benchmark-for-code-summarization

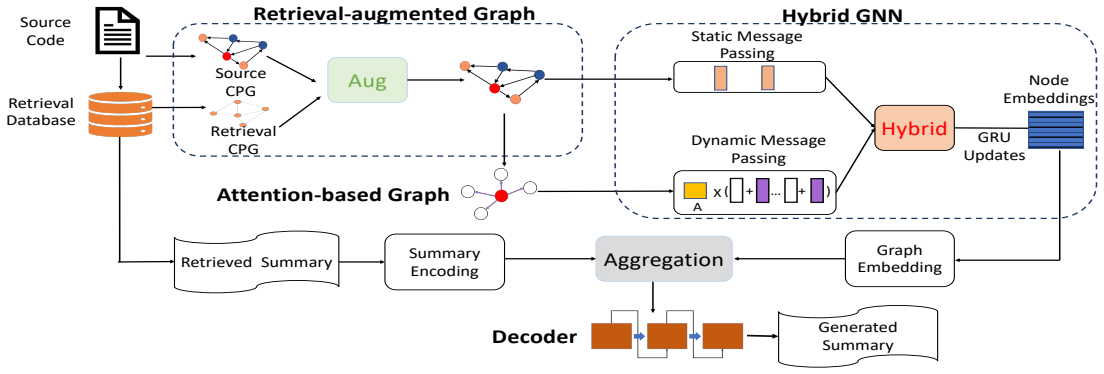


FIGURE 5.1: The overall architecture of the proposed *HGNN* framework.

- We conduct an extensive experiment to evaluate our framework. The proposed approach achieves state-of-the-art performance and improves existing approaches by **1.42**, **2.44** and **1.29** in terms of BLEU-4, ROUGE-L and METEOR metrics.

5.2 Hybrid GNN Framework

In this section, we introduce the proposed framework Hybrid GNN *(HGNN)*, as shown in Figure 5.1, which mainly includes four components: (1) Retrieval-augmented Static Graph Construction *(c.f.,* Section 5.2.2), which incorporates retrieved code-summary pairs to augment the original code for learning. (2) Attention-based Dynamic Graph Construction (*c.f.,* Section 5.2.3), which allows message passing among any pair of nodes via a structure-aware global attention mechanism. (3) *HGNN*, (*c.f.,* Section 5.2.4), which incorporates information from both static graphs and dynamic graphs with Hybrid Message Passing. (4) Decoder (*c.f.,* Section 5.2.5), which utilizes an attention-based LSTM [15] model to generate a summary.

5.2.1 Problem Formulation

In this work, we focus on generating natural language summaries for the given functions [19, 76]. A simple example is illustrated in Figure 5.2, which is crawled from Linux Kernel. Our goal is to generate the best summary "set the time of day clock" based on the given source code. Formally, we define a dataset as $D = \{(c, s)|c \in C, s \in S\}$, where c is the source code of a function in the function set C and s represents its targeted summary in the summary set S. The task of code summarization is,

FIGURE 5.2: A simple C function.

```
Source Code:
int pdc_tod_set(unsigned long sec, unsigned long usec){
     int retval; unsigned long flags;
     spin_lock_irqsave(&pdc_lock, flags);
     retval = mem_pdc_call(PDC_TOD, PDC_TOD_WRITE, sec,
     ↪ usec);
     spin_unlock_irqrestore(&pdc_lock, flags);
     return retval;
}
Ground-Truth: set the time of day clock
```

given a source code c, to generate the best summary consisting of a sequence of tokens $\hat{s} = (t_1, t_2, ..., t_T)$ that maximizes the conditional likelihood $\hat{s} = \text{argmax}_s P(s|c)$.

5.2.2 Retrieval-augmented Static Graph

5.2.2.1 Code Property Graph

The source code of a function can be represented as Code Property Graph (CPG) [153], which is built on the abstract syntax tree (AST), combines the different types of edges (i.e., Flow To, Control, Define/Use, Reach) to represent the semantics of the program. Specifically, we describe each representation combining with Figure 5.3 as follows:

- **Abstract Syntax Tree (AST).** AST contains syntactic information for a program and omits irrelevant details that have no effect on the semantics. Figure 5.3 shows the completed AST nodes on the left simple program and each node has a code sequence in the first line and the type attribute in the second line. The black arrows represent the child-parent relations among ASTs.

- **Control Flow Graph (CFG).** Compared with AST highlighting the syntactic structure, CFG displays statement execution order, i.e., the possible order in which statements may be executed and the conditions that must be met for this to happen. Each statement in the program is treated as an independent node as well as a designated entry and exit node. Based on the keywords *if*, *for*, *goto*, *break* and *continue*, control flow graphs can be easily built and "Flow to" with green dashed arrows in Figure 5.3 represents this flow order.

- **Program Dependency Graph (PDG).** PDG includes **data dependencies** and **control dependencies**: (1) data dependencies are described as the definition of a variable in a statement that reaches the usage of the same variable in another statement. In Figure 5.3, the variable "*b*" is defined in the statement "*int b = a++*" and used in "*call (b)*". Hence, there is a "Reach" edge with blue arrows pointing from "*int b = a++*" to "*call (b)*". Furthermore, Define/Use edge with orange double arrows denotes the definition and usage of the variable. (2) different from CFG displaying the execution process of the complete program, control dependencies define the execution of a statement may be dependent on the value of a predicate, which is more focused on the statement itself. For instance, the statements "*int b = a++*" and "*call(b)*" are only performed "if a is even". Therefore, a red double arrow "Control" points from "*if (a % 2) == 0*" to "*int b = a++*" and "*call(b)*".

5.2.2.2 Graph Initialization

Formally, one raw function c could be represented by a multi-edged graph $g(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of AST nodes, $(v, u) \in \mathcal{E}$ denotes the edge between the node v and the node u. A node v consists of two parts: the *node sequence* and the *node type*. An illustrative example is shown in Figure 5.3. For example, in the red node, $a\%2 == 0$ is the node sequence and $Condition$ is the node type. An edge (v, u) has a type, named *edge type*, e.g, AST type and Flow To type.

Initialization Representation. Given a CPG, we utilize a BiLSTM to encode its nodes. We represent each token of the node sequence and each edge type using the learned embedding matrix $\boldsymbol{E}^{seqtoken}$ and $\boldsymbol{E}^{edgetype}$, respectively. Then nodes and edges of the CPG can be encoded as:

$$\begin{aligned} \boldsymbol{h}_1, ..., \boldsymbol{h}_l &= \text{BiLSTM}(\boldsymbol{E}^{seqtoken}_{v,1}, ..., \boldsymbol{E}^{seqtoken}_{v,l}) \\ encode\_node(v) &= [\boldsymbol{h}_l^{\rightarrow}; \boldsymbol{h}_1^{\leftarrow}] \\ encode\_edge(v, u) &= \boldsymbol{E}^{edgetype}_{v,u} \;\; if \;\; (v, u) \in \mathcal{E} \;\; else \;\; \mathbf{0} \end{aligned} \tag{5.1}$$

where l is the number of tokens in the node sequence of v. For the sake of simplicity, in the following section, we use $\boldsymbol{h}_v$ and $\boldsymbol{e}_{v,u}$ to represent the embedding of the node v

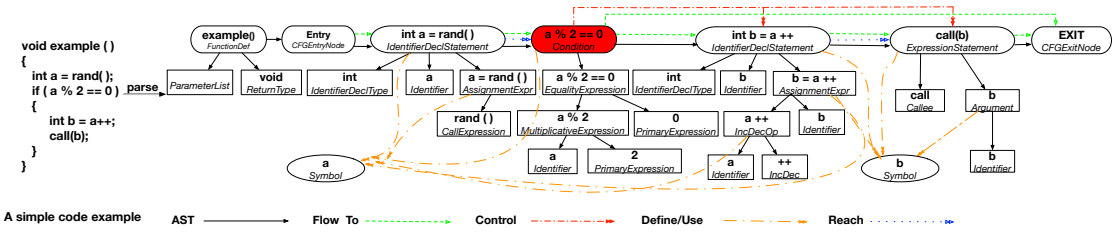


FIGURE 5.3: An example of Code Property Graph (CPG).

and the edge (v, u), respectively, i.e., $encode\_node(v)$ and $encode\_edge(v, u)$. Given the source code c of a function as well as the CPG $g(\mathcal{V}, \mathcal{E})$, $\boldsymbol{H}_c \in \mathbb{R}^{m \times d}$ denotes the initial node matrix of the CPG, where m is the total number of nodes in the CPG and d is the dimension of the node embedding.

5.2.2.3 Retrieval-based Augmentation

While retrieval-based methods can perform reasonably well on examples that are similar to those examples from a retrieval database, they typically have low generalization performance and might perform poorly on dissimilar examples. On the contrary, generation-based methods usually have better generalization performance, but cannot take advantage of similar examples from the retrieval database. In this work, we propose to combine the benefits of the two worlds and design a retrieval-augmented generation framework for the task of code summarization.

In principle, the goal of code summarization is to learn a mapping from source code c to the natural language summary $s = f(c)$. In other words, for any source code c', a code summarization system can produce its summary $s' = f(c')$. Inspired by this observation, conceptually, we can derive the following formulation $s = f(c) - f(c') + s'$. This tells us that we can actually compute the semantic difference between c and c', and further obtain the desired summary s for c by considering both the above semantic difference and s' which is the summary for c'. Mathematically, our goal becomes to learn a function which takes as input (c, c', s'), and outputs the summary s for c, that is, $s = g(c, c', s')$. This motivates us to design our Retrieval-based Augmentation mechanism, as detailed below.

Step 1: Retrieving. For each sample $(c, s) \in D$, we retrieve the most similar sample: $(c', s') = \text{argmax}_{(c',s') \in D'} sim(c, c')$, where $c \neq c'$, D' is a given retrieval database and $sim(c, c')$ is the text similarity. Following [19], we utilize Lucene for retrieval and calculate the similarity score z between the source code c and the retrieved code c' via dynamic programming [169], namely, $z = 1 - \frac{dis(c,c')}{max(|c|,|c'|)}$, where $dis(c, c')$ is the text edit distance.

Step 2: Retrieved Code-based Augmentation. Given the retrieved source code c' for the current sample c, we adopt a fusion strategy to inject retrieved semantics into the current sample. The fusion strategy is based on their initial graph representations ($\boldsymbol{H}_c$ and $\boldsymbol{H}_{c'}$) with an attention mechanism:

- To capture the relevance between c and c', we design an attention function, which computes the attention score matrix $\boldsymbol{A}^{aug}$ based on the embeddings of each pair of nodes in CPGs of c and c':

$$\boldsymbol{A}^{aug} \propto \exp(\text{ReLU}(\boldsymbol{H}_c \boldsymbol{W}^C)\text{ReLU}(\boldsymbol{H}_{c'} \boldsymbol{W}^Q)^T) \tag{5.2}$$

  where $\boldsymbol{W}^C, \boldsymbol{W}^Q \in \mathbb{R}^{d \times d}$ is the weight matrix with d-dim embedding size and ReLU is the rectified linear unit.

- We then multiply the attention matrix $\boldsymbol{A}^{aug}$ with the retrieved representation $\boldsymbol{H}_{c'}$ to inject the retrieved features into $\boldsymbol{H}_c$:

$$\boldsymbol{H}'_c = z\boldsymbol{A}^{aug}\boldsymbol{H}_{c'} \tag{5.3}$$

  where $z \in [0, 1]$ is the similarity score and computed from Step 1, which is introduced to weaken the negative impact of c' on the original training data c (i.e., when the similarity of c and c' is low).

- Finally, we merge $\boldsymbol{H}'_c$ and the original $\boldsymbol{H}_c$ to get the final representation of c.

$$\boldsymbol{comp} = \boldsymbol{H}_c + \boldsymbol{H}'_c \tag{5.4}$$

where $\boldsymbol{comp}$ is the augmented node representation additionally encoding the retrieved semantics.

Step 3: Retrieved Summary-based Augmentation. We further encode the retrieved summary s' with another BiLSTM model. We represent each token t'_i of s' using the learned embedding matrix $\boldsymbol{E}^{seqtoken}$. Then s' can be encoded as:

$$\boldsymbol{h}_{t'_1}, ..., \boldsymbol{h}_{t'_T} = \text{BiLSTM}(\boldsymbol{E}^{seqtoken}_{t'_1}, ..., \boldsymbol{E}^{seqtoken}_{t'_T}) \tag{5.5}$$

where $\boldsymbol{h}_{t'_i}$ is the hidden state of the BiLSTM model for the token t'_i in s' and T is the length of s'. We multiply $[\boldsymbol{h}_{t'_1}; ...; \boldsymbol{h}_{t'_T}]$ with the similarity score z, computed from Step 1, and concatenate it with the graph encoding results (i.e., the GNN encoder outputs) to obtain the input, namely, $[\text{GNN}_{\text{output}}; z\boldsymbol{h}_{t'_1}; ...; z\boldsymbol{h}_{t'_T}]$, to the decoder.

5.2.3 Attention-based Dynamic Graph

Due to that GNN-based encoders usually consider the k-hop neighborhood, the global relation among nodes in the static graph (see Section 5.2.2.2) may be ignored. In order to better capture the global semantics of source code, based on the static graph, we propose to dynamically construct a graph via a structure-aware global attention mechanism, which allows message passing among any pair of nodes. The attention-based dynamic graph can better capture the global dependency among nodes, and thus supplement the static graph.

Structure-aware Global Attention. The construction of the dynamic graph is motivated by the structure-aware self-attention mechanism proposed in [170]. Given the static graph, we compute a corresponding dense adjacency matrix $\boldsymbol{A}^{dyn}$ based on a structure-aware global attention mechanism and obtain the constructed graph, namely, *attention-based dynamic graph*.

$$\boldsymbol{A}^{dyn}_{v,u} = \frac{\text{ReLU}(\boldsymbol{h}_v^T \boldsymbol{W}^Q)(\text{ReLU}(\boldsymbol{h}_u^T \boldsymbol{W}^K) + \text{ReLU}(\boldsymbol{e}_{v,u}^T \boldsymbol{W}^R))^T}{\sqrt{d}} \tag{5.6}$$

where $\boldsymbol{h}_v, \boldsymbol{h}_u \in \boldsymbol{comp}$ are the augmented node embedding for any node pair (v, u) in the CPG. Note that global attention considers each pair of nodes of the CPG, regardless of whether there is an edge between them. $\boldsymbol{e}_{v,u} \in \mathbb{R}^{d_e}$ is the edge embedding and $\boldsymbol{W}^Q, \boldsymbol{W}^K \in \mathbb{R}^{d \times d}$, $\boldsymbol{W}^R \in \mathbb{R}^{d_e \times d}$ are parameter matrices, d_e and d are the dimensions of edge embedding and node embedding, respectively. The adjacency matrix $\boldsymbol{A}^{dyn}$ will be further row normalized to obtain $\tilde{\boldsymbol{A}}^{dyn}$, which will be used to compute dynamic message passing (see Section 5.2.4).

$$\tilde{\boldsymbol{A}}^{dyn} = \mathrm{softmax}(\boldsymbol{A}^{dyn}) \tag{5.7}$$

5.2.4 Hybrid GNN

To better incorporate the information of the static graph and the dynamic graph, we propose Hybrid Message Passing (HMP), which is performed on both retrieval-augmented static graph and attention-based dynamic graph.

Static Message Passing. For every node v at each computation hop k in the static graph, we apply an aggregation function to calculate the aggregated vector $\boldsymbol{h}_v^k$ by considering a set of neighboring node embeddings computed from the previous hop.

$$\boldsymbol{h}_v^k = \mathrm{SUM}(\{\boldsymbol{h}_u^{k-1} | \forall u \in \mathcal{N}_{(v)}\}) \tag{5.8}$$

where $\mathcal{N}_{(v)}$ is a set of the neighboring nodes which are directly connected with v. For each node v, $\boldsymbol{h}_v^0$ is the initial augmented node embedding of v, i.e., $\boldsymbol{h}_v \in \boldsymbol{comp}$.

Dynamic Message Passing. The node information and edge information are propagated on the attention-based dynamic graph with the adjacency matrices $\tilde{\boldsymbol{A}}^{dyn}$, defined as

$$\boldsymbol{h}_v'^k = \sum_u \tilde{\boldsymbol{A}}_{v,u}^{dyn} (\boldsymbol{W}^V \boldsymbol{h}_u'^{k-1} + \boldsymbol{W}^F \boldsymbol{e}_{v,u}) \tag{5.9}$$

where v and u are any pair of nodes, $\boldsymbol{W}^V \in \mathbb{R}^{d \times d}$, $\boldsymbol{W}^F \in \mathbb{R}^{d \times d_e}$ are learned matrices, and $\boldsymbol{e}_{v,u}$ is the embedding of the edge connecting v and u. Similarly, $\boldsymbol{h}_v'^0$ is the initial augmented node embedding of v in $\boldsymbol{comp}$.

Hybrid Message Passing. Given the static/dynamic aggregated vectors $\boldsymbol{h}_v^k/\boldsymbol{h}_v^{'k}$ for static and dynamic graphs, we fuse both vectors and feed the resulting vector to a Gated Recurrent Unit (GRU) to update node representations.

$$\boldsymbol{f}_v^k = \mathrm{GRU}(\boldsymbol{f}_v^{k-1}, \mathrm{Fuse}(\boldsymbol{h}_v^k, \boldsymbol{h}_v^{'k})) \tag{5.10}$$

where $\boldsymbol{f}_v^0$ is the augmented node initialization in $\boldsymbol{comp}$. The fusion function Fuse is designed as a gated sum of two inputs.

$$\mathrm{Fuse}(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{z} \odot \boldsymbol{a} + (1 - \boldsymbol{z}) \odot \boldsymbol{b} \quad \boldsymbol{z} = \sigma(\boldsymbol{W}_z[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{a} \odot \boldsymbol{b}; \boldsymbol{a} - \boldsymbol{b}] + \boldsymbol{b}_z) \tag{5.11}$$

where $\boldsymbol{W}_z$ and $\boldsymbol{b}_z$ are learnable weight matrix and vector, $\odot$ is the component-wise multiplication, σ is a sigmoid function and $\boldsymbol{z}$ is a gating vector. After K hops of GNN computation, we obtain the final node representation $\boldsymbol{f}_v^K$ and then apply max-pooling over all nodes $\{\boldsymbol{f}_v^K | \forall v \in \mathcal{V}\}$ to get the graph representation.

5.2.5 Decoder

The decoder is similar to other state-of-the-art Seq2seq models [5, 130] where an attention-based LSTM decoder is used. The decoder takes the input of the concatenation of the node representation and the representation of the retrieved summary s', namely, $[\boldsymbol{f}_{v_1}^K; ...; \boldsymbol{f}_{v_m}^K; z\boldsymbol{h}_{t'_1}; ...; z\boldsymbol{h}_{t'_T}]$, where m is the number of nodes in the input CPG graph. The initial hidden state of the decoder is the fusion (Eq. 6.7) of the graph representation and the weighted (i.e., multiply similarity score z) final state of the retrieved summary BiLSTM encoder.

We train the model with the cross-entropy loss, defined as $\mathcal{L} = \sum_t -\log P(s_t^* | c, s_{<t}^*)$, where s_t^* is the word at the t-th position of the ground-truth output and c is the source code of the function. To alleviate the exposure bias, we utilize schedule teacher forcing [40]. During the inference, we use beam search to generate final results.

5.3 Experimental Setup

In this section, we first introduce our collected dataset, then present the selected baselines for comparison, followed by the evaluation metrics and the configuration of our model.

5.3.1 Dataset

It is non-trivial to obtain high-quality datasets for code summarization. We noticed that despite some previous works [14, 171] released their datasets, however, they are all based on high-level programming languages i.e. Java, Python. Furthermore, they have been confirmed to have extensive duplication, making the model overfit to the training data that overlapped with the testset [164, 172]. We are the first to explore summarization on *C* programming language. Specifically, we crawled from popular *C* repositories (e.g., Linux and QEMU) on GitHub, and then extracted separate function-summary pairs from these projects. Specifically, we extracted functions and associated comments marked by special characters "/**" and "*/" over the function declaration. These comments can be considered as explanations of the functions. We filtered out functions with lines exceeding 1000 and any other comments inside the function, and the first sentence was selected as the summary. A similar practice can be found in [34]. Totally, we collected **500k+** raw function-summary pairs. Furthermore, functions with token size greater than 150 were removed for computational efficiency and there were **130k+** functions left. Since duplication is very common in existing datasets [164], followed by [172], we performed a de-duplication process and removed functions with similarity over 80%. Specifically, we calculated the cosine similarity by encoding the raw functions into vectors with sklearn. Finally, we kept **95k+** unique functions. We name this dataset *C* Code Summarization Dataset (CCSD). To further test the model generalization ability, we construct in-domain functions and out-of-domain functions by dividing the projects into two sets, denoted as a and b. For each project in a, we randomly select some of the functions in this project as the training data and the unselected functions are the in-domain validation/test data. All functions in projects b are regarded as out-of-domain test data. Finally, we obtain 84,316 training functions,

4,432 in-domain validation functions, 4,203 in-domain test functions and 2,330 out-of-domain test functions. For the retrieval augmentation, we use the training set as the retrieval database, i.e., $D' = D$ (see Step 1 in Section 5.2.2.3). The open-source code analysis platform Joern [153] was applied to construct the code property graph.

5.3.2 Comparison Baselines

We evaluate our proposed framework against a number of state-of-the-art methods. Specifically, we classify the selected baseline methods into three groups: (1) Retrieval-based approaches: TF-IDF [67] and NNGen [173], (2) Sequence-based approaches: CODE-NN [13, 14], Transformer [26], Hybrid-DRL [76], Rencos [19] and Dual model [79], (3) Graph-based approaches: SeqGNN [164]. In addition, we implemented two other graph-based baselines: GCN2Seq and GAT2Seq, which respectively adopt the Graph Convolution [21] and Graph Attention [22] as the encoder and a LSTM as the decoder for generating summaries. Note that Rencos [19] combines the retrieval information into Seq2Seq model, and we classify it into Sequence-based approaches. For papers that provide the source code, we directly reproduce their methods on CCSD dataset. Otherwise, we reimplement their approaches with reference to the papers. We introduce each baseline approach as follows:

5.3.2.1 Retrieval-based Approaches

TF-IDF [67] is the abbreviation of Term Frequency-Inverse Document Frequency, which is adopted in the early code summarization [67]. It transforms programs into weight vectors by calculating term frequency and inverse document frequency. We retrieve the summary of the most similar programs by calculating the cosine similarity on the weighted vectors.

NNGen [173] is a retrieved-based approach to produce commit messages for code changes. We reproduce such an algorithm on code summarization. Specifically, we retrieve the most similar top-k code snippets on a bag-of-words model and prioritize the summary in terms of BLEU-4 scores in top-k code snippets.

5.3.2.2 Sequence-based Approaches

CODE-NN [13, 14] adopts an attention-based Seq2Seq model to generate summaries on the source code.

Transformer [26] adopts the transformer architecture [16] with self-attention to capture long dependency in the code for source code summarization.

Hybrid-DRL [76] is a reinforcement learning-based approach, which incorporates AST and sequential code snippets into a deep reinforcement learning framework and employs evaluation metrics (e.g., BLEU) as the reward.

Dual Model [79] propose a dual training framework by training code summarization and code generation tasks simultaneously to boost each task performance.

Rencos [19] is the retrieval-based Seq2Seq model for code summarization. it utilized a pretrained Seq2Seq model during the testing phase by computing a joint probability conditioned on both the original source code and retrieved the most similar source code for the summary generation. Compared with Rencos, we propose a novel retrieval-augmented mechanism for a similar source code and use it during the model training phase.

5.3.2.3 Graph-based Approaches

We also compared with some latest GNN-based works, employing a graph neural network for source code summarization.

GCN2Seq, GAT2Seq modify Graph Convolution Network [21] and Graph Attention Network [22] to perform convolution operation and attention operation on the code property graph for learning and followed by a LSTM to generate summaries. We implement the related code from scratch.

SeqGNN [164] combines GGNNs and standard sequence encoders for summarization. They take the code and relationships between elements of the code as input. Specially,

a BiLSTM is employed on the code sequence to learn representations and each source code token is modelled as a node in the graph and employed GGNN for graph-level learning. Since our node sequences are sub-sequence of source code rather than individual tokens, we adjust to slice the output of BiLSTM and sum each token representation in node sequences as node initial representation for summarization. Furthermore, we implement the related code from scratch.

5.3.3 Evaluation Metrics

Similar to previous works [13, 19, 76, 164], BLEU [137], METEOR [139] and ROUGE-L [138] are used as our automatic evaluation metrics. These metrics are popular for evaluating machine translation and text summarization tasks. Except for these automatic metrics, we also conduct a human evaluation study. We invite 5 PhD students and 10 master students as volunteers, who have rich C programming experiences. The volunteers are asked to rank summaries generated from the anonymized approaches from 1 to 5 (i.e., 1: Poor, 2: Marginal, 3: Acceptable, 4: Good, 5: Excellent) based on the relevance of the generated summary to the source code and the degree of similarity between the generated summary and the actual summary. Specifically, we randomly choose 50 functions for each model with the corresponding generated summaries and ground-truths. We calculate the average score and the higher the score, the better the quality.

5.3.4 Model Settings

We embed the most frequent 40,000 words in the training set with 512-dims and set the hidden size of BiLSTM to 256 and the concatenated state size for both directions is 512. The dropout is set to 0.3 after the word embedding layer and BiLSTM. We set GNN hops to 1 for the best performance. The optimizer is selected with Adam with an initial learning rate of 0.001. The batch size is set to 64 and the early stop for 10. The beam search width is set to 5 as usual. All experiments are conducted on the DGX server with four Nvidia Graphics Tesla V100 and each epoch takes 6 minutes averagely. All hyperparameters are tuned with grid search on the validation set.

TABLE 5.1: Automatic evaluation results (in %) on the CCSD test set.

Methods	In-domain			Out-of-domain			Overall		
	BLEU-4	ROUGE-L	METEOR	BLEU-4	ROUGE-L	METEOR	BLEU-4	ROUGE-L	METEOR
TF-IDF	15.20	27.98	13.74	5.50	15.37	6.84	12.19	23.49	11.43
NNGen	15.97	28.14	13.82	5.74	16.33	7.18	12.76	23.93	11.58
CODE-NN	10.08	26.17	11.33	3.86	15.25	6.19	8.24	22.28	9.61
Hybrid-DRL	9.29	30.00	12.47	6.30	24.19	10.30	8.42	28.64	11.73
Transformer	12.91	28.04	13.83	5.75	18.62	9.89	10.69	24.65	12.02
Dual Model	11.49	29.20	13.24	5.25	21.31	9.14	9.61	26.40	11.87
Rencos	14.80	31.41	14.64	7.54	23.12	10.35	12.59	28.45	13.21
GCN2Seq	9.79	26.59	11.65	4.06	18.96	7.76	7.91	23.67	10.23
GAT2Seq	10.52	26.17	11.88	3.80	16.94	6.73	8.29	22.63	10.00
SeqGNN	10.51	29.84	13.14	4.94	20.80	9.50	8.87	26.34	11.93
HGNN w/o augment & static	11.75	29.59	13.86	5.57	22.14	9.41	9.98	26.94	12.05
HGNN w/o augment & dynamic	11.85	29.51	13.54	5.45	21.89	9.59	9.93	26.80	12.21
HGNN w/o augment	12.33	29.99	13.78	5.45	22.07	9.46	10.26	27.17	12.32
HGNN w/o static	15.93	33.67	15.67	7.72	24.69	10.63	13.44	30.47	13.98
HGNN w/o dynamic	15.77	33.84	15.67	7.64	24.72	10.73	13.31	30.59	14.01
HGNN	**16.72**	**34.29**	**16.25**	**7.85**	**24.74**	**11.05**	**14.01**	**30.89**	**14.50**

5.4 Experimental Results

In this section, we present the experimental results, specifically, we first show the automatic evaluation results compared with the baseline approaches, then we ablate the effectiveness of each component of our model and present the results of human evaluation and followed by a case study to show some examples. We further compared our approach against some baselines on the existing dataset to confirm the scalability of our approach.

5.4.1 Comparison with the Baselines

Table 5.1 shows the evaluation results including two parts: the comparison with baselines and the ablation study. Considering the comparison with state-of-the-art baselines, in general, we find that our proposed model outperforms existing methods by a significant margin on both in-domain and out-of-domain datasets, and shows good generalization performance. Compared with others, on the in-domain dataset, the retrieval-based approaches could achieve competitive performance on BLEU-4, however, ROUGE-L and METEOR are far less than ours. Moreover, they do not perform well on the out-of-domain dataset. Compared with the graph-based approaches (i.e., GCN2Seq, GAT2Seq and SeqGNN), even without augmentation (*HGNN w/o augment*), our approach still

outperforms them, which further demonstrates the effectiveness of Hybrid GNN for additionally capturing global graph information. Compared with Rencos which also considers the retrieved information in the Seq2Seq model, its performance is still lower than *HGNN*. On the overall dataset including both in-domain and out-of-domain data, our model achieves **14.01**, **30.89** and **14.50**, outperforming current state-of-the-art method Rencos by **1.42**, **2.44** and **1.29** in terms of BLEU-4, ROUGE-L and METEOR metrics.

5.4.2 Ablation Study

We also conduct an ablation study to evaluate the impact of different components of our framework, e.g, retrieval-based augmentation, static graph and dynamic graph in the last row of Table 5.1. Overall, we found that (1) retrieval-augmented mechanism could contribute to the overall model performance (*HGNN* vs. *HGNN w/o augment*). Compared with *HGNN*, we see that the performance of *HGNN w/o static* and *HGNN w/o dynamic* decreases, which demonstrates the effectiveness of the Hybrid GNN and (2) the performance without a static graph is worse than the performance without dynamic graph in ROUGE-L and METEOR, however, BLEU-4 is higher than the performance without dynamic graph. To further understand the impact of the static graph and dynamic graph, we evaluate the performance without augmentation and static graph/-dynamic graph (see *HGNN w/o augment& static* and *HGNN w/o augment& dynamic*). Compared with *HGNN w/o augment*, the results further confirm the effectiveness of the Hybrid GNN (i.e., static graph and dynamic graph).

We also conduct experiments to investigate the impact of code-based augmentation and summary-based augmentation. Overall, we found that summary-based augmentation could contribute more than code-based augmentation. For example, after adding the code-based augmentation, the performance can be 10.22, 27.54 and 12.49 in terms of BLUE-4, ROUGE-L and METEOR on the overall dataset. With the summary-based augmentation, the results can reach to 13.76, 30.59 and 14.11. Compared with the results without augmentation (i.e., 10.26. 27.17, 12.32 with *HGNN w/o augment*), we can see that code-based augmentation could have some improvement, but the effect is not significant compared with summary-based augmentation. We conjecture that due to that the code and summary are heterogeneous, the summary-based augmentation has a

TABLE 5.2: Human evaluation results on the CCSD test set.

Metrics	NNGen	Transformer	Rencos	SeqGNN	***HGNN***
Relevance	3.23	3.17	3.48	3.09	**3.69**
Similarity	3.18	3.02	3.32	3.06	**3.51**

more direct impact on the code summarization task. When combining both code-based augmentation and summary-based augmentation, we can achieve the best results (i.e., 14.01, 30.89, 14.50). We plan to explore more code-based augmentation (e.g, semantic-equivalent code transformation) in our future work.

5.4.3 Human Evaluation

As shown in Table 5.2, we perform a human evaluation on the overall dataset to assess the quality of the generated summaries by our approach, NNGen, Transformer, Rencos and SeqGNN in terms of relevance and similarity. As depicted in Table 5.1, NNGen, Rencos and SeqGNN are the best retrieval-based, sequence-based, and graph-based approaches, respectively. We also compare with Transformer as it has been widely used in natural language processing. The results show that our method can generate better summaries, which are more relevant to the source code and more similar to the ground-truth summaries.

5.4.4 Case Study

To perform qualitative analysis, we present two examples with generated summaries by different methods from the overall data set, shown in Figure 5.4 and Figure 5.5. We can see that, in the first example, our approach can learn more code semantics, i.e., p is a self-defined struct variable. Thus, we could generate a token *object* for the variable p. However, other models can only produce *string*. Example 2 is a more difficult function with the functionality to "release reference of cedar", as compared to other baselines, our approach effectively captures the functionality and generates a more precise summary.

FIGURE 5.4: The first example generated by different approaches on the CCSD test set.

```
Source Code:
static void strInit(Str *p){
    p->z = 0;
    p->nAlloc = 0;
    p->nUsed = 0;
}
Ground-Truth: initialize a str object
NNGen: free the string
Transformer: reset a string
Rencos: append a raw string to the json string
SeqGNN: initialize the string
HGNN: initialize a string object
```

FIGURE 5.5: The second example generated by different approaches on the CCSD test set.

```
Source Code:
void ReleaseCedar(CEDAR *c){
    if (c == NULL)
        return;
    if (Release(c->ref) == 0)
        CleanupCedar(c);
}
Ground-Truth: release reference of the cedar
NNGen: release the virtual host
Transformer: release of the cancel object
Rencos: release of the cancel object
SeqGNN: release cedar communication mode
HGNN: release reference of cedar
```

5.4.5 Extension on the Python Dataset

We conducted additional experiments on a public dataset, i.e., the Python Code Summarization Dataset (PCSD), which was also used in Rencos (the most competitive baseline). We follow the setting of Rencos and split PCSD into the training set, validation set and testing set with fractions of 60%, 20% and 20%. We construct the static graph based on AST. The decoding step is set to 50, followed by Rencos, and the other settings are the same with CCSD. We compare our methods on PCSD against various competitive baselines, i.e., NNGen, CODE-NN, Rencos and Transformer, which are either retrieval-based, generation-based or hybrid methods. The results are shown in Table 5.3. The results indicate that, compared with the best results from NNGen, CODE-NN, Rencos and Transformer, our method can improve the performance by 0.40, 3.70 and 1.41 in terms of BLEU-4, ROUGE-L and METEOR. We also conduct the ablation study on

TABLE 5.3: Automatic evaluation results (in %) on the PCSD test set.

Methods	BLEU-4	ROUGE-L	METEOR
NNGen	21.60	31.61	15.96
CODE-NN	16.39	28.99	13.68
Transformer	17.06	31.16	14.37
Rencos	24.02	36.21	18.07
HGNN w/o static	24.06	38.28	18.66
HGNN w/o dynamic	24.13	38.64	18.93
HGNN	**24.42**	**39.91**	**19.48**

PCSD to demonstrate the usefulness of the static graph (i.e., HGNN w/o dynamic) and dynamic graph (i.e., HGNN w/o static). The results also demonstrate that both static graph and dynamic graph can contribute to our framework. In summary, the results on both our released benchmark (CCSD) and existing benchmark (PCSD) demonstrate the effectiveness and scalability of our method.

5.5 Threats to Validity

One of the threats lines in the implementation of our approach. Our approach consists of complex matrix operations. To reduce this threat, the co-authors carefully check for the correctness of the implementation. Another threat is the reported values of the baselines. We directly adopt the hyper-parameters from the original papers to evaluate the performance, however, these settings may be not optimal for our benchmark.

5.6 Conclusion

In this chapter, we proposed a general framework for automatic code summarization. A novel retrieval-augmented mechanism is proposed for combining the benefits of both retrieval-based and generation-based approaches. Moreover, to capture global semantics among nodes, we design a hybrid message passing GNN based on both static and dynamic graphs. The extensive experiments demonstrate that our approach improves state-of-the-art techniques substantially. In the next chapter, we introduce a new graph-based technique for deep code search.

Chapter 6

GraphSearchNet: Graph-based Semantic Code Search

In Chapter 4 and Chapter 5, we have explored the way to encode program structures with different graph neural network variants to learn program semantics for software vulnerability identification and source code summarization. In this chapter, we focus on another widely concerned software engineering task (i.e., code search) and explore the way to mitigate the semantic gap between the program and natural language query for code search.

6.1 Introduction

With the fast development of the software industry over the past few years, the global source code over public and private repositories (*e.g.*, on GitHub or Bitbucket) is reaching an unprecedented amount. It is already commonly recognized that the software industry is entering the "*Big Code*" era. Code search, which aims to search the relevant code snippets based on the natural language query from a large code corpus (*e.g.*, Github, Stack Overflow, or private ones), has become a critical problem in the "*Big Code*" era. In addition, some studies [88, 174] also have shown that more than 90% of the efforts from the software developers aim at reusing the existing code. Hence,

an accurate code search system can greatly improve software productivity and quality, while reducing the software development cost.

Automated code search is far from settled. Some early attempts were started by leveraging information retrieval (IR) techniques to capture the relationship of the code and the query by keyword matching [85, 87–89]. However, such techniques are ad-hoc, making them limited especially when no common keywords exist in the source code and queries. Furthermore, the extracted keywords from the query tend to be short, which cannot represent the rich semantics behind the text. To address these limitations, many works expand the query format and reformulate it with different expressions [85,86,91,175,176]. For example, Lu et al. [85] expanded the query with some synonyms generated from WordNet [177] to improve the hit ratio. CodeMatcher [90] proposed an IR-based model by collecting metadata for query words to identify irrelevant/noisy ones and iteratively performing the fuzzy search with the important query words on the codebase to return the program candidates. These IR-based techniques essentially perform the matching process based on keywords. However since code or natural language query has different types of semantic variants, an effective code search system requires a high-level semantic mapping between code and natural language queries.

To address this limitation, more recent attempts shifted to deep learning (DL)-based techniques [92, 93, 95–100, 103], which encode the source code and the query into vectors (*i.e.*, learning the representations behind the program and the query). Then, the similarity between two vectors, such as cosine similarity, is computed to measure the semantic relevance between the code snippet and the query. Code snippets with a higher similarity score of the query are returned as the search results. However, most of these works only rely on sequential models *i.e.*, Long-Short Term Memory (LSTMs) [15], Self-Attention [16] to learn the vector representations for both code and query. These sequential models are struggling to learn the semantic relations because they ignore the structural information hidden in the text to learn. In addition, MMAN [103] encoded the program sequence, abstract syntax tree (AST) and control flow graph (CFG) with LSTM [15], Tree-LSTM [10] and GGNN [20] respectively on the C programming language. Then it further encoded the query sequence with another LSTM to learn

the mapping relationship between the code and query, however, MMAN ignored the structural information behind the query and the limitations of GGNN *i.e.*, the missed global dependencies in a graph [178, 179] in the process of GGNN learning, is not well-addressed. To sum up, although some recent progress has been made for automated code search, the key challenges still exist: (1) source code and the natural language queries are heterogeneous [92], they have completely different grammatical rules and language structure, which leads to the semantic mapping is hard; (2) the rich structural information behind in the code and the query fails to explore. Failing to utilize the rich structural information beyond the simple text may limit the effectiveness of these approaches for code search; (3) Although there are some existing works [103] attempted to use GNNs for code search, the limitations of GNNs are not well-addressed.

To the end, in this work, we propose a novel neural network framework, namely GraphSearchNet, towards learning the representations that fully use the structural information to learn the semantic relations of program and queries for automated code search. In particular, since the large corpus of the query dataset is hard to collect, we follow the existing works [92, 93, 180, 181] and use the summary of a code snippet for the replacement to jointly train the program encoder and the summary encoder. Specifically, we convert the program to a graph with syntactic edges (*AST Edge*, *NextToken SubToken*) and data-flow edges (*ComputedFrom*, *LastUse* and *LastWrite*) to represent the program semantics. Furthermore, we also build the summary graph based on dependency parsing [182]. For each encoder, we feed the constructed graph to Bidirectional Gated Graph Neural Network *i.e.*, BiGGNN [23] to capture the structural information in a graph. We further enhance BiGGNN by the multi-head attention module to supplement the missed global dependencies that BiGGNN fails to learn to improve the model learning capacity. Once GraphSearchNet is trained, at the query phase, given a natural language query, the summary encoder is utilized to obtain the query vector, then the top-k program candidates are returned based on the cosine similarity between the query vector and the program vectors that are embedded from the program encoder on the large search code database.

To demonstrate the effectiveness of our approach, we investigate the performance of

GraphSearchNet against **13** state-of-the-art baselines on Java and Python datasets from the open-sourced CodeSearchNet [93], which has over **2 million** functions and evaluate these approaches in terms of **5** evaluation metrics such as R@k, MRR, NDCG. We further conduct a quantitative analysis on **99** real queries to confirm the effectiveness of GraphSearchNet. The extensive experimental results show that GraphSearchNet significantly outperforms the baselines on the evaluation metrics. Furthermore, GraphSearchNet can also produce high-quality programs based on the real natural language query from the quantitative analysis. Our code is available at `https://github.com/shangqing-liu/GraphSearchNet`. Overall, we highlight our contributions as follows:

- We propose a novel graph-based framework to capture the structural and semantic information for accurately learning the semantic mapping of the program and the query for code search.
- We design GraphSearchNet to improve model learning capacity by BiGGNN to capture the local structural information in a graph and multi-head attention to capture the global dependencies that BiGGNN fails to learn.
- We conduct an extensive evaluation to demonstrate the effectiveness of GraphSearchNet on the large code corpora and the experimental results demonstrate that GraphSearchNet outperforms the state-of-the-art baselines by a significant margin. we have made our code public to benefit academia and the industry.

6.2 Background and Motivation

In this section, we first detail the motivation for the design of GraphSearchNet and followed by introducing the existing datasets for code search and the multi-head attention that we will use in GraphSearchNet.

6.2.1 Motivation

Program semantics learning by deep learning techniques can be considered as the fundamental problem for a variety of code-related tasks. Compared with considering the

program as a flat sequence of tokens with the sequential model such as LSTMs [15] or Self-Attention [16] to learn program semantics, Allamanis et al [120] lighted up this field by converting the program into a graph with different relations of the nodes to represent the program semantics and further utilized GNNs to capture the node relations. After that, many graph-based works for different tasks [9, 103, 164, 178, 183, 184] have emerged in both AI and SE community. These works have proved the effectiveness of GNNs [20, 21, 120] in modelling a program to capture program semantics. Furthermore, due to the powerful relation learning capacity of GNNs, they also have been widely used in many NLP tasks, *e.g.*, natural question generation (QG) [23, 185], conversational machine comprehension (MC) [186–188]. Annervaz et al. [189] further confirmed that augmenting graph information with LSTM can improve the performance of many NLP tasks. Hence, inspired by these works, in this paper, we propose our approach to convert the program and the query into graphs with GNNs to learn the semantic relations for code search.

However, recent advanced works on GNNs [178, 179, 184] have proved that GNNs are powerful to capture the signals from the short-range nodes, while the long-range information from the distant nodes cannot well-handled. It is caused by the message-passing computation process. Specifically, the total number of hops K limits the network can only sense the range of the interaction between nodes at a radius of K *i.e.*, the receptive field at K. We give a simple example as shown in Figure 6.1 for a better explanation. In particular, when $K = 1$, node 1 can only know the information from its neighbor *i.e.*, node 2. Furthermore, when $K = 2$, node 1 knows the information from its neighbor *i.e.*, node 2 and node 2's neighbor *i.e.*, node 3. When we increase K to 3, node 1 can sense the information from node 2, node 3 and node 4. Hence, to let node 1 know the information about its far-distant node, we must increase K to a large value. However, the hyper-parameter K can not be set to a large value due to the over-squashing [179] and over-smoothing [190, 191] problem in GNNs. Hence, empirically, K is often set to a small value [21, 178] to avoid this problem. It leads GNNs to be powerful in capturing the local structural information while failing to capture the global dependencies in a graph. We further provide a real code snippet to illustrate the global dependencies that GNNs fail to capture. As shown in Figure 6.2a, this function is from open-source

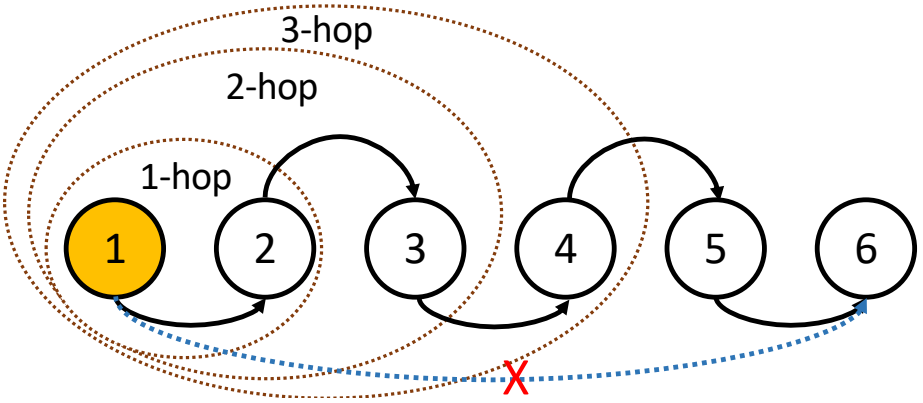


FIGURE 6.1: Message passing of GNNs within K-th neighborhood information where the dash area is the receptive field that node 1 knows.

GitHub project[1]. Since the completed constructed graph for this function is complex, we just extract a subgraph that contains the LastUse edge for the variable "n" for illustration, which is shown in Figure 6.2b. We can observe that in this directed graph, the nodes n_2, n_3, n_4, n_5 form a loop. In addition, there is a directed edge pointed from n_2 to n_1. Hence, for a specific node n_4, during the message passing in GNNs, it only requires $K = 4$ (i.e., receptive field) to sense the information from n_1, n_2, n_3 and n_5. We also find that n_4 cannot communicate with n_6 because they are unreachable (i.e., there is no directed path pointed from n_4 to n_6). From Figure 6.2a, we find that n_4 and n_6 are closely related in the program semantics, however, node n_4 cannot learn the effective features from node n_6 by the message passing in the graph. Hence, it is necessary to capture the global dependencies between n_4 and n_6 to mitigate the limitation of traditional GNNs. It inspires us to design a new neural network for code search.

6.2.2 Existing Datasets for Code Search

There are some existing datasets designed for code search, such as CodeSearchNet [93], DeepCS [101], FB-Java [180], CosBench [181]. The released dataset by DeepCS is processed by the authors and the raw data cannot be obtained for the graph construction. The other datasets such as FB-Java and CosBench are both collected from Java projects. In contrast, CodeSearchNet contains data from six programming languages, besides Java, it also consists of data from other programming languages such as Python.

[1] *https://github.com/glue-viz/glue-vispy-viewers*

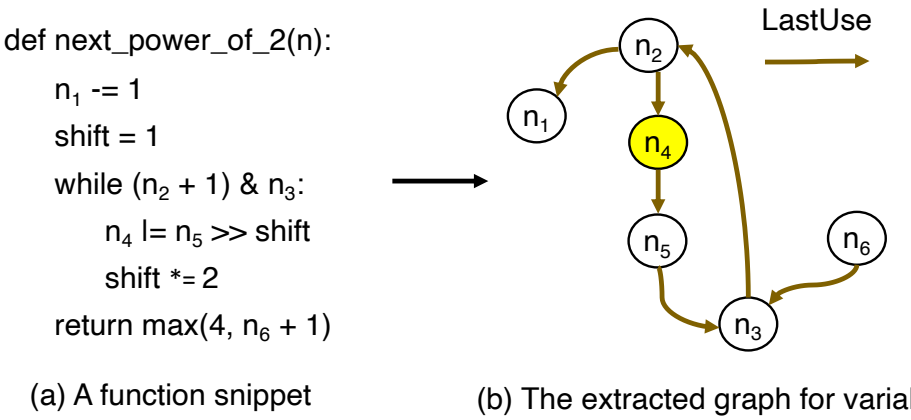


FIGURE 6.2: A real function example, please note that we distinguish the variable n into n, n_1, n_2, n_3, n_4, n_5 and n_6 for ease of presentation. Furthermore, since the constructed graph for this function is complex, we extract a subgraph that only contains the LastUse edge for the variable "n" for better presentation and explanation.

Since different programming language has specific grammatical features, GraphSearchNet aims to prove the proposed approach is robust against different languages, hence we select Python and Java data from CodeSearchNet for the evaluation. Furthermore, in deep code search, a sufficient amount of (program, query) pairs is hard to collect and CodeSearchNet utilizes (program, summary) pairs for the replacement at the learning phase, where the summary describes the functionality of a function. CodeSearchNet further provides a number of 99 real natural language queries *e.g.*, "convert decimal to hex" to retrieve the related programs from the search codebase, which is independent of the dataset that is used for training the model. We also follow these settings for evaluation.

6.2.3 Multi-Head Attention

Self-attention is the key idea in the transformer [16], which is widely used in NLP. An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values and output are all vectors. The particular Scaled Dot-Product Attentionr [16] can be expressed as follows:

$$\text{Attention}(\text{Q}, \text{K}, \text{V}) = \text{Softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}})\boldsymbol{V} \tag{6.1}$$

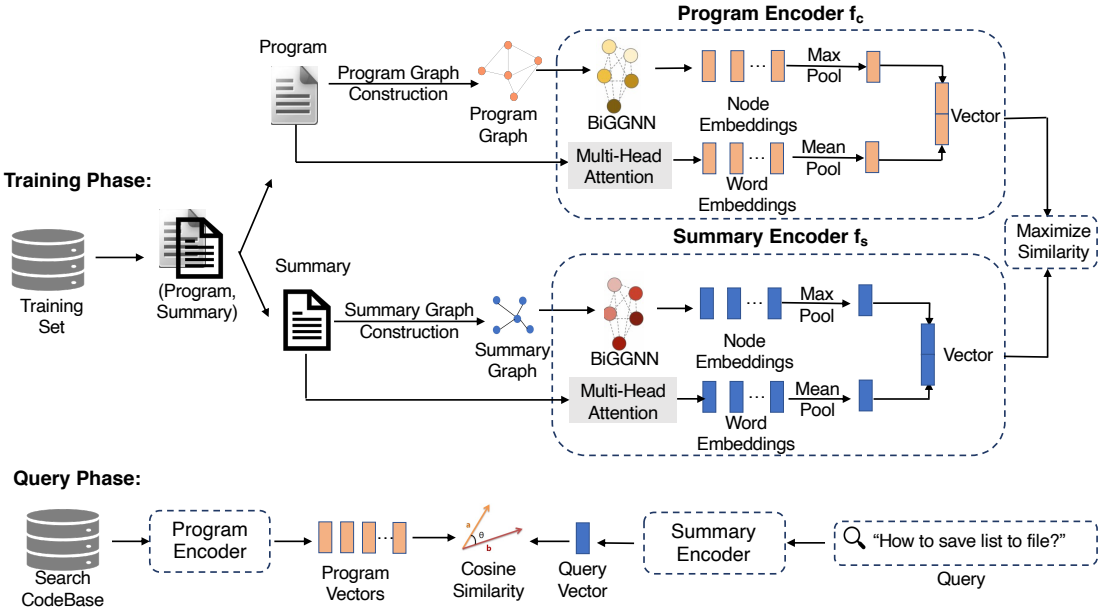


FIGURE 6.3: The framework of GraphSearchNet.

where d_k is the dimensional length, $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ represent the query, key, and value matrices. To further improve the expression capacity of the self-attention, it is beneficial to linearly project the queries, keys and values h times with different linear projections and then concatenate and project to obtain the final outputs and this calculation process is named multi-head attention [16]:

$$\begin{aligned} \text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= \text{Concat}(\text{head}_1, ..., \text{head}_h)\boldsymbol{W}^O \\ \text{where head}_i &= \text{Attention}(\text{QW}_\text{i}^\text{Q}, \text{KW}_\text{i}^\text{K}, \text{VW}_\text{i}^\text{V}) \end{aligned} \tag{6.2}$$

where $\boldsymbol{W}_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $\boldsymbol{W}_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $\boldsymbol{W}_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $\boldsymbol{W}^O \in \mathbb{R}^{hd_v \times d_{model}}$ are model parameters and h is the number of heads.

6.3 GraphSearchNet

In this section, we introduce our framework GraphSearchNet, as shown in Figure 6.3, which includes three sequential parts: 1) *Graph Construction*, which constructs directed graphs for programs and summaries with comprehensive semantics. 2) *Training Phase*, which jointly learns two separate encoders i.e., the program encoder f_c and the summary encoder f_s to obtain the vector representations respectively. Each encoder includes two

modules: BiGGNN, which aims at capturing local graph structural information, and multi-head attention, which supplements the global dependencies that GNNs missed to enhance the model learning capacity. 3) *Query Phase*, which returns a set of top-k candidates from the search codebase that are most similar to the given query based on the cosine similarity, where the query is independent with the summary used for model training and the search codebase is different from the training set that used to train the model.

6.3.1 Problem Formulation

The goal of code search is to find the most relevant program fragment c based on the query q in natural language. Formally, given a set of training data $D = \{(c_i, q_i)|c_i \in \mathcal{C}, q_i \in \mathcal{Q}\}, i \in \{1, 2, ..., n\}$, where $\mathcal{C}$ and $\mathcal{Q}$ denote the set of functions and the corresponding queries. We define the learning problem as:

$$\min \sum_{i=1}^{n} \mathcal{L}(f_c(c_i), f_q(q_i)) \tag{6.3}$$

where f_c and f_q are the separate encoders *i.e.*, the neural network for programs and queries, which are learnt from the training data D. However, in the real scenario, since D is hard to collect, it is a common practise [92, 93, 180, 181] to replace q_i with the summary s_i regarding the function c_i i.e., $D = \{(c_i, s_i)|c_i \in \mathcal{C}, s_i \in \mathcal{S}\}$ where $\mathcal{S}$ is the corresponding summary set. The summary s_i usually describes the functionality of the function c_i. Once the encoders are trained *i.e.*, f_c and f_s, given a query q and the search code database $\mathcal{C}_{base}$ where $\mathcal{C}_{base} \neq D$, we can obtain the most relevant program as:

$$c = \max_{c_i \in \mathcal{C}_{base}} \text{sim}(f_c(c_i), f_s(q)) \tag{6.4}$$

where sim is a function such as a cosine similarity function to measure the semantic similarity between the program vector and query vector, which are produced by the encoder f_c and f_s respectively.

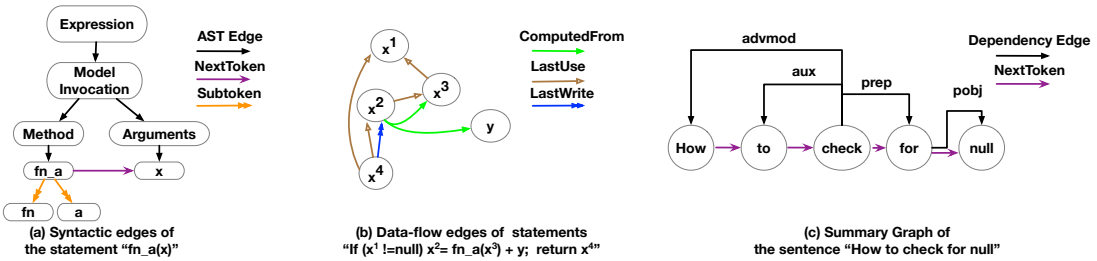


FIGURE 6.4: An example of the constructed graphs where (a) and (b) are the syntactic edges and data-flow edges of the program graph with a simple program snippet, please note that we distinguish the variable x into x^1, x^2, x^3, x^4 for ease of clarity in data-flow edges, (c) is the constructed summary graph.

6.3.2 Graph Construction

We introduce the graph construction for both programs and summaries that will be used for the encoder to learn.

6.3.2.1 Program Graph Construction

Given a program c, we follow Allamanis et al. [120] and extract its multi-edged directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes that are built on the Abstract Syntax Tree (AST) and $\mathcal{E}$ is the edges that represent the relationships between nodes. In particular, AST consists of terminal nodes and non-terminal nodes where terminal nodes correspond to the identifiers in the program and the non-terminal nodes represent different compilation units such as "Assign", "BinOp", "Expr". The edges can be categorized into syntactic edges *i.e.*, *AST Edge*, *NextToken* and data-flow edges *i.e.*, *ComputedFrom*, *LastUse*, *LastWrite*. In addition, according to Cvitkovic et al. [192], *SubToken* edge can further enrich the semantics of a program on the graph, we also include it and introduce extra subtoken nodes appearing in the identifier of the terminal node and connecting them to their original nodes. Hence, the node set $\mathcal{V}$ is the union of the entire AST nodes and subtoken nodes. The details of these types of edges with a simplified example of our constructed program graph in Figure 6.4a and Figure 6.4b are presented as follows:

- *AST Edge*, connects the entire AST nodes based on the abstract syntax tree.
- *NextToken*, connects each leaf node in AST to its successor. As shown in Figure 6.4a, for the statement "fn_a(x)", there is a *NextToken* edge points from "fn_a" to "x".

- *SubToken*, defines the connection of subtokens split from a identifier *i.e.*, variable names, function names based on *camelCase* and *pascal_case* convention. For example, "fn_a" will be divided into "fn" and "a". We further introduce the extra subtoken nodes apart from AST nodes to describe this relation.

- *LastUse*, represents the immediate last read of each occurrence of variables. As shown in Figure 6.4b, x^3 points to x^1 since x^1 is used as the conditional judgement, x^4 points to x^1 and x^2 because if the conditional judgement is not satisfied, x^4 points to x^1 directly, otherwise it will point to x^2.

- *LastWrite*, represents the immediate last write of each occurrence of variables. Since there is an assignment statement to update x^2, x^4 points to x^2 with a *LastWrite* edge.

- *ComputedFrom*, connects the left variable to all right variables that appeared in an assignment statement. In Figure 6.4b, x^2 connects x^3 and y by *ComputedFrom* edge.

6.3.2.2 Summary Graph Construction

Constituency parse tree [193] and dependency parse tree [182] are widely used to extract the structural information for the natural language text [23]. However, compared with the constituency parse tree, the dependency parse tree follows the dependency grammars [194] and describes the dependencies by the directed linked edges between tokens in a sentence, hence the constructed graph describes the relations among the tokens without redundant information *i.e.*, no non-terminal nodes in the constructed graph. Motivated by this, we also construct a directed graph based on dependency parsing [182] for the summary. Specifically, given a summary s, we extract its dependency parsing graph as $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$, where $\mathcal{V}'$ is a set of nodes where each node is the token in the original sequence s, $\mathcal{E}'$ denotes the relations between these tokens. Different edges represent tokens with different grammatical relations. Furthermore, there are a total of 49 different dependency edge types [195], for example, the edge "prep" is a prepositional modifier of a verb, adjective, or noun that serves to modify the meaning of the verb, adjective, noun [195]. In addition, we also construct *NextToken* and *SubToken* edges in the summary graph, which is similar to the program graph. To summarize, given a query "How to check for null", which is shown in Figure 6.4c, we can obtain its

summary graph where each token has different dependency relations with other tokens, for example, “How”, “to” and “for” are used to describe the relations with the verb “check” in the relations of ‘advmod”, “aux”, “prep” respectively.

6.3.3 Encoder

By the program graph construction and summary graph construction, we can get the program c with its program graph $\mathcal{G}$ and the summary s with its summary graph $\mathcal{G}'$, we further feed them into two encoders f_c and f_s to learn the vector representations for the program and the summary separately. A variety of GNN variants are proposed such as GCN [21], GGNN [20], GIN [196] to model the graph-structured data. Considering that most existing GNN variants ignore the direction for the message passing and treat the graph as the undirected graph, in this work, inspired by the recent advanced Bidirectional Gated Graph Neural Network (BiGGNN) [23], which leverages both incoming and outgoing message passing for the node interaction, we also use it for the graph learning. Furthermore, to supplement the global dependencies that are missed in GNNs, we propose a multi-head attention module to further improve the expression of BiGGNN. In the following, we only use the program c with its constructed program graph $\mathcal{G}$ for the explanation, the operations for the summary encoder f_s are the same with the program encoder f_c, the difference is that we feed the summary s with its constructed graph $\mathcal{G}'$ for the summary encoder to learn the representation.

6.3.3.1 Bidirectional GGNN

Given a program graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, BiGGNN learns the node embeddings from both incoming and outgoing directions for the graph. In particular, each node $v \in \mathcal{V}$ is initialized by a learnable embedding matrix $\boldsymbol{E}$ and gets its initial representation $\boldsymbol{h}_v^0 \in \mathbb{R}^d$ where d is the dimensional length. We apply the message passing function for a fixed number of hops i.e., K. At each hop $k \leq K$, for the node v, we apply a summation aggregation function to take as input a set of incoming (or outgoing) neighboring node vectors and output a backward (or forward) aggregation vector. The message passing is calculated as follows, where $N_{(v)}$ denotes the neighbors of node v and $\dashv$ / $\vdash$ is the

backward and forward direction.

$$\begin{aligned} \boldsymbol{h}^k_{\mathcal{N}_{\dashv(v)}} &= \text{SUM}(\{\boldsymbol{h}^{k-1}_u, \forall u \in \mathcal{N}_{\dashv(v)}\}) \\ \boldsymbol{h}^k_{\mathcal{N}_{\vdash(v)}} &= \text{SUM}(\{\boldsymbol{h}^{k-1}_u, \forall u \in \mathcal{N}_{\vdash(v)}\}) \end{aligned} \tag{6.5}$$

Then, we fuse the node embedding for both directions:

$$\boldsymbol{h}^k_{\mathcal{N}(v)} = \text{Fuse}(\boldsymbol{h}^k_{\mathcal{N}_{\dashv(v)}}, \boldsymbol{h}^k_{\mathcal{N}_{\vdash(v)}}) \tag{6.6}$$

Here the fusion function is designed as a gated sum of two inputs.

$$\begin{aligned} \text{Fuse(a, b)} &= \boldsymbol{z} \odot \boldsymbol{a} + (1 - \boldsymbol{z}) \odot \boldsymbol{b} \\ \boldsymbol{z} &= \sigma(\boldsymbol{W}_z[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{a} \odot \boldsymbol{b}; \boldsymbol{a} - \boldsymbol{b}] + \boldsymbol{b}_z) \end{aligned} \tag{6.7}$$

where $\odot$ is the component-wise multiplication, σ is a sigmoid function and $\boldsymbol{z}$ is a gating vector. Finally, we feed the resulting vector to a Gated Recurrent Unit (GRU) [62] to update node representations.

$$\boldsymbol{h}^k_v = \text{GRU}(\boldsymbol{h}^{k-1}_v, \boldsymbol{h}^k_{\mathcal{N}(v)}) \tag{6.8}$$

After K hops of computation, we obtain the final node representation $\boldsymbol{h}^K_v$ and then apply max-pooling over all nodes $\{\boldsymbol{h}^K_v, \forall v \in V\}$ to get a d-dim graph representation $\boldsymbol{h}^g$.

$$\boldsymbol{h}^g = \text{maxpool}(\text{FC}(\{\boldsymbol{h}^K_v, \forall v \in V\})) \tag{6.9}$$

where FC is a fully-connected layer.

6.3.3.2 Multi-Head Attention

To further capture the global interactions in the graph that BiGGNN missed [178, 179], we design a multi-head attention module to improve the model learning capacity. Note that in GraphSearchNet, the constructed program graph consists of terminal/non-terminal nodes and the terminal nodes are the tokens/subtokens of the original program c, hence we directly employ multi-head attention [16] over the sequence *i.e.*, c to capture the global dependency relations among these terminal nodes while ignoring the

non-terminal nodes for effective learning. Specifically, given the query[2], key and value matrix defined as $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$, where $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} = (\boldsymbol{E}_{t_1}, ..., \boldsymbol{E}_{t_l})$ are the embedding matrices for the program c and t_i is the i-th token in $c = \{t_1, ..., t_l\}$, the output by multi-head attention can be expressed as follows:

$$\boldsymbol{h}_{t_1}, ..., \boldsymbol{h}_{t_l} = \text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) \tag{6.10}$$

To get the final representation vector $\boldsymbol{h}^c$ over c, we use the mean pool operation on $\boldsymbol{h}_{t_1}, ..., \boldsymbol{h}_{t_l}$ as follows.

$$\boldsymbol{h}^c = \text{meanpool}(\{\boldsymbol{h}_{t_i}, \forall t_i \in c\}) \tag{6.11}$$

Here we choose the meanpool operation rather than the maxpool operation that used in BiGGNN is based on our experiments and we found that meanpool operation can get higher results in multi-head attention. Finally, we concatenate the vectors produced from Bidirectional GGNN i.e., $\boldsymbol{h}^g$ and multi-head attention i.e., $\boldsymbol{h}^c$ to get the final representation $\boldsymbol{r} = [\boldsymbol{h}^g; \boldsymbol{h}^c]$ for the program.

By the program encoder f_c and the summary encoder f_s, we can obtain the paired vector defined as $(\boldsymbol{r}, \boldsymbol{r}')$ where $\boldsymbol{r}$ and $\boldsymbol{r}'$ are the output vectors for the program and the summary respectively.

6.3.4 Training

Two different loss functions are used to train a model in deep code search systems [93, 101]. The ranking loss [197, 198] that was used in Gu et al. [101] needs to determine a hyper-parameter ϵ, however by our preliminary experiments, we found that the value is hard to determine manually, although it was set to 0.05 in their experiments. Instead, following CodeSearchNet [93], we directly use Cross-Entropy for the optimisation. Specifically, given n pairs of (c_i, s_i), we train f_c and f_s simultaneously by

[2]The query matrix is different from the natural language query, where the former is a matrix used in self-attention, and the latter is used to return the programs in code search. More explanations about the query matrix can be found in Section 6.2.3.

minimizing the loss as follows:

$$-\frac{1}{n}\sum_{i}^{n}\log\left(\frac{\boldsymbol{r}_i^T\boldsymbol{r}_i'}{\sum_j \boldsymbol{r}_j^T\boldsymbol{r}_i'}\right) \tag{6.12}$$

The loss function minimizes the inner product of $(\boldsymbol{r}_i', \boldsymbol{r}_i)$ between the summary s_i and its corresponding program c_i, while maximizing the inner product between the summary s_i and other distractor programs, *i.e.*, c_j where $j \neq i$.

6.3.5 Code Searching

Once the model is trained, and given a query q, GraphSearchNet aims to return a set of the most relevant programs. To achieve this, we first embed and stored all programs into vectors by the learnt program encoder f_c in an offline manner and these programs are from the search code database $\mathcal{C}_{base}$, which is independent with the data set used for model training. At the online query phase, for a new query q, GraphSearchNet embeds it via the summary encoder f_s and computes the cosine similarity between the query vector with all stored program vectors in the search codebase. The top-k programs whose vectors are the most similar *i.e.*, the highest top-k similarity values, to the query q are returned as the results. In GraphSearchNet, we select 10 candidate programs *i.e.*, $k = 10$ as the returned results.

6.4 Experimental Setup

In this section, we introduce the experimental setup including the dataset, evaluation metrics, compared baselines and the hyper-parameter configuration of GraphSearchNet.

6.4.1 Dataset

We select Java and Python data from CodeSearchNet [93] to evaluate our approach that can be generalized to different programming languages. The reason to choose Java and Python datasets is that there are some existing open-sourced tools [164, 199] for Java and Python programming language to facilitate constructing the program graph. We follow the same train-validation-test split with CodeSearchNet. In addition, we utilize

spaCy [200] to construct the summary graph. At the query phase, for all programs in the search codebase, we employ the open-sourced ElasticSearch [201] to store the vectors obtained by the learnt program encoder f_c for acceleration. Converting all programs from the search codebase into vectors costs nearly 70 minutes for Java or Python over **756k** samples in an offline manner. However, the query process provided by ElasticSearch is fast and it only takes about 0.15 seconds to produce 10 candidates for each query based on the cosine similarity. To sum up, in the evaluation section, we first investigate the performance of GraphSearchNet on the test set with some automatic metrics. Then we conduct a quantitative analysis over the real 99 queries with the programs from the search codebase to confirm the effectiveness of our approach.

6.4.2 Automatic Evaluation Metrics

Similar to the previous works [92, 95, 96], we use SuccessRate@k, Mean Reciprocal Rank (MRR) as our automatic evaluation metrics. We further add Normalized Discounted Cumulative Gain (NDCG) [93] as another evaluation metric.

- **SuccessRate@k.** SuccessRate@k measures the percentage of the correct results that existed in the top-k ranked results and it can be calculated as follows:

$$\text{SuccessRate@k} = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\delta(\text{FRank}_q \leq k) \tag{6.13}$$

  where Q is a set of queries, $\delta(\cdot)$ is a function, which returns 1 if the input is true and 0 otherwise. FRank is the rank position of the first hit result in the result list and we set k to 1, 5, 10.

- **MRR.** MRR is the average of the reciprocal ranks of results for a set of queries Q. The reciprocal rank of a query is the inverse of the rank of the first hit result, defined as follows:

$$\text{MRR} = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{\text{FRank}_q} \tag{6.14}$$

- **NDCG.** NDCG measures the gain of the result based on its position in the result list and it is the division of discounted cumulative gain (DCG) and ideal discounted

cumulative gain (IDCG) where DCG and IDCG are calculated as follows:

$$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{2^{\mathrm{rel}_i} - 1}{\log_2(i+1)} \quad \mathrm{IDCG_p} = \sum_{i=1}^{|\mathrm{REL}|_p} \frac{2^{\mathrm{rel}_i} - 1}{\log_2(i+1)} \tag{6.15}$$

where p is the rank position, rel_i is the graded relevance of the result at position i and $|\mathrm{REL}|_p$ is the list of relevant results up to position p. We set p equal to 10 for all experiments.

To sum up, for the above automatic metrics, the higher values, the better performance the approach achieves.

6.4.3 Compared Baselines

To demonstrate the effectiveness of GraphSearchNet, we select the following state-of-the-art approaches as our baselines, which are summarized as follows.

- **CodeSearchNet** [93]. CodeSearchNet provided a framework to encode programs and their summaries for code search. It consists of different encoders, *i.e.*, Natural Bag of Words (NBoW), 1D Convolutional Neural Network (1D-CNN), Bidirectional RNN (biRNN) and Self-Attention (SelfAtt) for encoding. Specifically, SelfAtt encoder has 3 identical multi-head attention layers with 128-dimensional lengths and each layer has 8 heads to learn different subspace features.

- **UNIF** [94]. UNIF followed the framework of CodeSearchNet [93], but it replaced the self-attention mechanism by combining each bag of code token vectors into a single code vector for the program and averaging a bag of query token vectors as a single vector for a query. Then it retrieved the most similar programs on the query.

- **DeepCS** [92]. DeepCS jointly learnt code snippets and natural language descriptions into a high-dimensional vector space. For code encoding, it fused method name, API sequence, and sequence tokens to learn the program vector. For the description, another RNN was utilized for learning the embedding. Based on the learnt vectors, it further used the ranking loss function [197, 198] to train the model and retrieved the most similar programs.

- **CARLCS-CNN** [95]. CARLCS-CNN embedded the representations for the code and query via a co-attention mechanism and co-attended the semantic relations of the embedded code and query via row/column-wise max-pooling. Then the learnt vectors were used for code search.

- **TabCS** [202]. TabCS incorporated the code textual features *i.e.*, method name, API sequence and tokens, the code structural feature *i.e.*, AST and the query feature *i.e.*, tokens with a two-stage attention network to learn the better code and query representation. Then two datasets were used for the evaluation. One of them is the Java dataset from CodeSearchNet [93], which is the same as ours.

- **QC-based CR** [97]. The original paper first designed a code annotations model to take the code as an input and output an annotation. Then it utilized the generated annotation with the query for a QN-based code retrieval model. In addition, they also added a QC-based code retrieval model to take the code with its query as the input to distinguish the code snippets from others. The entire model cannot run correctly by the official code, for simplicity, we only select QC-based code retrieval model as our baseline.

- **Code Summarization**. We add a code summarization baseline to validate whether the generated summaries by some code summarization systems with simple query strategies can produce comparable results. Specifically, we use CodeBERT [17] to fine tune two code summarization models for Java and Python respectively on our dataset with the official code from CodeXGLUE [82]. Then, we utilize the trained models to generate summaries for the samples from the testsets we used. Based on the generated summaries with the ground-truth summaries, we further utilize a word2vec model trained from the search codebase for vectorization. Then, we calculate the evaluation metrics based on the cosine similarity between two vectors and report the values for comparison.

- **CodeMatcher [90]**. CodeMatcher is a retrieval-based approach for code search. It first collected the metadata for query words to filter out the irrelevant noises and then iteratively performed the fuzzy search with the important words on the codebase.

Finally, it produced a set of returned candidate programs based on the importance of tokens in the candidate code snippet that matched the important words in a query.

We also select two widely used GNN variants *i.e.*, GGNN and GCN to investigate the performance of other GNNs as compared to BiGGNN. We directly replace BiGGNN module with GCN or GGNN for both encoders *i.e.*, f_c and f_s and keep the other settings unchanged for a fair comparison. For both GNN variants, since they are not used in the code search currently, we implement them from the scratch; for TabCS, since the evaluation dataset *i.e.*, Java dataset from CodeSearchNet, and the evaluation metrics are the same as us, we directly take the values reported in their paper [202] for comparison, for other baselines with the released source code, we directly reproduce their methods with the default settings on our dataset. We do not compare with MMAN [103] because their approach was conducted on C dataset and the relevant tool to construct the program graph for C programming language is not available so far.

6.4.4 Model Settings

We embed the most frequent 150,000 tokens for the programs and summaries in the training set with a 128-dimensional size. We set the dropout as 0.3 after the word embedding layer for the learning process. We use Adam [136] optimizer with an initial learning rate of 0.01 and reduce the learning rate by a factor of 0.5 with the patience equal to 2. We set the number of the maximum epoch as 100 and stop the training process when no improvement on MRR for 10 epochs. We clip the gradient at length 10. The batch size is set as 1,000, following the existing work [93]. 1000 samples in a batch can be explained as there is a corrected program c_i for the current summary s_i and the remaining 999 programs in this batch are distractors for the current s_i to distinguish. The number of hops on Java and Python is set to 4 and 3, respectively. The number of heads in multi-head attention is set to 2 on both Java and Python datasets for efficiency. All experiments were run on the DGX server with 80 cores and 180G RAM. Four 32 GB Nvidia Graphics Tesla V100 were used to train the models and the training process took nearly 4 hours to complete. All hyper-parameters are tuned on the validation set.

TABLE 6.1: Experimental results on Java and Python datasets as compared to baselines, where the marker * denotes the values are taken from the original paper and the marker - denotes the unreported metrics.

Model	Java					Python					p-value
	R@1	R@5	R@10	NDCG	MRR	R@1	R@5	R@10	NDCG	MRR	
CodeMatcher	53.77	74.63	80.30	61.67	62.71	58.74	79.46	85.03	67.32	68.47	0.15
Summarization	1.01	3.14	12.41	4.82	7.77	2.13	17.30	40.31	23.65	17.05	0.00
NBoW	52.11	72.65	78.98	54.30	61.60	56.85	78.47	84.21	55.18	66.60	0.06
biRNN	45.65	67.70	75.33	62.11	55.90	54.18	77.01	83.29	70.76	65.50	0.05
SelfAtt Encoder	42.09	62.96	71.01	67.18	52.00	57.81	79.02	84.46	76.62	67.40	0.07
1D-CNN	37.91	58.79	67.10	54.89	47.80	41.21	64.65	72.55	63.56	52.10	0.00
UNIF	52.79	73.39	79.67	46.70	62.20	57.47	78.95	84.43	44.25	67.10	0.05
DeepCS	33.40	56.50	67.30	48.78	44.64	53.60	73.40	78.90	66.10	62.91	0.01
CARLCS-CNN	42.60	57.90	67.50	47.90	43.31	68.70	78.10	82.80	70.08	67.00	0.03
TabCS	54.70*	68.30*	74.80*	-	53.90*	-	-	-	-	-	0.19
QC-based CR	19.03	40.77	51.68	33.94	29.72	21.02	43.83	54.17	36.26	32.03	0.00
GGNN	48.59	70.39	77.66	63.09	58.72	59.69	80.41	85.80	72.78	69.07	0.14
GCN	41.55	66.07	72.92	57.54	52.96	48.71	72.25	79.19	63.77	59.50	0.01
GraphSearchNet	**56.99**	**76.03**	**81.15**	**69.47**	**65.80**	**65.31**	**84.21**	**89.05**	**77.30**	**73.90**	-

6.5 Experimental Results

In this section, we aim to answer the following research questions by our experiments:

- **RQ1:** Can GraphSearchNet outperform current state-of-the-art baselines in terms of automatic metrics?
- **RQ2:** What is the performance of each component used in GraphSearchNet, are BiG-GNN and multi-head attention both beneficial in improving the performance?
- **RQ3:** What is the impact of setting different values of hops and heads on the performance? Whether these values perform consistently on Java and Python datasets?
- **RQ4:** Can GraphSearchNet provide more accurate programs for the real 99 queries provided by CodeSearchNet [93] compared with other baselines?

6.5.1 RQ1: Compared with Other Baselines.

Table 6.1 summarizes the results of GraphSearchNet in line with the baseline methods. Specifically, the columns R@1, R@5 and R@10 are the results of SuccessRate@k, where k is 1, 5 and 10. The second row is the retrieval-based approach. The third row presents the results by considering the program and the summary as the sequential text with different networks for retrieving, and the fourth row is the results for other GNN variants. From Table 6.1, we find that the performance of CodeMatcher is better than

DeepCS and UNIF, which is consistent with the conclusion from its original paper [90]. But its performance is still worse than GraphSearchNet which indicates that extracting hidden structures from programs and queries can help the model learn better semantic mapping relations. We can observe that in the third row, the performance of different approaches is inconsistent on different programming languages, for example, in terms of MRR, on the Java dataset, UNIF gets the best performance, while SelfAtt Encoder performs the best on Python dataset. One reason we conjecture is that these sequential approaches cannot capture the semantic mappings for both program and summary, hence they may bias to some specific programming languages and cannot generate well on others. Furthermore, we also find that QC-based CR has the worst performance on both datasets, we believe it is due to the fact that we only employ part of the officially released code for comparison, however, the entire project cannot run correctly by our huge efforts. From Table 6.1, we observe that using the simple query strategy to match the generated summary produced by the code summarization system with the ground truth has poor performance. We investigate the main reason for it. Since the smoothed BLEU-4 (a metric to evaluate the text similarity between the generated text with the ground-truth) on our Java and Python testset is 17.11 and 19.68, which are near to the reported values (17.65 and 19.06) from CodeXGLUE [82]. It demonstrates that our trained models are correct. Hence, we believe that it is because current code summarization models are still not able to generate summaries that are highly similar with ground-truths. These reported values from CodeXGLUE for code summarization models are still lower for the real application. In addition, we find that GGNN outperforms GCN by a significant margin for code search, which is reasonable since GRU cell in GGNN has been proven the effectiveness in filtering unnecessary features than GCN [179]. This also explains why current state-of-the-art models [9, 120, 183] select GGNN as the basic block.

The last row shows our results, we can see that GraphSearchNet outperforms these sequential approaches (the second row) by a significant margin on both Java and Python datasets, indicating that GraphSearchNet can return more relevant programs. The results indicate that, compared with these sequential models, which treat programs and summaries as sequences, by capturing the structural information, GraphSearchNet could

learn the semantic relations better, making it more effective and accurate in code search. Furthermore, we find that GraphSearchNet has a better performance than GGNN and GCN, we attribute it to the bidirectional message passing and multi-head attention module that can have powerful learning capacity. We also calculate the p-value of each baseline and GraphSearchNet on both Java and Python dataset. The values are shown in the rightmost column of Table 6.1 where the value of 0.00 denotes that the p-value is too small to be close to zero. We find that there are 10 baselines (13 baselines in total) that p-values are lower than 0.1, which demonstrates that GraphSearchNet provides statistically significant improvements compared with these baselines. More details about the performance of each module can be found that Section 6.5.2. Finally, we can see that the overall performance of Python is superior to Java, the main reason is that compared with Java, Python has simpler grammatical rules and language structures. The programs written by Python are closer to natural language queries. Therefore, the semantic mappings between the natural language queries and programs are easier for Python than Java.

✍ ►**Answer to RQ1**◄ The performance of the sequential-based approaches are inconsistent on Java and Python dataset, we contribute it to the missed structural information learnt by these networks. In contrast, GraphSearchNet outperforms them significantly. Furthermore, the bidirectional message passing and multi-head attention module could improve the model learning capacity to produce better results as compared to other GNN variants.

6.5.2 RQ2: Ablation Study on Each Component in GraphSearchNet.

We conduct the ablation study to investigate the impact of each component *i.e.*, BiGGNN and multi-head attention on the separate encoder to confirm the local structural information and the global dependencies in the graph are both beneficial for code search.

The experimental results are shown in Table 6.2, where the checkmark denotes the used component. We can observe that BiGGNN improves the performance significantly as compared to multi-head attention for the program encoder or the summary encoder. It

is an interesting finding because compared with the graphs used in the program scenario [9, 120, 178, 183], the transformer architecture [16], where multi-head attention is the key component in it, is dominating the NLP community. Hence, encoding the code into graphs is a common practice for a program, however, it is not very common for the natural language, even there are some existing works [23, 185–188]. One possible reason we conjecture is that the standard transformer [16] is equipped with the encoder-decoder architecture, which is more powerful for the text generation tasks such as machine translation [5, 16, 130], text summarization [203, 204]. The experimental results confirm the necessity to explore the structural information behind the query text for code search. Furthermore, we also find that the performance on Python dataset is still higher than Java when turning off some components, which indicates that the model is easier to learn semantics on Python rather than Java. We also calculate the p-value of GraphSearchNet and its different components on Java or Python testset respectively. The p-values are presented in the rightmost column of Table 6.2. We can observe that the majority of p-values are less than 0.1, which further confirms the effectiveness of each component in GraphSearchNet.

We further conduct an experiment to validate the effectiveness of our proposed encoder by replacing it with BiLSTM or transformer encoder for program or summary, respectively. Specifically, for BiLSTM encoder, we utilize the hidden states produced by a two-layer BiLSTM for the program or summary encoding, for the transformer encoder (abbr. Trans in Table 6.3), following Vaswani et al. [16], we utilize 6 encoder layers with 8 heads in each layer to encode the sequence and further utilize the max-pooling operation over the output to obtain the vector representations. The other settings are the same with GraphSearchNet and experimental results are presented in Table 6.3. We can observe that when modeling programs using the graph encoder while modeling the summaries using BiLSTM or transformer, MRR is 63.64, 64.38 on Java dataset, 72.12, 72.88 on Python dataset which is lower than GraphSearchNet, which demonstrates that the structural information in the query is beneficial for code search. Furthermore, we can also find that the values produced by replacing the program encoder with BiLSTM or transformer are also lower than GraphSearchNet. Hence, to sum up, the results confirm that our proposed encoder for both programs and summaries can help the model

TABLE 6.2: Ablation study of the performance on the encoder with different components on Java and Python data set.

Dataset	Program		Summary		R@1	R@5	R@10	NDCG	MRR	p-value
	Multi-Head	BiGGNN	Multi-Head	BiGGNN						
Java	✓		✓	✓	29.95	55.18	64.42	46.59	41.74	0.01
		✓	✓	✓	45.51	67.55	74.16	59.78	55.65	0.09
	✓	✓	✓		25.00	48.66	58.16	40.91	36.19	0.00
	✓	✓		✓	48.26	70.75	77.15	62.77	58.51	0.18
	✓	✓	✓	✓	**56.99**	**76.03**	**81.15**	**69.47**	**65.80**	-
Python	✓		✓	✓	13.76	35.27	46.63	28.53	24.52	0.00
		✓	✓	✓	55.55	77.11	83.24	69.28	65.41	0.12
	✓	✓	✓		27.55	52.49	61.89	43.89	39.23	0.00
	✓	✓		✓	60.03	80.90	86.37	73.31	69.59	0.27
	✓	✓	✓	✓	**65.31**	**84.21**	**89.05**	**77.30**	**73.90**	-

TABLE 6.3: Ablation study of the performance when the encoder is replaced with BiLSTM and Transformer encoder on Java and Python data set.

Dataset	Program			Summary			R@1	R@5	R@10	NDCG	MRR
	BiLSTM	Trans	GraphSearchNet	BiLSTM	Trans	GraphSearchNet					
Java	✓					✓	50.12	70.66	76.79	63.64	59.73
		✓				✓	52.39	73.49	79.51	66.10	62.04
			✓	✓			54.89	73.79	79.17	67.27	63.64
			✓		✓		55.45	74.88	80.36	67.83	64.38
			✓			✓	**56.99**	**76.03**	**81.15**	**69.47**	**65.80**
Python	✓					✓	59.56	80.28	85.85	72.69	68.96
		✓				✓	61.63	81.06	86.25	74.55	71.47
			✓	✓			62.93	82.37	87.87	75.02	72.12
			✓		✓		63.46	82.58	88.33	76.20	72.88
			✓			✓	**65.31**	**84.21**	**89.05**	**77.30**	**73.90**

achieve better performance.

✍ ►**Answer to RQ2**◄ BiGGNN and multi-head attention are both effective in improving the performance, however, the local structural information captured by BiGGNN is more critical. When incorporating both, GraphSearchNet can get the best performance.

6.5.3 RQ3: Hop&Head Analysis.

We further investigate the impact of the different number of hops (see K in Eq. 6.9) and heads (see h in Eq. 6.2) on the capacity to capture the local structural information and the global dependencies in the graph. Specifically, we set the number of hops in a range of 1 to 5 and heads in a range of 1 to 8 and keep the other settings unchanged for a fair comparison. Note that we only turn on BiGGNN with the multi-head attention closed for two encoders to analyse the impact of hops (and vice versa for head analysis), which is different to the settings in Section 6.5.2, where the specific module is closed for one

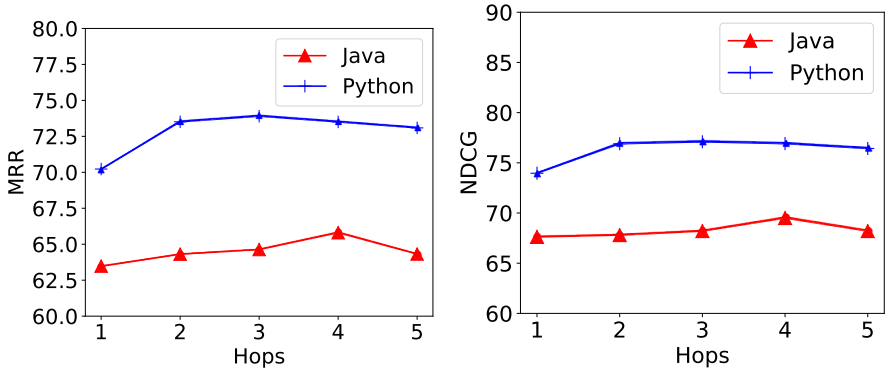
encoder. Since the trend on R@1, R@5 and R@10 is similar to MRR and NDCG, we only show the results of MRR and NDCG in Figure 6.5.

We can see that in Figure 6.5a and Figure 6.5b, with the increasing number of hops, MRR and NDCG improve gradually and they reach the highest at 4 and 3 for Java and Python respectively, after that the values begin to decrease, which is consistent with the findings that GNNs suffer from the over-squashing [179] when increasing the hops to learn the information from long-distant nodes. Hence, in our experiment, we set the value to 4 and 3 for Java and Python to achieve the best performance. The reason why Java needs more hops than Python is that the constructed graph for Java is larger than Python and the statistical distributions of nodes and edges on the dataset are shown in Table 6.4. We can observe that the average node size in the program graph and summary graph on the Java dataset is 125.99 and 17.44, which is larger than the Python dataset *i.e.*, 100.35 and 14.00 respectively. Furthermore, the average connections (edges) of Python on the program graph is larger than Java *i.e.*, 171.55 VS 125.22. Hence, the constructed graph for Python dataset tends to be smaller and better connected, which makes the model require fewer hops than Java to achieve the best performance. For multi-head attention to capture the global dependencies, from Figure 6.6a and Figure 6.6b, we find that the trend is basically the same on Java and Python. When increasing the number of heads to 2, the values reach the highest, which indicates 2 heads are enough to learn the global dependencies in a graph.

✍ ►**Answer to RQ3**◄ The optimal value of hops relies on the graph size in the dataset. In GraphSearchNet, we set the hops to 4 and 3 for Java and Python to achieve the best performance on the used dataset. The setting of the number of heads is consistent on both Java and Python datasets where 2 heads are enough to capture the global dependencies.

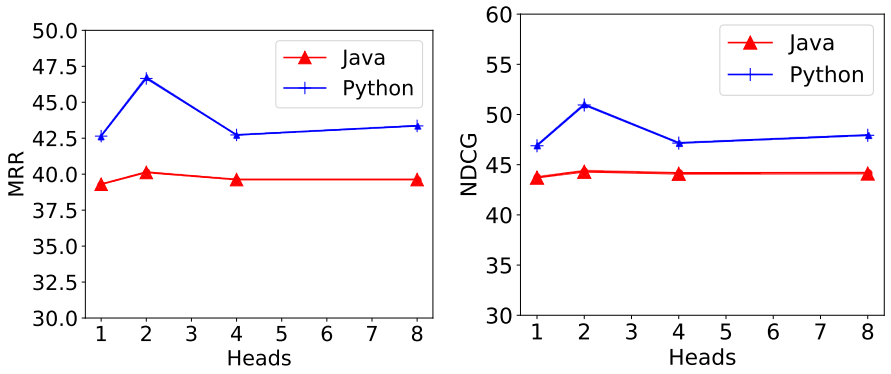
6.5.4 RQ4: Quantitative Analysis for Real 99 Queries.

We compare GraphSearchNet with CodeSearchNet, which consists of NBoW, biRNN, SelfAtt Encoder and 1D-CNN, and UNIF to evaluate the performance on the real 99 queries because the selected baselines tend to have a better performance than others



(A) MRR of Different Hops (B) NDCG of Different Hops

FIGURE 6.5: The effect of the number of hops to capture the local structural information.



(A) MRR of Different Heads (B) NDCG of Different Heads

FIGURE 6.6: The effect of the number of heads to capture the global dependency.

TABLE 6.4: The statistics of the graph size in the constructed program graph and the summary graph for Java and Python dataset.

Dataset		Program Graph			Summary Graph		
		min	max	avg	min	max	avg
Java	Node	55	200	**125.99**	3	200	**17.44**
	Edge	54	263	125.22	2	422	**31.29**
Python	Node	10	200	100.35	3	199	14.00
	Edge	9	477	**171.55**	2	408	24.21

TABLE 6.5: The average cosine similarity score of top-1 results over the 99 real queries.

Model	NBoW		biRNN		SelfAtt Encoder		1D-CNN		UNIF		GraphSearchNet	
	Java	Python	Java	Python	Java	Python	Java	Python	Java	Python	Java	Python
Avg	0.83	0.81	0.80	0.88	0.99	1.01	0.71	0.79	0.84	0.81	**1.25**	**1.25**

TABLE 6.6: Experimental results on 99 queries with their annotated programs.

Model	Java					Python				
	R@1	R@5	R@10	NDCG	MRR	R@1	R@5	R@10	NDCG	MRR
NBoW	79.79	93.93	94.94	84.81	86.50	77.77	95.95	96.96	75.42	86.00
biRNN	70.70	94.94	97.97	89.97	81.10	66.66	90.90	93.93	88.25	77.10
SelfAtt Encoder	67.67	92.92	94.94	90.79	78.30	75.75	92.92	93.93	87.46	83.40
1D-CNN	58.58	90.90	97.97	84.90	72.90	52.52	82.82	88.88	78.29	66.30
UNIF	79.79	96.96	96.96	86.03	87.10	78.78	94.94	96.96	77.08	86.20
GraphSearchNet	**83.73**	**97.92**	**97.97**	**91.57**	**88.33**	**78.88**	**96.91**	**96.97**	**89.73**	**87.29**

in Table 6.1. Specifically, we return the top-1 result for each query by different approaches. We manually check these top-1 results by different approaches and find that they are related to the query. To measure the degree of correlation, we calculate the cosine similarity between the query vector produced by the summary encoder f_s and the returned top-1 program vector produced by the program encoder f_c, and further average them over 99 queries to get the mean score. The mean similarity scores for these approaches are shown in Table 6.5. We can find that the cosine similarity scores produced by GraphSearchNet are higher than others, which indicates that the top-1 results returned by GraphSearchNet are more relevant to the queries than the baselines. We attribute it to the effectiveness of our approach to learning semantic relations between programs and queries.

CodeSearchNet [93] further provides 99 queries with manually annotated programs for evaluation. We also compare GraphSearchNet and the baselines on this evaluation set. Specifically, since one query in the total of 99 queries may have multiple annotated programs with different relevance scores in this evaluation set. Hence, for Java and Python, we first extract the annotated program with the highest relevance score for each query. If multiple programs share the same highest relevance score, we randomly select one program. In this way, we can construct a testset where each sample is a pair of query and its most relevant annotated program. We utilize this testset for evaluation. Since there are only 99 samples in this testset, hence the answer of each query is retrieved from 99 candidate programs instead of 1000 programs, which is used for the other experiments in the rest of the paper. We directly test the performance of different models on this testset and the experimental results are presented in Table 6.6. We find that GraphSearchNet still achieves better performance on this testset compared with baselines, which indicates the effectiveness of our approach on different testsets. In addition, we can observe

that these values are improved greatly compared with the values in Table 6.1, it is reasonable since the answer of each query is retrieved from 99 candidate programs, which is much easier for the model to distinguish than 1000 programs used in Table 6.1.

Furthermore, we show two examples of the returned top-1 results for Java by GraphSearchNet and the best baseline (i.e., SelfAtt Encoder in Table 6.5) in Figure 6.7 and Figure 6.8. We also show two examples for Python in Figure 6.9 and Figure 6.10 with the same queries on Java dataset. The completed top-10 results of 99 queries are provided in our official repository. We find that given the query "get executable path" and "how to determine a string is a valid word", the generated results by SelfAtt Encoder are less relevant to the query on both Java and Python. In particular, for the query, "get executable path", the results by GraphSearchNet are more accurate than SelfAtt Encoder to produce a program with the functionality of returning a path. For the query, "how to determine a string is a valid word", aims at verifying whether a string is a word, the produced results by SelfAtt Encoder indicate that it misunderstood the query semantics and return the programs related to the password verification for both Java and Python. In contrast, GraphSearchNet can produce more accurate and semantic-relevant programs based on the query. We also provide one example, which GraphSearchNet cannot provide the correct code snippet on Java and Python dataset. As shown in Figure 6.11 and Figure 6.12, for the query "aes encryption", the returned results for SelfAtt Encoder are more accurate than GraphSearchNet. Specifically, SelfAtt provides the accurate code snippet for the query, while GraphSearchNet provides a decryption code snippet on Java dataset and provides an irrelevant function on Python dataset. The main reason is that GraphSearchNet leverages dependency parsing to construct a graph to capture token relations in a query. However, for the query of 'aes encryption", since it is short and only has two tokens, the constructed graph is very simple without sufficient edges to fully express the relations between tokens. GraphSearchNet cannot learn sufficient semantics and thus provide inaccurate results. We further conduct statistical analysis on the queries that GraphSearchNet fails to provide accurate results over 99 queries. We find that in a total of 99 queries, 57 queries start with a verb such as "convert int to string", 12 queries start with the word how such as "how to get current date" and 11 queries start with the noun phrase such as "aes encryption". By comparing these queried results, we

FIGURE 6.7: The first example of the queried top-1 results for Java.

```
Query: get executable path
SelfAtt Encoder:
private Path getCollectionPath() {
    Path collectionPath = PathUtils.
    ReportsDirectory.getPathForObject(getInstance());
    return collectionPath.resolve(
     ↪  provider.getArtifactPath(getInstance()));
}
GraphSearchNet:
public static String getPathToExecutable (File featureDir,
 ↪  String path) {
    File scriptPath = new File(path);
    if ( ! scriptPath.isAbsolute() ) {
        scriptPath = new File(featureDir, path);
    }
    return scriptPath.getAbsolutePath();
}
```

find that generally, GraphSearchNet has relatively poor performance for noun phrases on Java and Python dataset compared with two other kinds of queries. The main reason is that the noun phrases are short. GraphSearchNet cannot learn sufficient semantics when the provided query has limited information.

✍ ►**Answer to RQ4**◄ By calculating the average cosine similarity score over the returned top-1 program vectors with the real query vector, we find that GraphSearchNet could produce more accurate and semantic-relevant programs. In addition, we present some examples as compared to the baselines for analysis.

FIGURE 6.8: The second example of the queried top-1 results for Java.

```
Query: how to determine a string is a valid word
SelfAtt Encoder:
private static String promptForPassword(String passwordName,
    String commandLineOption, String envVarName) throws
    Exception {
        final Console console = System.console();
        if(console == null) {
            throw new Exception("Cannot allocate a console. Set
                env var "+envVarName+" or "+commandLineOption+" on
                commandline in that case");
        }
        return new String(console.readPassword("[%s]",
            passwordName+" password:"));
    }
GraphSearchNet:
public static boolean isWord (@Nullable final String sStr) {
        if (StringHelper.hasNoText (sStr))
            return false;
        return isWord (sStr.toCharArray ());
    }
```

FIGURE 6.9: The first example of the queried top-1 results for Python.

```
Query: get executable path
SelfAtt Encoder:
def get_mp_bin_path():
    plat = platform.system()
    if plat == "Linux":
        return resource_filename(__name__,
        ↪ "bin/metaparticle/linux/mp-compiler")
    elif plat == "Windows":
        return resource_filename(__name__,
        ↪ "bin/metaparticle/windows/mp-compiler.exe")
    elif plat == "Darwin":
        return resource_filename(__name__,
        ↪ "bin/metaparticle/darwin/mp-compiler")
    else:
        raise Exception("Your platform is not supported.")
GraphSearchNet:
def get_path():
    return os.path.abspath(os.path.dirname(
    ↪ os.path.dirname(__file__)))
```

FIGURE 6.10: The second example of the queried top-1 results for Python.

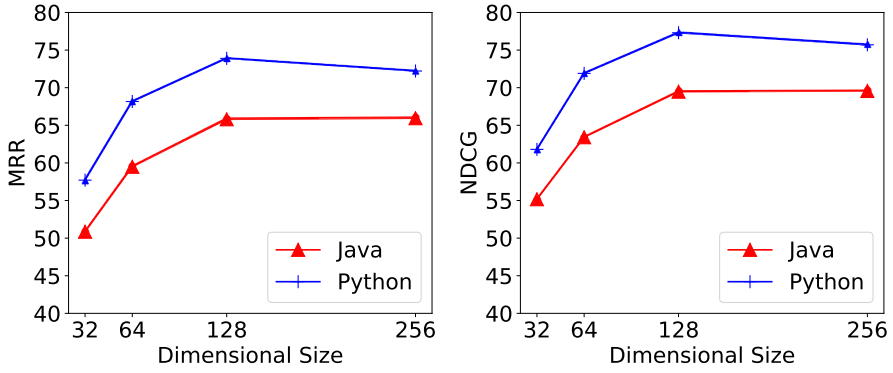
```
Query: how to determine a string is a valid word
SelfAtt Encoder:
def _validate_admin_password(admin_password):
    password_regex = r"[A-Za-z0-9@#$%^&+=]{6,}"
    pattern = re.compile(password_regex)
    if not pattern.match(admin_password):
        raise ValueError(red("The password must be at least 6
        ↪ characters and contain only the following
        ↪ characters:\n" "A-Za-z0-9@#$%^&+="))
    return admin_password
GraphSearchNet:
 def isValid(self, text, word):
    return bool(re.search(word, text, re.IGNORECASE))
```

FIGURE 6.11: One example of the queried top-1 results for the query of "aes encryption" on Java.

```
Query: aes encryption
SelfAtt Encoder:
public static byte[] aesEncrypt(byte[] input,
byte[] key, byte[] iv) throws Exception {
            return aesEncrypt(input, new SecretKeySpec
               (key, "AES"), iv);
    }
GraphSearchNet:
 public static String aesDecrypt(byte[] input,
byte[] key) {
            byte[] decryptResult = aes(input, key,
                Cipher.DECRYPT_MODE);
            return new
               String(decryptResult, Charsets.UTF_8);
    }
```

FIGURE 6.12: One example of the queried top-1 results for the query of "aes encryption" on Python.

```
Query: aes encryption
SelfAtt Encoder:
def _aes_encrypt(self, text, key):
pad = 16 - len(text) % 16
text = text + pad * chr(pad)
encryptor = AES.new(key, 2,
     '0102030405060708')
enc_text = encryptor.encrypt(text)
enc_text_encode = base64.b64encode(enc_text)
return enc_text_encode
GraphSearchNet:
 def _maybe_add_hash(tsig_alg, hash_alg):
try:
   _hashes[tsig_alg] = dns.hash.get(hash_alg)
except KeyError:
   pass
```



(A) MRR (B) NDCG

FIGURE 6.13: The effect of the dimensional size with regard to MRR and NDCG.

6.6 Discussion

This section presents the impact of the dimensional size used in GraphSearchNet. We further discuss the widely used pre-trained models in the program scenario and followed by the threats to the validity of our work.

6.6.1 Impact of Dimensional Size

We study the impact of different dimensional sizes on the performance of GraphSearchNet. We only change the dimensional size (32/64/128/256) and keep the remaining settings unchanged for the evaluation. The results are shown in Figure 6.13. We can observe that the performance improves greatly as the feature size increases since the learning capacity of the model is increased with the growing dimension size. However, we can also see that after the 128-dimensional size, MRR and NDCG on Java dataset improve slightly, while on Python dataset, there is an obvious decrease from the highest. We conjecture this is caused by when set to a higher dimensional size, the learning capacity is further increased to make the model overfit to the Python dataset, which harms the performance on Python dataset. In addition, with the increasing dimensional size, the training time and GPU memory consumption are also increased. For unification, we set the dimensional size to 128 on both Java and Python datasets for the evaluation.

TABLE 6.7: The MRR values as compared to CodeBERT and GraphCodeBERT.

Dataset	CodeBERT	GraphCodeBERT	GraphSearchNet
Java	64.44	**66.21**	65.80
Python	73.47	**74.87**	73.90

6.6.2 Pre-trained Models

Recently, there are some unsupervised pre-trained approaches e.g., CodeBERT [17] and GraphCodeBERT [18] that aim at learning the general program representations for a variety of code-related tasks. Code search is selected as one of the downstream tasks. We also compare the performance of GraphSearchNet with CodeBERT and GraphCodeBERT. Specifically, we use the officially released code by CodeBERT [17] and GraphCodeBERT [18] with the default hyper-parameters to validate the performance on our dataset. Similar to CodeSearchNet [93] and GraphSearchNet, we set the batch size on the testset equal to 1000, which means the answer of each query is retrieved from 1000 candidate programs. Following CodeBERT and GraphCodeBERT, we use MRR as the evaluation metric and the experimental results are presented in Table 6.7. We can observe that the MRR values of GraphSearchNet are higher than CodeBERT, which demonstrates that utilizing structural information in code and query can help the model achieve better performance than CodeBERT, which only treats them as the sequences, even GraphSearchNet has limited data for training. However, the MRR values of GraphSearchNet are slightly lower than GraphCodeBERT. On one hand, it demonstrates that using structural information (such as dataflow in GraphCodeBERT) is effective. On other hand, GraphCodeBERT utilizes 2.3M functions for pre-training and the model has a total number of 125M parameters. In contrast, GraphSearchNet only uses 0.25M data to train and the model only has 2M parameters. Considering the scale of the used data (0.25M VS 2.3M) and the model parameters (2M VS 125M), we believe GraphSearchNet achieves comparable performance with GraphCodeBERT.

6.6.3 Threats to Validity

The internal threats to validity lie in our implementations, the graph construction for the program and the summary, the model implementation. To reduce this threat, we

utilize the open-source tool [164, 199] for Java and Python program graph construction, Spacy [200] for the summary graph construction. To reduce the implementation threat, the co-authors carefully check the correctness of our implementation. We further make implementation public at `https://github.com/shangqing-liu/GraphSearchNet` for further investigation.

The external threats to validity include the selected datasets and the evaluation of the baselines and the evaluation metrics. To reduce the impact of the dataset, we select the Java and Python datasets from CodeSearchNet [93], which has been widely used for other works [17, 18, 82, 83] to evaluate the model performance. For the evaluation of baselines, we select the open-sourced baselines [92–95, 97, 202] and evaluate the performance based on the default hyper-parameters used in the original papers. We consider these default parameters are optimal. We carefully checked the released code to mitigate the threat and plan to evaluate more hyper-parameters in the future. For the evaluation metrics, there are some other metrics rather than what we used such as mean average precision (MAP) [205], which can be used to evaluate the retrieval system, however, we follow the existing works [92–95, 97, 202] in code search and select the widely used SuccessRate@k, MRR and NDCG for the evaluation.

6.7 Conclusion

In this chapter, we propose GraphSearchNet, a novel graph-based approach for code search to capture the semantic relations of the source code and the query. We construct the program graph and the summary graph with Bidirectional Graph Neural Network *i.e.*, BiGGNN to learn the local structural information hidden in the sequential text. We further employ multi-head attention to capture the global dependencies that BiGGNN fails to learn in a graph. The extensive experiments on Java and Python datasets demonstrate the effectiveness of GraphSearchNet.

Chapter 7

ContraBERT: Enhancing Code Pre-trained Models

In the previous Chapter 3 to Chapter 6, we explore the way to incorporate program structures for various software engineering tasks (i.e., commit message generation, vulnerability identification, source code summarization and code search). In this chapter, since the large-scale pre-trained model GraphCodeBERT also incorporates program structures to learn program semantics, we also want to explore its robustness and further propose a framework to enhance the robustness of these pre-trained models.

7.1 Introduction

It has already been confirmed that the "big code" era [163] is coming due to the ubiquitousness of software in modern society and the accelerated iteration of the software development cycle (design, implementation and maintenance). According to a GitHub official report [7] in 2018, GitHub has already reached 100 million hosted repositories. The Evans Data Corporation [82] also estimated that there are 23.9 million professional developers in 2019 and that number is expected to reach 28.7 million in 2024. As a result, the availability of code-related data is massive (e.g., billions of code, millions of changed code, bug fixes and code documentation), which yields a hot topic in both

academia and industry. That is how to adopt the data-driven approach (e.g., deep learning) to solve conventional software engineering (SE) problems.

Deep learning has been widely applied to diverse SE tasks (AI4SE) such as software vulnerability detection [9, 120, 206], source code summarization [127, 128, 178], deep code search [92, 207] and source code completion [208–210]. Besides, the early works [13, 14, 56, 93, 211] directly utilized vanilla deep learning techniques such as Long-Short Memory Networks (LSTMs) [15] and Convolutional Neural Networks (CNNs) [3] for different tasks. Later works [9, 120, 127, 178, 207, 212–214] customized different network architectures to satisfy the characteristics of the specific task for achieving the best performance. For example, since complicated data dependencies and control dependencies are easier to trigger software vulnerabilities, Devign [9] incorporated different kinds of program structure information with Code Property Graph [153] to Graph Neural Networks [20] for vulnerability detection. Considering code duplication [172] is common in the "big code" era, Liu et al. [178] combined the retrieved code-summary pair to generate high-quality summaries. Although these customized networks have achieved significant improvements on specific tasks, the generalization performance is still low. To address this limitation, some researchers propose to utilize unsupervised techniques with the massive amount of data to pre-train a general model [17, 18, 82, 83, 109–113] and then fine-tune it for different downstream tasks. For example, CuBERT [215] pre-trained BERT [216] on a large collected Python corpus (7.4M files) and then fine-tuned it on different tasks such as variable-misuse identification and wrong binary operator identification. CodeBERT [17] pre-trained RoBERTa [217] for programming languages (PL) with their natural language (NL) comments on the open-source six programming languages [93] and evaluated it on code search and source code summarization. GraphCodeBERT [18] further incorporated data flow information to encode the relation of variables in a program for pre-training and demonstrated its effectiveness on four downstream tasks.

The aforementioned pre-trained models have a profound impact on the AI4SE community and have achieved promising results on various tasks. With the widespread use of pre-trained models, an important question is whether these models are robust

to represent code semantics. Our preliminary study has demonstrated that state-of-the-art pre-trained models are not robust to a simple label-preserving program mutation such as variable renaming. Specifically, we utilize the test data of clone detection (POJ-104) [10] (a task to detect whether two functions are semantic equivalence with different implementations) provided by CodeXGLUE [82] and select those samples that are predicted correctly by the pre-trained CodeBERT [17] and GraphCodeBERT [217]. Then we randomly rename variables within these programs from 1 to 8 edits. For example, 8 edits mean that we randomly select 8 different variables in a function and rename them for all occurrences with the newly generated names. If one function has less than 8 variables, we will rename the maximum number of variables. We then utilize these newly generated mutated variants to evaluate the model prediction accuracy based on the cosine similarity of the embedded vectors of these programs. Surprisingly, we find that either CodeBERT or GraphCodeBERT suffers greatly from renaming operation and the accuracy reduces to around 0.4 when renaming edits reach to 8 (see Figure 7.1). It confirms that pre-trained models are not robust to adversarial examples. However, it is challenging to improve the robustness of pre-trained models. Although the latest work by Yang et al. [218] proposed some attack strategies to make CodeBERT and GraphCodeBERT have poor performance on adversarial samples. They further combined adversarial samples with original samples to fine-tune pre-trained models without any changes to the model architecture to improve prediction robustness on downstream tasks. However, a newly designed model that inherently solves the weakness of robustness is not involved in their paper.

In this paper, we propose ContraBERT, an unsupervised contrastive learning-based framework to enhance the robustness of existing pre-trained models in code scenarios. Compared with Yang et al. [218], we design a new pre-trained model that takes masked language modeling (MLM) and contrastive pre-training task as the pre-training tasks to improve model robustness. To design a contrastive pre-training task to help the model group similar samples while pushing away the dissimilar samples, we define nine kinds of simple or complex data augmentation operators that transform the original program and natural language sequence into different variants. Given an existing pre-trained model such as CodeBERT or GraphCodeBERT, we take the original sample as

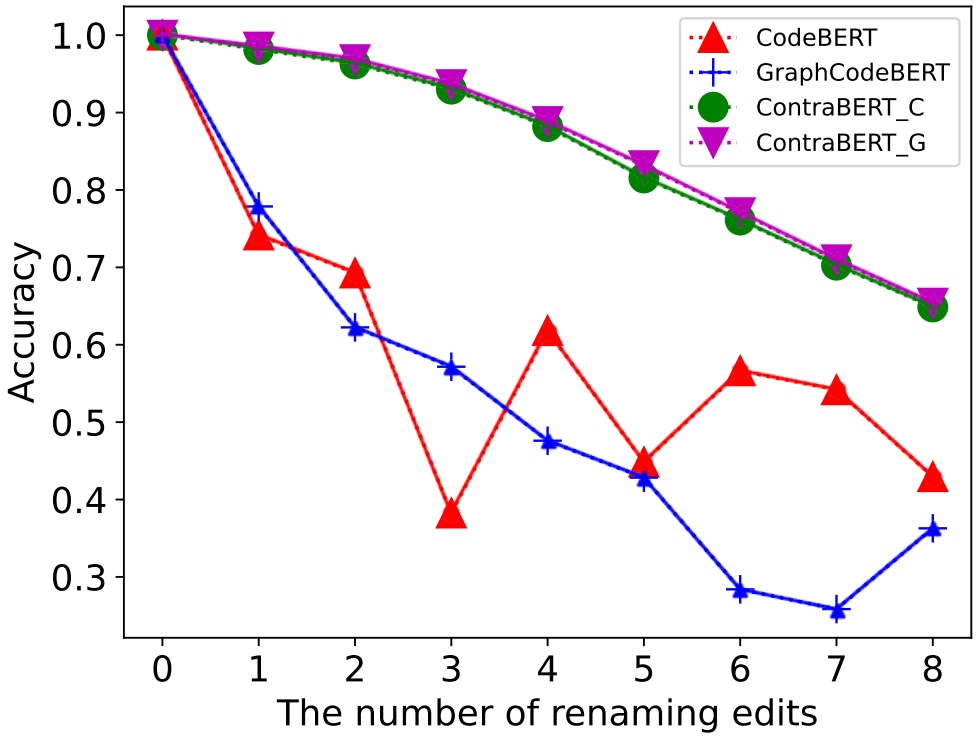


FIGURE 7.1: Adversarial attacks on clone detection(POJ-104).

well as its augmented variants as the input to train the model with MLM and contrastive pre-training task, where MLM is utilized to help the model learn better token representations and contrastive pre-training task is utilized to help the model group the similar vector representations to enhance model robustness. As shown in Figure. 7.1, ContraBERT_C and ContraBERT_G denote the models are pre-trained from CodeBERT and GraphCodeBERT with our approach respectively, we observe that with the increasing number of edits, although the performance continues to drop, the curve for ContraBERT is much smoother. The prediction accuracy of ContraBERT_C and ContraBERT_G outperform CodeBERT and GraphCodeBERT significantly, indicating that ContraBERT_C and ContraBERT_G are more robust than the original models. We further perform an ablation study to confirm each type of defined PL-NL augmentation operator is effective to improve the model robustness. Finally, we conduct broad research on four downstream tasks (i.e., clone detection, defect-detection, code-to-code-trans and code search) to illustrate that these robustness-enhanced models provide significant improvements as compared to the original models. In summary, our main contributions are as follows:

- We present a framework ContraBERT that enhances the robustness of existing pre-trained models in the code scenario by the pre-training tasks of masked language modeling and contrastive learning on original samples as well as the augmented variants.

- We design nine kinds of simple or complex data augmentation operators on the programming language (PL) and natural language sequence (NL). Each operator confirms its effectiveness to improve the model's robustness.

- The broad research on four downstream tasks demonstrates that the robustness-enhanced models provide improvements as compared to the original models. Our code and model are released on [219] for reproduction.

7.2 Background

In this section, we briefly introduce CodeBERT and GraphCodeBERT which will be adopted as our original pre-trained models for ContraBERT.

7.2.1 CodeBERT

CodeBERT [17] is pre-trained on an open-source benchmark CodeSearchNet [93], which includes 2.1M bimodal NL-PL (comment-function) pairs and 6.4M unimodal functions without comments across six programming languages. The model architecture is the same with RoBERTa [217], which utilizes multi-layer bidirectional Transformer [16] for unsupervised learning. Specifically, CodeBERT consists of 12 identical layers, 12 heads and the dimension size for each layer is 768. In total, the number of model parameters reaches 125M. Two different pre-training objectives are used, the first one is masked language modeling (MLM), which is trained on bimodal data. MLM objective targets predicting the original tokens that are masked out in NL-PL pairs. To fully utilize unimodal data, CodeBERT further uses Replaced Token Detection (RTD) objective on both bimodal and unimodal samples. RTD objective is designed to determine whether a word is original or not. At the fine-tuning phase, two downstream

tasks (i.e., code search and source code documentation generation) are used for evaluation. The experimental results demonstrate that CodeBERT outperforms supervised approaches on both tasks.

7.2.2 GraphCodeBERT

GraphCodeBERT [18] is a pre-trained model for code, which considers structures in code. Specifically, it incorporates the data flow of code to encode the relations of "where the value comes from" between variables in the pre-training stage. In addition to the pre-training task of masked language modeling (MLM), GraphCodeBERT further introduces two new structure-aware pre-training tasks. The first one edge prediction is designed to predict whether two nodes in the data flow are connected. The other node alignment is designed to align edges between code tokens and nodes. GraphCodeBERT utilizes NL-PL pairs for six programming languages from CodeSearchNet [93] for pre-training. It is fine-tuned on four downstream tasks including code search, clone detection, code translation and code refinement. The extensive experiments on these tasks confirm that code structures and the defined pre-training tasks help the model achieve state-of-the-art performance on these tasks.

7.3 Approach

In this section, we first present an overview of our approach, then detail each component including PL-NL augmentation, model design in pre-training and the fine-tuning settings for downstream tasks.

7.3.1 Overview

The overview of ContraBERT is shown in Figure. 7.2. Specifically, given a pair of the function C with its comment W (i.e., (C, W)), we first design a set of PL-NL augmentation operators $\{f(*)\}, \{g(*)\}$ to construct the simple or complex variants for C and W respectively. In the pre-training phase, initialized from existing pre-trained models such as CodeBERT or GraphCodeBERT, we further pre-train these models on the original samples and their augmented variants with masked language modeling (MLM) and

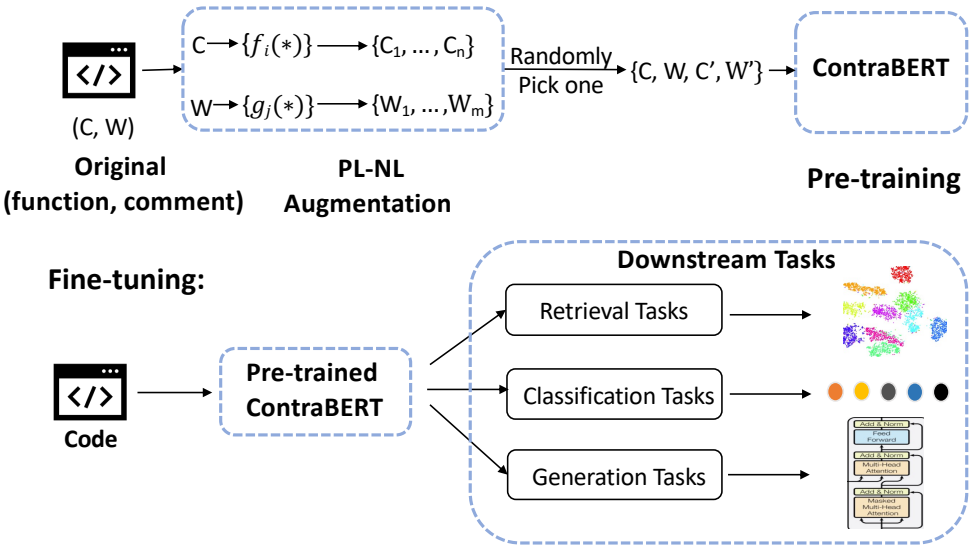


FIGURE 7.2: The Overview of ContraBERT.

contrastive pre-training task to enhance the model robustness. Finally, when ContraBERT is pre-trained over a large amount of unlabeled data, we fine-tune it for different types of tasks such as retrieval tasks, classification tasks and generation tasks with the task-specific data in a supervised manner.

7.3.2 PL-NL Augmentation

Given a program C, clone detection [11] could help to identify a semantically equivalent program C'. However, this technique is unrealistic in practice. For any function in a fixed dataset, we cannot guarantee that we will be able to find the semantically equivalent variants. Furthermore, clone detection usually takes a project for analysis, which is not applicable to a single function. Hence, we consider constructing augmented variants based on the original samples. Compared with the existing works [84, 220] that only focus on program mutations, we design a set of natural language (NL) sequence augmented operators. Specifically, we design a series of simple operators and complex operators for both PL and NL to construct variants.

7.3.2.1 Program (PL) Augmentation Operators

For program augmented operators, we design four kinds of complex operators and one kind of simple operator.

Complex Operators:

- Rename Function Name (RFN). It is designed to replace the function name with a new name that is taken randomly from an extra vocabulary set constructed on the pre-training dataset. We extract all function names in the pre-training dataset for the construction. Since each sample in the dataset is a single function, the renamed function preserves the equivalent semantics to the original function.

- Rename Variable (RV). It renames variables in a function. A random number of variables for all occurrences in the function will be replaced with the new names taken randomly from an extra vocabulary set. We extract all variable names from the pre-training dataset to construct this vocabulary set. This operator only mutates the variable names and all occurrences of them with the new variable names, which does not change the semantics of the original function.

- Insert Dead Code (IDC). It means to insert unused statements in a function. To generate unused code statements, we traverse AST to identify the assignment statements and then randomly select one assignment statement to rename its variables with new names that have never appeared in the same function. After that, we consider it as the dead code and insert it at the position after the original assignment statement. As the inserted dead code does not change the original program behaviour, IDC is regarded as the semantically equivalent operator.

- Reorder (RO). It randomly swaps two lines of statements that have no dependency on each other in a basic block in a function body such as two declaration statements appearing on two consecutive lines without other statements between them. We traverse AST and analyze the data dependency for extraction. Since the permuted statements are independent without data dependency, this operator preserves the original program semantics.

Simple Operators:

- Sampling (SP). It randomly deletes one line statement from a function body and preserves others. It can serve as regularizers to avoid overfitting [220].

7.3.2.2 Comment (NL) Augmentation Operators

Apart from the program augmentation, we further design one kind of complex operator and three kinds of simple operators for comment augmentation operators as follows:

Complex Operators:

- Back Translation Mutation (Trans). It refers to translating a source sequence into another language (target sequence) and then converting this target sequence to the original sequence [221]. We use the released tool [222] for the implementation where the source is in English and the target is in German.

Simple Operators:

- Delete. It randomly deletes a word in a comment.
- Switch. It randomly switches the positions of two words in a comment.
- Copy. It randomly copies a word and inserts it after this word in a comment.

Given a function C with its paired comment W, we utilize the above augmentation operators on C and W respectively to obtain the augmentation sets, which are defined as S_C and S_W respectively. Specifically, each operator is conducted once to get its corresponding augmented variant and insert it into the corresponding augmentation set. For the operator IDC, which may not get its variant for some specific functions, we ignore it and use other operators for the construction. Then we randomly select an augmented version from S_C and S_W (i.e., $C' \in S_C$ and $W' \in S_W$) and construct the quadruple (C, W, C', W') for the pre-training. Note that during the pre-training process, at each learning step, (C', W') is randomly selected from the augmented sets S_C and S_W respectively. Hence, each augmented sample in the sets is used when the model has sufficient learning steps.

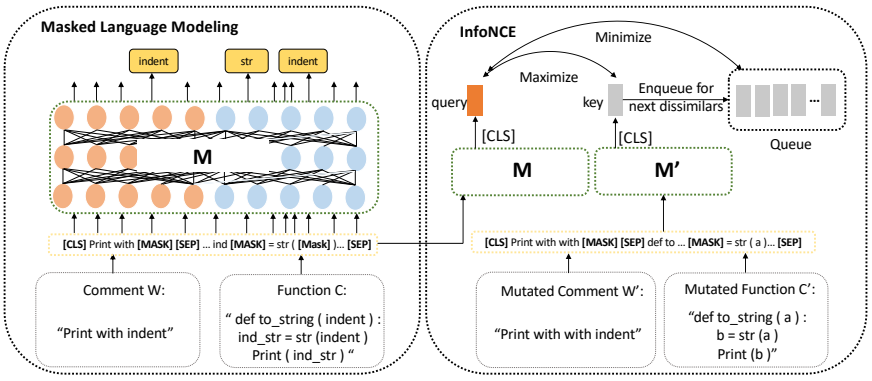


FIGURE 7.3: The model design for ContraBERT where the encoder M can be represented by the existing pre-trained models such as CodeBERT. The initial weights of the encoder M' are the same as the encoder M while the weight update is different.

7.3.3 Model Design and Pre-training

Basically, ContraBERT is further trained from existing pre-trained models. We directly utilize the existing pre-trained model and further pre-train it with masked language modeling (MLM) and contrastive pre-training task to enhance its robustness. The model design of ContraBERT is presented in Figure. 7.3.

7.3.3.1 Model Design

As shown in Figure. 7.3, ContraBERT consists of two separate encoders M and M', where M can be represented by any pre-trained models such as CodeBERT. The model architecture of M' is the same with the encoder M and the initial weights are also the same with M. However, the weight update strategy is different with M. Specifically, given a quadruple (C, W, C', W') from Section 7.3.2, we construct two input sequences $X = \{[CLS], W, [SEP], C, [SEP]\}$ and $X' = \{[CLS], W', [SEP], C', [SEP]\}$, where "$[CLS]$" indicates the beginning of a sequence and "$[SEP]$" is a symbol that concatenates two kinds of sequence. We utilize the encoder M and M' to encode the masked input sequence X and X' respectively.

7.3.3.2 Pre-training Tasks

Masked language modeling (MLM) is an effective and widely adopted pre-training task to learn the effective token representations [216,217], we also utilize it as one of our pre-training tasks. However, by our preliminary results, we observe that the models trained

by MLM are weak to the adversarial examples, we further introduce a contrastive pre-training task to group the similar data and push away the dissimilar data to reshape the learnt space for encoder M to enhance the model robustness.

Masked Language Modeling (MLM). We utilize MLM to learn token representations in a sequence. Specifically, given the sequence $X = \{[CLS], W, [SEP], C, [SEP]\}$, a random set of positions in X are masked out. We select 15% tokens to mask out and obtain the masked token set. Furthermore, we replace 80% of the masked tokens in this set with the "[MASK]" symbol, 10% with the random tokens from the vocabulary set and the remaining 10% unchanged. We configure these settings since they are confirmed effective to learn the token representations in a sequence [216, 217]. The loss function $\mathcal{L}_{\text{MLM}}$ can be expressed as follows:

$$\mathcal{L}_{\text{MLM}} = -\sum_{x_i \in M} \log p(x_i | X^{mask}) \tag{7.1}$$

where X^{mask} is the masked input sequence and M is the masked token set.

Contrastive Pre-training. We design a contrastive pre-training task that uses InfoNCE [223] as the loss function to enhance model robustness. It can be expressed as follows:

$$\mathcal{L}_{\text{InfoNCE}} = -\log \frac{\exp(\boldsymbol{q} \cdot \boldsymbol{k}_+ / t)}{\exp(\boldsymbol{q} \cdot \boldsymbol{k}_+ / t) + \sum_{i=1}^{n} \exp(\boldsymbol{q} \cdot \boldsymbol{k}_i / t)} \tag{7.2}$$

where t is a temperature hyper-parameter [224], the query vector $\boldsymbol{q}$ is the encoded vector representation, $\boldsymbol{k}_+$ is a similar key vector that $\boldsymbol{q}$ matches, $\boldsymbol{K} = \{\boldsymbol{k}_1, ..., \boldsymbol{k}_n\}$ is a set of dissimilar encoded vectors. InfoNCE tries to classify the query vector $\boldsymbol{q}$ into its similar sample $\boldsymbol{k}_+$ and pushes it away from dissimilar samples in the set $\boldsymbol{K}$. The similarity is measured by dot product $(\cdot)$ between two vectors. To obtain the query representation $\boldsymbol{q}$ and the similar key representation $\boldsymbol{k}_+$, inspired by the recent advance [225] on the image recognition, we adopt Momentum Contrast (MoCo) [225] for the encoding. Specifically, it introduces an extra encoder M' to get the key representation $\boldsymbol{k}_+$, which

can be expressed as follows:

$$\begin{aligned} \boldsymbol{q} &= \mathrm{LayerNorm}(\mathrm{M}(X)[0]) \\ \boldsymbol{k}_{+} &= \mathrm{LayerNorm}(\mathrm{M}'(X')[0]) \end{aligned} \tag{7.3}$$

where X and X' denote the original masked sequence and its mutated variant respectively. The index 0 denotes the position of "[CLS]" in the sequence, which can be considered as the aggregated sequence representation. The encoder M' is the same as the encoder M, but during the learning phase, it utilizes a momentum to update its learnt weights while the encoder M uses the gradient descent:

$$\theta_{\mathrm{M}'} \leftarrow m\theta_{\mathrm{M}'} + (1-m)\theta_{\mathrm{M}} \tag{7.4}$$

where $m \in [0, 1)$ is a momentum coefficient for scaling, $\theta_{\mathrm{M}'}$ and θ_{M} denote the learnt weights for model M' and M.

From Eq 7.3, we obtain the query representation $\boldsymbol{q}$ and the key representation $\boldsymbol{k}_{+}$, to compute the similarity with dissimilar vectors from $\boldsymbol{K}$, MoCo maintains a "*dynamic*" *queue* of length n. This queue stores the dissimilar keys from the previous batches. Specifically, during the learning phase, the current query $\boldsymbol{q}$ will calculate the similarity with all dissimilar vectors in this queue. Afterwards, the key vector $\boldsymbol{k}_{+}$ will be enqueued to *queue* to replace the oldest one and we take it as the dissimilar samples for the calculation of the next query. Hence, it is namely *dynamically updated*.

Finally, we add both loss values with the scaled factor to pre-train ContraBERT and this process is expressed as follows:

$$\mathcal{L}_{\mathrm{Loss}} = \mathcal{L}_{\mathrm{MLM}} + w\mathcal{L}_{\mathrm{InfoNCE}} \tag{7.5}$$

where w is the hyper-parameter to scale the weight for both pre-training tasks.

7.3.4 Fine-tuning

Once ContraBERT is further pre-trained from the original pre-trained model, we can utilize it to obtain the vector representation for a program. Furthermore, we can also transfer it to different downstream tasks during the fine-tuning phase. These downstream tasks can be roughly categorized into three groups: (1) retrieval tasks (e.g., clone detection [10, 226], code search [92, 93]); (2) classification tasks (e.g., defect-detection [9]); (3) generation tasks (e.g., code-to-code translation [24, 25], code-refinement [227] and source code summarization [178]). Since the output space may differ from the pre-trained space, similar to CodeBERT and GraphCodeBERT, we add the task-specific module and then fine-tune the completed network on the labeled data. Specifically, for retrieval tasks, we further train ContraBERT on a labeled dataset; for classification tasks, we add a multi-layer perceptron (MLP) to predict the probability for each class; for generation tasks, we add a Transformer-based decoder to generate the target sequence.

7.4 Experimental Setup

In experiments, we first evaluate the effectiveness of our approach (RQ1) in improving model robustness. Then we plot the feature space learnt by different pre-trained models for visualization to confirm the features are learnt better(RQ2). Finally, we conduct extensive experiments to demonstrate the robustness-enhanced models provide significant improvements on downstream tasks (RQ3-RQ4). The detailed research questions are described as follows:

- **RQ1:** What is the performance of different augmentation operators in enhancing the robustness of the pre-trained model?
- **RQ2:** Can ContraBERT reshape the vector space learnt from the pre-trained models to obtain better vector representations?
- **RQ3:** Can ContraBERT outperform the original pre-trained models on different downstream tasks?

- **RQ4:** Are the defined pre-training tasks both effective in improving the downstream task performance?

7.4.1 Evaluation Tasks, Datasets and Baselines

We select four downstream tasks for evaluation. They are clone detection [10, 11], code search [93], defect detection [9] and code translation [24, 25]. We briefly introduce each task as follows:

Clone Detection (Code-Code Retrieval). This task is to identify semantically equivalent programs from a set of distractors by measuring the semantic similarity between two programs. AI for clone detection calculates cosine similarity between two embedding vectors of programs produced by neural networks and selects the top-k most similar programs as the candidates.

Defect Detection (Code Classification). It aims to detect whether a function contains defects that will be exploited to attack the software systems. Since the defects in a program are still difficult to be effectively detected by the traditional techniques, recently advanced works [9, 56, 228] propose to employ a deep neural network to learn program semantics to facilitate the detection. These AI-based techniques predict the probability of whether a function is vulnerable or not.

Code Translation (Code-Code Generation). It aims to translate a program in a programming language (e.g., Java) to the semantically equivalent one in another language (e.g., C#). Some previous works [25] analogy it to machine translation [5, 16] in NLP and employ LSTMs [15] and Transformer [16] for code translation.

Code Search (Text-Code Retrieval). It aims at returning the desired programs based on the query in a natural language. Similar to clone detection, it measures the semantic relevance between queries and programs. The input for the deep code search system [92, 93] is a natural language query and the output is programs that meet the query requirements. The cosine similarity is used to compute semantic similarity between the vectors of a query and programs.

In terms of the pre-training dataset, we use the released dataset provided by CodeSearchNet [93] and this dataset is also used by CodeBERT and GraphCodeBERT. We use bimodal NL-PL pairs for pre-training, which consist of six programming languages including Java, Python, Ruby, Go, PHP and JavaScript. For the fine-tuning datasets, for the tasks of clone detection (POJ-104), defect detection, and code translation, we directly utilize the released task-specific dataset provided by CodeXGLUE [82]. For code search, we use the cleaned dataset provided by GraphCodeBERT [18] for evaluation. For each task, we utilize the official scripts to make a fair comparison. In addition, by the defined augmentation operators in Section 7.3.2, we obtain a large amount of extra data (C', W') used in *Devign* as compared to the original pre-training data used in CodeBERT and GraphCodeBERT. Hence, we further add two baselines CodeBERT_Intr and GraphCodeBERT_Intr, which utilize original data as well as the dataset of the extra data (C', W') to pre-train CodeBERT and GraphCodeBERT with MLM for comparison.

7.4.2 Evaluation Metrics

In ContraBERT, different metrics are used to evaluate downstream tasks. We follow the metrics that CodeXGLUE used for evaluation, and the details are listed below:

MAP@R. It is the abbreviation of the mean of average precision, which is used to evaluate the result of retrieving R most similar samples in a set given a query. MAP@R is used for clone detection, where R is set to 499 for evaluation.

Acc. It defines the ratio of correct predictions (i.e., the exact match) in the testset. Acc is used for the evaluation of defect detection and code translation.

BLEU-4. It is widely used to evaluate the text similarity between the generated sequence with the ground-truth in the generation systems. We use BLEU-4 for code translation.

MRR. It is the abbreviation of Mean Reciprocal Rank, which is widely adopted in information retrieval systems [92, 229]. We used it to evaluate the performance of code

search. Instead of retrieving 1,000 candidates like CodeBERT [17], we follow the settings of GraphCodeBERT [18] to retrieve the answer for each query from the whole test set.

7.4.3 Experimental Settings

We adopt CodeBERT and GraphCodeBERT as our original models. We set the maximum input sequence length X and the mutated sequence X' as 512 following CodeBERT. We use Adam for optimizing with 256 batch size and 2e-4 learning rate. At each iteration, X' is constructed by C' and W', which are randomly picked from S_C and S_W respectively. Following He et al. [225], the momentum coefficient m, temperature parameter t and *queue* size is set to 0.999, 0.07 and 65536 accordingly. We set the weight w in Eq 7.5 as 0.5 to accelerate the coverage process. The model is trained on a DGX machine with 4 NVIDIA Tesla V100 with 32GB memory. To alleviate the bias towards the high-resource languages (i.e., the number of samples for different programming languages is different), we refer to GraphCodeBERT [18] and sample each batch from the same programming language according to a multinomial distribution with probabilities $\{q_i\}_{i=1...N}$.

$$q_i = \frac{p_i^\alpha}{\sum_{j=1}^{N} p_j^\alpha} \; with \;\; p_i = \frac{n_i}{\sum_{k=1}^{N} n_k} \tag{7.6}$$

where n_i is the number of samples for i-th programming language, N is the total number of languages and α is set to 0.7. The model is trained with 50K steps to ensure each mutated sample is utilized for the learning process and it takes about 2 days to finish the pre-training process. At fine-tuning phase, we directly utilize the default settings of CodeXGLUE [82] and GraphCodeBERT [18] in ContraBERT for downstream tasks. All experiments of downstream tasks are conducted on Intel Xeon Silver 4214 Processor with 6 NVIDIA Quadro RTX 8000 with 48GB memory.

TABLE 7.1: Results of ContraBERT against the variable renaming operator in a zero-shot manner.

Model	Num	N=0 Acc	N=1 Acc	N=4 Acc	N=8 Acc	Model	Num	N=0 Acc	N=1 Acc	N=4 Acc	N=8 Acc
ContraBERT_C w/o RFN	10,087	1	0.977	0.868	0.634	ContraBERT_G w/o RFN	10,375	1	0.975	0.873	0.634
ContraBERT_C w/o RV	8,665	1	0.932	0.597	0.291	ContraBERT_G w/o RV	9,042	1	0.955	0.657	0.309
ContraBERT_C w/o IDC	9,997	1	0.969	0.865	0.618	ContraBERT_G w/o IDC	10,530	1	0.963	0.862	0.612
ContraBERT_C w/o RO	9,923	1	0.963	0.857	0.619	ContraBERT_G w/o RO	10,509	1	0.968	0.868	0.617
ContraBERT_C w/o SP	10,604	1	0.959	0.849	0.616	ContraBERT_G w/o SP	11,140	1	0.969	0.860	0.613
ContraBERT_C w/o Trans	9,536	1	0.971	0.856	0.621	ContraBERT_G w/o Trans	10,360	1	0.973	0.859	0.617
ContraBERT_C w/o Delete	10,199	1	0.978	0.871	0.639	ContraBERT_G w/o Delete	10,376	1	0.981	0.878	0.643
ContraBERT_C w/o Switch	9,809	1	0.975	0.877	0.637	ContraBERT_G w/o Switch	10,457	1	0.978	0.876	0.647
ContraBERT_C w/o Copy	10,749	1	0.977	0.874	0.635	ContraBERT_G w/o Copy	10,859	1	0.981	0.880	0.641
ContraBERT_C	10,463	1	**0.981**	**0.882**	**0.649**	ContraBERT_G	10,565	1	**0.985**	**0.888**	**0.654**

7.5 Experimental Results

7.5.1 RQ1: Robustness Enhancement.

We investigate the augmentation operators in enhancing model robustness by validating the accuracy of samples against adversarial attacks on clone detection (POJ-104). The main reason to choose clone detection is that it targets identifying the semantically equivalent samples from other distractors. Hence, although the variable renaming operator changes the text of a program, the original program semantics are still unchanged. We statistically analyse the correctly predicted results under a different number of renaming edits for illustration. The experiments are conducted in a zero-shot manner [230], which means that it does not involve fine-tuning phase and we directly utilize the pre-trained model for evaluation. Specifically, we remove one operator and keep the remaining operators in Section 7.3.2 to pre-train the model. For fairness, the other settings in the experiments are the same as ContraBERT. Then we utilize the testset (in total 12,000 samples) on clone detection (POJ-104) and randomly mutate the variables contained in the correctly predicted samples produced by different pre-trained models from 1 to 8 edits to test the prediction accuracy. The experimental results are shown in Table 7.1 where N is the number of edits and Num is the total number of correctly predicted samples without any edits in the testset for different models. ContraBERT_C/G defines the model is initialized by CodeBERT and GraphCodeBERT respectively and w/o $*$ defines the removed operator $*$.

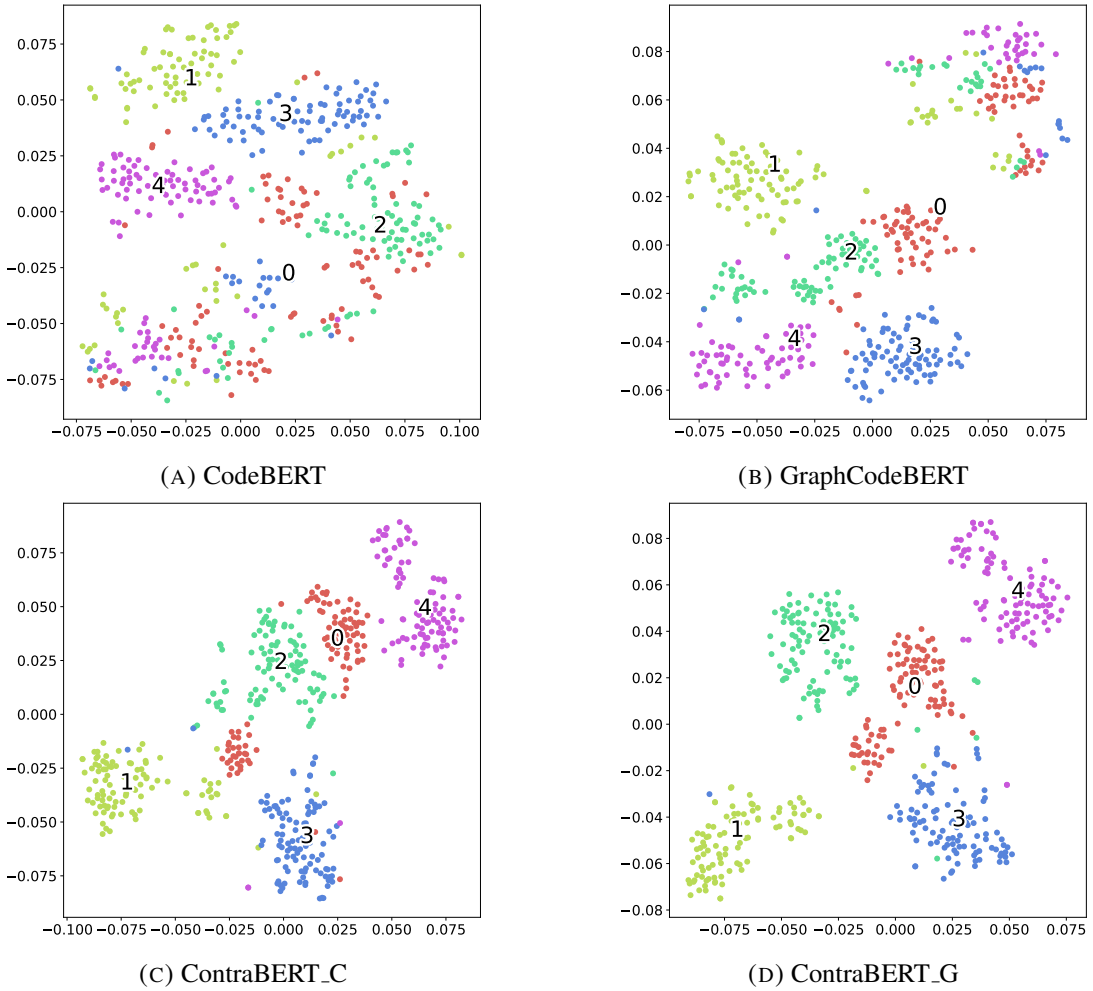
From Table 7.1, we find that in general, with the increasing number of edits, the performance continues to drop. It is reasonable, as the increasing number of edits, the difficulty for corrected predictions also increased. We also observe that each augmented

operator is beneficial to improve model robustness against the adversarial samples and when incorporating all operators, we obtain the best performance. It demonstrates the effectiveness of our designed PL-NL augmentation operators. In terms of NL augmentation operators, the operators Delete/Switch/Copy are relatively weaker in the robustness enhancement compared with the operator Trans. Since the operators (Delete/Switch/-Copy) just have a limited extent of modification on the original sequence (i.e., only one or two words are modified), the text similarity between W and W' is more similar than the operator Trans produces. Hence, the data diversity is limited by Delete/Switch/-Copy, which leads to the robustness improvement is not as obvious as the operator Trans. In terms of PL augmentation operators, we find that the number of correctly predicted samples of ContraBERT_C/G w/o RV is the lowest (e.g., 8,665 and 9,042). With the increasing number of edits, the accuracy drops by a great margin. This indicates that RV operator plays a critical role against adversarial attacks and removing it harms the performance significantly. In addition, removing RFN operator, ContraBERT also has higher accuracy than other PL operators (i.e., RV, IDC, RO and SP), which indicates that RFN has fewer contributions. It is caused by the generated program C' by RFN (i.e., rename function name) is more similar to the original program C compared with other PL augmentation operators.

✍ ►**RQ1**◄ Each operator in the designed PL-NL augmentation is effective in improving model robustness and when incorporating them, the robustness of pre-trained models is further enhanced.

7.5.2 RQ2: Visualization for Code Embeddings.

We visualize the code representation space learnt by different pre-trained models to confirm that the contrastive pre-training task can reshape the learnt vector space to ensure the model is more robust. Specifically, we use the clone detection (POJ-104) task provided by CodeXGLUE [82] for evaluation. The main reason for selecting clone detection is that it is more intuitive to observe and validate the similarity of code representation on the semantic equivalence programs. The dataset consists of 104 programming problems, where each problem has 500 semantically equivalent programs with different implementations. Theoretically, the program semantics for one problem should be the



(A) CodeBERT

(B) GraphCodeBERT

(C) ContraBERT_C

(D) ContraBERT_G

FIGURE 7.4: Visualization for vector representations of each 100 programs for 5 problems and they are randomly picked from clone detection (POJ-104). The vectors are produced by CodeBERT, GraphCodeBERT, ContraBERT_C and ContraBERT_G. The point with different colours indicates different problems that this function belongs to.

same. Hence, the code vectors (i.e., representations) of programs from pre-train models for one problem should be closer than the code vectors of programs for other problems. We randomly select 5 different problems with 100 samples and take them as the inputs to CodeBERT, GraphCodeBERT, ContraBERT_C and ContraBERT_G for visualization where C/G defines ContraBERT is initialized by CodeBERT or GraphCodeBERT respectively. We utilize the vector of the token "[CLS]" as the program representation. We further utilize T-SNE [231] to reduce the vector dimension to a two-dimensional space for visualization. Similar to Section 7.5.1, this process is also zero-shot [230], which helps us to validate the learnt space by different pre-training techniques.

As shown in Figure 7.4, the vectors produced by GraphCodeBERT (See Figure. 7.4b)

have a certain ability to group some problems of programs compared with CodeBERT (See Figure. 7.4a), which indicates that incorporating program structures such as data flow graph into pre-training is beneficial for the model to learn program semantics. However, we also find that the improvement is limited and the boundary in Figure 7.4b is not clear. Some data points are scattered, especially in the upper-right part of Figure 7.4b. In contrast, the visualization of ContraBERT is shown in Figure 7.4c and Figure 7.4d. We see that the programs in the same problem aggregate together closely as a cluster and different clusters have much clearer boundaries. This indicates that ContraBERT is more powerful than CodeBERT/GraphCodeBERT to group semantically equivalent data and push away dissimilar data. We attribute this ability to the defined PL-NL augmentation operators to capture the essence of programs. Furthermore, ContraBERT_G (See Figure 7.4d) has a better clustering performance than ContraBERT_C (See Figure 7.4c). For example in Figure 7.4c, the label 0 has two distant clusters while in Figure 7.4d, it only has one cluster. The improvements are from the used original model that GraphCodeBERT is superior to CodeBERT. In addition, we compute the distortion distance [1] [232] of the selected samples for these models to strengthen the conclusion. The distances of CodeBERT, GraphCodeBERT, ContraBERT_C and ContraBERT_G are 0.333, 0.212, 0.202, and 0.194 respectively. We can find that ContraBERT has a lower distortion distance than CodeBERT and GraphCodeBERT, which demonstrates their clusters are more compact.

✍ ▶**RQ2**◀ Through contrastive pre-training tasks to learn augmented variants constructed by a set of PL-NL operators, ContraBERT is able to group the semantically equivalent samples and push away the dissimilar samples, thus learning better vector representations.

7.5.3 RQ3: Performance of ContraBERT on Downstream Tasks.

We conduct extensive experiments on four downstream tasks to evaluate the performance of ContraBERT as compared to the original CodeBERT and GraphCodeBERT. We further add two baselines (i.e., CodeBERT_Intr and GraphCodeBERT_Intr), which

[1]The distortion distance refers to the sum of the squared distances of each sample to their assigned cluster centre.

TABLE 7.2: Results on clone detection and defect detection.

Model	Clone Detection MAP@R	Defect Detection Acc
CodeBERT	84.29	62.08
CodeBERT_Intr	86.34	62.41
ContraBERT_C (MLM)	86.21	62.25
ContraBERT_C (Contra)	81.44	62.22
ContraBERT_C	**90.46**	**64.17**
GraphCodeBERT	85.16	62.85
GraphCodeBERT_Intr	87.60	62.26
ContraBERT_G (MLM)	87.30	62.01
ContraBERT_G (Contra)	85.63	58.82
ContraBERT_G	90.06	63.32

TABLE 7.3: Results on code translation.

Model	Code Translation			
	Java → C#		C# → Java	
	BLEU-4	Acc	BLEU-4	Acc
CodeBERT	79.92	59.00	72.14	58.00
CodeBERT_Intr	79.93	59.20	75.71	58.60
ContraBERT_C (MLM)	79.90	59.10	75.03	58.10
ContraBERT_C (Contra)	51.99	34.60	46.75	38.30
ContraBERT_C	79.95	59.00	75.92	59.60
GraphCodeBERT	80.58	59.40	72.64	58.80
GraphCodeBERT_Intr	80.61	59.60	75.50	60.10
ContraBERT_G (MLM)	80.36	59.40	75.10	60.00
ContraBERT_G (Contra)	55.48	39.40	48.92	39.00
ContraBERT_G	**80.78**	**59.90**	**76.24**	**60.50**

TABLE 7.4: Results on code search where the evaluation metric is MRR.

Model	Ruby	Javascript	Go	Python	Java	PHP	Overall
CodeBERT	0.679	0.620	0.882	0.672	0.676	0.628	0.693
CodeBERT_Intr	0.686	0.623	0.883	0.676	0.678	0.630	0.696
ContraBERT_C (MLM)	0.675	0.621	0.888	0.670	0.675	0.631	0.693
ContraBERT_C (Contra)	0.593	0.532	0.864	0.622	0.618	0.584	0.636
ContraBERT_C	0.688	0.626	0.892	0.678	0.685	0.634	0.701
GraphCodeBERT	0.703	0.644	0.897	0.692	0.691	**0.649**	0.713
GraphCodeBERT_Intr	0.709	0.647	0.894	0.692	0.693	0.647	0.714
ContraBERT_G (MLM)	0.692	0.642	0.897	0.690	0.690	0.647	0.710
ContraBERT_G (Contra)	0.626	0.582	0.882	0.655	0.659	0.613	0.670
ContraBERT_G	**0.723**	**0.656**	**0.899**	**0.695**	**0.695**	0.648	**0.719**

are pre-trained by original data as well as the augmented variants. We supplement these two baselines to ensure the used scale of data is consistent with ContraBERT for a fair comparison. The results of clone/defect detection are shown in Table 7.2. Table 7.3 presents the results of code translation and Table 7.4 presents the results of code search where the rightmost "overall" column is the average value for six programming languages. Because the values for clone detection and defect detection of GraphCodeBERT are not reported by their original paper [18], we use official code for reproduction and report these values. The other values of CodeBERT and GraphCodeBERT are directly taken from CodeXGLUE [82] and Guo et al. [18].

From Table 7.2 and Table 7.3, we find that ContraBERT_C/G outperforms original pre-trained models CodeBERT or GraphCodeBERT on clone detection (POJ-104), defect detection and code translation. However, the absolute gains on code search (see Table 7.4) are minor. For these improvements, we attribute to the robustness-enhanced models providing better performance on downstream tasks. When it comes to minor improvements in code search, we ascribe to the difficulty of this task. Code search requires learning the semantic mapping between query and program. However, the semantic gap between programs and natural languages is huge. It makes the model difficult to achieve significant improvements. In total, considering the scale of test-set on code search, which contains 52,561 samples for six programming languages, the improvements are still promising. Furthermore, we find that compared with CodeBERT and GraphCodeBERT, CodeBERT_Intr and GraphCodeBERT_Intr have better performance on these tasks. It is reasonable since we add extra data to further pre-train CodeBERT and GraphCodeBERT. However, the performance of CodeBERT_Intr and GraphCodeBERT_Intr is worse than ContraBERT_C/G. It demonstrates that even with the same scale of data, ContraBERT_C/G are still better than CodeBERT and GraphCodeBERT, which further strengthen our conclusion that the improvements are brought by our proposed approach rather than the gains brought by the increased scale of the data.

✍ ►**RQ3**◄ ContraBERT comprehensively improves the performance of original

CodeBERT and GraphCodeBERT on four downstream tasks, we attribute the improvements to the enhanced robustness of the model has better performance on these tasks.

7.5.4 RQ4: Ablation Study for Pre-training Tasks.

ContraBERT utilizes two pre-training tasks, the first one is MLM, which learns the token representations and the second one is the contrastive pre-training task, which improves the model robustness by InfoNCE loss function. We further investigate the impact of each pre-training strategy on downstream tasks. The experimental results are shown in Table 7.2, Table 7.3 and Table 7.4 respectively, where the row of MLM or Contra denotes the results obtained by purely using MLM or contrastive pre-training task. For a fair comparison, the other settings are the same when combining both pre-training tasks for pre-training.

We can observe that the performance of purely using contrastive pre-training tasks is worse than purely using MLM on these downstream tasks, especially on the task of code translation. It is acceptable since both pre-training tasks are excellent in different aspects. Specifically, MLM is designed by randomly masking some tokens in a sequence to help the model learn token representations. The learnt token representations are important for generation tasks to generate a target sequence such as code translation, so it will help the model achieve good performance on these tasks. However, the contrastive pre-training task is designed by grouping the semantically equivalent samples while pushing away the dissimilar samples through InfoNCE loss function. The model robustness is enhanced by the contrastive pre-training task. Furthermore, when combing both pre-training tasks, our model achieves better performance compared with purely using one of the pre-training tasks, which indicates that ContraBERT is robust at the same time is able to achieve better performance on the downstream tasks.

✍ ►**RQ4**◄ Masked language modeling (MLM) and the contrastive pre-training task play different roles for ContraBERT. When combining them together, the model achieves higher performance on different downstream tasks.

7.6 Discussion

In this section, we first discuss the implications of our work, then discuss the limitations followed by discussing the threats to validity.

7.6.1 Implications

In this work, we find that the widely used pre-trained code models such as CodeBERT [17] or GraphCodeBERT [18] are not robust to adversarial attacks. Based on this finding, we further propose a contrastive learning-based approach for improvement. We believe that this finding in our paper will inspire the following-up researchers when designing a new model architecture for code, considering some other problems in the model such as robustness, generalization and not just focusing on the accuracy of the model on different tasks.

7.6.2 Limitations

By our experimental results, we find that the robustness of the model is enhanced significantly compared with the original models. We attribute it to the contrastive pre-training task to learn the semantically equivalent samples. However, these robustness-enhanced models only have slight improvements on the downstream task of code search. For this task, since it requires learning the semantic mapping between a query and its corresponding code, the designed augmentation operators just modify the code or query itself, hence their correlations are not captured and this leads to the improvements being limited. For code search, a possible solution to further improve the performance is to build the token relations between PL and NL for augmented variants, however, it involves intensive work to analyse the relations between the program and natural language comment. We will explore it in our future work.

Another limitation is the designed augmentation operators for PL and NL. We just design some basic operators to transform programs and comments. These operators are straightforward, although they are confirmed their effectiveness in improving model robustness. It is intriguing to explore more complex augmentation strategies such as

multiple operations on these operators for a sample to construct complex augmented variants.

7.6.3 Threats to Validity

Internal validity: The first threat is the hyper-parameter tuning for pre-training. More hyper-parameters need to tune than CodeBERT or GraphCodeBERT for example the temperature t, the momentum coefficient m and *queue* size. We follow the original settings from MoCo [225] and these parameters may not be optimal as they are designed for the task of image classification in computer vision. Due to that, the pre-training process is time-consuming and resource-consuming. We need nearly 2 days to complete one training process hence we ignore the hyper-parameter tuning process. However, we also find that even with the original parameters used in MoCo [225], ContraBERT still achieves higher performance than the original models. The second threat is that we use the same train-validation-test split that CodeXGLUE [82] and GraphCodeBERT [18] used. Adjusting the data split ratio or improving the training data quality may produce better results, however, we do not take these strategies to ensure a fair evaluation. The third threat is that we just use clone detection(POJ-104) to verify the robustness of the model is enhanced in Figure 7.1 and Section 7.5.1, we also plot the learnt space in Section 7.5.2. The reason to select clone detection is that it aims at identifying the semantically equivalent programs from other distractors, which is suitable for the evaluation.

External validity: Some other pre-training works in the code scenario such as CuBERT [109] are not included for evaluation. CuBERT was pre-trained on a large Python corpus with MLM. Our approach is orthogonal to these pre-trained models and we just need to replace the encoder M with other existing pre-trained models for evaluation.

7.7 Related Work

In this section, we briefly introduce the related works on contrastive learning, the pre-trained models for "big code" and the adversarial robustness of models of code.

7.7.1 Contrastive Learning

Contrastive learning is to learn representations by minimizing the distance between similar samples while maximizing the distance between different samples to help the similar samples closer to each other and different samples far apart from each other. Over the past few years, it has attracted increasing attention with many successful applications in computer vision [225, 233–236], natural language processing [237–240]. Recently, there are some works [84, 220, 241, 242] that utilize contrastive learning for different software engineering tasks. For example, Bui et al. [84] proposed Corder, a contrastive learning approach for code-to-code retrieval, text-to-code retrieval and code-to-text summarization. VarCLR [241] aimed to learn the semantic representations of variable names based on contrastive learning for different downstream tasks such as variable similarity scoring and variable spelling error correction. ContraCode [220] generated variants by a source-to-source compiler on JavaScript and further combined these generated mutated samples with contrastive learning for the task of clone detection, type inference and code summarization. Compared with these existing works which only focus on designing mutated variants for code, we first illustrate the widely concerned CodeBERT and GraphCodeBERT are weak to the adversarial examples. Then we design a set of simple and complex augmented operators on both programs and natural language sequences to obtain different variants. By contrastive learning to learn semantically equivalent variants, the robustness of existing pre-trained models is enhanced. We further confirm that the robustness-enhanced models provide improvements on different downstream tasks.

7.7.2 Pre-trained Models for "Big Code"

Recently, pre-trained models are widely applied to the "big code" era [17, 18, 82, 83, 109–113, 116, 220]. For example, Kanade et al. [109] pre-trained CuBERT based on BERT [216] with a massive corpus of Python programs from GitHub and then fine-tuned it for some classification tasks such as variable misuse classification. Feng et al. [17] proposed CodeBERT, a bimodal pre-trained model for programming language (PL) and natural language (NL) that learns the program representation to support code search and source code summarization. GraphcodeBERT [18] combines the variable

data-flow graph in a program with the code sequences and the natural language sequence to enhance CodeBERT. CodeXGLUE [82] also utilized CodeBERT and CodeGPT [114] to release a benchmark including several software engineering tasks. Liu et al. [116] proposed a CommitBART to support commit-related downstream tasks. Compared with existing pre-trained models, we illustrate they are not robust and further propose ContraBERT to enhance model robustness.

7.7.3 Adversarial Robustness on Models of Code

The research about adversarial robustness analysis on the models of code has attracted the attention [218,243–247]. Generally, these works can be categorized into two groups: white-box and black-box manner, where the white-box means that the approach provides some explanations on the decision-making while the black-box mainly focuses on the statistical evaluation. In terms of white-box works, Yefet et al. [244] proposed DAMP to select the semantic preserving perturbations by deriving the output distribution of the model with the input. Srikant et al. [243] provided a general formulation of a perturbed program that models site locations and perturbation choices for each location. Then based on this formulation, they further proposed a set of first-order optimization algorithms for the solving. In terms of the black-box works, HMH [247] generated adversarial examples of the source code by conducting iterative identifier renaming and evaluated on source code functionality classification task. The latest work by Yang et al. [218] proposed ALERT to transform the inputs while preserving the optional semantics of original inputs by replacing the variables with the substitutes. Their experiments are conducted on the pre-trained models CodeBERT and GraphCodeBERT. Compared with ALERT, which only designed the rename variable operation, in this paper, apart from the rename variable operation, we further design eight augmented operators over PL-NL pairs. Furthermore, a newly designed model to solve the weakness of robustness is not involved in ALERT. In contrast, we propose our general network architecture that uses contrastive learning to enhance model robustness. The extensive experiments have confirmed that our approach enhances the robustness of existing pre-trained models. We also demonstrate that these robustness-enhanced models provide improvements on different downstream tasks.

7.8 Conclusion

In this chapter, we observe that state-of-the-art pre-trained models such as CodeBERT and GraphCodeBERT are not robust to adversarial attacks and a simple mutation operator (e.g., variable renaming) degrades their performance significantly. To address this problem, in this paper, we propose ContraBERT, a contrastive learning-based framework to enhance the robustness of existing pre-trained models by designing nine kinds of PL-NL augmented operators to group the semantically equivalent variants. Through extensive experiments, we have confirmed that the model's robustness is enhanced. Furthermore, we also illustrate that these robustness-enhanced models provide improvements on four downstream tasks.

Chapter 8

Conclusion and Future Work

In this chapter, we first summarize the research works that we have completed in this thesis and then briefly discuss the potential research directions in the future.

8.1 Summary

Deep learning techniques have achieved great success in a variety of leading-edge research and they have also been applied to various software engineering applications [17, 18, 121, 248–253, 253–256]. However, behind these applications, learning program semantics by deep neural networks is a key challenge, which has attracted widespread attention from academia and industry. In this thesis, we explore the way to incorporate program structures into neural networks to learn comprehensive program semantics for various software engineering scenarios.

First, we seek to confirm the utilization of the program structures as the input for the sequential-based models (i.e., LSTMs) is beneficial for these models to learn program semantics for commit message generation. Since the sequential-based models usually require sequential input, however, the structures parsed by the programs such as abstract syntax tree (AST) are structured data, hence we propose to extract AST paths between leaf nodes on an AST of the program to incorporate the program syntax information

as the input for bidirectional LSTMs to learn program semantics. The experimental results demonstrate that incorporating program structures into the sequential-based models could provide significant improvements compared with the sequential-based models, which only consider the program as a flat sequence for learning.

Second, a program can be represented by different kinds of structures such as AST, control flow graph (CFG), and data flow graph (DFG) according to different requirements. We propose a technique to combine different dimensions of program structures (i.e., AST, CFG, DFG and NCS) with gated graph neural networks (GGNN) to learn the comprehensive program semantics for software vulnerability identification. We further innovate a convolution module followed by the output of GGNN for learning the fine-grained vulnerability features. The extensive experiments on our collected dataset demonstrate that encoding various program structures is more beneficial than each type of program structure to learn the comprehensive program semantics and our well-designed model could achieve new state-of-the-art performance for vulnerability identification.

Third, based on the code property graph (CPG) consisting of AST, CFG and DFG, we propose a graph-based encoder-decoder model for source code summarization, which targets describing the functionality of a program in the natural language. Specifically, to improve the learning capacity of GNN-based model, we propose the global attention mechanism on any pair of nodes over a graph to mitigate GNNs that can only capture the local neighbourhood information. We further innovate a retrieval-based augmentation mechanism to improve the quality of the generated summary. With the experiments compared with the conventional GNNs, we confirm that the learning capacity of our proposed model is improved. Furthermore, we also outperform current existing approaches significantly.

Fourth, code search aims at producing the semantic-equivalent code snippet based on the natural language query, however, the semantic gap between the program and the query is difficult to bridge for current existing code search systems. We propose a graph-based framework to alleviate this challenge. Besides constructing the program graph based on AST, we further construct the dependency graph for the query and takes

both graphs as the inputs for two separate GNN-based encoders to learn the semantic mapping relations. An extensive experiment compared with the existing baselines has demonstrated that jointly incorporating the structures for both program and natural language to GNNs is facilitated to learn the semantic mappings.

At last, the pre-trained models in the code scenario have attracted widespread attention due to their superior performance, however, by our preliminary experiments, we find that these models are not robust and simple mutations could degrade their performance significantly. Inspired by this finding, we propose a technique based on contrastive learning to enhance the robustness of these pre-trained models. We further confirm that these robustness-enhanced models could provide significant improvements on the various downstream tasks compared with the initialized models.

8.2 Future Work

This thesis presents some techniques to incorporate program structures to learn program semantics in different software engineering scenarios. In the future, we plan to investigate the following research topics:

1. Currently, our works focus on the raw code snippets to explore the hidden program structures behind the text to learn program semantics with deep learning techniques, however, the intermediate representation (IR) or the binary program can be considered as highly abstract of the raw code without unnecessary information such as identifier names, function names. In the future, we plan to customize our techniques on low-level code to validate the generalization of our techniques.

2. Presently, most of the proposed models in the code scenario are black-box and the learning capacity of the proposed models mostly depends on their performance on the specific task. Hence, the explanation [257] and testing [258–266] for these code models are urgent research problems to solve. An understanding of these models about decision-making could ensure the model's security and certainty, which could greatly reduce the risks in some fields with high requirements for security and reliability. Hence, it is valuable and necessary for exploration.

3. Deep learning techniques depend on the massive amount of data to achieve promising results, however, the quality of the data is usually ignored by deep learning practitioners. In addition, data-centric AI has proved that the refinement of the original data could produce significant improvements against the improvements brought by AI algorithms, however, the data-centric AI for code is less touched. We can move a step further by developing a systematic way to measure the program quality used for deep learning in terms of syntax and semantics.

4. The widely concerned pre-trained models have demonstrated superiority on various software engineering tasks, however, they all rely on sufficient computation resources (e.g., CodeBERT [17] utilized 16 interconnected GPU cards for parallel training and it cost 600 minutes to train 1,000 batches), which may not be applicable for the academia. The limitation of hardware support may greatly hinder the development of AI for Software Engineering. We are planning to research the way of ensuring the performance of these pre-trained models with affordable computation costs.

5. The majority of current AI for software engineering works put more weight on the model design (i.e., seeking a more powerful model) for the solution, while some conventional software engineering techniques are usually ignored such as static analysis, dynamic analysis and symbolic execution. We believe these techniques are also powerful in some scenarios. Hence, we plan to combine both techniques (i.e., AI and SE) and make the best of both worlds to improve current AI systems in software engineering.

Appendix A

List of Publications

1. **Shangqing Liu**, Cuiyun Gao, Sen Chen, Nie Lun Yiu, Yang Liu, "ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking," IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2020.3038681.

2. Yaqin Zhou, **Shangqing Liu**, Jing Kai Siow, Xiaoning Du, Yang Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.

3. **Shangqing Liu**, Yu Chen, Xiaofei Xie, Jing Kai Siow, Yang Liu, "Retrieval-Augmented Generation for Code Summarization via Hybrid GNN," 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.

4. **Shangqing Liu**, Xiaofei Xie, Jing Kai Siow, Lei Ma, Guozhu Meng, Yang Liu, "GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search," IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2022.3233901.

5. **Shangqing Liu**, Bozhi Wu, Xiaofei Xie, Guozhu Meng, Yang Liu, "ContraBERT: Enhancing Code Pre-trained Models via Contrastive Learning," The 45th International Conference on Software Engineering.

6. **Shangqing Liu**, "A Unified Framework to Learn Program Semantics with Graph Neural Networks," The 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020, pp. 1364-1366.

7. Xueyang Li, **Shangqing Liu**, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, Yang Liu, "TransRepair: Context-aware Program Repair for Compilation Errors," The 37th IEEE/ACM International Conference on Automated Software Engineering.

8. Bozhi Wu, **Shangqing Liu**, Ruitao Feng, Xiaofei Xie, Jing Kai Siow, Shang-Wei Lin, "Enhancing Security Patch Identification by Capturing Structures in Commits," IEEE Transactions on Dependable and Secure Computing, doi: 10.1109/TDSC.2022.3192631..

9. Yaqin Zhou, Jing Kai Siow, Chenyu Wang, **Shangqing Liu**, Yang Liu, "SPI: Automated Identification of Security Patches via Commits," in ACM Trans. Softw. Eng. Methodol., vol. 31, pp. 13:1–13:27, doi: 10.1145/3468854.

10. Jing Kai Siow, **Shangqing Liu**, Xiaofei Xie, Guozhu Meng, Yang Liu, "Learning Program Semantics with Code Representations: An Empirical Study," The 29th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, Hawaii, March 15-18, 2022.

11. Wenhan Wang, Kechi Zhang, Ge Li, **Shangqing Liu**, Anran Li, Zhi Jin, Yang Liu, "A Tree-structured Transformer for Program Representation Learning," The 30th IEEE International Conference on Software Analysis, Evolution and Reengineering Macao SAR, China, March 21st-24th, 2023.

12. Xiang Chen, Yanzhou Mu, Yubin Qu, Chao Ni, Meng Liu, Tong He, **Shangqing Liu**, "Do different cross-project defect prediction methods identify the same defective modules?" J. Softw. Evol. Process., vol. 32, doi: 10.1002/smr.2234.

Bibliography

[1] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[2] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[4] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, 2015.

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[7] "The github blog," https://github.blog/2018-11-08-100m-repos/.

[8] 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," in *25th Annual Network and Distributed System Security Symposium (NDSS 2018)*, 2018.

[9] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[10] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

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

[12] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 643–653.

[13] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[14] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *arXiv preprint arXiv:1707.02275*, 2017.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

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

[19] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the 42nd International Conference on Software Engineering. IEEE*, 2020.

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

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[23] Y. Chen, L. Wu, and M. J. Zaki, "Reinforcement learning based graph-to-sequence model for natural question generation," *arXiv preprint arXiv:1908.04942*, 2019.

[24] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.

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

[26] 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.

[27] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*. PMLR, 2016, pp. 2091–2100.

[28] R. P. L. Buse and W. Weimer, "Automatically documenting program changes," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, 2010, pp. 33–42.

[29] L. F. Cortes-Coy, M. L. Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, 2014, pp. 275–284.

[30] M. L. Vásquez, L. F. Cortes-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, 2015, pp. 709–712.

[31] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On automatic summarization of what and why information in source code changes," in *40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016*, 2016, pp. 103–112.

[32] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo, "Mining version control system for automatically generating commit comment," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, 2017, pp. 414–423.

[33] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings*

of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, 2018, pp. 373–384.

[34] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 135–146.

[35] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*, 2017, pp. 287–292.

[36] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019, pp. 3975–3981.

[37] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du, and Y. Qian, "Generating commit messages from diffs using pointer-generator network," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 299–309.

[38] H. Wang, X. Xia, D. Lo, Q. He, X. Wang, and J. Grundy, "Context-aware retrieval-based deep commit message generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–30, 2021.

[39] W. Tao, Y. Wang, E. Shi, L. Du, S. Han, H. Zhang, D. Zhang, and W. Zhang, "A large-scale empirical study of commit message generation: models, datasets and evaluation," *Empirical Software Engineering*, vol. 27, no. 7, pp. 1–43, 2022.

[40] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," *Advances in neural information processing systems*, vol. 28, 2015.

[41] W. Zhang, Y. Feng, F. Meng, D. You, and Q. Liu, "Bridging the gap between training and inference for neural machine translation," *arXiv preprint arXiv:1906.02448*, 2019.

[42] D. A. Wheeler. (2017) Flawfinder. [Online]. Available: https://www.dwheeler.com/flawfinder/

[43] CERN. (2007) Cppcheck. [Online]. Available: http://cppcheck.sourceforge.net/

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

[45] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "{MVP}: Detecting vulnerabilities using patch-enhanced vulnerability signatures," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1165–1182.

[46] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[47] 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*, 2019, pp. 533–544.

[48] 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*, 2017, pp. 627–637.

[49] H. Li, T. Kim, M. Bat-Erdene, and H. Lee, "Software vulnerability detection using backward trace analysis and symbolic execution," in *2013 International Conference on Availability, Reliability and Security*. IEEE, 2013, pp. 446–454.

[50] D. A. Ramos and D. Engler, "{Under-Constrained} symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[51] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 1–8.

[52] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[53] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 529–540.

[54] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, 2010, pp. 1–8.

[55] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[56] 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.

[57] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

[58] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 2017, pp. 1298–1302.

[59] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *arXiv preprint arXiv:1708.02368*, 2017.

[60] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "Poster: Vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2539–2541.

[61] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[62] 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," *arXiv preprint arXiv:1406.1078*, 2014.

[63] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[64] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 101–110.

[65] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 23–32.

[66] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 13–22.

[67] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

[68] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 380–389.

[69] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 562–567.

[70] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *2010 acm/ieee 32nd international conference on software engineering*, vol. 2. IEEE, 2010, pp. 223–226.

[71] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th international conference on Software engineering*, 2014, pp. 390–401.

[72] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[73] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[74] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[75] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 200–210.

[76] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[77] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

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

[79] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

[80] Y. Wang, E. Shi, L. Du, X. Yang, Y. Hu, S. Han, H. Zhang, and D. Zhang, "Cocosum: Contextual code summarization with multi-relational graph neural network," *arXiv preprint arXiv:2107.01933*, 2021.

[81] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees," *arXiv preprint arXiv:2108.12987*, 2021.

[82] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[83] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[84] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 511–521.

[85] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 545–549.

[86] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 260–270.

[87] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 681–682.

[88] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Software Engineering*, vol. 17, no. 4, pp. 424–466, 2012.

[89] K. Krugler, "Krugle code search architecture," *Finding Source Code on the Web for Remix and Reuse*, pp. 103–120, 2013.

[90] C. Liu, X. Xia, D. Lo, Z. Liu, A. E. Hassan, and S. Li, "Codematcher: Searching code based on sequential semantics of important query words," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–37, 2021.

[91] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[92] 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.

[93] 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.

[94] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[95] 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.

[96] R. Haldar, L. Wu, J. Xiong, and J. Hockenmaier, "A multi-perspective architecture for semantic code search," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 8563–8568. [Online]. Available: https://www.aclweb.org/anthology/2020.acl-main.758

[97] Z. Yao, J. R. Peddamail, and H. Sun, "Coacor: code annotation for code retrieval with reinforcement learning," in *The World Wide Web Conference*, 2019, pp. 2203–2214.

[98] S. Fang, Y.-S. Tan, T. Zhang, and Y. Liu, "Self-attention networks for code search," *Information and Software Technology*, vol. 134, p. 106542, 2021.

[99] A. A. Ishtiaq, M. Hasan, M. Haque, M. Anjum, K. S. Mehrab, T. Muttaqueen, T. Hasan, A. Iqbal, and R. Shahriyar, "Bert2code: Can pretrained language models be leveraged for code search?" *arXiv preprint arXiv:2104.08017*, 2021.

[100] L. Du, X. Shi, Y. Wang, E. Shi, S. Han, and D. Zhang, "Is a single model enough? mucos: A multi-model ensemble learning for semantic code search," *arXiv preprint arXiv:2107.04773*, 2021.

[101] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 631–642.

[102] Q. Zhu, Z. Sun, X. Liang, Y. Xiong, and L. Zhang, "Ocor: an overlapping-aware code retriever," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 883–894.

[103] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. S. Yu, "Multi-modal attention network learning for semantic source code retrieval," *arXiv preprint arXiv:1909.13516*, 2019.

[104] Y. Shi, Y. Yin, Z. Wang, D. Lo, T. Zhang, X. Xia, Y. Zhao, and B. Xu, "How to better utilize code graphs in semantic code search?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 722–733.

[105] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association

for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[106] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[107] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[108] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv preprint arXiv:1910.10683*, 2019.

[109] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," 2019.

[110] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[111] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang *et al.*, "Exploring software naturalness through neural language models," *arXiv preprint arXiv:2006.12641*, 2020.

[112] R.-M. Karampatsis and C. Sutton, "Scelmo: Source code embeddings from language models," *arXiv preprint arXiv:2004.13214*, 2020.

[113] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[114] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[115] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[116] S. Liu, Y. Li, and Y. Liu, "Commitbart: A large pre-trained model for github commits," *arXiv preprint arXiv:2208.08100*, 2022.

[117] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[118] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," *arXiv preprint arXiv:1804.00823*, 2018.

[119] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference*, 2019, pp. 417–426.

[120] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[121] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, "Learning program semantics with code representations: An empirical study," *arXiv preprint arXiv:2203.11790*, 2022.

[122] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in neural information processing systems*, 2015, pp. 2224–2232.

[123] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[124] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 422–431.

[125] The data of atom. [Online]. Available: https://zenodo.org/record/4077754#.X4K2b5MzZTY

[126] S. Jiang, "Boosting neural commit message generation with code semantic analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1280–1282.

[127] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[128] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[129] Javaparser documentation. [Online]. Available: https://javaparser.org/

[130] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[131] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[132] pygments documentation. [Online]. Available: http://pygments.org/

[133] P. Niemeyer and J. Knudsen, *Learning Java*. " O'Reilly Media, Inc.", 2005.

[134] Exuberant ctags documentation. [Online]. Available: http://ctags.sourceforge.net/

[135] Natural language toolkit nltk documentation. [Online]. Available: https://www.nltk.org/

[136] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[137] 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 on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[138] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[139] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[140] R. C. Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2vec: Learning distributed representations of code changes," *arXiv preprint arXiv:1911.07605*, 2019.

[141] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 1073–1083.

[142] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 765–777.

[143] C. Zhang, Y. Li, H. Chen, X. Luo, M. Li, A. Q. Nguyen, and Y. Liu, "Biff: Practical binary fuzzing framework for programs of iot and mobile devices," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1161–1165.

[144] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 417–428.

[145] Z. Du, Y. Li, Y. Liu, B. Mao, L. Chen, J. Guo, Z. He, D. Mu, C. Pang, R. Yu *et al.*, "Windranger: A directed greybox fuzzer driven by deviation basic blocks," in *2022 IEEE/ACM 44st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022.

[146] Y. Li, G. Meng, J. Xu, C. Zhang, H. Chen, X. Xie, H. Wang, and Y. Liu, "Vall-nut: Principled anti-grey box-fuzzing," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 288–299.

[147] X. He, X. Xie, Y. Li, J. Sun, F. Li, W. Zou, Y. Liu, L. Yu, J. Zhou, W. Shi *et al.*, "Sofi: Reflection-augmented fuzzing for javascript engines," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2229–2242.

[148] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 529–540. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315311

[149] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, ser. MetriSec '10. New York, NY, USA: ACM, 2010, pp. 3:1–3:8. [Online]. Available: http://doi.acm.org/10.1145/1853919.1853923

[150] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.81

[151] "CWE List Version 3.1," "https://cwe.mitre.org/data/index.html", 2018.

[152] Juliet test suite. [Online]. Available: https://samate.nist.gov/SRD/around.php

[153] 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*. IEEE, 2014, pp. 590–604.

[154] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*, 2016, pp. 2702–2711.

[155] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," in *Advances in Neural Information Processing Systems*, 2018, pp. 4805–4815.

[156] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[157] 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*, 2019, pp. 60–71.

[158] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 914–919. [Online]. Available: http://doi.acm.org/10.1145/3106237.3117771

[159] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[160] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2016, pp. 1480–1489.

[161] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[162] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[163] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[164] "Structured neural summarization," *arXiv preprint arXiv:1811.01824*, 2018.

[165] L. Zhao and L. Akoglu, "Pairnorm: Tackling oversmoothing in gnns," *arXiv preprint arXiv:1909.12223*, 2019.

[166] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view." in *AAAI*, 2020, pp. 3438–3445.

[167] G. Li, M. Müller, A. K. Thabet, and B. Ghanem, "Deepgcns: Can gcns go as deep as cnns?" in *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 2019, pp. 9266–9275. [Online]. Available: https://doi.org/10.1109/ICCV.2019.00936

[168] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[169] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[170] J. Zhu, J. Li, M. Zhu, L. Qian, M. Zhang, and G. Zhou, "Modeling graph structure in transformer for better amr-to-text generation," *arXiv preprint arXiv:1909.00136*, 2019.

[171] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 2018.

[172] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[173] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[174] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[175] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 842–851.

[176] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *2011 26th*

IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011, pp. 524–527.

[177] G. A. Miller, *WordNet: An electronic lexical database*. MIT press, 1998.

[178] 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.

[179] U. Alon and E. Yahav, "On the bottleneck of graph neural networks and its practical implications," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=i80OPhOCVH2

[180] H. Li, S. Kim, and S. Chandra, "Neural code search evaluation dataset," *arXiv preprint arXiv:1908.09804*, 2019.

[181] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 344–354.

[182] M.-C. De Marneffe, B. MacCartney, C. D. Manning *et al.*, "Generating typed dependency parses from phrase structure parses." in *Lrec*, vol. 6, 2006, pp. 449–454.

[183] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.

[184] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International Conference on Learning Representations*, 2019.

[185] D. Su, Y. Xu, W. Dai, Z. Ji, T. Yu, and P. Fung, "Multi-hop question generation with graph convolutional network," *arXiv preprint arXiv:2010.09240*, 2020.

[186] Y. Chen, L. Wu, and M. J. Zaki, "Graphflow: Exploiting conversation flow with graph neural networks for conversational machine comprehension," *arXiv preprint arXiv:1908.00059*, 2019.

[187] L. Song, Z. Wang, M. Yu, Y. Zhang, R. Florian, and D. Gildea, "Exploring graph-structured passage representation for multi-hop reading comprehension with graph neural networks," *arXiv preprint arXiv:1809.02040*, 2018.

[188] N. De Cao, W. Aziz, and I. Titov, "Question answering by reasoning across documents with graph convolutional networks," *arXiv preprint arXiv:1808.09920*, 2018.

[189] K. Annervaz, S. B. R. Chowdhury, and A. Dukkipati, "Learning beyond datasets: Knowledge graph augmented neural networks for natural language processing," *arXiv preprint arXiv:1802.05930*, 2018.

[190] Q. Li, Z. Han, and X.-M. Wu, "Deeper insights into graph convolutional networks for semi-supervised learning," in *Thirty-Second AAAI conference on artificial intelligence*, 2018.

[191] K. Oono and T. Suzuki, "Graph neural networks exponentially lose expressive power for node classification," *arXiv preprint arXiv:1905.10947*, 2019.

[192] M. Cvitkovic, B. Singh, and A. Anandkumar, "Open vocabulary learning on source code with a graph-structured cache," in *International Conference on Machine Learning*. PMLR, 2019, pp. 1475–1485.

[193] D. Jurafsky and J. H. Martin, "Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition."

[194] V. Keselj, "Speech and language processing daniel jurafsky and james h. martin," 2009.

[195] M.-C. De Marneffe and C. D. Manning, "Stanford typed dependencies manual," Technical report, Stanford University, Tech. Rep., 2008.

[196] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[197] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of machine learning research*, vol. 12, no. ARTICLE, pp. 2493–2537, 2011.

[198] A. Frome, G. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, and T. Mikolov, "Devise: A deep visual-semantic embedding model," 2013.

[199] A. Rice. (2018) features-javac. [Online]. Available: https://github.com/acr31/features-javac/

[200] M. Honnibal and I. Montani, "spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing," 2017, to appear.

[201] C. Gormley and Z. Tong, *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015.

[202] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, and Z. Xu, "Two-stage attention-based model for code search with textual and structural features," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 342–353.

[203] M. Maybury, *Advances in automatic text summarization*. MIT press, 1999.

[204] Y. Liu and M. Lapata, "Text summarization with pretrained encoders," *arXiv preprint arXiv:1908.08345*, 2019.

[205] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 39.

[206] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *arXiv preprint arXiv:2105.12787*, 2021.

[207] 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.

[208] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 329–340.

[209] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 245–256.

[210] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[211] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.

[212] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, 2020.

[213] B. Wu, S. Liu, R. Feng, X. Xie, J. Siow, and S.-W. Lin, "Enhancing security patch identification by capturing structures in commits," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[214] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu, "Transrepair: Context-aware program repair for compilation errors," *arXiv preprint arXiv:2210.03986*, 2022.

[215] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[216] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[217] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[218] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," *arXiv preprint arXiv:2201.08698*, 2022.

[219] Authors, "Contrabert: Enhancing code pre-trained models via contrastive learning," https://sites.google.com/view/contrabert.

[220] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

[221] R. Sennrich, B. Haddow, and A. Birch, "Improving neural machine translation models with monolingual data," *arXiv preprint arXiv:1511.06709*, 2015.

[222] E. Ma, "Nlp augmentation," https://github.com/makcedward/nlpaug, 2019.

[223] A. v. d. Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *arXiv preprint arXiv:1807.03748*, 2018.

[224] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin, "Unsupervised feature learning via non-parametric instance discrimination," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 3733–3742.

[225] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 9729–9738.

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

[227] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[228] 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.

[229] J. Wang and J. Zhu, "Portfolio theory of information retrieval," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 2009, pp. 115–122.

[230] M. M. Palatucci, D. A. Pomerleau, G. E. Hinton, and T. Mitchell, "Zero-shot learning with semantic output codes," 2009.

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

[232] "K-means," https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[233] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International conference on machine learning*. PMLR, 2020, pp. 1597–1607.

[234] M. Kim, J. Tack, and S. J. Hwang, "Adversarial self-supervised contrastive learning," *arXiv preprint arXiv:2006.07589*, 2020.

[235] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," *arXiv preprint arXiv:2103.00020*, 2021.

[236] B. Dai and D. Lin, "Contrastive learning for image captioning," *arXiv preprint arXiv:1710.02534*, 2017.

[237] Z. Yang, Y. Cheng, Y. Liu, and M. Sun, "Reducing word omission errors in neural machine translation: A contrastive learning approach," 2019.

[238] H. Fang, S. Wang, M. Zhou, J. Ding, and P. Xie, "Cert: Contrastive self-supervised learning for language understanding," *arXiv preprint arXiv:2005.12766*, 2020.

[239] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," *arXiv preprint arXiv:2104.08821*, 2021.

[240] D. Shen, M. Zheng, Y. Shen, Y. Qu, and W. Chen, "A simple but tough-to-beat data augmentation approach for natural language understanding and generation," *arXiv preprint arXiv:2009.13818*, 2020.

[241] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," *arXiv preprint arXiv:2112.02650*, 2021.

[242] X. Wang, Q. Wu, H. Zhang, C. Lyu, X. Jiang, Z. Zheng, L. Lyu, and S. Hu, "Heloc: Hierarchical contrastive learning of source code representation," *arXiv preprint arXiv:2203.14285*, 2022.

[243] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U.-M. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," *arXiv preprint arXiv:2103.11882*, 2021.

[244] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[245] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," *arXiv preprint arXiv:2002.03043*, 2020.

[246] P. Bielik and M. Vechev, "Adversarial robustness for code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 896–907.

[247] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1169–1176.

[248] 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.

[249] Z. Li, X. Xie, H. Li, Z. Xu, Y. Li, and Y. Liu, "Cross-lingual transfer learning for statistical type inference," 2022.

[250] W. Wang, K. Zhang, G. Li, S. Liu, Z. Jin, and Y. Liu, "A tree-structured transformer for program representation learning," *arXiv preprint arXiv:2208.08643*, 2022.

[251] W. Ma, M. Zhao, X. Xie, Q. Hu, S. Liu, J. Zhang, W. Wang, and Y. Liu, "Is self-attention powerful to learn code syntax and semantics?" *arXiv preprint arXiv:2212.10017*, 2022.

[252] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 207–216.

[253] R. Feng, S. Chen, X. Xie, G. Meng, S.-W. Lin, and Y. Liu, "A performance-sensitive malware detection system using deep learning on mobile devices," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1563–1578, 2020.

[254] B. Wu, S. Chen, C. Gao, L. Fan, Y. Liu, W. Wen, and M. R. Lyu, "Why an android app is classified as malware: Toward malware classification interpretation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–29, 2021.

[255] R. Feng, J. Q. Lim, S. Chen, S.-W. Lin, and Y. Liu, "Seqmobile: An efficient sequence-based malware detection system using rnn on mobile devices," in *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2020, pp. 63–72.

[256] 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.

[257] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, "Towards robustness of deep program processing models—detection, estimation, and enhancement," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–40, 2022.

[258] X. Du, Y. Li, X. Xie, L. Ma, Y. Liu, and J. Zhao, "Marble: model-based robustness analysis of stateful deep learning systems," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 423–435.

[259] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.

[260] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li, "Diffchaser: Detecting disagreements for deep neural networks." International Joint Conferences on Artificial Intelligence Organization, 2019.

[261] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: Model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 477–487.

[262] ——, "A quantitative analysis framework for recurrent neural network," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1062–1065.

[263] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 120–131.

[264] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 100–111.

[265] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "Deepct: Tomographic combinatorial testing for deep learning systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 614–618.

[266] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1158–1161.