Towards Intelligent Reliable Code Retrieval based on Code Semantics Learning

GU, Wenchao

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
February 2024

Thesis Assessment Committee

Professor PAN Jialin (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor KING Kuo Chin Irwin (Committee Member)

Professor ZHANG Hongyu (External Examiner)

Abstract of thesis entitled:

Towards Intelligent Reliable Code Retrieval based on Code Semantics Learning
Submitted by GU, Wenchao
for the degree of Doctor of Philosophy
at The Chinese University of Hong Kong in February 2024

With the large-scale application of software in various industries, the demand for software development has snowballed in recent decades. Code retrieval, which can retrieve the users' desired code snippets from the code database according to their natural language description, can significantly reduce the workload of software developers. Therefore, code retrieval is an important research topic. However, the retrieved code snippets may be vulnerable and cannot be directly used. Vulnerability detection for the retrieved code snippets is necessary. In this thesis, we present our exploration of the task of code retrieval and software vulnerability detection. Specifically, we aim to address several common challenges in effective code retrieval, code retrieval acceleration, and software vulnerability detection from the following four parts.

Firstly, we study the problem of code retrieval. Considering the highly structured characteristic of source code, we propose a novel neural network model named CRaDLe. CRaDLe couples both structural and semantic information of code at the statement level, where the code structures are extracted based

on the program dependency graph. The evaluation results show that CRaDLe can significantly outperform the state-of-the-art baseline models.

Secondly, we shift to the problem of code retrieval efficiency. Current deep learning-based approaches need to rank all the source code snippets in the corpus during searching, which will incur a large amount of computational cost. To address this problem, we propose a novel approach named CoSHC. CoSHC clusters the representation vectors into different categories and generates binary hash codes for both source code and queries. During the retrieval, CoSHC will retrieve the different number of code candidates for the given query in each category. The evaluation results show that CoSHC can preserve most of the performance from the original models while significantly reducing the retrieval time.

Thirdly, we focus on how to improve the code retrieval efficiency further. Although it is very efficient to calculate the Hamming distance, these Hamming distance-based methods have to scan the whole database, which leads to a considerable expensive computation cost. To address this problem, we propose a hash table-based code retrieval framework CSSDH that achieves advanced performance by replacing the Hamming distance calculation with lookup hash tables. Experimental results indicate that CSSDH can significantly reduce the retrieval time of current state-of-the-art deep hashing approaches, retain comparable performance, or even outperform the previous deep hashing approaches in the recall step.

Fourthly, we shift to the problem of vulnerability detection. Previous deep learning-based approaches have struggled to achieve accurate vulnerability localization, as they do not prioritize the

localization problem during training. Automatically predicting statement-level vulnerabilities in a supervised manner poses difficulties, as it necessitates labeled data for model learning. To address this issue, we propose a novel approach named WILDE for function-level vulnerability detection with statement-level localization. WILDE can achieve the statement-level vulnerability localization without the statement-level labeled data. The extensive experimental findings showcase that WILDE achieves comparable performance in detecting vulnerabilities at the function level compared, and its ability to localize vulnerabilities surpasses that of the previous models.

論文題目　：基於代碼語義學習的智慧程式設計
作者　　：顧文超
學校　　：香港中文大學
學系　　：計算機科學與工程學系
修讀學位　：哲學博士
摘要　　：
隨著軟體在各行業的大規模應用，近幾十年來軟體發展的需求快速增長。代碼檢索是一種根據使用者的自然語言描述，從代碼資料庫中檢索出使用者想要的代碼片段的技術，該技術可以大大減輕軟體發展人員的工作量。因此，代碼檢索是智慧程式開發中一個非常重要的研究課題。另外，檢索到的代碼片段可能存在漏洞問題，無法直接使用，需要對檢索到的代碼片段進行漏洞檢測。在本文中，我們展示了對代碼檢索和軟體漏洞檢測任務的探索。具體來說，我們旨在從以下五個部分解決有效代碼檢索、代碼檢索加速和軟體漏洞檢測方面的幾個常見挑戰。

首先，我們研究代碼檢索問題。考慮到程式碼高度結構化的特點，以往的研究提出整合代碼的結構資訊，如抽象語法樹和控制流圖。然而，抽象語法樹和控制流圖在為語義學習提供代碼結構資訊方面都有其自身的局限性。為了解決這個問題，我們提出了一種名為 CRaDLe 的新型神經網路模型。CRaDLe 在語句級別耦合代碼的結構和語義資訊，其中基於程式依賴圖提取代碼結構。評估結果表明，CRaDLe 可以顯著優於最先進的基線模型。

其次，我們轉向代碼檢索效率問題。當前基於深度學習的方法

在搜索時需要對語料庫中的所有原始程式碼片段進行排序，這將產生大量的計算成本。為了解決這個問題，我們提出了一種名為 CoSHC 的新方法。CoSHC 將表示向量聚類為不同的類別，並為程式碼和查詢語句生成二進位雜湊碼。在檢索過程中，CoSHC 將根據 CoSHC 預測的概率，利用雜湊碼檢索每個類別中給定不同數量的候選代碼。評估結果表明，CoSHC 可以保留原始模型的大部分性能，同時顯著減少檢索時間。

第三，我們關注如何進一步提高代碼檢索效率。雖然計算漢明距離非常有效，但它必須掃描整個大型資料庫。為了解決這個問題，我們提出了一種基於雜湊的代碼檢索框架 CSSDH，該框架通過用查找雜湊表代替漢明距離計算來實現更高效率。實驗結果表明，CSSDH 可以顯著減少當前最先進的深度雜湊方法的檢索時間，並在召回步驟中保持可比的性能，甚至優於以前的深度雜湊方法。

第四，我們轉向漏洞檢測問題。以前基於深度學習的方法很難實現準確的漏洞定位，因為它們在訓練過程中沒有優先考慮定位問題。以監督方式自動預測語句級漏洞會帶來困難，因為它需要標記資料來進行模型學習。為了解決這個問題，我們提出了一種名為 WILDE 的新方法，用於具有語句級定位的函數級漏洞檢測。WILDE 無需語句級標記資料即可實現語句級漏洞定位。大量的實驗結果表明，WILDE 在功能級別的漏洞檢測方面取得了可比的性能，並且其定位漏洞的能力超越了以前的模型。

Acknowledgement

First and foremost, I would like to express my heartfelt gratitude to Prof. Michael R. Lyu, my dedicated supervisor at CUHK. From selecting my research topic to the meticulous craft of technical writing, his unwavering guidance and patience have greatly facilitated my progress in this challenging research endeavor. Throughout my Ph.D. studies, I have not only had the privilege of imbibing a wealth of knowledge from his exemplary approach but also learned his enthusiastic attitude to doing research.

I extend my sincere appreciation to the members of my thesis assessment committee, Prof. Sinno Jialin Pan and Prof. Iwrin King, for their invaluable feedback and constructive suggestions on both this thesis and all of my term presentations. I am equally grateful to Prof. Hongyu Zhang from Chongqing University, who graciously served as the external examiner for this thesis.

I wish to acknowledge Prof. Cuiyun Gao from the Harbin Institute of Technology (Shenzhen) and Prof. Yanling Wang from Sun Yat-Sen University. The insightful discussions I have had with them immensely helped me complete the research in this thesis.

My gratitude also goes out to my exceptional group mates, including Jen-Tse Huang, Yun Peng, Jianping Zhang, Jinyang Liu,

Yintong Huo, Wenxuan Wang, Yichen Li, Shuqing Li, Wenwei Gu, Yizhan Huang, Jiaoqiao Zhao, Ziyuan Hu, Shuyao Jiang, Zhihan Jiang, Jinxi Kuang, and Renyi Zhong.

Lastly, but certainly not least, I would like to thank my family. Their profound love and unwavering support have been the driving force behind my pursuit of a doctorate.

To my family.

Contents

Abstract i

Acknowledgement vi

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Thesis Contributions . . . 8
- 1.3 Thesis Organization . . . 11

2 Background Review 14
- 2.1 Neural Network Basic . . . 14
  - 2.1.1 Recurrent Neural Networks . . . 15
  - 2.1.2 Transformer and Pre-Training . . . 16
- 2.2 Code Retrieval . . . 18
  - 2.2.1 Non Deep Learning Based Approaches . . . 18
  - 2.2.2 Deep Learning Based Approaches . . . 19
- 2.3 Hashing . . . 21
  - 2.3.1 Hash table-based approaches . . . 21
  - 2.3.2 Supervised cross-modal hashing approaches 22
  - 2.3.3 Unsupervised cross-modal hashing approaches 22
- 2.4 Vulnerability Detection . . . 23
  - 2.4.1 Deep Learning-based Vulnerability Detection . . . 23

2.4.2 Deep Learning-based Statement-Level Vulnerability Detection and Localization . . . 25

3 Deep Code Retrieval Based on Semantic Dependency Learning 27

3.1 Introduction . . . 28
3.2 Methodology . . . 30
3.2.1 Overview . . . 31
3.2.2 Code Encoder . . . 31
3.2.3 Description Encoder . . . 37
3.2.4 Similarity Measurement . . . 38
3.2.5 Model Training . . . 38
3.3 Experimental Setup . . . 39
3.3.1 Dataset Collection . . . 39
3.3.2 Performance Measurement . . . 40
3.3.3 Implementation Details . . . 42
3.3.4 Baseline Models . . . 43
3.4 Experimental Results . . . 45
3.4.1 Main Results . . . 45
3.4.2 Parameter Analysis . . . 46
3.4.3 Ablation Study . . . 50
3.4.4 Case Studies . . . 52
3.4.5 Error Analysis . . . 53
3.5 Discussion . . . 55
3.5.1 Dependency Embedding Approach . . . 55
3.6 Threats to Validity . . . 56
3.6.1 Threats to External Validity . . . 56
3.6.2 Threats to Internal Validity . . . 57
3.7 Summary . . . 57

4 Accelerating Code Retrieval with Deep Hashing and Code Classification **58**
4.1 Introduction . . . 59
4.2 Methodology . . . 62
4.2.1 Offline Stage . . . 62
4.2.2 Online Stage . . . 68
4.3 Experiments . . . 69
4.3.1 Dataset . . . 69
4.3.2 Experimental Setup . . . 70
4.3.3 Baselines . . . 71
4.3.4 Evaluation Metric . . . 72
4.4 Experimental Results . . . 72
4.4.1 RQ1: How much faster is CoSHC than the original code retrieval models? . . . 72
4.4.2 RQ2: How does CoSHC affect the accuracy of the original models? . . . 75
4.4.3 RQ3: Can the classification module help improve performance? . . . 77
4.5 Discussion . . . 79
4.6 Threats to Validity . . . 80
4.6.1 Threats to External Validity . . . 81
4.6.2 Threats to Internal Validity . . . 81
4.7 Summary . . . 81

5 Accelerating Code Retrieval via Segmented Deep Hashing **83**
5.1 Introduction . . . 84
5.2 Method . . . 88
5.2.1 Overview . . . 88
5.2.2 Recall and Re-rank with Deep Hashing . . . 90

5.2.3 Initial Hashing Projection Training 91
5.2.4 Iteration Training Strategy 92
5.2.5 Hash Alignment 98
5.2.6 Inference of Binary Hash Codes 99
5.3 Experimental Settings 99
5.3.1 Datasets 99
5.3.2 Baselines 100
5.3.3 Metrics 102
5.3.4 Implementation Details 103
5.4 Evaluation 104
5.4.1 RQ1: What is the Efficiency of CSSDH? . 104
5.4.2 RQ2: What is the Effectiveness of CSSDH? 107
5.4.3 RQ3: What is the Effectiveness of Adaptive Bits Relaxing? 111
5.4.4 RQ4: How Many Error Bits Have Been Fixed? 113
5.5 Threats to Validity 115
5.5.1 Threats to External Validity 115
5.5.2 Threats to Internal Validity 116
5.6 Summary 116

6 Weakly Supverised Vulnerability Detection and Localization via Multiple Instance Learning 117
6.1 Introduction 118
6.2 Methodology 122
6.2.1 Overview 122
6.2.2 Code Encoding 123
6.2.3 The Design of Code Encoder 124
6.2.4 Multiple Instance Learning-Based Training Strategy 130

6.2.5 Model Inference . . . 133
6.3 Experimental Setup . . . 134
6.3.1 Data Pre-processing . . . 134
6.3.2 Implementation Details . . . 136
6.3.3 Baselines . . . 137
6.3.4 Evaluation Metrics . . . 139
6.4 Experimental Results . . . 142
6.4.1 Comparison on function-level vulnerability detection and statement-level vulnerability localization . . . 142
6.4.2 Impact of the top-k statement selection on the performance of WILDE . . . 146
6.4.3 Impact of different channels on the performance of WILDE . . . 149
6.4.4 The influence of the training data size to the performance of WILDE . . . 152
6.4.5 The detection ability of WILDE for different types of CWE vulnerabilities . . . 155
6.5 Threats to Validity . . . 157
6.5.1 Threats to External Validity . . . 157
6.5.2 Threats to Internal Validity . . . 157
6.6 Summary . . . 158

7 Conclusion and Future Work 160
7.1 Conclusion . . . 160
7.2 Future Directions . . . 162
7.2.1 Repository-Level Code Generation with Large Language Model . . . 162
7.2.2 Reliable Code Generation with Large Language Model . . . 163

7.2.3 Vulnerability Detection with Static Analysis and Large Language Model 164

8 Publications during Ph.D. Study **166**

Bibliography **168**

List of Figures

1.1 An illustration of code retrieval 2
1.2 The roadmap of this thesis. 7
1.3 The organization of this thesis. 11

2.1 Illustration of RNN. 15
2.2 Illustration of Transformer. 17

3.1 Overview of the proposed CRaDLe. 32
3.2 Overall framework of the proposed CRaDLe. . . . 32
3.3 Workflow for extracting PDG of the code snippet. 37
3.4 Parameter sensitivity study for CodeSearchNet. . 49
3.5 Parameter sensitivity study for Code2Seq. 50

4.1 Overview of the proposed CoSHC. 63
4.2 Architecture of the hashing module. 65

5.1 Illustration of recall and re-rank mechanism with previous deep hashing approaches. 88
5.2 Illustration of recall and re-rank mechanism with the combination of deep hashing approaches and CSSDH. 89
5.3 Overall framework of CSSDH. 90
5.4 Steps in the iteration training strategy. 93

6.1 An overview architecture of WILDE. 123

List of Tables

3.1 Statistics of the number of statements in CodeSearchNet dataset. 41
3.2 Statistics of the number of statements in Code2Seq dataset. 41
3.3 Statistics of the CodeSearchNet dataset. 42
3.4 Statistics of the Code2seq dataset. 43
3.5 Comparison results with baseline models on the CodeSearchNet dataset. 47
3.6 Comparison results with baseline models on the Code2seq dataset. 48
3.7 Ablation study on the CodeSearchNet dataset. . . 51
3.8 Ablation study on the Code2seq dataset. 51
3.9 Comparison results with our original models. . . . 56

4.1 Time Efficiency of CoSHC. 73
4.2 Results of code retrieval performance comparison. 74
4.3 Classification accuracy of the code classification module in each model. 77

5.1 Dataset statistics. 100
5.2 Results of time efficiency comparison on the recall step of different deep hashing approaches with different code retrieval models. 105

5.3 Results of overall performance comparison of different deep hashing approaches with different code retrieval models. 108
5.4 The comparisons among the six CSSDH variants with the baseline of CodeBERT. 110
5.5 The repair ratio of adaptive bits relaxing in both code hashing model and query hashing model. . . 113
5.6 Average hash bits that both code and query hashing models predicted as unknown in single hash code segment. 114

6.1 Statistics of dataset. 134
6.2 Comparison results on function-level vulnerability. 143
6.3 Comparison results on function-level vulnerability and statement-level vulnerability localization. 144
6.4 Results of the function-level vulnerability detection performance comparison with different Top-K selection. 147
6.5 Results of the statement-level vulnerability localization performance comparison with different Top-K selection. 148
6.6 Results of the function-level vulnerability detection performance comparison with different channels. 150
6.7 Results of the statement-level vulnerability localization performance comparison with different channels. 151
6.8 Comparison results on function-level vulnerability detection and statement-level vulnerability localization with different sizes of training data. . . 153

6.9 Detection results for different CWE vulnerabilities with our proposed WILDE. 156

Chapter 1

Introduction

1.1 Overview

In an increasingly information-driven and digital world, software plays an increasingly important role in current human production and economic activities. The current global software market size is around 583.47 billion U.S. dollars in 2022, and the compound annual growth rate of this market from 2023 to 2030 is expected to be 11.5%[1]. The market's demand for software developers is also expected to increase rapidly. According to the U.S. Bureau of Labor Statistics, the demand for software developers will be expected to increase by 25% from 2021 to 2031, which is 20% higher than the average growth of all occupations[2]. As the market demand for programmers increases day by day, the cost of hiring programmers is also rising. For instance, the median hourly wage of software developers in the U.S. is 61.18 dollars, 2.75 times the median salary for all occupations[3]. To reduce the workload of software engineers and

[1]https://www.grandviewresearch.com/industry-analysis/software-market-report

[2]https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

[3]https://www.bls.gov/oes/current/oes_nat.htm

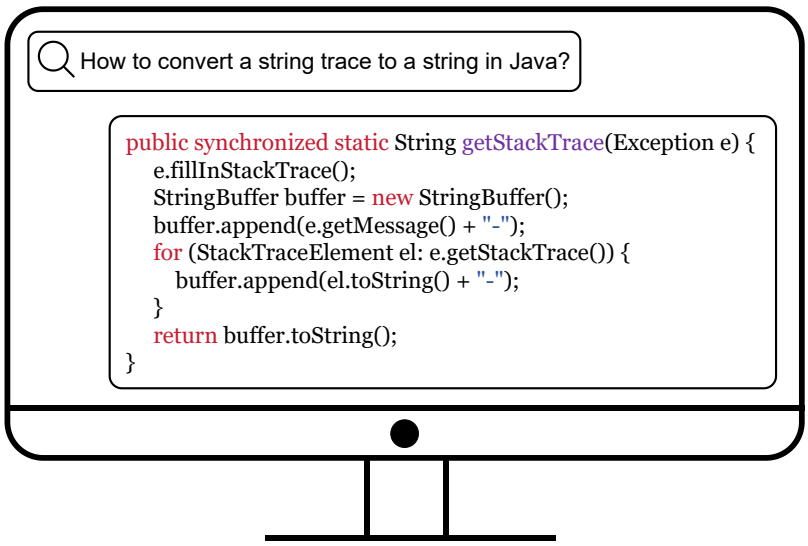


Figure 1.1: An illustration of code retrieval

shorten the software development cycle, the research of code retrieval has attracted attention from academia and industry over the past few decades.

Code retrieval is the technology that can return the desired code snippet to users according to their natural language description of the code functionality. The formal definition of code retrieval is shown as below:

Definition 1 (Code Retrieval) *Suppose we are given a set of code snippet* $C = \{c_i\}_{i=1}^{N}$*, and a set of natural language queries* $Q = \{q_i\}_{i=1}^{N}$*, where* N *is the number of pairs,* c_i *is the i-th code snippet and* q_i *is its corresponding natural language query. Our goal is to train a model* $p(s|c, q; \Theta)$ *to measure the degree of alignment when the code snippet and query are given. Here,* Θ *represents the parameters of the model,* c *denotes the given code snippet,* q *denotes the given query, and* s *represents the score utilized to describe the degree of alignment between the code snippet and query. The alignment score of a given query*

with every code snippet in the set C will be calculated and the code candidates will be returned to users in descending order of the alignment score.

Figure 1.1 shows an example of code retrieval. The user inputs the query "How to convert a string trace to a string in Java?" into the code retrieval engine, and the corresponding code snippet is returned to the user. Code retrieval technology has the following advantages for software developers. Firstly, it can shorten the software development cycle. Software engineers can directly retrieve the existing code snippet according to their requirements so that programming from scratch can be avoided. Secondly, it can reduce the difficulty of software development for software engineers. Some code functionality or algorithms are hard to achieve and require high professional skill from the software engineers. By adopting code retrieval technology, software engineers only need to inspect if the retrieved code snippets work correctly in their projects, which lowers the threshold for software development. Most current software engineers will utilize the Internet, including Stack Overflow[4] and GitHub[5] to search the relevant code during the software development. Thirdly, code retrieval technology can help software developers improve the code reuse rate in their projects. Software engineers can check whether their desired function has already been achieved in their project so that duplicate implementations of the same functionality can be avoided, which ensures the simplicity of the software structure.

```
1 public static < S > S deserialize(Class c, File xml)
      {
```

[4]https://www.stackoverflow.com/
[5]https://www.github.com/

```
2  try {
3  JAXBContext context = JAXBContext.newInstance(c);
4  Unmarshaller unmarshaller = context.
       createUnmarshaller();
5  S deserialized = (S) unmarshaller.unmarshal(xml);
6  return deserialized;
7  } catch (JAXBException ex) {
8  log.error("Error-deserializing-object-from-XML", ex)
       ;
9  return null;
10 }
11 }
```

Code Listing 1.1: Implementation for "read an object from an xml".

However, the semantic gap between the programming language and natural language hinders the performance of code retrieval technology. Conventional approaches [78, 79] have a limited ability to understand the semantic information inside the source code. For instance, Listing 1.1 is the implementation of the query "read an object from an xml". The only keyword overlapping between the code snippet and query is "xml" so that simple approaches like keyword matching will not work well in this case. How to effectively extract the semantic information between programming language and natural language becomes the critical point for code retrieval. Although deep learning-based code retrieval can perform better than conventional approaches, the performance is still unsatisfactory. One reason is that the structural information hidden inside the source code is not fully utilized during the training.

Meanwhile, the efficiency of code retrieval also needs to be improved. With the development of the open source community, the number of repositories and volume of code retrieval has snowballed in the past time. For instance, GitHub has over

100 million developers and 28 million public repositories in 2023. The acceleration of code retrieval with guaranteed accuracy is becoming increasingly important as the number of codes and visits increases. However, there are still few relevant studies on this efficiency problem.

Although the retrieved code snippets can greatly reduce the time of programming, there exists a security problem if we directly use it. The definition of software vulnerabilities including security defects [97], security bugs [109], and software weaknesses [62] is "software bugs that have security implication" [93]. The retrieved code snippets may contain software vulnerabilities, and hackers can exploit potential vulnerabilities in software to attack the software, causing huge economic losses to software users. With the rapid penetration of software in various industries, the economic losses caused are also increasing rapidly year by year. Cybercrime losses due to software vulnerabilities increased by 64% from 2020 to 2021 and further increased by 42% from 2021 to 2022[6]. To avoid involving vulnerabilities into the software, an automated software vulnerability detection tool should be employed for the retrieved code snippets. There are many automated software vulnerability detection approaches for software inspection. The conventional approaches for vulnerability detection can be classified into three categories which are static approaches, dynamic approaches, and hybrid approaches. Static approaches including symbolic execution [4, 11, 90], template matching [18,29,118], and code similarity detection [53,56], often face the problem of high false positives. Dynamic approaches such as fuzz testing [85, 103] and taint analysis [81, 87]

[6]https://www.synopsys.com/blogs/software-security/poor-software-quality-costs-us.html

will meet the problem of low code coverage. Hybrid approaches that combine static approaches and dynamic approaches are not efficient enough, so it is impractical to apply them in real scenarios. Machine learning-based approaches achieve better performance via learning the latent code patterns. However, these machine learning approaches still need experts to define the features of a software vulnerability, which requires professional domain knowledge from experts, and the cost of such a process is very high [54, 69, 120]. Therefore, the approaches that can automatically learn the code semantics from the software without human handcrafted features have become a research hotspot in the task of vulnerability detection. Here, we define the task of software vulnerability detection task as follows:

Definition 2 (Vulnerability Detection) *Suppose we are given a set of code snippets* $C = \{c_i\}_{i=1}^{N}$*, where* N *is the number of pairs, and* c_i *is the i-th code snippet. Our goal is to train a model* $p(d|c;\Theta)$ *to detect whether the given code snippet has a vulnerability problem. Here,* Θ *represents the parameters of the model,* c *denotes the given code snippet, and* d *represents the results of vulnerability detection.*

To better illustrate the structure of our thesis, we show the roadmap of our thesis in Figure 1.2. Our thesis targets intelligent reliable code retrieval, which can provide reliable code snippets to the software developers according to their description of the code functionality. The research of this thesis comprises three parts. In the first part, we focus on the effective code semantic learning approach by involving code syntax information for code retrieval. Specifically, we propose to extract both syntax and semantic information from the source code for better code representation learning. In the second part, we focus

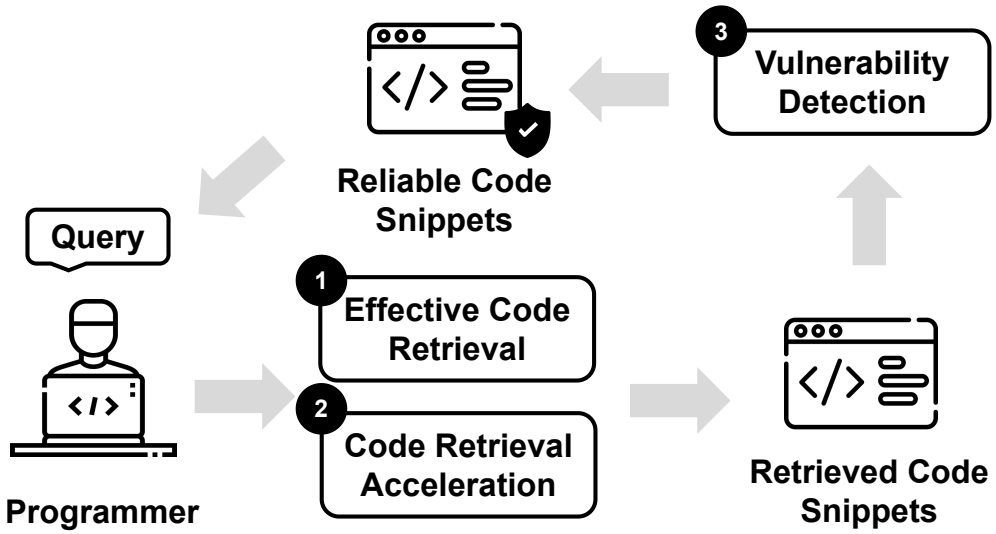


Figure 1.2: The roadmap of this thesis.

on the acceleration of code retrieval. In particular, we propose learning-based hash approaches to assist current learning-based code retrieval approaches with less performance degradation. In the third part, we focus on the software vulnerability detection task. Specifically, we propose a weakly supervised code semantic learning-based approach that can automatically learn the potential vulnerability patterns and localize the vulnerable statements without human labeling.

There for two reasons we detect the vulnerability for the retrieved code snippets rather than detect every code snippet inside the coda database in advance. At first, there are huge amounts of code snippets stored inside the code database and only a very small part will be retrieved by users. The code snippets are often updated frequently. Any update of the code snippets will lead to the re-detection. Therefore, it is unnecessary to detect all of them before the retrieval. Secondly, there are newly discovered software vulnerabilities every year and the detection

tools also need to be updated. Once the tools are updated, the re-detection of the code is needed. From the consideration of the detection efficiency, we choose such a process. Besides, such a code retrieval system will mark the code snippets which have already been detected. Unless the code snippet or the detection tools have been updated, the vulnerability detection process will be skipped when the marked code snippets have been retrieved again by the users.

1.2 Thesis Contributions

In this thesis, we make contributions to intelligent program development in the following ways:

- **Deep Code Retrieval Based on Semantic Dependency Learning**

  Previous research proposes to integrate the structural information of code, such as Abstract Syntax Tree (AST) and Control Flow Graph (CFG), for representing code semantics. However, the deep nature of the extracted trees in ASTs renders it hard for deep learning models to capture the structural information comprehensively. CFG may contain statement orders that do not contribute to the actual execution result and may lead to biased code representation learning. To address this issue, in this thesis, we propose a novel code retrieval model for Code Retrieval based on statement-level semantic Dependency Learning (CRaDLe) to encode both source code and natural language queries into unified vector representations. CRaDLe is the first code retrieval approach that integrates the dependency and semantics information at the statement level for learning code representations. The re-

sults demonstrate the superior performance of CRaDLe over the state-of-the-art baseline models.

- **Accelerating Code Retrieval with Deep Hashing and Code Classification**

  Current deep learning-based methods of code retrieval have shown promising results. However, previous methods focused on retrieval accuracy but lacked attention to the efficiency of the retrieval process. Hashing is a promising approach to improve retrieval efficiency. The performance degradation is still not avoidable during the conversion from representation vectors to binary hash codes, even when state-of-the-art hashing models are adopted. To preserve the performance of the original code retrieval models that adopt bi-encoders for the code-query encoding as much as possible, we propose a novel method for accelerating semantic Code Search with Deep Hashing and Code Classification (CoSHC). CoSHC, in this thesis. CoSHC is the first approach that adopts the recall and re-rank mechanism with the integration of code clustering and deep hashing to improve the retrieval efficiency of deep learning-based code search models. The results demonstrate that CoSHC can greatly improve retrieval efficiency while preserving almost the same performance as the baseline models.

- **Accelerating Code Retrieval via Segmented Deep Hashing**

  Although it is very efficient for previous deep hashing-based code retrieval approaches to calculate the Hamming distance, these Hamming distance-based have to scan the whole large database. To further improve the efficiency of deep hashing,

we propose a novel approach to accelerate Code Search via Segmented Deep Hashing (CSSDH). CSSDH firstly adopts hard matching objective optimization with adaptive bits relaxing to address the mismatch problem between the hash codes from different modalities. Then, CSSDH adopts the dynamic matching objective adjustment strategy, which allows the CSSDH to dynamically adjust the ground-truth label of the matching target to reduce the false positive hash collision condition. The comprehensive experiments on benchmarks demonstrate that CSSDH greatly reduces recall computational complexity while keeping the advanced performances of previous deep hashing approaches.

- **Weakly Supverised Vulnerability Detection and Localization via Multiple Instance Learning**

  Training the model for vulnerability localization usually requires ground-truth labels at the statement level, and labeling vulnerable statements demands expert knowledge, which incurs high costs. To tackle this problem, in this thesis, we propose a novel approach for Weakly supervIsed vuLnerability Detection lEarning (WILDE) to predict whether a given code snippet is vulnerable or not and meanwhile offer vulnerability localization ability. WILDE is the first approach to adopt multiple instance learning for detecting function-level vulnerabilities and localizing vulnerabilities at the statement level, all without requiring additional vulnerability labeling at the statement level. We integrate various pooling modules capable of capturing code features specific to vulnerabilities and validate the effectiveness of each pooling module on the overall performance. Moreover, we have performed compre-

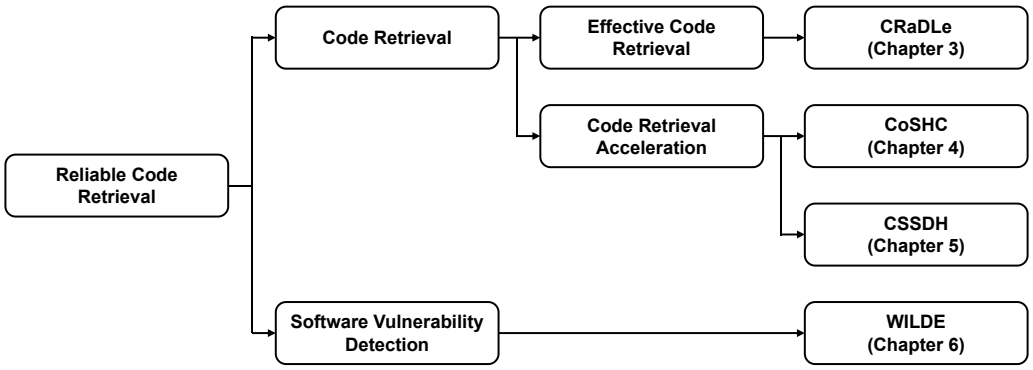


Figure 1.3: The organization of this thesis.

hensive experiments on public benchmarks, and the results indicate that WILDE achieves comparable performance in function-level vulnerability detection and outperforms previous models in statement-level vulnerability localization, showcasing state-of-the-art performance.

1.3 Thesis Organization

Figure 1.3 shows an overview of this thesis. The remainder of this thesis is organized as follows.

- **Chapter 2**
  In this chapter, we provide a systematic review of the background knowledge and related works about intelligent program development. Specifically, we first introduce the knowledge of the deep learning approaches utilized in this thesis. Then, we review the related works of code retrieval, which can be classified into non-deep learning-based approaches and deep learning-based approaches. After that, the background knowledge of hashing, which is adopted for the code retrieval acceleration, is reviewed. Finally, we review the related works

of software vulnerability detection, including function-level vulnerability detection and statement-level vulnerability localization.

- **Chapter 3**
  In this chapter, we propose a novel approach named CRaDLe for the code retrieval task. Section § 3.1 introduces the current challenge in the code retrieval task and the motivation of our approaches. Section § 3.2 gives a general overview of our approaches, which contains the design of the encoder for both modality of code and description and the fusion mechanism for dependency and semantic information. Section § 3.3 describes the dataset, evaluation metrics, and implementation detail for our approach and baselines. In Section § 3.4, we explain the experiment results and analyze the successful and failed cases in our approach. Section § 3.5 discusses the effect of different dependency information embedding ways on the overall performance. Section § 3.7 concludes this chapter.

- **Chapter 4**
  In this chapter, we propose a novel approach named CoSHC for the task of code retrieval acceleration. Section § 4.1 presents the current efficiency problem of code retrieval and our motivation. Then we introduce our proposed approach CoSHC in Section § 4.2. Section § 4.3 describes the datasets, evaluation metrics, and baseline models. Section § 4.4 discusses the experiment results, and Section § 4.7 concludes this chapter.

- **Chapter 5**
  To further improve the code retrieval efficiency, we propose a novel deep hashing lookup table-based approach named

CSSDH. We present the defect of previous deep hashing approaches and the motivation of our proposed approach in Section § 5.1. Section § 5.2 presents the overview of our proposed approach, and Section § 5.3 introduces the datasets, evaluation metrics, and implementation detail. Section § 5.4 explains the experiment results, and we conclude the chapter in Section § 5.6.

- **Chapter 6**
  We propose a novel approach named WILDE for function-level vulnerability detection with statement-level localization. In Section § 6.1, we introduce the task of vulnerability detection and our motivation. Section § 6.2 elaborates the overview of our proposed approach. Section § 6.3 explains our experiment settings and implementation details of our approach and baselines. Section § 6.4 analyze the experiment results, and we make a conclusion for this chapter in Section § 6.6.

- **Chapter 7**
  In this chapter, we first summarize this thesis in Section § 7.1. Then, in Section § 7.2, we discuss some potential future directions for intelligent program development, including repository-level code generation with large language models, reliable code generation with large language models, and vulnerability detection with static analysis and large language models.

□ **End of chapter.**

Chapter 2

Background Review

This chapter reviews some background knowledge for our proposed work and related works in the research direction we focused on. Firstly, we introduce the basic knowledge of neural networks we adopted in our thesis. Secondly, we review the related works of code retrieval, which can be classified into non-deep and deep learning-based approaches. Thirdly, we introduce the related works of hashing techniques, which are adapted to accelerate the code retrieval approaches. Finally, we review the related research of vulnerability detection, including both function-level vulnerability detection methods and statement-level vulnerability localization methods.

2.1 Neural Network Basic

As deep learning shows its great power in computer vision and natural language processing, researchers in more and more fields, including software engineering, have started integrating deep learning techniques with their research. The deep learning-based approaches made a significant breakthrough and achieved state-of-the-art performance in many tasks in software engineering.

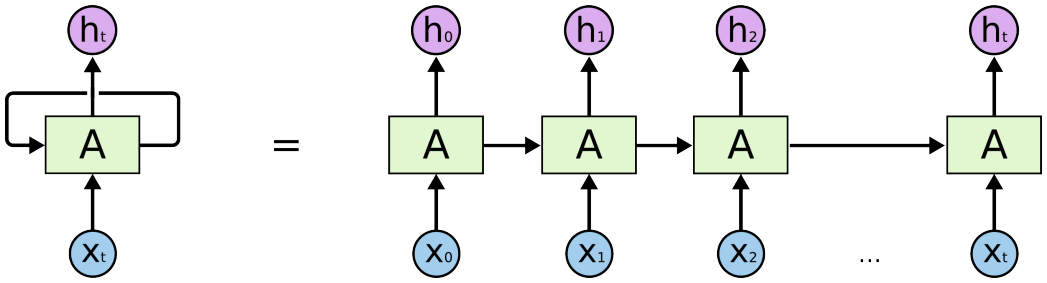


Figure 2.1: Illustration of RNN [21].

In this section, we first review the background knowledge of sequence encoders, including Recurrent Neural Networks (RNNs) and Transformer. Then, we review the pre-training techniques with Transformer architecture, which adopts the self-supervised training manner and can significantly improve the model's performance on the downstream tasks.

2.1.1 Recurrent Neural Networks

Recurrent neural network (RNN) is one of the popular sequential models that can encode sequential data into high-level representation vectors. Figure 2.1 is the illustration of RNN. The output of one RNN unit relies on the current input and the output of the last RNN unit. This feature makes RNN widely adopted in tasks with sequential inputs in software engineering, such as code retrieval, code summarization generation, commit message generation, etc. However, standard RNN suffers from the problem of long-term dependency learning. To address this issue, Long Short-Term Memory Network (LSTM) [48] and Gated Recurrent Unit (GRU) [19] were subsequently proposed. Since LSTM is adopted in the methods in this thesis, we will introduce LSTM in detail. Formally, an LSTM unit consist of a forget gate $f \in \mathbb{R}^{d_h}$, an input gate $i \in \mathbb{R}^{d_h}$, a cell state $c \in \mathbb{R}^{d_h}$,

and an output state $o \in \mathbb{R}^{d_h}$, d_h is the dimension of the hidden state. The function of the forget gate is to decide what information to discard from the cell state. The input gate decides what information will be updated in the cell state. The output gate decides what information needs to be passed to the hidden state. The detailed formulations are shown below:

$$f_t = \sigma(V_f \cdot x_t + W_f \cdot h_{t-1}), \tag{2.1}$$

$$i_t = \sigma(V_i \cdot x_t + W_i \cdot h_{t-1}), \tag{2.2}$$

$$\widetilde{c}_t = \tanh(V_c \cdot x_t + W_c \cdot h_{t-1}), \tag{2.3}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \widetilde{c}_t, \tag{2.4}$$

$$o_t = \sigma(V_o \cdot x_t + W_i \cdot h_{t-1}), \tag{2.5}$$

$$h_t = o_t \cdot \tanh(c_t) \tag{2.6}$$

Since the bi-directional RNN [22] can get the contextual information during the encoding, it can generate more effective representation vectors, and its application is more widely used than RNN.

2.1.2 Transformer and Pre-Training

Transformer [105] is another sequential model that can be utilized for representation learning. Figure 2.2 is the illustration of Transformer. The vanilla Transformer model contains an encoder and a decoder. Both of them are composed of L layers of Transformer blocks. Each Transformer block includes two sublayers: multi-head self-attention pooling and a position-wise feed-forward network. The multi-head self-attention layer calculates the dot-product attention as follows:

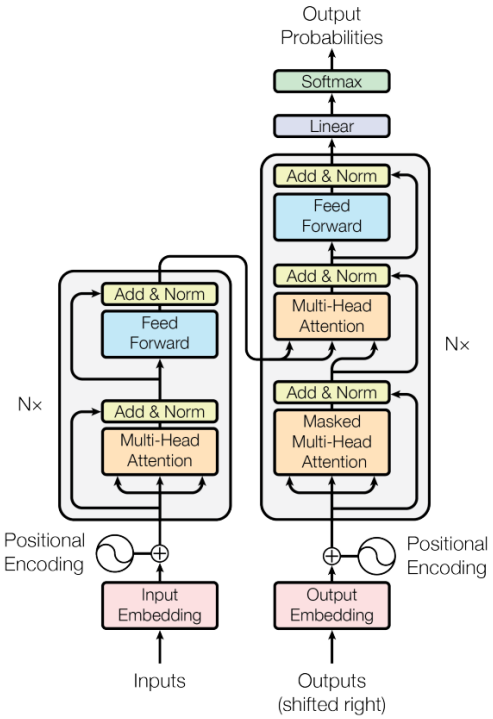


Figure 2.2: Illustration of Transformer [105].

$$\text{Attention}(q, K, V) = \text{softmax}(\frac{q, k}{\sqrt{d}})V, \tag{2.7}$$

where d denotes the dimension of vectors, q denotes the query, K denotes keys and V denotes values. Each attention head is expected to learn different semantic information from the input sentence. The attention vectors from all attention heads will be concatenated into a single vector and fed into the position-wise feed-forward network. Since Transformer has no assumption on the order of the input sequence during the self-attention operation, Transformer can effectively capture long-term dependencies in the input sequence. It has been demonstrated that Transformer can outperform all RNN-based models in many tasks. What's more, the pre-training technique greatly improves the performance of Transformer. Transformer can utilize huge

amounts of data via training in a self-supervised manner. Many researchers have proposed code-based pre-training models, including CodeBERT [33] and GraphCodeBERT [43]. These pre-training models only need to be finetuned with a small-scale dataset and achieve state-of-the-art performance in many tasks such as code summarization generation and commit message generation.

2.2 Code Retrieval

Code retrieval is a common practice for programmers to reuse existing code snippets in open-source repositories. Given a user query (i.e., a natural language description), code retrieval aims at searching for the most relevant ones from a set of code snippets. By adopting code retrieval approaches, the cost of software development for programmers can be significantly reduced. This section divides the current code retrieval approaches into two categories: non-deep learning-based and deep learning-based approaches.

2.2.1 Non Deep Learning Based Approaches

Prior works have explored several methods to find the implicit connections between human language queries and code databases. Early studies mainly concentrate on extracting valuable features from both codes and queries. For example, the work [98] extracts scattered verbs from queries and applies an action-oriented identifier graph model to inspect the result graph, which helps to optimize the queries. Lu et al. [76] reformulate and extract natural language phrases from source code identifiers since the synonyms in source codes and NL queries may signif-

icantly affect the code search result. The work [79] proposes Portfolio which uses random surfer to model the navigation behavior of programmers. Then, with an association model based on Spreading Activation Network [23], functional relevant functions can be set in the same list. Ponzanelli et al. [86] propose to retrieve pertinent discussions from Stack Overflow when given a context in the IDE, which saves developers' time spent on formulating more standardized queries.

2.2.2 Deep Learning Based Approaches

NCS [94] firstly adopts FastText [7] to embed both queries and source code into representation vectors and retrieves the target code by calculating the cosine similarity between the representation vectors of source codes and queries. UNIF [12] adopts the bag-of-words model to embed code snippets and queries into a shared embedding space. CODEnn [41] considers the features of API sequences, method name tokens, and code tokens and fuses these features in the final representation of the code snippets. CoaCor [125] regards code annotation and code search as dual tasks and adopts the reinforcement learning model to improve code search performance with the generated code annotations. Husain et al. [51] validate the effectiveness of different neural architectures for source code representation on the task of code retrieval and discover that the self-attention model performs the best among all the models. To capture the local structural information, Liu et al. [74] propose GraphSearchNet to construct graphs and jointly learn the high-level semantics between code and queries. Zeng et al. [129] propose a novel approach named deGraphCS, which can model code semantics more precisely by transferring the code into variable-based flow. CodeBERT [33]

is a bimodal pre-trained model for programming language and natural language, and its effectiveness on downstream tasks, including code retrieval, has also been demonstrated. GraphCodeBERT [43] is another pre-trained model similar to the CodeBERT but considers the program's data flow during the pre-training. CodeT5 [116] is a unified pre-trained encoder-decoder Transformer model that is trained with the identifier-aware pre-training task, and these tasks can help CodeT5 to distinguish the code tokens belonging to identifiers and recover the masked identifiers. Similarly, SPT-Code [83] is a sequence-to-sequence pre-trained model with three pre-training tasks. With these pre-training tasks, SPT-code can learn knowledge of source code, the corresponding code structure, and a natural language description of the code without relying on any bilingual corpus. SyncoBERT [114] is another pre-trained model with identifier prediction and AST edge prediction objectives. To address the problem that the encoder-decoder framework is sub-optimal for auto-regressive tasks, UniXcoder [42] is a unified cross-modal pre-trained model that utilizes mask attention matrices with prefix adapters to control the behavior of the model. Bui et al. [10] propose a self-supervised contrastive learning framework named Corder, which can distinguish similar and dissimilar code snippets via a contrastive learning objective during training. Shi et al. propose a soft data augmentation approach that dynamically masks tokens to generate positive source code examples for contrastive learning. Shi et al. [99] propose a soft data augmentation approach that dynamically masks tokens to generate positive source code examples for contrastive learning. To capture the local structural information, Liu et al. propose GraphSearchNet to construct graphs and jointly learn the high-level

semantics between code and queries.

2.3 Hashing

Hashing is a promising approach to improve retrieval efficiency and is widely adopted in many retrieval tasks. The hashing technique can convert high-dimensional vectors into low-dimensional binary hash code, significantly reducing storage and calculation costs. In this section, we briefly introduce some representative hash table-based hashing approaches and Hamming distance-based cross-modal hashing approaches, which can be classified into supervised and unsupervised cross-modal hashing approaches.

2.3.1 Hash table-based approaches

Locality Sensitive Hashing (LSH) [25] is one of the most popular approaches for recalling data in high-dimensional space. LSH maps high dimensional data to hash value by using random hash functions. Once the data is converted to the hash value, LSH can build the lookup hash tables for the data recall. There are several variants of LSH [5, 36, 50]. Most of them need to build many lookup hash tables to guarantee the recall rate of data. Semantic Hashing [95] is a learning-based approach and can also construct the lookup hash table for the data recall. By Compressing the data point within the Hamming distance threshold into the same hash bucket, Semantic Hashing does not need to construct many hash tables to guarantee the recall ratio. However, the storage cost will dramatically increase since many duplicated data are inside the hash table. In addition, most of

the above approaches are single-modal and do not consider the cross-modal problem.

2.3.2 Supervised cross-modal hashing approaches

Bronstein et al. [9] consider the embedding of the input data from two arbitrary spaces into the Hamming space as a binary classification problem with positive and negative examples and adopt boosting algorithms to learn the mapping. SCM [130] is proposed to reduce the training time complexity of most existing SMH methods by seamlessly integrating semantic labels into the hashing learning procedure for large-scale data modeling. SePH [71] converts semantic affinities of training data into a probability distribution and approximates it with to-be-learned hash codes via minimizing the Kullback-Leibler divergence. MCSCH [126] sequentially generates the hash code guided by different scale features through an RNN model with the scale information to reduce the error caused by the extreme situation in specific features.

2.3.3 Unsupervised cross-modal hashing approaches

CMFH [26] learns unified hash codes by collective matrix factorization with latent factor model from different modalities of one instance. UDCMH [119] incorporates Laplacian constraints into the objective function to preserve not only the nearest neighbors but also the farthest neighbors of data. DJSRH [101] proposes to train the hashing model with a joint-semantics affinity matrix that integrates the original neighborhood information from different modalities to capture the latent intrinsic semantic affinity for the multi-modal instances. Yang et al. [124] propose DSAH

with a semantic-alignment loss function to align the similarities between features and also attempt to reconstruct features of one modality with hash codes of the other one to bridge the modality gap further. JDSH [73] utilizes Distribution-based Similarity Decision and Weighting (DSDW) for unsupervised cross-modal hashing to generate more discriminative hash codes.

2.4 Vulnerability Detection

Software vulnerabilities are flaws in the logical design of software or operating systems that can be exploited maliciously by attackers. By exploiting these vulnerabilities, attackers can implant Trojan horses and viruses over networks, extract crucial user information, and even inflict severe damage to the system. In this section, we review the related works about deep learning-based function-level vulnerability detection and deep learning-based statement-level vulnerability detection and localization.

2.4.1 Deep Learning-based Vulnerability Detection

With the advent of deep learning technology, significant advancements have been made in various tasks, such as code retrieval and code generation. Consequently, researchers in the vulnerability detection field have also taken notice. Integrating deep learning into vulnerability detection approaches has resulted in a substantial performance improvement compared to conventional methods. Existing studies in this area can be broadly categorized into token-based and graph-based approaches, each utilizing different source code representations. In the following subsections, we provide a brief overview of these two types of approaches.

Token-based Vulnerability Detection Approaches

Several works [24, 68, 69] approach source code as flat sequences and adopt natural language processing techniques to represent input code and initialize tokenized code tokens with Word2Vec [80]. Li et al. [69] initiate the study of using deep learning for vulnerability detection. They transform programs into code gadgets consisting of multiple lines of code statements that are semantically related and propose VulDeePecker, a Bidirectional Long Short Time Memory (BiLSTM) neural network with a dense layer to learn representations. To further represent programs into vectors that accommodate the syntax and semantic information suitable for vulnerability detection, Li et al . [68] extract code slices according to data dependency and control dependency and utilize BiLSTM to obtain representation for detection. Another work [24] leverages Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) to learn features from source code directly.

Graph-based Vulnerability Detection Approaches

Source code is inherently structured and logical and has heterogeneous aspects of representation, such as Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), and Program Dependency Graph (PDG). Many works [15, 17, 63, 66, 82, 111, 121, 133] represent source code into single code graphs or composite graphs to improve the syntactic and semantic information. Zhou et al . [133] construct a heterogeneous joint graph consisting of AST, CFG, and DFG following [122]. They connect the neighboring leaf nodes of the AST to preserve the natural sequential order of the source code and utilize a Gated

Graph Neural Network (GGNN) with the convolution module for graph-level classification. Chakraborty et al . [15] extract the information of Code Property Graph (CPG) [122] from the given function and adopts the technique of Word2Vec [80] to initialize the embedding vector. They also utilize the GGNN model to learn graph representation and focus on solving the problem of dataset imbalance. Li et al . [66] also extract multiple types of graphs. However, unlike the above two methods, which use GNN to learn the representation of joint graphs directly, they leverage Gate Recurrent Unit (GRU) [20] or Tree-LSTM [104] to learn a single type of graph respectively and aggregate these vectors through convolution operation. Cheng et al. [17] and Wu et al . [121] both distill the function semantic information into a PDG, which contains control-flow and data-flow details of source code, and they train a GNN model and a Convolutional Neural Network (CNN) [60] model to detect the vulnerability, respectively. To address the learning problem of long-distance node relationships, Wen et al. [117] propose a novel approach named AMPLE which can reduce the sizes of code structure graphs.

2.4.2 Deep Learning-based Statement-Level Vulnerability Detection and Localization

Although deep learning-based approaches for function-level vulnerability detection have attracted many researchers and achieved significant progress, there are still limitations in practical applications. Even if the function-level vulnerability can be successfully detected, it still requires considerable effort to locate the vulnerable statements inside the function if the vulnerable function contains many statements.

Li et al. [66] leverage GCN for function-level predictions and GNNExplainer [128] to explore the subgraphs that contribute most to the predictions. However, it still cannot provide specific vulnerable statements and many statement-level vulnerability detection approaches [27, 35, 47] have been proposed. Ding et al. [27] and Hin et al. [47] provide labels for vulnerable statements in the training phase. Ding et al. [27] propose an ensemble learning approach named VELVET, which combines GGNN and Transformer [106] to capture the local and global context of the source code. VELVET shows good performance on both vulnerability classification and localization. Hin et al. [47] formulate statement-level vulnerability detection as a node classification task. They use CodeBERT [33] to initialize the embedding of each statement in the function, and further leverage Graph Attention Network (GAT) [107] to update these statement embeddings according to the control and data dependency between statements. The performance of these supervised approaches is sensitive to the quality and quantity of the annotated data. To solve this issue, Hin et al. [35] propose an unsupervised approach named LineVul that leverages the attention mechanism of CodeBERT [33] to find the most likely vulnerable statements. LineVul uses CodeBERT to capture the statements' long-term dependencies and semantic context and aggregate each statement's attention score for statement-level vulnerability detection. The experiment results indicate that unsupervised statement-level vulnerability detection is feasible and has room for further improvement.

Chapter 3

Deep Code Retrieval Based on Semantic Dependency Learning

In this chapter, we investigate semantic dependency learning for code retrieval. Code retrieval is a common practice for programmers to reuse existing code snippets in open-source repositories, and the current main challenge of effective code retrieval lies in mitigating the semantic gap between natural language descriptions and code snippets. The previous methods only partially involve the code structural information during the learning, which limits the performance of models. To address this problem, we propose CRaDLe, a novel approach for code retrieval based on statement-level semantic dependency learning. We first extract the program dependency information from the code snippets and integrate the dependency and semantics information at the statement level for learning code representations. Then, we conduct large-scale experimental evaluations on public benchmarks. The results demonstrate the superior performance of CRaDLe over the state-of-the-art baseline models.

3.1 Introduction

Implementing projects from scratch is tedious for programmers. In most cases, they know what they want to do, but do not have the capability to implement all the details. For example, a Python programmer may want to "*convert date_string into datetime format*", but not be able to recognize the proper syntax `datetime.strptime(date_string, format)` for the realization. To mitigate the impasse, it is common for programmers to search the web in natural language (NL), find relevant code snippets, and modify them into the desired form [8]. Many code retrieval approaches [8,78,79] have been proposed to improve the recommendation accuracy of the returned code snippets given a natural language description. The main challenge of effective code retrieval is the semantic gap between source code and natural language descriptions since the two sources are heterogeneous and share few common lexical tokens, synonyms, or language structures [41].

Prior efforts have been conducted for effective code retrieval. The existing research can be divided into two categories according to the involved techniques, i.e., Information Retrieval (IR)-based and Deep Neural Network (DNN)-based. The IR-based techniques rely on token-wise similarities between source code and queries. Since the variable and API definitions in code are generally word combinations or abbreviations in natural language, more semantically similar tokens in code and queries can indicate more relevancy between them. For example, McMillan et al. propose Portfolio which utilizes keyword matching and PageRank to return a list of functions [79]. Lv et al. propose CodeHow to combine API matching for code retrieval [78]. With

an increasing amount of available source code and the flourishing development of deep learning techniques, many studies [41, 51] propose to adopt neural network models for jointly encoding tokens of source code and queries into a single and joint vector space, where one encoder is employed for each input (natural or programming) sequence. The objective is to map semantically relevant code and language into vectors that are near each other in the vector space.

Considering the highly structured characteristic of source code, recent research proposes to integrate the structural information of code such as Abstract Syntax Tree (AST) and Control Flow Graph (CFG) for representing code semantics [110, 127, 131], demonstrating the effectiveness of involving structural information for the task. However, the deep nature of the extracted trees in ASTs renders it hard for deep learning models to comprehensively capture the structural information [131]. CFG, which represents all possible execution paths for a program, may contain statement orders that are not contributing to the actual execution result, probably leading to biased code representation learning [110]. In this chapter, we propose to utilize statement-level dependency relations in a code snippet based on the Program Dependency Graph (PDG). The PDG is established based on AST but less deeper than AST in the structure and only retains the execution paths that will affect the execution result. The dependency relations are then explicitly integrated with the statement-level semantics to capture the code semantics. Actually, the effectiveness of incorporating dependency relations for code representation learning has proven in tasks such as bug detection [67] and code clone detection [45]; while no prior work has explored the impact on the code retrieval task so far.

Specifically, we introduce a novel neural network model named CRaDLe, an abbreviation of Code Retrieval based on semantic Dependency Learning. CRaDLe couples both structural and semantic information of code at the statement level, where the code structures are extracted based on PDG. Extensive experiments have been conducted to verify the performance of the proposed approach. The evaluation results show that CRaDLe can significantly outperform the state-of-the-art models by at least 36.38% and 22.34% on two real datasets respectively, in terms of R@1, one standard metric for validating recommendation performance.

In summary, the main contributions of this chapter include:

- We propose a novel code retrieval model, CRaDLe, to encode both source code and natural language queries into unified vector representations. CRaDLe is the first code retrieval approach that integrates the dependency and semantics information at the statement level for learning code representations.
- We conduct large-scale experimental evaluations on public benchmarks. The results demonstrate the superior performance of CRaDLe over the state-of-the-art and baseline models.

3.2 Methodology

In this section, we elaborate on the overview and detailed design of the proposed approach CRaDLe, including the code encoder, description encoder and the similarity measurement component.

3.2.1 Overview

Figure 3.1 depicts the overview of the proposed approach, CRaDLe. The implementation includes both offline and online modes. During the offline stage, we first collect datasets containing ⟨code, description⟩ pairs. The collected code and descriptions are then preprocessed and separately encoded into vectors by the code encoder and query encoder respectively. Unified representations of code and corresponding descriptions are finally learned after the offline training process, where semantically similar code and descriptions are located closely to each other in the same embedding space. During the online process, when a new natural language query arrives, the trained model recommends the most related code snippets to the programmer according to the semantic distances between the code and the query in the embedding space.

Figure 3.2 illustrates the overall framework of the CRaDLe approach, which details the design of the code encoder and description encoder. The code encoder fuses the statement-level token semantics and distills dependency information to represent the code semantics. The description encoder also embeds the token sequences in the descriptions to vectors. Finally, similarity matching scores between the code and descriptions are learned based on their respective vector representations.

3.2.2 Code Encoder

The code encoder aims at embedding code snippets into vector representations. We propose to integrate the statement-level token semantics with the dependency information between statements for accurately capturing the code semantics. We first

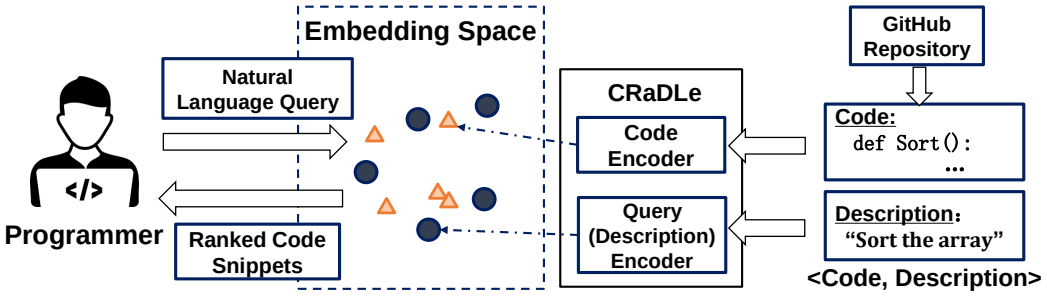


Figure 3.1: Overview of the proposed CRaDLe.

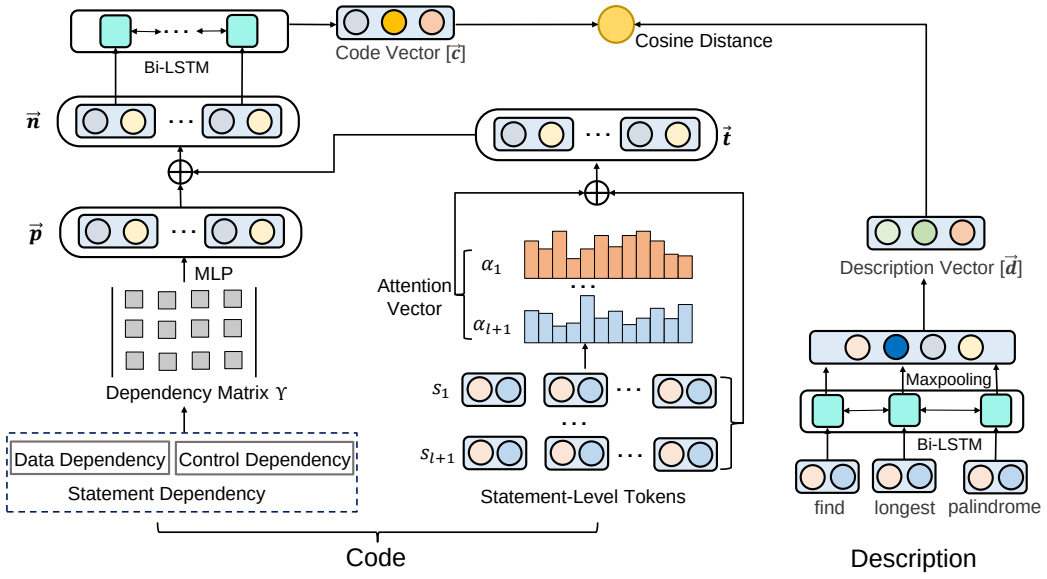


Figure 3.2: Overall framework of the proposed CRaDLe.

illustrate the process conducted for the dependency information extraction, and then describe the networks proposed for learning statement-level dependency and semantic representations.

Algorithm 1 shows the procedures for the code encoder. The input of the code encoder includes the token matrix E comprised by a sequence of token embedding vectors $\{\mathbf{e}_{1,1}, ..., \mathbf{e}_{i,j}, ...\}$ and dependency matrix Υ. First, the dependency embedding layer encodes the dependency matrix Υ into dependency embeddings P. The token embedding layer then represents the token

matrix E into statement-level representations T comprised by $\{\mathbf{t}_1, ..., \mathbf{t}_i, ...\}$. Finally, the token embeddings are concatenated with the dependency embeddings at the statement level, and the newly comprised vectors are fed into the Bi-LSTM layer. The last hidden state vector from the Bi-LSTM layer is treated as the representation vector of the code.

Algorithm 1: The algorithm of code encoding

input : the token matrix E, the matrix of input dependency: Υ
output: The representation vector of code: C
Function `CODEENCODER(`E, Υ`):`

$P \leftarrow DependencyEmbedding(\Upsilon)$; `// corresponding to Equation 3.1`
$T \leftarrow TokenEmbedding(\mathrm{E})$;
$S \leftarrow StatementAttention(T)$; `// corresponding to Equation 3.2 and Equation 3.3`
$C \leftarrow SemanticDependencyEmbedding([S; P])$;
`// corresponding to Equation 3.4`
return C;

Dependency Information Extraction

We obtain the dependency information between statements by adopting PDGs of the code snippets. PDG explicitly indicates the data dependency and control dependency of a program, where the data dependency can represent the relevant data flow relationships and control dependency exhibits the essential control flow relationships [34]. Since there exists no mature tool for extracting the PDG of one code snippet in interpreted languages such as Python, we propose to establish the PDG based on the AST of a code snippet.

For clarifying the PDG establishment process, we use the code example illustrated in Listing 3.1. Figure 3.3 (a) depicts the

mark for each statement in the code example, in which we regard the function name and required parameters as two separate statements. Function name can be treated as a short summary of the code functionality; while the definitions of the required parameters generally reflect the semantics of the input data. Treating function names and parameters separately could be helpful for capturing their respective semantics. In our approach, the definition of a statement is a line of the code.

Figure 3.3 (b) demonstrates the simplified AST of the code example where we construct the AST at the statement level and hide the details of each statement. The *data dependency* of one statement with the other statement can be identified if the variable used in one statement is (re)defined in the other statement and the value of the variable is unchanged on the execution path between these two statements. The *control dependency* of one statement with the other is determined if the execution of the statement relies on the execution results of the other one. The control dependency can be directly captured by the tree structure in the AST, i.e., statements in child leaf nodes are considered to possess dependent relations with the statements in the parent nodes. The extracted PDG is depicted in Figure 3.3 (c), with red arrowed lines and black arrowed lines indicating data dependency and control dependency between the two statements, respectively. Tokens beside each statement block denote the related variables, in which we use black or red underlined variables to distinguish whether the variables are used or (re)defined in the corresponding statement. For example, the parent nodes of S_{10} in the AST include S_3, S_5, S_7, and S_9 (as shown in Figure 3.3 (b)), so the control dependency between S_{10} and $S_{3,5,7,9}$ is marked in the obtained PDG. Also, the

variable `mid` in S_6, corresponding to line 5 in the Listing 3.1, is from S_4, i.e., line 3 in the code example; so S_6 shows a data dependency relation to S_4 in the PDG.

Statement-Level Dependency Embedding

The dependency embedding network is designed to encode the data dependency and control dependency involved in the PDG of a code snippet into a vector representation. According to the extracted PDG (as shown in Figure 3.3 (c)), we can build a dependency matrix $\Upsilon \in \{0,1\}^{(l)\times(l)}$, where l indicates the number of statements in the code. The element $\upsilon_{ij} = 1$ if the i-th statement has a data/control dependency on the j-th statement; otherwise $\upsilon_{ij} = 0$. Note that $\upsilon_{ij} \neq \upsilon_{ji}$. For example, S_4 and S_6 exhibit a data dependency relation, so $\upsilon_{64} = 1$. To embed the obtained dependency matrix Υ, we employ one layer of multi-layer perceptron (MLP):

$$\begin{aligned} \mathbf{p}_i &= \tanh(\mathbf{W}^{\Gamma}\upsilon_i), \forall i = 1, 2, ..., l, \\ \mathbf{P} &= [\mathbf{p}_1, ..., \mathbf{p}_{(l)}], \end{aligned} \tag{3.1}$$

where $\mathbf{W}^{\Gamma}$ is the matrix of trainable parameters in MLP and $\mathbf{p}_i$ is the embedding of the dependency information for each statement.

Statement-Level Token Embedding

Since the dependency matrix will be extremely large if we encode the code at the token level, here we embed the code tokens at the statement level and then feed them into the code model. The token embedding network is designed to capture the semantics of each statement based on the constituted tokens. We

first tokenize the statements into sequences of tokens following Gu et al.'s work [41], during which process duplicate tokens and the keywords in the programming language such as `while` and `break` are removed. Then tokens in each sequence are embedded into vectors individually through an embedding layer. An attention layer is utilized to compute a weighted average. Given a sequence of token embedding vectors $\{\mathbf{e}_{i,1}, ..., \mathbf{e}_{i,j}, ...\}$ for the i-th statement, the attention weight $\alpha_{i,j}$ for each $\mathbf{e}_{i,j}$ is calculated as follows:

$$\alpha_{i,j} = \frac{\exp(\mathbf{e}_{i,j}^{\intercal})}{\sum_j \exp(\mathbf{e}_{i,j}^{\intercal})} \tag{3.2}$$

Each statement is embedded based on the attention weights $\alpha_{i,j}$.

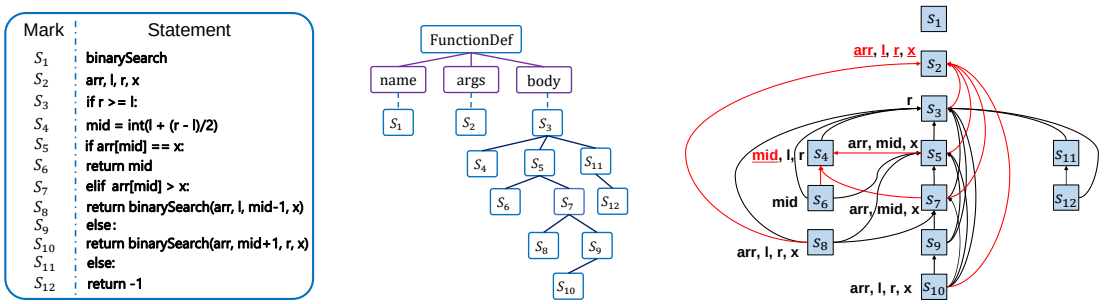
$$\mathbf{t}_i = \sum_j \alpha_{i,j} e_{i,j}^{\intercal}, \tag{3.3}$$

where i indicates the i-th statement.

Semantic Dependency Embedding

We consider both statement-level dependency and semantic information for learning the vector representation of a code snippet. Specifically, for each statement s_i, we concatenate its dependency embedding $\mathbf{p}_i$ and token embedding $\mathbf{t}_i$ as the representation of the statement, i.e., $\mathbf{s}_i = [\mathbf{t}_i; \mathbf{p}_i]$. We finally adopt bi-LSTM to encode the sequence of the statement embeddings and use the last hidden state as the vector representation of the code.

$$\mathbf{c} = \text{BiLSTM}(\mathbf{h}_l, \mathbf{s}_l), \tag{3.4}$$



(a) Marked statements. (b) The simplified AST. (c) The extracted PDG.

Figure 3.3: Workflow for extracting PDG of the code snippet in Listing 3.1. For the extracted PDG in (c), red and black arrowed lines indicate data dependency and control dependency respectively. The tokens beside each statement block denote the variables (re)defined (highlighted in red underlined font) or used in the corresponding statement.

where l indicates the number of statements.

```
1  def binarySearch (arr, l, r, x):
2      if r >= l:
3          mid = int(l + (r - l)/2)
4          if arr[mid] == x:
5              return mid
6          elif arr[mid] > x:
7              return binarySearch(arr, l, mid-1, x)
8          else:
9              return binarySearch(arr, mid+1, r, x)
10     else:
11         return -1
```

Code Listing 3.1: An example of Python code snippet for illustrating the semantic dependency learning process.

3.2.3 Description Encoder

The description encoder aims at embedding natural language descriptions into vectors. Given a description $D = \{w_1, ..., w_k, ..., w_{N_d}\}$ comprising a sequence of N_d words, the description encoder em-

beds it into a vector $\mathbf{d}$ using a bi-LSTM model with max-pooling:

$$\begin{aligned} \mathbf{h}_k &= \text{BiLSTM}(\mathbf{h}_{k-1}, \mathbf{w}_k), \forall k = 1, 2, ..., N_d, \\ \mathbf{d} &= \text{maxpooling}([\mathbf{h}_1, ..., \mathbf{h}_{N_d}]) \end{aligned} \tag{3.5}$$

The max-pooling layer is used to mitigate the effect of long-term information loss caused by the LSTM mechanism and catch the global feature of the whole sentence.

3.2.4 Similarity Measurement

The semantic similarity between the code vector $\mathbf{c}$ and description vector $\mathbf{d}$ is calculated based on its cosine distance in the embedding space:

$$\cos(\mathbf{c}, \mathbf{d}) = \frac{\mathbf{c}^\intercal \mathbf{d}}{||\mathbf{c}||||\mathbf{d}||} \tag{3.6}$$

The vector features of the two different embedding models are trained using the loss function, i.e., Equation 3.6, to maximize the cosine similarities in the projected space, so aligned code and descriptions would be close to each other in the space. Such design is widely adopted in prior code search studies [12, 41, 94]. The target of the design is to get unified representations for both code and description, so as to mitigate the problem of semantic gap between them. The higher the similarity, the more relevant the code is to the description.

3.2.5 Model Training

We obtain the representation vectors for code snippets and descriptions based on the proposed code encoder and description encoder, respectively. Following previous studies [12, 41, 94], we

project the code vectors and description vectors to the same space and train the vectors for aligned code snippets and descriptions to be close in the space.

Specifically, every single code snippet in the training data T will be constructed as a triplet $< C, D+, D- >$. C represents the code snippet from the training Corpora, $D+$ indicates the description that semantically matches the code snippet in the ground truth, and $D-$ denotes the negative description which is randomly chosen from the training corpora with the true description excluded. The loss function is as below:

$$\mathcal{L}(\theta) = \sum_{<C,D+,D->\in T} \max(0, \epsilon - \cos(\mathbf{c}, \mathbf{d}+) + \cos(\mathbf{c}, \mathbf{d}\text{-})), \quad (3.7)$$

where θ denotes the parameters in the proposed model, $\mathbf{c}$ denotes the code vector of C, $\mathbf{d}+$ and $\mathbf{d}$- denote the description vectors of $D+$ and $D-$, respectively. Based on the training loss function, we can get unified representations for both code and description, thus mitigating the semantic gap between them.

3.3 Experimental Setup

In this section, we introduce the collected dataset for experimentation, the evaluation metrics, implementation details, and baseline models.

3.3.1 Dataset Collection

Two datasets are adopted for our experimental evaluation. One dataset is obtained from CodeSearchNet [51], a publicly-available GitHub repository. We focus on the Python program language

since it is one of the most popular programming languages, accounting for more than 30% of the total market share as PYPL reported [88]. Detailed statistics of the dataset can be found in Table 3.3. All the code in the corpus is in Python and with English descriptions. We have 407,126, 22,302, and 21,902 ⟨code, description⟩ pairs for training, validating, and testing, respectively. The median and average numbers of the statements in the code are around 10. We also observe that the statements contain around three tokens on average, with the minimum at zero which is because the input parameters beside the method name are treated as an individual statement and some code snippets may not require any input parameters. Another dataset is from Code2seq [2], with the statistics illustrated in Table 3.4. We only select the code written in Python 3 from both datasets since the PDG extraction tool (introduced in Section § 3.2.2) is specifically designed for Python 3 and may fail to parse the code written in Python 2.

Table 3.1 and Table 3.2 illustrate the distribution of statement numbers of the codes in the two datasets, i.e., CodeSearchNet and Code2Seq, respectively. We can observe that the long tail phenomenon occurs in the two datasets. Besides, more than 50% of the code has $\leq$10 statements and more than 80% has $\leq$20 statements.

3.3.2 Performance Measurement

Following the evaluation settings in [110], we fix a set of 999 distractor snippets $\mathbf{c_j}$ for each test pair $(\mathbf{c_i}, \mathbf{d_i})$ and calculate the average ranking score for all the testing pairs as the evaluation result. We involve two metrics: $R@k$ and MRR, for validating the ranking performance.

#Statements	Training Set	Validation Set	Test Set
$0 \sim 10$	230,183	12,413	12,326
$11 \sim 20$	117,060	6,364	6,361
$21 \sim 30$	32,904	1,875	1,843
$31 \sim 40$	12,834	755	633
$41 \sim 50$	5,723	386	326
$51 \sim$	8,422	509	413

Table 3.1: Statistics of the number of statements in CodeSearchNet dataset.

#Statements	Training Set	Validation Set	Test Set
$0 \sim 10$	218,679	32,429	33,210
$11 \sim 20$	73,870	11,301	12,251
$21 \sim 30$	2,0956	3,215	3,478
$31 \sim 40$	7,957	1,251	1,370
$41 \sim 50$	3,540	573	632
$51 \sim$	4,326	650	786

Table 3.2: Statistics of the number of statements in Code2Seq dataset.

R@k

$R@k$ is a common metric to evaluate whether an approach can retrieve the correct answer in the top k returning results. It is widely used by many studies on the code retrieval task. The metric is calculated as follows:

$$R@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\delta(FRank_q < k), \tag{3.8}$$

where Q denotes the query set and $FRank_q$ denotes the rank of the correct answer for query q. The function $\delta(Frank_q < k)$ returns 1 if the rank of the correct answer is within the top k returning results otherwise the function returns 0. A higher $R@k$ indicates a better code retrieval performance.

	Training	Validating	Testing
# ⟨code, description⟩	407,126	22,302	21,902
Statistics of # statements in code			
Min.	1	1	1
Med.	7	8	7
Max.	1,385	909	363
Ave.	11.45	11.87	11.24
Statistics of # tokens in the statements			
Min.	0	0	0
Med.	3	3	3
Max.	514	155	83
Ave.	3.92	3.87	3.91

Table 3.3: Statistics of the CodeSearchNet dataset.

MRR

Mean Reciprocal Rank (MRR) is the average of the reciprocal ranks of the correct answers of the query set Q, which is another popular evaluation metric for the code retrieval task. The metric MRR is calculated as follows:

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{FRank_q} \tag{3.9}$$

The higher the MRR value is, the better performance the model has.

3.3.3 Implementation Details

In our experiment, we select the top 10,000 words according to the word frequencies as the vocabularies of code snippets and descriptions, respectively. All the word embeddings are randomly initialized and adjusted during training. The dimension

	Training	Validating	Testing
# ⟨code, description⟩	329,328	49,419	51,727
Statistics of # statements in code			
Min.	2	2	2
Med.	7	7	7
Max.	1,463	416	1,463
Ave.	10.17	10.33	10.68
Statistics of # tokens in the statements			
Min.	0	0	0
Med.	3	3	3
Max.	682	199	1864
Ave.	3.75	3.73	3.73

Table 3.4: Statistics of the Code2seq dataset.

of word embedding is set as 256. All LSTMs have 1024 hidden units in each direction. The maximum number of considered statements in the code and the maximum number of tokens in each statement are set as 20 and 5, respectively. The sequence lengths of descriptions are limited to 30 following the work [41]. The CRaDLe model is trained via the AdamW algorithm [58] and the learning rate is 2.08e-4. To mitigate the over-fitting issue, we add a dropout layer with a dropout rate of 0.25. We train our models on a server with one Nvidia GeForce RTX 2080 Ti and 11 GB memory. The training lasts ∼20 hours with 200 epochs and the early stopping strategy [37] is adopted to avoid overfitting.

3.3.4 Baseline Models

We compare our proposed model with several state-of-the-art baseline models. **CODEnn** is one of the state-of-the-art mod-

els proposed in [41]. This model extracts the method name, API sequence, and tokens from the code and utilizes a neural network to learn the unified vector representation of the query and these code features. **UNIF** [12] focuses on the semantic information from the tokens in the code and utilizes embedding techniques and attention mechanisms to embed the tokens in the query and code into a single vector respectively. The projection of the query and code vector in the same space is learned by this model. **NeuralBoW** [113] embeds each token in the two input sequences to a learnable embedding. The token embeddings are then combined into a sequence embedding using max-pooling and an attention-like weighted sum mechanism. The **RNN** baseline adopts a two-layer bi-directional LSTM model [19] to encode the input sequences. **CONV** [57] uses a 1D convolutional neural network over both the input sequences of tokens. **CONVSelf** [72] combines a 1D convolutional neural network and a self-attention layer to embed both input sequences. **SelfAttn** [51] utilizes the multi-head attention [105] to encode both input sequences of tokens, and has proven effective on multiple types of programming languages such as Python and JavaScript. The hyper-parameters of the baselines are defined according to the original papers [12, 41, 51]. During implementing CODEnn, NeuralBoW, RNN, CONV, CONVSelf, and SelfAttn, we directly utilized the released code; while for UNIF, we tried our best to replicate the code according to the paper and will make the replication publicly available.

3.4 Experimental Results

In this section, we present the evaluation results, including the main results, parameter analysis, case studies, and error analysis.

3.4.1 Main Results

Involving semantic dependency embeddings increases the code search performance. Table 3.5 and 3.6 illustrate the evaluation results compared with the baseline models. As can be seen, CRaDLe presents the best performance compared with all the baseline models, increasing the performance of 36.38% in terms of R@1, 17.13% in terms of R@5, 12.54% in terms of R@10 and 25.26% in terms of MRR at least on the dataset of CodeSearchNet. CRaDLe can achieve the improvement of the performance at least 22.34%, 22.51%, 21.54%, and 21.79% in R@1, R@5, R@10, and MRR on the dataset of Code2Seq, respectively. This indicates that CRaDLe can rank the correct answer at the top more accurately when given a natural language query. The improvement on R@1 is the most significant among all the metrics in our proposed model, which is over 20% in both datasets. R@1 is the metric that concerns most by programmers since they prefer to use the code search system which can return the best results first. The higher MRR score further verifies the effectiveness of CRaDLe. The difference between CRaDLe and the baseline models is the code representation strategy, which shows the effectiveness of the semantic dependency embeddings for code search.

Attention mechanism can be helpful for effective code search. By comparing CONV with CONVSelf, we can ob-

serve that with the attention mechanism integrated, CONV presents a better performance than the pure CONV model on both datasets. For example, CONVSelf increases the accuracy of CONV by 20.21% and 15.37% in terms of $R@1$ and MRR on the CodeSearchNet dataset, respectively. A similar result also appears on the Code2Seq dataset. The results imply the effectiveness of the attention mechanism on the code search task. We also compared with the performance of the $\text{CRaDLe}_{\text{maxpooling}}$ where the attention mechanism is replaced with the max pooling strategy [61]. As can be seen in Table 3.5 and Table 3.6, CRaDLe with the attention mechanism involved outperforms the CRaDLe with max pooling strategy integrated on both datasets, which further demonstrates the effectiveness of the attention mechanism on the task.

CRaDLe shows better generalizability than baseline models. As can be observed from Table 3.5 and Table 3.6, one baseline model's extraordinary performance on a specific dataset can not transfer to other datasets. For example, SelfAttn achieves the best performance among all the baselines on the CodeSearchNet dataset with respect to $R@1$ but performs worse than NeuralBoW on the Code2seq dataset. Compared with the baselines, CRaDLe presents the best performance on both datasets, which can explicate the good generalizability of CRaDLe.

3.4.2 Parameter Analysis

In this section, we will discuss how the hyperparameters affect the performance of CRaDLe. Three hyperparameters are analyzed, including the number of hidden units in LSTMs, the maximum number of considered statements in the code, and

Approach	R@1	R@5	R@10	MRR
CODEnn	0.367	0.573	0.652	0.465
UNIF	0.379	0.615	0.706	0.490
NeuralBoW	0.521	0.747	0.807	0.622
RNN	0.556	0.772	0.832	0.654
CONV	0.475	0.703	0.776	0.579
CONVSelf	0.571	0.788	0.845	0.668
SelfAttn	0.580	0.786	0.840	0.673
$\text{CRaDLe}_{\text{maxpooling}}$	0.777	0.914	0.946	0.838
CRaDLe	**0.791**	**0.923**	**0.951**	**0.843**

Table 3.5: Comparison results with baseline models on the CodeSearchNet dataset. The best results are highlighted in **bold** fonts.

the maximum number of considered tokens in each statement. Figure 3.4 and Figure 3.5 depict the results of the parameter analysis.

Hidden units in LSTMs

As shown in Figure 3.4(a) and Figure 3.5(a), all the metric values present an increasing trend as the number of hidden units grows. The phenomenon is understandable since more hidden units imply that the model has more parameters to learn and can extract more knowledge from the same input. We can also observe that for each doubling of the number of hidden units, the growth rates of the R@1 scores are 1.9%, 0.54%, and 0.41% respectively on the CodeSearchNet dataset. The trend is identical for the Code2Seq dataset. So we can summarize that with an increasing number of the hidden units, the model performance would increase but the increasing rates show a declining tendency. Due to the limitation of the computing source and the marginal enhancement when the number of hidden units is

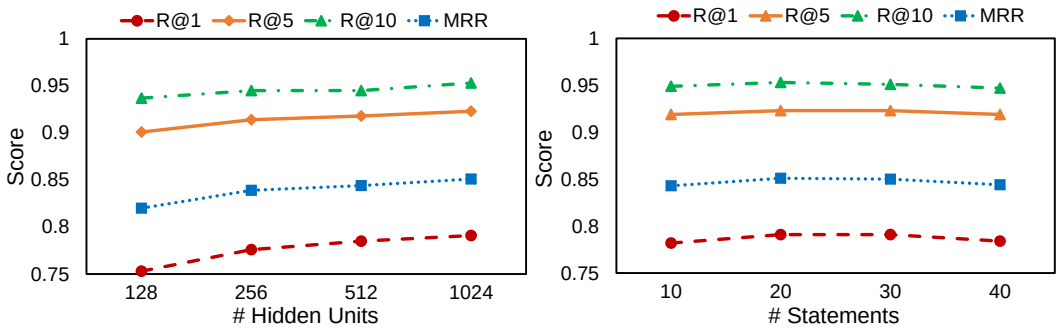
Approach	R@1	R@5	R@10	MRR
CODEnn	0.330	0.532	0.617	0.427
UNIF	0.380	0.588	0.668	0.478
NeuralBoW	0.546	0.693	0.738	0.615
RNN	0.438	0.623	0.688	0.526
CONV	0.425	0.584	0.645	0.502
CONVSelf	0.470	0.642	0.700	0.552
SelfAttn	0.525	0.683	0.731	0.599
$\text{CRaDLe}_{\text{maxpooling}}$	0.664	0.843	0.892	0.745
CRaDLe	**0.668**	**0.849**	**0.897**	**0.749**

Table 3.6: Comparison results with baseline models on the Code2seq dataset. The best results are highlighted in **bold** fonts.

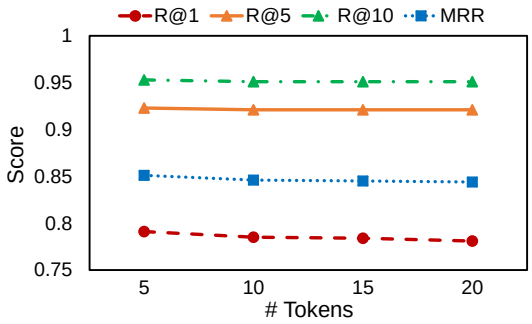
larger than 1,024, we chose 1,024 as the number of hidden units for our experiment.

Maximum statements in code

Figure 3.4(b) and Figure 3.5(b) illustrate the variations of the model performance as the maximum number of considered statements increases. We can observe that the metrics achieve the highest values when the number equals 20 and manifest a declining trend as the statement number further increases. As can be found in Table 3.3 and Table 3.4, the median numbers of the statements in both CodeSearchNet and Code2seq datasets are 7, with the average at around 10. Thus, more statements considered would not be beneficial for capturing the code semantics for most code snippets. In the experiment, we set the maximum number of considered statements in code as 20.



(a) # hidden units in LSTMs (b) # maximum statements in code

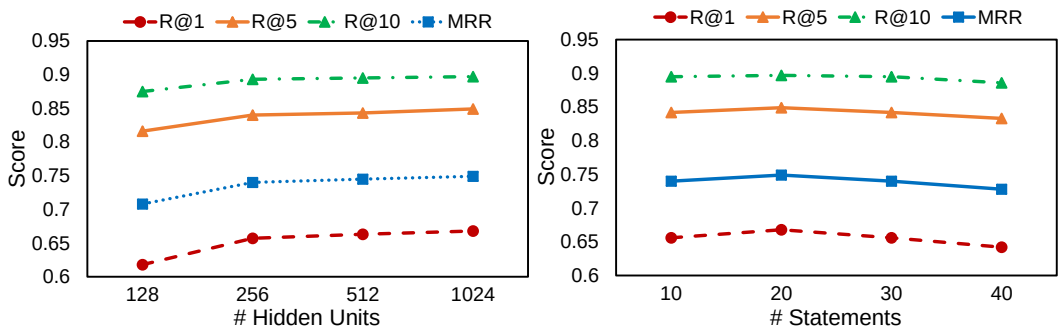


(c) # maximum tokens in statement

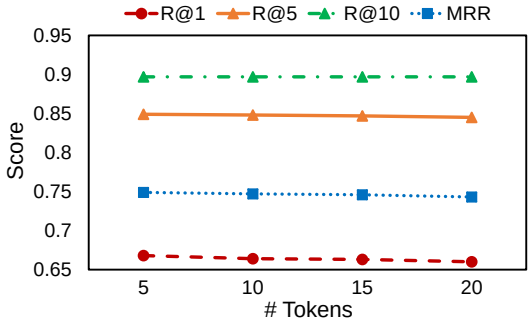
Figure 3.4: Parameter sensitivity study for CodeSearchNet.

 # Maximum tokens in statement

The impact of different maximum numbers of involved tokens in a statement is shown in Figure 3.4(c) and Figure 3.5(c). We can find that when the involved token number increases, the performance presents a downward trend. According to Table 3.3 and Table 3.4, the average number of tokens in the statements is ∼3. So with more tokens recognized, the model could not learn more knowledge of the code snippets. In the experiment, to balance the model performance with the number of tokens considered, we define the maximum number of the tokens in a statement as 5.



(a) # hidden units in LSTMs (b) # maximum statements in code



(c) # maximum tokens in statement

Figure 3.5: Parameter sensitivity study for Code2Seq.

3.4.3 Ablation Study

In the ablation study, we validate the contribution of data dependency or control dependency to CRaDLe and the effectiveness of combining both dependency types. Table 3.7 and Table 3.8 show the results of the ablation study on the datasets of CodeSearchNet and Code2seq, respectively. $\text{CRaDLe}_{\text{Full}}$ represents the model utilizes both data dependency and control dependency, $\text{CRaDLe}_{\text{DataDependency}}$ represents the model only employs data dependency and $\text{CRaDLe}_{\text{ControlDependency}}$ represents the model only utilizes control dependency.

From the results, we can find that the performance of the model that only utilizes data dependency is very close to the

Approach	R@1	R@5	R@10	MRR
$\text{CRaDLe}_{\text{Full}}$	0.791	0.923	0.951	0.843
$\text{CRaDLe}_{\text{DataDependency}}$	0.779	0.910	0.946	0.840
$\text{CRaDLe}_{\text{ControlDependency}}$	0.785	0.918	0.950	0.845

Table 3.7: Ablation study on the CodeSearchNet dataset.

Approach	R@1	R@5	R@10	MRR
$\text{CRaDLe}_{\text{Full}}$	0.668	0.849	0.897	0.749
$\text{CRaDLe}_{\text{DataDependency}}$	0.645	0.827	0.880	0.724
$\text{CRaDLe}_{\text{ControlDependency}}$	0.645	0.828	0.882	0.730

Table 3.8: Ablation study on the Code2seq dataset.

performance of the model with only control dependency, which shows that the importance of data dependency and control dependency is relatively equivalent under our implementation. However, we can find that the model that contains both data dependency and control dependency outperforms the model that only contains one dependency type, especially in terms of the R@1 metric. The results indicate that the combination of data dependency and control dependency is beneficial for effective code search.

```
def logs(self, prefix='worker'):
    logs = []
    logs += [('success_rate', np.mean(self.success_history))]
    if self.compute_Q:
        logs += [('mean_Q', np.mean(self.Q_history))]
    logs += [('episode', self.n_episodes)]

    if prefix != '' and not prefix.endswith('/'):
        return [(prefix + '/' + key, val) for key, val in logs]
    else:
        return logs
```

Code Listing 3.2: Successful case 1.

```
def tile_images(img_nhwc):
    img_nhwc = np.asarray(img_nhwc)
    N, h, w, c = img_nhwc.shape
    H = int(np.ceil(np.sqrt(N)))
    W = int(np.ceil(float(N)/H))
    img_nhwc = np.array(list(img_nhwc) + [img_nhwc[0]*0 for _ in
    ↪  range(N, H*W)])
    img_HWhwc = img_nhwc.reshape(H, W, h, w, c)
    img_HhWwc = img_HWhwc.transpose(0, 2, 1, 3, 4)
    img_Hh_Ww_c = img_HhWwc.reshape(H*h, W*w, c)
    return img_Hh_Ww_c
```

Code Listing 3.3: Successful case 2.

3.4.4 Case Studies

Listing 3.2 shows our predicted code snippet for the query "*Generates a dictionary that contains all collected statistics*". We can find that our predicted result correctly matches the given query. Although no overlapping words exist between the code and query, CRaDLe could capture that the code tokens such as `rate` and `compute` are semantically related to the query word "*statistics*". Besides, since the semantically related tokens mainly appear in lines 3, 4, and 5, and do not span the entire code, we guess that the involved dependency information helps establish the relationships among the statements.

Listing 3.3 shows another predicted code snippet that accurately matches the given query "*Tile N images into one big PxQ image (P, Q)*". Clearly, the function name contains the keywords in the query, e.g., "*tile*" and "*images*". Moreover, the core idea of this query is to tile N images into one image, essentially related to matrix operations. As shown in the Listing 3.3,

the code contains tokens associated with matrix transformation such as `reshape` and `transpose`. So with statement-level tokens explicitly incorporated, CRaDLe could well catch the code functionality.

Overall, the above two examples indicate that CRaDLe can accurately capture the code semantics with the statement-level dependency and semantic information integrated.

```
def transform_matrix_offset_center(matrix, y, x):
    o_x = (x - 1) / 2.0
    o_y = (y - 1) / 2.0
    offset_matrix = np.array([[1, 0, o_x], [0, 1, o_y], [0, 0, 1]])
    reset_matrix = np.array([[1, 0, -o_x], [0, 1, -o_y], [0, 0, 1]])
    transform_matrix = np.dot(np.dot(offset_matrix, matrix),
    ↪  reset_matrix)
    return transform_matrix

```

Code Listing 3.4: Failure case 1.

```
def succ_key(self, key, default=_sentinel):
        item = self.succ_item(key, default)
        return default if item is default else item[0]
```

Code Listing 3.5: Failure case 2.

3.4.5 Error Analysis

Although our model returns correct code snippets in most cases, we still notice that our model fails under the following two particular circumstances.

Code Containing Complex Mathematical Logic

Listing 3.4 provides a failure case where the code contains complex mathematical logic. The description corresponding to the code is "*Convert directly the matrix from Cartesian coordinates (the origin in the middle of image) to Image coordinates (the origin on the top-left of image)*", which includes some mathematical concepts such as "*Cartesian coordinates*". Nevertheless, no words related to the mathematical concepts appear in the code. Less knowledge learned about the mathematical terminology renders the model harder to capture the semantic relevance between the code and natural language. Future work can incorporate external knowledge such as API documentation or Wikipedia to enhance the understanding of the mathematical concepts.

Code Containing Function Invocation

We also find that the proposed model may fail to capture the code semantics when the code involves function invocation but the details of the invoked function are missing. Listing 3.5 illustrates such an example, and the corresponding description is "*Get successor to key, raises KeyError if a key is max key or key does not exist*". As can be seen in the code example, the execution results strongly rely on the invoked function `succ_item()`, however, the implementation of the invoked function is not detailed. In this case, the code semantics are difficult to be fully captured by the model, leading to failure.

3.5 Discussion

3.5.1 Dependency Embedding Approach

In this section, we design another method for representing the dependency information. Specifically, we enrich the dependency matrix with the semantics of the tokens at the statement level. The statement-level dependency embedding is calculated as below:

$$\begin{aligned} \mathbf{p}_i &= \frac{\sum_j \mathbf{t}_j \upsilon_{ij}}{\max\left(1, \sum_j \upsilon_{ij}\right)}, \forall i = 1, 2, ..., l, \\ \mathbf{P} &= [\mathbf{p}_1, ..., \mathbf{p}_{(l)}], \end{aligned} \tag{3.10}$$

where $\mathbf{t}_j$ represents the statement-level token embedding for j-th statement, which is calculated via Equation 3.1. υ_{ij} indicates whether the i-th statement has a data/control dependency on the j-th statement and $\mathbf{p}_i$ is the new dependency embedding.

We evaluated the performance of new dependency embedding methods on the datasets of CodeSearchNet and Code2seq, as shown in Table 3.9.

From the table, we can find that the new strategy for encoding the dependency information outperforms our original approach in terms of the R@1 and MRR metrics for both datasets. The results indicate that the new approach for the dependency embedding may be more effective than the original approach for the task.

Graph neural networks (GNNs) are also a potential way to represent the dependency between different statements in one code snippet. However, using GNNs for representing the semantic dependency of code is beyond the scope of the work, since the assumption of GNNs that adjacent nodes share similar

Dataset	Approach	R@1	R@5	R@10	MRR
CodeSearchNet	$\text{CRaDLe}_{\text{original}}$	0.791	**0.923**	**0.951**	0.843
	$\text{CRaDLe}_{\text{new}}$	**0.794**	0.920	0.949	**0.851**
Code2seq	$\text{CRaDLe}_{\text{original}}$	0.668	0.849	0.897	0.749
	$\text{CRaDLe}_{\text{new}}$	**0.676**	**0.852**	**0.899**	**0.756**

Table 3.9: Comparison results with our original models. The best results are highlighted in **bold** fonts.

semantics no longer holds for the control dependency information, and it would be more challenging to encode the semantic dependency information through GNNs [64, 115].

3.6 Threats to Validity

After careful analysis, we have identified the following potential threats to the validity.

3.6.1 Threats to External Validity

Since we have only chosen two public datasets to evaluate the effectiveness of CRaDLe, the results obtained from these datasets may not accurately reflect the performance of our approach in real-world scenarios. Besides, since the tool to extract the program dependency graph from the source code varies depending on the specific programming language, we only implement the tool for the programming language of Python. The different programming languages may affect the performance of our proposed approach.

3.6.2 Threats to Internal Validity

Although we implemented the tool to extract the program dependency graph for the programming language of Python, we still cannot guarantee the extracted program dependency graph is 100% correct. The wrongly extracted program dependency graph may affect the retrieval performance of our proposed approach.

3.7 Summary

In this chapter, we have proposed a novel deep neural network named CRaDLe for code retrieval. According to our knowledge, CRaDLe is the first deep learning model that utilizes the program dependency information for the task. CRaDLe learns the code representations with the semantic dependency information combined. Specifically, the dependency information and statement-level tokens are jointly embedded for learning code semantics. Finally, CRaDLe learns unified representations for both code and natural language queries. The experiment results have shown that CRaDLe outperforms the state-of-the-art approaches and that semantic dependency learning is helpful for effective code retrieval.

□ End of chapter.

Chapter 4

Accelerating Code Retrieval with Deep Hashing and Code Classification

In this chapter, we investigate deep hashing for code retrieval acceleration. With the development of deep learning, it has become the mainstream to adopt deep learning-based approaches in the task of code retrieval. However, previous methods focused on retrieval accuracy but lacked attention to the efficiency of the retrieval process. To address this problem, we propose a novel method, CoSHC, to accelerate code retrieval with deep hashing and code classification, aiming to perform efficient code retrieval without sacrificing too much accuracy. Specifically, we cluster the representation vectors into different categories and generate binary hash codes for both source code and queries. Then, our proposed model gives the normalized prediction probability of each category for the given query. The number of code candidates for the given query in each category will be decided according to the probability predicted from our proposed model. Then, we conduct a comprehensive experimental evaluation on public benchmarks. The results demonstrate that CoSHC can

significantly improve retrieval efficiency while preserving almost the same performance as the baseline models.

4.1 Introduction

Code reuse is a common practice during the software development process. It improves programming productivity as developers' time and energy can be saved by reusing existing code. According to previous studies [8, 78], many developers tend to use natural language to describe the functionality of desired code snippets and search the Internet/code corpus for code reuse.

Many code retrieval approaches [8, 28, 78, 79] have been proposed over the years. With the rapid growth of open source code bases and the development of deep learning technology, recently deep learning-based approaches have become popular for tackling the code retrieval problem [39, 41, 51]. Some of these approaches adopt neural network models to encode source code and query descriptions into representation vectors in the same embedding space. The distance between the representation vectors whose original code or description is semantically similar should be small. Other approaches [28, 33, 43] regard the code retrieval task as a binary classification task and calculate the probability of code matching the query.

In the past, deep learning-based methods focused on retrieval accuracy but lacked attention to the efficiency of retrieval on large-scale code corpus. However, both types of these deep learning-based approaches directly rank all the source code snippets in the corpus during searching, which will incur a large amount of computational cost. For the approaches that separately encode code and description representation vectors, the

similarity of the target query vector with all code representation vectors in the corpus needs to be calculated for every single retrieval. In order to pursue high retrieval accuracy, a high dimension is often set for the representation vectors. For example, in CodeBERT, the dimension of the final representation vector is 768. The similarity calculation between a pair of code and query vectors will take 768 multiplications and 768 additions between two variables with the double data type. The total calculation of a single linear scan for the whole code corpus containing around 1 million code snippets is extremely large - around 1 billion times multiplications and additions. As for the approaches adopting binary classification, there are no representation vectors stored in advance and the inference of the target token sequence with all the description token sequences needs to be done in real-time for every single retrieval. Due to the large number of parameters in the current deep learning models, the computation cost will be significant.

Hashing is a promising approach to improve retrieval efficiency and is widely adopted in other retrieval tasks such as image-text search and image-image search. Hashing techniques can convert high dimensional vectors into low dimensional binary hash code, which greatly reduces the cost of storage and calculation [77]. Hamming distance between two binary hash codes can also be calculated in a very efficient way by running XOR instruction on the modern computer architectures [112]. However, the performance degradation is still not avoidable during the conversion from representation vectors to binary hash codes even the state-of-the-art hashing models are adopted. The tolerance of performance degradation from most users is quite low and many of them are willing to sweep the performance with

efficiency. In order to preserve the performance of the original code retrieval models that adopt bi-encoders for the code-query encoding as much as possible, we integrate deep hashing techniques with code classification, which could mitigate the performance degradation of the hashing model in the recall stage by filtering out the irrelevant data.

Specifically, in this chapter, we propose a novel approach **CoSHC** (Accelerating Semantic Code Search with Deep Hashing and Code Classification) for accelerating the retrieval efficiency of deep learning-based code retrieval approaches. CoSHC firstly clusters the representation vectors into different categories. It then generates binary hash codes for both source code and queries according to the representation vectors from the original models. Finally, CoSHC gives the normalized prediction probability of each category for the given query, and then CoSHC will decide the number of code candidates for the given query in each category according to the probability. Comprehensive experiments have been conducted to validate the performance of the proposed approach. The evaluation results show that CoSHC can preserve more than 99% performance of most baseline models. We summarize the main contributions of this chapter as follows:

- We propose a novel approach, CoSHC, to improve the retrieval efficiency of previous deep learning-based approaches. CoSHC is the first approach that adopts the recall and re-rank mechanism with the integration of code clustering and deep hashing to improve the retrieval efficiency of deep learning-based code retrieval models.
- We conduct comprehensive experimental evaluations on public benchmarks. The results demonstrate that CoSHC can greatly

improve the retrieval efficiency meanwhile preserving almost the same performance as the baseline models.

4.2 Methodology

We propose a general framework to accelerate existing Deep code retrieval models by decoupling the search procedure into a recall stage and a re-rank stage. Our main technical contribution lies in the recall stage. Figure 4.1 illustrates the overall framework of the proposed approach. CoSHC consists of two components, i.e., Offline and Online. In the Offline stage, we take the code and description embeddings learned in the given code retrieval model as input and learn the corresponding hash codes by preserving the relations between the code and description embeddings. In the Online stage, we recall a candidate set of code snippets according to the Hamming distance between the query and code, and then we use the original code retrieval model to re-rank the candidates.

4.2.1 Offline Stage

Multiple Code Hashing Design with Code Classification Module

Since the capacity of binary hashing space is very limited compared to Euclidean space, the Hamming distance between similar code snippets will be too small to be distinguishable if we adopt a single Hashing model. To be specific, we cluster the codebase using the K-Means algorithm with the code embeddings learned from the given code retrieval model. The source code whose representation vectors are close to each other will be classified into the same category after the clustering.

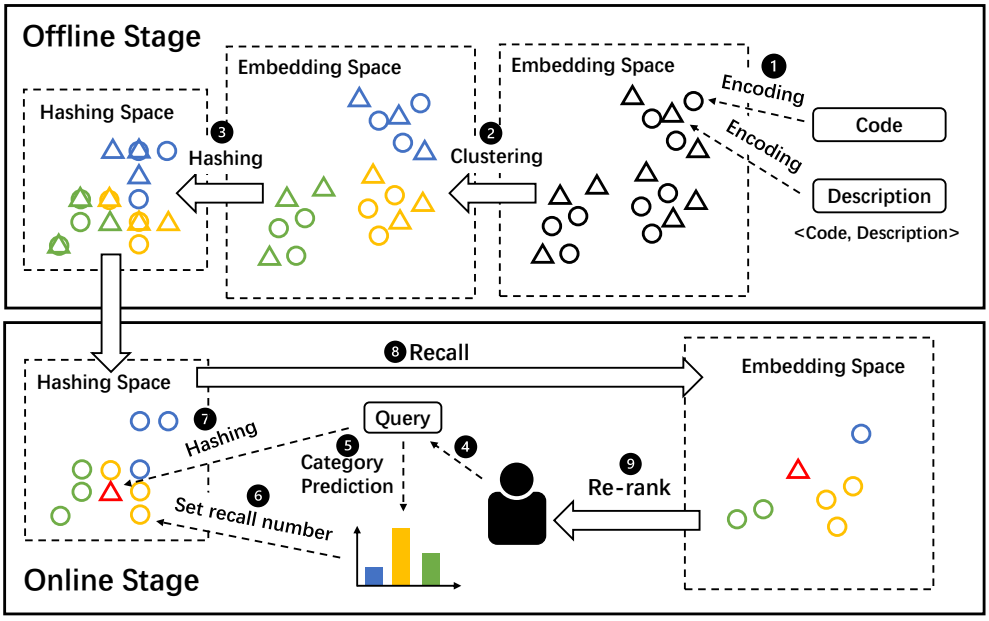


Figure 4.1: Overview of the proposed CoSHC. ① Encoding the code token sequence and description token sequence via original code retrieval models. ② Clustering the code representation vectors into several categories. ③ Converting the original code representation vectors into binary hash codes.⑤ ⑥ Predicting the category of the query given by users and set the number of code candidates for different categories. ⑦ Converting the input query into binary hash code. ⑧ Recall the code candidates according to the Hamming distance and the number of code candidates for each category. ⑨ Re-ranking all the code candidates according to the cosine similarity between the input query description vectors and code candidates' representation vectors and return the results to the user.

Deep Hashing Module

The deep hashing module aims at generating the corresponding binary hash codes for the embeddings of code and description from the original code retrieval model. Figure 4.2 illustrates the framework of the deep hashing module. To be specific, three fully connected (FC) layers with $\tanh(\cdot)$ activation function are adopted to replace the output layer in the original code retrieval model to convert the original representation vectors into a soft

binary hash code.

The objective of the deep hashing module is to force the Hamming distance between hashing representations of code pairs and description pairs approaching the Euclidean distance between the corresponding embeddings. Thus, we need to calculate the ground truth similarity matrix between code pairs and description pairs first. For performance consideration, we calculate the similarity matrix within a mini-batch.

To construct such a matrix, we first define the code representation vectors and the description representation vectors in the original code retrieval model as $V_C = \{v_c^{(1)}, ..., v_c^{(n)}\}$ and $V_D = \{v_d^{(1)}, ..., v_d^{(n)}\}$, respectively. V_C and V_D represent the representation vectors matrix for the entire batch, while $v_c^{(i)}$ and $v_d^{(i)}$ represent the representation vector for the single code snippet or query. After normalizing V_C, V_D to $\hat{V}_C$, $\hat{V}_D$ with l_2-norm, we can calculate the code similarity matrices $S_C = \hat{V}_C \hat{V}_C^T$ and summary similarity matrices $S_D = \hat{V}_D \hat{V}_D^T$ to describe the similarity among code representation vectors and summary representation vectors, respectively. In order to integrate the similarity information in both S_C and S_D, we combine them with a weighted sum:

$$\tilde{S} = \beta S_C + (1 - \beta) S_D, \beta \in [0, 1], \tag{4.1}$$

where β is the weight parameter. Since the pairwise similarity among the code representation vectors and description representation vectors still cannot comprehensively present the distribution condition of them in the whole embedding space, we involve a matrix $\tilde{S}\tilde{S}^T$ to describe a high order neighborhood similarity that two vectors with high similarity should also have the close similarity to other vectors. Finally, we utilize a weighted

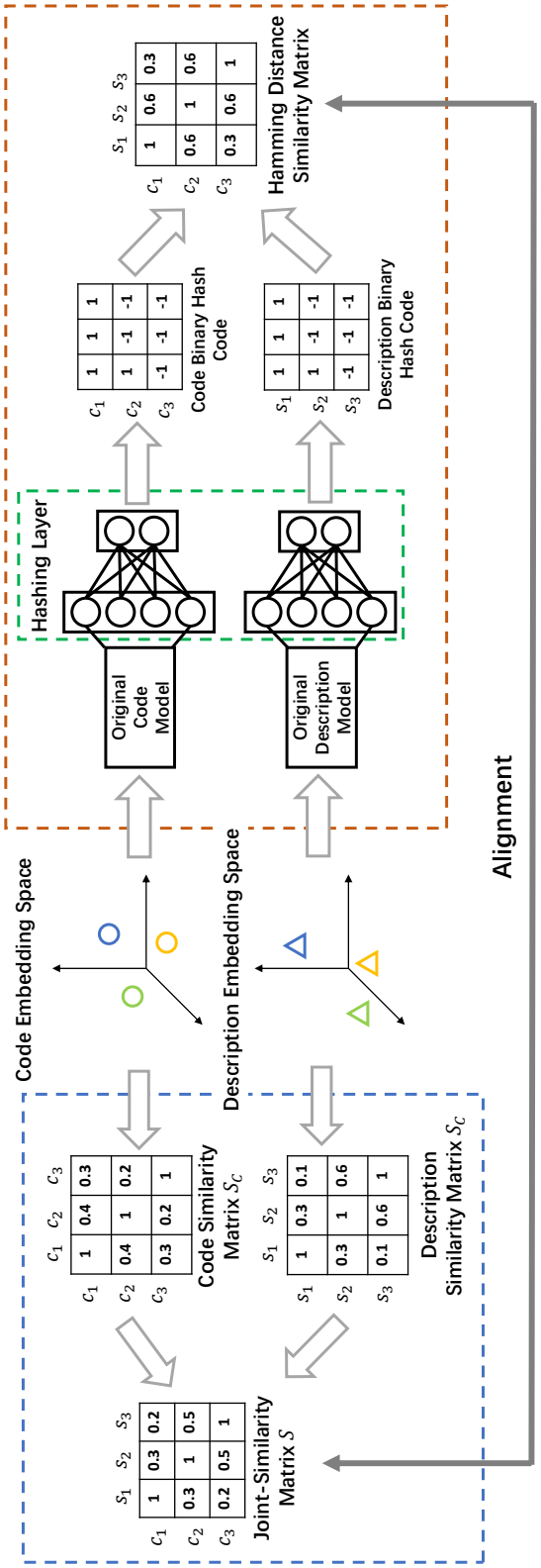


Figure 4.2: Architecture of the hashing module. The original representation vectors will be utilized for the joint-similarity matrix construction at first. Then the joint-similarity matrix will be utilized as the labels for training binary hash codes generation. The training objective is to make the Hamming distance similarity matrix to be identical as the joint-similarity matrix.

equation to combine both of these two matrices as follows:

$$S = (1-\eta)\tilde{S} + \eta\frac{\tilde{S}\tilde{S}^T}{m}, \tag{4.2}$$

where η is a hyper-parameter and m is the batch size which is utilized to normalize the second term in the equation. Since we hope the binary hash codes of the source code and its corresponding description are the same, we replace the diagonal elements in the similarity matrix with one. The final high-order similarity matrix is:

$$S_{F_{ij}} = \begin{cases} 1, & i = j \\ S_{ij}, & \text{otherwise} \end{cases} \tag{4.3}$$

Binary Hash Code Training

We propose to replace the output layer of the original code retrieval model with three FC layers with $\tanh(\cdot)$ activate function. We define the trained binary hash code for code and description as $B_C = \{b_c^{(1)}, ..., b_c^{(n)}\}$ and $B_D = \{b_d^{(1)}, ..., b_d^{(n)}\}$, respectively. To ensure that the relative distribution of binary hash codes is similar to the distribution of representation vectors in the original embedding space, the following equation is utilized as the loss function of the deep hashing module:

$$\begin{aligned} \mathscr{L}(\theta) = \min_{B_C, B_D} & \| \min(\mu S_F, 1) - \frac{B_C B_D^T}{d} \|_F^2 \\ & + \lambda_1 \| \min(\mu S_F, 1) - \frac{B_C B_C^T}{d} \|_F^2 \\ & + \lambda_2 \| \min(\mu S_F, 1) - \frac{B_D B_D^T}{d} \|_F^2, \\ & s.t.\ B_C, B_D \in \{-1, +1\}^{m \times d}, \end{aligned} \tag{4.4}$$

where θ are model parameters, μ is the weighted parameters to adjust the similarity score between different pairs of code and description, λ_1, λ_2 are the trade-off parameters to weight different terms in the loss function, and d is the dimension of the binary hash code generated by this deep hashing module. These three terms in the loss function are adopted to restrict the similarity among binary hash codes of the source codes, the similarity among binary hash codes of the descriptions, and the similarity between the binary hash codes of the source code and description, respectively.

Note that we adopt $B_C B_D^T/d$ to replace $\cos(B_C, B_D)$ because $\cos(B_C, B_D)$ only measures the angle between two vectors but neglects the length of the vectors, which makes $\cos(B_C, B_D)$ can still be a very large value even the value of every hash bits is close to zero. Unlike $\cos(B_C, B_D)$, $B_C B_D^T/d$ can only achieve a high value when every bit of the binary hash code is 1 or -1 since the value of $B_C B_D^T/d$ will be close to zero if the value of every hash bits is close to zero.

Since it is impractical to impose on the output of the neural network to be discrete values like 1 and -1, we adopt the following equation to convert the output of the deep hashing module to be strict binary hash code:

$$B = \text{sgn}(H) \in \{-1, +1\}^{m \times d}, \tag{4.5}$$

where H is the output of the last hidden layer without the activation function in the deep hashing module and $\text{sgn}(\cdot)$ is the sign function and the output of this function is 1 if the input is positive and the output is -1 otherwise.

However, the gradient of the sign function will be zero in backward propagation which will induce the vanishing gradients problem and affect model convergence. To address this

problem, we follow the previous research [13, 49] and adopt a scaling function:

$$B = \tanh(\alpha H) \in \{-1, +1\}^{m \times d}, \tag{4.6}$$

where α is the parameter that is increased during the training. The function of $\tanh(\alpha H)$ is an approximate equation of $\text{sgn}(H)$ when α is large enough. Therefore, the output of Eq. 4.6 will finally be converged to 1 or -1 with the increasing of α during the training and the above problem is addressed.

4.2.2 Online Stage

Recall and Re-rank Mechanism

The incoming query from users will be fed into the description category prediction module to calculate the normalized probability distribution of categories at first. Then the number of code candidates R_i for each category i will be determined according to this probability distribution. The Hamming distance between the hash code of the given query and all the code inside the database will be calculated. Then code candidates will be sorted by Hamming distance in ascending order and the top R_i code candidates in each category i will be recalled. In the re-rank step, the original representation vectors of these recalled code candidates will be retrieved and utilized for the cosine similarity calculation. Finally, code snippets will be returned to users in descending order of cosine similarity.

Description Category Prediction Module

The description category prediction module aims to predict the category of source code that meets the user's requirement according to the given natural language description. The model

adopted for category prediction is the same as the original code retrieval model, except that the output layer is replaced with a one-hot category prediction layer and the cross-entropy function is adopted as the loss function of the model.

Since the accuracy of the description category prediction module is not perfect, we use the probability distribution of each category instead of the category with the highest predicted probability as the recall strategy for code retrieval. We define the total recall number of source code as N, the normalized predicted probability for each code category as $P = \{p_1, ..., p_k\}$, where k is the number of categories. The recall number of source code in each category is:

$$R_i = \min(\lfloor p_i \cdot (N - k) \rfloor, 1), \ i \in 1, ..., k, \tag{4.7}$$

where R_i is the recall number of source code in category i. To ensure that the proposed approach can recall at least one source code from each category, we set the minimum recall number for a single category to 1.

4.3 Experiments

4.3.1 Dataset

We use two datasets (Python and Java) provided by CodeBERT [33] to evaluate the performance of CoSHC. CodeBERT selects the data from the CodeSearchNet [41] dataset and creates both positive and negative examples of <description, code> pairs. Since all the baselines in our experiments are bi-encoder models, we do not need to predict the relevance score for the mismatched pairs so we remove all the negative examples from the dataset. Finally, we get 412,178 ⟨code, description⟩ pairs as

the training set, 23,107 ⟨code, description⟩ pairs as the validation set, and 22,176 ⟨code, description⟩ pairs as the test set in the Python dataset. We get 454,451 ⟨code, description⟩ pairs as the training set, 15,328 ⟨code, description⟩ pairs as the validation set, and 26,909 ⟨code, description⟩ pairs as the test set in the Java dataset.

4.3.2 Experimental Setup

In the code classification module, we set the number of clusters to 10. In the deep hashing module, we add three fully connected (FC) layers in all the baselines, the hidden size of each FC layer is the same as the dimension of the original representation vectors. Specifically, the hidden size of the FC layer for CodeBERTa, CodeBERT, and GraphCodeBERT is 768. The hidden size of the FC layer for UNIF is 512 and for RNN is 2048. The size of the output binary hash code for all the baselines is 128. The hyperparameters $\beta, \eta, \mu, \lambda_1, \lambda_2$ are 0.6, 0.4, 1.5, 0.1, and 0.1, respectively. The parameter α is the epoch number and will be linearly increased during the training. In the query category prediction module, a cross-entropy function is adopted as the loss function and the total recall number is 100.

The learning rate for CodeBERTa, CodeBERT, and GraphCodeBERT is 1e-5 and the learning rate for UNIF, and RNN is 1.34e-4. All the models are trained via the AdamW algorithm [58].

We train our models on a server with four 4x Tesla V100 w/NVLink and 32GB memory. Each module based on CodeBERT, GraphCodeBERT, and CodeBERTa is trained with 10 epochs and Each module based on RNN and UNIF is trained with 50 epochs. The early stopping strategy is adopted to avoid over-

fitting for all the baselines. The time efficiency experiment is conducted on the server with Intel Xeon E5-2698v4 2.2Ghz 20-core. The programming for evaluation is written in C++ and the program is allowed to use a single thread of CPU.

4.3.3 Baselines

We apply CoSHC on several state-of-the-art and representative baseline models. UNIF [12] regards the code as the sequence of tokens and embeds the sequence of code tokens and description tokens into representation vectors via a fully connected layer with the attention mechanism, respectively. RNN baseline adopts a two-layer bi-directional LSTM [19] to encode the input sequences. CodeBERTa[1] is a 6-layer, Transformer-based model pre-trained on the CodeSearchNet dataset. CodeBERT [33] is a pre-trained model based on Transformer with 12 layers. Similar to CodeBERT, GraphCodeBERT [43] is a pre-trained Transformer-based model with not only token information but also dataflow of the code snippets. As we introduced, the inference efficiency of cross-encoder-based models like CodeBERT is quite low and the purpose of our approach is to improve the calculation efficiency between the representation vectors of code and queries. Here we slightly change the model structure of CodeBERTa, CodeBERT, and GraphCodeBERT. Rather than concatenating code and query together and inputting them into a single encoder to predict the relevance score of the pair, we adopt the bi-encoder architecture for the baselines, which utilize the independent encoder to encode the code and queries into representation vectors, respectively. Also, cosine similarity between the given representation vector pairs is adopted as the

[1] https://huggingface.co/huggingface/CodeBERTa-small-v1

training loss function to replace the cross entropy function of the output relevance score.

4.3.4 Evaluation Metric

$SuccessRate@k$ is widely used by many previous studies [32, 44, 46, 100]. The metric is calculated as follows:

$$SuccessRate@k = \frac{1}{|Q|}\sum_{q=1}^{Q}\delta(FRank_q \leq k), \tag{4.8}$$

where Q denotes the query set and $FRank_q$ is the rank of the correct answer for query q. If the correct result is within the top k returning results, $\delta(FRank_q \leq k)$ returns 1, otherwise it returns 0. A higher $R@k$ indicates better performance.

4.4 Experimental Results

In this section, we present the experimental results and evaluate the performance of CoSHC from the aspects of retrieval efficiency, overall retrieval performance, and the effectiveness of the internal classification module.

4.4.1 RQ1: How much faster is CoSHC than the original code retrieval models?

Table 4.1 illustrates the results of the efficiency comparison between the original code retrieval models and CoSHC. Once the representation vectors of code and description are stored in the memory, the retrieval efficiency mainly depends on the dimension of representation vectors rather than the complexity of the original retrieval model. Therefore, we select CodeBERT as the baseline model to illustrate efficiency comparison. Since the

	Python	**Java**
	Total Time	
CodeBERT	572.97s	247.78s
CoSHC	33.87s (↓94.09%)	15.78s (↓93.51%)
	(1) Vector Similarity Calculation	
CodeBERT	531.95s	234.08s
CoSHC	14.43s (↓97.29%)	7.25s (↓96.90%)
	(2) Array Sorting	
CodeBERT	41.02s	13.70s
CoSHC	19.44s (↓53.61%)	8.53s (↓37.74%)

Table 4.1: Time Efficiency of CoSHC.

code retrieval process in both approaches contains vector similarity calculation and array sorting, we split the retrieval process into these two steps to calculate the time cost.

In the vector similarity calculation step, CoSHC reduces 97.29% and 96.90% of time cost in the dataset of Python and Java respectively, which demonstrates that the utilization of binary hash code can effectively reduce vector similarity calculation cost in the code retrieval process.

In the array sorting step, CoSHC reduces 53.61% and 37.74% of time cost in the dataset of Python and Java, respectively. The classification module makes the main contribution to the improvement of sorting efficiency. The sorting algorithm applied in both the original code retrieval model and CoSHC is quick sort, whose time complexity is $O(n\log n)$. The classification module divides a large code dataset into several small code datasets, reducing the average time complexity of sorting to $O(n\log\frac{n}{m})$. The reason why the improvement of sorting in the Java dataset is not as significant as in the Python dataset is

Model	Python			Java		
	R@1	**R@5**	**R@10**	**R@1**	**R@5**	**R@10**
UNIF	0.071	0.173	0.236	0.084	0.193	0.254
$\text{CoSHC}_{\text{UNIF}}$	**0.072 (↑1.4%)**	**0.177 (↑2.3%)**	**0.241 (↑2.1%)**	**0.086 (↑2.4%)**	**0.198 (↑2.6%)**	**0.264 (↑3.9%)**
−w/o classification	0.071 (0.0%)	0.174 (↑0.6%)	0.236 (0.0%)	0.085 (↑1.2%)	0.193 (0.0%)	0.254 (0.0%)
−one classification	0.069 (↓2.8%)	0.163 (↓5.8%)	0.216 (↓8.5%)	0.083 (↓1.2%)	0.183 (↓5.2%)	0.236 (↓7.1%)
−ideal classification	0.077 (↑6.9%)	0.202 (↑16.8%)	0.277 (↑17.4%)	0.093 (↑10.7%)	0.222 (↑15.0%)	0.296 (↑16.5%)
RNN	0.111	0.253	0.333	0.073	0.184	0.250
$\text{CoSHC}_{\text{RNN}}$	**0.112 (↑0.9%)**	**0.259 (↑2.4%)**	**0.343 (↑5.0%)**	**0.076 (↑4.1%)**	**0.194 (↑5.4%)**	**0.265 (↑6.0%)**
−w/o classification	**0.112 (↑0.9%)**	0.254 (↑0.4%)	0.335 (↑0.6%)	0.073 (0.0%)	0.186 (↑1.1%)	0.253 (↑1.2%)
−one classification	**0.112 (↑0.9%)**	0.243 (↓4.0%)	0.311 (↓6.6%)	0.075 (↑2.7%)	0.182 (↓1.1%)	0.240 (↓4.0%)
−ideal classification	0.123 (↑10.8%)	0.289 (↑14.2%)	0.385 (↑15.6%)	0.084 (↑15.1%)	0.221 (↑20.1%)	0.302 (↑20.8%)
CodeBERTa	0.124	0.250	0.314	0.089	0.203	0.264
$\text{CoSHC}_{\text{CodeBERTa}}$	**0.123 (↓0.8%)**	**0.247 (↓1.2%)**	**0.309 (↓1.6%)**	**0.090 (↑1.1%)**	**0.210 (↑3.4%)**	**0.272 ((↑3.0%)**
−w/o classification	0.122 (↓1.6%)	0.242 (↓3.2%)	0.302 (↓3.8%)	0.089 (0.0%)	0.201 (↓1.0%)	0.258 (↓2.3%)
−one classification	0.116 (↓6.5%)	0.221 (↓11.6%)	0.271 (↓13.7%)	0.085 (↓4.5%)	0.189 (↓6.9%)	0.238 (↓9.8%)
−ideal classification	0.135 (↑8.9%)	0.276 (↑10.4%)	0.346 (↑10.2%)	0.100 (↑12.4%)	0.235 (↑15.8%)	0.305 (↑15.5%)
CodeBERT	0.451	0.683	0.759	0.319	0.537	0.608
$\text{CoSHC}_{\text{CodeBERT}}$	**0.451 (0.0%)**	**0.679 (↓0.6%)**	**0.750 (↓1.2%)**	**0.318 (↓0.3%)**	**0.533 (↓0.7%)**	**0.602 (↓1.0%)**
−w/o classification	0.449 (↓0.4%)	0.673 (↓1.5%)	0.742 (↓2.2%)	0.316 (↓0.9%)	0.527 (↓1.9%)	0.593 (↓2.5%)
−one classification	0.425 (↓5.8%)	0.613 (↓10.2%)	0.665 (↓12.4%)	0.304 (↓4.7%)	0.483 (↓10.1%)	0.532 (↓12.5%)
−ideal classification	0.460 (↑2.0%)	0.703 (↑2.9%)	0.775 (↑2.1%)	0.329 (↑3.1%)	0.555 (↑3.4%)	0.627 (↑3.1%)
GraphCodeBERT	0.485	0.726	0.792	0.353	0.571	0.640
$\text{CoSHC}_{\text{GraphCodeBERT}}$	**0.483 (↓0.4%)**	**0.719 (↓1.0%)**	**0.782 (↓1.3%)**	**0.350 (↓0.8%)**	**0.561 (↓1.8%)**	**0.625 (↓2.3%)**
−w/o classification	0.481 (↓0.8%)	0.713 (↓1.8%)	0.774 (↓2.3%)	0.347 (↓1.7%)	0.553 (↓3.2%)	0.616 (↓3.7%)
−one classification	0.459 (↓5.4%)	0.653 (↓10.1%)	0.698 (↓11.9%)	0.329 (↓7.8%)	0.505 (↓11.6%)	0.551 (↓13.9%)
−ideal classification	0.494 (↑1.9%)	0.741 (↑2.1%)	0.803 (↑1.4%)	0.361 (↑2.3%)	0.585 (↑2.5%)	0.649 (↑1.4%)

Table 4.2: Results of code retrieval performance comparison. The best results among the three CoSHC variants are highlighted in **bold** font.

that the size of the Java dataset is much smaller than the size of the Python dataset. However, the combination of the algorithm of divide and conquer and max-heap, rather than quick sort, is widely applied in big data sorting, which can greatly shrink the retrieval efficiency gap between these two approaches. Therefore, the improvement of efficiency in the sorting process will not be as large as what is shown in Table 4.1.

In the overall code retrieval process, the cost time is reduced by 94.09% and 93.51% in the dataset of Python and Java, respectively. Since the vector similarity calculation takes most of the cost time in the code retrieval process, CoSHC still can reduce at least 90% of cost time, which demonstrates the effectiveness of the efficiency improvement in the code retrieval task.

4.4.2 RQ2: How does CoSHC affect the accuracy of the original models?

Table 4.2 illustrates the retrieval performance comparison between the original code retrieval models and CoSHC. We have noticed that the performance of the conventional approaches like BM25 [91] is not good enough. For example, we set the token length for both code and queries as 50, which is the same as the setting in CodeBERT, and apply BM25 to recall the top 100 code candidates for the re-rank step on the Python dataset. BM25 can only retain 99.3%, 95.6%, and 92.4% retrieval accuracy of CodeBERT in terms of R@1, R@5 and R@10 on the Python dataset. Here we only compare the performance of our approach with the original code retrieval models since the purpose of our approach is to preserve the performance of the original code retrieval models. As can be observed, CoSHC can

retain at least 99.5%, 99.0%, and 98.4% retrieval accuracy of most original code retrieval models in terms of R@1, R@5 and R@10 on the Python dataset. CoSHC can also retain 99.2%, 98.2%, and 97.7% of the retrieval accuracy as all original code retrieval baselines in terms of R@1, R@5, and R@10 on the Java dataset, respectively. We can find that CoSHC can retain more than 97.7% of performance in all metrics. R@1 is the most important and useful metric among these metrics since most users hope that the first returned answer is the correct answer during the search. CoSHC can retain at least 99.2% of performance on R@1 in both datasets, which demonstrates that CoSHC can retain almost the same performance as the original code retrieval model.

It is interesting that CoSHC presents a relatively better performance when the performance of the original code retrieval models is worse. $\text{CoSHC}_{\text{CodeBERTa}}$ even outperforms the original baseline model in the Java dataset. $\text{CoSHC}_{\text{RNN}}$ and $\text{CoSHC}_{\text{UNIF}}$ outperform the original model in both Python and Java datasets. The integration of deep learning and code classification with recall makes a contribution to this result. The worse performance indicates more misalignment between the code representation vectors and description representation vectors. Since the code classification and deep hashing will filter out most of the irrelevant codes in the recall stage, some irrelevant code representation vectors that have high cosine similarity with the target description representation vectors are filtered, which leads the improvement in the final retrieval performance.

Model	Python Acc.	Java Acc.
$\text{CoSHC}_{\text{UNIF}}$	0.558	0.545
$\text{CoSHC}_{\text{RNN}}$	0.610	0.535
$\text{CoSHC}_{\text{CodeBERTa}}$	0.591	0.571
$\text{CoSHC}_{\text{CodeBERT}}$	0.694	0.657
$\text{CoSHC}_{\text{GraphCodeBERT}}$	0.713	0.653

Table 4.3: Classification accuracy of the code classification module in each model.

4.4.3 RQ3: Can the classification module help improve performance?

Table 4.2 illustrates the performance comparison between the CoSHC variants which adopt different recall strategies with query category prediction results. $\text{CoSHC}_{\text{w/o classification}}$ represents CoSHC without code classification and description prediction module. $\text{CoSHC}_{\text{one classification}}$ represents the CoSHC variant that recalls $N - k + 1$ candidates in the code category with the highest prediction probability and one in each of the rest categories. $\text{CoSHC}_{\text{ideal classification}}$ is an ideal classification situation we set. Assuming the correct description category is known, $N - k + 1$ candidates are recalled in the correct category, and one candidate is recalled in each of the rest categories. Note that the display of $\text{CoSHC}_{\text{ideal classification}}$ is only to explore the upper threshold of performance improvement of the category prediction module and will not be counted as a variant of CoSHC we compare.

By comparing the performance between $\text{CoSHC}_{\text{ideal classification}}$ and $\text{CoSHC}_{\text{w/o classification}}$, we can find that correct classification can significantly improve retrieval performance. With the ideal category labels, CoSHC can even outperform all baseline models. As mentioned in Section § 4.4.2, code classification can

mitigate the problem of vector pairs misalignment by filtering out wrong candidates whose representation vectors have high cosine similarity with the target representation vectors in the recall stage. The more serious the misalignment problem, the more effective the code classification. That is the reason why the improvement of CoSHC with ground-truth labels on UNIF, RNN, and CodeBERTa is more significant than the improvement of it on CodeBERT and GraphCodeBERT since the retrieval accuracy of former models is much lower than the latter ones. Similar conclusions can also be drawn at the aspect of binary hash code distribution via the comparison between CoSHC and $\text{CoSHC}_{\text{ideal classification}}$ since CoSHC utilizes the distribution of the original representation vectors as the guidance for model training. Therefore, the distribution of binary hash codes will be similar to the distribution of original representation vectors.

Since we have explored the theoretical upper limit of the effectiveness of code classification for code retrieval, the effectiveness of code classification for code retrieval in real applications will be validated. By comparing the experimental results between $\text{CoSHC}_{\text{w/o classification}}$ and $\text{CoSHC}_{\text{one classification}}$, we can find that the performance of CoSHC with predicted labels is even worse than the performance of CoSHC without code classification module. The reason is that the accuracy of description category prediction is far from satisfactory. Table 4.3 illustrates the accuracy of the description category prediction module in all baseline models. We regard the category with the highest probability as the predicted category from the description category prediction module and check whether the module could give a correct prediction. It can be seen that the classification accuracy is not very high (less than 75%). By observing the ex-

perimental results of CoSHC in GraphCodeBERT on the Java dataset, we can also find that low accuracy greatly affects the performance of $\text{CoSHC}_{\text{oneclassification}}$, which makes 7.8%, 11.6%, and 13.9% performance drop in terms of $R@1$, $R@5$, and $R@10$, respectively.

Fortunately, although the description category prediction module cannot accurately tell the exact category to which this description belongs, the module still can give a relatively high predicted probability of the correct category. By comparing the experimental results among all the variants of CoSHC, we can find the performance is increased significantly once the recall strategy is replaced so that the number of code candidates for each category is determined by the normalized prediction probability. CoSHC with the new recall strategy almost achieved the best performance in all metrics on all baseline models. Even on RNN in the Python dataset, CoSHC still achieves the same performance as CoSHC without classification under $R@1$ and achieves similar performance in other metrics. The above experimental results have demonstrated the effectiveness of the adoption of code classification in code retrieval.

4.5 Discussion

In this section, we will discuss the difference between the deep hashing approaches we adopted in our framework and previous deep hashing approaches. Here we select four previous deep hashing approaches, which are DBRC, UGACH, UDCMH, and DJRSH, for the comparison. DBRC utilizes the adaptive Tanh activation function to binarize the representations from the cross-modality and is trained to minimize the reconstruc-

tion error based on these binary representations, which is quite different from our proposed approaches. UGACH is a generative adversarial network that utilizes a generative model to fit the distribution for the given data in one modality and select the data of another modality to challenge the discriminative model. In our deep hashing approach, we didn't adopt such a framework, either. UDCMH combines deep learning technologies with matrix factorization to preserve not only the data point of nearest neighbors but also the data point of farthest neighbors. However, UDCMH doesn't consider the information of high-order neighbors, which hinders its performance. Different from UDCMH, DJSRH involves the high-order neighbors' information to construct a joint-semantics affinity matrix for model learning. Inspired by DJSRH, we modify their model by adjusting the training objective of the positive pairs in the matrix, which can achieve better performance in our experiment.

We are the first to adopt deep hashing approaches into the code retrieval task and we found that the selection of deep hashing approaches can greatly affect the overall performance of our framework. Therefore, how to further improve the performance of deep hashing approaches can be one of the potential research directions in the future.

4.6 Threats to Validity

In this chapter, we have identified the following threats to validity.

4.6.1 Threats to External Validity

At first, we only select one Python dataset and one Java dataset in our evaluation. The limited number and size of the dataset may not accurately reflect the performance and efficiency of CoSHC.

Secondly, we only select five code retrieval models. According to our experiment results, the performance of our proposed framework will be dropped when the performance of the baseline is better. Therefore, the performance of our proposed framework may be not as good as the results shown in the chapter when a powerful code retrieval model is adopted in the real application.

At last, we only evaluate the proposed framework with the metrics of R@k in the overall performance experiment. However, these metrics may not sufficiently reveal the performance gap between CoSHC and original code retrieval models.

4.6.2 Threats to Internal Validity

In our experiment, we didn't show the inference time of our hashing model and category prediction models. Although the time cost of these two models is fixed and it will not change the conclusion shown in this chapter, the efficiency improvement may be not as large as the results shown in this chapter if the size of the code database is not large enough.

4.7 Summary

In this chapter, we proposed CoSHC, a general method that incorporates deep hashing techniques and code classification, to accelerate code retrieval. We leverage the two-staged recall and

re-rank paradigm in the information retrieval field and apply deep hashing techniques for fast recall. Furthermore, we propose to utilize a code classification module to retrieve better-quality code snippets. Experiments on five code retrieval models show that compared with the original code search models, CoSHC can greatly improve the retrieval efficiency meanwhile preserving almost the same performance.

□ **End of chapter.**

Chapter 5

Accelerating Code Retrieval via Segmented Deep Hashing

In this chapter, we continue to investigate the deep hashing for the code retrieval acceleration. Although previous deep hashing-based approaches can significantly reduce the code retrieval time, it still needs to scan the entire database during the search due to its Hamming distance calculation mechanism. Therefore, to further improve the efficiency of deep hashing, we propose a novel approach, CSSDH, to improve the retrieval efficiency of previous deep learning-based approaches. Specifically, CSSDH can convert the long hash code from previous deep hashing-based approaches into several segmented hash codes and construct the look-up hash table for these segmented hash codes. The scan of the entire code database can be avoided by utilizing these look-up hash tables, which leads to efficiency improvement. The comprehensive experiments on benchmarks demonstrate that CSSDH significantly reduces recall computational complexity while keeping the advanced performances of previous deep hashing approaches.

5.1 Introduction

Code retrieval is the approach that retrieves appropriate code snippets from a code base according to natural language queries. Such a technique can help software developers find target code snippets from millions of lines of code in a short time, thereby improving developer productivity. For novice developers, code retrieval can provide code samples for them to learn.

Since code retrieval is a very useful tool for developers, many code retrieval approaches [8, 78, 79] have been proposed in the past decades. The rapid development of open-source communities such as GitHub and Stack Overflow provides a huge amount of open-source code with natural language descriptions (comments). The big code data make it possible to adopt deep learning-based models for code retrieval [39, 41, 51].

The paradigms of existing deep learning-based approaches can be classified into two categories: cross-encoder and bi-encoder paradigms, both of which suffer from efficiency problems. (1) For cross-encoder approaches [28, 33], the model takes a pair of the source code and query together into a neural network and determines whether or not the given source code and query are matched by generating a match probability. However, in this way, the retrieval efficiency is extremely low since the model needs to pair the given query with every source code in the database and feed it into the neural network for inference. (2) For bi-encoder approaches [12, 39, 41], they use two neural networks to encode source code and queries separately into representation vectors. After the encoding, dense retrieval is adopted to retrieve the representation vectors of the source code, which have a large similarity (e.g., inner product) with the given rep-

resentation vector of the query [12, 84]. Recently, ColBERT [55] introduced a late interaction architecture to save the dense vectors offline and adopt representation retrieval to improve the retrieval efficiency for cross-encoder approaches.

However, the efficiency of code retrieval remains a problem even with the offline dense vectors [40]. To achieve high precision, code retrieval models usually use high dimensional representation vectors of source code and queries. The similarity calculation cost between such high dimensional representation vectors is high. Since dense retrieval requires a linear scan of the whole code database, the calculation cost of the retrieval for a single query will be extremely high. For example, the calculation cost for a single code retrieval with a d-dimensional vector in a code database with n code snippets will be $O(dn)$.

Deep hashing is a promising technique to address the efficiency problem of dense retrieval and has been widely adopted in other retrieval tasks such as image-text search [101, 119, 132] and image-image search [123, 134]. By converting high-dimensional dense vectors into low-dimensional binary hash codes, the deep hashing technique can greatly reduce storage and calculation cost [77]. In addition, the calculation of the Hamming distance between two binary hash codes can also be achieved by running the `XOR` instruction on modern computer architectures [112], further improving computation efficiency. Gu et al. [40] firstly adopted the deep hashing techniques for the generation of binary hash codes to improve the retrieval efficiency in the task of code retrieval. Their framework adopts a "search-rerank" pipeline that searches code candidates with Hamming distance of hash code and reranks them with semantic similarity with the query.

Most existing deep hashing approaches focus on generating binary hash codes that preserve the semantic similarity of original representation vectors. However, they target embedding projection to the binary hash embedding space, while less focus on multi-modal alignment between code and query. Thus, the mismatch between the hash codes of the code and query makes building an accurate lookup hash table hard. Although it is very efficient to calculate the Hamming distance with the `XOR` instruction [112], these Hamming distance-based methods have efficiency issues as they have to scan the whole large database. To address this, we propose a hash-based code retrieval framework CSSDH that achieves advanced performance by replacing the Hamming distance calculation with lookup hash tables.

In this chapter, we propose CSSDH, a deep hashing lookup table-based approach for code retrieval. Unlike CoSHC which utilizes the deep hashing approach to generate the hash code and adopt Hamming distance calculation for the retrieval, CSSDH can be integrated with previous deep hashing methods to fine-tune their outputted hash code. And Hamming distance calculation can be replaced with the hash table looking up, which leads to the further improvement of retrieval efficiency. Specifically, in CSSDH, we adopt an adaptive bit relaxing strategy and dynamic matching objective strategy to address the above-mentioned issues. Specifically, the adaptive bit relaxing strategy allows CSSDH to predict the hash bit with high error probability as "unknown". The dynamic matching objective strategy allows CSSDH to dynamically adjust the alignment target of hash code for each pair of the code snippet and query during the training, which aims to reduce the hash collision between the unmatched code snippets and queries.

Extensive experiments have been conducted to validate the performance of the proposed approach. Experimental results indicate that CSSDH can reduce at least 95% of the retrieval time of current state-of-the-art deep hashing approaches, which sort the candidates by calculating the Hamming distance. Meanwhile, CSSDH can retain the comparable performance or even outperform the previous deep hashing approaches in the recall step.

We summarize the main contributions of this chapter as follows:

- We propose a novel approach, CSSDH, to improve the retrieval efficiency of previous deep learning-based approaches. CSSDH is the first approach that adopts hard matching objective optimization with adaptive bits relaxing to address the mismatch problem between the hash codes from different modalities.

- CSSDH adopts the dynamic matching objective adjustment strategy, which allows the CSSDH to dynamically adjust the ground-truth label of the matching target to reduce the false positive hash collision condition.

- The comprehensive experiments on benchmarks demonstrate that CSSDH greatly reduces recall computational complexity while keeping advanced performances of previous deep hashing approaches, and improved efficiency brought by adaptive bits relaxing strategy and dynamic matching objective adjustment strategy.

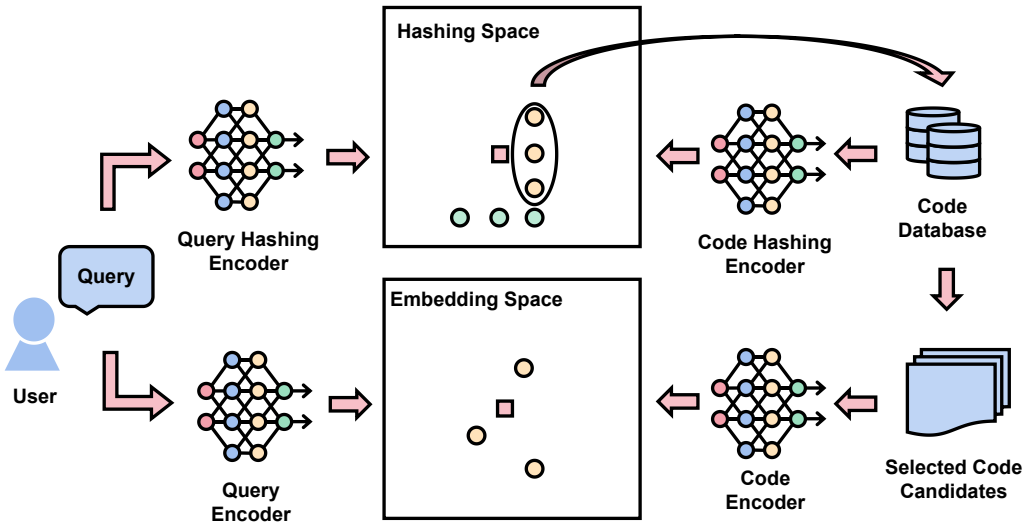


Figure 5.1: Illustration of recall and re-rank mechanism with previous deep hashing approaches.

5.2 Method

5.2.1 Overview

To improve searching efficiency, deep hashing approaches encode code snippets and queries into binary hash codes rather than high dimensional vectors. With these hash codes, the similarity between code snippets and queries can be measured by the Hamming distance between them. By adopting deep hashing approaches as the recall strategy, code candidates can be determined in a short time. In the previous deep hashing framework [40], the time complexity can be divided into two parts, which are the Hamming distance calculation and sorting. In the Hamming distance calculation part, it needs to calculate the Hamming distance between the given query and an arbitrary code snippet in the code database, whose time complexity is $O(n)$. In the sorting part, it needs to sort all the code snippets according to the previously calculated Hamming distance,

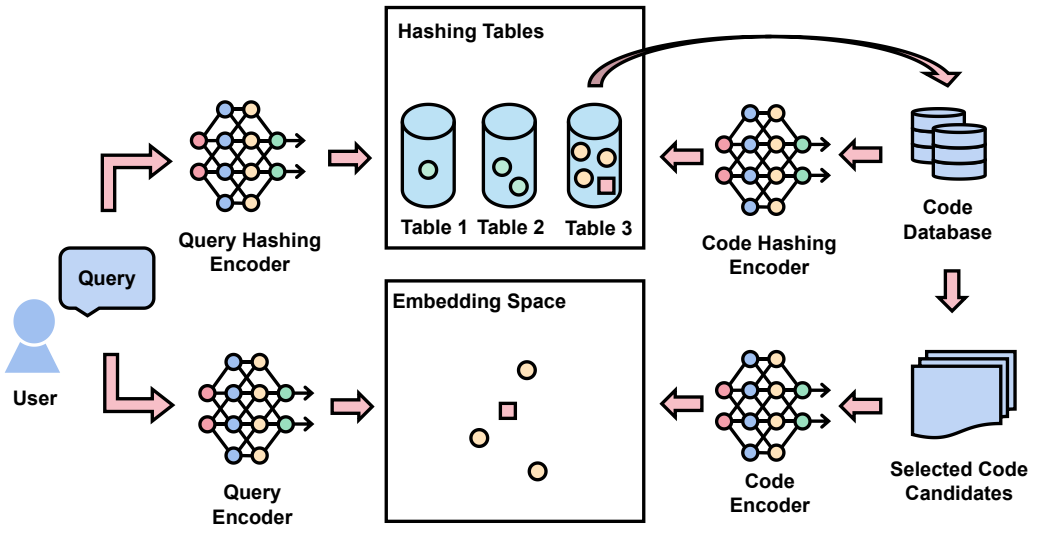


Figure 5.2: Illustration of recall and re-rank mechanism with the combination of deep hashing approaches and CSSDH.

whose time complexity is $O(nlogn)$. To further improve the efficiency of previous deep hashing approaches in the recall stage, we propose a general deep code hashing framework (as shown in Figure 5.3) to accelerate the Hamming distance calculation part.

Since keeping the hash codes of queries and their corresponding code snippets identical is not the optimization objective of previous deep hashing methods, the construction of lookup hash tables suffers from a severe hash bit mismatching problem. To alleviate the mismatching problem, we propose a novel iteration training strategy to tune the initial hashing codes. After obtaining the final code hashing representations, we introduce a retrieval algorithm based on our learned hashing model to efficiently recall a code candidate set.

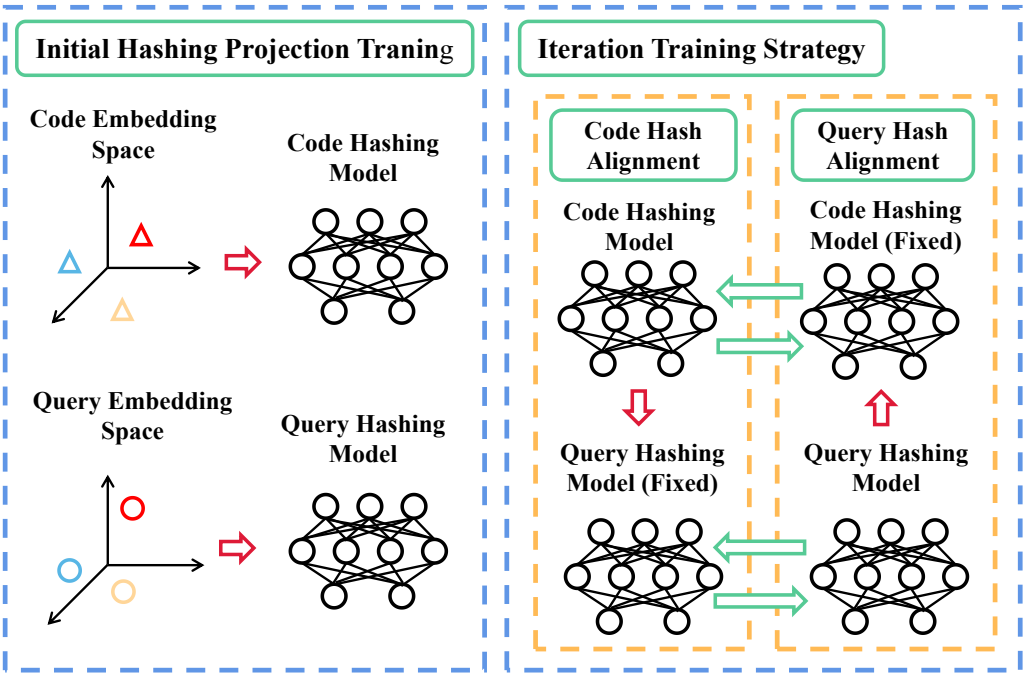


Figure 5.3: Overall framework of CSSDH. *Initial Hashing Project Training*: train the code hashing model and query hashing model; *Iteration Training Strategy*: fix the query/code hashing model and utilize the hash codes from the hashing model to construct the suitable matching objective for the hashing model.

5.2.2 Recall and Re-rank with Deep Hashing

Figure 5.1 illustrates the recall and re-rank mechanism with deep hashing techniques in code retrieval [40]. Firstly, the representation vectors and the corresponding binary hash codes of code snippets will be encoded via the bi-encoder code retrieval model and the deep hashing model, respectively. Once the system receives queries from the users, the representation vectors and the corresponding binary hash codes will also be generated. During the searching process, the binary hash codes will be utilized to recall the code candidates and the ranking of the returned results will be determined by the calculation of the similarity between

the representation vectors. As illustrated in Figure 5.2, unlike previous deep hashing approaches [40] that calculate the Hamming distance between the binary hash code of given query with the binary hash codes of the entire code database, CSSDH could reduce the time complexity for the Hamming distance calculation part in previous approaches from $O(n)$ to $O(1)$ via building up lookup hash tables according to the binary hash codes generated by the deep hashing approach.

5.2.3 Initial Hashing Projection Training

We leverage the existing deep hashing works for image-text search to design our initial hashing method for code-text search. It is worth noting that our following alignment and retrieval techniques do not depend on the specific image-text hashing method, we will introduce our design based on the typical general framework of deep hashing works for image-text search. The typical image-text hashing models include an image encoder, a query encoder, a soft binary transformation module (e.g. Tanh activation), and a contrastive loss based on positive and negative sample pairs. Our adaptation method is to replace the image encoder with a pre-trained code encoder. We employ CodeBERT [33] and GraphCodeBERT [43] as the code encoder in our experiments, separately. The loss function remains the same as the original image hashing method, which has the following mathematical form:

$$L = \sum_{i}^{n} f\left(sim(\mathbf{c}^{(i)}, \mathbf{q}^{(i)})\right) + \kappa \cdot \mathbb{E}_{(j,k)\sim P_n}\left[g(sim(\mathbf{c}^{(j)}, \mathbf{q}^{(k)}))\right], \tag{5.1}$$

where n is the number of positive training pairs, $\mathbf{c}^{(i)}$ is the i-th code hashing representation, $\mathbf{q}^{(i)}$ is the i-th query hashing representation, $sim(\cdot, \cdot)$ is a similarity function (e.g., cosine similarity or dot production), $f(\cdot)$ is a monotonically decreasing function to rescale the similarity, $g(\cdot)$ a monotonically increasing function correlated to $f(\cdot)$, P_n is a negative sampling distribution from which we can sample a negative pair (i, j), and κ is the number of negative samples. After optimization, we will discretize the learned vectors, i.e., if a value is greater than 0, it will be set as 1 otherwise it will be set as 0.

We can find that the loss is actually based on a continuous similarity function and hence the discretized vectors are hard to exactly match even for the actual positive pairs. It prevents us from using hash tables for fast index lookup. In the following subsection, we will show how our method addresses the hash code mismatch problem.

5.2.4 Iteration Training Strategy

To build the efficient hash table, we need to assign a suitable hash code for every code and query so that the hash code from the matched code and query should be the same and there should be at least one hash bit difference between the hash codes from the unmatched code and query. However, there is no natural ground-truth label, which means the hash code, for each pair of the code snippet and query during the training. To address this problem, we adopt a novel training strategy named iteration training strategy for hash training. In the iteration training strategy, only the model for one modality will be trained and the other one will be fixed. The fixed model and the updated model will be alternated with certain training epochs. Specifi-

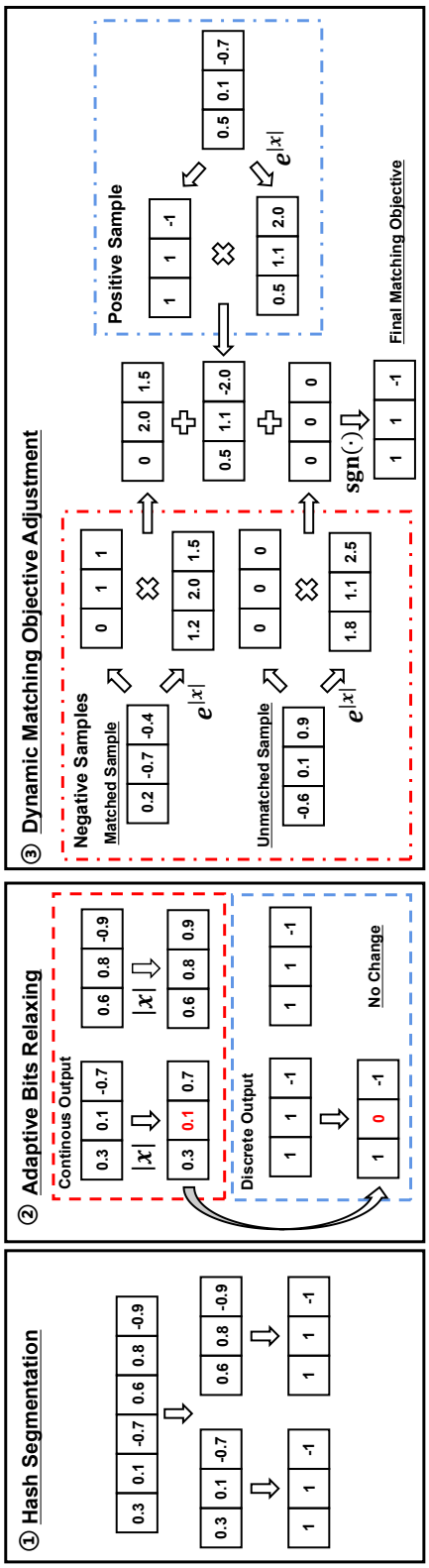


Figure 5.4: Steps in the iteration training strategy. *Hash Segmentation*: split the long hash code into several segmented hash codes and convert of continuous output value into the discrete value; *Adaptive Bits Relaxing*: select the hash bits from each segmented hash code according to the absolute output value from the model and overwrite the hash value as "unknown" on these hash bits, which represented as 0; *Dynamic Matching Objective Adjustment*: Assign the suitable matching objective for each pair of query and code snippet. The hash code from the positive sample will be utilized as the ground-truth label and adjusted according to the hash collision condition with the negative samples from the same batch.

cally, the iteration training strategy contains three steps: hash segmentation, adaptive bits relaxing, and dynamic matching objective adjustment. In the step of hash segmentation, the long hash code will be split into several short hash segments. In the step of adaptive bits relaxing, the hash bits that may have a high error probability will be relaxed to increase the probability of hash collision. In the step of dynamic matching objective adjustment, the output from the fixed model will be regarded as the temporary ground-truth label and will be adjusted according to the hash collision condition with the hash codes from the negative samples in the same training batch. Figure 5.4 shows the steps in the iteration training strategy and the detail will be introduced in the following sections.

Hash Segmentation

Precious deep hashing techniques utilize the Hamming distance as the metric to measure the similarity between two hash codes. The need for high precision drives these deep hashing techniques to generate long hash codes. However, it is very hard to achieve the hash collision between two long hash codes. Besides, the recall ratio will be very low for the hash collision approaches if only one hash table is constructed. To address the above problem, we split the initial binary hash code from the initial hashing projection into several segments and construct a lookup hash table for each segment. The segmented hash code is

$$H_i = \{h_{i1}, ..., h_{ik}\}, \tag{5.2}$$

where H_i is the i-th segmented hash code of the code or query, which is composed of k hash bits from the initial hash code. The j-th hash bit in the i-th segmented hash code is determined by

$$h_{ij} = \text{sgn}(o_{(i-1)*k+j}), \tag{5.3}$$

where $o_{(i-1)*k+j}$ is the output value of the $((i-1)*k+j)$-th hash bit from the neural network and $\text{sgn}(\cdot)$ is the sign function. For a more concise representation, $o_{(i-1)*k+j}$ will be replaced by o_{ij} in the following section.

Adaptive Bits Relaxing

Although we have already split the long hash codes into several short hash segments, it is still hard to keep the hash codes from the same pair of the code snippet and its query to be the same. To address this mismatch problem, we propose the strategy named *adaptive bits relaxing*. The target hash code in the training is a discrete value, which is +1 and -1. The closer the output of the model to the target value is, the better the convergence of the model is. So there is a high probability that the output with a low absolute value is incorrect. To mitigate the mismatching problems brought by these hash bits, we give up the prediction on these uncertain hash bits and overwrite the output on these hash bits as both +1 and -1.

To achieve the adaptive bits relaxing, we first select the hash bits with top k smallest absolute value as the uncertain bits in each hash segment, which is shown below:

$$S_i = \{j \,|\, |o_{ij}| \text{ is top k smallest in } \mathrm{O_i}\}, \tag{5.4}$$

where S_i is the set that contains the hash bits with the top k smallest absolute value. k is the maximum number of relaxing bits allowed in a single segmented hash code. For these relaxing bits in each hash segment, we replace the initial hash value with

0 as the intermediate value, which is shown below:

$$\tilde{h}_{ij} = \begin{cases} 0, & j \in S_i \text{ and } |o_{ij}| \leq t \\ h_i, & \text{otherwise} \end{cases}, \tag{5.5}$$

where $\tilde{h}_{ij}$ is the hash code on the i-th hash bit after the relaxing. Since the convergence condition of the model may be good on all the hash bits, we pre-define a threshold value t for the relaxing. Only the hash bits whose absolute value is lower than t are allowed to be relaxed.

Dynamic Matching Objective Adjustment

To build the perfect hash table, we hope that the hash code from the matched code and query are identical and the hash codes from the unmatched code and query have at least a one-bit difference. The determination of the suitable hash code for each pair of the code and query becomes the key problem. To assign the suitable hash code from each pair of code and query, we first get the hash codes from the fixed model as the ground-truth label and then check the hash collision condition of this hash code with the negative samples from the batch. Finally, this ground-truth label will be adjusted according to the hash collision condition with the negative samples and utilized in the training.

In the first step, we need to check whether the hash code of the negative samples in the batch is the same as the hash code we retrieved from the fixed model. Equation 5.6 is the matching results for every bit in the hash code:

$$c_{ij} = \tilde{h}_{ij}^{-} \cdot \tilde{h}_{ij}^{+}, \tag{5.6}$$

where $\tilde{h}_{ij}^{-}$ is the j-th hash bit in the i-th segmented negative hash codes from the modality which needs to be updated and $\tilde{h}_{ij}^{+}$ is

the j-th hash bit in the i-th segmented positive hash codes from the fixed model. c_{ij} indicates whether the j-th hash bit between the i-th segmented positive hash code and the i-th segmented negative hash code is matched. Since we know that $\tilde{h}_{ij}^{-}, \tilde{h}_{ij}^{+} \in \{+1, 0, -1\}$, then we can get that $c_{ij} \in \{+1, 0, -1\}$. $c_{ij} = +1$ indicates that the two hash bits are identical, $c_{ij} = 0$ indicates that there is at least one hash bit is relaxed, and $c_{ij} = -1$ indicates that the two hash bits are unmatched. Then we define the C_i as follows:

$$C_i = \min\{c_{i1}, ..., c_{ik}\}, \tag{5.7}$$

where $C_i = -1$ indicates that there exists at least one hash bit unmatched between two segmented hash codes. otherwise, these two segmented hash codes can be regarded as identical. For convenient calculation, we define $\tilde{C}_i$ to indicate whether the negative segmented hash code has the hash collision with the positive segmented hash code as

$$\tilde{C}_i = \begin{cases} 0, C_i = -1 \\ 1, \text{otherwise} \end{cases}, \tag{5.8}$$

where $\tilde{C}_i = 1$ indicates that the i-th segmented negative sample has the hash collision with the segmented positive sample, otherwise does not.

Since we have checked the hash collision condition with negative samples, then we can adjust the ground-truth label we get from the fixed model for the hash alignment with such information. The adjusted ground-truth label can be determined as follows:

$$l_{ij} = h_{ij}^{+} \cdot e^{\gamma \cdot |o_{ij}^{+}|} - \sum_{n=1}^{m} C_{in} \cdot \tilde{h}_{ij}^{-} \cdot e^{\gamma \cdot |o_{ijn}^{-}|}, \tag{5.9}$$

where l_{ij} is the j-th hash bit in the ground-truth label for the i-th segmented hash code. m is the negative sample number in the batch. γ is the constant parameter. Since the value range of o_{ij} is $[1, +1]$, γ can be utilized to amplify the value range so that there is less probability for the hash bits with good convergence conditions to change their sign. For the positive sample, hash bits before the adaptive bits relaxing are selected in the above equation since we still need to offer a clear optimization target for the neural network and the output of these hash bits may get out of the ill convergence condition in the following training epochs. For the negative samples, the hash bits after the adaptive bits relaxing are selected since the hash collision with the negative samples cannot be avoided by changing the output on these relaxed hash bits. Finally, we need to discrete the ground-truth label for the hash code bit as

$$\tilde{l}_{ij} = \text{sgn}(l_{ij}), \tag{5.10}$$

where $\tilde{l}_{ij}$ is the j-th hash bit in the i-th final segmented hash code template. The final ground-truth label of the i-th segmented hash code for the hash alignment is

$$\tilde{L}_i = \{\tilde{l}_{i1}, ..., \tilde{l}_{ik}\} \tag{5.11}$$

5.2.5 Hash Alignment

Since we get the alignment template for every pair of code and query, we can align the hash code with the following cross-entropy loss:

$$L = -(1 - \tilde{l}_{ij}) * \log(1 - o_{ij}) - (1 + \tilde{l}_{ij}) * \log(1 + o_{ij}), \tag{5.12}$$

where $\tilde{l}_{ij}$ is the j-th hash bit in the final ground-truth label for the i-th segmented hash code. o_{ij} is the output of j-th hash

bit in the i-th segmented hash code from the neural network. The reason why the loss function of Mean Squared Error (MSE) is not adopted is that the output of the model can hardly be changed when it is close to 1 or -1.

5.2.6 Inference of Binary Hash Codes

In the inference stage of binary hash codes, binary hash codes of source code and queries will be generated by the corresponding hashing model. Firstly, the hashing model will output the continuous hashing value. Then hash code will be split into several segmented hash codes with adaptive bits relaxing strategy as we introduced in Section § 5.2.4. The unknown state for the hash bit will only be treated as an intermediate state in the inference and finally, it will be converted into both 1 and -1. The hash value of the rest hash bits where be converted into 1 or -1 according to the output hash value as a positive number or a negative number.

During searching with lookup hash tables, all the hit code snippets will be added to the recall candidate set. If the users want to set the maximum size of the recall candidate set, we will use a hash table to count the matched times and then apply a Bucket sort to re-rank these candidates.

5.3 Experimental Settings

5.3.1 Datasets

We follow the same data processing approach described in Section § 4.3.1 for the dataset of CodeSearchNet [51]. As Table 5.3.1 shows, we finally get 412,178 $\langle \text{code}, \text{query} \rangle$ pairs as

Dataset	Training	Validation	Test
Python	412,178	23,107	22,176
Java	454,451	15,328	26,909

Table 5.1: Dataset statistics.

the training set, 23,107 $\langle \text{code}, \text{query} \rangle$ pairs as the validation set, and 22,176 $\langle \text{code}, \text{query} \rangle$ pairs as the test set in the Python dataset. We get 454,451 $\langle \text{code}, \text{query} \rangle$ pairs as the training set, 15,328 $\langle \text{code}, \text{query} \rangle$ pairs as the validation set, and 26,909 $\langle \text{code}, \text{query} \rangle$ pairs as the test set in the Java dataset.

5.3.2 Baselines

We select two state-of-the-art deep learning-based code retrieval models with four deep hashing approaches and a non-learning-based approach as our baselines.

Code Retrieval Baselines

We select CodeBERT [33] and GraphCodeBERT [43] as our base code retrieval models. Both of them are state-of-the-art models in the code retrieval task.

- **CodeBERT** is a bi-modal pre-trained model based on a Transformer with 12 layers for programming languages and natural languages. CodeBERT utilizes the last hidden vector of the special `[CLS]` token as the embedding of source code or description to predict whether the given pair of source code and description are matched.
- **GraphCodeBERT** is another pre-trained Transformer-based model. Unlike previous pre-trained models which only utilize the sequence of code tokens as the features, GraphCodeBERT

additionally considers the data flow of code snippets in the pre-training stage. Similar to CodeBERT, GraphCodeBERT utilizes the last hidden vector of the `[CLS]` token as the representation vector.

Deep Hashing Baselines

We select four state-of-the-art baseline models of deep hashing, which are CoSHC [40], DJSRH [101], DSAH [124], and JDSH [73]. We also select a hash table-based approach LSH [25].

- **CoSHC** is the first approach that combines the deep hashing techniques with the module of code classification to accelerate code retrieval. For the sake of fairness of the experiment, we only adopt the deep hashing parts from this approach.
- **DJRSH** constructs a novel joint-semantic affinity matrix that contains specific similarity values instead of similarity order as in previous approaches. DJRSH can generate binary codes that are capable of preserving the neighborhood structure of the original data.
- **DSAH** designs a semantic-alignment loss function to align the similarity between input features and generated binary hash codes. To bridge the gap between different modalities, DSAH also utilizes an additional fully connected layer to reconstruct features of one modality with the generated hash codes of the other.
- **JDSH** is a deep hashing approach that jointly trains the model for different modalities with a joint-modal similarity matrix, which can fully preserve cross-modal semantic correlations. JDSH employs distribution-based similarity decision

and weighting (DSDW) to generate more discriminative hash codes.

- **LSH** is one of the most popular approaches that can map high dimensional data to hash value by using random hash functions and constructing the lookup hash tables for the data searching. It is widely applied in data recall for a single modality.

5.3.3 Metrics

We use $R@k$ (recall at k) and MRR (mean reciprocal rank) as the evaluation metrics in this chapter. $R@k$ is the metric widely used to evaluate the performance of the code retrieval models by many previous studies [32, 44, 46, 100]. It is defined as

$$R@k = \frac{1}{|Q|}\sum_{q=1}^{Q}\delta(FRank_q \leq k), \tag{5.13}$$

where Q denotes the query set and $FRank_q$ is the rank of the correct answer for query q. $\delta(Frank_q \leq k)$ returns 1 if the correct result is within the top k returning results, otherwise it returns 0. A higher $R@k$ indicates better performance.

MRR is another metric widely used in the code retrieval task to evaluate the performance [33, 43]:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{Q}\frac{1}{FRank_q} \tag{5.14}$$

A higher MRR indicates better performance.

5.3.4 Implementation Details

Since the target of CSSDH is to accelerate the recall efficiency with deep hashing techniques for dense retrieval, we slightly modify the model structure in our experiment. We use the bi-encoder paradigm to use two CodeBERT or GraphCodeBERT to encode the source codes and queries into representation vectors. We apply a mean pooling function on the last hidden layer of both CodeBERT and GraphCodeBERT to get the representation vectors. The dimension of the representation vectors is 768.

The network architectures of CoSHC, DJSRH, DSAH, and JDSH are very similar. All of them replace the output layer of the initial retrieval model with three fully connected layers. However, the computation cost of training is high since we adopt CodeBERT and GraphCodeBERT as the initial retrieval model. To accelerate the training process, we directly apply three fully connected layers as the hashing model and utilize the representation vectors from the initial retrieval model as the input feature of the hashing model. We follow the hyperparameter settings of deep hashing baselines described in their original papers. We experiment on 128-bit and 256-bit for the generated binary hash codes. The hidden size of all deep hashing models is 1,536, which is twice the dimension of the representation vectors for the source codes and queries. We set the size of the binary hash code segment to 16 bits and we allow the deep hash model to predict no more than three unknown bits in each segment. Due to the low recall ratio of LSH, we reduce the length of the hash segment to 8 bits. In addition, we set the threshold value t as 0.5. In the experiment of overall performance comparison in Section § 5.4.2, deep hashing models retrieve the top

300 candidates of code snippets at first. Then the code retrieval models including CodeBERT and GraphCodeBERT determine the final ranking order of these candidates. In the experiment of time efficiency in Section§ 5.4.1, we only compare the time cost for the top 300 candidates retrieved by the deep hashing approaches since our focus is on the retrieval efficiency of the recall step.

The learning rate of the code retrieval models including CodeBERT and GraphCodeBERT is 1e-5 and the learning rate for all the deep hashing baselines is 1e-4. All the models are optimized via the AdamW algorithm [58].

We train our models on a server with a Tesla V100. We train CodeBERT or GraphCodeBERT for 10 epochs to get representation vectors for both code snippets and queries. The training epoch number for either initial hashing projection or iteration training is 100. An early stopping strategy is adopted to avoid over-fitting for all models. We evaluate the retrieval efficiency of the proposed approach on a server with Intel Xeon E5-2698v4 2.2Ghz 20-core. The code for efficiency evaluation is written in C++ and the program is only allowed to use a single thread of CPU for fair comparison.

5.4 Evaluation

5.4.1 RQ1: What is the Efficiency of CSSDH?

Table 5.2 shows the experiment results of time efficiency comparison in the recall step of different approaches with different sizes of the Python dataset. To compare the recall efficiency of the previous deep hashing approaches with and without CSSDH, only the time cost in the recall step is recorded and the time cost

		50,000		100,000		200,000		400,000	
		128bit	**256bit**	**128bit**	**256bit**	**128bit**	**256bit**	**128bit**	**256bit**
CodeBERT	LSH	3.8s	7.5s	8.1s	16.4s	16.7s	37.6s	38.8s	82.8s
	CoSHC	31.9s	43.7s	66.4s	90.1s	137.7s	184.8s	280.1s	375.7s
	CoSHC$_{\text{CSSDH}}$	1.2s (↓96.2%)	2.2s (↓95.0%)	2.1s (↓96.8%)	3.8s (↓95.8%)	4.0s (↓97.1%)	7.1s (↓96.2%)	7.8s (↓97.2%)	14.2s (↓96.2%)
	DJSRH	31.2s	43.1s	65.2s	88.7s	135.1s	185.8s	274.5s	367.8s
	DJSRH$_{\text{CSSDH}}$	1.2s (↓96.2%)	1.4s (↓96.8%)	2.1s (↓96.8%)	2.5s (↓97.1%)	3.9s (↓97.1%)	4.4s (↓97.6%)	7.9s (↓97.1%)	8.4s (↓97.6%)
	DSAH	31.2s	44.0s	65.3s	90.5s	135.0s	186.0s	275.5s	376.8s
	DSAH$_{\text{CSSDH}}$	1.0s (↓96.8%)	1.4s (↓96.8%)	1.9s (↓97.1%)	2.5s (↓97.2%)	3.5s (↓97.4%)	4.5s (↓97.6%)	6.8s (↓97.5%)	8.4s (↓97.8%)
	JDSH	31.1s	44.0s	65.2s	90.6s	135.0s	185.7s	274.4s	368.6s
	JDSH$_{\text{CSSDH}}$	1.2s (↓96.1%)	1.5s (↓96.6%)	2.0s (↓96.9%)	2.6s (↓97.1%)	3.8s (↓97.2%)	4.6s (↓97.5%)	7.5s (↓97.3%)	8.6s (↓97.7%)
GraphCodeBERT	LSH	3.7s	7.3s	7.7s	15.5s	16.9s	34.9s	38.2s	82.2s
	CoSHC	31.9s	43.7s	66.5s	90.1s	137.6s	184.9s	280.0s	375.9s
	CoSHC$_{\text{CSSDH}}$	1.1s (↓96.6%)	2.2s (↓95.0%)	2.0s (↓97.0%)	3.8s (↓95.8%)	3.8s (↓97.2%)	7.0s (↓96.2%)	7.4s (↓97.4%)	13.8s (↓96.3%)
	DJSRH	31.2s	43.0s	65.2s	88.5s	134.9s	181.7s	274.5s	367.8s
	DJSRH$_{\text{CSSDH}}$	1.1s (↓96.5%)	1.5s (↓96.5%)	2.0s (↓96.9%)	2.6s (↓97.1%)	3.8s (↓97.2%)	4.6s (↓97.5%)	7.6s (↓97.2%)	8.8s (↓97.6%)
	DSAH	31.1s	43.9s	65.2s	90.4s	134.9s	185.7s	274.7s	377.4s
	DSAH$_{\text{CSSDH}}$	1.0s (↓96.7%)	1.5s (↓96.6%)	1.8s (↓97.2%)	2.6s (↓97.1%)	3.4s (↓97.5%)	4.7s (↓97.5%)	6.6s (↓97.6%)	8.8s (↓97.7%)
	JDSH	31.1s	43.8s	65.1s	90.3s	134.9s	185.6s	275.2s	376.3s
	JDSH$_{\text{CSSDH}}$	1.1s (↓96.5%)	1.5s (↓96.6%)	2.0s(↓96.9%)	2.6s (↓97.1%)	3.7s (↓97.3%)	4.9s (↓97.3%)	7.2s (↓97.4%)	9.1s (↓97.6%)

Table 5.2: Results of time efficiency comparison on the recall step of different deep hashing approaches with different code retrieval models on the Python dataset with the size 50,000, 100,000, 200,000 and 400,000.

of re-ranking is neglected in this experiment.

First of all, we can find that CSSDH can reduce more than 95% of the searching time for all sizes of the dataset and hash bits compared to previous deep Hamming distance-based hashing approaches. What's more, the efficiency of CSSDH is also higher than the conventional LSH approach, which demonstrates the effectiveness of CSSDH in the improvement of recall efficiency.

According to Table 5.2, we can find that the retrieval time of different hashing approaches with the same hash code length in the same database is almost the same while the time cost of CSSDH is not as stable as the deep hashing approaches, which has a small variation among different deep hashing approaches even with the same set of hash code length. The reason for this result is the difference in the retrieval mechanism of CSSDH. Unlike calculating the Hamming distance between the fixed number of hash codes, CSSDH will search with the given hash code segments in every lookup hash table, count the appearance frequency of every hash code, and sort the candidates of source code according to their appearance frequency of them in all the lookup hash tables. Since the hash collision condition in each lookup hash table will be varied depending on the different deep hashing approaches, the search time will also be varied.

From Table 5.2, we can find that the time cost of the deep hashing approaches with CSSDH has sublinear growth while the time cost of the deep hashing approaches has superlinear growth as the size of the dataset grows, which demonstrates that the deep hashing approach with CSSDH is more efficient than the deep hashing approach without CSSDH with the larger dataset. However, we can notice that although the increasing tendency of

time cost of CSSDH with the increase of dataset size is sublinear, it still does not meet the $O(1)$ complexity. The reason for this is that CSSDH contains both searching and sorting processes in the recall step. Although the time complexity of the searching process with the lookup hash tables is $O(1)$, we still need to count the appearance times of the hit candidates in each lookup hash table and sort these candidates to determine the list of recall candidates according to preset the recall number. The reason why we set the maximum recall number for the single retrieval is that the matched binary hash codes in each hash table will be increased while the increasing of dataset size. Too many recall candidates will harm the efficiency of the re-ranking step, which is more than the sorting cost in the recall step. It is unnecessary to worry whether the sorting process will harm the effectiveness of CSSDH since the previous deep hashing approaches also contain the sorting process with the time complexity of $O(n\log n)$ for the entire dataset in the recall step. Since CSSDH cannot recall the code candidates more than the dataset has, the upper bound of the time complexity of the sorting process in CSSDH is $O(n\log n)$ with the dataset containing n code snippets, which is no large than the time complexity of the sorting process in previous deep hashing approaches. The efficiency of deep hashing approaches with CSSDH will keep increasing with the increase of the dataset compared to the previous deep hashing approaches.

5.4.2 RQ2: What is the Effectiveness of CSSDH?

Table 5.3 illustrates the results of the overall performance comparison of different approaches with different code retrieval models. First of all, we can find that CSSDH can preserve at least 98.0%, and 97.0% of the performance in terms of R@1, MRR

	Model	Python				Java			
		128bit		256bit		128bit		256bit	
		R@1	MRR	R@1	MRR	R@1	MRR	R@1	MRR
CodeBERT	Original	0.455	0.562	0.455	0.563	0.321	0.419	0.322	0.420
	LSH	0.390	0.460	0.438	0.531	0.264	0.330	0.302	0.386
	CoSHC	0.455	0.562	0.455	0.563	0.321	0.419	0.322	0.420
	CoSHC$_{\text{CSSDH}}$	0.447 (↓1.8%)	0.547 (↓2.7%)	0.452 (↓0.7%)	0.554 (↓1.6%)	0.316 (↓1.6%)	0.408 (↓2.6%)	0.319 (↓0.9%)	0.415 (↓1.2%)
	DJSRH	0.454	0.561	0.455	0.563	0.321	0.418	0.322	0.420
	DJSRH$_{\text{CSSDH}}$	0.446 (↓1.8%)	0.546 (↓2.7%)	0.451 (↓0.9%)	0.553 (↓1.8%)	0.316 (↓1.6%)	0.409 (↓2.2%)	0.319 (↓0.9%)	0.414 (↓1.4%)
	DSAH	0.450	0.552	0.451	0.554	0.317	0.411	0.319	0.414
	DSAH$_{\text{CSSDH}}$	0.447 (↓0.7%)	0.547 (↓0.9%)	0.452 (↑0.2%)	0.554 (0.0%)	0.316 (↓0.3%)	0.409 (↓0.5%)	0.319 (0.0%)	0.415 (↑0.2%)
	JDSH	0.448	0.549	0.450	0.552	0.317	0.410	0.318	0.412
	JDSH$_{\text{CSSDH}}$	0.447 (↓0.2%)	0.547 (↓0.4%)	0.452 (↑0.4%)	0.554 (↑0.4%)	0.316 (↓0.3%)	0.409 (↓0.2%)	0.319 (↑0.3%)	0.415 (↑0.7%)
GraphCodeBERT	Original	0.489	0.598	0.489	0.598	0.355	0.457	0.355	0.457
	LSH	0.412	0.480	0.471	0.565	0.279	0.340	0.334	0.420
	CoSHC	0.489	0.597	0.489	0.598	0.355	0.455	0.355	0.457
	CoSHC$_{\text{CSSDH}}$	0.479 (↓2.0%)	0.580 (↓2.8%)	0.484 (↓1.0%)	0.587 (↓1.8%)	0.348 (↓2.0%)	0.443 (↓2.6%)	0.353 (↓0.6%)	0.451 (↓1.3%)
	DJSRH	0.489	0.597	0.489	0.598	0.354	0.454	0.355	0.457
	DJSRH$_{\text{CSSDH}}$	0.479 (↓2.0%)	0.579 (↓3.0%)	0.482 (↓1.4%)	0.586 (↓2.0%)	0.348 (↓1.7%)	0.444 (↓2.2%)	0.353 (↓0.6%)	0.450 (↓1.5%)
	DSAH	0.482	0.586	0.484	0.589	0.351	0.447	0.352	0.449
	DSAH$_{\text{CSSDH}}$	0.480 (↓0.4%)	0.580 (↓1.0%)	0.484 (0.0%)	0.587 (↓0.3%)	0.349 (↓0.6%)	0.444 (↓0.7%)	0.353 (↑0.3%)	0.450 (↑0.2%)
	JDSH	0.482	0.585	0.483	0.587	0.350	0.446	0.351	0.448
	JDSH$_{\text{CSSDH}}$	0.478 (↓0.8%)	0.579 (↓1.0%)	0.483 (0.0%)	0.586 (↓0.2%)	0.349 (↓0.3%)	0.443 (↓0.7%)	0.353 (↑0.6%)	0.450 (↑0.4%)

Table 5.3: Results of overall performance comparison of different deep hashing approaches with different code retrieval models.

with all the deep hashing-based code retrieval baselines in all the datasets, respectively. In addition, CSSDH also outperforms the conventional LSH baselines in all the metrics. These results demonstrate that CSSDH can retain most of the retrieval performance.

What's more interesting, we can find that the performance gap between the deep hashing approaches with and without CSSDH shrinks when the hash codes have more bits. DASH and JDSH with CSSDH even outperform the baselines with 256 hash bits. The reason for this performance improvement is the mechanism of CSSDH. The increase of the hash code length will directly increase the number of lookup hash tables under the setting of CSSDH, which can effectively increase the possibility of the recall for the corresponding code. Since the hash codes are very space-efficient and the extra space cost for the increase of the hash code's length can be almost neglected. This phenomenon indicates that the problem of the performance drop with CSSDH can be addressed by the increasing hash code's length, which further demonstrates the potential of CSSDH.

Secondly, we can find that the performance of CSSDH is stable under different deep hashing approaches, which demonstrates the generalizability of CSSDH. However, we can still find that there is a small performance difference under different deep hashing approaches. The reason for this phenomenon is the hashing projection distribution, which will be discussed in Section § 5.4.3.

Model	Python				Java			
	128bit		256bit		128bit		256bit	
	R@1	MRR	R@1	MRR	R@1	MRR	R@1	MRR
$\text{CoSHC}_{\text{NA\_NR}}$	0.270	0.312	0.334	0.390	0.214	0.263	0.251	0.313
$\text{CoSHC}_{\text{A\_NR}}$	0.383	0.459	0.410	0.493	0.267	0.338	0.290	0.370
$\text{CoSHC}_{\text{NA\_SR}}$	0.417	0.499	0.440	0.533	0.301	0.384	0.311	0.400
$\text{CoSHC}_{\text{A\_SR}}$	0.435	0.527	0.445	0.542	0.304	0.391	0.313	0.406
$\text{CoSHC}_{\text{NA\_BR}}$	0.445	0.543	0.451	**0.554**	0.315	0.405	**0.319**	0.414
$\text{CoSHC}_{\text{A\_BR}}$	**0.447**	**0.547**	**0.452**	**0.554**	**0.316**	**0.408**	**0.319**	**0.415**
$\text{DJSRH}_{\text{NA\_NR}}$	0.078	0.086	0.125	0.140	0.061	0.072	0.110	0.132
$\text{DJSRH}_{\text{A\_NR}}$	0.384	0.460	0.414	0.500	0.268	0.338	0.289	0.368
$\text{DJSRH}_{\text{NA\_SR}}$	0.250	0.289	0.319	0.375	0.183	0.226	0.249	0.312
$\text{DJSRH}_{\text{A\_SR}}$	0.432	0.524	0.445	0.541	0.305	0.392	0.313	0.404
$\text{DJSRH}_{\text{NA\_BR}}$	0.396	0.472	0.414	0.497	0.273	0.345	0.297	0.379
$\text{DJSRH}_{\text{A\_BR}}$	**0.446**	**0.546**	**0.451**	**0.553**	**0.316**	**0.409**	**0.319**	**0.414**
$\text{DSAH}_{\text{NA\_NR}}$	0.313	0.365	0.374	0.444	0.232	0.288	0.271	0.341
$\text{DSAH}_{\text{A\_NR}}$	0.388	0.466	0.417	0.502	0.268	0.339	0.292	0.371
$\text{DSAH}_{\text{NA\_SR}}$	0.421	0.509	0.438	0.533	0.299	0.383	0.310	0.398
$\text{DSAH}_{\text{A\_SR}}$	0.436	0.529	0.443	0.540	0.305	0.392	0.314	0.405
$\text{DSAH}_{\text{NA\_BR}}$	0.441	0.537	0.449	0.550	0.312	0.402	0.317	0.410
$\text{DSAH}_{\text{A\_BR}}$	**0.447**	**0.547**	**0.452**	**0.554**	**0.316**	**0.409**	**0.319**	**0.415**
$\text{JDSH}_{\text{NA\_NR}}$	0.326	0.384	0.384	0.459	0.246	0.307	0.280	0.354
$\text{JDSH}_{\text{A\_NR}}$	0.388	0.465	0.416	0.502	0.269	0.340	0.290	0.370
$\text{JDSH}_{\text{NA\_SR}}$	0.425	0.513	0.438	0.533	0.304	0.388	0.311	0.400
$\text{JDSH}_{\text{A\_SR}}$	0.436	0.529	0.445	0.543	0.308	0.396	0.314	0.405
$\text{JDSH}_{\text{NA\_BR}}$	0.441	0.537	0.448	0.549	0.314	0.404	0.318	0.411
$\text{JDSH}_{\text{A\_BR}}$	**0.447**	**0.547**	**0.452**	**0.554**	**0.316**	**0.409**	**0.319**	**0.415**

Table 5.4: The comparisons among the six CSSDH variants with the baseline of CodeBERT.

5.4.3 RQ3: What is the Effectiveness of Adaptive Bits Relaxing?

Table 5.4 illustrates the performance comparison of the six variants of CSSDH with the baseline of CodeBERT. The performance of CSSDH with both baselines of GraphCodeBERT is very similar to CodeBERT. There are five types of subscripts in Table 5.4, which are NA, A, NR, SR, and BR. NA represents the model that only splits the long hash code into several segmented hash codes without iteration training. A represents the model that splits the long hash code into several segmented hash codes with iteration training. NR represents the model doesn't adopt the adaptive bits relaxing strategy. SR represents the model that only adopts the adaptive bits relaxing strategy on the code hash model. BR represents the model that adopts the adaptive bits relaxing strategy on both the code hash model and query hash model. The strategy of NA or A can be combined with the strategy of NR, SR, or BR arbitrary, For example, $\text{Model}_{\text{A\_BR}}$ represents the model splits the long hash code into several segmented hash codes with both iteration training strategy and adaptive relaxing strategy.

As shown in Table 5.4, $\text{Model}_{\text{A\_BR}}$ achieves the best performance among six variants, which demonstrates the effectiveness of the combination of iteration strategy and adaptive bits relaxing strategy. Besides, we can find that either iteration strategy or adaptive bits relaxing strategy can greatly improve the model performance from the results. By comparing the performance between Model_{NA} and Model_{A}, we can find the performance for all the settings is improved. The performance of Model_{A} can be greatly improved when the performance of Model_{NA} is low. For example, we can find the performance of $\text{Model}_{\text{NA\_NR}}$ is

far from the performance of the baselines, and the performance of $\text{Model}_{\text{A\_NR}}$ improved a lot. By comparing the performance among Model_{NR}, Model_{SR}, and Model_{BR}, we can also get similar results as above. What's more, we can find the performance improvement brought by the adaptive bits relaxing strategy is higher than the improvement brought by the iteration training strategy. $\text{Model}_{\text{NA\_BR}}$ can preserve more than 98% performance of $\text{Model}_{\text{A\_BR}}$ for, CoSHC, DSAH, and JDSH baselines. However, the adaptive relaxing strategy does not always work well alone. By comparing the performance of $\text{DJSRH}_{\text{NA\_BR}}$ and $\text{DJSRH}_{\text{A\_BR}}$, we can find the performance serious declines when only the adaptive bits relaxing strategy is adopted. This result also indicates the necessity of the combination of the iteration training strategy and the adaptive bits relaxing strategy.

Interestingly, we can find that the performance of DJSRH_{NA} is much worse than the rest models. The hashing projection distribution is the reason for this phenomenon. Good deep hashing approaches should not only shorten the Hamming distance between the positive pairs but also enlarge the Hamming distance between the negative pairs. Since the Hamming distance between the positive pairs in DJSRH is not short enough, the initial hashing projection from DJSRH leads to poor performance of the hash collision. Fortunately, the combination of iteration training strategy and adaptive bit relaxing strategy can make up for such bad initial projection and the final performance of DJSRH is just slightly worse than other deep hashing approaches, which also further demonstrates the effectiveness and generalizability of CSSDH.

	Model	Python		Java	
		128bit	**256bit**	**128bit**	**256bit**
CodeBERT	$\text{CoSHC}_{\text{Code}}$	0.356	0.358	0.359	0.370
	$\text{CoSHC}_{\text{Query}}$	0.356	0.358	0.357	0.369
	$\text{DJSRH}_{\text{Code}}$	0.356	0.377	0.365	0.387
	$\text{DJSRH}_{\text{Query}}$	0.356	0.378	0.362	0.384
	$\text{DSAH}_{\text{Code}}$	0.353	0.374	0.360	0.382
	$\text{DSAH}_{\text{Query}}$	0.355	0.374	0.358	0.382
	$\text{JDSH}_{\text{Code}}$	0.353	0.374	0.360	0.385
	$\text{JDSH}_{\text{Query}}$	0.350	0.374	0.357	0.386
GraphCodeBERT	$\text{CoSHC}_{\text{Code}}$	0.353	0.358	0.359	0.375
	$\text{CoSHC}_{\text{Query}}$	0.355	0.359	0.356	0.374
	$\text{DJSRH}_{\text{Code}}$	0.356	0.377	0.365	0.387
	$\text{DJSRH}_{\text{Query}}$	0.356	0.378	0.362	0.384
	$\text{DSAH}_{\text{Code}}$	0.350	0.376	0.357	0.385
	$\text{DSAH}_{\text{Query}}$	0.348	0.376	0.355	0.384
	$\text{JDSH}_{\text{Code}}$	0.349	0.374	0.354	0.382
	$\text{JDSH}_{\text{Query}}$	0.350	0.374	0.351	0.381

Table 5.5: The repair ratio of adaptive bits relaxing in both code hashing model and query hashing model.

5.4.4 RQ4: How Many Error Bits Have Been Fixed?

Table 5.5 illustrates the repair ratio of the adaptive bits relaxing in both the code hashing model and the query hashing model. $\text{Model}_{\text{Code}}$ and $\text{Model}_{\text{Query}}$ are the repair ratio of the adaptive bits relaxing in the code hashing model and query hashing model, respectively. The definition of the repair ratio is the ratio of whether the bits predicted as unknown are the misaligned bits of the binary hash codes generated from the two hashing models in the initial hashing projection training process. Note that hash bits that are relaxed by both sides of the hashing model are not

	Model	Python		Java	
		128bit	**256bit**	**128bit**	**256bit**
CodeBERT	$\text{CoSHC}_{\text{CSSDH}}$	1.10	1.10	1.07	1.04
	$\text{DJSRH}_{\text{CSSDH}}$	1.10	1.02	1.06	1.00
	$\text{DSAH}_{\text{CSSDH}}$	1.11	1.04	1.07	1.01
	$\text{JDSH}_{\text{CSSDH}}$	1.11	1.03	1.07	0.99
GraphCodeBERT	$\text{CoSHC}_{\text{CSSDH}}$	1.11	1.09	1.08	1.01
	$\text{DJSRH}_{\text{CSSDH}}$	1.12	1.01	1.08	1.01
	$\text{DSAH}_{\text{CSSDH}}$	1.13	1.03	1.09	1.01
	$\text{JDSH}_{\text{CSSDH}}$	1.13	1.03	1.09	1.01

Table 5.6: Average hash bits that both code and query hashing models predicted as unknown in single hash code segment.

counted.

As shown in Table 5.5, the repair ratio of all the baselines with CSSDH is very close. Another finding is that the repair ratio of DJSRH is slightly higher than the other two baselines. As shown in Section 5.4.3, the initial hash projection of DJSRH is much worse than others, which provides more space for CSSDH to play its advantages. In addition, we can find the repair ratio with 256 bits higher than that with 128 bits. The reason is that the relatively minimum resolution of the hash codes increases when the hash codes get longer, making it easier for CSSDH to distinguish which hash bits have a higher probability of making mistakes.

Table 5.6 illustrates the average hash bits which both code and query hashing models predicted as unknown bits in the single hash code segment. It is unnecessary to predict as unknown in the same bit from both the hashing models since the two hash code segments can be matched as long as the misaligned hash bit is predicted as unknown in either the code hashing model or

query hashing model. On the contrary, the prediction of uncertainty in one hash bit will also reduce the hamming distance of unrelated hash codes, which may bring false positives. Therefore, the average number of unknown predictions in both code and query hashing models is smaller, and the performance of our proposed approach will be better. As shown in Table 5.6, all the baselines with CSSDH have similar performance. Similar to the condition in repair ratio, DJSRH has fewer average hash bits which both hashing models predicted as unknown. Besides, we can find the average hash bits predicted as unknown from both hashing models with 256 bits is less than the average hash bits with 128 bits. The reason for this phenomenon is similar to the condition in repair ratio.

5.5 Threats to Validity

In this chapter, we have identified the following threats to validity.

5.5.1 Threats to External Validity

From the consideration of the experiment cost, we only select one Python dataset and one Java dataset in our evaluation. Such an amount of data size may not be sufficient to demonstrate the performance and efficiency of CSSDH under huge databases.

From the consideration of experiment cost, we only select two code retrieval models with three deep hashing baselines. However, it is possible that when applying CSSDH to other models, there is no significant time boost or the accuracy may be well preserved.

At last, we only evaluate the proposed approach with the metric of R@1 and MRR in the overall performance experiment. However, these two metrics may not sufficiently reveal the performance gap between CSSDH and deep hashing baselines.

5.5.2 Threats to Internal Validity

Due to the mechanism difference between the Hash table-based approaches and Hamming distance-based approaches, our proposed approach requires a relatively large recall number, which is set as 300 in our experiment. The performance of our proposed approach may be dropped if there is a strict requirement on the recall number.

5.6 Summary

In this chapter, we have explored the efficiency aspect of code retrieval, which has received little attention in the existing literature. We propose a novel hashing approach based on existing deep hashing methods. By adopting our approach, the long hash codes from the existing deep hashing methods can be converted into several segmented hash codes and these segmented hash codes can be utilized for the construction of hash tables, which are used for the recall of code candidates. Experimental results show that CSSDH can significantly reduce the retrieval time while achieving comparable or even higher performance than previous deep hashing approaches.

□ End of chapter.

Chapter 6

Weakly Supverised Vulnerability Detection and Localization via Multiple Instance Learning

In this chapter, we investigate weakly supervised learning for vulnerability detection. Most previous approaches focus on coarse-grained vulnerability detection, such as at the function or file level. However, the developers would still encounter the challenge of manually inspecting a large volume of code to identify the specific vulnerable statements. Training the model for vulnerability localization usually requires ground-truth labels at the statement level, and labeling vulnerable statements demands expert knowledge. To tackle this problem, we propose a novel approach called WILDE for weakly supervised vulnerability detection learning, which does not need additional statement-level labels during the training. Specifically, WILDE converts the ground-truth label at the function level into pseudo labels for individual statements, eliminating the need for additional statement-level labeling. These pseudo labels are utilized to train the clas-

sifiers for the function-level representation vectors. Extensive experimentation on three popular benchmark datasets demonstrates that, in comparison to previous baselines, our approach achieves comparable performance in vulnerability detection and state-of-the-art performance in statement-level vulnerability localization.

6.1 Introduction

Software vulnerabilities are flaws in the logical design of software or operating systems that can be exploited maliciously by attackers. By exploiting these vulnerabilities, attackers can implant Trojan horses and viruses over networks, extract crucial user information, and even inflict severe damage to the system [38]. The detection of software vulnerabilities has emerged as a crucial issue in the realm of software security, garnering considerable interest from researchers and developers in recent decades.

Most traditional vulnerability detection tools [30,52,102,108, 118], such as Flawfinder [118], employ static analysis techniques to identify vulnerabilities in programs. These tools typically rely on predefined vulnerability patterns to determine whether the target programs are vulnerable. Although these tools can effectively detect well-defined vulnerabilities such as use-after-free issues, they often struggle to identify vulnerabilities that are not easily defined, such as incorrect business logic. Furthermore, the manual definition of vulnerability patterns is a time-consuming process that can hinder the efficiency of these methods. Moreover, these tools often generate a large number of false positives/negatives in their reported vulnerabilities [17],

further diminishing their utility.

With the advancement of deep learning techniques, there has been a growing interest in using these methods for vulnerability detection in recent years. Various deep learning-based approaches have been proposed by researchers, leveraging neural networks like Convolutional Neural Networks (CNNs) [92], Recurrent Neural Networks (RNNs)) [48], and Graph Neural Networks (GNNs) [15,133]. These approaches enable the automatic acquisition of vulnerability features or patterns from training data. Notably, these techniques have demonstrated their effectiveness in detecting unreported or unknown vulnerabilities [68].

However, most current deep learning-based approaches for vulnerability detection only offer predictions at the function level. This falls short of developers' needs because the majority of vulnerable statements within the code tend to be relatively concealed and challenging to uncover. Even with function-level predictions, developers still face the time-consuming task of locating these vulnerable statements. Some previous approaches have aimed to enhance the interpretability of models to help developers identify vulnerable statements. However, these methods have struggled to achieve accurate localization, as they do not prioritize the localization problem during training and solely rely on attention scores or GNN explainers [128] to explain the model's behavior after training. Automatically predicting statement-level vulnerabilities in a supervised manner poses difficulties, as it necessitates labeled data for model learning. Therefore, there is an urgent demand for unsupervised or weakly supervised approaches for statement-level vulnerability localization.

Multiple instance learning (MIL) is a weakly supervised learn-

ing that has been widely applied in various tasks such as drug activity prediction [6], image retrieval [3, 89], and text classification [59]. MIL handles training data arranged in sets called bags, where each bag contains multiple instances. It enables the construction of pseudo-labels for individual instances based on the ground-truth label of the entire bag. While multiple instance learning has been applied successfully in many fields, its application in vulnerability detection and localization tasks has not been explored. There are two main challenges to the application of multiple instance learning in this task. Firstly, when using deep learning models for code learning, the smallest unit of input is typically variable names or subtokens, rather than entire statements. Effectively generating statement-level representation vectors to capture vulnerability information becomes problematic. Secondly, conventional MIL approaches assume independence among instances within the same bag, whereas many software vulnerabilities are caused by incorrect control flow or data flow across multiple statements, necessitating interactions between different statements. To tackle these challenges, we propose a Transformer-based model within the framework of multiple instance learning. This model can effectively capture local and global vulnerability information for each statement and allows statements within the same function to interact during the training. Additionally, it retains the core concept of generating pseudo instance labels from the bag label. By adopting this approach, we can construct pseudo-labeled training instances, reducing the labor-intensive task of manually labeling vulnerabilities at the statement level.

In this chapter, we propose a novel approach named WILDE for function-level vulnerability detection with statement-level lo-

calization. WILDE first converts an input code snippet into a token sequence and feeds it into a Transformer-based encoder. During the encoding process, tokens from the same or different statements interact freely, enabling the model to learn contextual information for each statement. There are two channels aiming to capture local and global features separately. The statement-level classifier for each channel is then trained individually to determine whether the statement-level representation vectors are vulnerable or not. The results from these two classifiers are combined to produce a single prediction for a single statement. The evaluation of WILDE is conducted using three widely used datasets, and extensive experimental findings showcase that WILDE achieves comparable performance in detecting vulnerabilities at the function level compared to previous models. Furthermore, its ability to localize vulnerabilities surpasses that of the previous models.

We summarize the main contributions of this chapter as follows:

- We propose a novel approach, WILDE, to predict whether a given code snippet is vulnerable or not and meanwhile offer the vulnerability localization ability. To our best knowledge, WILDE is the first approach to adopt multiple instance learning for detecting function-level vulnerabilities and localizing vulnerabilities at the statement level, all without requiring additional vulnerability labeling at the statement level.

- We integrate various pooling modules capable of capturing code features specific to vulnerabilities. We also validate the effectiveness of each pooling module on the overall performance.

- We have performed comprehensive experiments on public benchmarks, and the results indicate that WILDE achieves comparable performance in function-level vulnerability detection and outperforms previous models in statement-level vulnerability localization, showcasing state-of-the-art performance.

6.2 Methodology

In this section, we propose a novel deep learning-based vulnerability detection approach that can achieve function-level detection and statement-level localization simultaneously with weakly supervised learning. Specifically, we present the overview and detailed design of the proposed approach WILDE, including model design, model training strategy, and inference strategy.

6.2.1 Overview

Figure 6.1 illustrates an overview of the proposed approach WILDE. Our approach consists of three steps: code encoding, multiple instance learning-based training strategy, and model inference. In the step of code encoding, the Transformer-based encoder learns to generate representation vectors for each statement within the given function. Two linear classifiers are trained to classify whether these statement-level representation vectors indicate vulnerability or not. In the multiple instance learning-based training strategy, we convert function-level ground-truth labels for vulnerability detection into statement-level pseudo labels and utilize these pseudo labels for the model training. During the inference step, the model also generates representation vectors for each statement and employs the previous two trained linear classifiers to determine their vulnerability status. Addition-

ally, the overall vulnerability prediction for the entire function is determined by considering the vulnerability predictions of each statement within the function. Further details about these three steps will be presented in the following sections.

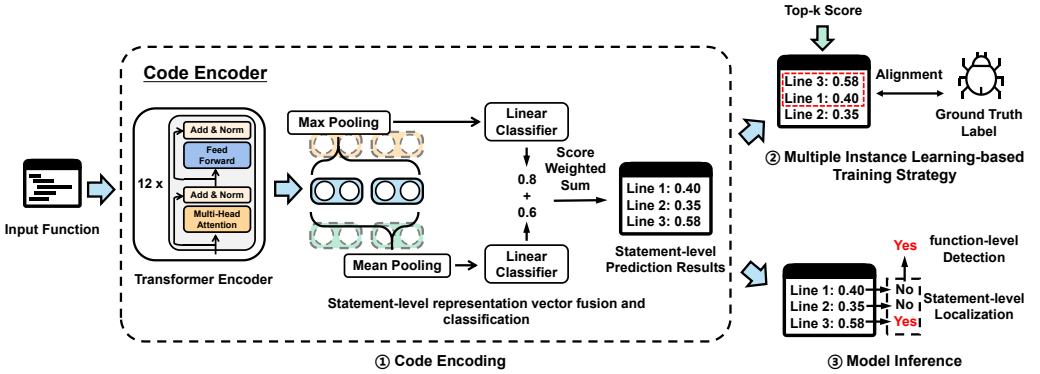


Figure 6.1: An overview architecture of WILDE, containing three main steps. ① Code Encoding: The target function will first be transformed into a token sequence and then fed into a Transformer-based Encoder. Subsequently, the token vectors will be integrated into statement-level vectors, and two linear classifiers will be employed to classify them. ② Multiple Instance Learning-based Training Strategy: The statements will be ranked in descending order based on the statement-level predicted results. The top-k statements will then be assigned pseudo labels identical to the function label for model training purposes. ③ Model Inference: The results obtained in Step 1 will be utilized to predict the vulnerability of each statement. The prediction results from all the statements will then be used to determine the vulnerability of the entire function.

6.2.2 Code Encoding

The given code snippet will be converted into a token sequence and each token will be split into subwords by the tokenizer. We adopt the Byte Pair Encoding (BPE) approach [96] from Roberta as our tokenizer to tokenize the word. To incorporate the positional relationships between the subwords into the

model, positional embedding vectors are encoded to represent the token's position within the token sequence. The token embedding vector and the positional embedding vector are then combined into a unified representation vector, which represents the corresponding token within the input sequence.

It is necessary to record the location information of each token for the generation of statement-level representation vectors in subsequent stages. In order to capture this information, we create a binary statement indicative matrix. The matrix, denoted as S, is defined as follows:

$$S = \{s_{11}, ..., s_{1n}, ..., s_{m1}, ..., s_{mn}\}, \tag{6.1}$$

where n is the token number, m is the statement number, and s_{ij} indicates whether the i-th token belongs to the j-th statement in the given function. The value of s_{ij} will be 1 if the i-th token belongs to the j-th statement in the given function; otherwise, s_{ij} will be zero.

In the subsequent sections, we will introduce the design of the code encoder, which takes both the token embedding sequence and the binary statement indicative matrix as inputs and generates statement-level prediction scores.

6.2.3 The Design of Code Encoder

In this subsection, we present the code encoder of our proposed approach WILDE. The code encoder consists of a Transformer-based encoder, as well as linear classifiers for both the max pooling channel and the mean pooling channel.

Transformer Encoder with Self-attention

In our approach WILDE, we utilize a Transformer-based encoder. The encoder comprises 12 stacked Transformer blocks, each consisting of a multi-head self-attention layer and a fully connected feed-forward neural network. The multi-head self-attention layer's purpose is to generate the attention vector based on the attention score assigned to each code token. To accomplish this, the dot product between the query vector of the current code token and the key vectors of the other tokens is computed. Subsequently, the dot product is normalized to probabilities via the Softmax function. Finally, the attention vector is obtained by taking the dot product between the value vectors and previous normalized probabilities. The equation for calculating the attention score is provided below:

$$Attention(Q, K, V) = \text{Softmax}(\frac{QK^T}{\sqrt{d_k}}V), \tag{6.2}$$

where Q, K , V is the query vector, key vector, and value vector, respectively.

The multi-head mechanism enables the model to create several subspaces, each dedicated to different aspects of the input sequence. This allows the model to effectively capture diverse semantic information from the input. Initially, the multi-head mechanism divides the input vectors into h heads, with each head having a dimension of $\frac{d}{h}$. Following the self-attention operation on each head, these heads are then concatenated back together as:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O, \tag{6.3}$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$ and W^O is the projection matrix for the concatenated vectors.

Finally, the concatenated vectors will be fed into a fully connected feed-forward neural network. This neural network comprises two linear layers, with a ReLU activation function sandwiched between them.

Statement-level Representation Vector Fusion

After the encoding of the input token sequence from the previous Transformer-based encoder, we obtained the representation vector for each token from the last hidden states in the model. Since the number of tokens in each statement varies, it is necessary to merge the embedding vectors of tokens belonging to the same statement into a single statement-level representation vector. To achieve efficient fusion of these vectors, we establish two channels for vector fusion: max pooling and mean pooling. These channels effectively capture informative features from different perspectives.

```
1  char* trimTrailingWhitespace(char *strMessage, int
       length) {
2  char *retMessage;
3  char *message = malloc(sizeof(char)*(length+1));
4
5  // copy input string to a temporary string
6  char message[length+1];
7  int index;
8  for (index = 0; index < length; index++) {
9        message[index] = strMessage[index];
10 }
11 message[index] = '\0';
12
13 // trim trailing whitespace
14 int len = index-1;
15 while (isspace(message[len])) {
16     message[len] = '\0';
17     len--;
18 }
19
20 // return string without trailing whitespace
21 retMessage = message;
22 return retMessage;
23 }
24 ...
25 }
```

Code Listing 6.1: A motivating example for max pooling. The code is simplified due to the space limit.

Max Pooling Channel: Max pooling is a down-sampling technique commonly employed in deep learning to effectively retain local features from the original feature. This characteristic holds significant importance in vulnerability detection, as numerous software vulnerabilities arise from variable or API misuse. An illustrative example is provided in Listing 6.1 from CWE-787, demonstrating the *Out-of-bounds Write* vulnerability. In line 16

of the code snippet, the variable `len` is employed as an array index, potentially leading to a buffer underwrite issue if the input consists solely of whitespaces. Consequently, the while statement may traverse beyond the beginning of the string, thereby invoking the `isspace()` API on an address outside the limits of the local buffer on certain systems. To resolve this vulnerability, it is necessary to validate the value of the variable `len` before utilizing it as an array index. By employing the max pooling function, it becomes possible to detect improper variable usage within the statement by capturing the local features from the vector of the improper variable.

The operation of max pooling is illustrated as follows:

$$v_{max\_j} = max(h_1 \cdot s_{1j}, ..., h_n \cdot s_{nj}), \tag{6.4}$$

where v_{max_j} is the representation vector for the locality information in the j-th statement, h_i is the hidden vector for the i-th token from the encoder, and s_{ij} is the indicator to show whether the i-th token belongs to the j-th statement. $h_i \cdot s_{ij}$ can remove the irrelevant token vectors during the operation of max pooling.

```
1  #define JAN 1
2  #define FEB 2
3  #define MAR 3
4
5  short getMonthlySales(int month) {...}
6
7  float calculateRevenueForQuarter(short quarterSold)
       {...}
8
9  int determineFirstQuarterRevenue() {
10
11 // Variable for sales revenue for the quarter
12 float quarterRevenue = 0.0f;
13
```

```
14 short JanSold = getMonthlySales(JAN); /* Get sales
       in January */
15 short FebSold = getMonthlySales(FEB); /* Get sales
       in February */
16 short MarSold = getMonthlySales(MAR); /* Get sales
       in March */
17
18 // Calculate quarterly total
19 short quarterSold = JanSold + FebSold + MarSold;
20
21 // Calculate the total revenue for the quarter
22 quarterRevenue = calculateRevenueForQuarter(
       quarterSold);
23
24 saveFirstQuarterRevenue(quarterRevenue);
25
26 return 0;
27 }
```

Code Listing 6.2: A motivating example for mean pooling.

Mean Pooling Channel: Mean pooling is a down-sampling technique commonly used in deep learning to effectively preserve global features from the original input. This technique is particularly useful for identifying some software vulnerabilities caused by overflow. An illustrative example from CWE-190, specifically "Integer Overflow" or "Wraparound," is presented in Listing 6.2. In line 19, the values of variables `JanSold`, `FebSold`, and `MarSold` are summed and assigned to the variable `quarterSold`. However, there exists a potential risk of integer overflow if the sum exceeds the maximum value allowed for the short int primitive type. Integer overflow can result in severe consequences such as data corruption, unexpected behavior, infinite loops, or system crashes. To mitigate this issue, it is necessary to validate the sum before performing the assignment.

The operation of mean pooling is shown as follows:

$$v_{mean\_j} = \frac{\sum_{i=1}^{n} h_i \cdot s_{ij}}{\sum_{i=1}^{n} s_{ij}}, \tag{6.5}$$

where v_{mean_j} is the representation vector for the global information in the i-th statement, h_i is the hidden vector for the i-th token from the encoder, and s_{ij} is the indicator to show whether the i-th token belongs to the j-th statement. $h_i \cdot s_{ij}$ can remove the irrelevant token vectors during the operation of mean pooling.

Representation Vector Classification and Fusion

Upon combining the original embedding vectors of the tokens within a statement, we generate two representation vectors that encompass both local and global information. Then we employ two linear classifiers to classify these two representation vectors as the binary classification task, respectively. Each classifier consists of a fully connected layer followed by a softmax activation function. The scores produced by these classifiers are then combined through a weighted sum, resulting in a single score that serves as the final prediction for the statement.

6.2.4 Multiple Instance Learning-Based Training Strategy

Despite generating the representation vector and utilizing the classifier for individual statements, the absence of statement-level labels poses an ongoing challenge. Inspired by the concept from multiple instance learning, we convert the label of the entire function into pseudo labels for each statement within the

target function to tackle this issue. These pseudo-labels serve as the supervised signal during training.

It is known that a vulnerable function must contain at least one vulnerable statement, whereas a non-vulnerable function has no vulnerable statements. In a given code snippet, the label y_i represents the vulnerability status of the i-th statement, where y_i is 0 for non-vulnerable statements and 1 for vulnerable statements. The label Y indicates whether the code snippet as a whole is vulnerable or not. Thus, we can determine the label of the entire code snippet as follows:

$$Y = \max\{y_1, ..., y_n\}, \tag{6.6}$$

where n is the number of statements in the code snippet.

By referring to Equation 6.6, it becomes evident that the label of every statement within a function labeled as 0 will also be 0. However, there are two challenges in determining the statement labels within a function labeled as 1. Firstly, the vulnerable function contains only a few vulnerable statements, while the majority of statements inside the function are non-vulnerable. This raises the issue of how to assign labels to these statements. Secondly, even if we solve the problem of label assignment for the vulnerable function, we encounter another challenge with the label ratio. In non-vulnerable functions, the statement labels will be predominantly 0, and the same applies to most of the statement labels in the vulnerable function. The ratio of pseudo-positive/negative labels becomes much smaller than the ratio of the original positive/negative labels in the dataset. The sample ratio imbalance will lead the model to favor predicting functions as non-vulnerable, impacting its overall performance [70].

We address the first issue based on the assumption that the

non-vulnerable statements exhibit a distinct pattern that differs from the vulnerable statement. This pattern can be detected and distinguished by the classifier model. Specifically, we sort the statements within the function in descending order, based on their previous classification prediction scores. Next, we generate pseudo labels for the top k statements, assigning them the same label as the entire function for training purposes. The value of k can be determined as the average number of vulnerable statements, as indicated by the dataset statistics. It is important to note that the pseudo labels for the vulnerable statements are not accurate at the beginning of the training. However, the pseudo labels for the non-vulnerable statements must be accurate because all the statements in the non-vulnerable statements are non-vulnerable. As negative samples are incorporated into the training process, the prediction scores for statements that are semantically similar to non-vulnerable statements will progressively decrease. The pseudo labels for vulnerable statements will gradually become more relatively accurate as training progresses [14].

To address the second issue, we set the training objective exclusively for the top-k statement within the given function, regardless of whether the function is vulnerable or non-vulnerable. This operation can effectively address the issue of imbalanced sample ratios resulting from the conversion of pseudo labels. From the experiment results, we find that the selection of hyperparameter k will have a slight effect on the overall performance, which will be discussed in Section § 6.4.2

In summary, the loss function used in WILDE is cross-entropy loss, which is defined as follows:

$$loss = \frac{1}{N \cdot k} \sum_{i} \sum_{j}^{k} -[Y_i log(p_{ij}) + (1 - Y_i) log(1 - p_{ij})], \quad (6.7)$$

where N is the number of the function, Y_i is the label for the function i, p_{ij} is the vulnerability probability for the j-th statement in the function i, and k is the pre-defined parameter.

6.2.5 Model Inference

During the inference stage, WILDE does not provide a direct prediction of the vulnerability of the entire function. Instead, WILDE focuses on predicting vulnerable conditions for each statement within the function. The determination of the function-level prediction relies on the results obtained at the statement level. If at least one statement within the function is predicted as vulnerable, the function is considered vulnerable as well. Conversely, if no statements are predicted as vulnerable, the function is deemed non-vulnerable.

Two methods for localizing vulnerabilities are employed: absolute label prediction and relative score ranking. In the absolute label prediction method, the model identifies and reports only the statements it predicts as vulnerable. On the other hand, the relative score ranking method involves evaluating the statements within the function based on their prediction scores and sorting them in descending order. The top k statements are then selected as potential candidates for vulnerable statements, which are presented to the users. However, it is important to note that the absolute label prediction might not be accurate enough since there is no ground-truth label provided for training. Therefore, we recommend utilizing the relative scores ranking method as

	Fan *et al.*			Reveal			FFMPeg+Qemu		
	Train	**Valid**	**Test**	**Train**	**Valid**	**Test**	**Train**	**Valid**	**Test**
Vul function	4,993	624	626	801	98	104	7,078	887	879
Non-vul function	142,188	17,774	17,774	10,371	1,256	1,296	8,526	1,058	1,024
Avg stat num	20.12	20.69	20.41	N/A	N/A	N/A	N/A	N/A	N/A
Avg vul stat num	3.03	3.27	3.28	N/A	N/A	N/A	N/A	N/A	N/A

Table 6.1: Statistics of dataset.

the preferred approach for vulnerability localization. The performance of both methods will be further discussed in the following section.

6.3 Experimental Setup

In this section, we provide an overview of the statistics information for the dataset used in our study, the steps taken for data pre-processing, the baseline models employed, the evaluation metrics utilized, and the implementation details concerning both our proposed tool and the other baseline models included in our experiment.

6.3.1 Data Pre-processing

In our experiment, we evaluate the performance of vulnerability detection and localization using three datasets: FFMPeg+Qemu [133], Reveal [15], and Fan et al. [31]. The FFMPeg+Qemu dataset, collected by Devign, comprises data from two open-source C projects and has been labeled by experts. It consists of approximately 10,000 vulnerable functions and 12,000 non-vulnerable functions. The Reveal dataset is obtained from Linux Debian Kernel and Chromium, containing about 2,000 vulnerable functions and 20,000 non-vulnerable functions. Fan

et al. dataset, a C/C++ dataset, is gathered from over 300 open-source GitHub projects, covering 91 different Common Vulnerabilities and Exposures (CVE) databases from 2002 to 2019. This dataset includes around 10,000 vulnerable functions and 177,000 non-vulnerable functions. Although the purpose of vulnerability detection in this thesis is to ensure whether the retrieved code is vulnerable or not, here we split the workflow of this thesis into code retrieval and vulnerability detection separately for evaluation. Therefore, we only utilize the datasets for vulnerability detection in this chapter.

The FFMPeg+Qemu and Reveal datasets only provide labels indicating whether a given function is vulnerable or not. Therefore, we solely assess the performance of function-level vulnerability detection using these two datasets. In contrast, Fan et al. [31] not only offer function-level labels but also provide the fixed version of the vulnerable function. This allows us to pinpoint the vulnerable statements by comparing the function before and after fixing. Therefore, we can evaluate both function-level vulnerability detection and statement-level vulnerability localization on this dataset.

We have imposed a length limitation on the input token sequence to accommodate the fixed-length input requirement of Transformer. Any token exceeding the maximum input length is discarded. However, the vulnerable statements in the code snippets may be contained in the discarded tokens, which means that the input statements are not vulnerable although the label for the entire function is vulnerable. To address this label conflict, we remove code snippets whose function-level label is vulnerable but do not contain any vulnerable statements in the input to our model. Additionally, different baseline models used in our ex-

periments necessitate different data pre-processing tools. Some of the data in our dataset cannot be processed correctly by all of these tools. In the interest of experimental fairness, we only retain the data that can be processed by all data pre-processing tools.

Within Table 6.1, "Vul function" denotes the number of vulnerable functions in the dataset, while "Non-vul function" represents the number of non-vulnerable functions. "Avg stat num" indicates the average number of statements in a single function within the dataset, and "Avg vul stat num" signifies the average number of vulnerable statements in a single function. As previously mentioned, Reveal and FFMPeg+Qemu do not provide information regarding vulnerable statements, thus preventing us from offering statistics on the average statement number and average vulnerable statement number for these two datasets.

6.3.2 Implementation Details

In our proposed WILDE, we set the number of encoder layers to 12, the number of attention headers to 12, and the hidden size to 768. The batch size and learning rate were set to 16 and 2e-5, respectively. Our model supports a maximum input token length of 512. As the hyperparameter top-k is sensitive to the average number of vulnerable statements in the target dataset, which can vary between datasets, we experimented with values of 1, 3, and 5 for top-k and selected the best performance for each dataset. For optimization, we utilized the AdamW [75] optimizer. We set a maximum of 50 epochs for the training with 10-step patience for early stopping.

We replicated all the baselines, except for Devign, using publicly released source code and adopted the same hyperparameter

settings as described in their original paper. For Devign, since they did not make their code public, we reproduced it based on the code provided by Chakraborty et al [15]. In the case of the baseline model called IVDetect, it requires the training dataset to have an equal ratio of vulnerable and non-vulnerable functions. Since none of the datasets we used had this ratio, we retained all the vulnerable functions and randomly selected an equal number of non-vulnerable functions from each training dataset to create a new training dataset for IVDetect. There are two versions of the baseline model named LineVul, which are LineVul with pre-training and LineVul without pre-training. We observed that LineVul with pre-training performed exceptionally well on the dataset of Fan et al, while there was no significant difference between LineVul with and without pre-training on the other two datasets. This led us to suspect the presence of a data leakage problem specifically in the Fan et al dataset so we excluded the pre-training model from our experiments. All the models were trained on a server equipped with NVIDIA A100-SXM4. The training process for WILDE consumed approximately 10 GPU hours.

6.3.3 Baselines

We compare WILDE with six state-of-the-art vulnerability detection methods, including two token-based methods [68, 69], three structure-based methods [15, 66, 133], and one unsupervised statement-level detection method [35]. Here, we provide a brief description of these baseline methods:

(1) VulDeePecker [69]: VulDeePecker extracts code gadgets from the given code snippet, which are several lines of code that are semantically related to each other, and adopts the bi-

directional LSTM-based neural network with an attention mechanism for vulnerability detection.

(2) SySeVR [68]: SySeVR extracts the vulnerability syntax characteristics (SyVCs) from the given function at first and then transforms these SyVCs into semantics-based vulnerability candidates (SeVCs) which contain the statements related to the given SyVCs via the program slicing technique. Finally, a bi-directional recurrent neural network is employed to encode these SeVCs into vectors, facilitating the detection of vulnerable code snippets.

(3) Devign [133]: Devign extracts the information of abstract syntax tree (AST), control flow graph (CFG), data flow graph (DFG), and code token sequence from the given function to construct the graph which can represent the given functions and generate the embedding vector for each node inside the graph. Subsequently, the graph is inputted into a Gated Graph Neural Network (GGNN) for classification training.

(4) Reveal [15]: Reveal extracts the information of Code Property Graph (CPG) from the given function to construct the representation graph and adopts the technique of Word2Vec to generate the embedding vector for each node. The resulting graph is then inputted into GGNN, where all the vectors from the representation graph are combined into a single vector. This fused vector serves as the representation for the entire graph, facilitating vulnerability detection.

(5) IVDetect [66]: IVDetect is a code analysis tool that divides the code into multiple statements and extracts various features from each statement. These features encompass sub-token sequences, AST sub-trees, variable names, variable types, data dependency context, and control dependency context. To

capture these features, they are embedded into representation vectors. Subsequently, these vectors are combined into a unified representation vector for each statement using an attention-based bi-directional GRU. These statement-level representation vectors serve as node embedding features, which are then fed into a Graph Convolutional Network (GCN) to acquire a comprehensive graph representation for detection purposes.

(6) LineVul [35]: LineVul adopts the Byte Pair Encoder (BPE) technique to tokenize the given code into a sub-token sequence and utilize a 12-layer Transformer based model for vulnerability detection. Not only predicting the function-level vulnerability, LineVul can also localize the vulnerable statement by calculating the attention scores for each sub-token.

6.3.4 Evaluation Metrics

In our experiment, we adopt four metrics, which are Acc, P, R, and F1, to evaluate the performance of all the models in function-level vulnerability detection. Additionally, we utilized nine metrics, which are Top-1, Top-5, Top-10, MFR, MAR, IFA, P, R, F1, to evaluate the performance of all the models in statement-level vulnerability localization.

The metric employed to determine the accuracy of the model is denoted as Acc. It quantifies the ratio of accurately classified samples to the total number of samples. The definition of Acc is presented below:

$$Acc = \frac{S_c}{S}, \tag{6.8}$$

where S_c represents the number of samples correctly labeled by the model, while S denotes the total number of samples. A

higher value of Acc indicates a better performance of the model.

P is the metric used to assess the accuracy of the model's detection of vulnerable samples. The definition of P is provided below:

$$P = \frac{TP}{TP + FP}, \tag{6.9}$$

where TP represents the count of samples where both the label and the model's prediction are true, while FP refers to the count of samples where the label is true but the model's prediction is incorrect. A higher value for P signifies improved performance of the model.

R is a metric used to assess the percentage of vulnerable samples correctly detected out of all the vulnerable samples predicted by the model. The specific definition of this metric is provided below:

$$R = \frac{TP}{TP + FN}, \tag{6.10}$$

where TP represents the count of samples with true labels that the model correctly predicts, while FN represents the count of samples with false labels that the model incorrectly predicts. A higher value of R signifies superior performance of the model.

F1 is a metric that represents the harmonic mean of precision and recall. It is commonly employed to assess a model's performance by taking into account both precision and recall. The formula for calculating F1 is as follows:

$$F1 = 2 \times \frac{P \times R}{P + R}, \tag{6.11}$$

where P represents the precision of the model, and R denotes the recall of the model. A higher F1 score indicates superior

performance of the model.

Top-k is a metric used to assess the model's ability to identify vulnerable statements among the top k results it returns. The definition of Top-k is as follows:

$$Top - k = \frac{1}{|S_v|} \sum_{s=1}^{S_v} \delta(FRank_s \leq k), \tag{6.12}$$

where S_v represents the count of vulnerable functions, and $FRank_s$, denotes the ranking assigned to the first vulnerable statement in the statement set. A higher value for Top-k signifies improved performance in vulnerability localization.

MFR (Mean First Ranking) is calculated as the average of the rankings assigned to the first vulnerable statement among the returned statements. The formula for calculating MFR is provided below:

$$MFR = \frac{1}{|S_v|} \sum_{s=1}^{S_v} FRank_s \tag{6.13}$$

A lower value of MFR indicates superior performance in vulnerability localization.

MAR (Mean Average Ranking) is calculated as the average ranking across all vulnerable statements present in the returned statements. The formula for MAR is provided below:

$$MAR = \frac{1}{|S_v|} \sum_{s=1}^{S_v} \frac{1}{|N|} \sum_{i=1}^{N} Rank_{si}, \tag{6.14}$$

where $Rank_{si}$ represents the ranking of the i-th vulnerable statement within the returned statements of the s-th vulnerable function. A lower MAR value indicates superior performance in terms of vulnerability localization.

IFA (Initial False Alarm) is a metric that quantifies the number of statements that are erroneously predicted as vulnerable by the models before correctly identifying the first vulnerable statement. The definition of this metric is provided below:

$$IFA = \frac{1}{|S_v|} \sum_{s=1}^{S_v} (FRank_s - 1) \tag{6.15}$$

A lower value of IFA indicates superior performance in vulnerability localization.

6.4 Experimental Results

In this section, we first present the experimental results and assess the performance of WILDE in terms of function-level vulnerability detection and statement-level vulnerability localization. Secondly, we evaluate the impact of Top-K statement selection on the overall performance. Thirdly, we investigate the contribution of each channel to the overall performance. Fourthly, we examine the influence of data size on both function-level vulnerability detection and statement-level vulnerability localization abilities. Lastly, we evaluate the ability of WILDE to detect different types of vulnerability.

6.4.1 Comparison on function-level vulnerability detection and statement-level vulnerability localization

Table 6.2 illustrates the comparison results of function-level vulnerability detection performance. In the datasets of Fan et al. and Reveal, where there is an imbalance in the proportion of

Model	Fan *et al.*				Reveal				FFMPeg+Qemu			
	Acc	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**
VulDeePecker	0.913	0.155	0.146	0.150	0.763	0.211	0.131	0.162	0.496	0.461	0.326	0.381
SySeVR	0.904	0.129	0.194	0.155	0.743	0.401	0.249	0.307	0.479	0.461	0.588	0.517
Devign	0.957	0.257	0.143	0.184	0.875	0.316	0.367	0.339	0.569	0.525	0.647	0.580
Reveal	0.928	0.270	**0.661**	0.383	0.818	0.316	**0.611**	0.416	**0.611**	**0.555**	0.707	0.622
IVDetect	0.696	0.073	0.600	0.130	0.808	0.276	0.556	0.369	0.573	0.524	0.576	0.548
LineVul	0.972	0.632	0.436	0.516	0.847	0.248	0.519	0.335	0.541	0.496	**0.909**	**0.642**
WILDE	**0.977**	**0.724**	0.522	**0.607**	**0.922**	**0.471**	0.394	**0.429**	0.589	0.530	0.812	0.641

Table 6.2: Comparison results on function-level vulnerability. The best results are highlighted in **bold** font.

Model	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
LineVul	N/A	N/A	N/A	N/A	9.49	7.17	6.17	0.005	0.252	0.375
WILDE	0.983	0.183	0.338	0.237	**9.08**	**6.46**	**5.46**	**0.283**	**0.484**	**0.609**

Table 6.3: Comparison results on function-level vulnerability and statement-level vulnerability localization. The best results are highlighted in **bold** font.

positive and negative examples, the F1 metric holds more significance compared to other metrics. The results reveal that WILDE outperforms other approaches and achieves state-of-the-art performance in terms of F1 on both the Fan et al. and Reveal datasets. Notably, WILDE demonstrates a relative improvement of 17.6% and 3.1% in F1 on the Fan et al. and Reveal datasets, respectively. Regarding the FFMPeg+Qemu dataset, while WILDE does not surpass all the baselines, its performance is closely aligned with the state-of-the-art baseline, with only a 0.2% difference in F1. These findings demonstrate that WILDE can attain performance comparable to the current state-of-the-art baselines for function-level vulnerability detection.

Table 6.3 presents the results of statement-level vulnerability localization performance for our proposed approach and the baselines. Due to the availability of sentence-level annotation labels in the dataset by Fan et al., we exclusively display the experimental results for this dataset. To evaluate accuracy, precision, recall, and F1, we employ the first method outlined in Section § 6.2.5, which utilizes absolute label prediction. For the remaining metrics, we utilize the second method introduced in Section § 6.2.5, employing relative scores to ensure a fair comparison between WILDE and the baseline.

LineVul employs the attention score for each statement to estimate the likelihood of a statement being vulnerable, rather than

directly determining its vulnerability. The metrics such as accuracy, precision, recall, and F1 are not applicable to this baseline. Instead, we present the results of our proposed WILDE. A comparison between the results in Table 6.2 and Table 6.3 reveals that the ability of WILDE to detect vulnerabilities at the statement level is inferior to its ability to detect vulnerabilities at the function level. Therefore, relying solely on the statement-level predictions from WILDE may not be advisable. It can be easily understood that it is quite hard to localize the vulnerable statement since there is no explicit ground-truth label for the training. Moreover, the recall of WILDE is significantly higher than its precision according to the results from Table 6.3, indicating that WILDE tends to predict statements as vulnerable at the cost of a higher false alarm rate. Comparing the performance of WILDE and the baselines on the other metrics in Table 6.3, we observe that our proposed WILDE achieves state-of-the-art performance in statement-level vulnerability localization across all metrics. Notably, there is a substantial improvement in the Top-1 metric and significant improvements in the Top-3 and Top-5 metrics. Unlike previous approaches that lack supervised signals in the attention score, our mechanism enhances vulnerability localization by providing pseudo labels during training, resulting in improved localization ability. Furthermore, the accuracy of the top 5 predictions from WILDE is considerably higher than the F1 score, suggesting that our proposed WILDE can effectively notify users about vulnerable statements by ranking their relative scores when the target function is predicted to be vulnerable.

In conclusion, the comprehensive experiment results show that WILDE can achieve a comparable function-level vulnerability

detection performance and state-of-the-art statement-level vulnerability localization performance compared to previous baselines, which demonstrates the effectiveness of our proposed WILDE.

6.4.2 Impact of the top-k statement selection on the performance of WILDE

Table 6.4 presents the impact of the Top-K hyperparameter on the performance of function-level vulnerability detection models. It is evident that there is no universally optimal fixed value for top-k that guarantees optimal performance across all datasets. This is primarily due to the variation in the average number of vulnerable statements within functions in each dataset. Our proposed WILDE employs pseudo statement-level labels as supervised signals for vulnerability localization during training. However, if the predefined top-k hyperparameter does not align with the actual number of vulnerable statements in a function, incorrect pseudo labels may be assigned. Setting a larger or smaller value of k can result in misclassifying non-vulnerable statements as vulnerable or missing some vulnerable statements, introducing noise into the model and adversely affecting performance. The results from Section Table 6.1 reveal that the average number of vulnerable statements in the Fan et al. dataset is approximately 3, which explains why the best performance is achieved when k is set to 3 in this dataset. As we lack information about the number of vulnerable statements in other datasets, we cannot determine if similar patterns exist in those cases.

Table 6.5 illustrates how the hyperparameter of Top-K influences the model performance of statement-level vulnerability localization. We observed an interesting phenomenon: different

Model	Fan et al.				Reveal				FFMPeg+Qemu			
	Acc	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**
$\text{WILDE}_{\text{top}-1}$	0.976	0.724	0.481	0.578	0.913	0.415	0.423	0.419	**0.589**	**0.530**	0.812	**0.641**
$\text{WILDE}_{\text{top}-3}$	**0.977**	**0.724**	0.522	**0.607**	0.894	0.365	**0.471**	0.397	0.575	0.519	**0.824**	0.637
$\text{WILDE}_{\text{top}-5}$	0.974	0.653	**0.524**	0.582	**0.922**	**0.471**	0.394	**0.429**	0.576	0.520	**0.824**	0.638

Table 6.4: Results of the function-level vulnerability detection performance comparison with different Top-K selection. The best results among the three variants of WILDE are highlighted in **bold** font.

Model	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
$\text{WILDE}_{\text{top}-1}$	**0.987**	0.155	0.142	0.148	9.24	6.53	5.53	0.219	0.446	0.577
$\text{WILDE}_{\text{top}-3}$	0.983	**0.183**	0.338	**0.237**	9.08	6.46	5.46	0.283	0.484	**0.609**
$\text{WILDE}_{\text{top}-5}$	0.979	0.164	**0.380**	0.229	**9.02**	**6.43**	**5.43**	**0.294**	**0.497**	0.605

Table 6.5: Results of the statement-level vulnerability localization performance comparison with different Top-K selection. The best results among the three variants of WILDE are highlighted in **bold** font.

hyperparameter values for top K lead to varying tendencies in metrics for absolute label prediction (accuracy, precision, recall, and F1) and relative scoring (MFR, MAR, IFA, Top-1, Top-3, and Top-5), as described in Section § 6.2.5. Specifically, the model with a k value of 1 performs the worst across all metrics (Due to the imbalance proportion of the positive examples and negative examples, the metric of accuracy is meaningless under this setting). The mislabeling of the statements can be the reason leading to these results. During training, a significant number of vulnerable statements are incorrectly labeled as non-vulnerable. Unfortunately, the model learns this bias and it ultimately results in a decline in performance. Interestingly, the model with a k value of 3 performs better in absolute label prediction metrics compared to the model with a k value of 5, but worse in relative scoring metrics. A higher k value indicates that fewer vulnerable statements are mistakenly labeled as non-vulnerable, but more non-vulnerable statements are incorrectly labeled as vulnerable during training. This bias causes the model to be more inclined to predict the target sentence as vulnerable, resulting in a higher recall score but lower precision. The experimental results from Table 6.4 and Table 6.5 support this interpretation. While a higher k value may mislabel more non-vulnerable statements as vulnerable, the mislabeled

non-vulnerable statements still exhibit a similar pattern to non-vulnerable statements in non-vulnerable functions, whereas real vulnerable statements differ from non-vulnerable statements in non-vulnerable functions. This characteristic causes the model to predict those mislabeled non-vulnerable statements as vulnerable, but their relative score will be lower than truly vulnerable statements. As a result, the model achieves a more accurate relative ranking of vulnerable statements. These findings suggest that a higher k value may be more appropriate if we intend to use WILDE as a tool to prompt users about vulnerable statements through relative ranking when the target function is predicted as vulnerable.

In conclusion, the comprehensive experimental results demonstrate that the choice of top-k significantly impacts the performance of WILDE in both function-level vulnerability detection and statement-level vulnerability localization.

6.4.3 Impact of different channels on the performance of WILDE

In this experiment, we conducted an analysis of the channels utilized in our model to determine their contribution to the performance of WILDE. The results of the function-level vulnerability detection experiment using different channels are presented in Table 6.6. It is observed that the model combining max pooling and mean pooling achieves the best performance across all datasets, except for the Reveal dataset. This outcome highlights the effectiveness of fusing max pooling and mean pooling. There are two potential explanations for why the model with only max pooling performs best in terms of the F1 metric in the dataset of Reveal. Firstly, the size of the train-

Model	Fan et al.				Reveal				FFMPeg+Qemu			
	Acc	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**
$\text{WILDE}_{\text{max}}$	0.976	0.721	0.500	0.591	**0.933**	**0.574**	0.375	**0.453**	0.567	0.513	**0.816**	0.630
$\text{WILDE}_{\text{mean}}$	0.973	0.660	0.446	0.532	0.922	0.470	0.385	0.423	0.560	0.509	0.752	0.608
WILDE	**0.977**	**0.724**	**0.522**	**0.607**	0.922	0.471	**0.394**	0.429	**0.589**	**0.530**	0.812	**0.641**

Table 6.6: Results of the function-level vulnerability detection performance comparison with different channels. The best results are highlighted in **bold** font.

Model	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
$\text{WILDE}_{\text{max}}$	**0.984**	**0.188**	0.299	0.231	9.17	6.66	5.66	0.268	0.481	**0.617**
$\text{WILDE}_{\text{mean}}$	0.977	0.155	**0.416**	0.226	9.78	7.27	6.27	0.224	0.443	0.586
WILDE	0.983	0.183	0.338	**0.237**	**9.08**	**6.46**	**5.46**	**0.283**	**0.484**	0.609

Table 6.7: Results of the statement-level vulnerability localization performance comparison with different channels. The best results are highlighted in **bold** font.

ing data could be a factor. Since the Reveal training dataset contains only around 800 vulnerable functions, the training of the model becomes unstable. As WILDE has a more complex structure compared to $\text{WILDE}_{\text{max}}$, overfitting occurs. Secondly, the specific vulnerability type prevalent in the Reveal dataset might play a role. As explained in Section § 6.2.3, the design of max pooling and mean pooling aims to capture features associated with different vulnerability types. It is possible that most vulnerabilities in the Reveal dataset align closely with the vulnerability type effectively captured by max pooling. Consequently, mean pooling may have limited contribution or even adversely affect the overall model performance. Another noteworthy finding is that $\text{WILDE}_{\text{max}}$ performs very similarly to WILDE, while $\text{WILDE}_{\text{mean}}$ exhibits considerably worse performance than $\text{WILDE}_{\text{max}}$. This discrepancy may be attributed to the fact that most vulnerabilities are related to specific keywords within statements, and these features are effectively captured by the max pooling mechanism. Mean pooling, on the other hand, averages the information from each token in the statement, potentially diluting these keyword features and thereby diminishing the model's performance.

Table 6.7 presents the experimental results for statement-level vulnerability localization using different channels. The results

demonstrate that our WILDE performs the best across almost all metrics, showcasing the effectiveness of the fusion of max pooling and mean pooling. Similar to the findings in Table 6.6 for function-level vulnerability detection, the performance of $\text{WILDE}_{\text{mean}}$ is significantly lower compared to $\text{WILDE}_{\text{max}}$. However, $\text{WILDE}_{\text{max}}$ exhibits performance levels very close to WILDE in terms of relative scores metrics discussed in Section § 6.4.1. Additionally, the fusion of max pooling and mean pooling helps the model accurately identify vulnerable statements, resulting in improved rankings. This fusion particularly enhances the Top-1 metric performance, with diminishing improvements as the top-k value increases. Regarding absolute label prediction metrics, the performance gap is not as substantial as that observed in relative scores metrics. This suggests that $\text{WILDE}_{\text{mean}}$ possesses a similar ability to predict individual statements within a given function, as $\text{WILDE}_{\text{max}}$. Interestingly, $\text{WILDE}_{\text{mean}}$ achieves the best performance in terms of recall, indicating its capability to detect more vulnerable statements, albeit with a higher number of false positives.

In conclusion, the comprehensive experimental results validate the effectiveness of both max pooling and mean pooling in the proposed WILDE. This combination significantly enhances the ability of function-level vulnerability detection and statement-level vulnerability localization.

6.4.4 The influence of the training data size to the performance of WILDE

In this experiment, we assess the impact of training data size on the performance of our proposed WILDE for function-level vulnerability detection and statement-level vulnerability local-

Model	Function-level Detection				Statement-level Localization								
	Acc	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**	**MAR**	**MFR**	**IFA**	**Top-1**	**Top-3**	**Top-5**
$\text{WILDE}_{10\%}$	0.969	0.580	0.265	0.364	0.991	0.074	0.014	0.024	9.33	6.73	5.73	0.235	0.444	0.585
$\text{WILDE}_{20\%}$	0.971	0.610	0.398	0.482	0.987	0.137	0.118	0.127	**9.07**	6.48	5.48	0.227	0.431	0.583
$\text{WILDE}_{50\%}$	0.971	0.591	0.521	0.554	0.985	0.170	0.217	0.191	9.33	6.64	5.64	0.260	0.460	0.593
WILDE	**0.977**	**0.724**	**0.522**	**0.607**	**0.983**	**0.183**	**0.338**	**0.237**	9.08	**6.46**	**5.46**	**0.283**	**0.484**	**0.609**

Table 6.8: Comparison results on function-level vulnerability detection and statement-level vulnerability localization with different sizes of training data in the dataset of Fan et al.. The best results are highlighted in **bold** font.

ization. Since only the dataset of Fan et al. includes statement-level labels, we exclusively evaluate our WILDE using this dataset. To thoroughly examine the influence of training data size on model performance, we randomly select 10%, 20%, 50%, and 100% of the data from the training dataset to construct new training datasets. Table 6.8 presents the experimental results for function-level vulnerability detection and statement-level vulnerability localization across different training data sizes. As expected, the model's performance in function-level vulnerability detection consistently improves as the training data size increases. Interestingly, even though the data lacks statement-level labels, the model's performance in statement-level vulnerability localization also improves as the training data size increases. These findings demonstrate that our proposed WILDE becomes more proficient at locating vulnerability statements as the training data size grows, even without explicit annotations. However, the performance improvement of WILDE on absolute label prediction and relative scores, as introduced in Section § 6.2.5, varies with the increase in training data size. Specifically, the performance improvement in absolute label prediction, measured by accuracy, precision, recall, and F1, is substantial with larger training data sizes. Conversely, WILDE maintains most of its performance in relative scores metrics even with limited training data, and the performance improvement in relative scores is not as rapid as the improvement in absolute label prediction with increasing training data size.

In summary, increasing the size of the training data can enhance the performance of function-level vulnerability detection and statement-level vulnerability localization, even without any additional information about the vulnerable statements in the

data.

6.4.5 The detection ability of WILDE for different types of CWE vulnerabilities

Table 6.9 presents the detection results of our proposed WILDE for different types of vulnerabilities from CWE. To ensure meaningful data analysis, we have filtered out vulnerability types with a small number of occurrences and retained only those with at least five instances. The CWE has released the 2023 Top 25 Most Dangerous Software Weaknesses on their website [1]. The rank metric in the table indicates the CWE's ranking in their list. In cases where the vulnerability types in our test dataset are not included in the Top 25 list, we denote them as "N/A" in the rank metric. Additionally, the TPR metric in the table represents the true positive rate, indicating the percentage of vulnerabilities successfully detected by the model.

The results reveal an interesting finding: the detection ability of WILDE does not vary significantly between different types of CWE vulnerability, with a TPR of approximately 56% for most CWE types. However, certain vulnerabilities, such as CWE-415 and CWE-284, exhibit notably high or low TPRs. It is important to note that the limited number of vulnerabilities makes it difficult to conclusively determine whether WILDE performs well or poorly in these specific types. Nevertheless, some exceptions stand out. Notably, WILDE demonstrates a strong detection ability for vulnerabilities classified as CWE-119, successfully identifying almost all instances of this type. Conversely, WILDE displays a poor detection ability for vulnerabilities categorized as CWE-200 and CWE-399, as it only identifies less than half of the vulnerabilities within these types.

CWE-ID	**Description**	**Rank**	**TPR**	**Proportion**
CWE-787	Out-of-bounds Write	1	50.0%	7/14
CWE-416	Use After Free	4	61.5%	8/13
CWE-20	Improper Input Validation	6	56.6%	61/108
CWE-125	Out-of-bounds Read	7	54.2%	13/24
CWE-476	NULL Pointer Dereference	12	55.6%	5/9
CWE-190	Integer Overflow or Wraparound	14	77.8%	14/18
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	17	56.0%	70/125
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	21	52.4%	11/21
CWE-284	Improper Access Control	N/A	37.5%	3/8
CWE-189	Numeric Errors	N/A	52.6%	10/19
CWE-732	Incorrect Permission Assignment for Critical Resource	N/A	57.1%	4/7
CWE-254	7PK - Security Features	N/A	44.4%	4/9
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	N/A	42.9%	12/28
CWE-415	Double Free	N/A	71.4%	5/7
CWE-399	Resource Management Errors	N/A	48.7%	19/39
Total			54.8%	246/449

Table 6.9: Detection results for different CWE vulnerabilities with our proposed WILDE.

In conclusion, WILDE demonstrates comparable detection abilities for most types of vulnerabilities. Nevertheless, it exhibits superior performance in detecting vulnerabilities related to Integer Overflow or Wraparound, while its effectiveness is relatively weaker in identifying vulnerabilities associated with Exposure of Sensitive Information to an Unauthorized Actor and Resource Management Errors.

6.5 Threats to Validity

After careful analysis, we have identified several potential threats to the validity of our study.

6.5.1 Threats to External Validity

While we have chosen three commonly used datasets to assess the effectiveness of our vulnerability detection approach, it is important to note that these datasets have limited size. Consequently, the results obtained from these datasets may not accurately reflect the performance of our approach in real-world scenarios.

6.5.2 Threats to Internal Validity

In this chapter, we adopt the hyperparameters from LineVul [35] to maintain consistency. While we acknowledge the potential impact of hyperparameters on the performance of our proposed WILDE, we did not investigate their influence due to the considerable cost associated with model training. However, it is important to note that different hyperparameter settings may indeed affect WILDE's performance. Among these settings, the

maximum input token length holds particular significance. Currently, in WILDE, we have set the maximum input token length to 512, discarding any additional tokens. The performance of WILDE for longer code samples has not been thoroughly explored under this constraint.

Moreover, it is worth mentioning that certain vulnerabilities may be closely tied to the context of the code, such as use-after-free vulnerabilities. In some cases, the division of code segments could lead to the disappearance of existing vulnerabilities or even the emergence of new ones, potentially altering the label of the target function. Therefore, we need to carefully consider the implications of code segmentation on WILDE's effectiveness in identifying such vulnerabilities. Further investigations into the impact of these factors are necessary for a comprehensive understanding of WILDE's performance.

Furthermore, the vulnerability labels at the statement level are determined by checking whether the statements have been modified in the commit. Consequently, all the modified statements are considered vulnerable. However, simply altering a statement does not guarantee the presence of an actual vulnerability. This data pre-processing approach could potentially introduce biases into the vulnerable statement labels.

6.6 Summary

In this chapter, we proposed a novel approach named WILDE for vulnerability detection. WILDE incorporates the multiple instance learning framework to predict whether a given function is vulnerable or not, while also offering precise localization information about the vulnerable statements within the function.

Through experiments conducted on public datasets, we have demonstrated that WILDE achieves comparable performance to previous baselines in function-level vulnerability detection, and outperforms state-of-the-art baselines in statement-level vulnerability localization.

□ **End of chapter.**

Chapter 7

Conclusion and Future Work

7.1 Conclusion

With the large-scale application of software, how to reduce the workload of software developers has gradually become a research hotspot in software engineering. This thesis aims at intelligent reliable code retrieval. Specifically, we propose effective code retrieval approaches and code retrieval acceleration approaches to efficiently retrieve the users' desired code snippets from the large-scale code database. Then, we propose the vulnerability detection approaches to check whether our retrieved code snippets contain the vulnerability problems. Extensive experiment results in this thesis demonstrate the effectiveness of our proposed approaches. Specifically, we make the contributions as follows:

In Chapter 3, we introduce a novel neural network model named CRaDLe. CRaDLe couples both structural and semantic information of code at the statement level, where the code structures are extracted based on PDG. Extensive experiments have been conducted to verify the performance of the proposed approach. The evaluation results show that CRaDLe can signif-

icantly outperform the state-of-the-art models.

In Chapter 4, we propose a novel approach, CoSHC, for accelerating the retrieval efficiency of deep learning-based code search approaches. CoSHC first clusters the representation vectors into different categories and then generates binary hash codes for both source code and queries. Finally, CoSHC gives the normalized prediction probability of each category for the given query. Then, CoSHC will decide the number of code candidates for the given query in each category according to the probability. Comprehensive experiments have been conducted to validate the performance of the proposed approach. The evaluation results show that CoSHC can preserve more than 99% performance of most baseline models

In Chapter 5, we propose CSSDH, a deep hashing lookup table-based approach for code retrieval. CSSDH adopts an adaptive bit relaxing strategy and dynamic matching objective strategy to convert the long hash code from previous deep hashing approaches into segmented hash code. These short hash codes will be utilized to construct the lookup hash tables for code retrieval. Experimental results indicate that CSSDH can reduce at least 95% of the retrieval time of current state-of-the-art deep hashing approaches, which sort the candidates by calculating the Hamming distance. Meanwhile, CSSDH can retain the comparable performance or even outperform the previous deep hashing approaches in the recall step.

In Chapter 6, we propose a novel approach named WILDE for function-level vulnerability detection with the statement-level localization. WILDE first converts an input code snippet into a token sequence and feeds it into a Transformer-based encoder. Two channels aim to capture local and global features separately.

The results from these two channels will be combined to produce a single prediction for a single statement. The extensive experimental findings showcase that WILDE achieves comparable performance in detecting vulnerabilities at the function level compared to previous models. Furthermore, its ability to localize vulnerabilities surpasses that of the earlier models.

In summary, this thesis studies reliable code retrieval, including effective code retrieval, code retrieval acceleration, and vulnerability detection. Extensive experiments on public datasets confirm the efficiency and effectiveness of our proposed approaches.

7.2 Future Directions

Reliable code retrieval with code semantic learning has garnered significant attention in recent years, and it is a promising research topic. Although we have proposed several novel approaches that achieve state-of-the-art performance, there are still many exciting research topics that can be considered as future work.

7.2.1 Repository-Level Code Generation with Large Language Model

Automated code generation, which can automatically generate code snippets based on natural language descriptions provided by users, is one of the potential technologies that can replace code retrieval. Since this technology can significantly shorten the software development period and reduce the workload of software engineers, it has emerged as a critical research priority in the realm of automated program development.

With the emergence of large language models, the ability of deep learning for code generation has been dramatically improved. For example, DeepMind proposes a large language model named AlphaCode [65]. According to their experiment results, AlphaCode can achieve the average (54% percentile) level of human programmers in real-world programming competitions. Codex [16] is another large language model proposed by OpenAI. The famous real-time code suggestions tool Copilot is empowered by this large language model. Later, OpenAI proposed a next-generation large language model named ChatGPT. ChapGPT exhibits the unique zero-shot code generation ability.

While large language models have demonstrated impressive code generation capabilities, a gap exists in their real-world application within software development scenarios. In practical software development, engineers often need to write code specific to their projects, frequently relying on self-defined functions or classes and third-party libraries. Although large language models can accept input containing over 30,000 tokens, modern software projects often consist of hundreds, or even thousands, of files. Thus, accommodating such extensive source code as input remains a significant challenge for current large language models, and it can be one potential research topic.

7.2.2 Reliable Code Generation with Large Language Model

Although the current large language model has demonstrated unique code generation ability, code security issues have yet to receive enough attention. The generated code may contain a vulnerability problem, and hackers can utilize such vulnerabilities to implant Trojans or steal data, which causes users to

suffer huge losses. Although vulnerability detection tools can be adapted to detect whether the generated code from the large language model contains the vulnerability problem, current vulnerability detection tools are still far from perfect, and they still cannot completely solve the vulnerability issues. In addition, simply connecting vulnerability detection tools with large language models will reduce the model's efficiency. It will be better to make a large language model to generate reliable code directly.

Reinforcement Learning from Human Feedback (RLHF) is the technology that can align model output to human preferences and avoid generating harmful content. Such technology has been adopted in the training of ChatGPT. It is a promising research direction if we adopt RLHF in the code large language model training to avoid generating vulnerable code.

7.2.3 Vulnerability Detection with Static Analysis and Large Language Model

Static analysis-based approaches are widely used to detect software vulnerabilities in the industry. Since the static analysis approaches utilize the pattern defined by human experts to detect the vulnerability problem, it has better interpretability than current deep learning approaches. However, these approaches suffer the problem of a high false positive ratio, which means these approaches will wrongly report the non-vulnerable code as vulnerable. Such kind of detection error will make the programmer inspect the code without any problem, which leads to the unnecessary waste of labor and seriously weakens the usefulness of these static analysis tools.

The large language model has shown its impressive ability in

code generation and analysis. Combining static analysis technology with a large language model for vulnerability detection can be a promising research topic. By inputting the static analysis result of the given code and designing the suitable chain of thought template, we can utilize the large language model to give its judgment about whether the detection is false positive and generate the corresponding reasons for this judgment. It can help current static analysis tools to reduce the false positive ratio and increase the practicality of these tools.

□ **End of chapter.**

Chapter 8

Publications during Ph.D. Study

(i) Ensheng Shi, Yanlin Wang, **Wenchao Gu**, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. *CoCoSoDa: Effective Contrastive Learning for Code Search.* In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 2198-2210). IEEE.

(ii) WeiZhe Zhang∗, **Wenchao Gu**∗, Cuiyun Gao, and Michael R. Lyu. *A transformer-based approach for improving app review response generation.* Software: Practice and Experience, 53(2), 438-454.

(iii) **Wenchao Gu**, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Michael R. Lyu. *Accelerating Code Search with Deep Hashing and Code Classification.* In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 2534-2544).

(iv) Zi Gong, Cuiyun Gao, Yasheng Wang, **Wenchao Gu**, Yun Peng, Zenglin Xu, *Source code summarization with struc-*

tural relative position guided transformer. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 13-24). IEEE.

(v) **Wenchao Gu**, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. *CRaDLe: Deep code retrieval based on semantic dependency learning.* Neural Networks, 141, 385-394.

(vi) (*In submission*) **Wenchao Gu**, Zongyi Li, Yanlin Wang, Hongyu Zhang, Cuiyun Gao, Michael Lyu. *A Framework with Self-adaptive Model Distillation for Efficient Code Retrieval*

(vii) (*In submission*) **Wenchao Gu**, yupan Chen, Yanlin Wang, Hongyu Zhang, Cuiyun Gao, Michael R. Lyu. *Weakly Supverised Vulnerability Detection and Localization via Multiple Instance Learning*

(viii) (*In submission*) **Wenchao Gu**, Ensheng Shi, Yanlin Wang, Lun Du, Shi Han, Hongyu Zhang, Meidong Zhang, Michael R. Lyu. *Accelerating Code Search via Segmented Deep Hashing*

□ **End of chapter.**

Bibliography

[1] 2023 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 20123.

[2] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[3] S. Andrews, I. Tsochantaridis, and T. Hofmann. Support vector machines for multiple-instance learning. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*, pages 561–568. MIT Press, 2002.

[4] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[5] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In A. Ellis and T. Hagino, editors, *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 651–660. ACM, 2005.

[6] C. Bergeron, G. M. Moore, J. Zaretzki, C. M. Breneman, and K. P. Bennett. Fast bundle algorithm for multiple-instance learning. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34(6):1068–1079, 2012.

[7] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Trans. Assoc. Comput. Linguistics*, 5:135–146, 2017.

[8] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In D. R. O. Jr., R. B. Arthur, K. Hinckley, M. R. Morris, S. E. Hudson, and S. Greenberg, editors, *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, pages 1589–1598. ACM, 2009.

[9] M. M. Bronstein, A. M. Bronstein, F. Michel, and N. Paragios. Data fusion through cross-modality metric learning using similarity-sensitive hashing. In *The Twenty-Third IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2010, San Francisco, CA, USA, 13-18 June 2010*, pages 3594–3601. IEEE Computer Society, 2010.

[10] N. D. Q. Bui, Y. Yu, and L. Jiang. Self-supervised contrastive learning for code retrieval and summarization via

semantic-preserving transformations. In F. Diaz, C. Shah, T. Suel, P. Castells, R. Jones, and T. Sakai, editors, *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, pages 511–521. ACM, 2021.

[11] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[12] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra. When deep learning met code search. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 964–974. ACM, 2019.

[13] Z. Cao, M. Long, J. Wang, and P. S. Yu. Hashnet: Deep learning to hash by continuation. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 5609–5618. IEEE Computer Society, 2017.

[14] M. Carbonneau, V. Cheplygina, E. Granger, and G. Gagnon. Multiple instance learning: A survey of prob-

lem characteristics and applications. *Pattern Recognit.*, 77:329–353, 2018.

[15] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Trans. Software Eng.*, 48(9):3280–3296, 2022.

[16] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[17] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.*, 30(3):38:1–38:33, 2021.

[18] B. Chess and M. Gerschefske. Rough auditing tool for security. https://code.google.com/archive/p/rough-auditing-tool-for-security/, 2019.

[19] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. In D. Wu, M. Carpuat, X. Carreras, and E. M. Vecchi, editors, *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014*, pages 103–111. Association for Computational Linguistics, 2014.

[20] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[21] Colah. Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2020.

[22] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, and A. Bordes. Supervised learning of universal sentence representations from natural language inference data. In M. Palmer, R. Hwa, and S. Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 670–680. Association for Computational Linguistics, 2017.

[23] F. Crestani. Application of spreading activation techniques in information retrieval. *Artif. Intell. Rev.*, 11(6):453–482, 1997.

[24] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. *CoRR*, abs/1708.02368, 2017.

[25] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In J. Snoeyink and J. Boissonnat, editors, *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, pages 253–262. ACM, 2004.

[26] G. Ding, Y. Guo, and J. Zhou. Collective matrix factorization hashing for multimodal data. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 2083–2090. IEEE Computer Society, 2014.

[27] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. E. Kaiser, and B. Ray. VELVET: a novel ensemble learning approach to automatically locate vulnerable statements. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, pages 959–970. IEEE, 2022.

[28] L. Du, X. Shi, Y. Wang, E. Shi, S. Han, and D. Zhang. Is a single model enough? mucos: A multi-model ensemble learning approach for semantic code search. In G. Demartini, G. Zuccon, J. S. Culpepper, Z. Huang, and H. Tong, editors, *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, pages 2994–2998. ACM, 2021.

[29] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In K. Marzullo and M. Satyanarayanan, editors, *Pro-*

ceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001, pages 57–72. ACM, 2001.

[30] Facebook. Infer. https://fbinfer.com/., 2021.

[31] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 508–512. ACM, 2020.

[32] S. Fang, Y. Tan, T. Zhang, and Y. Liu. Self-attention networks for code search. *Inf. Softw. Technol.*, 134:106542, 2021.

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, and Y. Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[34] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[35] M. Fu and C. Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *19th IEEE/ACM International Conference on Mining Software*

Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022, pages 608–620. ACM, 2022.

[36] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 541–552. ACM, 2012.

[37] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[38] D. Goodin. An nsa-derived ransomware worm is shutting down computers worldwide (2017).

[39] W. Gu, Z. Li, C. Gao, C. Wang, H. Zhang, Z. Xu, and M. R. Lyu. Cradle: Deep code retrieval based on semantic dependency learning. *Neural Networks*, 141:385–394, 2021.

[40] W. Gu, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and M. R. Lyu. Accelerating code search with deep hashing and code classification. In S. Muresan, P. Nakov, and A. Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 2534–2544. Association for Computational Linguistics, 2022.

[41] X. Gu, H. Zhang, and S. Kim. Deep code search. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *Proceedings of the 40th International Conference*

on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pages 933–944. ACM, 2018.

[42] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin. Unixcoder: Unified cross-modal pre-training for code representation. In S. Muresan, P. Nakov, and A. Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics, 2022.

[43] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[44] R. Haldar, L. Wu, J. Xiong, and J. Hockenmaier. A multi-perspective architecture for semantic code search. In D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 8563–8568. Association for Computational Linguistics, 2020.

[45] T. A. D. Henderson and A. Podgurski. Sampling code clones from program dependence graphs with GRAPLE. In *Proceedings of the 2nd International Workshop on Soft-*

ware Analytics, SWAN@SIGSOFT FSE 2016, Seattle, WA, USA, November 13, 2016, pages 47–53, 2016.

[46] G. Heyman and T. V. Cutsem. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *CoRR*, abs/2008.12193, 2020.

[47] D. Hin, A. Kan, H. Chen, and M. A. Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 596–607. ACM, 2022.

[48] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[49] D. Hu, F. Nie, and X. Li. Deep binary reconstruction for cross-modal hashing. *IEEE Trans. Multim.*, 21(4):973–985, 2019.

[50] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proc. VLDB Endow.*, 9(1):1–12, 2015.

[51] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

[52] Israel. Checkmarx. https://checkmarx.com/., 2021.

[53] J. Jang, M. Woo, and D. Brumley. Redebug: Finding unpatched code clones in entire OS distributions. *login Usenix Mag.*, 37(6), 2012.

[54] J. Jiang, S. Wen, S. Yu, Y. Xiang, and W. Zhou. Identifying propagation sources in networks: State-of-the-art and comparative studies. *IEEE Commun. Surv. Tutorials*, 19(1):465–481, 2017.

[55] O. Khattab and M. Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over BERT. In J. Huang, Y. Chang, X. Cheng, J. Kamps, V. Murdock, J. Wen, and Y. Liu, editors, *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, pages 39–48. ACM, 2020.

[56] 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 2017, San Jose, CA, USA, May 22-26, 2017*, pages 595–614. IEEE Computer Society, 2017.

[57] Y. Kim. Convolutional neural networks for sentence classification. In A. Moschitti, B. Pang, and W. Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1746–1751. ACL, 2014.

[58] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations,*

ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[59] D. Kotzias, M. Denil, N. de Freitas, and P. Smyth. From group to individual labels using deep features. In L. Cao, C. Zhang, T. Joachims, G. I. Webb, D. D. Margineantu, and G. Williams, editors, *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 597–606. ACM, 2015.

[60] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012.

[61] C.-Y. Lee, P. W. Gallagher, and Z. Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. In *Artificial intelligence and statistics*, pages 464–472, 2016.

[62] Y. J. Lee, S.-H. Choi, C. W. Kim, S. Lim, and K.-W. Park. Learning binary code with deep learning to detect software weakness. 2017.

[63] M. Li, C. Li, S. Li, Y. Wu, B. Zhang, and Y. Wen. ACGVD: vulnerability detection based on comprehensive graph via graph neural network with attention. In D. Gao, Q. Li, X. Guan, and X. Liao, editors, *Information and*

Communications Security - 23rd International Conference, ICICS 2021, Chongqing, China, November 19-21, 2021, Proceedings, Part I, volume 12918 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2021.

[64] M. Li, Z. Ma, Y. G. Wang, and X. Zhuang. Fast haar transforms for graph neural networks. *Neural Networks*, 128:188–198, 2020.

[65] Y. Li, D. H. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022.

[66] Y. Li, S. Wang, and T. N. Nguyen. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 292–303. ACM, 2021.

[67] Y. Li, S. Wang, T. N. Nguyen, and S. V. Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.*, 3(OOPSLA):162:1–162:30, 2019.

[68] 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 Trans. Dependable Secur. Comput.*, 19(4):2244–2258, 2022.

[69] 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.

[70] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(2):318–327, 2020.

[71] Z. Lin, G. Ding, M. Hu, and J. Wang. Semantics-preserving hashing for cross-view retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 3864–3872. IEEE Computer Society, 2015.

[72] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio. A structured self-attentive sentence embedding. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[73] S. Liu, S. Qian, Y. Guan, J. Zhan, and L. Ying. Joint-modal distribution-based similarity hashing for large-scale unsupervised deep cross-modal retrieval. In J. Huang, Y. Chang, X. Cheng, J. Kamps, V. Murdock, J. Wen, and Y. Liu, editors, *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, pages 1379–1388. ACM, 2020.

[74] S. Liu, X. Xie, J. K. Siow, L. Ma, G. Meng, and Y. Liu. Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search. *IEEE Trans. Software Eng.*, 49(4):2839–2855, 2023.

[75] I. Loshchilov and F. Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[76] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan. Query expansion via wordnet for effective code search. In Y. Guéhéneuc, B. Adams, and A. Serebrenik, editors, *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 545–549. IEEE Computer Society, 2015.

[77] X. Luo, C. Chen, H. Zhong, H. Zhang, M. Deng, J. Huang, and X. Hua. A survey on deep hashing methods. *CoRR*, abs/2003.03369, 2020.

[78] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on API understanding and extended boolean model (E). In M. B. Cohen, L. Grunske, and M. Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 260–270. IEEE Computer Society, 2015.

[79] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In R. N. Taylor, H. C. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference*

on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, pages 111–120. ACM, 2011.

[80] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013*, 2013.

[81] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society, 2005.

[82] V. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 178–182. ACM/IEEE, 2022.

[83] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1–13. ACM, 2022.

[84] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang. Retrieval augmented code generation and summarization. In M. Moens, X. Huang, L. Specia, and

S. W. Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2719–2734. Association for Computational Linguistics, 2021.

[85] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging semantic signatures for bug search in binary programs. In C. N. P. Jr., A. Hahn, K. R. B. Butler, and M. Sherr, editors, *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 406–415. ACM, 2014.

[86] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the IDE into a self-confident programming prompter. In P. T. Devanbu, S. Kim, and M. Pinzger, editors, *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 102–111. ACM, 2014.

[87] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In Y. Berbers and W. Zwaenepoel, editors, *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, pages 15–27. ACM, 2006.

[88] Pypl popularity of programming language. http://pypl.github.io/PYPL.html, 2020.

[89] R. Rahmani and S. A. Goldman. MISSL: multiple-instance semi-supervised learning. In W. W. Cohen and A. W. Moore, editors, *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, volume 148 of *ACM International Conference Proceeding Series*, pages 705–712. ACM, 2006.

[90] D. A. Ramos and D. R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In J. Jung and T. Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 49–64. USENIX Association, 2015.

[91] S. E. Robertson and H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, 2009.

[92] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley. Automated vulnerability detection in source code using deep representation learning. In M. A. Wani, M. M. Kantardzic, M. S. Mouchaweh, J. Gama, and E. Lughofer, editors, *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*, pages 757–762. IEEE, 2018.

[93] C. Sabottke, O. Suciu, and T. Dumitras. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In J. Jung and T. Holz, editors, *24th USENIX Security Symposium, USENIX Se-*

curity 15, Washington, D.C., USA, August 12-14, 2015, pages 1041–1056. USENIX Association, 2015.

[94] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra. Retrieval on source code: a neural code search. In J. Gottschlich and A. Cheung, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 31–41. ACM, 2018.

[95] R. Salakhutdinov and G. E. Hinton. Semantic hashing. *Int. J. Approx. Reason.*, 50(7):969–978, 2009.

[96] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[97] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos. Towards security defect prediction with AI. *CoRR*, abs/1808.09897, 2018.

[98] D. C. Shepherd, Z. P. Fry, E. Hill, L. L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In B. M. Barry and O. de Moor, editors, *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*, volume 208 of *ACM Interna-*

tional Conference Proceeding Series, pages 212–224. ACM, 2007.

[99] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun. Cocosoda: Effective contrastive learning for code search. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2198–2210. IEEE, 2023.

[100] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei. Improving code search with co-attentive representation learning. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 196–207. ACM, 2020.

[101] S. Su, Z. Zhong, and C. Zhang. Deep joint-semantics reconstructing hashing for large-scale unsupervised cross-modal retrieval. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 3027–3035. IEEE, 2019.

[102] Y. Sui and J. Xue. SVF: interprocedural static value-flow analysis in LLVM. In A. Zaks and M. V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 265–266. ACM, 2016.

[103] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. London, U.K.:Pearson Education, 2007.

[104] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 1556–1566. The Association for Computer Linguistics, 2015.

[105] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[106] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[107] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May*

3, 2018, Conference Track Proceedings. OpenReview.net, 2018.

[108] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference (ACSAC 2000), 11-15 December 2000, New Orleans, Louisiana, USA*, page 257. IEEE Computer Society, 2000.

[109] D. Votipka, R. Stevens, E. M. Redmiles, J. Hu, and M. L. Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 374–391. IEEE Computer Society, 2018.

[110] 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. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 13–25. IEEE, 2019.

[111] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Trans. Inf. Forensics Secur.*, 16:1943–1958, 2021.

[112] J. Wang, W. Liu, S. Kumar, and S. Chang. Learning to hash for indexing big data - A survey. *Proc. IEEE*, 104(1):34–57, 2016.

[113] S. I. Wang and C. D. Manning. Baselines and bigrams: Simple, good sentiment and topic classification. In *The 50th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, July 8-14, 2012, Jeju Island, Korea - Volume 2: Short Papers*, pages 90–94. The Association for Computer Linguistics, 2012.

[114] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*, 2021.

[115] Y. Wang, M. Li, Z. Ma, G. Montúfar, X. Zhuang, and Y. Fan. Haar graph pooling. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 9952–9962. PMLR, 2020.

[116] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In M. Moens, X. Huang, L. Specia, and S. W. Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021.

[117] X. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao. Vulnerability detection with graph simplifica-

tion and enhanced graph representation learning. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2275–2286. IEEE, 2023.

[118] D. A. Wheeler. Flawfinder. https://dwheeler.com/flawfinder/, title = Flawfinder., 2021.

[119] G. Wu, Z. Lin, J. Han, L. Liu, G. Ding, B. Zhang, and J. Shen. Unsupervised deep hashing via binary latent factor models for large-scale cross-modal retrieval. In J. Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2854–2860. ijcai.org, 2018.

[120] T. Wu, S. Wen, Y. Xiang, and W. Zhou. Twitter spam detection: Survey of new approaches and comparative study. *Comput. Secur.*, 76:265–284, 2018.

[121] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin. Vulcnn: An image-inspired scalable vulnerability detection system. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2365–2376. ACM, 2022.

[122] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014*, pages 590–604. IEEE Computer Society, 2014.

[123] C. Yan, B. Gong, Y. Wei, and Y. Gao. Deep multi-view enhancement hashing for image retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 43(4):1445–1451, 2021.

[124] D. Yang, D. Wu, W. Zhang, H. Zhang, B. Li, and W. Wang. Deep semantic-alignment hashing for unsupervised cross-modal retrieval. In C. Gurrin, B. Þ. Jónsson, N. Kando, K. Schöffmann, Y. P. Chen, and N. E. O'Connor, editors, *Proceedings of the 2020 on International Conference on Multimedia Retrieval, ICMR 2020, Dublin, Ireland, June 8-11, 2020*, pages 44–52. ACM, 2020.

[125] Z. Yao, J. R. Peddamail, and H. Sun. Coacor: Code annotation for code retrieval with reinforcement learning. In L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 2203–2214. ACM, 2019.

[126] Z. Ye and Y. Peng. Multi-scale correlation for sequential cross-modal hashing learning. In S. Boll, K. M. Lee, J. Luo, W. Zhu, H. Byun, C. W. Chen, R. Lienhart, and T. Mei, editors, *2018 ACM Multimedia Conference on Multimedia Conference, MM 2018, Seoul, Republic of Korea, October 22-26, 2018*, pages 852–860. ACM, 2018.

[127] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. In R. Barzilay and M. Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 440–450. Association for Computational Linguistics, 2017.

[128] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. Gnnexplainer: Generating explanations for graph neural

networks. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 9240–9251, 2019.

[129] C. Zeng, Y. Yu, S. Li, X. Xia, Z. Wang, M. Geng, L. Bai, W. Dong, and X. Liao. degraphcs: Embedding variable-based flow graph for neural code search. *ACM Trans. Softw. Eng. Methodol.*, 32(2):34:1–34:27, 2023.

[130] D. Zhang and W. Li. Large-scale supervised multimodal hashing with semantic correlation maximization. In C. E. Brodley and P. Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2177–2183. AAAI Press, 2014.

[131] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In J. M. Atlee, T. Bultan, and J. Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 783–794. IEEE / ACM, 2019.

[132] P. Zhang, Y. Luo, Z. Huang, X. Xu, and J. Song. High-order nonlocal hashing for unsupervised cross-modal retrieval. *World Wide Web*, 24(2):563–583, 2021.

[133] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning compre-

hensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, pages 10197–10207, 2019.

[134] H. Zhu, M. Long, J. Wang, and Y. Cao. Deep hashing network for efficient similarity retrieval. In D. Schuurmans and M. P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 2415–2421. AAAI Press, 2016.