Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

LEARNING REPRESENTATIONS FOR EFFECTIVE AND EXPLAINABLE SOFTWARE BUG DETECTION AND FIXING

by
Yi Li

Software has an integral role in modern life; hence software bugs, which undermine software quality and reliability, have substantial societal and economic implications. The advent of machine learning and deep learning in software engineering has led to major advances in bug detection and fixing approaches, yet they fall short of desired precision and recall. This shortfall arises from the absence of a 'bridge,' known as learning code representations, that can transform information from source code into a suitable representation for effective processing via machine and deep learning.

This dissertation builds such a bridge. Specifically, it presents solutions for effectively learning code representations using four distinct methods—context-based, testing results-based, tree-based, and graph-based—thus improving bug detection and fixing approaches, as well as providing developers insight into the foundational reasoning. The experimental results demonstrate that using learning code representations can significantly enhance explainable bug detection and fixing, showcasing the practicability and meaningfulness of the approaches formulated in this dissertation toward improving software quality and reliability.

LEARNING REPRESENTATIONS FOR EFFECTIVE AND EXPLAINABLE SOFTWARE BUG DETECTION AND FIXING

by
Yi Li

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology,
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Information Systems

Department of Informatics

August 2023

APPROVAL PAGE

LEARNING REPRESENTATIONS FOR EFFECTIVE AND EXPLAINABLE SOFTWARE BUG DETECTION AND FIXING

Yi Li

Michael J. Lee, Dissertation Co-advisor — Date
Associate Professor of Informatics, NJIT

Iulian Neamtiu, Dissertation Co-advisor — Date
Professor of Computer Science, NJIT

Tomer Weiss, Committee Member — Date
Assistant Professor of Informatics, NJIT

Hai Phan, Committee Member — Date
Assistant Professor of Data Science, NJIT

Tien N. Nguyen, Committee Member — Date
Professor of Computer Science, The University of Texas at Dallas,
Richardson, Texas

BIOGRAPHICAL SKETCH

Author: Yi Li

Degree: Doctor of Philosophy

Date: August 2023

Undergraduate and Graduate Education:

- Doctor of Philosophy in Information Systems,
  New Jersey Institute of Technology, Newark, NJ, 2023
- Master of Science in Information Systems,
  Stevens Institute of Technology, Hoboken, NJ, 2016
- Bachelor of Science in Applied Mathematics,
  Renmin University of China, Beijing, China, 2013

Major: Information Systems

Presentations and Publications:

Wenbo Wang, Tien N. Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally, "DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection", In *Proceedings of the 45th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 2249-2261, 2023.

Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang, "Does Data Sampling Improve Deep Learning-Based Vulnerability Detection? Yeas! and Nays!", In *Proceedings of the 45th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 2287-2298, 2023.

Yi Li, Shaohua Wang, and Tien N. Nguyen, "DEAR: A Novel Deep Learning-Based Approach for Automated Program Repair", In *Proceedings of the 44nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 511-523, 2022.

Yi Li, Shaohua Wang, and Tien N. Nguyen, "Fault Localization to Detect Co-Change Fixing Locations", In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 659-671, 2022.

Yi Li, Shaohua Wang, and Tien N. Nguyen, "UTANGO: Untangling Commits with Context-Aware, Graph-Based, Code Change Clustering Learning Model", In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 221-232, 2022.

Yi Li, Shaohua Wang, Wenbo Wang, Tien N. Nguyen, Yan Wang, and Xinyue Ye "RAP4DQ:Learning to Recommend Relevant API Documentation for Developer Questions", *Empirical Software Engineering (EMSE)*, 27(1), pages 23, 2022.

Yi Li, Shaohua Wang, and Tien N. Nguyen, "A Context-Based Automated Approach for Method Name Consistency Checking and Suggestion", In *Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 574-586, 2021.

Yi Li, Shaohua Wang, and Tien N. Nguyen, 'Fault Localization with Code Coverage Representation Learning", In *Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 661-673, 2021.

Yi Li, Shaohua Wang, and Tien N. Nguyen, "Vulnerability Detection with Fine-Grained Interpretations", In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 292-303, 2021.

Wenbo Wang, Yi Li, Shaohua Wang, and Xinyue Ye, "QA4GIS: A Novel Approach Learning to Answer GIS Developer Questions with API Documentation", *Transactions in GIS*, 25(5), pages 2675-2700, 2021.

Yi Li, Shaohua Wang, and Tien N. Nguyen, "DLFix: Context-Based Code Transformation Learning for Automated Program Repair", In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 602-614, 2020.

Yi Li, Shaohua Wang, and Tien N. Nguyen, "An Empirical Study on the Characteristics of Question-Answering Process on Developer Forums", In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering: Companion Proceedings (ICSE), Poster*, pages 318-319, 2020.

Yi Li, Shaohua Wang, and Tien N. Nguyen, "Improving Automated Program Repair using Two-Layer Tree-Based Neural Networks", In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering: Companion Proceedings (ICSE), Poster*, pages 316-317, 2020.

Yi Li, "Improving Bug Detection and Fixing via Code Representation Learning", In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering: Companion Proceedings (ICSE), Student Competition*, pages 137-139, 2020.

Jiahao Fan, Yi Li, Shaohua Wang and Tien N. Nguyen, "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries", In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR), Data Track*, pages 508-512, 2020.

Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen, "Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks", In *Proceedings of the 2019 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1-30, 2019.

Son Van Nguyen, Tien N. Nguyen, Yi Li, and Shaohua Wang, "Combining Program Analysis and Statistical Language Model for Code Statement Completion", In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering Conference (ASE)*, pages 710-721, 2019.

I dedicate this to my wife, Yujiao Lei, who always supported me, even on the tough days.

ACKNOWLEDGMENT

Words cannot express my gratitude to my co-advisor and chair of my dissertation defense committee, Dr. Michael J. Lee, for his patience and help. The same thanks I should give to my other co-advisor, Dr. Iulian Neamtiu, for his professional guidance and advice. I also could not go on such a long trip on research without my previous advisor, Dr. Shaohua Wang, who generously provided suggestions and knowledge for my research.

Additionally, special thanks should be given to my dissertation defense committee members, Dr. Tomer Weiss, Dr. Hai Phan, and Dr. Tien N. Nguyen, who kindly offered knowledge and expertise.

I am very thankful to the Department of Informatics for giving me the opportunity to teach. Also, the research funding and foundation provided by Dr. Shaohua Wang, along with the department's start-up funding for a few semesters, was crucial to the success of my study and my research.

I am also grateful to Dr. Tien N. Nguyen, who collaborated with me on research by offering his professional vision and knowledge.

Thanks should also go to the labmates and research participants, including Wenbo Wang, Jiaxing Zhang, Jiahao Fan, Jiyuan Zhang, Son Van Nguyen, Aashish Yadavally, and Xu Yang, for their assistance and support on the research.

Last but not least, I would like to thank my family, especially my spouse, Yujiao Lei, and my parents, Yan Sun and Hansheng Li. Without their belief, I would not have had the motivation and courage to face all the challenges during this process.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

Software quality and reliability are essential to software development. Software quality reflects how well the software complies with or conforms to a given design, while software reliability measures the level of risk and the likelihood of potential application failures. Software bugs, which can arise during the development process and potentially lead to serious problems, represent a significant threat to software quality and reliability, capable of causing severe consequences and undermining the overall robustness of the system. For example, several airports and/or airlines in the last decade have been disrupted by software bugs affecting departure boards and the check-in system. In these instances, thousands of passengers experience wasted time, while airlines incur significant losses in terms of both business and finances. Software bugs can lead to severe economic loss and, more distressingly, endanger lives. For instance, Nissan had to recall over a million cars due to a software bug in their airbag sensors. This bug interfered with the car's ability to recognize an adult occupant in the passenger seat, consequently disabling the airbag's proper function, and has been implicated in two reported accidents. Because of the serious influence of the software bugs we mentioned above, fixing them as early as possible is desirable.

In software engineering, bugs, faults, and vulnerabilities are often mentioned in different research topics for different software development stages. The fault is an incorrect step, process, or data definition found in the testing stage by running the test cases. The vulnerability is a weakness that can be exploited by a threat actor, such as an attacker, to cross privilege boundaries within the software. Assuming a broad definition of a bug as any error or flaw within software engineering, faults and

vulnerabilities can also be classified as bugs. Hence, the term "bug" covers all such issues in this dissertation.

In the software quality and reliability relevant research area, there are many existing research solving software bugs-related problems [112, 67, 24, 21, 46] or improving the existing techniques to help avoid software bugs to cause serious issues [80, 82, 166, 115, 165]. Within these research studies, a broad scope encompasses various software engineering research fields, including bug detection, fault localization, program automated repair, and more. Given the immense scale of these research topics, conducting studies necessitates highly effective and high-performing approaches. Machine learning (ML) and deep learning (DL) are excellent choices for researchers in these fields due to their capacity for handling large datasets, identifying intricate patterns, learning from previous computations, and delivering more precise results over time, thereby providing a robust framework to tackle complex software issues effectively.

The swift evolution of ML and DL in software engineering has led to a surge of research [137, 109, 140, 21, 24] leveraging these technologies in recent years. Among these, data-driven studies that depend on ML and DL have attracted substantial attention from academic and industry circles. However, a consistent linkage between software engineering and ML/DL is not established during their development, primarily because information carriers intrinsic to software engineering, such as source code, are not directly applicable to ML and DL approaches. Bridging this divide, learning representations play an integral role in converting software engineering data into formats that ML and DL models can efficiently process. Therefore, it is virtually impossible to overlook the relevance of learning representations in establishing ML or DL-based methodologies within software engineering research.

Learning representations primarily consist of two components: representations and learning models. Representations are methods of embodying software engineering

data in various data structures, such as trees, graphs, or identifiers to represent source code, code commits, etc. Learning models, in contrast, are tools that learn vector representations from different data structures. Among these learning representations, the learning code representations, a term we use to describe source code learning representation, is a vital aspect of software engineering due to the intimate link between source code and software engineering issues. In simple terms, learning code representations model the code by converting the information within the source code into vectors, facilitating their use by ML and DL methods. This dissertation focuses primarily on the application and significance of learning code representations.

Despite many researchers recognizing the importance of learning code representations and developing numerous code representation methods, choosing the right approach for varying tasks remains challenging due to the vast array of options for generating representation vectors. Upon reviewing existing literature in software bug-related research areas, such as bug detection, fault localization, and automated program repair, we noticed a significant gap: many studies lack comprehensive research on learning code representations and the knowledge to choose the right approach for different software engineering tasks. For instance, some approaches treat code as the natural language for processing, thereby overlooking the inherent structure and relationships within the source code—a scenario we describe as lacking code representations. Others may choose inappropriate learning models, such as using a sequence-based learning model to generate representation vectors for tree or graph structures.

To address these issues, we first need to discern the crucial information required for a specific task. Once this information is identified, choosing the code representation becomes more straightforward. For example, in bug detection, the code's structure and context are important for identifying bugs. In such a case, Abstract Syntax Trees (AST), encompassing both code context and structure, could

be an excellent choice for code representation. When selecting the learning model, both the downstream tasks and code representation are needed to consider. As for the same example in bug detection, if we select AST as code representation, a tree-based learning model is necessary. Furthermore, considering that the output of bug detection is a label indicating whether the code is buggy, a tree-based learning model that produces a single vector output to represent the code may be a better choice in this scenario.

Given the challenges previously mentioned and our proposed strategy to address them, we have come up with the following dissertation scope, problem description, and dissertation objectives.

1.1 Dissertation Scope

This dissertation focuses on three main directions: bug detection, automated program repair, and model agnostic explanation. Within the bug detection, we will introduce how the learning code presentations help improve bug detection and fault localization. As for automated program repair, this dissertation discusses the specific usage of learning code representations in fixing single-statement and multi-hunk/multi-statement bugs. And in the model agnostic explanation, this dissertation focuses on applying learning code representations to generate explainable results for vulnerability detection tasks. All detailed applications mentioned in this dissertation are bug detection and fixing related software engineering tasks, and all our approaches introduced are DL-based approaches.

1.2 Problem Description

As software bugs have such a huge influence on software quality and reliability, in this dissertation, we mainly want to focus on the problem:

What are the suitable ways of learning code representation to improve the existing approaches for different software engineering tasks?

Specifically, the problems we want to focus on in this dissertation are as follows:

- Many existing bug detection approaches can detect bugs from source code. But there are still many kinds of bugs that are hard to detect. How to use different ways of learning code representation to improve the bug detection tasks could be one of the problems we want to deal with. (Section 3.2)

- With so much information in the testing process, knowing how to combine different information for source code, test case, test case running results, and error message information to do the fault localization is important for fault localization. (Section 3.3)

- Faults found during the testing process are sometimes caused by more than one statement in more than one method. Using the source code representation to catch the relationships between these statements to help locate these statements at the same is an interesting challenge. (Section 3.4)

- Automatically fixing the bugs in the real-world project development process could always be an interesting idea. But improving the speed and accuracy of the automated program repair tools is difficult. Using deep learning with suitable code representation is a way worth trying. (Section 4.2)

- Fixing the simple bugs in a single statement is not enough for automated program repair tools. Determining how to find the correct position to fix and how to select the buggy statements to fix is the key problem we need to solve. (Section 4.3)

- Vulnerability detection helps reduce the vulnerability in the software. But all existing models can only let developers know if the code is vulnerable without any explanation to help developers understand why it is vulnerable. Providing an extra explanation for these models, especially the graph-based model here, is meaningful. (Section 5.2)

1.3 Dissertation Objectives

With the problems mentioned above, in this dissertation, we have the following objectives:

- Select a suitable way of generating code representation and deep learning models to deal with the bug detection task and improve the existing approaches. The code representation should cover code content information, code structure information, and code relationships. (Section 3.2)

- Devise an approach for fault localization tasks. The learning code representation in this approach should be suitable and helpful for combining different features from source code and test case running results. (Section 3.3)

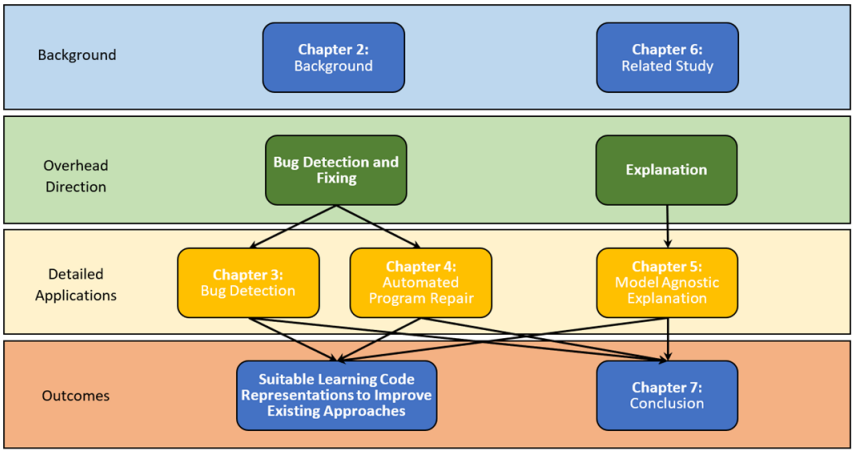


Figure 1.1 Overview of the scope and organization of this dissertation.

- Create a fault localization framework for the faults that involve more than one statement in more than one method. The code representation should cover the relationships between different statements and different methods. (Section 3.4)

- Identify required code representation structure and deep learning model to fix single statement bugs automatically. During the fixing, the code representation should contain the code content changes and code structure changes. (Section 4.2)

- Build a fault locating and buggy hunk selection model to improve the automated problem repair technique on single statement bugs. The improved automated problem repair model should be able to fix multi hunks multi statements bugs. (Section 4.3)

- Generate explanation for graph-based vulnerability detection model. The explanation should contain part of the source code to determine which part of the code is important and also need to have some natural language description to help explain the vulnerability detection results. (Section 5.2)

1.4 Dissertation Organization

The dissertation is organized as follows: Chapter 2 introduces some background concepts for the whole dissertation. From Chapter 3, the dissertation starts to introduce the details for four detailed software engineering tasks within each chapter. Chapter 3 begins by analyzing the limitation of current bug detection approaches and explaining the main ideas on using what way of learning code representation to improve bug detection. This chapter also includes the other interesting task,

fault localization, which is useful for pinpointing faults when developers are already aware of potential issues within the software that need to be addressed in subsequent processes. In this chapter, this dissertation introduces the current fault localization approaches and how we would like to improve the existing ML and DL-based approaches by using different ways of learning code representations on this task. Also, a separate challenging idea about locating the faults that influence more than one statement in more than one method is described in detail. Chapter 4 follows Chapter 3, talking about how we can use ML and DL-based approaches with suitable ways of learning code representation to automatically fix the bugs that have been successfully found and located by tasks in the last chapter. Chapter 5 introduces how to generate explanations for graph-based models on vulnerability detection problems. Chapter 6 lists related work for each task in Chapters 3-5. Chapter 7 summarizes the dissertation, discusses the research studies' contributions, and discusses future work.

CHAPTER 2

BACKGROUND

As we mentioned in the introduction chapter, we mainly focus on applying different code representation learning to bug detection, fault localization, automated program repair, and the explanation for graph-based vulnerability detection and bug detection. In this Chapter, we first define some basic concepts.

Learning Code Representation. Learning code representation is learning the representation vectors for the code. The common ways of learning code representation include tree-based, graph-based, set-based, and sequential-based. The tree-based learning code representation transfers the code into a tree structure, often the AST, and then uses a tree-based model like tree-LSTM to generate representation vectors. The graph-based learning code representation transfers the code into graphs, including program dependency graph, control flow graph, data flow graph, call graph, etc. Then it uses a graph-based learning model like GCN or GNN to generate the representation vectors. As for the set-based and sequential-based learning code representation, they often directly collect keywords like identifiers to build a set or directly regard the tokens in source code as a sequence of words. And then apply a token-based or sequential-based learning model to generate representation vectors. Various strategies for choosing suitable ways of learning code representation exist within different tasks and situations.

Bug Detection. Bug detection is the process of learning the information from the source code to detect possible bugs in the software. Bug detection can be used in most parts of the software development process, which is used to help developers keep the software quality and reliability. Existing bug detection studies often contain three kinds, including rule-based bug detection [48], mining-based bug detection [13, 55,

117, 34, 29, 155, 80], and machine learning-based bug detection [166, 165, 164, 125]. All of these three kinds of approaches have their limitation. For example, rule-based studies require manually defined new rules for the new bugs, while mining-based studies cannot distinguish the cases of incorrect code versus infrequent/rare code. Machine learning-based studies can overcome these limitations but often still have a high false-positive rate based on using unsuitable code representation learning, which often includes the sequence of tokens, identifiers, and so on. To overcome the limitation of these, we built our approach based on the data collected from git commits in GitHub and the bug reports collected from the Apache forum. We use both the tree-based learning code representation and the graph-based learning code representation to solve this problem. We will introduce our study in detail in Chapter 3.

Fault Localization. Fault localization is the task of working in the testing process to locate the faults in the software. The fault localization is mainly based on the test cases used in the testing process. Existing fault localization techniques include three main types, spectrum-based approaches[58, 57, 1], mutation-based approaches[103, 118, 119] and machine learning/deep learning-based approaches[72]. Spectrum-Based and mutation-based approaches often have the limitation of covering limited information in some fields, while the existing machine learning/deep learning-based approaches can combine them. However, within existing machine learning/deep learning-based approaches, choosing the suitable code representation learning is still important. In this dissertation, we use the code coverage matrix and sequential tokens to learn the code representation to support our two new ideas on fault localization introduced in Chapter 3.

Automated Program Repair. Automated program repair is the flow to automatically fix the bugs detected and located by the previous three tasks in the process mentioned in the above three tasks. Running the automated program repair

in testing is slightly different from other processes. During the testing, automated program repair can use test cases to help validate the fixing results, while in other processes, it requires human manual verifying. The existing automated program repair approaches contain six types. Each of them is based on mining and learning fixing patterns[112, 67, 86, 60], fixing templates [67, 112, 86, 87], mining code changes[167, 52, 64], machine learning[92, 90], information retrieval [134],and deep learning [41, 168, 169, 169, 46, 158, 24]. These six types can be grouped into two main concepts: pattern-based and machine learning/deep learning-based approaches. The pattern-based approaches often have high accuracy and high time costs for generating the fixing. Machine learning/deep learning-based approaches can do the fixing quickly, but because the code fixings are different. It is hard for the model to learn perfectly and automate fixes correctly. By analyzing the existing machine learning/deep learning-based approaches, we would like to build our deep learning-based automated program repair approach with tree-based learning code representation. The two new ideas in this task are introduced in Chapter 4.

Vulnerability Detection. Vulnerability detection is the process of detecting the vulnerability from the source code. This process has often been used in the deployment and maintenance process. Existing vulnerability can be grouped into two big groups, including rule/pattern-based vulnerability detection [38, 127, 161, 23, 49, 30] and machine learning-based vulnerability detection [137, 109, 140]. But both of these kinds do not explain why it is vulnerable, which will cost developers more time to find and understand the vulnerabilities. To generate interpretable vulnerability detection, we proposed our ideas based on an existing collected vulnerability dataset built by collecting from the vulnerability reports on the CWE website. In this task, we use the graph-based learning code representation with the program dependency graph. The idea, model design, and results can be seen in Chapter 5.

Interpretable AI. Interpretable AI explains why deep learning-based models generate certain results. The reason for generating this kind of explanation is that most of the deep learning-based models are more like black boxes. Therefore, it is hard for developers to understand the reason for getting the results from the model. Existing interpretable AI approaches are mainly in three kinds. The first type is creating simple proxy models [66, 131] or finding key features through feature gradients [187, 150]. The second type is the post-hoc interpretability method [37, 4]. And the third type is analyzing the information within the model and generating the interpretation [184]. The first two types cannot generate an explanation for graphs, while the existing approaches in the third type that can deal with graphs are still not solid and helpful enough. We proposed our ideas to build interpretable AI models based on existing bug detection and fixing related task models. We use the graph-based learning code representation to generate the explanation. The idea, model design, and results can be seen in Chapter 5.

CHAPTER 3

BUG DETECTION

3.1 Introduction

In this chapter, we introduce our work on three research topics related to bug detection. We separate them into two directions: general bug detection, which contains the first research topic, and fault localization, which includes the second and third research topics. (1) The first direction focuses on learning code representation with the deep learning model to improve the existing bug detection approaches. (2) The second direction learns code representation to enhance the existing fault localization approaches. The difference between these two directions is that the first one is for the general bug detection approaches that only depend on the source code information. In contrast, the second focuses on locating the bugs in the testing process. The second direction is not only depending on the source code but also using the test cases running results as key features to help locate the bug. Although the second one is still doing bug detection-related tasks, the different inputs separate them. Within the second direction, the difference between the second and third research topics is that the second topic mainly focuses on single-statement fault localization. In contrast, the third focuses on simultaneously locating the faulty statements in different methods for the same test case running error. All of these three research topics are published as conference papers. The roadmap of this chapter, along with the specific code representations utilized for each research topic, is depicted in Figure 3.1. We will separately introduce these three research topics in the following sections.

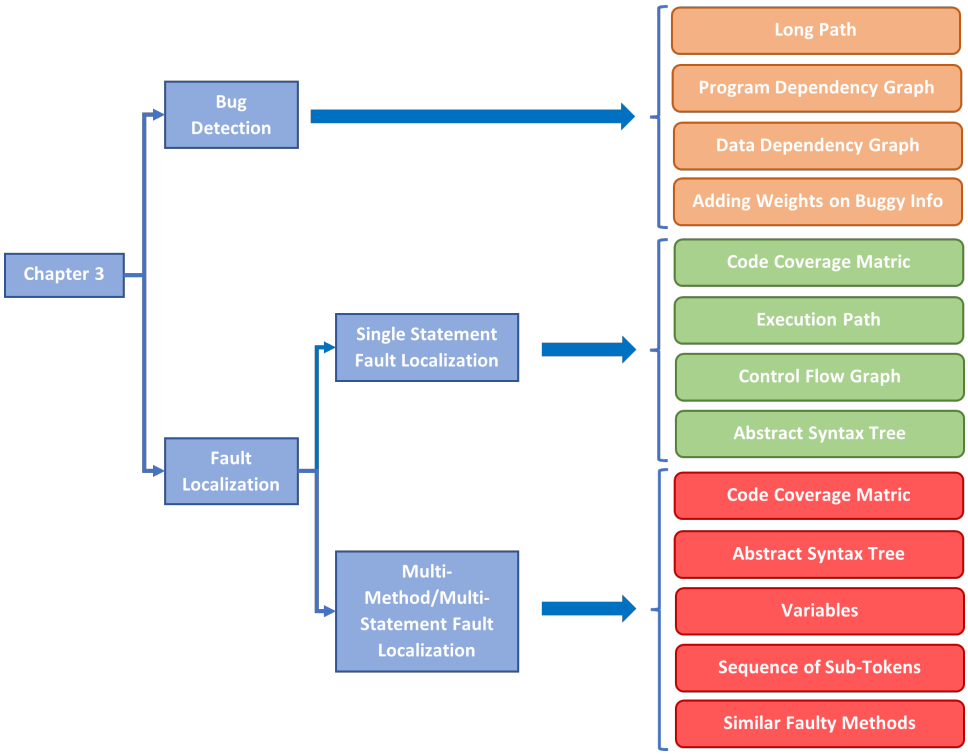


Figure 3.1 Bug Detection: Roadmap for Chapter 3.

3.2 Bug Detection with Context-Based Code Representation Learning

3.2.1 Introduction

Bug detection plays an important role in software engineering, it helps developers to detect bugs early by scanning the source code statically and determine if a given source code is buggy [80, 82, 166, 115, 165, 164, 125]. Bug detection has been shown to be effective in improving software quality and reliability [13, 55, 117, 34, 29, 155]. The existing state-of-the-art bug detection approaches can be classified into the following:

- *Rule-Based bug detection.* In this type of approaches, several programming rules are pre-defined to statically detect common programming flaws or defects. A popular example of this type of approaches is FindBugs [48]. While this type of approaches is very effective, new rules are needed to define to detect new types of bugs.

- *Mining-Based bug detection [13, 55, 117, 34, 29, 155, 80].* To overcome the pre-defined rules, the mining-based approaches rely on mining from existing source code. Typically, this type of approaches automatically extracts implicitly programming rules from program source code using data mining approaches

(e.g., mining frequent itemsets or sub-graphs) and detects violations of the extracted rules as potential bugs. These mining-based approaches still have a key limitation in a very high false positive rate due to the fact that it cannot distinguish the cases of incorrect code versus infrequent/rare code.

- *Machine learning-based bug detection [166, 165, 164, 125].* With the advances of machine learning (ML) and especially deep learning models, several approaches have been proposed to learn from previously known and reported bugs and fixes to detect bugs in the new code. While the ML-based bug detection models [107] rely on feature selections, the deep learning-based ones [125, 165] take advantages of the capability to learn the important features from the training data for bug detection. Showing the advantages over the traditional ML-based bug detection models, the deep learning-based approaches are still limited to detect bugs in individual methods without investigating the dependencies among different yet relevant methods. In practice, there exist several cases that bugs occur across more than one method. That is, to decide whether a given method is buggy or not, a model needs to consider other methods that have data and/or control dependencies with the method under investigation. Due to that, the existing deep learning-based approaches have high false positive rates, making them less practical in the daily use of software developers. For example, DeepBugs [125] is reported to have a high false positive rate of 32%. That is, approximately one out of 3 reported bugs is not a true bug, thus, wasting much developers' efforts. Our study (Sub-Section 3.2.7.3.1) also showed a false positive rate of 41% for DeepBugs on our dataset.

To overcome the aforementioned limitations of the state-of-the-art approaches while still taking advantage of deep learning capability, we propose a combination approach with the use of contexts and attention neural networks. In order to detect whether given methods are involved in bugs that might involve individual or multiple methods, we propose to use the Program Dependence Graph (PDG) [36] and Data Flow Graph (DFG) [185] as the *global context* to connect the method under investigation with other relevant methods that might contribute to identifying the buggy code. The global context is complemented by the local context extracted from the path on the AST built from the method's body. The use of PDG and DFG enables our model to reduce the false positive rate when matching the given code against the buggy code in the past because two source code fragments are similar not only if their ASTs are similar but also if the global contexts in the PDG and DFG are similar. With

this strategy, our model would increase its precision in detecting the buggy methods. However, to complement for the potential reduction in recall (i.e., our model might miss buggy code due to its stricter conditions on code similarity when additionally using the PDG/DFG), we make use of the attention neural network mechanism to put more weights onto the buggy paths in source code. That is, the paths that are similar to the buggy paths will be ranked higher, thus, improving recall.

Our approach works in three phases. In the first phase of building the representation for local context for buggy and non-buggy code, our model constructs the AST for a given method's body and extracts the paths along the AST's nodes to capture the syntactic structure of the source code. Prior works [7, 111] have shown that syntactic structure of source code can be approximately captured via the paths along their nodes with certain lengths. Word2Vec [99] is used on the AST nodes along the collected paths to convert them into vectors to capture the surrounding nodes in the paths. After using Word2Vec, the generated AST node vectors are fed into an attention-based Gated Recurrent Unit (GRU) layer [25] that allows our model to encode and emphasize on the order of the nodes in a path, i.e., on the nested structures in the AST. Also, we convert the node vectors into matrices and feed them to an attention Convolutional layer [183] that allows our model to emphasize the local coherence patterns in matrices and put more weights on the buggy paths. After that, we use Multi-Head Attention [160] to combine the results from the attention GRU layer and attention Convolutional layer together as the path *local context* representation modeling the content of a path.

In the second phase of integrating the *global context* modeling relations among paths from methods , we build the PDG and DFG, and extract the subgraphs relevant to a method. Unlike the learning of *local context* representations for paths within an AST built from a method's body using GRU and Convolutional layers, our model uses Node2Vec [39] to encode the PDG and DFG into embedded vectors to capture

relations between relevant paths. Node2Vec is a widely used network embedding algorithm to convert large graphs (e.g., a PDG of a project) into low-dimensional vectors without too much graph structural information loss for efficient processing. After having both local context and global context representations for each path, we can get the representation for each method by directly linking all merged path vectors. In the last phase, we use a convolutional layer to classify the vectors into two classes of buggy and non-buggy code. Based on the results vectors, we use SoftMax to process and set up a threshold to pick the number of potential buggy methods to report for the source code under investigation.

We have conducted several experiments to evaluate our approach on a very large dataset with +4.973M methods in 92 different versions of 8 large, open-source projects. We compare our approach with the state-of-the-art approaches in two aspects. First, we compare our tool against the existing bug detection tools using *rule-based techniques* including FindBugs [9], *mining techniques* including Bugram [164] and NAR-miner [13], and *deep learning techniques* including DeepBugs [125]. The results show that our tool can have a relative improvement up to 160% on F-score when comparing with other baselines in the unseen project setting and a relative improvement up to 92% on F-score in the unseen version setting. Our tool can detect 48 true bugs in the list of top 100 reported bugs, which is 24 more true bugs when comparing with the baselines. Second, we compare our representation with local and global contexts against the state-of-the-art code representations that are used for deep learning models in code similarity including DeepSim [195], code2vec [7], Code Vectors [47], Deep Learning Similarity [157], and Tree LSTM [151]. We used those representations with our attention-based bug detection model and compared with the results from our tool with our representation. We reported that our representation can improve over the other representations up to 206% in F-score in the unseen project setting and up to 104% on F-score in the unseen version setting. Our tool can detect

48 true bugs in top 100 reported ones, which is 27 more true bugs when comparing with the baselines. Furthermore, we conducted experiments to study the impact of the components in our model on its accuracy. We found that while global context in the PDG and DFG improves much in Precision, Multi-Head Attention mechanism helps improve much in Recall to make up for the reduction in Recall caused by the stricter condition in the PDG and DFG as the global context.

In this research topic, we make the following contributions:

- **A new code representation specialized for bug detection.** To the best of our knowledge, our work is the first to learn code representation specializing toward bug detection in three novel manners: 1) directly adding weights for differentiating buggy and non-buggy paths into code representation learning; 2) combining the local context within an AST and global context (relations among paths in the PDG and DFG) for bug detection; and 3) integrating inter-procedural information using the PDG and DFG.
- **A novel bug detection approach.** We build a new attention-based mechanism bug detection approach that learns to aggregate different AST path-based code representations into a single vector of a code snippet and to classify the code snippet into buggy or non-buggy.
- **An extensive comparative evaluation and analysis.** Through a series of empirical studies, our results show that our approach outperforms the state-of-the-art ones. We also compare our learned code representation with other existing ones and the comparative empirical results show that our code representation is more suitable for detecting bugs than others.

3.2.2 Motivation

3.2.2.1 Motivating example. In this sub-section, we will present a real-world example and our observations to motivate our approach.

Figure 3.2 shows an example of a real-world defect in the project named *hive* in GitHub. The bug involves three methods in which the method *getSkewed-ColumnNames* (method 1) retrieves the column type information from the input alias via the method *getTableForAlias* (method 3) and then compares it with the provided column names via the method *getStructFieldTypeInfo* (method 2) in order to find the suitable constant description for the current processing node. The bug occurred due

Method 1

```
1  public List<String> getSkewedColumnNames(String alias) {...
2      else {
3          TypeInfo typeInfo = TypeInfoUtils.getTypeInfoFromObjectInspector
4              (this.metaData.getTableForAlias(tabAlias).
5              getDeserializer().getObjectInspector());
6          desc = new exprNodeConstantDesc(typeInfo.getStructFieldTypeInfo
7              (colName), null);
8          }...
9      }
```

Method 2

```
1  public TypeInfo getStructFieldTypeInfo(String field) {
2      String fieldLowerCase = field.toLowerCase();
3      for(int i=0; i<allStructFieldNames.size(); i++) {
4          if (field.equals(allStructFieldNames.get(i))) {
5              return allStructFieldTypeInfos.get(i);
6          }
7      }
8      throw new RuntimeException("cannot␣find␣field␣" + field +
9          "(lowercase␣form:␣" + fieldLowerCase + ")␣in␣" +
10         allStructFieldNames);
11     }
```

Method 3

```
1  public Table getTableForAlias(String alias) {
2  -   return this.aliasToTable.get(alias);
3  +   return this.aliasToTable.get(alias.toLowerCase());
4  }
```

Figure 3.2 Bug Detection: A motivating example from the Project *hive*.

to the inconsistency in handling the case-sensitivity of the names: the type field name is in the lowercase (line 2, method 2) while the alias name is not. To fix this bug, the developer added a method call to convert the alias name into lowercase (lines 2–3, method 3).

From this example, we have drawn the following observations:

Observation 1 (O1). This bug involves multiple methods. For developers to completely understand this bug or for a model to automatically detect this, it is necessary to consider multiple methods and the dependencies among them. This type of cross-method bugs could easily occur in practice. However, the existing ML-based bug detection approaches [13, 125] examine the code within a method individually, without considering the inter-procedural dependencies. As an example, NAR-miner [13] derives non-association rules (e.g., if there is A, then there is no B), and uses them to detect bugs occurring in each individual method. That approach cannot detect this bug because it considers each method individually and this bug does not involve a non-association rule on the appearances of any elements. As another example, DeepBugs [125] uses deep learning on the names of the program entities in each method to detect bugs.

DeepBugs cannot detect this bug either because it does not perform intraprocedural analysis. Moreover, the fixed method *getTableForAlias* does not contain the names similar to those of buggy methods. In fact, *toLowerCase* is a widely used API and *get* is a popular method name. A model cannot determine the error-proneness for this method by solely relying on those popular names.

Observation 2 (O2). When considering multiple methods to detect a bug, there exist multiple paths on the representations that would be used to model the methods and their interdependencies, e.g., the control flow graph (CFG), program dependence graph (PDG), or the abstract syntax tree (AST). Due to the large sheer amount of paths needed to be considered, a model should not put the same weights

on all the paths. To learn from database of buggy code in the past, a model should put more weights on the buggy paths than others.

However, existing approaches [7, 111] to represent code as graph-based embedding vectors are either putting weights on frequently occurring paths or not weighting at all. For example, code2vec [7] extracts the paths in the AST among program entities with dependencies and gives more weights to frequent paths. Those paths are used as input to a deep learning model to learn the graph-based embedding vectors for source code. Frequent paths might not be the most error-prone ones in a program. In contrast, the buggy paths might not be the frequent ones. For example, the lines 5–7 in the method 1 are in a buggy path, however, they are not one of the frequent paths in our collected data for our experiment (which will be explained later). This is reasonable because code2vec [7] was designed for measuring code similarity, rather than for bug detection. On the other hand, Exas [111] represents source code with a vector by encoding the paths with up to certain lengths in the PDG. While it is successful in code similarity at the semantic level, it considers all the paths with the same weights, thus, cannot be applied well to bug detection.

3.2.2.2 Key ideas. With the above observations, we have built our approach with the following key ideas. First, to capture the source code containing defects, in addition to represent the body of the given code, we also use as the global context the Program Dependency Graph (PDG) [36] and Data Flow Graph (DFG) [185]. Such graphs enable us to consider the dependencies among program entities across multiple methods, thus, enabling the representation of the buggy source code involving multiple methods. In our example, it allows our model to capture the relationships among the methods 1, 2, and 3 involving in the current bug. Specifically, it allows the connections of the important nodes such as the connection between the variable *typeInfo* at line 5 of the method *getSkewedColumnNames* (Method 1) and the parameter *alias* at line 2 of the method *getTableForAlias* (Method

3), and the connection between the variable *typeInfo* at line 7 of the method *getSkewedColumnNames* (Method 1) and the variable *field* at line 2 of the method *getStructFieldTypeInfo* (Method 2). From the two connections, the relation between the line 2 of the Method 2 and the line 2 of the Method 3 can be captured. Thus, if our model has seen a similar relation that causes a bug in the training data, it could catch this in the example.

Secondly, from Observation 2, we design an attention-based deep learning model to emphasize to learn the buggy paths and use the PDGs and DFGs as the context to capture the relations among the methods involving in a bug. The attention mechanism also helps in improving the ranking of buggy candidates, thus improving the recall that was potentially affected by the use of PDG and DFG in code matching. For example, in the history, there exists a bug that are revealed by the above connection between the line 2 of the Method 2 and the line 2 of the Method 3. By adding a weight for buggy paths with the attention mechanism, we could make all buggy paths have a higher weight than the normal paths. In our approach, we would like to only use long paths because the short paths are covered by the longer ones. In our example, the long paths can sufficiently cover the needed information between the three methods to detect the bug.

3.2.3 Approach overview

Let us explain the overview of our approach. To determine whether source code in a given method is buggy or not, our model relies on the following three main steps as illustrated in Figure 3.3:

- **Step 1: Attention-Based Local Context Representation Learning**. First, our model parses the given method to build an AST. It extracts the long paths and then uses the attention GRU layer and the attention convolutional layer to build the representation for the method's body. Let us call it the local context because the paths are extracted within the given method.
- **Step 2: Network-Based Global Context Representation Learning**. Second, in addition to considering the given method's body, we also encode

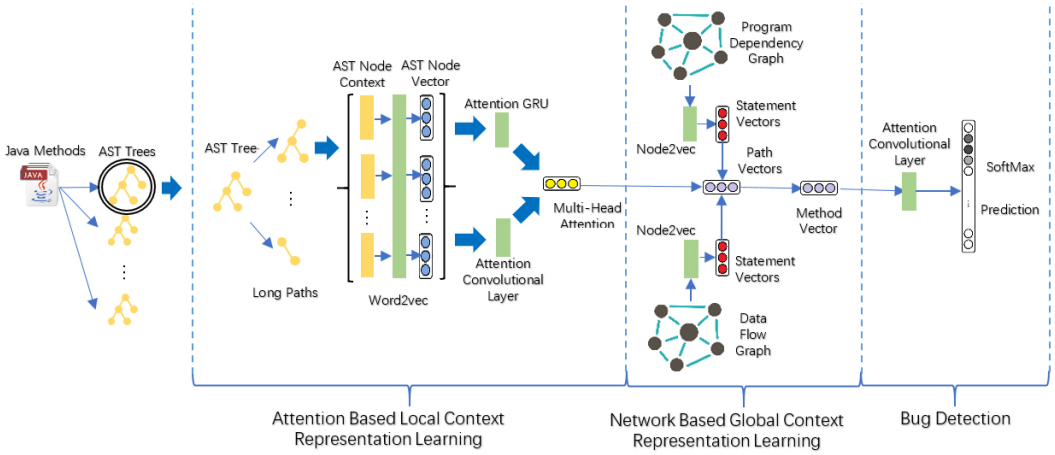


Figure 3.3 Bug Detection: Overview of our approach.

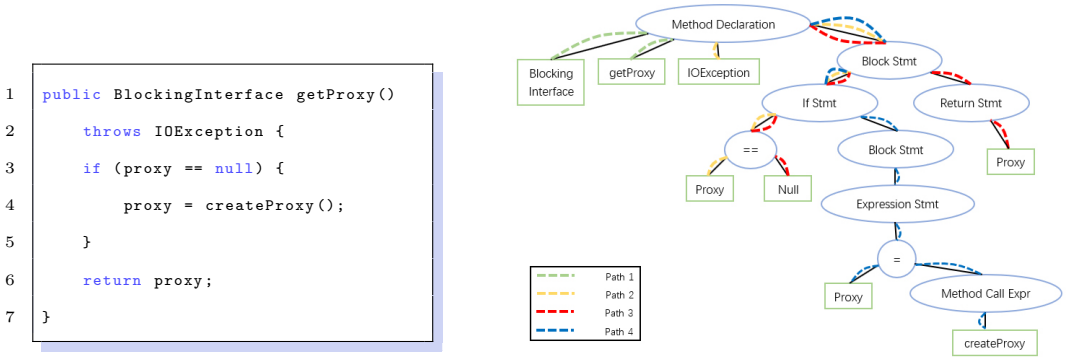
the context of the method by building the Program Dependence Graph (PDG) and the Data Flow Graph (DFG) relevant to the method. We call them the global context because they provide the relations between the given method and other relevant methods in the project. We use the Node2Vec [39] for the encoding of the PDG and DFG.

- **Step 3: Bug Detection**. Finally, with the local and global contexts of the given method, we use a softmax-based classifier to decide whether the method is buggy or not.

In the step of building the local context, we choose the long paths over the AST built from the method's body. A long path is a path that starts from a leaf node and ends at another leaf node and passes the root node of the AST. As shown in previous works [7, 111], the AST structure can be captured and represented via the paths with certain lengths across the AST nodes. The reason for a path to start and end at leaf nodes is that the leaf nodes in an AST are terminal nodes with concrete lexemes. The nodes in a path are encoded into a continuous vector via Word2Vec and the vectors are fed into two layers: attention-based GRU layer [25] and attention Convolutional layer [183]. The GRU layer allows our model to encode and emphasize on the order of the nodes in a path, i.e., on the parent-child relationships of the AST nodes. In other words, the nested structures in an AST are captured and represented with GRU layer. Moreover, the attention-based Convolutional layer allows our model to emphasize and

put more weights on the buggy paths. After that, we use Multi-Head Attention [160] to combine the result from attention GRU layer and attention Convolutional layer together as the path local context representation.

In the step of building the global context, we use the Node2Vec [39] to encode the PDG and DFG into embedded vectors. These two vectors are combined by the space vectors of all nodes in each path. We use matrix multiplication and convert results together to get the path representation vector. Then, we can have method representation by appending all paths' vectors for each method. For the bug detection step, with the method vectors we can do the bug detection with a softmax-based classifier. We will explain all the steps in details next.



(a) A method example. **(b)** The AST of the code in Figure 3.4a.

Figure 3.4 Bug Detection: Source code example and the AST from the project *hive*.

3.2.4 Step 1: Attention-Based local context representation learning

Given a method, we use the following steps to learn a code representation using AST paths within the method. We call it the local context as the paths used for code representation learning are within the method. Figure 3.5 shows the overall steps of learning local context code representation.

3.2.4.1 Path extraction. We extract paths from an AST built for a method, instead of source code directly, because an AST helps capture better code structures.

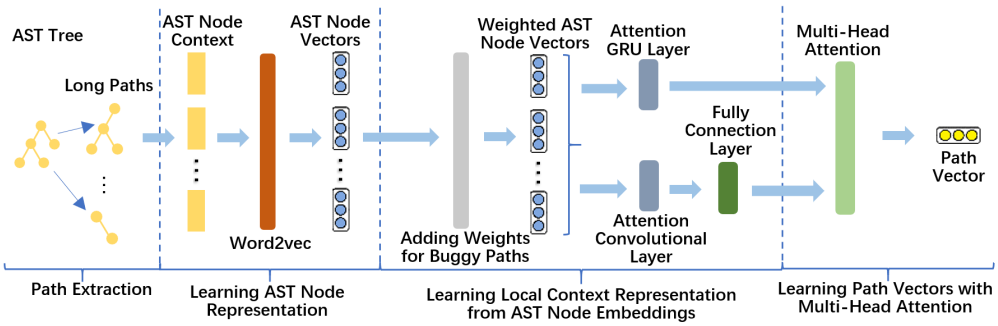


Figure 3.5 Bug Detection: The process of learning local context representation.

With the explicit representation of structures via ASTs, our model could make distinction better between buggy and non-buggy code structures.

We use the well-known and widely-used Eclipse JDT package to build an AST for a given Java method. As we do not want to lose important information of each method for bug detection, we extract the minimum number of long paths that can cover all nodes in an AST of a method in a greedy way. In Figure 3.4, for the method in Figure 3.4a, we build an AST and extract four long paths as shown in different colors in Figure 3.4b. As a long path passes from a leaf node to another via the root node in an AST, there are overlapping nodes among different long paths.

3.2.4.2 Learning AST node representation. We use Word2vec [99], a neural network that takes a text corpus as an input and generates a set of feature vectors for words in the corpus, to learn a vector representation for each AST node. This step takes all of the AST nodes of a method as the input, and generates a learned vector representation for each given AST node.

Specifically, in the process of AST node encoding, we treat the node content of an AST node as one word. For example, in Figure 3.4b, the node "Block Stm" having a real value "{}" in AST is considered as one word, $word =$ "{}". Thus, we generate a sequence of words for each path. For instance, we can generate the following ordered set of words: {"Null", "==", "if()", "{}", "root","{}", "Return", "proxy"}, for the red path in Figure 3.4b, where "if()" is the *If Stmt*, the first "{}" right after *If Stmt*

is the *Block Stmt*, the "root" is the *Method Declaration*, the second "{}" is also the *Block Stmt* as the path passes through the root.

Given a version of a project, we extract long paths from an AST for each method and generate ordered sets of words for each path.

We order the nodes in an AST path according to their appearance order in source code. We generate embeddings for each node and preserve their order. Moreover, we do not embed comments and do not differentiate specific strings and numbers during embedding, as their values are normally too specific and do not contribute to model training for bug detection.

Collectively, we obtain a large corpus of words for AST nodes from all source code under study. Each node is mapped to a word. We run Word2Vec on a large corpora to learn a vector $NodeV_i$ to represent an AST node n_i in a path of nodes $P = n_1, n_2, ..., n_i$. All nodes in training are considered to maximize the log value of the probability of neighboring nodes in the input dataset. We use Word2Vec to train our own node representations by using all of the AST nodes from each project. The loss function is defined as follows:

$$Loss_i = \min_i \frac{1}{i} \sum_{j=1}^{i} \sum_{k \in NNS^r} -\log HS\{NodeV_k | NodeV_j\}$$

$$L = \min \frac{1}{i} \sum_{j=1}^{i} \sum_{k \in NNS} -\log HS\{NodeV_k | NodeV_j\}$$

where L is the lose function for the nodes in $P = n_1, n_2, ..., n_i$, NNS is the set of the neighboring nodes of a node n_i, and $HS\{NodeV_k | NodeV_j\}$ is the hierarchical softmax of node vectors $NodeV_k$ for the node n_k and $NodeV_j$ for the node n_j.

3.2.4.3 Learning local context representation from AST node representations.

After the previous step of learning AST node representation,

each path, P, can be represented as an ordered set of AST node vectors, $P = \{NodeV_i, NodeV_i, \ldots, NodeV_n\}$, where n is the total number of nodes of the path, and $1 \leq i \leq n$.

To incorporate the previous buggy information into the representation learning, we add a weight to a path if the path passes an AST node that is in one or multiple bug fixes. We use the addition of weights to differentiate buggy and non-buggy AST paths. Specifically, if a node n_i from a path was in a bug fix, we apply the same weight w on all of the node vectors in the path, $P = w * \{NodeV_i, NodeV_i, \ldots, NodeV_n\}$, and the paths without any nodes in previous bug fixes have no weight added. For example, if there is a bug fix (e.g., removing the whole line) in the line 4, *proxy = createProxy()*, of the code example in Figure 3.4a, all of the node vectors of the blue path in Figure 3.4b are assigned with the same weight w.

To learn a unified vector representation from the node vectors for a path, we use two different approaches to learn path vectors capturing different aspects of key information.

[1] Attention-Based GRU approach. We apply an attention-based Gated Recurrent Unit (GRU) layer [25], using an attention layer on top of the GRU layer as the weights to apply the importance on each time step during the training, to learn a path vector from a given set of node vectors. We use GRU to capture the sequential patterns from ASTs. The GRU layer, using a gating mechanism reported in [25], is a powerful and efficient model for learning a representation from a given set of vectors. There are two key gate calculations at a time step t, the reset gate r_t and the update gate z_t. The following calculation is for the j-th hidden unit at the time step t:

$$r_t^j = \sigma(W_r x_t + U_r h_{t-1})^j \tag{3.1}$$

$$z_t^j = \sigma(W_z x_t + U_z h_{t-1})^j \tag{3.2}$$

$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \tilde{h}_t^j \tag{3.3}$$

$$\tilde{h}_t^j = tanh(W x_t + U(r_t \odot h_{t-1}))^j \tag{3.4}$$

where x_t is the node vector for the AST node$_k$ in path$_i$ as an input of the t time step, j denotes the j-th element of a vector, the h_t^j, h_{t-1}^j is the j-th element of the output embedding vectors at the t and $t-1$ time steps, the r_t^j is the j-th element of the reset gate vector at the t time step, the z_t^j is the j-th element of the update gate vector at the t time step, σ is the logistic sigmoid function, W, U are the parameter matrices that can be learned during the training and the $\odot$ is the Hadamard product.

Given a sequence of AST node vectors of a path, we pass one node vector to the GRU layer at each time step and the GRU layer also generates an output vector at each time step. At the final time step of the GRU, one path vector is generated. Over the whole process, the GRU takes a set of node vectors as input vectors and produces a set of intermediate output vectors (i.e., the last output vector is the final generated vector). We store the set of input vectors and the set of intermediate output vectors from the GRU layer for the next step.

[2] Convolutional-Based Approach. We use a Convolutional layer in the Convolutional Neural Networks (CNN) [31], with an attention mechanism on top of it, to learn a path vector from a given set of node vectors. In the CNN, sequences of node vectors are modeled into matrices. We use the CNN to capture the local coherence patterns from the matrices of ASTs. Different node embeddings can be used to construct a matrix $\mathbf{D}$, where it has a structure $n \times d$ of $\mathbf{D}$ with only one channel (d is the size of node embedding).

In a convolution operation, a filter can convolute a window of nodes (e.g., 3 or 4) to produce a new feature using the following equation: $c_i = \sigma(\mathbf{W} \cdot \mathbf{h}_\ell(x) + \mathbf{b})$, where c_i is the dot product of $h_\ell(x)$ and a filter weight $\mathbf{W}$. $h_\ell(x)$ is a region matrix for the region x at location ℓ. σ is non-saturating nonlinearity $\sigma(x) = \max(0, x)$ [65]. Then the filter can convolute each possible window of nodes and produce a feature map: $\mathbf{c} = [c_1, c_2, ..., c_i, ...]$. The weight $\mathbf{W}$ and biases $\mathbf{b}$ are shared during the training process for the same filter, which enables to learn meaningful features regardless of the window and memorizing the location of useful information when appearing.

Given a sequence of AST node vectors of a path, we use all node vectors to build a $n * m$ matrix, where n is the number of node vectors and m is the length of a node vector, and send the matrix to the Convolutional (Conv) layer. At each time step, a sub-matrix is selected and used as an input for the Conv operation and the Conv layer generates an output matrix. Over the whole process, a set of matrices are built and used as inputs for the Conv layer that produces a set of intermediate output matrices (i.e., the intermediate output matrix at the last time step is the final matrix). In the Conv layer, a sequence of node vectors is transformed into a set of matrices. To reduce an obtained matrix into one dimension vector, we apply another layer: fully connected layer in the CNN [31]. We pass intermediate output matrices generated at different time steps into the fully connected layer to generate 1-dimensional intermediate vectors. We store the set of input matrices and the set of intermediate output 1-dimensional vectors from the Conv layer for the next step.

The attention-based mechanism enables our model to emphasize on the important paths that have been observed to be buggy. That is the key advantage of our model in comparison with the traditional or vanilla (standard backpropagation) GRU and CNN. While attention mechanism allows a NLP model to focus on certain important words in a sentence, it is expected to help our bug detection model to put and update more weights on the observed buggy paths.

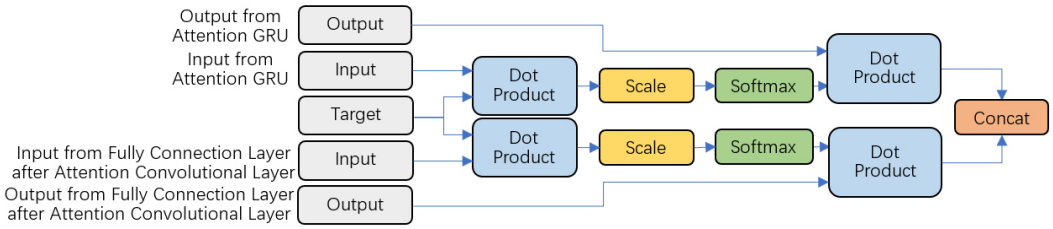


Figure 3.6 Bug Detection: Multi-Head attention.

3.2.4.4 Learning path vectors with multi-head attention. In the previous step, given a sequence of AST nodes, we use two approaches to learn vector representations for a path and different vector representations capture different aspects of information of a path. To learn the unified vector for a path, we still need to find a way to combine the two path vectors into one code vector without too much information loss.

To do so, we apply the Multi-Head Attention (MHA) model [160] to learn a unified representation. The Multi-Head Attention model is effective in learning representation form different other representations. We build the MHA on top of the GRU and Convolutional (Conv) layers as shown in Figure 3.6. Due to the page limit, we only introduce the basics of MHA. In our case, we build a two-head attention and both heads have the exact same architecture, but they take different inputs. Both GRU and Conv layers take input vectors at different time steps and generate the corresponding output vectors at different time steps.

One head for the GRU layer, namely H^G, takes the following inputs: a training target vector (T), all of the input vectors of the GRU layer at different time steps, namely V_G^I, and all of the intermediate output vectors from the GRU layer at different time steps, namely V_G^O. We define a target vector to have the same length as the input vector. Also, we set all values in the T as 1 for buggy and 0 for non-buggy. V_G^I and V_G^O are both obtained in the previous step in Sub-Section 3.2.4.3. Then the MHA conducts a dot product between the T and the V_G^I (i.e., AST node vectors), and scale the product result by dividing it using the square root of the vector dimension of V_G^I,

denoted as d. After the scale process, we apply a *softmax* function on the result of the scale process, denoted as $softmax(\frac{T \cdot V_G^I}{\sqrt{d}})$. Last, we apply the doc product operation between the result of the *softmax* operation and all of the intermediate output vectors from GRU, V_G^O. The whole process can be expressed as $V_G^p = softmax(\frac{V^T \cdot V_G^I}{\sqrt{d}}) \cdot V_G^O$, where V_G^p denotes a path vector learned from H^G.

Another head for the Convolutional layer, namely H^C, works the same way as H^G, except that H^C takes the input vectors and the intermediate output vectors from the Convolutional layer. We use V_C^p to denote a path vector learned from H^C.

The final step of MHA is to concatenate V_G^p and V_C^p to generate a unified one-dimensional path vector that incorporates local context within a method.

3.2.5 Step 2: Network-Based global context representation learning

3.2.5.1 Overview. As shown in Sub-Section 4.3.2.1, a bug can involve multiple methods, thus it is critical to model the relations among methods, even the paths in different methods, into code representations for bug detection. To complement the local context of buggy code, which is represented by buggy paths in the AST, we capture the global context to integrate the relations among buggy methods into our model via program and data flow dependencies. Figure 3.7 shows the general overview of our process to model the global context. The first step is to extract the Program Dependence Graph (PDG) and Data Flow Graph (DFG) from the source code of a Java project. In the second step, the graphs are used as the input for a process to vectorize the nodes. To achieve that, we use the Node2Vec [39] to capture the data and control dependencies between relevant program statements. Then, at the third step, the related statements in the long paths in the AST identified in the previous step for the local context are used and encoded via the representation vectors computed via the Node2Vec [39]. Finally, the representation vectors for the PDGs and those for the DFGs are combined via matrix multiplication. The resulting vectors are then integrated with the vectors for the local context computed earlier to

produce the path representation vectors, namely global context representation. Let us explain each step of this process in details.

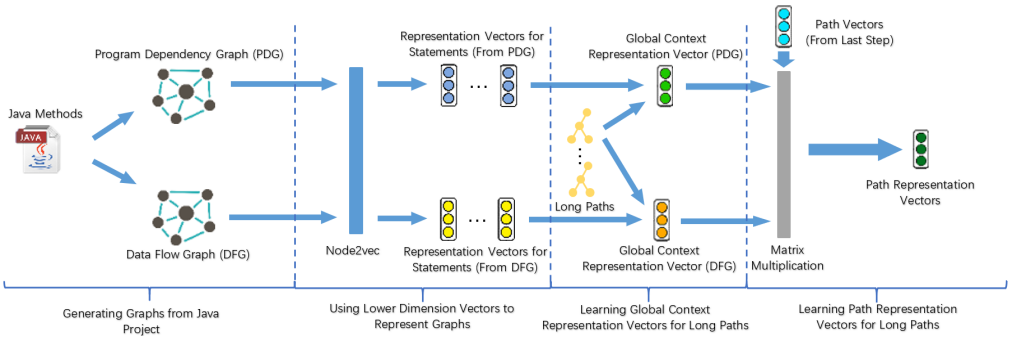


Figure 3.7 Bug Detection: Learning global context representation and generating long path representation vectors.

3.2.5.2 Generating graphs from Java projects. As we use path-based code representation to model a method, we aim to represent the fine-grained relations among methods, i.e., the relations among paths from different methods. Specifically, we use the Program Dependence Graph (PDG) [36] and Data Flow Graph (DFG) [185]. Given a version of a project, we generate the PDG and DFG for the entire project at the statement level. We used the Eclipse plugin Soot [147] to produce the PDG, and the plugin WALA [162] to produce the DFG.

For the PDG, we use the classes from *soot.toolkits.graph.pdg* in Soot to implement a Program Dependence Graph as defined in [36]. Soot can handle inter-procedural analysis for PDGs. As for the DFG, we use the classes from *com.ibm.wala.dataflow* in WALA to generate a data flow graph [59]. In our problem, we would like to generate a large data flow graph which contains all data flows in a whole project. Given a project, the WALA can generate a set of data flow graphs for the project and we connect them to build the entire DFG for the whole project.

Currently, for both PDG and DFG, we handle virtual method calls in a conservative way using declared types for program entities. We support no

pointer analysis, data flow through heap-allocated objects is approximately and conservatively captured via explicitly declared objects.

3.2.5.3 Using lower dimension vectors to represent graphs. Once the two graphs are generated for a project, we convert the large graphs (e.g., a PDG can be very large with a high number of nodes and edges) to low-dimensional vectors without much information loss for efficient processing. We apply the widely used network embedding algorithm, the Node2Vec [39], to encode all of the nodes in our PDGs and DFGs. During the learning of node embeddings, the Node2Vec can encode a node with the information of the node's surrounding structures.

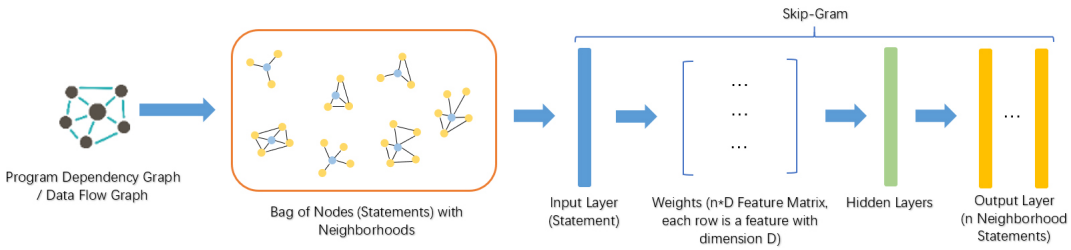


Figure 3.8 Bug Detection: Using lower dimension vectors to representation graphs.

Technically, for each graph (i.e., the PDG or DFG), a bag of nodes is extracted in which each node m represents a code statement and the neighboring nodes of m represent the code statements with dependencies on m. A neural network in the form of a skip-gram model [100] is trained with the input layer containing each node m for a statement and the output layer containing the neighboring nodes of m for the statements with dependencies on the current statement. The output of the process is the feature matrix with the dimension of $n \times D$, where n is the number of nodes/statements in the input layer and D is the number of representation features in the lower dimension vector space. In other words, each row is a feature vector representing a code statement in the input graph. The representation vectors capture the neighboring structures of the statements/nodes.

As it is not designed for source code, the Node2Vec models the network data that can flow two ways between two nodes. Our graphs, PDG and DFG, are one-directional. Therefore, we adapt the Node2Vec in the following ways. Inspired from NAR-miner [13], we set the weight $weight = 1$ to a directional edge from node A to node B if these two nodes have a dependency in a program graph (i.e., the PDG or DFG). We also assign another weight $weight = -1$ to the opposite directional edge from node B to node A, meaning that there is no dependency from B to A in a program graph. For example, in the example code in Figure 3.4a, there is a relationship between $if(proxy == null)$ and $proxy = createProxy()$. In the PDG, the weight on the edge from $if(proxy == null)$ to $proxy = createProxy()$ is 1, and the weight on the edge from $proxy = createProxy()$ to $if(proxy == null)$ is -1. Then, we apply the Node2Vec on the PDG and DFG, separately to compute network embedding, and obtain a vector for each node representing each code statement.

3.2.5.4 Learning global context representation vectors for long paths. Once we obtain the vector representations for all of the nodes/statements in the PDG and DFG, we use them to encode the long paths in the AST that were extracted in the local context computation step. We denote a node in a PDG as N^P, and a node in a DFG as N^D. Several long paths can go through the same statement and a statement can have multiple AST nodes. For example, in Figure 3.4b, the statement, $if(proxy == null)$, includes four AST nodes: *Proxy*, ==, *Null*, and *If Stmt*. There are three paths, the yellow, red, and blue paths, passing through these nodes. Therefore, given a project, we can extract all of the mappings between the long paths and the statements, where a path is mapped to a set of unique statements and each statement is a node in the graph (i.e., a PDG/DFG). The vectors representing each of those statements, i.e., the nodes in the PDG and DFG, were computed via the Node2Vec in the previous step. Thus, we can use the vectors for those nodes N^P and N^D to capture the global context and represent a long path in the AST. Specifically, a long

path is represented by the average vector of the vectors for the corresponding PDG nodes (namely V^P_{PDG}) and the average vector of the vectors for the corresponding DFG nodes (namely V^P_{DFG}).

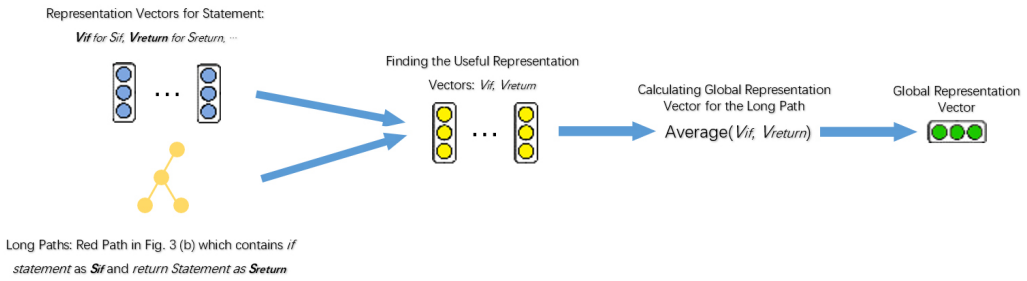


Figure 3.9 Bug Detection: Learning global context representation vectors for long paths.

3.2.5.5 Learning path representation vectors for long paths. In the final step for learning the representation vectors for long paths, we combine the local context and the global context vector representations for the paths. Given a path with a local context vector representation (in Sub-Section 3.2.4), denoted as V^P_{local}, we use the following steps to combine V^P_{PDG} and V^P_{DFG} with V^P_{local}: First, we apply Matrix Multiplication (denoted as $\cdot$) to V^P_{local} and V^P_{PDG}, and also V^P_{local} and V^P_{DFG}. Then we can have the path representation vector by simply merging the above two results: $V^P = V^P_{PDG} \cdot {V^P_{local}}^T, V^P_{DFG} \cdot {V^P_{local}}^T$. Matrix multiplication can effectively combine such vectors to make the combined vector more expressive. We tested other aggregation mechanisms, e.g., vector concatenation, and matrix multiplication produces better result.

Once we have all of the path representation vectors, a method M can be represented as a set of path vectors with local and global contexts: $M = \{V^P_1 \dots V^P_i \dots V^P_n\}$, where V^P_i is the unified path vector for the i-th path in M, $1 \leq i \leq n$, and n the total number of long paths for the method M. Within a project, each method can have a different number of long paths. To make sure all of the methods can be modeled with the same number of path vectors, we choose

the method with the largest number of long paths among all methods, and use that number as the default value for the number of paths to model a method. If a method has fewer long paths than the default number, we perform zero padding for the vector representations.

3.2.6 Step 3: Bug detection

After the above two steps, we obtain a representation for each method. We build a classic CNN architecture to classify whether a method is buggy or not, given a set of method representations. We build the following layers to process method representations: First, a Convolutional layer is applied on the set of method representations. Second, the Max pooling and fully connected layers are applied. Third, we use a SoftMax layer to the classification.

3.2.7 Empirical evaluation

3.2.7.1 Research questions. We have conducted several experiments to evaluate our model. Specifically, we seek to answer the following research questions:

RQ1. Bug Detection Comparative Study. How well our approach perform in comparison with the existing state-of-the-art bug detection approaches?

RQ2. Code Representation Comparative Study. Is our code representation with local and global contexts more suitable than existing code representations in bug detection?

RQ3. Sensitivity Analysis. How do various factors affect the overall performance of our approach?

3.2.7.2 Experimental methodology.

3.2.7.2.1 Data collecting and processing. We conduct our study on eight well-known and large open-source Java projects with different versions of each project. In total, we got 92 versions of these projects with +4.9 million Java methods (Table 3.1). For each project, we collect source code and commits from the Github repository, and

Table 3.1 Bug Detection: Statistics on Dataset.

Project Name	# of Versions	# of Files	# of Methods	# of Buggy Methods
pig	9	8k	95k	21k
avro	7	2k	30k	1k
lucene-solr	14	93k	1.032M	518k
hbase	9	14k	318k	258k
flink	14	49k	419k	173k
hive	18	53k	981K	411k
cloudstack	11	55k	766k	307k
camel	10	172k	1.332M	135k
Total	92	402K	4.973M	1.824M

bug reports from the issue tracking system of the project. Specifically, we use the following steps to process the data of a project:

- First, we download all bug reports that are marked as *resolved or closed* and *bug* from the JIRA issue tracking system. The details of a report has a field named *Fix Versions* which indicates the bug fix locations. We extract the bug id and the version numbers from a bug report.
- Second, for a version of a project, we download the commits from the Git repository. We use the same approach as the ones used in [102, 130, 128] to process each commit message and mark it as a *bug-fix* if the message contains a bug id and at least one of the error related keywords: {error, bug, fix, issue, mistake, incorrect, fault, defect, flaw and type}.
- Once we identify all of the bug-fixes in the previous steps, we download the source code of the project version as a clean version and use the bug-fix commits to recover the *buggy* version from the source code, as a code commit records all of the additions and deletions. Specifically, we use the additions in commits to locate the methods where code fixes occurred, and then roll the methods back to the states before the fixes to obtain the buggy code.

3.2.7.2.2 Experiment setup and procedure.

To answer our research questions, we use the following procedures and setups.

RQ1. (Bug Detection Comparative Study) Analysis Approach.

Comparable Baselines: We compare our approach with the following state-of-the-art approaches:

- **DeepBugs**[125]: DeepBug is a bug detection approach with a deep learning model on the name-based information in source code. We used the default values of their model and tried different values for vocabulary size, and kept the value giving the best result.

- **Bugram**[164]: Bugram uses n-gram to evaluate a given method and decides its bugginess by picking the top-ranked possibilities for each n-gram. With our dataset, we tried various values of n and sequence lengths, and kept the ones giving best results.

- **NAR-miner**[13]: NAR-miner mines negative rules on the code (e.g., if A then not B), and then use them to detect bugs. We used the default values for their model.

- **FindBugs**[9]: FindBugs is an open-source static analysis tool that analyzes Java class files to detect program defects. The analysis engine statically encodes more than 300 different bug patterns using a variety of techniques. We used the default values for FindBugs.

We conduct our experiments in two settings:

- **Detecting bugs in unseen projects (Cross-Project).** This setting is used to test the ability of a model to detect bugs on unseen projects (i.e., on a project that is not included in the training data). We train a model on all of the versions of 7 randomly chosen projects in our dataset, and test the trained model on the remaining project. We repeat 8 times for cross validation and calculate the average.

- **Detecting bugs in unseen versions of a project.** This setting is used to test the ability of a model trained on all of the existing versions of a project and other projects to detect bugs on an unseen version of a project. We use all of the versions of 7 randomly picked projects and all of the previous versions of the 8th project as the training data and use the newest version of the 8th project as the testing data.

Qualitative Analysis: In this experiment, for comparison, we make qualitative analysis by comparing the top 100 results that each model reported on the same

randomly chosen version of a project. We manually verified each reported bug. We computed how many true bugs each model can detect in top 100 results, how many true bugs detected by the baseline models were covered by our model, how many bugs our model did not cover, and how many new bugs our model can find when comparing with the baseline models.

Tuning our approach and the baseline models: We tuned approaches in the cross-project setting. For simplicity, we use the same set of parameter settings for approaches in both of the above mentioned experimental settings once the best settings are identified.

We tuned our model with the following key hyper-parameters:

1. Epoch size (i.e., 100, 200, 300),
2. Batch size (i.e., 32, 64, 128, and 256),
3. Learning rate (i.e., 0.005, 0.010, 0.015),
4. Vector length of word representation and its output (i.e., 150, 200, 250, 300),
5. Convolutional core size (i.e., $1x1$, $3x3$, $5x5$), and
6. The number of convolutional core (i.e., 1, 3, and 5).

We tuned the baseline models with some parameters to obtain the best results on our dataset. We tuned the vocabulary size for the *DeepBugs*, the gram size, sequence length range, minimum token occurrence, and reporting size for the *Bugram*, and the threshold of frequent itemsets, the maximum support threshold of infrequent itemsets, the minimum confidence threshold of interesting negative rules min_conf for *NAR-miner*. FindBugs is a rule-based tool and we directly used its default setting.

RQ2. (Code Representation Comparative Study) Analysis Approach.

In this work, for bug detection, we introduce a novel path-based code representation with graph-based local and global contexts. We aim to compare our representation with other baseline representations that are used for source code similarity in the context of bug detection.

Comparable Baselines: We compare our code representation with the following state-of-the-art code representations on bug detection:

- **DeepSim**[195]: DeepSim represents source code for code similarity. It encodes control flows and data flows into a semantic matrix in which each element is a high dimensional sparse binary feature vector.
- **code2vec**[7]: code2vec uses most frequently paths on the AST from one leaf node to another via going up in the tree.
- **Code Vectors**[47]:The approach uses abstractions of traces obtained from symbolic execution of a program as a representation for learning word embedding.
- **Deep Learning Similarity** (DL Similarity) [157]: This approach applies deep learning on 4 different types of representations, including Identifiers, AST, Control Flow Graph, and Bytecode of a method, to learn a code representation.
- **Tree-Structured LSTM** (Tree-Based LSTM) [151]: Tree-Based LSTM gets the representation of each method by training a tree-structured LSTM model with the AST of the source code.

As those representations are not aimed for bug detection, to be fair, we compare only the code representation part of our approach with those representations. To do so, we build a baseline as follows: *we use each of those code representations with our model to detect bugs.*

Similar to the analysis approach for RQ1, we also conduct our experiments in the two settings cross-project and cross-version, and the top-ranked qualitative analysis.

Tuning our model and the baselines: As in RQ1, we use the same set of parameter settings of an approach for both experimental settings. For our approach, we use the best parameter settings learned in RQ1 to run our approach in this experiment. For the baselines, there are no key parameters in the code representation learning of the baselines.

RQ3. Sensitivity Analysis Approach.

For sensitivity analysis, we would like to evaluate how various factors including paths over AST, multi-head attention, Program Dependency Graph, and Data Flow

Graph impact on our model's accuracy in bug detection. To perform sensitivity analysis, we add each element into the model one by one in each of the two settings with different parameters.

3.2.7.2.3 Experiment metrics. We use the following metrics to measure the effectiveness of a model:

$$Recall = \frac{TP}{TP + FN},\ Precision = \frac{TP}{TP + FP},\ F\_score = \frac{2 * Recall * Precision}{Recall + Precision}$$

Where TP = True Positives; FP = False Positives; FN = False Negatives; TN = True Negatives.

Recall measures how many of the labeled bugs can be correctly detected, while Precision is used to measure how many of the detected bugs are indeed labeled as a bug in the bug tracking system. Note that in the bug tracking system, there exist cases in which the occurrences of the labels or bug-indicating words do not really show bug fixes. Thus, to complement for that, in the comparative study, we picked 100 results and manually verified if they are truly bugs or not.

In our qualitative analysis, we also computed *related Recall, Precision, and F-score*. We use the term "related" to refer that we only consider the top 100 results from a model. For related Recall, we collect all true bugs found by a model in the top 100 results and regard them as the total true bugs. We calculate the related Recall in the top 100 results with the total true bugs. For related Precision, we regard top 100 results as the total reported results and calculate the Precision with 100 results. Thus, $TP+FP$ is equal to 100. For related F-score, we use related Recall and related Precision for the calculation in the same way as in F-score.

3.2.7.3 Experiment results.

3.2.7.3.1 Results of RQ1 (Comparative study on bug detection). As seen in Table 3.2, **our model outperforms the state-of-the-art bug detection baselines in the cross-project setting**. Specifically, our model improves over the

baselines in every measurement metric, except the recall values in the rule-based approaches NAR-miner and FindBugs. The Recall values of NAR-miner and FindBugs are 6% and 12% higher than ours. However, NAR-miner and FindBugs have a high false positives rate, i.e., 52% and 66%, that is approximately 1.5x and 2.1x higher than ours. False positives waste developers' time in investigating incorrect cases. **Our model improves F-score over the baselines DeepBugs, Bugram, NAR-miner, and FindBugs by 1.38x, 1.16x, 2.63x, and 3.57x, respectively**. Importantly, our model can generate fewer false positives than all of the baselines.

Table 3.2 Bug Detection: RQ1. Comparison with the Baselines in the Cross-Project Setting.

Category	Our Approach	DeepBugs	Bugram	NAR-miner	FindBugs
Recall	0.68	0.62	0.64	0.72	**0.76**
Precision	**0.39**	0.25	0.32	0.11	0.08
F-score	**0.50**	0.36	0.43	0.19	0.14
FP Rate	**0.21**	0.41	0.39	0.52	0.66

Notes: FP: False Positives.

Table 3.3 Bug Detection: RQ1. Comparison with the Baselines in Detecting Bugs in Unseen Versions of a Project.

Category	Our Approach	DeepBugs	Bugram	NAR-miner	FindBugs
Recall	**0.74**	0.63	0.67	0.68	0.73
Precision	**0.56**	0.27	0.41	0.22	0.15
F-score	**0.64**	0.38	0.51	0.33	0.25
FP Rate	**0.29**	0.37	0.38	0.35	0.43

Table 3.3 shows that **our model outperforms four state-of-the-art bug detection baselines in detecting bugs in the unseen versions of a project.**

Specifically, our model relatively improves DeepBugs, Bugram, NAR-miner, and FindBugs by 107%, 37%, 155%, and 273% respectively, in terms of Precision, and by 69%, 25%, 92%, and 156% respectively, in terms of F-score.

Furthermore, consolidating the results in Table 3.2 and Table 3.3, we can see that including previous versions of a project in the training can improve the overall effectiveness of all approaches. This is reasonable because including previous versions of the same project increases the knowledge to train the model. Our model obtains a large gain in terms of F-score (i.e., increasing 0.13 from 0.51 to 0.64), which shows that our model has a better learning ability in both settings. Although NAR-miner also obtains a large gain in terms of F-score, its F-score is still very low, i.e., 0.33.

Qualitative Analysis of RQ1.

Table 3.4 and Figure 3.10 show the overlapping analysis on the 100 top-ranked results from all of the models. As seen, our model discovers more true bugs than all other baselines. Specifically, 69%, 74%, 71% and 69% of the true bugs detected by DeepBugs, Bugram, NAR-miner, and FindBugs, respectively, can also be detected by our model. Although DeepBugs, Bugram, NAR-miner, and FindBugs can detect 8, 9, 7, and 4 true bugs that our model cannot detect, our model detects 30, 23, 31, and 39 more new true bugs than those baseline models, respectively. The key reason for our model to detect more bugs than DeepBugs and Bugram is that it combines both local and global contexts, along with program dependencies. On the other hand, the rule-based approaches, i.e., NAR-miner and FindBugs, are less flexible than our model because their rule-based detection engine is strict and cannot match the bugs that are not encoded in their dataset of rules.

3.2.7.3.2 Results of RQ2 (Code representation comparative study). Table 3.5 shows that **our code representation with local and global contexts is more suitable than other existing code representations in bug detection in the cross project setting.**

Table 3.4 Bug Detection: RQ1 Qualitative Analysis on the Top 100 Reported Bugs of the Approaches.

Category	Our Approach	DeepBugs	Bugram	NAR-miner	FindBugs
# of True Bugs	**48**	26	34	24	13
Related Recall	**0.70**	0.38	0.49	0.35	0.19
Related Precision	**0.48**	0.26	0.34	0.24	0.13
Related F-score	**0.58**	0.32	0.41	0.29	0.16

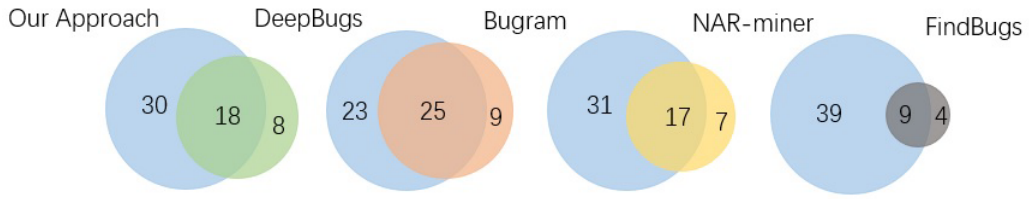


Figure 3.10 Bug Detection: Overlapping among results from our model and the baselines in RQ1.

Specifically, our approach can outperform the five baselines using different code representations in every measurement metric, except that Recall of Tree-Based LSTM code representation is 21% higher than ours. However, Tree-Based LSTM has much lower Precision (i.e., 9%) and higher false positives rate (i.e., 69%) that is 2.29X higher than our false positives rate, thus, making Tree-Based LSTM impractical.

Our model using path-based code representation with local and global contexts can improve relatively over all of the five baseline code representations: DeepSim, DL-similarity, code2vec, Tree-based LSTM, and Code Vectors by 67%, 38%, 82%, 206%, and 101%, respectively, in terms of F-score. Importantly, our false positives rate is lower than all of the baselines.

Table 3.5 Bug Detection: RQ2. Comparison with the Baselines in the Cross-Project Setting.

Category	Our Approach	DeepSim	DL Similarity	code2vec	Tree-Based LSTM	Code Vectors
Recall	0.68	0.67	0.71	0.69	**0.82**	0.70
Precision	**0.39**	0.19	0.24	0.17	0.09	0.15
F-score	**0.50**	0.30	0.36	0.27	0.16	0.25
FP Rate	**0.21**	0.35	0.36	0.43	0.69	0.45

Table 3.6 Bug Detection: RQ2. Comparison with the Baseline Code Representations in Detecting Bugs in Unseen Versions.

Category	Our Approach	DeepSim	DL Similarity	code2vec	Tree-Based LSTM	Code Vectors
Recall	**0.74**	0.69	0.57	0.64	0.61	0.59
Precision	**0.56**	0.42	0.33	0.41	0.21	0.25
F-score	**0.64**	0.52	0.42	0.50	0.31	0.35
FP Rate	**0.29**	0.38	0.34	0.39	0.44	0.41

Table 3.6 shows that **our code representation is also more suitable than other existing code representations in detecting bugs in unseen versions of a project by including other previous versions of the project in training data.** Our approach can outperform all of the baseline code representations in every measurement metric. Overall, the effectiveness of all approaches on detecting bugs in an unseen version of a project can be improved by adding more previous versions of the project into training.

Table 3.7 Bug Detection: RQ2. Qualitative Analysis on the Top 100 Reported Bugs of Each Model.

Category	Our Approach	DeepSim	DL Similarity	code2vec	Tree-Based LSTM	Code Vectors
# of True Bugs	**48**	25	34	39	21	23
Related Recall	**0.65**	0.34	0.46	0.53	0.28	0.31
Related Precision	**0.48**	0.25	0.34	0.39	0.21	0.23
Related F-score	**0.55**	0.29	0.39	0.45	0.24	0.26

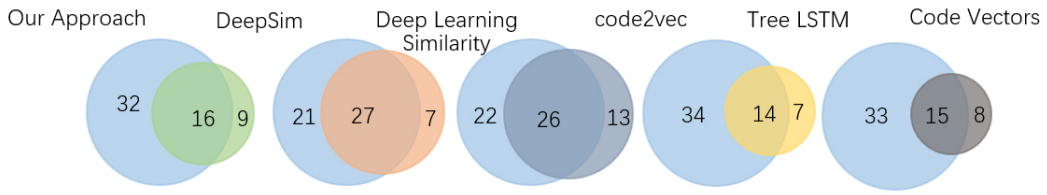


Figure 3.11 Bug Detection: Overlapping among results from our model and the baselines in RQ2.

Qualitative Analysis of RQ2. We conduct the overlapping analysis on the results from all of the models. Table 3.7 and Figure 3.11 show that our model detects 64%, 79%, 67%, 67%, and 65% of the bugs that DeepSim, Deep Learning Similarity, code2vec, Tree-Based LSTM, and Code Vectors detect, respectively. Although the five baselines can detect some true bugs that ours cannot detect, our approach can detect 32, 21, 22, 34, and 33 new true bugs that the code representations DeepSim, DL similarity, code2vec, Tree-Based LSTM, and Code Vectors cannot detect. Thus, our representation is more suitable than the baselines in bug detection in the cross-project setting.

3.2.7.3.3 Results of RQ3 (Sensitivity analysis). We conducted an experiment to study how different factors including the Local Contexts, the Multi-Head Attention, the Program Dependency Graph and Data Flow Graph (both graphs are global

contexts) affect our model's accuracy. Due to the page limit, we report the results in the cross-project setting.

Table 3.8 Bug Detection: RQ3. Sensitive Analysis of the Impact of Different Factors on Our Approach.

Models	Precision	Recall	F-Score
LC	0.19	0.75	0.30
LC+Attention	0.2	0.75	0.32
LC+PDG	0.29	0.54	0.38
LC+PDG+Attention	0.28	0.71	0.40
LC+DFG	0.26	0.51	0.34
LC+DFG+Attention	0.26	0.66	0.37
LC+PDG+DFG	0.37	0.43	0.40
LC+PDG+DFG+Attention	0.36	0.62	0.46

Notes: LC: Local Context. PDG: Program Dependency Graph. DFG: Data Flow Graph. Attention: Multi-Head Attention.

As shown in Table 3.8, we build 8 variants of our approach with different factors and their combinations. We analyze our results in Table 3.8 as follows:

From *LC* in Table 3.8, we can see that by using only local contexts, i.e., context in individual Abstract Syntax Tree, to model source code, our model can achieve a high recall of 0.75.

To study the impact of the PDG, we compare the results obtained from two variants: *LC* and *LC+PDG*. The results show that adding the PDG as a context can increase Precision and F-score relatively by 52.6% and 26.7%, but decrease Recall from 0.75 to 0.54 (i.e. 28%). This is because the PDG puts a stricter condition on source code similarity.

To study the impact of DFG, we compare the results from two variants: *LC* and *LC+DFG*. The results show that adding the contexts in DFG can increase Precision

and F-score by 36.8% and 13.3%, but decrease Recall from 0.75 to 0.51 (i.e., 32.0%). Overall, adding global contexts can improve Precision, but hurts Recall, which reduce the false positives. However, the overall F-score is improved using either of the two graphs. From the above results, we can see that the PDG can contribute more than DFG. This is reasonable because the PDG contains richer information than DFG.

To study the impact of Multi-Head Attention, we compare the results from the variants *LC* and *LC+Attention*. We can see that Multi-Head Attention cannot help much if we consider only the paths within the method's body because the attention is aimed to put weights on the global context via the PDG and DFG. However, if we compare *LC+PDG* and *LC+PDG+Attention*, we can see that Recall is improved from 0.54 to 0.71. Similar trends can be observed when we compare *LC+DFG* and *LC+DFG+Attention*, or *LC+PDG+DFG* and *LC+PDG+DFG+Attention*. This implies that **Multi-Head Attention contributes much in term of improving Recall** because it helps better ranking of the buggy methods in the resulting list.

To compare *LC+PDG+DFG* with *LC+PDG* and with *LC+DFG*, we can see that both of the graphs in the global context contributes positively on our model's accuracy. When we put together local context in LC, global context in the PDG and DFG, and attention, our model achieves the highest accuracy in all metrics.

Local context enables a high recall. Global context in the PDG and DFG improves much in Precision and reduces false positive rates due to the stricter similarity condition with the use of the PDG and DFG.

However, it hurts Recall. To make up for such reduction in Recall, Multi-Head Attention mechanism emphasizes on the buggy paths, and helps collect more potential buggy methods, and push the buggy methods to be ranked higher in the resulting list. Thus, **Multi-Head Attention mechanism helps our model make up for the reduction of Recall. As a result, F-score of our model is improved.**

3.2.8 Discussion and implications

3.2.8.1 In-Depth case studies. Let us present in-depth case studies to understand why our attention neural network-based approach using local and global contexts for learning code representations achieves better results than other approaches. Let us illustrate via the following case studies.

Method 1.

```
-public void print(Doc doc, int copies, boolean sendToPrinter, String
    mimeType, String jobName) throws PrintException {
+public void print(Doc doc, boolean sendToPrinter, String mimeType, String
    jobName) throws PrintException {
    ...
}
```

Method 2.

```
private void print(InputStream body, String jobName) throws PrintException {
    if (printerOperations.getPrintService().isDocFlavorSupported(
        printerOperations.getFlavor())) {
        PrintDocument printDoc = new PrintDocument(body, printerOperations.
            getFlavor());
-       printerOperations.print(printDoc, config.getCopies(), config.
            isSendToPrinter(), config.getMimeType(), jobName);
+       printerOperations.print(printDoc, config.isSendToPrinter(),
            config.getMimeType(), jobName);
        }
    }
```

Figure 3.12 Bug Detection: Case study 1.
Notes: The code fixes are from *Camel version* 2.20.0 for the bug *Camel* – 12228. The errors are marked in red. For simplicity purpose and page limitation, we only show the key lines of fixes.

Case Study 1. This case study shows a typical example of a bug involving multiple interdependent methods. Figure 3.12 shows a bug-fix example involving

two methods of the project *Camel*, for the bug with an id $Camel - 12228$. The bug report states that Method 1 *print(Doc doc, int copies, boolean sendToPrinter, String mineType, String jobName)* has a bug causing *"print command fails in case of multiple copies"*, as it requires to cancel the loop of print and reduce one parameter *int copies*. The second method *print(InputStream body, String jobName)* has a call (line 6) to Method 1. Therefore, to fix bug id $Camel - 12228$, it requires to fix both methods. Both methods are identified as buggy by our model, but not by the baselines, which consider only individual methods.

The capability to detect this type of popular bugs involving multiple methods is due to the way we model the **relations among paths in the ASTs of a project into code representation**. In the process of code representation learning, we use the dependencies of the entities in the PDG and DFG to capture the relations among paths from the ASTs of a project. In this way, when analyzing the contexts in the AST paths and their relations of the above two methods, our model, trained with existing bug knowledge, syntax, and dependencies from the program entities, can learn to decide that both methods are buggy.

Other baselines, such as MAR-miner, Bugram, and DeepBugs, do not consider the relationships among methods and paths from a perspective of a whole version of a project. They often consider methods individually. In the above example, if a model looks only at the second method *print(InputStream body, String jobName)* without analyzing the dependencies among AST paths from both methods, it cannot detect a bug in this method.

Case Study 2. Figure 3.13 shows an example of two real bug fixes on the same method *main*() in two versions 0.2.0 and 0.8.0 of the project named *pig*. The method *main()* in the version 0.2.0 was fixed before. However, the fix (line 3, Method 3 in version 0.3.0) was marked as a bug in the version 0.8.0. By including the previous

Method 3 in version 0.2.0

```
1  public static void main(String args[])
2      ...
3  -   pigContext.getProperties().setProperty("pig.logfile", logFileName);
4  +   if(logFileName != null) {
5  +       log.info("Logging error messages to:  " + logFileName);
6  +   }
7      ...
8  }
```

Method 3 in version 0.8.0

```
1  public static void main(String args[])
2      ...
3  -   if(logFileName != null) {
4  -       log.info("Logging error messages to:  " + logFileName);
5  -   }
6      pigContext.getProperties().setProperty("pig.logfile", (logFileName
7          == null?"": logFileName));
8      configureLog4J(properties, pigContext);
9  +   if(logFileName != null) {
10 +       log.info("Logging error messages to:  " + logFileName);
11 +   }
12     ...
13 }
```

Figure 3.13 Bug Detection: Case study 2.
Notes: The code fixes are from project *pig version* 0.2.0 and *version* 0.8.0 for the bugs, $PIG-695$ and $PIG-1407$. The errors are marked in red and the fixes are highlighted in green. For simplicity purpose and page limitation, we only show the key lines of fixes that affect both methods.

fixes in the version 0.3.0, our approach is able to identify the *main()* in the version 0.8.0 as buggy, while the baselines cannot.

Extracting long paths and using attention model to add weights in previous buggy paths into code representation. Our model adds weights to previous buggy paths and extracts long AST paths to cover each node in the ASTs to make sure that key information on the bug in the methods to be considered, which makes our code representation more *specialized for bug detection*. In the case study 2, we can see that when the method *main*() was fixed in the version 0.2.0, the AST paths covering the line 3 (i.e., the buggy line) will be given a weight and our model automatically learns the value for the weight based on a large number of previous fixes. Once the weight is learned, our model can learn to analyze the same or similar AST paths as buggy with higher possibilities. Thus, our model can classify the method in the version 0.8.0 as buggy based on the bug in the earlier version. In addition, our model uses long paths to cover all contexts, such as nodes and their relations in ASTs. In the above example, the long paths help our model cover the buggy line (line 3 of the method in the version 0.2.0). However, the baselines consider only some portion of AST contexts. For example, *code2vec* considers only the most-frequent AST paths (i.e., no buggy information is considered in code2vec). In case study 2, *code2vec* assigns a weight of 0.3 to the paths covering the buggy line (line-3). However, *code2vec* does not consider the paths with a weight of 0.3 as a top ranked path, so it misses the buggy path. Also, all other baselines, such as *code2vec, Tree-Based LSTM, Code Vectors, and DL Similarity*, do not incorporate buggy information in their code representation learning, which makes them miss important information in bug detection.

Attention models help add weights to buggy paths in learning code representation, thus improve the ranking, leading to improve Recall. *Attention GRU layer* and *Attention Convolutional layer* extract different types of

key information from the AST in a method. Second, we use a powerful multi-head attention model [160] to combine the key information from different attention layers. The baselines only concatenate different vectors into a unified vector without learning, which may contain less information than our approach in code representations.

Learning to detect bugs, rather than memorization. We checked AST path duplication in training and testing data and found that 26.1% of paths in testing are in training. Our model achieves the precision of 39% (i.e., higher that 26.1%), proving that our model is able to learn from data to detect bugs, rather than simply memorization and retrieving from the stored data.

Table 3.9 Bug Detection: Time Consumption.

Time	Ours	DB	BR	NAR	DS	DLS	c2v	TL	CV	FB
Training/Mining Time	654	238	6	2	497	428	125	219	246	N/A
Detection Time	4	2	2	1	2	3	2	2	2	5

Notes: Time Consumption in Minutes of Approaches in Training and Detecting Bugs from a Project under the setting of detecting bugs in unseen versions of a project.DB: DeepBugs. BR: Bugram. NAR: NAR-miner. DS: DeepSim. DLS: Deep Learning Similarity. c2v: code2vec. TL: Tree LSTM. CV: Code Vectors. FB:FindBugs. N/A:not applicable.

3.2.8.2 Time complexity. Table 3.9 shows that all deep learning based approaches take more time to train, which is well expected. The models can be trained off-line, so the detection time is more important. On average, our approach uses 4 minutes to finish detecting bugs in a project. Although our model costs more time in detecting bugs in a project than other baselines except for the FindBugs, there is only 1 or 2 minutes difference between our model and the baselines due to the time complexity of handling graphs. However, our model performs much better than the baselines on detecting bugs. Due to the page limit, we report only the running time of the models in the setting of detecting bugs in unseen versions of a project. The

time complexity evaluated in the cross-project setting is roughly similar as the ones in Table 3.9.

3.2.8.3 Limitations of our approach. Through analysis on the bugs that our model cannot detect, we identify the following limitations:

- *Our approach does not work well on the bugs about parameters in loops.* Our approach examines all of the contexts in paths and their relations, but our AST path-based modeling cannot accurately capture the relations among parameters in a loop due to the limitations of our static analysis approach. Dynamic analysis on execution paths could be useful to complement with our approach.
- *Our approach does not work well on the bugs about the fixes in strings.* Our approach does not analyze the semantics of string literals and variables, so we cannot detect the bugs that is relevant to changes in string literals. We found that NAR-miner performs better on this kind of bugs by generating the negative rules to pick out the buggy words in the strings.

3.2.8.4 Explanation ability. To improve the ability to explain the buggy code in our solution, we could improve our solution in the following two directions in the future:

- Statement-Level bug detection. Code statement-level bug detection is a natural next step of method-level bug detection. In order to detect buggy code lines, on the top of method information, we plan to utilize and incorporate the following information related to a code statement, *cs*: (1) sequential information of characters and tokens in *cs*; (2) *cs*'s relations with other code statements within one code method; and (3) *cs*'s relations with other relevant code statements from other code methods. Based on the above information, we plan to build code representations for code statements and propose deep learning based approaches to classify code statements.
- Fine-Grained bug classification. In this research topic, our focus is to determine whether a method is buggy or non-buggy. In the future, we plan to show the types of bugs associated with a detected code statement or method. To do so, we first plan to study bugs collected in our big dataset and manually create a small well-classified dataset of bugs. Next, we will explore to develop deep learning and active learning based approaches to automatically label more bugs using the small dataset of bugs as a seed. Last, we can develop and train machine learning (including deep learning) models on the built large dataset containing code and bug types to conduct more explainable bug detection.

3.2.9 Threats to validity

We have identified the following threats to the validity:

Implementation of baselines. To compare with existing bug detection approaches, we have re-implemented a learning-based approach, *Bugram* [164], since the *Bugram* code has been removed from the public repository. The source code of other baselines studied in our study is publicly available and we directly use their code in our experiments.

The *Bugram* paper reported a slightly higher precision and F-score than what we reported using our implementation of Bugram in this work. One possible reason is that *Bugram* performs differently on different datasets and some implementation details are not mentioned in their paper, which makes our version of *Bugram* slightly different from the one in the original paper. However, we tried our best to build and tune the Bugram parameters on our dataset and this is the best effort we can make when the code is not publicly available. We tuned our approach and Bugram both on our dataset, which would make it fair for both of our approach and Bugram.

Applying all baselines on our dataset. Some baselines reported better results in their original papers than the ones we obtained in our work. The main reason is that some baselines were not evaluated on Java code. Although we did not compare our tool with the baselines on their datasets, we compare all approaches on our collected dataset and tune them for the best results.

Collecting bug reports. During the bug report collection, we solely rely on the bug metadata that is manually created by the developers. We only download the bug reports with tags "bug" and "resolved or fixed". However, sometimes, a bug report marked as "bug" is not really a bug, but rather a code refactoring, which is commonly well-known problem in bug management. Due to the large amount of bug reports collected, we cannot verify all of them and make sure all of our data is correctly marked, which is common in large-scale data analysis. However, when

evaluating our results, we also manually verify the top ranked 100 bugs to identify the true bugs to try our best to minimize this potential bias in our study.

Verifying true bugs in our qualitative analysis. Following prior studies [80, 89, 40, 164, 13], we manually check if the reported bugs in qualitative analysis are true bugs. Although this part of work is common in studies for bug detection, this process will bring bias to our results since the authors of this research work are not the developers of these projects. Sometimes we may misunderstand the code and come up with the wrong idea of a bug being a true bug.

Selection of programming languages. In this study, we only apply our approach on Java code. Thus, we cannot claim that our approach is generic for all programming languages. We choose Java code because Java is a widely used programming languages with many mature projects. However, the key drivers of our approach outperforming the baselines are general across programming languages: AST paths, PDG, DFG, and attention mechanism. Our methodology is general, no techniques/algorithms on the above extracted programming structures are tied to any programming languages. However, the results might be different for different languages.

3.3 Fault Localization with Code Coverage Representation Learning

3.3.1 Introduction

Finding and fixing software defects is an important process to ensure a high-quality software product. Much time and effort from developers have been spent in that process. To reduce such effort, several *fault localization* (FL) approaches [174] have been proposed to help developers localize the source of a defect (also called a *bug* or *fault*). In the FL problem, given the execution of test cases, an FL tool identifies the set of *suspicious lines of code* with their associated suspiciousness scores [174]. The key input of an FL tool is the *code coverage matrix* in which the rows and columns

correspond to the source code statements and test cases, respectively. Each cell is assigned with the value of 1 if the respective statement is executed in the respective test case, and with the value of 0, otherwise. In recent FL, several researchers also advocate for fault localization at method level [72]. Both kinds of FL are useful for developers in locating and fixing bugs.

Spectrum-Based fault localization (SBFL) approaches [58, 57, 1] take the recorded lines of code that were covered by each of the given test cases, and assigned each line of code a suspiciousness score based on the code coverage matrix. Despite using different formulas to compute that score, the idea is that a line covered more in the failing test cases than in the passing ones is more suspicious than a line executed more in the passing ones. A key drawback of those approaches is that the same score is given to the lines that have been executed in both failing and passing test cases. An example is the statements that are part of a block statement and executed at the same nested level. Another example is the conditions of the condition statements, e.g., *if*, *while*, *do*, and *switch*.

To improve SBFL, *mutation-based fault localization* (MBFL) approaches [103, 118, 119] enhance the code coverage information by modifying a statement with mutation operators, and then collecting code coverages when executing the mutated programs with the test cases. They apply suspiciousness score formulas in the same manner as spectrum-based FL approaches on the code coverage matrix for each original statement and its mutated ones. Despite the improvement, MBFL are not effective for the bugs that require the fixes that are more complex than a mutation (Sub-Section 4.3.2.1).

Machine learning (ML) and *deep learning (DL)* have been used in fault localization. DeepFL [72] computes for each faulty method a vector with +200 scores in which each score is computed via a specific feature, e.g., a spectrum-based or mutation-based formula, or a code complexity metric. Despite its success, the

accuracy of DeepFL is still limited. A reason could be that it uses various calculated scores from different formulas as a proxy to learn the suspiciousness of a faulty element, instead of fully exploiting the code coverage. Some formulas, such as the spectrum- and mutation-based formulas, inherently suffer from the issues as explained earlier with the statements covered by both failing and passing tests.

We propose DeepRL4FL, a fault localization approach for buggy statements/methods that exploits the image classification and pattern recognition capability of the Convolution Neural Network (CNN) [65] to apply on the code coverage (CC) matrix. Instead of summarizing each row in that matrix with a suspiciousness score, we use its full details and enhance it to facilitate the application of the CNN model in *recognizing the key characteristics in the matrix* to discriminate between faulty and non-faulty statements/methods more easily. We order the columns (test cases) of the CC matrix so that the *test cases with the non-zero values on nearby statements are close together*. The next test case shares with the previous one as many executed statements as possible. This puts *the non-zero cells in the matrix close together*. We expect that *the CNN model with its capability to learn the relationships among nearby cells via a filter* will recognize visual characteristic features to discriminate faulty and non-faulty statements/methods.

Inspired by the method in crime scene investigation, we use three sources of information for FL: 1) code coverage matrix with failed test cases (the crime scene and victims), 2) similar buggy code in the history (*usual suspects* who have committed a similar crime in the past), and 3) the statements with data dependencies (related persons), First, the CC matrix for the occurrence of the fault is an analogy of the evidences at the scene. Second, an investigator also makes a connection from the crime scene to *the usual suspects*. This is analogous to the modeling of the code of the faults that have been encountered in the training dataset. The idea is that if the persons (analogous to the code) who have committed the crimes with similar modus

operandi (M.O.) in the past are likely the suspects (code with high suspiciousness) in the current investigation.

Third, in addition to the crime scene, the investigator also looks at the relationships between the victim or the things happening at the scene and other related persons. Thus, in addition to the statement itself, its suspiciousness is viewed taking into account the data dependencies to other statements in execution flows and data flows. The idea is that some statements, even far away from the buggy line, could have impacts or exhibit the consequences of the buggy line when they are data-dependent. Thus, for a test, we first identify the error-exhibiting (EE) line (defined as the line where the program crashed or exhibited an incorrect value(s)/behavior(s)). That is, if the program crashes, the error-exhibiting line is listed. If no crash and an assertion fails, assertion statement is EE line. EE line is usually specified in a test execution. To identify the related statements, from the EE line, we consider the execution order. However, if the statements are in the same block of code (i.e., being executed sequentially), we also consider the data dependencies among them and with the EE line. Finally, all three sources of information are encoded into vector/matrix representations, which are used as input to the CNN model to act as a classifier to decide whether a statement/method as a faulty or not.

We conducted several experiments to evaluate DeepRL4FL on Defects4J benchmark [32]. Our empirical results show that DeepRL4FL locates 245 faults and 71 faults at the method level and the statement level, respectively, using only top-1 candidate (i.e., the first ranked element is faulty). It can improve the top-1 results of the state-of-the-art *statement-level* FL baselines by 317.7%, 273.7%, 173.1%, 195.8%, and 491.7% when comparing with Ochiai [1], Dstar [173], Muse [103], Metallaxis [119], and RBF-Neural-Network-Based FL (RBF) [172], respectively. DeepRL4FL also improves the top-1 results of the existing *method-level* FL baselines, MULTRIC [180], FLUCCS [146], TraPT [73], and DeepFL [72], by 206.3%, 53.1%, 57.1%, and 15.0%,

respectively. Our results show that three sources of information in DeepRL4FL positively contribute to its high accuracy.

We also evaluated DeepRL4FL on ManyBugs [69], a benchmark of C code with 9 projects. The results are consistent with the ones on Java code. DeepRL4FL localizes 27 faulty statements and 98 faulty methods using only top-1 results.

The contributions of this research are listed as follows:

1. Novel code coverage representation. Our representation enables fully exploiting test coverage matrix and take advantage of the CNN model in image recognition to localize faults.

2. DeepRL4FL: Novel DL-based fault localization approach. Test case ordering and three sources of information allows treating FL as a pattern recognition. Without ordering and statement dependencies, the CNN model will not work well.

3. Extensive evaluation. We evaluated DeepRL4FL against the most recent FL models at the statement and method levels, in both within-project and cross-project settings, and for both **C and Java**.

3.3.2 Motivation

3.3.2.1 Motivating example. Figure 3.14 shows a real-world example of a bug in Defect4J [32]. The bug occurs at line 10 in which the length of the string to be built via *StringBuilder* was not set correctly. The developers fixed the bug by modifying lines 10–11 into line 12.

To localize the buggy line, there exist three categories of approaches. The first one is spectrum-based fault localization (SBFL). The key idea in SBFL is that in a test dataset, *a line executed more in the failing test cases than in the passing ones is considered as more suspicious than a line executed more in the passing ones.* A summary of the CC matrix for this bug is shown in Figure 3.15a. The lines 3, 6–7, and 10–11 in Figure 3.14 are executed in both passing and failing test cases, and as a

result, given *the same suspiciousness scores*. Thus, SBFL is ineffective to detect the buggy line 10 and this buggy method.

```
1  public static String join(Object[] array, char separator, int startIndex,
2    int endIndex) {
3          if (array == null) {
4                  return null;
5          }
6          int noOfItems = (endIndex - startIndex);
7          if (noOfItems <= 0) {
8                  return EMPTY;
9          }
10 -       StringBuilder buf = new StringBuilder((array[startIndex]
11          == null ?  16 :  array[startIndex].toString().length())+1);
12 +       StringBuilder buf = new StringBuilder(noOfItems * 16);
13         for (int i = startIndex; i < endIndex; i++) {
14                 if (i > startIndex) {
15                         buf.append(separator);
16                 }
17                 if (array[i] != null) {
18                         buf.append(array[i]);
19                 }
20         }
21         return buf.toString();
22 }
```

Figure 3.14 Bug Detection: Motivating example 1.

The second category is mutation-based fault localization (MBFL). A MBFL approach (e.g., Metallaxis [119]) modifies a statement using mutation operators. After collecting code coverage information for each statement regarding to multiple mutations, it computes the suspiciousness score for each statement using a spectrum-based formula (e.g., Ochiai [1]) on the CC matrix for each original statement and

line-3	○ ...● ...● ...○ ...● ...●
line-4	○ ...○ ...○ ...○ ...○ ...○
line-6	○ ...● ...● ...○ ...● ...●
line-7	○ ...● ...● ...○ ...● ...●
line-8	○ ...○ ...● ...○ ...○ ...●
line-(10-11)	○ ...○ ...● ...○ ...● ...●
line-13	○ ...● ...● ...○ ...○ ...●
line-14	○ ...● ...● ...○ ...○ ...●
line-15	○ ...● ...● ...○ ...○ ...●
line-17	○ ...● ...● ...○ ...○ ...●
line-18	○ ...● ...● ...○ ...○ ...●
line-21	○ ...● ...● ...○ ...○ ...●

(a)

	t_9 t_{33} ...
line-3	● ● ● ● ○ ... ○
line-4	○ ○ ○ ○ ○ ... ○
line-6	● ● ● ● ○ ... ○
line-7	● ● ● ● ○ ... ○
line-8	○ ○ ● ● ○ ... ○
line-(10-11)	● ⋆ ● ● ○ ... ○
line-13	⋆ ● ● ● ○ ... ○
line-14	● ○ ● ● ○ ... ○
line-15	● ○ ● ● ○ ... ○
line-17	● ○ ● ● ○ ... ○
line-18	● ○ ● ● ○ ... ○
line-21	● ○ ● ● ○ ... ○

(b)

Figure 3.15 Bug Detection: Code coverage for Figure 3.14.
Notes: ○, ●, ⋆ for 0,1,-1

for its mutated ones. However, the fix for the buggy line 10 requires more complex code transformations than a mutation. Thus, an MBFL approach cannot detect this buggy line and buggy method.

The third category is deep learning and ML-based FL approaches [172, 72]. Specifically, Wong *el al.* [172] use a backpropagation neural network on code coverage for each statement. Since the lines 3, 6–7, and 10–11 are executed in both passing and failing test cases, the model cannot learn to distinguish them to detect the buggy line 10. DeepFL [72], uses multilayer perceptron (MLP) on a matrix in which each row corresponds to a statement, while each column is a suspiciousness score computed by a formula, or a code complexity metric. In our experiment, DeepFL could not detect the buggy line 10 in this method. Despite combining several scores, the aforementioned lines are given the same suspiciousness scores by each spectrum-based formula.

Observation 1. The state-of-the-art spectrum-based [93, 58], mutation-based [103, 118, 119], and deep learning-based [72] FL approaches do not consider the full details of the CC matrix. Instead, they summarize each statement/row with a suspiciousness score, limiting their capabilities.

To address that, we aim to exploit the full details of the CC matrix via the use of the CNN model [65], which has been shown to be effective in image pattern recognition. However, there is a challenge: if we do not enforce an order on the test cases (columns), we might end up with a CC matrix with the dark cells (the values of 1) that are far apart (Figure 3.15a). Note that *the CNN model is effective to learn the relationships among the nearby cells in a matrix with its small sliding window (filter)* [65]. Thus, *we need to enforce an order on the test cases, i.e., the columns of the CC matrix so that the values of 1 on the same or nearby rows get to be close to one another*. For example, if we enforce an order with the mentioned strategy (we will explain the detailed algorithm later) for the running example, we will have the matrix in Figure 3.15b. That is, the results for the test cases 1, 142, 190, and 235 in the test dataset of Defect4J for this example are shown in the leftmost columns. We expect that *the CNN model with its sliding window is more effective in the resulting matrix after the ordering due to the nearby dark cells on the left side*. The empirical study on the impact of such ordering will be explained in Sub-Section 3.3.8.

Let us consider another example in Figure 3.16. The bug occurs at line 5 and is fixed in line 6. The program fails in two test cases: 1) x=5, y=0, z=1, and 2) x=7, y=1, z=9. In this example, the lines 2, 3, 4, 5, 15, and 16 are all executed in both passing and failing test cases. Thus, the spectrum-based, mutation-based approaches, and DeepFL give them the same suspiciousness scores, and do not detect the buggy line 5 and this buggy method. The line 16 returns the unexpected results for the two failing test cases. In fact, the spectrum-based and mutation-based approaches locate line 16 as the buggy line. However, the actual error occurs at line 5, steering

```
1  public int Compute(int x, int y, int z){
2           int i = x + 1;
3           int j = x + y;
4           int m = 5;
5  -        if (i < y + 4)
6  +        if (i < y + 7)
7                   if (j > 5 & z > j){
8                           m = m + z;
9                   } else {
10                          m = m + j;
11                  }
12          } else {
13                  m = m + i;
14          }
15          i = m + 1;
16          return m;
17 }
```

Figure 3.16 Bug Detection: Motivating example 2.

the execution to the incorrect branch of the *if* statement. This implies that *while the source of the bug is at line 5, the error exhibits at line 16, which is far apart from line 5, yet has a dependency with it.* However, the line 15, immediate preceding of line 16 does not contribute to the incorrect result at line 16.

Observation 2. We observe that the line that exhibits erroneous behavior (e.g., line 16) might not be the buggy line (line 5). However, the buggy line 5 has a dependency with the line 16. Thus, *identifying the key line exhibiting the erroneous behavior is crucial for FL.* We also observe that in fact, the lines with program dependencies with one another are more valuable in helping localize the buggy line than the lines without such dependencies. Thus, *while considering the execution order of statements, an FL approach should consider the statements with program dependencies as well.*

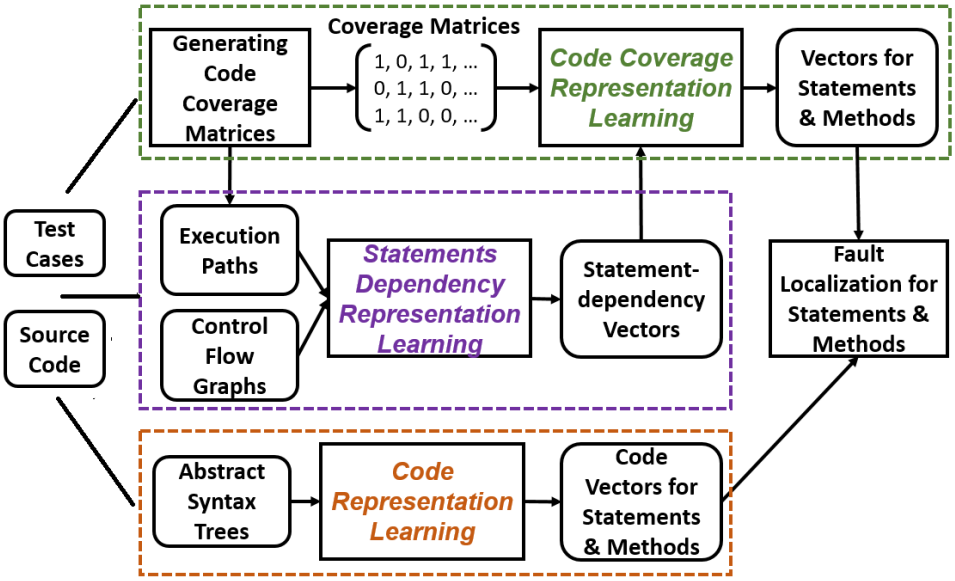


Figure 3.17 Bug Detection: DeepRL4FL's architecture.

3.3.3 Approach overview

Inspired by the crime scene investigation method, we explore three aforementioned sources of information. Correspondingly, DeepRL4FL has three representation learning processes: code coverage representation learning (crime scene), statements dependency representation learning (relations), and source code representation learning (usual suspects) (Figure 3.17).

Step 1: Code Coverage Representation Learning

This learning is dedicated to the "crime scene" analysis of the bug. This process has two parts. First, to help the CNN model recognize the patterns, we take *the given (un-ordered) set of test cases* and *perform an ordering algorithm* to arrange the columns of the CC matrix. The strategy of ordering is to enable the values of 1 to be closer to form darker spots in the left side of the matrix, expecting that the CNN model can work effectively to recognize nearby cells to distinguish the buggy and non-buggy statements (see empirical results in our evaluation).

Second, we also perform the analysis on the output of tests cases to locate the *error-exhibiting (EE)* lines (Observation 2). If the execution of a test crashes, the line

information is always available. Even if there is no crash, the test fails, the program often explicitly lists the lines of code that exhibit the incorrect results/behaviors. We use such information to locate the EE line in the buggy source code corresponding to each test case. The cells in the matrix corresponding to the EE lines of test cases will be marked with -1 values (see the stars in Figure 3.15b). The columns with the values of -1 indicate that the corresponding test cases are the *failing* ones, while the columns with the values of 1 and 0 represent the *passing* ones. The values of 1 and 0 are for the execution or non-execution. The resulting matrix is called the *enhanced CC matrix* (ECC).

Step 2: Dependencies Representation Learning

The suspiciousness of a statement is seen taking into account the data dependencies to other statements in the execution flows and data flows, in addition to the statement itself (Observation 2). Specifically, we consider the execution order. However, if the statements are executed sequentially in the same nested level as part of a block statement, we also *consider the data dependencies among those statements*. Additionally encoding the statements with such dependencies has the same effect as putting together the rows corresponding to the dependent statements in the CC matrix. In our example, in addition to the entire matrix in Figure 3.16, we also encode the data dependencies among statements (i.e., in the same spirit with the case of putting closer the rows 2, 3, 4, 5, 13, 15, and 16), and feed them into the CNN model. In our tool, we collect execution paths and data flow graph for each test case.

Step 3: Source Code Representation Learning

For each buggy code in the training data, esent the code structure by the long paths that are adapted from a prior work [7, 77]. A long path is a path that starts from a leaf node, ends at another leaf node, and passes through the root node of the AST. The AST structure can be captured and represented via the paths with certain lengths across the AST nodes [7]. After this, we have the vectors for the buggy code.

Finally, all the representation vectors are used as the inputs of the CNN model, which is part of the FL module in Figure 3.17.

3.3.4 Step 1: Code coverage representation learning

3.3.4.1 Generating code coverage matrices. Following prior studies [1, 3, 83], we obtain a code coverage matrix for each method of a given project and error messages of failing test cases using GZoltar [42], a tool for code coverage analysis. We further modify GZolter to record the actual execution path of statements within a method during the execution of a test case. For example, for the method in Figure 3.14, the execution path of running the first selected test case is $line\_3 \Rightarrow line\_6 \Rightarrow line\_7 \Rightarrow line\_(10-11) \Rightarrow line\_13 \Rightarrow \underbrace{line\_14 \Rightarrow line\_15 \Rightarrow line\_17 \Rightarrow line\_18 \Rightarrow ...}_{Statements\ repeated\ in\ the\ for\ loop} \Rightarrow line\_21$.

We also use mutation to generate more coverage information. First, we apply the same mutators as in DeepFL [72] to mutate each statement within a method using the mutation tool PIT-1.1.5 [124]. To generate a mutation-based matrix, we apply one mutator to mutate a statement of a method using GZolter. Thus, given n mutators that can be applicable to a statement, we generate n new versions of a method. Given a method having m statements, we generate $n * m$ matrices for the method. We refer the mutation-generated $n * m$ matrices as **mutation-based matrices** and for clarification, we refer the non-mutator-generated matrix as the **spectrum-based matrix**.

3.3.4.2 Identifying error-exhibiting lines. A cell in the CC matrix can have three values: {1,0,-1}. While the values of 1 and 0 indicate passing, those of -1 indicate failing. We obtain *-1* for an error-exhibiting statement or crashed statement from the error messages of failing test cases. An error message shows the names of classes, methods, and line numbers exhibiting an error. We directly use the line numbers, method and class names to assign *-1s* to the statements in the matrix. Figure 3.18

```
java.lang.IllegalArgumentException: Color parameter outside of expected range: Red Green Blue
    at java.awt.Color.testColorValueRange(Color.java:310)
    at java.awt.Color.<init>(Color.java:395)
    at java.awt.Color.<init>(Color.java:369)
    at org.jfree.chart.renderer.GrayPaintScale.getPaint(GrayPaintScale.java:128)
    at org.jfree.chart.renderer.junit.GrayPaintScaleTests.testGetPaint(GrayPaintScaleTests.java:107)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at junit.framework.TestCase.runTest(TestCase.java:176)
    at junit.framework.TestCase.runBare(TestCase.java:141)
    at junit.framework.TestResult$1.protect(TestResult.java:122)
    ……
```

Figure 3.18 Bug Detection: Error message example.

shows an example of the error message containing a stack trace produced by running a test case on the project *Chart* with the bug *Chart-24*. Because the current method under investigation is *getPaint*, our algorithm searches for that method in the stack trace to derive the EE statement at the line 128 of the file *GrayPaintScale.java* (which contains that method). Each failing test case has only one EE statement for the current method under study.

3.3.4.3 Test case ordering algorithm. Our algorithm (Algorithm 1) takes the set of test cases S and enforces an order on S. The strategy is to move the values of 1 and -1 closer to one another in the left side. First, if there exist failing test cases, i.e., test cases with -1s, we select the test case with the value of -1 at the statement appearing latest in the code. We then find the test case that shares the same statement having *-1* with the last selected test case (line 9). That is, we group together the test cases that go through the same statement and also fail. If we do not have such test case, then we repeat the process of looking for another failing test case (i.e., with -1). In Figure 3.15b, the test case 9 is selected as the first one with only one *-1* (marked with a star) at the line 13 (latest statement). We search for the next test case that has a *-1* at the latest. The test case 33 is chosen at the 2nd column.

If we do not have any failing test case left, we select the test case that has the most *1s* (line 13). Next, we select the next test case that shares the most number of the same statements having the values of *1s* with the last selected test case.

Algorithm 1 Test Case Ordering Algorithm.

```
1: function OrderingTestCases(S : testSet)
2:     List = []
3:     while (S <> ∅) do
4:         if HaveTestCasesWithMinusOne(S) then
5:             selT = FindTestCaseWithMinusOneWithHighestIndex(S)
6:             S.remove(selT)
7:             List.append(selT)
8:             while HaveTestCaseSameStmtWithMinusOne(selT, S) do
9:                 selT = FindTestCaseSameStmtWithMinusOne(selT, S)
10:                S.remove(selT)
11:                List.append(selT)
12:        else
13:            selT = FindTestCaseWithMostOne(S)
14:            S.remove(selT)
15:            List.append(selT)
16:            while HaveTestCaseWithSameStmtsWithOne(selT, S) do
17:                selT = FindTestCaseWithMostSameStmtsWithOne(selT, S)
18:                S.remove(selT)
19:                List.append(selT)
20:    return List
```

This helps move the values of 1 closer. We repeat this step to select a new test case compared with the previously selected one until all the test cases were ordered. We stop this step if no test case has the same statements with *1s* as the one in the last selected column. If two test cases are tie, we select the one with the last value of 1 at a statement appearing latter. The rationale is that such a test case covers more statements than the other. If they are still tie, the selection of either of them

will result similar visual effects locally at that row. In brief, in any cases of ties, the visual effects around the statements are similar.

In addition to the spectrum-based matrices, we also apply the same enhancements, *identifying error-exhibiting lines* and *ordering text cases* to mutation-based code coverage matrices.

3.3.5 Step 2: Statements dependency representation

We aim to model the *execution orders* and *data dependencies* among statements of a method under test.

3.3.5.1 Execution order representation. We obtain the execution path (e-path) as each test case was executed. We only consider the relations among statements within a method. Since an e-path is a sequence of statements, we apply Word2Vec [100] on all execution paths of test cases to learn the vectors that encode the relations among statements. Thus, *each statement has a word2vec-generated vector.*

3.3.5.2 Data dependencies representation. Using execution paths is not sufficient due to the following. First, the statements in a loop may repeat multiple times in an e-path, thus, they may dominate vector learning using Word2Vec and weaken the relations between the statements inside and outside of a loop, which is also crucial in FL. Second, interdependent statements might not be nearby in an e-path, yet are useful in detecting the buggy line (Observation 2). To overcome those, we also use a data-flow graph (DFG) for the statements in a method.

We use WALA [162] to generate DFGs where a node represents a statement and an edge represents a data flow between two nodes. If A connects to B, we assign the weight of 1. If there is no edge from B to A, we then create that edge but assign the weight of -1. This makes node2vec [39], a widely used network embedding technique, applicable to our graph. The value of -1 helps distinguish between the artificial edges

and the real ones. After this step, some statements (nodes) with data dependencies have node2vec-generated vectors.

3.3.5.3 Vectors for statements with dependencies. If a statement is not in any block, we use its *word2vec*-generated vector as the final one, otherwise the *node2vec*-generated vector is chosen for the statement. Finally, the output is **a statement-dependency vector for a statement**, modeling the dependencies and/or execution orders among statements.

3.3.5.4 Combining statement dependencies and ECC matrices. To further enrich the ECC matrix (a spectrum-/mutation-based matrix), we incorporate the dependencies among the statements in a method under study into that matrix. In the enhanced matrix, we have the i-th statement (S_i) of a method under test with the test cases, $T = \{T_1, \ldots, T_j \ldots, T_n\}$, where j indicates the j-th test case, $1 \leq j \leq n$, and n is the number of test cases. The statement S_i under a test case T_j has a cell value v_{ij} that can be either {1, 0, or -1}. Thus, the statement S_i can be represented as a vector $\vec{S_i} = \{v_{i1}, \ldots, v_{ij}, \ldots, v_{in}\}$. Each statement ($S_i$) has a statement-dependency vector ($\vec{S_i^{sd}}$). We multiply each v_{ij} with $\vec{S_i^{sd}}$, to obtain $v_{ij} * \vec{S_i^{sd}}$, for each cell of S_i and T_j in the enhanced matrix. Thus, the statement S_i can be represented as a new 2-dimensional vector $\vec{S_i^{2d}} = < v_{i1} * \vec{S_i^{sd}}, \ldots, v_{ij} * \vec{S_i^{sd}}, \ldots, v_{in} * \vec{S_i^{sd}} >$. Any vector $\vec{S_i^{sd}}$ multiplied by a $v_{ij} = 0$ results a vector with all 0s.

A method often has multiple statements {S_1,..., S_i, ... S_m}, where i indicates the i-th statement, $1 \leq i \leq m$, and m is the number of statements. Thus, a method is presented as a 3-D matrix, i.e., a list of 2-D statement vectors.

The same steps are used to enhance and combine statement dependencies into a mutation-based matrix. A statement S_i in a mutation-based matrix is represented as a set of mutated statements and each mutated statement is represented as a 2-D vector. Thus, in this case, the statement S_i is represented as a 3-D vector. After

enhancing the ECC matrix and combining statement-dependencies as explained, we obtain the following:

- In a spectrum-based matrix, a statement is represented as a 2-D vector and a method as a 3-D matrix;
- In mutation-based matrices, a statement is represented as a 3-D matrix and a method as a 4-D matrix.

3.3.5.5 Encoding code coverage matrices with a CNN model. After obtaining those representations for statements/methods, we apply the Convolution Neural Network (CNN) [61] to learn features. We use a typical CNN with the following layers: a convolutional layer, a pooling layer and a fully connected layer. We feed the followings into the CNN model separately to detect buggy a statement/method:
(1) For spectrum-based matrices (SBM), we fed a 2-D vector representing for a statement and a 3-D matrix for a method,
(2) For mutation-based matrices (MBM), we fed a 3-D matrix representing for a statement and a 4-D matrix for a method.

We apply a fully connected layer before CNN on the method in a mutation-based matrix (i.e., represented as a 4-D matrix) to reduce the 4-D into 3-D. The outputs include

1) V_{ss}, 1-D vector for a statement in SBM, 2) V_{sm}, 1-D vector for a method in SBM, 3) V_{ms}, 1-D vector for a statement in MBM, 4) V_{mm}, 1-D vector for a method in MBM.

3.3.6 Step 3: Source code representation learning

Let us explain how we capture the usual suspicious source code via code representation learning.

For a statement, we tokenize it and treat each token in the statement as a word and the entire statement as a sentence. We use *word2vec* [100] on all the statements of a project to compute a token vector for each token. After having the vectors for

all the tokens, for a statement, we have a matrix [Token-Vector$_1$, Token-Vector$_2$, ..., Token-Vector$_m$]. To obtain a unified vector to represent a statement instead of a matrix, we apply a fully connected layer to reduce the matrix into 1-D vector. Thus, we have one vector for each statement.

At the method level, we used two existing code representation learning techniques *code2vec* [7] and ASTNN [188] for a method. In *code2vec*, we use *long paths* over the AST. A long path is a path that starts from a leaf node, ends at another leaf node, and passes through the root node of the AST. The AST structure can be represented via the paths with certain lengths across the AST nodes. Specifically, we regard a long path as a sequence and apply *word2vec* on all long paths of methods to generate a vector representation for each AST node. Now, each path can be represented as an ordered list of node vectors (the order is based on the appearance order of the nodes in a path), and each method can be represented as a bag of paths, i.e., ordered lists of node vectors. Essentially, a method is represented by a matrix. We use a fully connected layer to transform the matrix into 1-D vector for a method.

At the method level, we also used tree-based representation ASTNN [188]. ASTNN splits the AST of a method into small subtrees at the statement level and applies a Recursive Neural Network (RNN) [145] to learn vector representations for statements. The ASTNN exploits the bidirectional Gated Recurrent Unit (GRU) [154] to model the statements using the sequences of sub-tree vectors. After obtaining the long-path-based vector and the tree-based vector for a method, we apply a fully connected layer as the one in CNN [61] to combine these two vectors into one unified vector for a method.

3.3.7 Step 4: Fault localization with CNN model

3.3.7.1 Statement-Level fault localization. After all the previous steps, each statement has 3 vectors:

1) $\vec{V}_{ss}$, a SBM-based statement vector (Sub-Section 3.3.5.5);

2) $\vec{V}_{ms}$, a MBM-based statement vector (Sub-Section 3.3.5.5); and

3) $\vec{V}_{cs}$, a source code-based statement vector (Sub-Section 3.3.6).

The vectors are combined via Hadamard Product [43]:

$$M_s = [len(\vec{V}_{ss}), 1, 1], M_m = [1, len(\vec{V}_{ms}), 1], M_c = [1, 1, len(\vec{V}_{cs})]$$

$$M_{combined} = broadcast(M_s) \circ broadcast(M_m) \circ broadcast(M_c)$$

M is the matrix which is expanded from v by keeping one dimension as v and adding two more dimensions with the size of 1. $broadcast()$ is the operation to copy a dimension into multiple times to expand the matrix to the suitable size for Hadamard product. The rationale is that all three vectors from three different aspects should be fully integrated. The resulting matrix is of the size $[len(\vec{V}_{ss}), len(\vec{V}_{ms}), len(\vec{V}_{cs})]$. Next, we use the trained CNN model with a softmax on the matrix to classify a statement into faulty or non-faulty. The output of the softmax is standardized to be between 0 to 1. To train the model, the same combined matrix for a statement is used at the input layer and the corresponding classification (faulty or not) is used at the output layer of the CNN model.

3.3.7.2 Method-Level fault localization. Similar to statement-level FL, each method has three vectors:

1) $\vec{V}_{sm}$, a SBM-based method vector (Sub-Section 3.3.5.5);

2) $\vec{V}_{mm}$, a MBM-based method vector (Sub-Section 3.3.5.5); and

3) $\vec{V}_{cm}$, a source code-based method vector (Sub-Section 3.3.6).

Moreover, we also consider the similarity between the source code and the error messages of the failing test cases as in DeepFL [72]. We first collect 3 types of information from failed tests, including the name of the failed tests, the source code of the failed tests and the complete failure message (including exception type, message, and stacktrace). Second, we collect 5 types of information from source code, including the full qualified name of the method, accessed classes, method invocations, used

variables, and comments. For each combination, we calculate the similarity score between each information from the failed tests and each from the source code using the popular TF-IDF method [72]. We generate 15 similarity scores as 15 features for a method. Thus, a method also has the fourth vector, $\vec{V}_m^{sim}$ with 15 features.

For fault localization, we combine the above method vectors into a matrix by using the Hadamard product as in Sub-Section 3.3.7.1, then use the trained CNN model with a softmax to classify a method into faulty or non-faulty. We train the model in the same manner as FL at the statement level.

3.3.8 Empirical evaluation

3.3.8.1 Research questions. We seek to answer the following research questions:

RQ1. Statement-Level FL Comparison. How well does our tool perform compared with the state-of-the-art *statement-level* FL approaches?

RQ2. Method-Level FL Comparison. How well does our tool perform compared with the state-of-the-art *method-level* FL approaches?

RQ3. Impact Analysis of Different Matrix Enhancing Techniques. How do those techniques including test case ordering, and statements dependency affect the accuracy?

RQ4. Impact Analysis of Different Representations Learning. How do different types of information affect the accuracy?

RQ5. Cross-Project Analysis. How does DeepRL4FL perform in the cross-project setting for FL?

RQ6. Performance on C Code. How does DeepRL4FL perform in C projects for FL?

3.3.8.2 Experimental methodology.

3.3.8.2.1 Data set. We conduct our study on the well-known benchmark, Defects4J [32]. We use all of the 6 projects in Defects4J V1.2.0 with 395 real bugs

Table 3.10 Bug Detection: Defects4J Dataset.

Identifier	Project name	# of bugs
Chart	JFreeChart	26
Closure	Closure compiler	133
Lang	Apache commons-lang	65
Math	Apache commons-math	106
Mockito	Mockito	38
Time	Joda-Time	27

(Table 3.10). In the dataset, each bug contains a buggy version of the project that includes a large number of training instances (i.e., method/statements). For example, the project Math has 140,000+ training instances. To reduce the influence of the overfitting problem, we applied L2 regularization and added dropout layers.

3.3.8.2.2 Experiment metrics. Following prior studies [72, 73], we use the following metrics to evaluate an FL model:

Recall at Top-K: is the number of faults with at least one faulty statement that is correctly predicted in the ranked list of K statements. We report Top-1, Top-3, and Top-5.

Mean Average Rank (MAR): We compute the average rank of all of the faulty elements for each fault. MAR of each project is the mean of the average rank of all of its faults.

Mean First Rank (MFR): For a fault with multiple faulty elements (methods/statements), locating the first one is critical since the others may be located after that. MFR of each project is the mean of the first faulty element's rank for each fault.

3.3.8.2.3 Experiment setup and procedure.

RQ1: Statement-Level Fault Localization Comparison.

Baselines. We compare DeepRL4FL with the following statement-level FL approaches:

- Two spectrum-based fault localization (SBFL) techniques: **Ochiai** [1] and **Dstar** [173];
- Two recent mutation-based fault localization (MBFL) techniques: **MUSE** [103] and **Metallaxis** [119];
- Two deep-learning based FL approaches: **RBF Neural Network (RBF)** [172] and **DeepFL** [72]. DeepFL [72] works at the *method level* with several features. For comparison, in this RQ1 for the statement level, we can only use DeepFL's spectrum- and mutation-based features applicable to detect faulty statements.

Following prior FL work [10, 73, 72] using Defects4J, we used the setting of leave-one-out cross validation on the faults for each individual project (i.e., within-project setting). Specifically, we use one bug (i.e., with one buggy statement or method) as testing and the remaining bugs in a project for training. This setting provides sufficient training data for the models because a project contains a large number of buggy statements. We performed the cross-project setting in RQ4.

Tuning DeepRL4FL and the baselines. We tuned our model with the following key hyper-parameters to obtain the best performance: (1) Epoch size (i.e., 100, 200, 300); (2) Batch size (i.e., 64, 128, 256); (3) Learning rate (i.e., 0.001, 0.003, 0.005, 0.010); (4) Vector length of word representation and its output (i.e., 150, 200, 250, 300); (5) Convolutional core size (i.e., $3x3$, $5x5$, $7x7$, $9x9$, $11x11$); (6) The number of convolutional core (i.e., 3, 5, 7, 9, 11).

As for word2vec, for a method, we consider all tokens in the source code order as a sentence. We tune the following hyper-parameters for DeepFL (using only the features relevant to statements): Epoch number (5, 10, 15, ..., 60), Loss Functions (softmax, pairwise), and learning rate (0.001, 0.005, 0.010).

RQ2: Method-Level Fault Localization Comparison.

Baselines: We also compare our approach with the following state-of-the-art approaches that localize faulty methods.

MULTRIC [180] is a learning-based approach to combine different spectrum-based ranking techniques using learning-to-rank for effective fault localization.

FLUCCS [146] is a learn-to-rank based technique using spectrum-based scores and change metrics (e.g., code churn and complexity metrics) to rank program elements.

TraPT [73] is a learn-to-rank technique to combine spectrum-based and mutation-based fault localization.

DeepFL [72] is a DL-based model to learn the existing/latent features from multiple aspects of test cases and programs. We used all the features of DeepFL in this method-level study.

Tuning DeepRL4FL and the baselines. Similar to RQ1, we perform our experiments using leave-one-out cross validation on the faults for each project. We use the same settings in RQ1 to train our model. Note that in DeepFL paper [72], **DeepFL**, **MULTRIC**, **FLUCCS**, and **TraPT** have been evaluated using leave-one-out cross validation and other settings on the same data set of Defects4J V1.2.0. DeepRL4FL is also evaluated on Defects4J V1.2.0 using the same settings and procedure as DeepFL. Thus, we used the result on the numbers of detected bugs reported in DeepFL [72] for those models.

RQ3: Impact Analysis of Different Matrix Enhancing Techniques.

We aim to evaluate the impact of our CC matrix enhancing techniques on performance. We evaluate the following (1) test case ordering algorithm utilizing the EE lines (**Order**); (2) adding statements' dependencies (**StatDep**). We first built a base model by using only the spectrum- and mutation-based matrices in DeepRL4FL (without using the above techniques), then apply the above techniques on the matrices to build two variants of DeepRL4FL: {*Base + Order*}, and {*Base + Order + StatDep*

(DeepRL4FL)}. We train each variant using the same settings as in RQ1. Due to space limit, we show only the analysis results obtained in the within-project setting at method-level FL.

RQ4: Impact Analysis of Learning Representations.

We have the following representation learning schemes: learning on the new enhanced spectrum-based CC matrix (**NewSpecMatrix**) and another learning on the new enhanced mutation-based CC matrix (**NewMutMatrix**). We also conducted source code representation learning (**CodeRep**) and textual similarity learning between source code and error messages in failing tests (**TextSim**). To test the impact of those representation learning schemes, we first built a base model using only NewSpecMatrix, then another three variants: {*NewSpecMatrix + NewMutMatrix*}, {*NewSpecMatrix + NewMutMatrix + CodeRep*}, and {*NewSpecMatrix + NewMutMatrix + CodeRep+TextSim*}. We trained each variant using the same settings as in RQ1. Due to the space limit, we show only the results for the within-project setting at method-level FL.

RQ5: Cross-Project Analysis.

We also setup the cross-project scenario: testing one bug in a project, but training a model on all of the bugs of other projects. For a project, we test every bug and sum up the total number of bugs in the project that can be localized in the cross-project scenario.

RQ6: DeepRL4FL's Fault Localization Performance on C Code.

We also evaluated DeepRL4FL on C projects from the benchmark dataset, ManyBugs [69, 95], with 185 bugs from 9 projects. We used the same model in RQ1 for statement-level FL and the model in RQ2 for method-level FL.

3.3.8.3 Experimental results.

Table 3.11 Bug Detection: RQ1. Results of Comparative Study at Statement-Level Fault Localization.

Approach	Top1	Top3	Top5	P%	MFR	MAR
Ochiai	17	88	115	4.3%	54.29	71.32
Dstar	19	92	115	4.8%	48.67	69.51
MUSE	26	47	63	6.6%	36.34	48.73
Metallaxis	24	81	108	6.1%	34.59	49.21
RBF	12	37	52	3.0%	22.54	57.47
DeepFL	39	114	129	9.9%	24.09	31.28
DeepRL4FL	**71**	**128**	**142**	**18.0%**	**20.32**	**28.63**

Notes: P% = |Top-1|/{395 Bugs}

3.3.8.3.1 RQ1-Results (Statement-Level fault localization comparison). As seen in Table 3.11, DeepRL4FL improves over the state-of-the-art statement-level FL baselines.

Specifically, DeepRL4FL improves Recall at Top-1 by 317.6%, 273.7%, 173.1%, 195.8%, 491.7%, and 82.1% in comparison with Ochiai, Dstar, Muse, Metallaxis, RBF, and DeepFL.

We examined the results and report the following. The key reason for the spectrum-based FL approaches fail to localize the buggy statements is that they give the same suspiciousness score to the statements at the same nested level. For the mutation-based FL approaches, the key reason for not being able to localize the buggy statements/methods is that the fix requires a more sophisticated change than a mutation.

3.3.8.3.2 RQ2-Results (Method-Level fault localization comparison). Table 3.12 shows that DeepRL4FL can outperform all baselines on the method level fault localization.

Table 3.12 Bug Detection: RQ2. Results of Comparative Study at Method-Level Fault Localization.

Approach	Top1	Top3	Top5	P%	MFR	MAR
MULTRIC	80	163	195	20.3%	37.71	43.68
FLUCCS	160	249	275	40.5%	16.53	21.53
TraPT	156	249	281	39.5%	9.94	12.70
DeepFL	213	282	305	53.9%	6.63	**8.27**
DeepRL4FL	**245**	**294**	**311**	**62.0%**	**5.94**	8.57

Notes: P% = |Top-1|/{395 Bugs}

DeepRL4FL can improve Recall at Top-1 results by 206.3%, 53.1%, 57.1%, and 15.0% in comparison with MULTRIC, FLUCCS, TraPT, and DeepFL, respectively. DeepRL4FL's MAR value is slightly higher (i.e., 3.6% higher) than DeepFL's. On average, DeepRL4FL ranks the correct elements higher than DeepFL, as its MFR value is lower (i.e., 10.4% lower).

The spectrum-based and mutation-based FL approaches fall short of DeepFL and DeepRL4FL. A key reason is that they consider only dynamic information in test cases, while DeepFL and our model use both static and dynamic information. In comparison with DeepFL, we further analyzed the bugs that our tool can locate, but DeepFL missed. We found that the mean first rank of a buggy method in the ranking lists of potential buggy methods returned by DeepFL is 7.08. Without the ordering and statement dependency in our model, the mean first rank is 6.84. With only ordering in our model, the mean first rank is 2.82. With the only dependency in our model, the mean first rank is 4.45. With both ordering and dependency, our model can locate the bugs that DeepFL missed.

Table 3.13 Bug Detection: RQ3. Ordering (Order) and Adding Dependencies (StatDep) in Method-Level FL.

Variants	Top1	P%	MFR	MAR
Base (DeepRL4FL *w/o* Order,StatDep)	173	43.8%	8.23	10.27
Base + Order	226	57.2%	6.57	8.97
Base + Order + StatDep (DeepRL4FL)	245	62.0%	5.94	8.57

Notes: P% = |Top-1|/{395 Bugs}

3.3.8.3.3 RQ3-Results (Impact analysis of different matrix enhancing techniques). Table 3.13 shows that our proposed matrix enhancing techniques positively contribute to DeepRL4FL.

Specifically, comparing {*Base*} with {*Base + Order*}, ordering the test cases can improve every metric. *Order* can help localize 53 more bugs (13.4%) using Top-1. It helps improve MFR and MAR by 20.1% and 12.7%, respectively, showing that ordering can help DeepRL4FL push the faulty methods higher in the ranked list.

Comparing {*Base + Order*} with {*Base + Order+StatDep*}, the results show that modeling dependencies into matrices is useful to improve the performance of DeepRL4FL. StatDep can improve 8.4%, 9.6%, and 4.5% on Top1, MFR, and MAR.

3.3.8.3.4 RQ4-Results (Impact analysis of learning representations). Table 3.14 shows that our representation learning has positive contributions on the results.

Comparing {*Base*} with {*Base + NewMutMatrix*}, we can see that the representation learning on mutation-based matrices can help locate 23 more bugs using Top-1 and improve MFR and MAR by 8.2% and 3.5%. By adding the source code representation learning into the model, we improved DeepRL4FL to localize 9 more bugs and gain an increase on MFR and MAR by 7.9% and 4.0%, respectively. Furthermore, TextSim also positively contributes to our model.

Table 3.14 Bug Detection: RQ4. Results of Learning Representations in Method-Level FL.

Variants	Top1	P%	MFR	MAR
Base (NewSpecMatrix)	189	47.8%	8.09	9.91
Base + NewMutMatrix	212	53.7%	7.43	9.56
Base + NewMutMatrix + CodeRep	221	55.9%	6.84	9.18
Base + NewMutMatrix + CodeRep + TextSim (DeepRL4FL)	245	62.0%	5.94	8.57

Notes: P% = |Top-1|/{395 Bugs}

3.3.8.3.5 RQ5-Results (Cross-Project analysis). As seen in Table 3.15, DeepRL4FL achieves better results in the within-project setting than in the cross-project one.

Table 3.15 Bug Detection: RQ5. Cross-Project Versus Within-Project.

Projects	Cross-Project				Within-Project			
	Top1	P%	MFR	MAR	Top1	P%	MFR	MAR
Chart	13	50.0%	3.15	5.62	15	57.7%	2.85	4.65
Time	13	48.1%	9.78	14.70	14	51.9%	8.41	13.33
Math	61	57.5%	3.81	4.88	64	60.4%	2.93	4.83
Closure	71	53.4%	11.70	15.23	73	54.9%	9.38	12.37
Mockito	12	31.6%	11.42	16.42	14	36.8%	9.39	15.11
Lang	47	72.3%	2.13	2.49	50	76.9%	1.97	2.31

As seen in Table 3.15, DeepRL4FL achieves better results in the within-project setting than in the cross-project one. This is expected as the training and testing data is from the same project in the within-project setting and a model may see similar faults.

In the cross-project setting, DeepRL4FL correctly detects 217 bugs at the Top1 in comparison with the best result (207 bugs) from the baselines. In the within-project scenario, DeepRL4FL correctly detects 230 bugs at the Top1 in comparison with 80/160/156/213 bugs (not shown) from the baseline models MULTRIC/FLUCCS/TraPT/DeepFL, respectively.

Training time. On average, training time is 350-380 mins per project in the cross-project scenario, and 120-130 mins per project in the within-project scenario. Once the model is trained, the prediction time per fault is 2-7 seconds in both the cross and within project scenarios.

3.3.8.3.6 RQ6-Results (Performance on C code). As seen in Table 3.16, DeepRL4FL can localize 27 faulty statements and 98 faulty methods with only top-1 statements and methods.

The empirical results show that the performance of DeepRL4FL on the C projects is consistent with the one on the Java projects.

Table 3.16 Bug Detection: RQ6. ManyBugs (C Projects) VS. Defects4J (Java Projects).

Level	ManyBugs (C projects)				Defects4J (Java projects)			
	Top1	P%	MFR	MAR	Top1	P%	MFR	MAR
Statement	27	14.6%	25.74	31.33	71	18.0%	20.32	28.63
Method	98	53.0%	6.91	9.89	245	62.0%	5.94	8.57

Notes: P% = |Top-1|/{Total Bugs in Datasets}

Specifically, at the statement level, the percentages of the total C and Java bugs that can be localized are similar, i.e., 14.6% vs. 18.0%, respectively. At the method level, the percentages of the total C and Java bugs that can be localized are also consistent, i.e., 53.0% vs. 62.0%, respectively.

```
1  public void translate(CharSequence input, Writer out) throws IOException {
2      ...
3      int pos = 0;
4      int len = input.length();
5      while (pos < len) {
6          int consumed = translate(input, pos, out);
7          if (consumed == 0) {
8              char[] c = Character.toChars(Character.codePointAt(input, pos));
9              out.write(c);
10             pos+= c.length;
11             continue;
12         }
13         for (int pt = 0; pt < consumed; pt++) {
14 -             pos += Char.charCount(Char.codePointAt(input,pos));
15 +             pos += Char.charCount(Char.codePointAt(input, pt));
16         }
17     }
18 }
```

Figure 3.19 Bug Detection: Case study 1.

3.3.9 Discussion and implications

3.3.9.1 In-Depth case studies. **Case Study 1.** In Figure 3.19, the fault is caused by incorrect variable. To fix it, the variable was changed from *pos* to *pt* at *line 14*. The state-of-art spectrum-based approaches cannot localize this fault because *line 6, line 7, line 13*, and *line 14 have the same score* (They were executed in both passing and failing test cases). For the mutation-based FL approaches, there is none of mutation operators that changes the variable *pos* into *pt* in a method call at the buggy line 14. Thus, they cannot observe the impact of mutations on the code coverage. As a consequence, they cannot locate the buggy line 14.

Figure 3.20 Bug Detection: A feature map produced by CNN for Figure 3.19.

To gain insights on DeepRL4FL, we perform a visualization of a feature map for this case. This is a common technique in image processing with CNN. Note that, during training, CNN learns the values for small sliding windows, called *filters*. The feature maps of a CNN capture the result of applying the filters to an input matrix. That is, at each layer, the feature map is the output of that layer. In image processing, visualizing a feature map for an input helps gain understanding on whether the model detects some part of our desired object and what features the CNN observes. Figure 3.20 shows a feature map for the example in Figure 3.19. We can see that around the lines 6–8 and 13–14, the feature map is visually dark. Without ordering (i.e., a random order of test cases), the feature map does not exhibit such visualization.

To further study the impacts of the ordering and data dependencies, we modified DeepRL4FL in the following settings: 1) No ordering + No dependencies: the buggy line 14 is ranked at 43th; 2) No ordering + dependencies: it is ranked at 29th; 3) ordering + No dependencies: it is ranked at 7th; and 4) ordering + dependencies: it is ranked at the top.

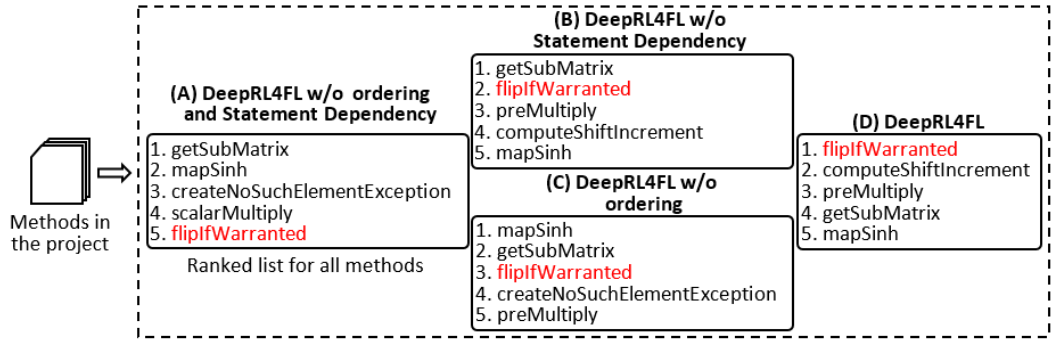


Figure 3.21 Bug Detection: Case study 2.

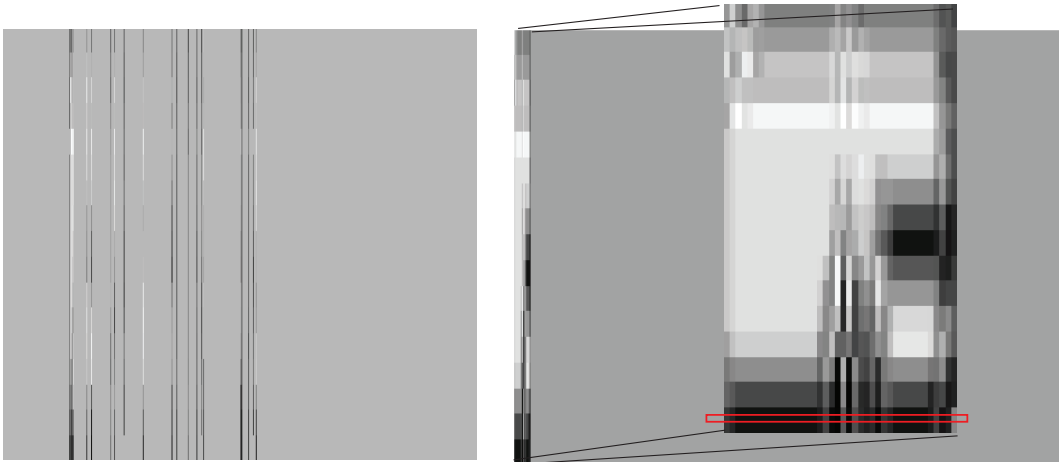


Figure 3.22 Bug Detection: Case study 3.

Case Study 2. The case in Figure 3.21 is the fault that our model detected but DeepFL missed. The (buggy) method *flipIfWarranted* together with the other methods in the same project was fed into four variants of our model. As seen, with the setting where removing both ordering and statement dependency, *flipIfWarranted* is ranked 5th in the list of all methods. For the setting where only ordering is used, it is ranked at 2nd. For the setting where only statement dependency is used, it is ranked 3rd. With both, our model ranks the buggy method *flipIfWarranted* at the 1st position. This analysis shows that ordering test cases and statement dependencies are the key drivers that help our model locate more bugs than DeepFL.

Case Study 3. To further study *the impact of the ordering*, we visualize the feature maps for the 53 bugs that *Order* can detect and *Base* did not. Those are

the cases where ordering helps the FL. Visualizing the feature maps for those inputs allows us to understand what features the CNN detects in both cases of ordering or not. That is, observing the feature maps allows us to see if ordering can help the CNN model learns better the discriminative features in locating the buggy statements. To do so, for each of those bugs, we used the CNN model as part of *Base* and *Order* to produce two feature maps: one corresponds to *Base* (no ordering) and one to *Order*. We then visualized and manually compared those two feature maps as gray-scale images. The CNN model generates 9 feature maps as the output from 9 different convolutional cores.

In all the bugs, we observed the same phenomenon. Let us use an real case as example. Figure 3.22 shows two feature maps for one of those bugs. The ones on the left and on the right are for *Base* (without ordering) and for *Order* (with ordering), respectively. We zoomed out the leftmost columns in the right feature map. The row corresponding to the buggy line is in the red rectangle. With ordering, one of those 9 feature maps has visually darker lines around the buggy statement. In contrast, without ordering, all the feature maps are similar to the one on the left, i.e., do not show any clear visual lines. That is, with ordering, the CNN, which focuses on the relations of neighboring cells, can detect the features along the buggy statement.

3.3.9.2 Limitations. The quality of test cases is important for our approach. If there are only a couple of passing test cases or the crash occurs far apart from the faulty method, DeepRL4FL does not learn useful representation matrix to localize the faults. It does not work well on locating the faults that require statement additions to fix (all of the baselines in this research work do not either). Moreover, it does not work well for short methods, as they provide less statement dependencies. It is also hard for our model to localize the uncommon faults. Because it is DL-based, if there is a very uncommon fault that may not been seen in the training dataset, it will not work correctly.

3.3.10 Threats to validity

We have identified the following threats to the validity:

Baseline implementation. To compare with existing approaches, we implemented Ochiai, Dstar, MUSE, Metallaxis, and RBF-neural-network for statement-level FL. We followed the same approach in [72] to implement MUSE and Metallaxis using PIT-1.1.5. RBF-neural-network approach is built for artificial faults and our real bug dataset cannot match the requirements.

Result generalization. Our comparisons with the baselines were only carried out on the Defects4J dataset, which is a widely used benchmark for FL research. Further validation of the evaluation processes on other datasets should be done in the future.

3.4 Fault Localization to Detect Co-Change Fixing Locations

3.4.1 Introduction

To assist developers in the bug-detecting and fixing process, several approaches have been proposed for *Automated Program Repair* (APR) [68]. A common usage of an APR tool is that one needs to use a *fault localization* (FL) tool [174] to locate the faulty statements that must be fixed, and then uses an APR tool to generate the fixing changes for those detected statements. The input of an FL model is the execution of a test suite, in which some of the test cases are passing or failing ones. Specifically, the key input is the *code coverage matrix* in which the rows and columns correspond to the statements and test cases, respectively. Each cell is assigned with the value of 1 if the statement is executed in the respective test case, and with the value of 0, otherwise. An FL model uses such information to identify the list of *suspicious lines of code* that are ranked based on their associated *suspiciousness scores* [174]. In recent advanced FL, several approaches also support fault localization at method level to locate faulty methods [72, 75].

The FL approaches can be broadly divided into the following categories: *spectrum-based fault localization* (SBFL) [2, 57, 58], *mutation-based fault localization* (MBFL) [103, 118, 119], and *machine learning (ML)* and *deep learning (DL) fault localization* [72, 75]. For SBFL approaches, the key idea is that a line covered more in the failing test cases than in the passing ones is more suspicious than a line executed more in the passing ones. To improve SBFL, MBFL approaches [103, 118, 119] enhance the code coverage matrix by modifying a statement with mutation operators, and collecting code coverage when executing the mutated programs with the test cases. The MBFL approaches apply suspiciousness score formulas in the same manner as in SBFL approaches on the matrix for each original statement and its mutated code. Finally, ML and DL-based FL approaches explore the code coverage matrix and apply different neural network models for fault localization.

Despite their successes, the state-of-the-art FL approaches are still limited in locating all dependent fixing locations that need to be repaired at the same time in the same fix. In practice, there are many bugs that require *dependent changes in the same fix to multiple lines of code in one or multiple hunks of the same or different methods for the program to pass the test cases*. For those bugs, applying the fixing change to individual statements once at a time will not make the program pass the test case after the change to one statement. This capability to detect the fixing locations of the co-changes in a fix for a bug (let us call them *Co-Change (CC) Fixing Locations*) is crucial for an APR tool. Such capability will enable an APR tool to *make the correct and complete changes to fix a bug*.

The state-of-the-art FL approaches do not satisfy that requirement. From the ranked list of suspicious statements returned from an existing FL model, a naive approach to detect CC fixing locations would be to take the top k statements in that list and to consider them as to be fixed together. This solution might be ineffective because the mechanisms used in the state-of-the-art FL approaches have

never considered the co-change nature of those fixes. Our empirical evaluation also confirmed that (Sub-Section 3.4.7.3.1).

Detecting all the CC fixing locations at multiple statements in potentially multiple methods is challenging. A naive solution would be detecting the potential methods that need to be fixed together and then detecting potential statements that need to be changed together in each of those methods. However, doing so will create a confounding effect from the inaccuracy of the detection of the co-fixed methods to that of the co-fixed statements.

We propose FixLocator, a fault localization approach to derive the co-change fixing locations in the same fix for a fault (i.e, multiple faulty statements in possible multiple faulty methods). To avoid the confounding effect in that naive solution, we treat this problem as *dual-task learning* with two dedicated models. First, the *method-level FL* model (*MethFL*) learns the methods that need to be modified in the same fix. Second, the *statement-level FL* model (*StmtFL*) learns the co-fixed statements in the same or different methods. The intuition is that they are closely related, which we refer to as *duality*. *Correct learning for a model can benefit the other and vice versa*. If two statements in two methods are fixed together for a bug, those methods are also co-fixed. If two methods are co-fixed, some of their statements are also co-fixed. Exploring this duality can provide useful constraints to detect CC fixing locations for a bug. Thus, instead of cascading the two models *MethFL* and *StmtFL*, we train them simultaneously with the soft-sharing of the models' parameters to exploit this duality. Specifically, we leverage the cross-stitch units [101] to connect *MethFL* and *StmtFL*. In a cross-stitch unit, the sharing of representations between *MethFL* and *StmtFL* is modeled by learning a linear combination of the input features from the two models. The cross-stitch units enable the *propagation of the impact of MethFL and StmtFL on each other.*

In addition to the new solution in dual-task learning, we utilize a novel feature for this CC fixing location problem: *co-changed statements*, which have never been exploited in FL. The rationale is that the co-changed statements in the past might become the statements that will be fixed together in the future. Finally, since the co-fixed statements are often interdependent, we use Graph-Based Convolution Network (GCN) [81] to integrate different types of program dependencies among statements, e.g., data and control dependencies, execution traces, stack traces, etc. We also encode test coverage and co-changed/co-fixed statements in the graph. The GCN model learns and predicts the bugginess of the statements.

We conducted several experiments to evaluate FixLocator on Defects4J-v2.0 [32]. Our empirical results show that FixLocator improves the baselines, CNN-FL [193], DeepFL [72], DeepRL4FL [75], and DEAR's FL [76] by 16.6%, 16.9%, 9.9%, and 20.6% respectively, in terms of Hit-1 (i.e., the percentage of bugs in which the predicted set overlaps with the oracle set for *at least* one faulty statement), and by 33.6%, 40.3%, 26.5%, and 57.5% in terms of Hit-2 (i.e., the percentage of bugs in which the number of overlapping statements between the predicted and oracle sets is $\geq$2), 43.9%, 46.4%, 28.1%, and 51.9% in terms of Hit-3, respectively. FixLocator also improves those baselines by 32.0%, 38.8%, 20.8%, and 46.1% in terms of Hit-All (i.e., the predicted set exactly matches with the oracle set for a bug).

To evaluate its usefulness in APR, we combined it with the APR tools, DEAR [76] and CURE [54]. We replaced DEAR's FL module with FixLocator for a variant, DEAR^{FixL}. Our result shows that DEAR^{FixL} and FixLocator+CURE improve relatively DEAR and Ochiai+CURE by 10.5% and 42.9% in terms of numbers of fixed bugs.

Through our ablation analysis on the impact of different features and modules of FixLocator, we showed that all designed features/modules have contributed to its high performance. Specifically, the proposed dual-task learning significantly improves

the statement-level FL by up to 12.8% in terms of Hit-1. The designed feature of co-change relations among methods and statements has also positively contributed to FixLocator's high accuracy level.

The contributions of this research topic are listed as follows:

1. **FixLocator: Advancing DL-based Fault Localization** to derive the **co-change fixing locations** (multiple faulty statements) in the same fix for a bug. We treat that problem as ***dual-task learning to propagate the impact*** between the method-level and statement-level FL.

2. **Novel graph-based representation learning with GCN and novel type of features in co-changed statements for FL** enable dual-task learning to derive CC fixing locations.

3. **Extensive empirical evaluation.** We evaluated FixLocator against the recent DL-based FL models to show its accuracy and usefulness in APR.

3.4.2 Motivating example

3.4.2.1 Example and observations. Let us start with a real-world example. Figure 3.23 shows a bug fix in the Defects4J dataset that require multiple interdependent changes to multiple statements in different methods. The bug occurred when the method call to *setTagAsStrict* did not consider the first output in its arguments. Therefore, for fixing, a developer adds a new argument in the method *toSource* at line 18, and uses that argument in the method call *setTagAsStrict (firstOutput,...)* at line 22. Because the method *toSource* at line 17 was changed, the two callers at line 3 of the method *toSource* (line 1) and at line 13 of the method *toSource* (line 11) need to be changed accordingly.

3.4.2.1.1 Observation 1 [Co-Change fixing locations]. In this example, the changes to fix this bug involve multiple faulty statements that are dependent on one another. Fixing only one of the faulty statements will not make the program pass the failing test(s). Fixing individual statements once at a time in the ranked list returned from an existing FL tool will also not make the program pass the tests. For an APR model to work, an FL tool needs to point out all of those faulty statements to be

changed in the same fix. For example, all four faulty statements at lines 17, 21, 3, and 13 need to modified accordingly in the same fix to fix the bug in Figure 3.23.

```
1 public void toSource(final CodeBuilder cb, int inputSeqNum, Node root) {
2     ...
3 -   String code = toSource(root, sourceMap);
4 +   String code = toSource(root, sourceMap, inputSeqNum == 0);
5     if (!code.isEmpty()) {
6         cb.append(code);
7     } ...
8 }
9 //------------------------------------------------------------------------
10 @Override
11 String toSource(Node n) {
12     initCompilerOptionsIfTesting();
13 -   return toSource(n, null);
14 +   return toSource(n, null, true);
15 }
16 //------------------------------------------------------------------------
17 -private String toSource(Node n, SourceMap sourceMap)
18 +private String toSource(Node n, SourceMap sourceMap, boolean firstOutput)
19     ......
20     builder.setSourceMapDetailLevel(options.sourceMapDetailLevel);
21 -   builder.setTagAsStrict(
22 +   builder.setTagAsStrict(firstOutput &&
23         options.getLanguageOut(a) == LanguageMode.ECMASCRIPT5_STRICT);
24     builder.setLineLengthThreshold(options.lineLengthThreshold);
25     ......
26 }
```

Figure 3.23 Bug Detection: Co-Change fixing locations for a fault.

3.4.2.1.2 Observation 2 [Multiple faulty methods]. As seen, this bug requires an APR tool to make changes to multiple statements in three different

methods in the same fix: *toSource(...)* at lines 17 and 21, *toSource(...)* at line 3, and *toSource (...)* at line 13. Thus, it is important for an FL tool to connect and identify these multiple faulty statements in potentially different methods.

Traditional FL approaches [192, 196] using program analysis (PA), e.g., execution flow analysis, are restricted to specific PA techniques, thus, not general to locate all types of CC fixing locations. Spectrum-Based [56, 2], mutation-based [103, 118, 119]), statistic-based [84], and machine learning (ML)-based FL approaches [72, 75] could implicitly learn the program dependencies for FL purpose. However, despite their successes, the non-PA FL approaches *do not support the detection of multiple locations that need to be changed in the same fix for a bug, i.e., Co-Change (CC) Fixing Locations*. The spectrum-based and ML-based FL models return a ranked list of suspicious statements according to the corresponding suspiciousness scores. In this example, the lines 13, 17, 21, and the other lines (*e.g.*, 12, 20 and 24) are executed in the same passing or failing test cases, thus assigned with the same scores by spectrum- and mutation-based FL approaches. A user would not be informed on what lines need to be fixed together. Those non-PA, especially ML-based FL approaches, do not have a mechanism to detect CC fixing locations.

In this work, we aim to advance the level of *deep learning (DL)-based FL approaches to detect CC fixing statements*. However, it is not trivial. A solution of assuming the top-k suspicious statements from a FL tool as CC fixing locations does not work because even being the most suspicious, those statements might not need to be changed in the same fix. In this example, all of the above lines with the same suspiciousness scores would confuse a fixer.

Moreover, another naive solution would be to use a method-level FL tool to detect multiple faulty methods first and then use a statement-level FL tool to detect the statements within each faulty method. As we will show in Sub-Section 3.4.7,

the inaccuracy of the first phase of detecting faulty methods will have a confounding effect on the overall performance in detecting CC fixing statements.

3.4.2.2 Key ideas. We propose FixLocator, an FL approach to locate all the CC fixing locations (i.e., faulty statements) that need to be changed in the same fix for a bug. In designing FixLocator, we have the following key ideas in both new model and new features:

3.4.2.2.1 Key Idea 1 [Dual-Task learning for fault localization]. To avoid the confounding effect in a naive solution of detecting faulty methods first and then detecting faulty statements in those methods, we design an approach that treats this FL problem of detecting dependent CC fixing locations as *dual-task learning* between the method-level and statement-level FL. First, the *method-level FL* model (*MethFL*) aims to learn the methods that need to be modified in the same fix. Second, the *statement-level FL* model (*StmtFL*) aims to learn the co-fixing statements regardless of whether they are in the same or different methods.

Intuitively, *MethFL* and *StmtFL* are related to each other, in which the results of one model can help the other. We refer to this relation as *duality*, which can provide some useful constraints for FixLocator to learn dependent CC fixing locations. We conjecture that the joint training of the two models can improve the performance of both models, when we leverage the constraints of this duality in term of shared representations. For example, if two statements in two different methods m_1 and m_2 were observed to be changed in the same fix, then it should help the model learn that m_1 and m_2 were also changed together to fix the bug. If two methods were observed to be fixed together, then some of their statements were changed in the same fix as well. In our model, we jointly train *MethFL* and *StmtFL* with the soft-sharing of the models' parameters to exploit their relation. Specifically, we use a mechanism, called *cross-stitch unit* [101], to learn a linear combination of the input features from those two models to *enable the propagation of the impact of MethFL and StmtFL on each*

other. We also add an attention mechanism in the two models to help emphasize on the key features.

3.4.2.2.2 Key Idea 2 [Co-Change representation learning in fault localization]. In detecting CC fixing locations, in addition to a new dual-task learning model in key idea 1, we use a new feature: *co-change information* among statements/methods, which has never explored in prior fault localization research. The rationale is that the co-changed statements/methods in the past might become the statements/methods that will be fixed together for a bug in the future. We also encode the co-fixed statements/methods in the same fixes. The co-changed/co-fixed statements/methods in the same commit are used to train the models.

3.4.2.2.3 Key Idea 3 [Graph modeling for dependencies among statements/methods]. The statements/methods that need to be fixed together are interdependent via several dependencies. Thus, we use Graph-Based Convolution Network (GCN) [81] to model different types of dependencies among statements/methods, e.g., data and control dependencies in a program dependence graph (PDG), execution traces, stack traces, etc. We encode the *co-change/co-fix relations* into the graph representations with different types of edges representing different relations. The GCN model enables nodes' and edges' attributes and learns to classify the nodes as buggy or not.

3.4.3 FixLocator: Approach overview

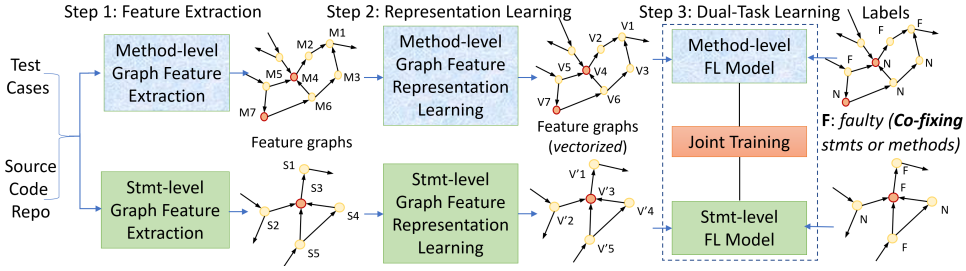


Figure 3.24 Bug Detection: FixLocator training process.

3.4.3.1 Training process. Figure 3.24 summarizes the training process. The input of training includes the passing and failing test cases, and the source code under study. The output includes the trained method-level FL model (detecting co-fixed methods) and the trained statement-level FL model (detecting co-fixed statements). The training process has three main steps: 1) *Feature Extraction*, 2) *Graph-Based Feature Representation Learning*, and 3) *Dual-Task Learning Fault Localization*.

3.4.3.1.1 Step 1. Feature extraction. (Sub-Section 3.4.4). We aim to extract the important features for FL from the test coverage and source code including co-changes. The features are extracted from two levels: statements and methods. At each level, we extract the important *attributes* of statements/methods, as well as the crucial *relations* among them. We use graphs to model those attributes and relations.

For a method m, we collect as its attributes 1) method content: the sequences of the sub-tokens of its code tokens (excluding separators and special tokens), and 2) method structure: the Abstract Syntax Tree (AST) of the method. For the relations among methods, we extract the relations involving in the following:

1. *Execution flow* (the calling relation, i.e., m calls n),
2. *Stack trace* after a crash, i.e., the order relation among the methods in the stack trace (the dynamic information in execution and stack traces have been showed to be useful in FL [75, 72]),
3. *Co-Change relation* in the project history (two methods that were changed in the same commit are considered to have the co-change relation),
4. *Co-Fixing relation* among the methods (two methods that were fixed for the same bug are considered to have the co-fixing relation),
5. *Similarity*: we also extract the similar methods in the project that have been buggy before in the project history. We keep only the most similar method for each method.

For a statement s, we extract both static and dynamic information. First, for static data, we extract the AST subtree that corresponds to s to represent its structure. We also extract the list of variables in s together with their types, forming

a sequence of names and types, e.g., *"name String price int ..."*. Second, for dynamic data, we encode the test coverage matrix for s into the feature vectors.

At both method and statement levels, we use graphs to represent the methods and statements, and their relations. Let us call them the method-level and statement-level *feature graphs*.

3.4.3.1.2 Step 2. Graph-Based feature representation learning. This step is aimed to learn the vector representations (i.e., embeddings) for the nodes in the feature graphs from step 1. The input includes the method-level and statement-level feature graphs. The output includes the embeddings for the nodes in the method-/statement-level feature graphs. The graph structures for both feature graphs are un-changed after this step.

For the content of a method or statement, we use the embedding techniques accordingly to feature representations (Sub-Section 3.4.5). For the method's content and a list of variables in a statement, the representation is a sequence of sub-tokens. We use GloVe [122] to produce the embeddings for all sub-tokens as we consider a method or statement as a sentence in each case. We then use Gated Recurrent Unit (GRU) [26] to produce the vector for the entire sequence.

For the structure of a method or statement, the representation is a (sub)tree in the AST. For this, we first use GloVe [122] to produce the embeddings for all the nodes in the sub-tree, considering the entire method or statement as a sentence in each case. After obtaining the sub-tree where the nodes are replaced by their GloVe's vectors, we use TreeCaps [20], which captures well the tree structure, to produce the embedding for the entire sub-tree.

For the code coverage representation, we directly use the two vectors for coverage and passing/failing and concatenate them to produce the embedding. The embedding for the most similar buggy method is computed in the same manner as explained with GloVe and TreeCaps. Finally, the embeddings for the attributes of the nodes are used

in the fully connected layers to produce the embedding for each node in the feature graph at the method level. Similarly, we obtain the feature graph at the statement level in which each node is the resulting vector of the fully connected layers.

3.4.3.1.3 Step 3. Dual-Task learning fault localization. After the feature representation learning step, we obtain two feature graphs at the method and statement levels, in which a node in either graph is a vector representation. The two graphs are used as the input for dual-task learning. For dual-task learning, we use two Graph-Based Convolution Network (GCN) models [62] for the method-level FL model (*MethFL*) and the statement-level FL model (*StmtFL*) to learn the CC fixing methods and CC fixing statements, respectively. During training, the two feature graphs at the method and statement levels are used as the inputs of *MethFL* and *StmtFL*. The two GCN models play the role of binary classifiers for the bugginess for the nodes (i.e., methods/statements). We train the two models simultaneously with soft-sharing of parameters. Details will be given in Sub-Section 3.4.6.

3.4.3.2 Predicting process. The input of the prediction process (Figure 3.25) includes the test cases and the source code in the project. The steps 1–2 of the process is the same as in training. In step 3, the feature graph g_M at the statement level built from the source code is used as the input of the *trained StmtFL* model, which predicts the labels of the nodes in that graph. The labels indicate the bugginess of the corresponding statements in the source code, which represent the CC fixing statements. If one aims to predict the faulty methods, the trained *MethFL* model can be used on the feature graph to produce the CC fixing methods.

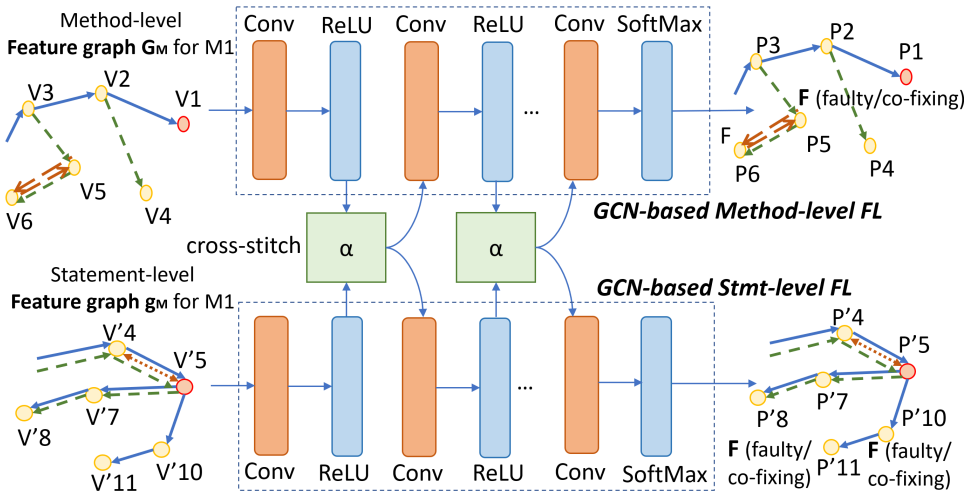


Figure 3.25 Bug Detection: FixLocator prediction process.

3.4.4 Feature extraction

3.4.4.1 Method-Level feature extraction G_M. Figure 3.26 illustrates the key attributes and relations that we collect. For each method M_1, we extract the following attributes:

1) *The method's content*: we remove special characters and separators in the method's interface and body, and use naming convention to break each code token into the sub-tokens. For example, in Figure 3.26, the node M_1 represents the method *computeGeometricalProperties* in Figure 3.27. For the content for M_1, the extracted sequence of sub-tokens is *protected*, *void*, *compute*, *Geometrical*, *Properties*, etc.

2) *The method's structure*: the corresponding parser is used to build the AST of the method (e.g., JDT [51] for Java code).

3) *Most similar faulty method*: we keep the most similar faulty method M_b with M_1. Note that we keep M_b as an attribute of M_1, rather than representing M_b in the feature graph. The rationale is that M_b might be in the past and might not be present in the current version of the project. Two methods are similar when they have similar sequences (measured by the cosine similarity) of the sub-tokens (represented by the GloVe embeddings [122]). For M_b, we build its AST and keep it as an attribute for M_1.

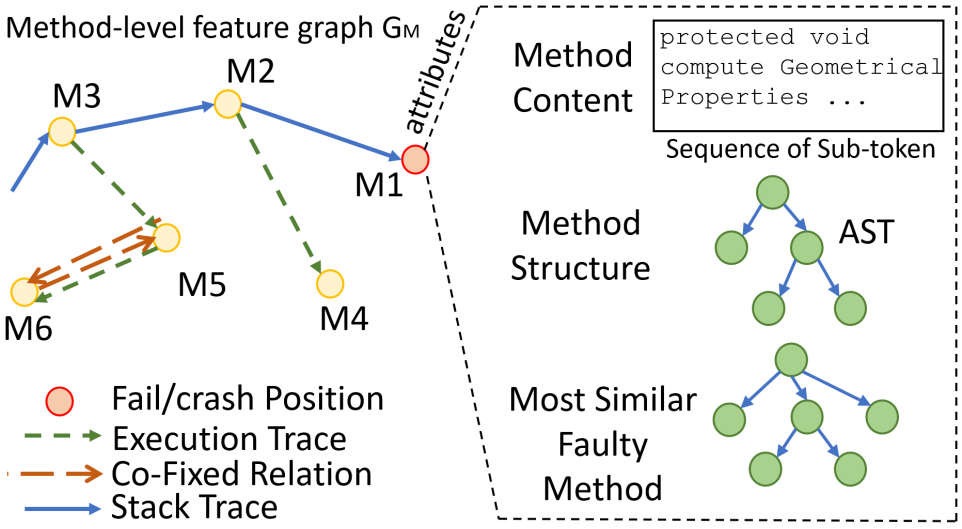


Figure 3.26 Bug Detection: Method-Level feature extraction G_M for $M1$.

We encode as the edges three types of relations:

1) *Calling relation in a stack trace*: we encode into the feature graph the calling relations in a stack trace of a failed test case as we ran it. In Figure 3.26, a blue edge connects M_i to M_j for that relation. Since the stack trace can be long, from the failing/crash point, we collect only part of the stack trace with n levels of depth from that point. Following a prior work [177], in our experiment, n=10.

2) *Calling relations in an execution trace*: Similar to the stack trace, an execution trace needs to be encoded in the feature graph. It can be very long from the failing/crash point. Thus, we keep the methods with only m levels of length in calling relations from that point. In our experiment, we use m=10. Figure 3.26 illustrates a few calling relations (in green color) in execution traces.

3) *Co-Change/co-fixing relation*: Such a relation exists between two methods that were changed/fixed in a commit. Such an edge is made into two one-directional edges (e.g., $M_5 \leftrightarrows M_6$ in Figure 3.26).

3.4.4.2 Statement-Level feature extraction g_M. For each statement, we extract the following attributes.

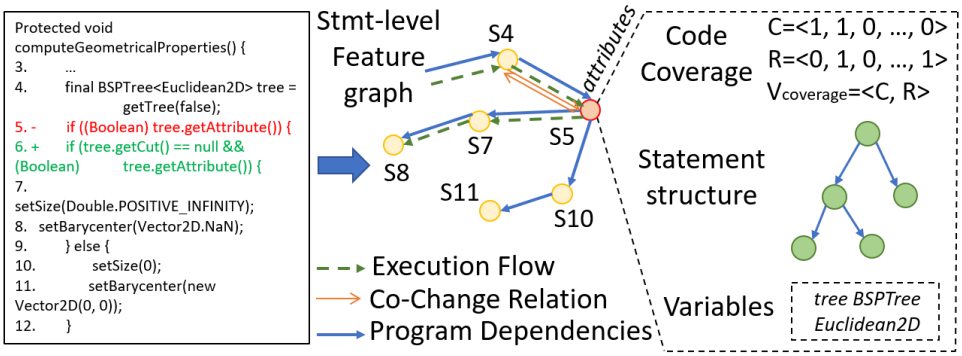


Figure 3.27 Bug Detection: Stmt-Level feature extraction g_M for $M1$ in Figure 3.26.

1) *Code coverage*: we run the test cases and collect code coverage information. For each statement s, we use a vector $C = <c_1, c_2, ..., c_K>$ (K is the number of test cases) to encode code coverage in which $c_i = 1$ if the test t_i covers s, and $c_i = 0$ otherwise. We use another vector $R = <r_1, r_2, ..., r_K>$ to encode the passing/failing of a test case in which $r_i = 1$ if the test case t_i is a passing one and $r_i = 0$ otherwise. R is common for all the statements. We concatenate C and R for each statement to obtain the code coverage feature vector $V_{Cov} = <c_1, c_2, ..., c_K, r_1, r_2, ..., r_K>$. We used DeepRL4FL's test ordering algorithm [75] as the ordering of test cases is useful in FL. For the different numbers of test cases across files, we perform zero padding to make the vectors have the same length.

2) *AST structure*: we extract the sub-tree in the AST that corresponds to the current statement.

3) *List of variables*:. We break the names into sub-tokens. In Figure 3.27, the sequence for the variable list is [*tree*, *BSPTree*, *Euclidean2D*,...].

We encode the following types of relations among statements:

1) *Program dependence graph (PDG)*: as suggested in [75], the relations among statements in an PDG are important in FL, thus, we integrate them into the feature graph. In Figure 3.27, the blue edges represent the relations in the PDG for the given

code. The statement at line 4 has a control/data dependency with the one at line 5, which connects to the ones at lines 7–8, and to the ones at lines 10–11.

2) *Execution flow in an execution trace*: if two statements are executed consecutively in an execution trace, we will connect them together. In Figure 3.27, we have the execution flow $S_5 \rightarrow S_7$, $S_7 \rightarrow S_8$.

3) *Co-Change/co-fixing relation*: we maintain the co-change/co-fixing relations among statements. In Figure 3.27, S_4 and S_5 have been changed in a commit, thus, two co-change edges connect them.

3.4.5 Feature representation learning

The goal of this step is to learn to build the vector representations for the nodes in the feature graphs at the method and statement levels. At either level, the input includes the attributes of either a method or a statement as in Figures 3.26 and 3.27. The output is each feature graph in which the nodes are replaced by their embeddings.

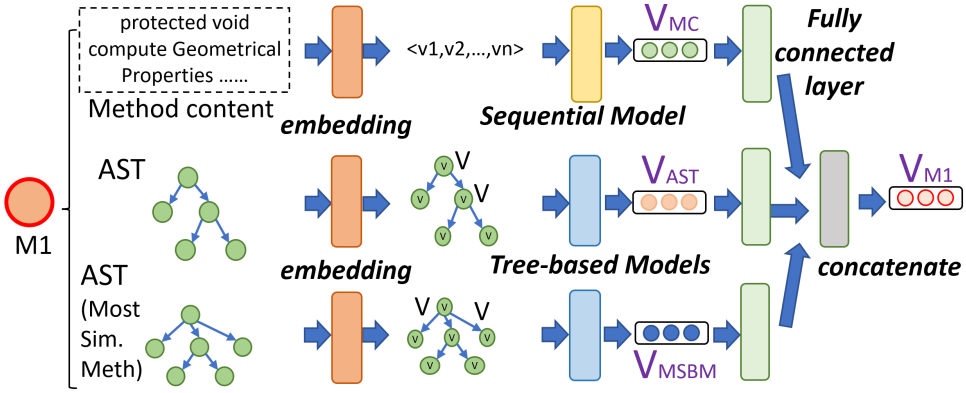


Figure 3.28 Bug Detection: Method-Level feature representation learning.

3.4.5.1 Method-Level representation learning.

Figure 3.28 shows how we build the vectors for a method's attributes.

1) *The method's content*: the method's content is represented by the sequence Seq_c of the sub-tokens built from the code tokens in the interface and the body of the method. To vectorize each sub-token in Seq_c, we use a word embedding model, called GloVe [122], and treat each method as a sentence. After this vectorization, for the

method, we obtain the sequence $<v_1, v_2, ..., v_n>$ of the vectors of the sub-tokens in Seq_c. We then apply a sequential model on the sequence $<v_1, v_2, ..., v_n>$ to learn the "summarized" vector V_{MC} that represents the method's content. Specifically, we use Gated Recurrent Unit (GRU) [27], a type of RNN layer that is efficient in learning and capturing the information in a sequence.

2) *The method's structure*: we first treat the method as the sequence of tokens and use GloVe to build the embeddings for all the tokens as in 1). We then replace every node in the AST of the method with the GloVe's vector of the corresponding token of the node (Figure 3.28). From the tree of vectors, we use a tree-based model, called TreeCaps [20], to capture its structure to produce the "summarized" vector V_{AST} representing the method's structure.

3) *Most similar faulty method*: for a method, we process the most similar buggy method M_b in the same way as the method's structure via GloVe and TreeCaps to learn the vector V_{MSBM} for M_b.

Finally, for each method M_1, we obtain V_{MC}, V_{AST}, and V_{MSBM}.

3.4.5.2 Statement-Level representation learning. Figure 3.29 shows how we build the vectors for a statement's attributes.

1) *Code Coverage*: we directly use the vector $V_{Cov} = <c_1, c_2, ..., c_K, r_1, r_2, ..., r_K>$ computed in Sub-Section 3.4.4.2 for the next computation.

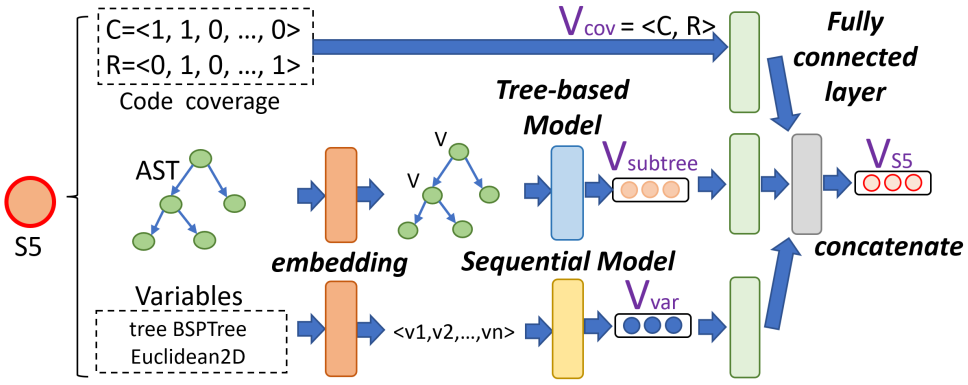


Figure 3.29 Bug Detection: Statement-Level feature representation learning.

2) *The statement's structure*: we process the AST subtree representing the statement's structure in the same manner (via GloVe and TreeCaps) as for the method's structure to produce $V_{subtree}$.

3) *List of variables*: as with the method's content, we run GloVe on the sequence of sub-tokens to produce a sequence of vectors and use GRU to produce the summarized vector V_{var} for the list.

Finally, for each statement S, we obtain V_{COV}, $V_{subtree}$, and V_{var}.

3.4.5.3 Feature representation learning. After computing the three embeddings for three attributes of each method, we use three fully connected layers to standardize each vector's length to a chosen value l. Similarly, we use three fully connected layers for the three embeddings for each statement. Then, for a method or a statement, we concatenate the three output vectors from the fully connected layer to produce the vector V_M for the method and the other three vectors for V_S for the statement with the length of $(l \times 3)$.

After all, for a method M, we have the method-level graph G_M and the statement-level graph g_M with its statements. The nodes in G_M (Figure 3.26) now are the vectors computed for methods, and the nodes in g_M are the vectors V_S for the statements in M (Figure 3.27).

3.4.6 Dual-Task learning for fault localization

Figures 3.30 and 3.31 illustrate our dual-task learning for fault localization. In the training dataset, for each bug B, *to ensure the matching of a method and its corresponding statements*, we build for each faulty method M the pairs (G_M, g_M): 1) G_M, the method-level graph (Figure 3.26 with nodes replaced by vectors); and 2) g_M, the statement-level graph (Figure 3.27) containing all the statements belonging to M. *To ensure the co-fixing connections among the buggy methods for the same bug* B, we model the *co-fixed methods* of M via co-fixed relations in G_M (Figure 3.26). At

the output layer, we label those methods as *faulty/co-fixing*. The *co-fixed statements* within g_M for the bug B are also labeled as *faulty/co-fixing*. The non-buggy methods or statements are labeled as *non-faulty*. The pairs (G_M, g_M) are used as the input of this dual-task learning model (Figure 3.30). We process all the faulty methods M for each bug B, and non-buggy methods.

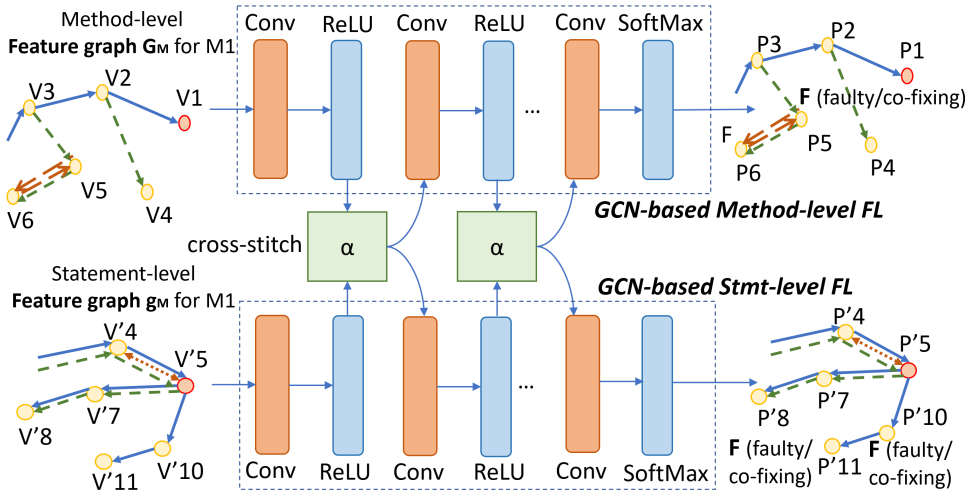


Figure 3.30 Bug Detection: Dual-Task learning fault localization.

In prediction, for each method M^* in the project, we build the pair (G_{M^*}, g_{M^*}) and feed it to the trained dual-task model. In the output graphs, each node (for a method or a statement) will be classified as either *faulty/co-fixing* or *non-faulty*. The nodes with *faulty/co-fixing* labels in g_{M^*} are the co-fixing statements for the bug. Let us explain our dual-task learning in details.

3.4.6.1 Graph Convolutional Network (GCN) for FL. First, FixLocator has two GCN models [62], each for FL at the method and statement levels. GCN processes the attributes of the nodes (vectors) and their edges (relations) in feature graphs. Each GCN model has $n-1$ pairs of a graph convolution layer ($Conv$) and a rectified linear unit ($ReLU$). They are aimed to consume and learn the characteristic features in the input feature graphs. The last pair of each GCN model is a pair of a graph convolution layer ($Conv$) and a softmax layer ($SoftMax$). The $SoftMax$

layer plays the role of the classifier to determine whether a node for a method or a statement is labeled as *faulty/co-fixing* or *non-faulty*.

3.4.6.2 Dual-Task learning with Cross-Stitch units. In a regular GCN model, those above pairs of *Conv* and *ReLU* are connected to one another. However, to achieve dual-task learning between method-level and statement-level FL (*methFL* and *stmtFL*), we apply a cross-stitch unit [101] to connect the two GCN models. The sharing of representations between *methFL* and *stmtFL* is modeled by learning a linear combination of the input features in both feature graphs G_M and g_M. At each of the *ReLU* layer of each GCN model (Figure 3.31), we aim to learn such a linear combination of the output from the graph convolution layers (*Conv*) of *methFL* and *stmtFL*.

The top sub-network in Figure 3.30 gets direct supervision from *methFL* and indirect supervision (through cross-stitch units) from *stmtFL*. Cross-Stitch units regularize *methFL* and *stmtFL* by learning and enforcing shared representations by combining feature maps [101].

Formulation. For each pair of the GCN model, the outputs of the *ReLU* layer, called the hidden states, are computed as follows:

$$\hat{A} = D'^{-\frac{1}{2}} A' D' - \frac{1}{2} \tag{3.5}$$

$$H^i = \Delta(\hat{A} X^i W^i) \tag{3.6}$$

Where A' is the adjacency matrix of each feature graph; D' is the degree matrix; W^i is the weight matrix for layer i; X^i is the input for layer i; H^i is the hidden state of layer i and the output from the *ReLU* layer; and Δ is the activation function *ReLU*. In a regular GCN, H^i is the input of the next layer of GCN (i.e., the input of *Conv*).

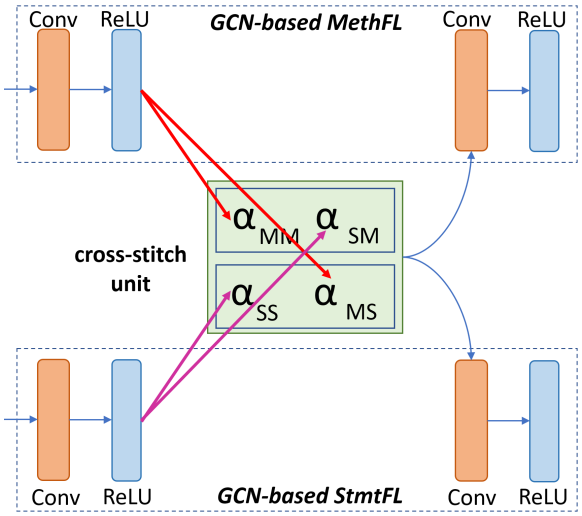


Figure 3.31 Bug Detection: Dual-Task learning via cross-stitch unit.

In Figures 3.30 and 3.31, a cross-stitch unit is inserted between the *ReLU* layer of the previous pair and the *Conv* layer of the next one. The input of the cross-stitch unit includes the outputs of the two *ReLU* layers: H_M^i and H_S^i (i.e., the hidden states of those layers in *methFL* and *stmtFL*). We aim to learn the linear combination of both inputs of the cross-stitch unit, which is parameterized using the weights α. Thus, the output of the cross-stitch unit is computed as:

$$\begin{bmatrix} X_M^{i+1} \\ X_S^{i+1} \end{bmatrix} = \begin{bmatrix} \alpha_{MM} & \alpha_{MS} \\ \alpha_{SM} & \alpha_{SS} \end{bmatrix} \begin{bmatrix} H_M^i \\ H_S^i \end{bmatrix} \tag{3.7}$$

α is the trainable weight matrix; X_M^{i+1} and X_S^{i+1} are the inputs for the $(i+1)^{th}$ layers of the GCNs at the method and statement levels.

X_M^{i+1} and X_S^{i+1} contain the information learned from both *MethFL* and *StmtFL*, which helps achieve the main goal for dual-task learning to enhance the performance of fault localization at both levels.

In general, αs can be set. If α_{MS} and α_{SM} are set to zeros, the layers are made to be task-specific. The α values model linear combinations of feature maps. Their initialization in the range [0,1] is important for stable learning, as it ensures that

values in the output activation map (after cross-stitch unit) are of the same order of magnitude as the input values before linear combination [101].

If the sizes of the H_M^i and H_S^i are different, we need to adjust the sizes of the matrices. From Formula 3.7, we have:

$$X_M^{i+1} = \alpha_{MM} H_M^i + \alpha_{MS} H_S^i \tag{3.8}$$

$$X_S^{i+1} = \alpha_{SM} H_M^i + \alpha_{SS} H_S^i \tag{3.9}$$

We resize H_s^i in Formula 3.8 and resize H_m^i in Formula 3.9 if needed. We use the *bilinear interpolation* technique [159] in image processing for resizing. We pad zeros to the matrix to make the aspect ratio 1:1. If the size needs to be reduced, we do the center crop on the matrix to match the required size.

FixLocator also has a trainable threshold for *SoftMax* to classify if a node corresponding to a method or a statement is faulty or not.

3.4.7 Empirical evaluation

3.4.7.1 Research questions. For evaluation, we seek to answer the following research questions:

RQ1. Comparison with State-of-the Art Deep Learning (DL)-Based Approaches. How well does FixLocator perform compared with the state-of-the-art DL-based fault localization approaches?

RQ2. Impact Analysis of Dual-Task Learning. How does the dual-task learning scheme affect FixLocator's performance?

RQ3. Sensitivity Analysis. How do various factors affect the overall performance of FixLocator?

RQ4. Evaluation on Python Projects. How does FixLocator perform on Python code?

RQ5. Extrinsic Evaluation on Usefulness. How much does FixLocator help an APR tool improve its bug-fixing?

3.4.7.2 Experimental methodology.

3.4.7.2.1 Data set. We use a benchmark dataset Defects4J V2.0.0 [32] with 835 bugs from 17 Java projects. For each bug in a project P, Defects4J has the faulty and fixed versions of the project. The faulty and fixed versions contain the corresponding test suite relevant to the bug. With the *Diff* comparison between faulty and fixed versions of a project, we can identify the faulty statements. Specifically, for a bug in P, Defects4J has a separate copy of P but with only the corresponding test suite revealing the bug. For example, P_1, a version of P, passes a test suite T_1. Later, a bug B_1 in P_1 is identified. After debugging, P_1 has an evolved test suite T_2 detecting the bug. In this case, Defects4J has a separate copy of the buggy P_1 with a single bug, together with the test suite T_2. Similarly, for bug B_2, Defects4J has a copy of P_2 together with T_3 (evolving from T_2), and so on. We do not use the whole T of all test suites for training/testing. For within-project setting, we test one bug B_i with test suite $T_{(i+1)}$ by training on all other bugs in P. We conducted all the experiments on a server with 16 core CPU and a single Nvidia A100 GPU.

In Defects4J-v2.0, regarding the statistics on the number of buggy/fixed statements for a bug, there are 199 bugs with one buggy/fixed statement, 142 bugs with two, 90 bugs with three, 78 bugs with four, 43 bugs with five, and 283 bugs with >5 buggy statements. Regarding the statistics on the number of buggy/fixed methods/hunks for a bug, there are 199 bugs with one-method/one-statement, 105 bugs with one-method/multi-statements, 142 bugs with multi-methods/one-statement for each method, 61 bugs with multi-methods/multi-statements for each method, and 357 bugs with multiple methods, each has one or multiple buggy statements. Thus, there are *665 (out of 864 bugs) with CC fixing statements*.

3.4.7.2.2 Experiment setup and procedure.

RQ1. Comparison with DL-Based FL Approaches.

Baselines. Our tool aims to output a **set** of CC fixing statements for a bug. However, the existing Deep Learning-Based FL approaches can produce only the ranked **lists** of suspicious statements with scores. Thus, we chose as baselines the most recent, state-of-the-art, DL-based, statement-level FL approaches: (1) **CNN-FL** [193]; (2) **DeepFL** [72]; and (3) **DeepRL4FL** [75]; then, we use the predicted, ranked list of statements as the output set. For the *comparison in ranking with those ranking baselines*, we convert our tool's result into a ranked list by ranking the statements in the predicted set by the classification scores (i.e., before deriving the final set). We also compare with the CC fixing-statement detection module in **DEAR** [76], a multi-method/multi-statement APR tool.

Procedures. We use the leave-one-out setting as in prior work [72, 73] (i.e., testing on one bug and training on all other bugs). We also consider the order of the bugs in the same project via the revision numbers. Specifically, for each buggy version B of project P in Defects4J, all buggy versions from the other projects are first included in the training data. Besides, we separate all the buggy versions of the project P into two groups: 1) one buggy version as the test data for model prediction, and 2) all the buggy versions of the same project P that have occurred before the buggy version B are also included in the training data. If the latter group is empty, only the buggy versions from the other projects are used for training to predict for the current buggy version in P.

We tune all models using autoML [98] to find the best parameter setting. We directly follow the baseline studies to select the parameters that need to be tuned in the baselines. We tuned our model with the following key hyper-parameters to obtain the best performance: (1) Epoch size (i.e., 100, 200, 300); (2) Batch size (i.e., 64, 128, 256); (3) Learning rate (i.e., 0.001, 0.003, 0.005, 0.010); (4) Vector length of

word representation and its output (i.e., 150, 200, 250, 300); (5) The output channels of convolutional layer (16, 32, 64,128); (6) The number of convolutional layers (3, 5, 7, 9).

DeepFL was proposed for the method-level FL. For comparison, following a prior study [75], we use only DeepFL's spectrum-based and mutation-based features applicable to detect faulty statements.

Evaluation Metrics. We use the following metrics for evaluation:

(1) **Hit-N** measures the number of bugs that the predicted set contains *at least* N faulty statements (i.e., the predicted and oracle sets for a bug overlap at least N statements regardless of the sizes of both sets). Both precision and recall can be computed from Hit-N.

(2) **Hit-All** is the number of bugs in which the predicted set covers the correct set in the oracle for a bug.

(3) **Hit-N@Top-K** is the number of bugs that the predicted list of the top-K statements contains at least N faulty statements. This metric is used when we compare the approaches in ranking.

RQ2. Impact Analysis of Dual-Task Learning Model.

Baselines. To study the impact of dual-task learning, we built two variants of FixLocator: (1) *Statement-Only model:* the method-level FL model (*methFL*) is removed from FixLocator and only statement-level FL (*stmtFL*) is kept for training. (2) *Cascading model:* in this variant, dual-task learning is removed, and we cascade the output of *methFL* directly to the input of *stmtFL*.

Procedures. The statement-only model has only the statement-level fault localization. We ran it on all methods in the project to find the faulty statements. We use the same training strategy and parameter tuning as in RQ1. We use Hit-N for evaluation.

RQ3. Sensitivity Analysis. We conduct ablation analysis to evaluate the impact of different factors on the performance: every *node feature*, *co-change relation*, and *the depth limit on the stack trace and the execution trace*. Specifically, we set FixLocator as the complete model, and each time we built a variant by removing one key factor, and compared the results. Except for the removed factor, we keep the same setting as in other experiments.

RQ4. Evaluation on Python Projects.

To evaluate FixLocator on different programming languages, we ran it on the Python benchmark BugsInPy [19, 171] with 441 bugs from 17 different projects.

RQ5. Extrinsic Evaluation. To evaluate usefulness, we replaced the original CC fixing-location module in DEAR [76] with FixLocator to build a variant of DEAR, namely DEAR^{FixL}. We also added FixLocator and Ochiai FL [1] to CURE [54] to build two variants: CURE^{FixL} (FixLocator + CURE) and CURE^{Ochi} (Ochiai+CURE).

3.4.7.3 Empirical results.

3.4.7.3.1 RQ1. Comparison results with state-of-the-art DL-Based FL approaches. Table 3.17 shows *how well FixLocator's coverage is on the actual correct CC fixing statements (recall)*. The result is w.r.t. the bugs in the oracle with *different numbers K of CC fixing statements*: $K = \#CC\text{-}Stmts = 1, 2, 3, 4, 5,$ and $5+$. For example, in the oracle, there are 90 bugs with 3 faulty statements. FixLocator's predicted set correctly contains all 3 buggy statements for 21 bugs (Hit-All), 2 of them for 25 bugs, and 1 faulty statement for 51 bugs. As seen, regardless of N, FixLocator performs better in any Hit-N over the baselines for all Ks. Note that Hit-All = Hit-N when N(#overlaps) = K(#CC-Stmts).

Table 3.18 shows the summary of the comparison results in which we sum all the corresponding Hit-N values across different numbers K of CC fixing statements in Table 3.17. As seen, FixLocator can improve CNN-FL, DeepFL, DeepRL4FL, and

DEAR by 16.6%, 16.9%, 9.9%, and 20.6%, respectively, in terms of Hit-1 (i.e., the predicted set contains at least one faulty statement). It also improves over those baselines by 33.6%, 40.3%, 26.5%, and 57.5% in terms of Hit-2, 43.9%, 46.4%, 28.1%, and 51.9% in terms of Hit-3, 100%, 155.6%, 64.5%, and 142.1% in terms of Hit-4. Note: Any Hit-N reflects the cases of multiple CC statements. For example, Hit-1 might include the bugs with more than one buggy/fixed statement.

Table 3.17 Bug Detection: RQ1. Detailed Comparison w.r.t. Faults with Different # of CC Fixing Statements in an Oracle Set (Recall).

#CC-Stmts in Oracle	Metrics	CNN-FL	DeepFL	DeepRL4FL	DEAR	FIX-LOCATOR
1 (199 bugs)	Hit-1	78	76	84	74	93
2 (142 bugs)	Hit-1	67	64	70	65	75
	Hit-2	33	30	34	28	41
3 (90 bugs)	Hit-1	46	44	47	42	51
	Hit-2	21	20	23	20	25
	Hit-3	11	10	13	12	21
4 (78 bugs)	Hit-1	41	42	42	40	45
	Hit-2	22	19	21	20	24
	Hit-3	9	7	8	5	12
	Hit-4	3	2	4	2	9
5 (43 bugs)	Hit-1	15	14	16	13	18
	Hit-2	9	8	9	7	12
	Hit-3	6	5	6	5	7
	Hit-4	3	2	3	2	3
	Hit-5	1	1	1	0	1
5+ (283 bugs)	Hit-1	85	91	93	87	105
	Hit-2	40	42	45	41	65
	Hit-3	31	34	37	32	42
	Hit-4	17	14	21	15	34
	Hit-5	4	3	5	2	8
	Hit-5+	1	2	3	1	3

Table 3.18 Bug Detection: RQ1. Comparison Results with DL-based FL Models.

Metrics	CNN-FL	DeepFL	DeepRL4FL	DEAR	FixLocator
Hit-1	332	331	352	321	387
Hit-2	125	119	132	106	167
Hit-3	57	56	64	54	82
Hit-4	23	18	28	19	46
Hit-5	5	4	6	2	9
Hit-5+	1	2	3	1	3
Hit-All	127	121	139	115	168

Importantly, our tool produced the exact-match sets for 168/864 bugs (19.5%), relatively improving over the baselines 32%, 38.8%, 20.8%, and 46.1% in Hit-All. It performs well in Hit-All when the number of CC statements K=1-4. However, producing the exact-matched sets for all statements when $K \geq 5$ is still challenging for all the models.

Table 3.19 shows the comparison on *how precise the results are* in a predicted set. For example, when *the number of the CC statements in a predicted set* is K'=3, there are 23 bugs in which all of those 3 faulty statements are correct (there might be other statements missing). There are 27 bugs in which two of the 3 predicted, faulty statements are correct. There are 55 bugs in which only one of the 3 predicted, faulty statements are correct. As seen, regardless of N, FixLocator is more precise than the baselines for all K's.

Table 3.20 shows the comparison as ranking is considered (Hit-N@Top-K). As seen, in the ranking setting, FixLocator locates more CC fixing statements than any baseline. For example, FixLocator improves the best baseline DeepRL4RL by

23.9% in Hit-2@Top-5, 22.6% in Hit-3@Top-5, 43.8% in Hit-4@Top-5, and 22.2% in Hit-5@Top-5, respectively. The same trend is for Hit-N@Top-10.

Table 3.19 Bug Detection: RQ1. Detailed Comparison w.r.t. Faults with Different # of CC Fixing Statements in a Predicted Set (Precision).

#Stmts in Predicted Set	Metrics	CNN-FL	DeepFL	DeepRL4FL	DEAR	FixLocator
1 (203 bugs)	Hit-1	83	79	87	75 (183)	99
2 (165 bugs)	Hit-1	75	72	78	71 (172)	83
	Hit-2	36	34	39	34 (172)	45
3 (120 bugs)	Hit-1	52	46	48	41 (129)	55
	Hit-2	24	22	26	19 (129)	27
	Hit-3	12	11	14	10 (129)	23
4 (96 bugs)	Hit-1	47	49	46	33 (78)	51
	Hit-2	24	21	22	14 (78)	26
	Hit-3	11	9	10	5 (78)	14
	Hit-4	5	3	6	1 (78)	11
5 (73 bugs)	Hit-1	17	16	17	12 (55)	19
	Hit-2	10	10	11	7 (55)	14
	Hit-3	8	6	7	4 (55)	9
	Hit-4	3	3	4	1 (55)	5
	Hit-5	2	1	2	0 (55)	2
5+ (178 bugs)	Hit-1	58	69	76	68(218)	80
	Hit-2	31	32	34	32 (218)	55
	Hit-3	26	30	33	24 (218)	36
	Hit-4	15	12	18	16 (218)	30
	Hit-5	3	3	4	5 (218)	7
	Hit-5+	1	2	3	2 (218)	3

We did not compare with the spectrum-/mutation-based FL models since DeepRL4FL [75] was shown to outperform them.

We also performed the analysis on the overlapping between the results of FixLocator and each baseline. As seen in Table 3.21, FixLocator can detect at least

one correct faulty statement in 103 bugs that CNN-FL missed, while CNN-FL can do so only in 48 bugs that FixLocator missed. Both FixLocator and CNN-FL can do so in the same 284 bugs. In brief, FixLocator can detect at least one correct buggy statement in more "unique" bugs than any baseline.

Table 3.20 Bug Detection: RQ1. Comparison with Baselines w.r.t. Ranking.

	Hit-N@Top-5					Hit-N@Top-10					
N=	1	2	3	4	5	1	2	3	4	5	5+
CNN-FL	533	311	133	33	4	578	386	166	42	10	81
DeepFL	525	298	131	35	6	563	364	156	42	10	83
DeepRL4FL	586	339	159	32	9	623	407	186	48	13	92
DEAR	501	274	119	25	3	544	341	142	36	7	71
FixLocator	633	420	195	46	11	690	470	217	51	13	94

Table 3.21 Bug Detection: Overlapping Analysis Results for Hit-1.

	FixLocator		
	Unique-Baseline	Overlap	Unique-FixLocator
CNN-FL	48	284	103
DeepFL	54	277	110
DeepRL4FL	61	291	96
DEAR	35	286	101

3.4.7.3.2 RQ2. Impact analysis results on dual-task learning. Table 3.22 shows that FixLocator has better performance in detecting CC fixing statements than the two variants (statement-only and cascading models). This result shows that the **dual-task learning helps improve FL over the cascading model**

(*methFL* → *stmtFL*). Moreover, without the impact of method-level FL (*methFL*), the performance decreases significantly, indicating *methFL*'s contribution.

Table 3.22 Bug Detection: RQ2. Impact Analysis of Dual-Task Learning.

Variant	Hit-1	Hit-2	Hit-3	Hit-4	Hit-5	Hit-5+
Stmt-only	304	111	51	11	3	2
Cascading	343	125	61	19	3	3
FixLocator	387	167	82	46	9	3

3.4.7.3.3 RQ3. Sensitivity analysis results.

Impact of the Method-Level (ML) Features and ML Co-Change Relation

Among all the method-level features/attributes of FixLocator, the *feature of co-change relations among methods has the largest impact*. Specifically, without the co-change feature among methods, Hit-1 is decreased by 8.3%. Moreover, the *method structure feature, represented as AST, has the second largest impact*. Without the method structure feature, Hit-1 is decreased by 7.8%.

Among the last two method-level features with least impact, the method content feature has less impact than the similar-buggy-method feature. This shows that *the bugginess nature of a method and similar ones has more impact than the tokens of the method itself*.

Impact of the Statement-Level (SL) Features and SL Co-Change Relation Among all the statement-level features, *Code Coverage has the largest impact*. Without *Code Coverage* feature, Hit-1 is decreased by 10.1%. The co-change relations among statements have the second largest impact among all SL features/attributes. Specifically, without the co-change relations among statements, Hit-1 is decreased by 9.3%.

Impact of the Depth Level of Stack Trace As seen in Table 3.24, FixLocator can achieve the best performance when depth=10. The cases with depth=

5 or 15 can bring into analysis too few or too many irrelevant methods, causing more noises to the model. Thus, we chose depth=10 for our experiments.

Table 3.23 Bug Detection: RQ3. Sensitivity Analysis of Method- and Statement-Level Features.

Model Variant		Hit-N					
		1	2	3	4	5	5+
ML	w/o Method Content	366	158	78	39	9	3
	w/o Method Structure	357	155	80	40	8	3
	w/o Similar Buggy Method	361	157	79	44	9	3
	w/o ML Co-Change Rel.	355	152	77	40	8	3
SL	w/o Code Coverage	348	151	75	38	7	2
	w/o AST Subtree	354	153	77	41	8	3
	w/o Variables	373	162	78	42	9	3
	w/o SL Co-Change Relation	351	150	76	39	7	2
	FixLocator	387	167	82	46	9	3

Notes: ML: Method-Level; SL: Statement-Level.

Table 3.24 Bug Detection: RQ3. Sensitivity Analysis (Depth of Traces).

Depth	Hit-N					
	1	2	3	4	5	5+
5	371	162	74	42	8	3
10	387	167	82	46	9	3
15	368	158	71	39	7	3

Illustrating Example Table 3.25 displays the ranking from the models for Figure 3.32. FixLocator correctly produces all 4 CC fixing statements in its predicted

```
1 public UnivariateRealPointValuePair optimize(final FUNC f, GoalType goal,
2     double min, double max) throws FunctionEvaluationException {
3 -    return optimize(f, goal, min, max, 0);
4 +    return optimize(f, goal, min, max, min + 0.5 * (max - min));
5 }
6 public UnivariateRealPointValuePair optimize(final FUNC f, GoalType goal,
7     double min, double max, double startValue) throws Func...Exception {
8     ...
9     try {
10 -       final double bound1 = (i == 0) ?  min :  min + generator.nextDouble()...;
11 -       final double bound2 = (i == 0) ?  max :  min + generator.nextDouble()...;
12 -       optima[i] = optimizer.optimize(f, goal, FastMath.min(bound1, bound2),...;
13 +       final double s = (i == 0) ?  startValue :  min + generator.nextDouble()...;
14 +       optima[i] = optimizer.optimize(f, goal, min, max, s); ...
15    }
16 }
```

Figure 3.32 Bug Detection: An illustrating example.

set (lines 2,8,9, and 10 in two methods). The statement-only model detects only line 2 as faulty. It completely missed lines 8–10 of the *optimize* method. In contrast, the cascading model detects lines 8–10, however, its *MethFL* considers the first method (*optimize(...)* at line 1) as non-faulty, thus, it did not detect the buggy line 2 due to its cascading.

Table 3.25 Bug Detection: Ranking of CC Fixing Locations for Figure 3.32.

LOC	CNN-FL	DeepFL	DeepRL4FL	DEAR	FixLocator
Line 2	**1**	22	**2**	27	★ (no rank)
Line 8	24	**3**	6	12	★ (no rank)
Line 9	25	**4**	7	13	★ (no rank)
Line 10	50+	13	16	39	★ (no rank)

The baselines CNN-FL, DeepFL, DeepRL4FL, and DEAR detect only 1, 2, 1, and 0 faulty statements (bold cells) in their top-4 resulting lists, respectively. In brief, the baselines are not designed to detect CC fixing locations, thus, their top-K lists are not correct.

3.4.7.3.4 RQ4. Evaluation on python projects. As seen in Table 3.26, FixLocator can localize 193 faulty statements with Hit-1. This shows that the performance on the Python projects is consistent with that on the Java projects. Specifically, at the statement level, the percentages of the total Python and Java bugs that can be localized are similar, e.g., 43.8% vs. 46.3% with Hit-1.

Table 3.26 Bug Detection: RQ4. BugsInPy (Python Projects) VS. Defects4J (Java Projects). P% = |Located Bugs|/|Total Bugs in Datasets|

Metrics	BugsInPy (Python projects)		Defects4J (Java projects)	
	P%	Cases	P%	Cases
Hit-1	43.8%	193	46.3%	387
Hit-2	16.3%	72	20.0%	167
Hit-3	10.2%	45	9.8%	82
Hit-4	3.4%	15	5.5%	46
Hit-5	0.7%	3	1.1%	9
Hit-5+	0%	0	0.4%	3

3.4.8 Further analysis

3.4.8.1 Running time. As seen in Table 3.27, except for DeepFL (using a basic neural network), the other approaches have similar training and prediction time. Importantly, prediction time is just a few seconds, making FixLocator suitable for interactive use.

Table 3.27 Bug Detection: Running Time.

Models	CNN-FL	DeepFL	DeepRL4FL	DEAR	FixLocator
Training Time	4 hours	5 mins	7 hours	21 hours	6 hours
Prediction Time	2 seconds	1 second	4 seconds	9 seconds	2 seconds

3.4.8.2 Limitations. First, our tool does not detect well the sets with +5 CC fixing statements since it does not learn well those large co-changes. Second, it does not work in locating a fault that require only adding statements to fix (neither do all baselines). Third, if the faulty statements/methods occur far from the crash method in the execution traces, it is not effective. Finally, it does not have any mechanism to integrate program analysis in expanding the faulty statements having dependencies with the detected faulty ones.

3.4.9 Threats to validity

(1) We evaluated FixLocator on Java and Python. Our modules are general for any languages. (2) We compared the models only on two datasets that have test cases. (3) For comparison, we use only DeepFL's features applicable to statement-level FL although it works at the method level. Other baselines work directly at the statement level. (4) In 501 bugs in BugsInPy, the third-party tool cannot process 60 of them. (5) We focus on CC fixing statements, instead of methods, due to bug fixing purpose.

3.5 Conclusion

In this chapter, we designed our approach with suitable learning code representations on three research topics: bug detection, single-statement fault localization, and multi-statement/multi-method fault localization.

As for bug detection, we propose a new deep learning-based bug detection to improve the existing state-of-the-art detection approaches. The key ideas that enable

our approach are (1) modeling and analyzing the relations among paths of ASTs from different methods using the Program Dependency Graph (PDG) and Data Flow Graph (DFG) and (2) using weights and attention mechanisms to emphasize previous buggy paths and differentiate them from non-buggy ones.

We propose a deep learning-based fault localization (FL) approach, DeepRL4FL, to improve existing FL approaches for single-statement fault localization. The key ideas include (1) treating the FL problem as the image recognition; (2) enhancing code coverage matrix by modeling the relations among statements and failing test cases; (3) combining code coverage representation learning with statement dependencies, and the code representation learning for usual suspicious code.

And for multi-statement/multi-method fault localization, we present FixLocator, a novel DL-based FL approach that aims to locate co-change fixing locations within one or multiple methods. The key ideas of FixLocator include (1) a new dual-task learning model of method- and statement-level fault localization to detect CC fixing locations; (2) a novel graph-based representation learning with co-change relations among methods and statements; (3) a novel feature in co-change methods/statements.

All the empirical results show that our approaches can outperform all the studied state-of-the-art approaches for each research topic. It proves that our idea of applying suitable learning code representations to help increase the performance of bug detection-related tasks is efficacious. In the testing stage, the next step is to fix those located bugs in the software after we have the location for the bugs by using fault localization. We will introduce how we use learning code representations to help fix those bugs automatically in Chapter 4.

CHAPTER 4

AUTOMATED PROGRAM REPAIR

4.1 Introduction

This chapter introduces two of our works related to automated program repair. (1) The first one focuses on using code representation learning with the deep learning model to improve the existing deep learning-based studies. (2) The second one is the improvement of the first one after we found that most of the existing approaches, including the first idea we have, can only fix the bugs that happen in one line. During our study of the real bugs, many were caused by multiple statements or methods. To deal with it, in the second idea, we would like to find a new way to fix more types of bugs, including the bugs involved in multiple hunks and multiple lines. Both of these two research topics are published as conference papers. The layout of this chapter, along with the associated code representations utilized for each research topic, can be visualized in Figure 4.1. We will separately introduce these two ideas in the following sections.

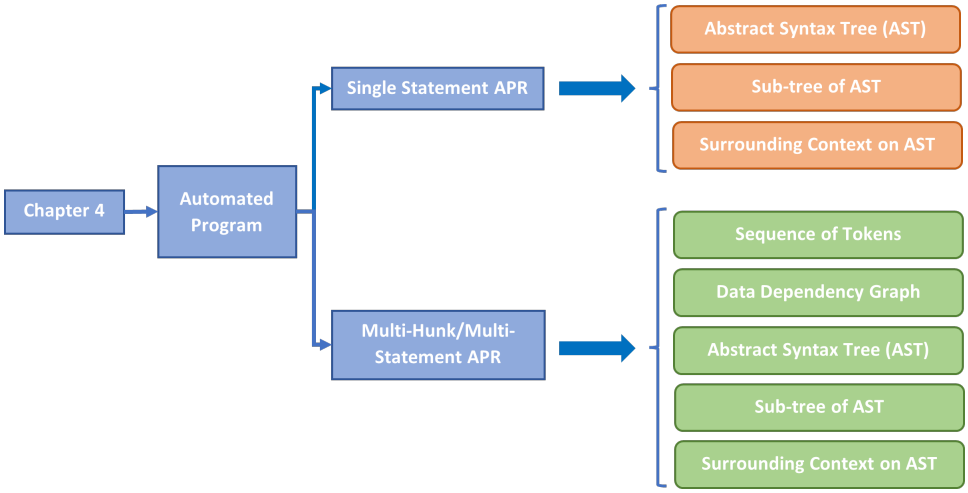


Figure 4.1 Automated Program Repair: Roadmap for Chapter 4.

4.2 Automated Program Repair for Single Statement Single Hunk Bugs

4.2.1 Introduction

Program repair is a vital activity aiming to fix defects during software development to ensure software quality. It requires much effort, time, and budgets for a software development team. Recognizing the importance of program repairing, several automated techniques/tools have been proposed to help developers in automatically identifying and fixing software defects in programs.

Recently, some approaches aim to *mine and learn fixing patterns* from prior bug fixes [112, 67, 86, 60]. The fixing patterns, also called *fixing templates*, could be automatically or semi-automatically mined [67, 112, 86, 87]. Some approaches are integrated with static and dynamic analysis, and constraint solving to synthesize a patch [112, 86]. Instead of mining buggy and fixed code, other approaches focus on *mining code changes* to generate a similar patch (e.g., CapGen [167], SimFix [52], FixMiner [64]). *Machine learning* has been used to implicitly mine the fixing patterns and/or further rank the candidate fixes based on existing patches (e.g., Prophet [92], Genesis [90]). Other approaches [134] explore *information retrieval* for better selecting/ranking candidate fixes.

With recent advances in deep learning (DL), several researchers have applied DL to automated program repair (APR). The first group of approaches (e.g., DeepFix [41], DeepRepair [168, 169]) leverages the capability of DL models in *learning similar source code for similar fixes*. For example, DeepRepair explores learned code similarities, captured with recursive auto-encoders [169], to select the repair ingredients from code fragments similar to the buggy code. The second group of approaches treats APR as a *statistical machine translation* that translates the buggy code to the fixed code. Rachet [46] and Tufano *et al.* [158] use sequence-to-sequence translation. They use neural network machine translation

(NMT) with attention-based Encoder-Decoder, and different code abstractions to generate patches, while SequenceR [24] uses sequence-to-sequence NMT with a copy mechanism [138]. CODIT [21] learns code edits with encoding code structures in an NMT model to generate fixes. Tufano *et al.* [156] learn code changes using sequence-to-sequence NMT with code abstractions and keyword replacing.

Despite their successes, the above neural network machine translation (NMT)-based models still have the following three key limitations in applying to APR. The first issue is that those approaches completely treat APR as a machine translation from buggy to fixed code. Specifically, during training, the *knowledge on what parts of the buggy code have changed and what have not for fixing a defect is not encoded in those NMT-based models for APR.* Instead, those NMT-based models are trained with a parallel corpus of the pairs of the source code of a buggy method and that of the corresponding fixed one. Without such knowledge, the models must learn to implicitly align the source code before and after the fix, and at the same time, to statistically derive the fixing patterns. *The learned alignment might be imprecise*, making those NMT-based models *incorrectly identify the fixing locations* in the new buggy code.

The second issue of NMT-based APR models is with the sequence-based representation for source code. Source code has well-defined syntax and semantics. The sequence representations and sequence-based DL models are not suitable for capturing code structures. Thus, the process of learning the mappings between the program elements/tokens in the buggy and fixed code must implicitly recover the code structure and map two structures to learn the code transformations corresponding to the bug fix. This could lead to imprecise mappings and eventually incorrect fixes.

The third issue of NMT-based APR models is with how they handle *the context of the code surrounding the fixing locations* where the fixes occur. In general, the fix at a specific location depends on the surrounding code. For example, assuming that

a bug fix of adding an operation of closing a file with *FileOutputStream.close* occurs in the context of the file object being already open and written to. A file closing operation is needed after that. An addition of *FileOutputStream.open* cannot be a candidate fix because the file was already open. The state-of-the-art approaches [41, 158, 156] using NMT to translate buggy to good code currently have limitations in considering such context. Specifically, they often take the entire methods before and after the fixes as the pairs to train a sequence-to-sequence NMT model. The issue is that the context of entire method contains too much noise that makes the model difficult to correctly align the changed and unchanged code between the buggy and the fixed code. At the meantime, other approaches [24, 21, 46] limit the scope of the input code only to the buggy statement and the corresponding fixed one. With this, the models avoid the noise caused by too much context, however, facing the opposite issue that they lack the contexts to derive the correct fix. Thus, they often pick and rank higher the popular yet incorrect fixes in the corpus, with limited consideration of surrounding code.

To address those challenges, we introduce DLFix, a two-layer tree-based deep learning model to learn code transformations from prior bug fixes to apply to fix a given buggy code. We treat the APR problem as *code transformation learning* (rather than a machine translation problem), in which transformations corresponding to bug fixes including (un)-changed parts are encoded as the input for model training. This helps DLFix to avoid the mapping task. Instead of using sequence-to-sequence NMT, we use a tree-based RNN in which the source code of the method is represented by the corresponding abstract syntax tree (AST), where the changed and un-changed sub-trees are encoded and fed to the model. To address the issue on context, we *separate the learning of the context of surrounding code* of bug fixes from *the learning of the code transformations* for bug fixes with two layers in our model. The buggy sub-tree in the AST of a buggy method is identified and replaced with a summarized

node, which encodes the detailed structures of the buggy sub-tree. The non-buggy AST sub-trees together with the summarized node constitute the context and are learned with the RNN model at the context learning layer. The output of the context learning layer is a vector representing the surrounding context.

For the code transformation learning, the changed sub-tree before and after the fix is used for training another tree-based RNN model to learn the bug-fixing code transformations. The context of the transformation computed as the vector in the context learning layer is used as anadditional input in this step.

The separation of context learning and transformation learning, and using the result from context learning in the latter process helps DLFix sufficiently consider the surrounding code. This strategy also enables DLFix to learn the context better due to the relative position of the summarized node (the changed sub-tree for a fix) with the other nodes (un-changed sub-trees) of the AST. Moreover, during training, the separation between two learning phases helps DLFix avoid the incorrect alignments between changed code and surrounding context code. Only the changed sub-trees before and after fixing are aligned and DLFix learns the transformations.

We conducted several experiments to evaluate DLFix in two standard bug datasets *Defects4J*, and *Bugs.jar*, and in a newly built bug datasets with a total of +20K real-world bugs in 8 large Java projects. We have compared against a total of 13 state-of-the-art pattern-based APR tools. Our results show that DLFix can auto-fix more bugs than eleven of them, and is comparable and complementary to the top two pattern-based APR tools. However, we can fix 7 and 11 new unique bugs compared with the above top two APR tools. Importantly, DLFix is fully automated and data-driven, and does not require hard-coding of bug-fixing patterns as in those tools. We also compared DLFix against four state-of-the-art deep learning (DL)-based APR models. DLFix is able to detect 2.5 times and 19.8 times more bugs than the

best and worst performing baselines, respectively. In this chapter, we contribute the following:

A. DL for APR: DLFix is the first DL APR that generates comparable and complementary results with powerful pattern-based tools, as recently published DL-based APR can only fix very few bugs on Defects4J. DLFix helps confirm that further research on building advanced DL to improve APR is promising and valuable.

B. Model: A novel DL-based APR approach with a novel two-layer tree-based model, effective program analysis techniques, and CNN-based re-ranking approach to better identify correct patches.

C. Empirical Results:

1) **Comparable and Complementary to Pattern-Based APR**. We show DLFix can auto-fix more bugs than 11/13 state-of-the-art pattern-based APR tools, and its result is comparable and complementary to the ones from the two best pattern-based tools. DLFix do not require hard-coding of bug-fixing patterns as in those tools. DLFix is fully automatic and data-driven.

2) **Improving over all the DL-Based APR**. DLFix is able to detect 2.5 times more bugs than the best performing baseline. DLFix can fix 253 new bugs (out of 1158 in Bugs.jar) than all the other DL-based APR techniques combined.

4.2.2 Motivation

4.2.2.1 Motivating example. In this sub-section, we present a real-world example and our observations to motivate our approach.

Figure 4.2 shows an example of a real-world bug fix in the project *PIG* in the *Bugs.jar* dataset [133]. In the method *deleteDir*, as part of the task of deleting a directory, the string for the command is first built (lines 5–8). A bug occurs when the API call *runCommand* needs to have the third argument as $false$ (an option to indicate no connection to a socket), and it does not need to return an object *Process*.

```
1  private void deleteDir(String server, String dir) {
2     if (server.equals(LOCAL)){ ...
3     }else {
4        String[] cmdarray = new String[3];
5        cmdarray[0] = "rm";
6        cmdarray[1] = "-rf";
7        cmdarray[2] = dir;
8        try {
9  -        Process p = runCommand(server, cmdarray);
10 +        runCommand(server, cmdarray, false);
11       } catch(Exception e){
12       log.warn("Failed to remove ..." + dir);
13       }
14    }
15    ...
16 }
```

Figure 4.2 Automated Program Repair: A bug-fixing example from project *PIG* in *Bugs.jar*.

The fix is shown at line 11 where the variable p of *Process* was deleted and the new argument $false$ was added.

From this example, we have drawn the following observations:

Observation 1 (Code Structure). The change to fix a bug could range from a simple change to a program entity, to a complex transformation of the code structure in the buggy code. For example, in Figure 4.2, the fix involves the removal of the variable p and the declaration type *Process*, and the addition of the third argument $false$. In other words, the structure of the code statement was changed. In other cases, the changes might be more complex. Generally, *a bug-fixing change can be viewed as a code transformation applying to the buggy code to transform it to the correct code.*

Observation 2 (Context). A bug fix often depends on the context of its surrounding code. That is, bug fixes might vary for different contexts of surrounding code. For example, the fix in Figure 4.2 makes sense in the context of the surrounding code when the string command was built (lines 5-8), and the API *runCommand* was called (line 10), and an exception catching occurs afterward at lines 12–14. The fix might not be the same in a different usage of those objects. Generally, *the decision to choose a specific bug-fixing change needs to consider the context of surrounding code where the fix occurs.*

Table 4.1 Automated Program Repair: Results from Different NMT-Based Approaches on Figure 4.2.

Models	Fixed Results
NMT_M [156, 158]	$log.warn(\text{"}Failed\ to\ remove...\text{"}); (line 13)$
NMT_S [46, 21]	$Process\ cmdarray{=}runCommand(server, dir);$

Notes: NMT: Neural Machine Translation. NMT_M: NMT at method-level. NMT_S: NMT at statement-level.

The state-of-the-art approaches for automated program repair, particularly the ones that rely on statistical machine translation (SMT) have dealt with code structure and bug-fixing context in different ways with different results. *NMT_M* [156, 158] uses a neural network-based machine translation model (NMT) that considers an entire method as a sentence consisting of words and learns to translate buggy code into correct code.

While using the entire method body as the context, *NMT* and generally, SMT-based APR approaches [156, 158] have an important limitation in the way that *they treat automated bug fixing as a machine translation problem: during training, a NMT-based model needs to implicit learn the alignments of code elements in a pair of buggy code and its fixed code in order to learn the fixing changes to apply to a new buggy code.* During training, the bug-fixing changes between the buggy and fixed code are

known. Instead of equipping a model with the knowledge on such transformations, SMT-based APR approaches [156, 158] do not use the knowledge on the changes and let a model learn the alignments between the buggy and fixed code. Thus, the model might incorrectly align the similar code in different positions in the same method to one another. For example, the lines containing the variable *cmdarray* could be incorrectly aligned to one another in the two versions before and after the fix. Incorrect alignment could lead to incorrectly identifying the fixing locations. As seen in Table 4.1, *NMT_M* fixes the statement at line 13.

Another issue is that *SMT-based model treats source code as sequences of words* for translation. Source code has well-defined structure and semantics. Bug fixes involve the transformations in code structure, which are disregarded during learning the alignments between the buggy and fixed code in the existing SMT-based approaches. Such mappings between the code structures of buggy and fixed code might not be correctly learned by such models, leading to imprecise alignments and inaccuracy in translation.

To address the issue of incorrect fixing locations, the state-of-the-art NMT-based APR approach [46, 21] restricts the fixing scope to a statement. For example, a fault localization method [2] could be applied first to localize the fix to the statement at line 10. Then, an NMT model is used to translate the buggy statement into a fixed one. However, by limiting the fixing scope, the model cannot leverage the context of the surrounding code of the bug fixes, often leading to incorrect fixes. Despite code abstractions, those models cannot compensate for the lack of context of a bug fix. For example, for the example, as seen in Table 4.1, keyword NMT_S model made an incorrect fix as in $Process\ cmdarray{=}runCommand(server, dir)$;.

4.2.2.2 Key ideas. From the observations, we draw the following key ideas for DLFix:

[**Transformation Learning**] First, we aim to develop DLFix, a novel deep learning model to learn the code transformations from the previous bug-fixing changes, and apply to fix a given buggy code. The key departure point from the state-of-the-art neural network machine translation (NMT)-based models is that we consider the buggy and fixed code in the same space and during training, we equip our model with the knowledge on code transformations corresponding to prior bug fixes, rather than letting a sequence-to-sequence NMT-based model learn the mappings between the buggy and fixed code. That eliminates the incorrect alignments that could lead to imprecise fixing locations. For example, in Figure 4.2, for training our model, we encode the transformations from the abstract syntax tree (AST) of the fragment $Process\ p = runCommand(server, cmdarray)$; to that of the fragment $runCommand(server, cmdarray, false)$;.

[**Explicit Context Learning**] Second, the context of the code surrounding a fix remains the same after bug fixing. We encode such code structures surrounding the changes into DLFix as contextual information. For example, the code structures surrounding the fix at line 10 are encoded as the context, and such context is treated separately and used as a weight in emphasizing the code transformations from buggy to fixed code. That is our second departure point from existing models that helps DLFix avoid the issues with context. As an NMT_M-based model [156, 158] uses the source code of an entire method as the context for training a translation model, the incorrect alignments could occur. In contrast, an NMT_S model [46, 21] uses only the fixing statement for training faces the issue of lacking the context to correctly learn bug fixes. In comparison with the existing code change learning approach [156], in DLFix, we explicitly represent the context as a vector capturing the surrounding code, while the changes and the contexts are mixed as the input of their model.

[**Patch Re-ranking**] Third, ideally, a correct patch for a bug should be pushed onto the top-1 position (i.e., top-1 recall) in a list of candidate patches, so that it

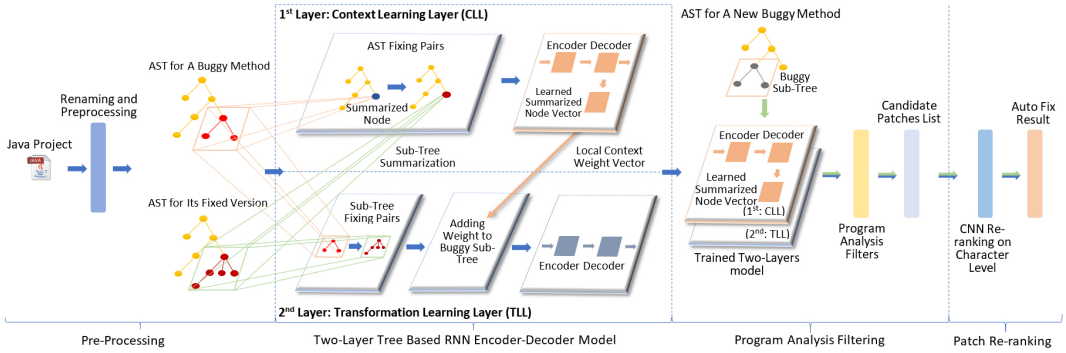


Figure 4.3 Automated Program Repair: Overview of our approach.

can be the first to be picked up in the patch validation phrase. Therefore, during the training of a ranking model, the training target should be the top-1 recall, instead of the relatively ranked positions of a correct patch in a list. Our third departure point from existing approaches is that during the training, given a list of candidate patches containing the ground-truth patch for a bug, we build a CNN-based binary classification approach and train it using the ground-truth patch as one positive example and the rest candidate patches as negative ones. Given a new list of patches, we aim to push the correct patch onto the top of the list. In this motivation example, our first two key ideas can help get a group of possible candidates for auto-fix. Our patch re-ranking helps make the right answer at line 11 be the top-1 patch.

4.2.3 Approach overview

For training, DLFix consists of the following key steps (Figure 4.3).

Step 1: Pre-Processing

To prepare to train DLFix, we take the following steps: first, DLFix performs alpha-renaming on the names of variables within a method of a project. This helps it learn from a method to fix in another method since different methods might use different variables' names despite they have similar structures or functionality. Given a method pair (M_b, M_f), M_b is a buggy method and M_f is M_b's fixed version, DLFix uses Word2Vec [99] on the sequence of code tokens to obtain their vector

representations for each unique code token in M_b and M_f. DLFix identifies the buggy sub-tree (namely T_b^{sub}) within the AST of M_b, and T_b^{sub}'s corresponding changed sub-tree (namely T_f^{sub}) within the AST of M_f. The root of the changed sub-tree is defined as the common ancestor node of all changed and added nodes in the buggy AST and all changed and inserted nodes in the fixed AST. The final task of this step is to obtain a vector for T_f^{sub}, and another vector for T_b^{sub}. We adopted a DL-based code summarization model [163] to summarize a tree into a vector for a node (called *a summarized node*). The obtained individual vectors for T_b^{sub} and T_f^{sub} are used in the next step of our process.

Step 2: Context Learning

We designed a two-layer learning model that learns the code transformations for bug fixes and the context of the code surrounding the fixes. The first one is dedicated for local **C**ontext **L**earning **L**ayer (CLL). To train our model at this layer, we replace the changed sub-tree with a *summarized node* obtained in the previous step. All other AST nodes that were not changed by the fix are kept the same to provide the context of the code surrounding the fix. The vector for the summarized node is obtained as explained earlier and used in this step. Given pre-processed pairs of methods, we develop a tree-based encoder-decoder model using tree-based LSTMs [152] for this local context learning. We compute the vector for the AST of a method with the summarized node in a pair of methods (M_b, M_f). The obtained vectors are used as weights representing context in the next step, which learns the code transformations for bug fixes.

Step 3: Code Transformation Learning

The second layer is dedicated to code **T**ransformation **L**earning **L**ayer (TLL) for bug-fixing changes. In this TTL, the changed sub-tree before and after the fix is used for training to learn the bug-fixing code transformations. Moreover, the context of the transformation computed as the vector in the context learning layer is used an

additional input in this step. Specifically, we use the same tree-based encoder-decoder model in the first layer, CLL, to encode both structural and token changes.

Step 4: Program Analysis Filtering

Next, we setup program analysis filters, including the filter to check the existence of variables, methods, and class names, the filter to convert the keywords back to the right names, and the filter to check the syntax of final result. With these filters, DLFix derives the possible candidate fixes.

Step 5: Patch Re-Ranking

Finally, we use a Convolutional Neural Network (CNN) [61] based binary classification model to re-rank the generated candidate patches. This module performs more analysis on the detailed code contexts for the buggy statement and aims to push the correct patch onto the top of a list. In this step, we re-rank the list of possible patches based on the detailed contextual information, which helps better selecting the results.

For fixing, as in other DL-based models, a new buggy method M_b and a specific fixing location are the inputs. Our trained model processes M_b with the same steps to produce the candidate fixes.

4.2.4 Step 1: Pre-Processing

The goal of pre-processing (Figure 4.4) is to compute the vector representations for the code that has been changed for bug-fixing as well as the code in the context surrounding the fixing changes. Given method pairs and each method pair containing, M_b (i.e., a buggy method) and M_f (M_b's fixed version), to pre-process them for training, DLFix works in four main steps: renaming, AST generation, token vectors learning, and sub-tree summarization.

4.2.4.1 Renaming. The goal of this step is to alpha-rename the variables in a method in order to increase the chance for the model to learn the fix in one place

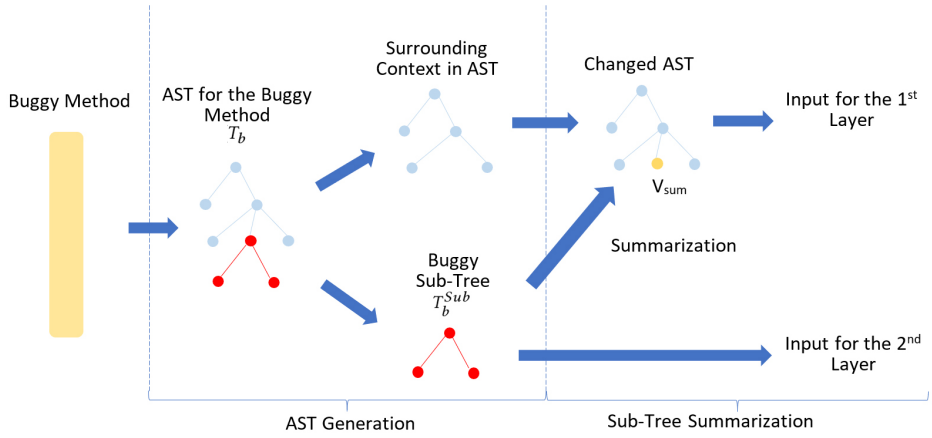


Figure 4.4 Automated Program Repair: Key steps of pre-processing.

to apply to another in the similar/same scenarios because different methods in the same or different projects might use different variable names. In addition, we also keep the type of a variable in the new name to avoid the accidental clashing names. For example, the variable a of the type A in the method M_1 calls the method m (of A) as in $a.m()$. The variable b of the type B in the method M_2 calls the method m (of B) as in $b.m()$. If we do not keep the types A and B of a and b during alpha-renaming, $a.m()$ and $b.m()$, which mean two different operations m in two classes, might get accidentally renamed to the same one, e.g., $v.m()$. In this case, $a.m()$ becomes $v[A].m()$, and $b.m()$ becomes $v[B].m()$. We maintain a data structure of the variables and associated information similar to an entity table in a compiler to recover the actual names.

4.2.4.2 AST generation. Next, DLFix generates the ASTs for a method pair, M_b and M_f (i.e., M_b is a buggy method and M_f is M_b's fixed version). Given the fault location information of a method, DLFix detects the root of the changed sub-tree (i.e., the buggy sub-tree) in the AST. It then marks all the nodes under that root. Therefore, for a given method pair, we generate four ASTs: an AST for M_b (namely T_b), an AST for M_f (namely T_f), a buggy sub-tree of T_b (namely T_b^{Sub}), and a changed sub-tree of T_f (namely T_f^{Sub}).

4.2.4.3 Token vectors learning.

Next, we aim to compute the vector representations for all of the code tokens within the method pairs in a project. In this step, we use the source code after alpha-renaming. We consider each statement S in the block of statements within a method as a sentence. We collect all the sentences in all the projects in the training corpus. Then, we use Word2Vec [99] on all the sentences in the corpus to obtain the vectors for all the code tokens.

4.2.4.4 Sub-Tree summarization.

The goal of this step is to represent the changed/buggy sub-tree as a single vector representation, which will be used in the local context learning layer. The rationale of representing the entire changed sub-tree as a single node is that in the context learning layer, we focus on the context of surrounding code of the fixing changes. If we include all the nodes in the changed sub-tree for learning the context, those nodes become noises for such learning. At the same time, we do not want to replace that changed/buggy sub-tree with a dummy node because the changes are important as well. Therefore, we decide to encode the buggy sub-tree (T_b^{Sub}) with a vector as well as the changed sub-tree (T_f^{Sub}). We will use them as additional inputs in the process of learning the transformations from the buggy sub-tree to the fixed one.

To achieve that, we adopted an existing model in [163]. The model is capable of representing a sub-tree with a vector by combining the encoding of the tree structure and that of the sequence of tokens belonging to the sub-tree. The authors use a tree-based RNN model to combine with a regular RNN model with a deep reinforcement learning mechanism. Using that combined model, we obtain the vector called a summarized vector V_{sum} to represent a sub-tree.

4.2.5 Step 2: Local context learning

The goal of the Context Learning Layer, CLL, is to learn local context of the code surrounding the bug-fixing changes (i.e., the unchanged code surrounding the changed

one). Specifically, given a pair of ASTs (T_b and T_f for buggy and fixed methods), DLFix first builds changed T_b and T_f in which the buggy and changed sub-trees (T_b^{Sub} and T_f^{Sub}) are replaced by the summarized nodes. Each summarized node is represented by a vector generated in the Step 1.4 (Sub-Section 4.2.4.4). The learned summarized node vector in CLL is computed as the output of the decoder in the auto-fix phase. In training, the summarized nodes for T_b^{Sub} and T_f^{Sub} are given to train the model parameters. For all other nodes in a changed T_b (or T_f) except the summarized node, each node is represented as a vector generated by using Word2Vec in the Step 1.3 (Sub-Section 4.2.4.3). As the buggy and changed sub-trees (T_b^{Sub} and T_f^{Sub}) have different structures and AST nodes, the summarized node vector of T_b^{Sub} should be different from the one of T_f^{Sub}. During the generation of vector embeddings, we set each Word2Vec vector and a summarized node vector to have the same length.

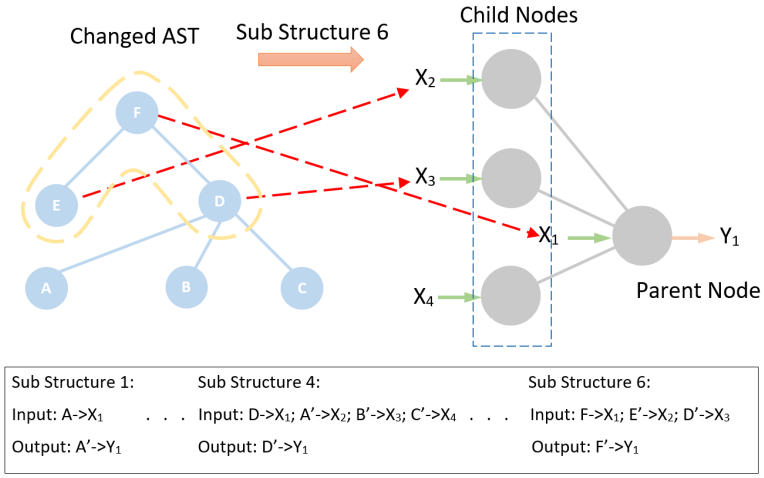


Figure 4.5 Automated Program Repair: Tree-Based LSTM.
Notes: ′ means the encoded vector for the node and will be used for further training, X_2, X_3, and X_4 include hidden result vector h_j and cell state vector c_j

The changed T_b and T_f represents the context of the bug fix for a buggy method. To learn such context, we use the pair of changed T_b and T_f as an input for training an encoder-decoder model with a tree-based RNN model [152] as an encoder and another same tree-based RNN as a decoder. We keep all hidden results from the

encoder-decoder model and use them as the context of a fix. The hidden results will be passed down to the Code Transformation Learning Layer (TLL).

The tree-based RNN model [152] is designed to work with tree-structured data. Unlike the regular RNN model that loops for each time-step, the tree-based RNN loops for each sub structure. In DLFix, we use a tree-based LSTM, especially the Child-Sum Tree-LSTM [152] because of the different numbers of child nodes. It is reasonable to directly use Tree-LSTM to model the code for preserving both code structures and sequential syntax, instead of applying normal RNN just on sequences of code tokens. Figure 4.5 illustrates how the Child-Sum Tree-LSTM models an AST. As seen in Figure 4.5, Child-Sum Tree-LSTM model considers a parent node and its children nodes each time. As for this, we have

$$h^j_{sum} = \sum_{k \in Child(j)} h_k \tag{4.1}$$

$$i_j = \sigma(W_i x_j + U_i h^j_{sum} + b_i) \tag{4.2}$$

$$f_{jk} = \sigma(W_f x_j + U_f h_k + b_f) \tag{4.3}$$

$$o_j = \sigma(W_o x_j + U_o h^j_{sum} + b_o) \tag{4.4}$$

$$u_j = tanh(W_u x_j + U_u h^j_{sum} + b_u) \tag{4.5}$$

$$c_j = i_j \circ u_j + \sum_{k \in Child(j)} f_{jk} \circ c_k \tag{4.6}$$

$$h_j = o_j \circ tanh(c_j) \tag{4.7}$$

Where $Child(j)$ is set of the children nodes of the parent node j; h^j_{sum} is the sum of all hidden results from children nodes; i_j is the input/update gate's activation vector; o_j is the output gate's activation vector; f_{jk} is the forget gate's activation vector; c_j

is the cell state vector; W, U are weight matrices; b are bias vector parameters; σ is sigmoid function; and $\circ$ denotes the Hadamard product. For example, in Figure 4.5, the node A has no child nodes, the node A becomes the input X_1 and the A' becomes the output Y_1 after going through the model. Another example is the node F that has children nodes, the node F becomes the input X_1, the E' node becomes the input X_2, the D' become the input X_3, and F' become the output Y_1 to go through the model. In the example of Figure 4.5, the model first process the nodes A,B, and C in the same level, then goes up to process the nodes E and D in the same level, and lastly, processes the node F. All the nodes go through the model.

4.2.6 Step 3: Code transformation learning

The goal of the Code Transformation Learning layer (TLL) is to learn the code transformations of the bug fix with the additional input information of the context computed by the Context Learning Layer (CLL). We employ a child-sum tree-based LSTM encoder-decoder model that has the same architecture as the one in CLL (Sub-Section 4.2.5) to learn code transformations. To train such a model, we feed the pairs of sub-trees (i.e., a pair contains a buggy sub-tree, T_b^{Sub}, and its corresponding changed sub-tree, T_f^{Sub}) along with their learned context vectors (W_{lc}) into the tree-based encoder-decoder.

Specifically, to integrate the context vector W_{lc} as a weight, we use cross-product multiplication to combine the weight vector with the vector of each AST node in a sub-tree. Cross-Product is better than concatenation of vectors, because the local context and detailed node information contains different kinds of information, and cross-product multiplication is more expressive and effective in combining different kinds of information. Next, after performing cross-product, the results are fed into the child-sum tree-based LSTM encoder-decoder model. Once the tree-based LSTM encoder-decoder model is trained, given a buggy sub-tree, this layer TLL automatically generates the fixed sub-tree for the buggy sub-tree.

Both CLL and TLL use the same loss function as defined in *seq2seq* [16]. Even though we used tree-based RNN, the output of each timestep is in the same vector format as *seq2seq* [16].

4.2.7 Step 4: Program analysis filtering

The goal of the filtering step has two folds. First, DLFix aims to recover the candidate fixes in the source code form, and second, it aims to filter the candidate fixes that violate certain pre-defined program analysis rules. For the first task, DLFix takes the AST_{fix} of a candidate result, and the AST of the entire method as the input, and produces the fixed code. To do that, DLFix first builds the complete AST for the method $\text{AST}_{MethFix}$ after the fix by replacing the buggy sub-tree AST_{sub} with the fixed sub-tree AST_{fix}. It then uses the Word2Vec vectors computed in the token vectors learning step (Sub-Section 4.2.4.3) to find the most likely candidates for each token in the $\text{AST}_{MethFix}$. Next, DLFix uses the data structure similar to the entity table that it maintains to reverse the alpha-renaming process. Finally, it obtains the list of candidate tokens of the entire method after the fix.

For the second task, we use a set of filters to verify program semantics. First, the syntax checking filter is used. If there is a syntax error, we will go back to change the buggy token to the next possible token and check it again. The second filter is to validate the names of the variables, methods, and classes in the project. If a name is not valid, DLFix goes back to use the next most likely token in the candidate list at the position and the process is repeated until the newly code passes this filter. The third filter aims to verify if a particular name is correctly referred to. We use our entity table for this task. It repeats the same process as before until finding the right candidate. Finally, as a result, DLFix has a list of fix candidates, which will be used as the input for the re-ranking process.

4.2.8 Step 5: Patch re-ranking

The output of the previous step is a list of possible patches. Each patch is an automatically generated code statement for the bug. The goal of this re-ranking step aims to push the correct patch onto the top of the list. To do so, we propose a Convolutional Neural Network (CNN) based binary classification model. Specifically, given a list of patches, we first process each candidate patch into characters. Next, we use Word2Vec to train vectors for each character. A candidate patch is represented as a set of character vectors. Then, we apply CNN [61] containing one Convolutional layer [31], pooling and fully connected layers, and a softmax function, on the character vectors of a candidate patch to classify the candidate patch into correct or incorrect. Last, we re-rank the given list of patches based on their possibilities of being a correct patch. During the training, given lists of patches with ground-truth correct patches, for each list of patches, we classify the ground-truth patch into one group as a positive example and the rest goes into another group as negative examples. The training target is to achieve the highest number of times a correct patch is placed onto the top-1 position in a list. We apply the CNN on the characters of a patch instead of tokens, as to classify a patch, the character-level contexts of a patch can carry more information than the tokens for classification. Empirically, we also tested the CNN on tokens of patches, and the CNN on characters outperforms the CNN on tokens for classifying patches

4.2.9 Empirical evaluation

We conducted several experiments to evaluate DLFix against the state-of-the-art APR approaches. All experiments were conducted on a desktop with a 4-core Intel CPU and a single GTX Titan GPU.

4.2.9.1 Research questions. To evaluate DLFix, we seek to answer the following questions:

RQ1. Pattern-Based Automated Program Repair (APR) Comparative Study. How well does DLFix perform in comparison with the state-of-the-art pattern-based APR approaches?

RQ2. Deep Learning-Based APR Comparative Study. How well does DLFix perform in comparison with the state-of-the-art deep learning-based APR approaches?

RQ3. Sensitivity Analysis. How do various factors affect the overall performance of DLFix in APR?

4.2.9.2 Experimental methodology.

4.2.9.2.1 Datasets. In this research, we evaluate approaches on three different datasets: Defects4J [32], Bugs.jar [133], and BigFix (our newly built dataset). BigFix is built from a public dataset [77] for bug detection. The bug detection dataset contains +4.9 million Java methods, and among them, +1.8 million Java methods are buggy. The bug detection dataset contains corresponding bug fixes for the buggy methods. As in the previous studies [134, 52], we evaluate the approaches on fixing one-line bugs. We use the following steps to process the bug detection dataset to build BigFix. We setup several filters to select the appropriate bugs (one-statement bugs) from all the bug reports of a project. The filters include 1) method check filter, which is used to check if the bug is inside of a method, 2) comment check filter, which is to check if the bug is in code statement instead of in comments, 3) one-hunk bug filter which is to check if the buggy position is only one hunk of code and after fixing, if the fixed code is also only one. If the bug passes all these three filters, we mark it as a bug fix and include into the dataset. In total, we collected +20K method pairs with single-hunk bugs. A method pair contains a buggy method and its fixed version.

4.2.9.2.2 Analysis approaches for RQs. To answer our research questions, we use the following settings.

RQ1. Pattern-Based APR Comparative Study

Comparative Study with Baseline Models. We compare DLFix with 13 pattern-based state-of-the-art APR approaches as listed in Table 4.2. We ran DLFix on the well-known benchmark dataset Defect4J. Specifically, we trained it on the real bug fixes in BigFix and tested it on Defect4J. We ran it on 101 one-statement bugs in Defect4J as same as the ones in *Tbar* [87]. Note that, each of those bugs can be fixed by at least one previous approaches. As in the previous studies [135, 167, 87], we simply take the results reported in the respective papers, since all of the above approaches have already been well evaluated on Defect4J.

For this RQ, we compare DLFix with the pattern-based APR approaches only on Defect4J due to the following main reason. Generally, search-based baseline models take the *Generate-and-Validate* approach. Therefore, they often require test cases to conduct validation on candidate patches one-by-one. However, BigFix has no test cases and the test cases in Bugs.jar are not consistent among projects. Due to the inconsistency of test cases in Bugs.jar and quite often no published code for the above studied pattern-based approaches, it is hard to apply all approaches on Bugs.jar.

Moreover, because those pattern-based APR approaches have two following additional steps, in this experiment only, we added them into DLFix for comparison:

(1) Fault localization (FL): Conceptually, DLFix can employ any fault localization techniques to produce an ordered list of suspicious statements that require fixes. We chose Ochiai algorithm [2, 121], which has been widely used in APR [52, 179, 64, 178, 167, 88]. After Ochiai localizes a buggy line, all of the AST nodes including intermediate ones that are labeled by the parser with that buggy line are collected into an AST's subtree as a replaced subtree.

(2) Patch Validation: Once DLFix generates a ranked list of candidate patches, we use a validation technique [134, 52] to validate each candidate. Once a candidate patch pases all available test cases, DLFix stops and reports the candidate patch for manual investigation. We report the patches that are exactly matched or semantically

equivalent to the ground-truth fixes in Defect4J. Finally, we performed overlapping analysis among the results from the models.

Tuning DLFix. We turned DLFix with the following key hyper-parameters using beam-search, such as the vector length of word2vec (100, 150, 200), learning rate (0.001, 0.005, 0.01), and Epoch size (100, 200, 300). We set a 5-hour running-time limit for DLFix to generate candidate patches and validate them as in SimFix [52].

RQ2. Deep Learning-Based APR Comparative Study

Comparative Study with Baseline Models. We compare DLFix with the following state-of-the-art DL-based APR approaches:

1. **Ratchet.** [46] using sequence-to-sequence NMT model;
2. **Tufano et al. (2018)** [158] using encoder-decoder NMT model;
3. **CODIT** [21] using sequence-to-sequence NMT model with some abstractions on tree structures;
4. **Tufano et al. (2019)** [156] using a code change learning approach adopting NMT with some code abstractions and program analysis filtering.

We used all three datasets Defects4J, Bugs.jar, and BigFix for comparison. Deep learning (DL)-based APR approaches do not contain fault localization and patch validation steps. To have a fair comparison and avoid the bias that fault localization can introduce with its false positives [85], we did not run fault localization and patch validation steps for all DL-based APR approaches. We directly provided correct localization information to all DL-based APR approaches and compared the results using the ground-truth fixes from developers. We recorded the results of the models without fault localization and patch validation.

For a comparison on Defect4J and Bugs.jar, we trained all DL-based APR models using BigFix, and tested them on Defect4J (101 bugs) and Bugs.jar, separately. To compare the models on BigFix, we split BigFix at random into 90% for training and 10% for testing.

Qualitative Analysis. For comparison, we also performed qualitative analysis by comparing results from the models on all three datasets. We verified each patch at top-1 position automatically against the ground-truth patch. We computed how many bugs each model can fix among all one-line bugs, how many bugs that can be fixed by other baseline models were covered by our model, how many bugs our model did not cover, and how many new bugs our model can fix when comparing with other baseline models.

Tuning and evaluation metrics. We performed the same tuning process as in RQ1. As in previous APR works, we use the following metrics for evaluation: *Top K is the number of times that a correct patch is in the ranked list of top K candidate patches.*

RQ3. Sensitivity Analysis of DLFix

We evaluate the impacts of the following three main factors on DLFix's performance: (1) two-layer Tree-Based Encoder-Decoder model, (2) program analysis techniques, and (3) the re-ranking of candidate patches. To do so, we added each element into the model one by one. We conducted our sensitivity analysis on BigFix and use *Top1* metric.

4.2.9.3 Experimental results.

4.2.9.3.1 Results of RQ1 (Pattern-Based APR comparative study). Table 4.2 shows that DLFix can correctly fix 30 bugs and outperform the most recent pattern-based APR approaches on Defect4J, except for *SimFix* and *TBar*.

Comparing with *SimFix*, our fully automatic DLFix without human-crafted fix patterns can generate comparable and complementary results. *TBar* collects all possible fix patterns from the recent APR tools and applies them to fix bugs. Naturally, *TBar* can be considered as a collection of tools and it is reasonable that DLFix fixes fewer bugs than *Tbar* on Defect4J. However, DLFix is data-driven and no hand-crafted pattern is needed.

Table 4.2 Automated Program Repair: RQ1. Comparison with the Pattern-Based APR Baselines on Defect4J.

Project	Chart	Closure	Lang	Math	Mochito	Time	Total	P(%)
jGenProg	0/7	0/0	0/0	5/18	0/0	0/2	5/27	18.5
HDRepair	0/2	0/7	2/6	4/7	0/0	0/1	6/23	26.1
Nopol	1/6	0/0	3/7	1/21	0/0	0/1	5/35	14.3
ACS	2/2	0/0	3/4	12/16	0/0	1/1	18/23	78.3
ELIXIR	4/7	0/0	8/12	12/19	0/0	**2**/3	26/41	63.4
ssFix	3/7	2/11	5/12	10/26	0/0	0/**4**	20/60	33.3
CapGen	4/4	0/0	5/5	12/16	0/0	0/0	21/25	**84.0**
SketchFix	6/8	3/5	3/4	7/8	0/0	0/1	19/26	73.1
FixMiner	5/8	5/5	2/3	12/14	0/0	1/1	25/31	80.6
LSRepair	3/8	0/0	8/14	7/14	1/1	0/0	19/37	51.4
AVATAR	5/12	**8/12**	5/11	6/13	**2/2**	2/3	28/53	50.9
SimFix	4/8	6/8	**9**/13	14/26	0/0	1/1	34/56	60.7
TBar	**9/14**	**8/12**	5/**14**	**19/36**	1/**2**	1/3	**43/81**	53.1
DLFix	5/12	6/10	5/12	12/28	1/1	1/2	30/65	46.2

Notes: P is the probability of the generated plausible patches to be correct.
In the cells, x/y: x means the number of correct fixes and y means the number of candidate patches that can pass all test cases. For example, for DLFix, 65 candidate patches can pass all test cases. However, 30 out of 65 are the correct fixes compared with the fixes in the ground truth.

Qualitative Analysis of RQ1. Figure 4.6 shows the overlapping analysis of the approaches on Defect4J. Due to the page limitation, we only compare with the best APR approaches including *Elixir*, *Avatar*, *SimFix*, and *Tbar*. As seen, DLFix can fix 12, 17, 11, and 7 unique bugs when comparing with *Elixir*, *Avatar*, *SimFix*, and *Tbar*, respectively (i.e., they did not detect those bugs). Specifically, our approach can fix one more new bug, *CL*-10, that cannot be fixed by *Elixir, Avatar, SimFix,* and *Tbar.*

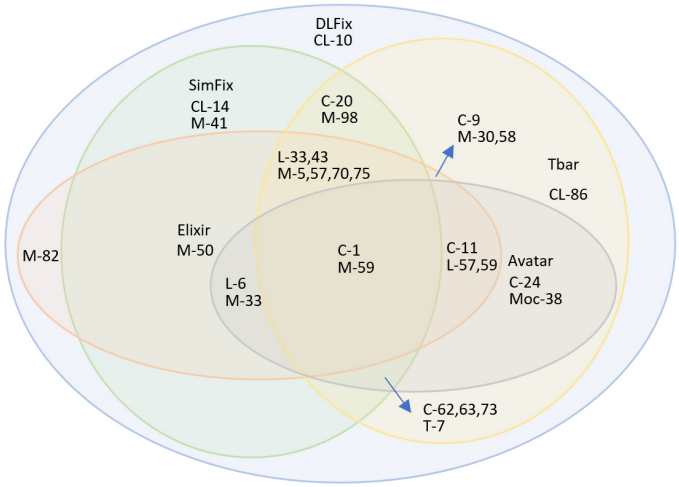


Figure 4.6 Automated Program Repair: Overlapping analysis for RQ1.
Notes: Project names: C:Chart, CL:Closure, L:Lang, M:Math, Moc:Mockito, T:time.

In brief, in comparison with pattern-based approaches, DLFix can obtain comparable results in addition to complementing with them. Furthermore, DLFix is fully automatic and data-driven, and it does not require any human-crafted patterns/templates.

4.2.9.3.2 Results of RQ2 (Deep learning-based APR comparative study).

Table 4.3 shows that DLFix outperforms the state-of-the-art DL-based APR baselines on all three datasets. Our model improves the baselines by 150.6% and up to 1,980.0% in terms of *Top1*. Specifically, in 39.6% of cases on Defect4J, 34.2% of cases on Bugs.jar, and 29.4% of cases on BigFix, the top-1 ranked candidate patch from DLFix is the ground-true patch, meaning that DLFix can directly generate the correct patches for 40 bugs in Defect4J, 396 bugs in Bugs.jar, and 639 bugs in a testing dataset of BigFix. DLFix outperforms the other DL-based models in every metric. For instance, on BigFix, within 10 best guesses, DLFix can achieve 33.4%, and improve the baselines: Ratchet, Tufano *et al.*('18), CODIT, and Tufano *et al.*('19) by 384.1%, 176.0%, 82.5%, and 56.1%, respectively.

Qualitative Analysis of RQ2. Figure 4.7 shows the results of overlapping analysis on three datasets using the *Top1* metric. DLFix can fix more new bugs than any other

baselines. Specifically, it can fix 27, 253, and 291 more new bugs than all of the other four models combined (i.e., the results from Ratchet + Tufano *et al.*('18) + CODIT + Tufano *et al.*('19)) on Defects4J, Bugs.jar, and BigFix, respectively. Also, there are 13, 145, and 349 bugs that can be fixed by DLFix and can also be fixed by at least one baseline.

Table 4.3 Automated Program Repair: RQ2. Accuracy Comparison with DL-based APR approaches on three Datasets.

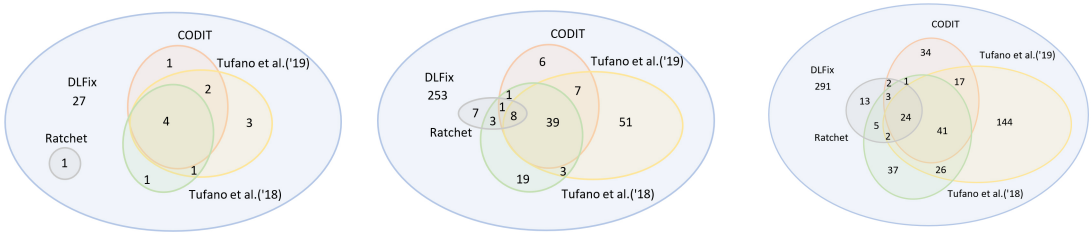
Approach	Defect4J (101 Bugs in Testing)			Bugs.jar (1,158 Bugs in Testing)			BigFix (2,176 Bugs in Testing)		
	Top1	Top5	Top10	Top1	Top5	Top10	Top1	Top5	Top10
Ratchet	2.0%	4.0%	6.9%	2.4%	4.4%	6.8%	3.0%	4.1%	6.9%
Tufano et al.('18)	6.9%	9.9%	11.9%	8.4%	11.1%	12.9%	7.9%	10.6%	12.1%
CODIT	8.9%	13.9%	15.8%	7.0%	11.8%	14.8%	6.9%	13.7%	18.3%
Tufano et al.('19)	15.8%	20.8%	23.8%	13.5%	18.7%	23.1%	15.4%	17.3%	21.4%
DLFix	**39.6%**	**43.6%**	**48.5%**	**34.2%**	**36.4%**	**37.9%**	**29.4%**	**31.1%**	**33.4%**

4.2.9.3.3 Results of RQ3 (Sensitivity analysis). Table 4.4 shows that we build three variants of DLFix with different factors and their combinations. We analyze our results as follows:

(1) **Impact of Two-Layer-EDM**. Our Two-Layer-EDM can improve the one-layer sequence-to-sequence model by 550% and using only *seq2seq* cannot get good results. Two-Layer-EDM is designed to learn the local context of a bug fix and code transformations.

(2) **Impact of PAT**. Using program analysis (PA) techniques, PAT, including alpha-renaming and PA-filtering, is effective to improve Two-Layer-EDM by 109%. The alpha-renaming process can help improve DLFix for better training and the PA-

filtering process can help elminate more irrelevant patches. Adding program analysis to the basic *seq2seq* model can improve it by 256%. However, the Two-Layer-EDM can improve seq2seq by 550%. So the Two-Layer-EDM has more impact than PAT.



(a) Defects4J **(b)** Bugs.jar **(c)** BigFix

Figure 4.7 Automated Program Repair: RQ2. Qualitative analysis results from DL-based APR approaches on three datasets.

(3) **Impact of Re-Ranking**. The results of *Two-Layer-EDM + PAT + Re-Ranking* show that having re-ranking can increase accuracy relatively by 20.5%. The reason is that the re-ranking process, which uses a Convolutional Layer to distinguish the best result from the others, can help increase DLFix's accuracy by pushing the right results to the top of the list of the candidate fixes.

Table 4.4 Automated Program Repair: Sensitive Analysis – Impact of Different Factors on DLFix's Accuracy in terms of Top1 on BigFix Dataset.

Models	Top1	Improvement
Seq2Seq	1.8%	
Seq2Seq + PAT	6.4%	256%
Two-Layer-EDM	11.7%	550%
Two-Layer-EDM + PAT	24.4%	109%
Two-Layer-EDM + PAT + Re-Ranking	29.4%	20.5%

Notes: Seq2seq: a simple sequence-to-sequence model; Two-Layer-EDM: Two-Layer tree-based LSTM encoder-decoder model; PAT: program analysis (PA) Techniques including Renaming and PA Filters.

4.2.10 Discussion and implications

4.2.10.1 In-Depth case studies. Let us present in-depth case studies to show why DLFix can work.

Case Study 1. This case study shows a typical example of a bug using incorrect method call in the code. In Figure 4.8, the incorrect method call $allResultsMatch()$ was changed into $anyResultsMatch()$. DLFix is data-driven and automatically learns a large amount of code transformation patterns from previous fixes. In this way, DLFix can detect to use $anyResultsMatch()$ to replace $allResultsMatch()$. However, the most advanced pattern-based APR approaches, including *Elixir*, *Avatar*, *SimFix*, and *Tbar*, cannot correctly fix this bug, as they rely on human-crafted rules/patterns/templates. If the defined rules/patterns/templates of a tool cannot cover the scenarios of a bug such as the one in this case study, the tool will not be able to automatically fix the bug. In this case, such replacement of a method call requires a pattern-based model to hand-craft the change. For example, *TBar* is so far the best performing APR tool that collects all of the existing patterns in the APR literature, and it still cannot fix this bug. Instead, it attempted to fix the parameters n and $MAY\_BE\_STRING\_PREDICATE$ for the method call $allResultsMatch$. For *Tbar* to be able to fix this bug, one needs to encode in *Tbar* the rule for the code transformation from $allResultsMatch()$ to $anyResultsMatch()$.

We believe that combining the well-defined crafted patterns into deep learning-based APR tools is a promising direction, as the crafted patterns can be used as a seed to automatically learn new, high-quality patterns using DL, which can lead to fixing more bugs.

Case Study 2. Figure 4.9 shows a bug-fix example in which the missing parameter f was added into the method. Because this bug fix requires changes to the structure of the statement, to fix the bug, we need to know which part of the statement structure need to be changed and how to change it. DLFix can fix this bug, because it can

```
static boolean mayBeString(Node n, boolean recurse) {
    if (recurse) {
-       return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
+       return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
    } else {
        ...
    }
}
```

Figure 4.8 Automated Program Repair: Case study 1.
Notes: The Bug-Fix Example from Project *Closure* with the Bug ID *Closure*-10 in Defects4J.

```
public double solve(final UnivariateRealFunction f, double min,
      double max, double initial) throws
      MaxIterationsExceededException, FunctionEvaluationException {
-     return solve(min, max);
+     return solve(f, min, max);
}
```

Figure 4.9 Automated Program Repair: Case study 2.
Notes: The Bug-Fix Example from Project *Math* with the Bug ID *Math*-70 in Defects4J.

learn both the surrounding local context (i.e., unchanged code) of a fix and code transformations of the fix. Importantly, we use Tree-Based RNN to model code and learn local contexts and code transformations from tree structures, so that DLFix can identify that the method call *solve*() needs an additional parameter.

Some recent deep learning baselines, including Ratchet [46], Tufano *et al.*('18) [158], and Tufano *et al.*('19) [156], use sequence-to-sequence translation model (*seq2seq*) to deal with the bug fixing problem. *seq2seq* takes the statement as a sequence, and learns the relationships between tokens. Therefore, *seq2seq* cannot learn the structure changes during the training, even though several algorithms have been applied to transform the code. Due to the lack of their ability in learning structural changes, the

```
1     public Paint getPaint(double value) {
2         double v = Math.max(value, this.lowerBound);
3         v = Math.min(v, this.upperBound);
4 -       int g = (int) ((value - this.lowerBound) / (this.upperBound
5 +       int g = (int) ((v - this.lowerBound) / (this.upperBound
6             - this.lowerBound) * 255.0);
7         return new Color(g, g, g);
8     }
```

Figure 4.10 Automated Program Repair: Case study 3.
Notes: The Bug-Fix Example from Project *Chart* with the Bug ID *Chart*-24 in Defects4J.

existing models cannot fix this bug because they cannot learn the missing parameter of the method call *solve*().

Another DL-based APR approach, CODIT [21], uses the tree structure to learn code transformations. However, it does not directly model code using tree-based DL model. Instead, it learns rules from ASTs and then applies *seq2seq* to learn code changes. CODIT still suffers the lack of ability in learning structural changes. CODIT cannot get the right number of parameters for the method call *solve*. That is the reason that CODIT cannot fix this bug.

Case Study 3. Figure 4.10 shows a bug-fix example from Defect4J. There are two changes in this fix: 1) the variable *value* was changed to v, and 2) the expression in the denominator was modified. We can see that the alpha-renaming and PA-filtering process is useful in this example because PA-filtering enables DLFix to consider the valid variable options at that fixing location and alpha-renaming helps with the renaming the variable into the correct one. Importantly, our tree-based LSTM model helps with the recognition of the code structure and helps the modification of the expression in the denominator of the right-hand side of the assignment.

For this example, the sequence-to-sequence translation models could easily produce a syntactically incorrect fix because they do not consider the code structure.

Moreover, they might not be able to rename the variable *value* because they do not have the program analysis component in their solution.

4.2.10.2 Limitations of our approach. Through the manual analysis of the results from DLFix, especially the bugs that it cannot fix, we identify the following limitations:

1. DLFix does not work well on very unique bugs. DLFix is the deep learning-based approach that needs a large amount of data for training. But even we have a very large dataset, very unique bugs can still exist. Therefore, if there is not sufficiently similar data in the training, DLFix cannot fix the unique bug well.
2. DLFix does not work well on multiple bugs in one method. Our approach can only deal with one bug at a time.
3. DLFix only works on one statement bugs and the bug and fix locations have to be the same.

4.2.11 Threats to validity

Programming language (PL). Our approach has been tested on Java program repair. However, the techniques used in DLFix are not tied to Java. In principle, our approach can applied to other PLs.

Generalization of the results. Our comparisons with pattern-based APR approaches were only carried out on the Defects4J dataset, which is a widely used benchmark for APR research. Further validation of the comparisons with pattern-based APR baselines on other datasets should necessarily be done in future.

Implementation of the deep learning-based baseline models. We re-implemented CODIT as its code and data is not publicly available. We tried our best to follow steps in CODIT. However, some implementation details are not mentioned in their paper, which makes that our version of CODIT could be slightly different from the one in the original paper. However, we tried our best to build and tune the CODIT on our dataset and this is the best effort we can make when the code is not

publicly available. We tuned DLFix and CODIT both on our dataset. Therefore, the comparison in our study is fair for both.

4.3 Automated Program Repair for Multi Statement Multi Hunk Bugs

4.3.1 Introduction

Researchers have proposed several approaches to help developers in automatically identifying and fixing the defects in software. Such approaches are referred to as *automated program repair* (APR). The APR approaches have been leveraging various techniques in the areas of *search-based software engineering*, *software mining*, *machine learning (ML)*, and *deep learning (DL)*.

For *search-based approaches* [68, 70, 96, 126], a search strategy is performed in the space of potential solutions produced by mutating the buggy code via operators. Other approaches use software mining to *mine and learn fixing patterns* from prior bug fixes [60, 67, 86, 87, 112] or similar code [114, 129]. Fixing patterns are at the source code level [86, 87] or at the change level [167, 53, 64]. *Machine learning* has been used to mine fixing patterns and the candidate fixes are ranked according to their likelihoods [92, 90, 134]. While some DL-based APR approaches learn similar fixes [41, 168, 169], other ones use machine translation or neural network models with various code abstractions to generate patches [21, 24, 46, 158, 138, 156, 74].

Despite their successes, the state-of-the-art DL-based APR approaches are still limited in fixing the *general defects, which involve the fixing changes to multiple statements in the same or different parts of a file or different files* (which are referred to as *hunks*). None of existing DL-based approaches can automatically fix the bug(s) with dependent changes to multiple statements in multiple hunks at once. They supports fixing only individual statements. If we use such a tool on the current statement, the tool treats that statement as incorrect and treats the other statements as correct. This does not hold since to fix the current statement, the

remaining unfixed statements must not be treated as correct code. Thus, it might be inaccurate when using existing DL-based APR tools to fix individual statements for multi-hunk/multi-statement bugs. While DL provides benefits for fix learning, this limitation makes the DL-based APR approaches less capable than the other directions (search-based and pattern-based APR), which support multiple-statement fixes.

In this research topic, we aim to advance deep learning-based APR by introducing DEAR, a DL-based model that supports *fixing for the general bugs with dependent changes at once to one or multiple buggy statements belonging to one or multiple buggy hunks of code*. To do that, we make the following key technical contributions.

First, we develop a *fault localization (FL) technique for multi-hunk, multi-statement bugs that combines traditional spectrum-based FL (SBFL) with DL and data-flow analysis*. DEAR uses a SBFL method to identify the ranked list of suspicious buggy statements. Then, it uses that list of buggy statements to *derive the buggy hunks that need to be fixed together* by fine-tuning the pre-trained BERT model [33], to learn the fixing-together relationships among statements. We also design an expansion algorithm that takes a buggy statement s in a hunk as a seed, and expands to include other suspicious consecutive statements around s. To achieve that, we use an RNN model to classify the statements as buggy or not, and use data-flow analysis for adjustment and then form the buggy hunks.

Second, after the expansion step, we have identified all the buggy hunk(s) with buggy statement(s). We develop a *compositional approach to learning and then generating multi-hunk, multi-statement fixes*. In our approach, from the buggy statements, we use a *divide-and-conquer strategy* to learn each subtree transformation in Abstract Syntax Tree (AST). Specifically, we use an AST-based differencing technique to derive the fine-grained, AST-based changes and the mappings between

buggy and fixed code in the training data. Those fine-grained subtree mappings help our model avoid incorrect alignments of buggy and fixed code, thus, is more accurate in learning multiple AST subtree transformations of a fix.

Third, we have enhanced and orchestrated a tree-based, two-layer Long Short-Term Memory (LSTM) model [74] with *an attention layer and a cycle training* to help DEAR to learn the proper code fixing changes in the suitable context of surrounding code. For each buggy AST subtree identified by our fault localization, we encode it as a vector representation and apply that LSTM model to derive the fixed code. In the first layer, it learns the fixing context, i.e., the code structures surrounding a buggy AST subtree. In the second layer, it learns the code transformations to fix that buggy subtree using the context as an additional weight.

Finally, there might be likely multiple buggy subtrees. To build the surrounding context for each buggy subtree B, in training, we include the AST subtrees *after* the fixes of the other buggy subtrees (rather than those buggy subtrees themselves). The rationale is that the subtrees after fixes actually represent the correct surrounding code for B. (Note: in training, the fixed subtrees are known).

We conducted experiments to evaluate DEAR on three datasets: *Defects4J* [32] (395 bugs), *BigFix* [74] (+26k bugs), and *CPatMiner dataset* [110] (+44k bugs). The baseline DL-based approaches include DLFix [74], CoCoNuT [94], SequenceR [24], Tufano19 [156], CODIT [21], and CURE [54]. DEAR fixes 31% (i.e., +11), 5.6% (i.e., +41), and 9.3% (i.e., +31) more bugs than the best-performing baseline CURE on all three datasets, respectively, using only Top-1 patches and with seven times fewer training parameters on average. On Defects4J, it outperforms those baselines from 42%–683% in terms of the number of fixed bugs. On BigFix, it fixes 31–145 more bugs than those baselines with the top-1 patches. On CPatMiner, among 667 fixed bugs from DEAR, there are 169 (25.3%) multi-hunk/multi-statement ones. DEAR fixes 71, 164, and 41 more bugs, including 52, 61, and 40 more multi-hunk/multi-

statement bugs, than existing DL-based APR tools CoCoNuT, DLFix, and CURE. We also compared DEAR against 8 state-of-the-art pattern-based APR tools. Our results show that DEAR generates comparable and complementary results to the top pattern-based APR tools. On Defects4J, DEAR fixes 12 bugs (out of 47) including 7 multi-hunk/multi-statement bugs that the top pattern-based APR tool could not fix.

In brief, our contributions in the work include

A. Advancing DL-based APR for general bugs with multi-hunk/multi-statement fixes: DEAR advances DL-based APR for general bugs. We show that DL-based APR can achieve the comparable and complementary results as other APR directions.

B. Advanced DL-Based APR Techniques:

1) A *novel FL technique* for multi-hunk, multi-statement fixes that combines spectrum-based FL with DL and data-flow analysis;

2) A *compositional approach* with a *divide-and-conquer strategy* to learn and generate multi-hunk, multi-statement fixes; and

3) The design and orchestration of the two-layer LSTM model with the enhancements via the attention layer and cycle training.

C. Extensive Empirical Evaluation: 1) DEAR outperforms the existing DL-based APR tools; 2) DEAR is the first DL-based APR model performing at the same level in terms of the number of fixed bugs as the state-of-the-art, pattern-based tools and generate complementary results.

4.3.2 Motivation

4.3.2.1 Motivating example. Let us present a bug-fixing example and our observations for motivation. Figure 4.11 shows an example of a bug in *verifyUserInfo*, which verifies the given user ID, password and Social Security Number against users' records in the database. This bug manifests in three folds. First, the developer forgot to handle the case when *UID* is *null*. Thus, for fixing, (s)he added an *else* branch at

```
1  public boolean verifyUserInfo(String UID, String password, String SSN) {
2      String retrieved_password = "";
3      String retrieved_SSN = "";
4      if (UID != null) {
5  -       retrieved_password = getPassword(UID);
6  +       retrieved_password = getPassword(toUpperCase(UID));
7  +   } else {
8  +       return false;
9  +   }
10 -   boolean password_check= compare(password,retrieved_password);
11 +   boolean password_check= compare(passwordHash(password),retrieved_password);
12     if (password_check) {
13         retrieved_SSN = getSSN(UID);
14         boolean SSN_check = compare(SSN, retrieved_SSN);
15         if (SSN_check) {
16             return true;
17         }
18     }
19     return false;
20 }
```

Figure 4.11 Automated Program Repair: A general fix with multiple dependent changes.

the lines 7–9. Second, the developer forgot to perform the uppercase conversion for the *UID*, causing an error because the records for user IDs in the database all have capital letters. The corresponding bug-fixing change is the addition of the call to *toUpperCase()* on *UID* at line 6. Third, because the passwords stored in the database are encoded via hashing, the input *password* from a user needs to be hashed before it is compared against the one in the database. Thus, the developer added the call to *passwordHash()* on *password* before calling the method *compare()* at line 11. From this example, we have the following observations:

Observation 1 [A Fix with Dependent Changes to Multiple Statements]: This bug requires the *dependent fixing changes to multiple statements at once in the same fix*: 1) adding the *else* branch with the *return* statement (lines 7–9), 2) adding *toUpperCase* at line 6, and 3) adding *passwordHash* at line 11. Making changes to the individual statements one at a time would not fix the bug since both the given arguments *UID* and *password* need to be properly processed. *UID* needs to be null checked and capitalized, and *password* needs to be hashed. Those dependent changes to multiple statements must *occur at once in the same fix* for the program to pass the test cases.

The state-of-the-art DL-based APR approaches [24, 74] *fix one individual statement at a time*. In Figure 4.11, the fault localization tool returns two buggy lines: line 5 and line 10. Assume that such a DL-based APR tool is used to fix the statement at line 5. It will make the fixing change to the statement at line 5 (e.g., modify line 5 and add lines 7–9), however, with the assumption that the statement at line 10 and other lines are correct. With this incorrect assumption, such a fix will not make the code pass the test cases since both changes must be made. Thus, the *individual-statement, DL-based APR tools cannot fix this bug by fixing one buggy statement at a time*. In general, *a bug might require dependent changes to multiple statements (in possibly multiple hunks) in the same fix*.

Moreover, the pattern-based APR tools might not be able to fix this defect because the code in this example is project-specific and might not match with any bug-fixing patterns.

Observation 2 [Many-to-Many AST Subtree Transformations]: A fix can involve the changes to *multiple* subtrees. For example, the *if* statement has a new *else* branch. The argument of the call to *getPassword()* was modified into the call to *toUpperCase()*. This fix also involves *many-to-many subtree transformations*. In this example, a fix transforms the two buggy statements (line 5 and line 10), into four

statements (the *if* statement having new *else* branch, the *return* statement at line 8, the modified statement with *toUpperCase* at line 6, and the modified statement with *passwordHash* at line 11). Thus, *a fix can be broken into multiple subtree transformations*, and if using a *composition approach with a divide-and-conquer strategy*, we can learn the individual transformations.

Observation 3 [Correct Fixing Context]: *A bug fix often depends on the context of surrounding code.* For example, to get the password from a given *UID*, one needs to capitalize the ID, thus, in correct code, the method call to *toUpperCase* is likely to appear when the method call to *getPassword* is made. Therefore, *building correct fixing context is important.* In Figure 4.11, a model needs to learn the fix (line 5 $\rightarrow$ line 6) w.r.t. surrounding code, which needs to include the *fixed code* at line 11 (rather line 10 because line 10 is buggy). To fix line 5, the correct context must include *passwordHash* at line 11. Thus, the correct context for a fix to a buggy statement must include the fixed code of another buggy statement s', rather than s' itself.

4.3.3 Key ideas

From the observations, we draw the following key ideas:

Key Idea 1. A Fault Localization Method for Multi-Hunk, Multi-Statement Patches: From Observation 1, we design a novel FL method that *combines traditional spectrum-based FL (SBFL) with DL and data-flow analysis.* We use a SBFL to obtain a ranked list of candidate statements to be fixed with their suspiciousness scores. We extend the result from SBFL in two tasks. First, we design a ***hunk-detection algorithm*** to use DL to detect the hunks that need *to be dependently changed together in the same patch*, because SBFL tool returns the suspicious candidates for the fault, but not necessarily to be fixed together. Second, we design ***an expansion algorithm*** that takes each of those detected fixing-together hunks and expands it to include consecutive suspicious statements in the hunk. In

Figure 4.11, the SBFL tool returns line 5 as suspicious. After hunk detection, DEAR uses data dependencies via variable *retrieve_password* to include the statement at line 10 as to be fixed as well.

Key Idea 2. A Compositional Approach to Learning and Generating Multi-Hunk, Multi-Statement Fixes:

Divide-and-Conquer Strategy in Learning Multi-Hunk/Multi-Stmt Fixes. To auto-fix a bug with multiple statements, a tool needs to make m-to-n statement changes, i.e., m statements might generally become n statements after the fix. A naive approach would let a model learn the code structure changes and make the alignment between the code before and after the fix. Because a fix involves multiple subtree transformations (Observation 2), during training, a model might incorrectly align the code before and after the fix, thus, leading to incorrect learning of the fix. For example, without this step, the model might map *retrieved_password* at line 10 to the same variable at line 6 (the correct map is line 11). Thus, to facilitate learning bug-fixing code transformations, during training, we use *a divide-and-conquer strategy*. We integrate into DEAR a fine-grained AST-based change detection model to map the ASTs before and after the fix. Such mappings enable DEAR to learn the *more local fixing changes* to subtrees. For example, the fine-grained AST change detection can derive that the statements at lines 4–5, and 7 become the statements at lines 4, and 6–9; and the statement at line 10 becomes the one at line 11. We can break them into two groups and align the respective AST subtrees for DEAR to learn.

Compositional Approach in Fixing Multiple Subtrees. We support the fixes having multiple statements in one or multiple hunks by enhancing the design and orchestration of a tree-based LSTM model [74] to *add an attention layer and cycle training* (Sub-Section 4.3.5.3). While that model fixes one subtree at a time, we need to enhance it to fix multiple AST subtrees at once.

Specifically, we modify its operations in the two-layers to consider multiple buggy subtrees at once. For example, during training, we mark each of the AST subtrees of the statements at line 5 and line 10 before the fix as buggy. At the first layer, for each subtree for a buggy statement, we *replace* it with a pseudo-node, and consider the new AST with its (pseudo-)nodes as the fixing context for the buggy statement. The pseudo-node is computed via an embedding technique to capture the structure of the buggy statement (Sub-Section 4.3.5.2). At the second layer, DEAR learns the transformation from the subtree for the statement at line 5 into the subtree for the fixed statements at the lines 6–9. The vector for the fixing context learned from the first layer is used as a weight in the code transformation learning in the second layer. We repeat the same process for every buggy statement. For fixing, we perform the composition of the fixing transformations for all buggy statements at once.

Key Idea 3. Transformation Learning with Correct Surrounding Fixing Context: To learn the correct context for a fix to a statement, we need to *train the model* with *the fixed versions of the other buggy statements* (Observation 3). For example, for training, to learn the fix to the statement at line 5 with *toUpperCase*, a model needs to integrate the fixed version of the other buggy line, i.e., the code at line 11 with *passwordHash* as the fixing context (instead of the buggy line 10). If the surrounding code before the fix is used (i.e., line 10), the model will learn the incorrect context to fix the line 5.

4.3.4 Approach overview

4.3.4.1 Training process. The input for training includes the source code before and after a fix (Figure 4.12), which is parsed into ASTs. The output includes the two *trained models for context learning* and *for tree transformation learning* (fixing). The context learning model (CTL) aims to learn the weights (representing the impact of the context) to make an adjustment to the tree transformation

learning result. The tree transformation learning model (TTL) aims to learn code transformation for the fix to a buggy AST subtree.

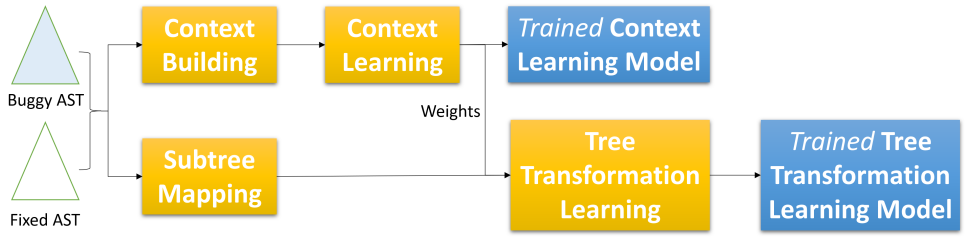


Figure 4.12 Automated Program Repair: Training process overview.

Context Learning (Sub-Sections 4.3.5.2–4.3.5.3). The first step is to build the before-/after-fixing contexts for training. With divide-and-conquer strategy, we use CPatMiner [110] to derive the *changed, inserted*, and *removed* subtrees (key idea 2). As a result, the AST subtrees for the buggy statements are mapped to the respective fixed subtrees. For each buggy subtree and respective fixed subtree, we build two ASTs of the entire method as contexts, one before and one after the fix, and use both of them for training at the input layer and the output layer of the tree-based LSTM context learning model (Sub-Section 4.3.5.3). To build the correct context for each buggy subtree, we leverage key idea 3: we train our model with the fixed versions of the other buggy subtrees. Finally, the vectors computed from this learning are used as the weights in tree transformation learning.

Tree Transformation Learning (Sub-Section 4.3.5.4). We first use CPatMiner [110] to derive the subtree mappings. To learn bug-fixing tree transformations, each buggy subtree T itself and its fixed subtree T' after the fix are used at the input layer and the output layer of the second tree-based LSTM for training. Moreover, the weight representing the context computed as the vector in the context learning model is used an additional input in this step.

4.3.4.2 Fixing process.

Figure 4.13 illustrates the fixing process. The input includes the buggy source code and the set of test cases.

Fault Localization and Buggy-Hunk Detection (Sub-Section 4.3.6.1). From key idea 1, we first use a SBFL tool to locate buggy statements with suspiciousness scores. Hunk detection algorithm uses those statements to derive the buggy hunks that need to be fixed together.

Multi-Statement Expansion (Sub-Section 4.3.6.2). Because SBFL might return one statement for a hunk, we aim to expand to potentially include more consecutive buggy statements. To do so, we combine RNN [25] and data-flow analysis to detect more buggy statements.

Tree-Based Code Repair (Sub-Section 4.3.6.2.6). For the detected buggy statements from multi-statement expansion, we use key idea 2 to derive fixes to multiple buggy subtrees at once. For a buggy subtree T, we build the AST of the method as the context, and use it as the input of the trained context learning model (CTL) to produce the weight representing the impact of the context. The buggy subtree T is used as the input of the trained tree transformation model (TTL) to produce the context-free fixed subtree T'. Finally, that weight is used to adjust T' into the fixed subtree T'' for a candidate patch. We apply grammatical rules and program analysis on the current candidate code to produce the fixed code. We re-rank and validate the fixed code using test cases in the same manner as in DLFix [74].

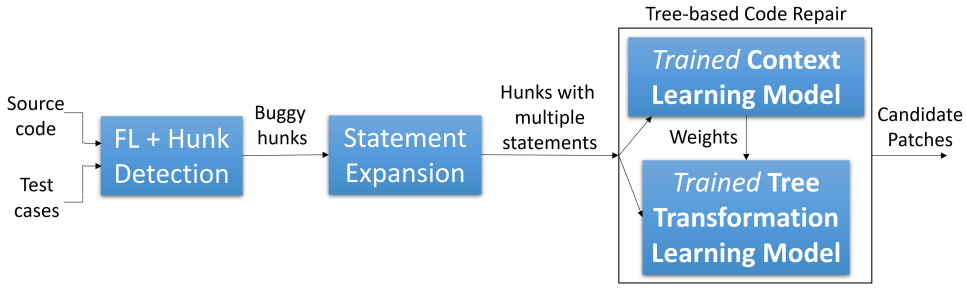


Figure 4.13 Automated Program Repair: Fixing process overview.

4.3.5 Training process

4.3.5.1 Pairing buggy and fixed subtrees. The training data contains the pairs of the source code of the methods before and after the fixes. Note that a fix

might involve multiple methods. Instead of pairing the entire buggy method with the fixed one, we use a divide-and-conquer strategy to help the model to better learn the fixing transformations in the proper contexts. First, we use the CPatMiner tool [110] to derive the fixing changes.

If a subtree corresponds to a statement, we call it *statement subtree*. From the result of CPatMiner, we use the following rules to pair the buggy subtrees with the corresponding fixed subtrees:

1. A buggy subtree (S-subtree) is a subtree with *update* or *delete*.

2. If a S-subtree is *deleted*, we pair it with an empty tree.

3. If a buggy S-subtree is marked as *update*, (i.e, it is *updated* or its children node(s) could be *inserted, deleted* or *updated*), we paired this buggy S-subtree with its corresponding fixed S-subtree.

4. If a S-subtree is *inserted* and its parent node is another S-subtree, we pair it with that parent S-subtree. If the parent node is not an S-subtree, we pair an empty tree to the corresponding inserted S-subtree.

4.3.5.2 Context building. Figure 4.14 illustrates our context building process. For each pair of the buggy AST I_1 and fixed AST O_1 (Sub-Section 4.3.5.1), we perform alpha-renaming on the variables. In Step 1, we encode each AST node with the vector using the word embedding model GloVe [122] (which captures well code structure) by considering a statement node as a sentence and each code token as a word. We use those vectors to label the AST nodes in I_1 and O_1. The ASTs after this step are the vectorized ASTs I_2 and O_2, before and after the fix.

In Step 2, we process each pair of the buggy S-subtree T_b in I_2 and the corresponding fixed S-subtree T_f in O_2. First, we perform node summarization on T_b and T_f by using TreeCaps [50] to capture the tree structures of T_b and T_f into V_s and V_s', respectively. Second, for each of the other buggy S-subtrees, e.g., T_b', and their corresponding fixed S-subtrees, e.g., T_f', we process as follows. Because T_f' is the

fixed version of T'_b, we replace T'_b with T'_f in the building of the resulting context I_3 before fixing (key idea 3). That is, we replace each of the other buggy subtrees with its fixed version. However, to build the resulting context O_3 after fixing, we keep T'_f because it is the fixed subtree, thus, providing the correct context.

The resulting AST, I_3, is used as the before-the-fix context for the buggy S-subtree T_b and used at *the input layer* of the encoder in the context learning model (CTL). The resulting AST, O_3, is used as the after-the-fix context for T_f and used at *the output layer* of the decoder in CTL (Figure 4.14). Finally, the vectors V_s and V'_s will be used as the weighting inputs for tree transformation learning later.

For context learning and tree transformation learning, we enhance the two-layer, tree-based, LSTM models in DLFix [74] with attention layer and cycle training. We have added an attention layer into that model, which now has 3 layers: encoder layer, decoder layer and attention layer (Figure 4.15). For the encoder and decoder, to learn the fixing context expressed in ASTs, we use Child-Sum Tree-Based LSTM [151]. Unlike the regular LSTM that loops for each time step, this model loops for each subtree to capture structures.

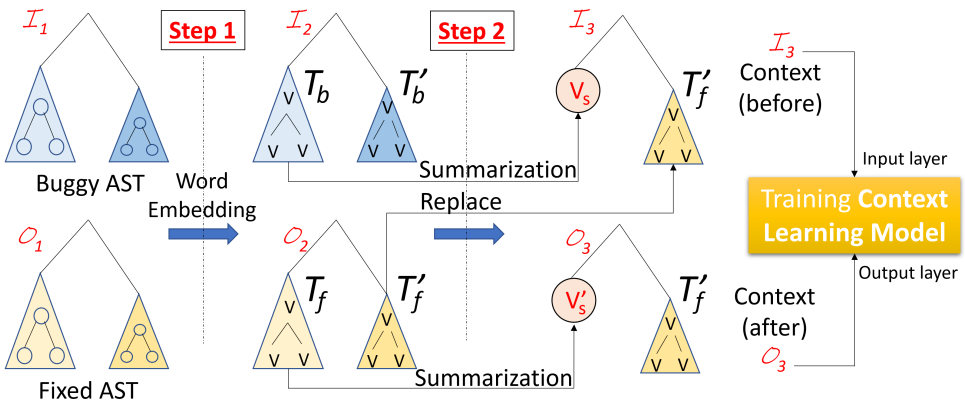


Figure 4.14 Automated Program Repair: Context building to train context learning model.

4.3.5.3 Context learning via tree-based LSTM with attention layer and cycle training.

We also use *cycle training* [199] for further improvement. Cycle training aims to help a model learn better the mapping between the input and output

by continuing to train and re-train to emphasize on the mapping between them. This is helpful in the situations in which a buggy code can be fixed in multiple ways into different fixed code, or multiple buggy code can be fixed into one fixed code. This makes the regular tree-based LSTM less accurate. With cycle training, the pair of an input and the most likely output is emphasized to reduce the noise of such one-to-many or many-to-one relations.

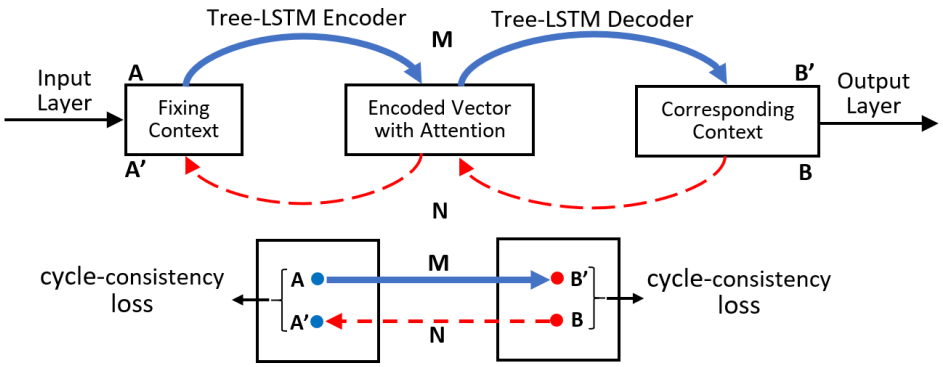


Figure 4.15 Automated Program Repair: Cycle training in attention-based tree-based LSTM.

Cycle training occurs between encoder and decoder. We use the forward mapping M: $A \rightarrow B$ to denote the process of *encoder* $\rightarrow$ *attention* $\rightarrow$ *decoder*, and the backward mapping N: $B \rightarrow A$ to denote the process of *decoder* $\rightarrow$ *attention* $\rightarrow$ *encoder* (Figure 4.15). We apply the adversarial losses for both M and N to get the two loss functions $L_{run}(M, D_B, A, B)$ and $L_{run}(N, D_A, B, A)$. The difference between $N(M(A))$ and A, and that between $M(N(B))$ and B are used to generate cycle-consistency loss $L_{cyc}(M, N)$ for M and N to ensure the learned mapping functions are cycle-consistent. Mathematically, we have two loss functions $L_{run}(M, D_B, A, B)$ and $L_{run}(N, D_A, B, A)$. With the incentive cycle consistency loss $L_{cyc}(M, N)$, the overall loss function is computed as follows:

$$\begin{aligned} L_{cyc}(M, N) = & E_{b \sim p_{data}(b)}[||N(M(a)) - a||_1] \\ & + E_{a \sim p_{data}(a)}[||M(N(b)) - b||_1] \end{aligned} \tag{4.8}$$

$$L(M, N, D_a, D_b) = L_{run}(M, D_B, A, B) + L_{run}(N, D_A, B, A) + \alpha L_{cyc}(M, N) \tag{4.9}$$

Where $L(M, N, D_a, D_b)$ is the loss function for the entire cycle training; M and N are the mapping functions to map A to B and B to A; D_A is aimed to distinguish between the predicted result $N(M(A))$ and the real result A; D_B is aimed to distinguish between the predicted result $M(N(B))$ and the real result B; L_{run} is the cycle consistency loss function for the running function M, N; L_{cyc} is the incentivized cycle consistency loss; and α is the parameter to control the relative importance of the two objectives.

4.3.5.4 Tree transformation learning. Figure 4.16 illustrates the tree transformation learning process. We use the same tree-based LSTM model with attention layer and cycle training as in Sub-Section 4.3.5.3 to learn the code transformation for each buggy S-subtree T_b. In Step 1, we build the word embeddings for all the code tokens as in Sub-Section 4.3.5.2. Each AST node in the buggy S-subtree (T_b) and the fixed one (T_f) is labeled with its vector representation (Figure 4.16). Next, we use the summarized vectors V_s and V_s' computed from context learning in Figure 4.14 as the weights and perform cross-product for each vector of the node in the buggy S-subtree T_b and for each one in the fixed S-subtree T_f, respectively. The two resulting subtrees after cross-product are used at the input and output layers of the tree-based LSTM model for tree transformation learning. We use cross-product because we aim to have a vector as the label for a node and use it as a weight representing the context to learn code transformations for bug fixes.

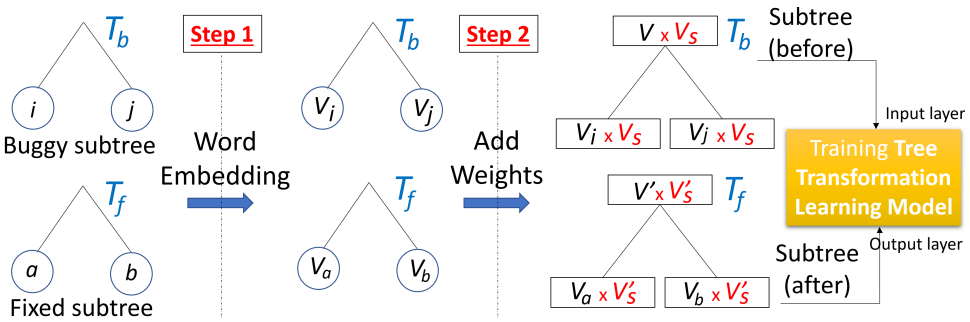


Figure 4.16 Automated Program Repair: Tree transformation learning (V_S, V'_S in Figure 4.14).

4.3.6 Fixing process

4.3.6.1 Fixing-Together hunk detection algorithm. The first step of fixing multi-hunk, multi-statement bugs is for our FL method to *detect buggy hunk(s) that are fixed together in the same patch.* To do that, we fine-tune Google's pre-trained BERT model [33] to learn the *fixing-together relationships among statements* using BERT's sentence-pair classification task. Then, we use the fine-tuned BERT model in an algorithm to detect fixing-together hunks. Let us explain our hunk detection algorithm in details.

4.3.6.1.1 Fine-Tuning BERT to learn fixing-together relationships among statements. We first fine-tune BERT to learn if two statements are needed to be fixed together or not. Let H be a set of the hunks that are fixed together for a bug. The input for the training process is all the sets Hs for all the bugs in the training set.

Step 1 For a pair of hunks H_i and H_j in H, we take every pair of statements S_k and S_l, one from each hunk, and build the vectors with BERT. We consider the pair of statements (S_k, S_l) as being fixed-together in the same patch to fine-tune BERT.

Step 2 Step 1 is repeated for all the pairs of the statements (S_k, S_l)s in all the pairs H_i and H_j in H. We also repeat Step 1 for all Hs. We use them to fine-tune the BERT model to learn the fixing-together relationships among any two statements in all pairs of hunks.

4.3.6.1.2 Using fine-tuned BERT for hunk detection. After obtaining the fine-tuned BERT, we use it in determining whether the hunks of code need to be fixed together or not. The input of of this procedure is the fine-tuned BERT model, buggy code P, and test cases. The output is the groups of hunks that need to be fixed together. The process is conducted in the following steps.

Step 1 We use a spectrum-based FL tool (in our experiment, we used Ochiai [2]) to run on the given source code and test cases. It returns the list of buggy statements and suspiciousness scores.

Step 2 The consecutive statements within a method returned by the FL tool are grouped together to form the hunks H_1, H_2, ..., H_m.

Step 3 To decide if a pair of hunks (H_i,H_j) needs to be fixed together, we use the BERT model that was fine-tuned. Specifically, for every pair of statements (S_k,S_l), one from each hunk (H_i,H_j), we use the fine-tuned BERT to measure the fixing-together relationship score for (S_k,S_l). The fixing-together score between H_i and H_j is the average of the scores of all the pairs of statements within H_i and H_j, respectively. If the average score for all the statement pairs is higher than a threshold, we consider (H_i,H_j) as needed to be fixed together. From the pairs of the detected hunks, we build the groups of the fixing-together hunks. The group of hunks that has any statement with the highest Ochiai's suspiciousness score will be ranked and fixed first. The rationale is that such a group contains the most suspicious statement, thus, should be fixed first.

4.3.6.2 Multiple-Statement expansion algorithm. A detected buggy hunk from the algorithm in Sub-Section 4.3.6.1 might contain only one statement since each of those suspicious statements is originally derived by a SBFL tool, which does not focus on detecting consecutive buggy statements in a hunk. Thus, in this step, we take the result from the hunk detection algorithm, and expand it to include potentially more statements in a hunk.

4.3.6.2.1 Key idea. Our idea is to combine deep learning with data flow analysis. We first train an RNN model with GRU cells [25] (will be explained in Sub-Section 4.3.6.2.3) to learn to decide whether a statement is buggy or not. We collect the training data for that model from the real buggy statements. We then use data-flow analysis to adjust the result. Specifically, if a statement is labeled as buggy by the RNN model, no adjustment is needed. However, even when the RNN model decides a given statement s as *non-buggy*, and if s has a data dependency with a buggy statement, we still mark s as *buggy*.

4.3.6.2.2 Expansion algorithm. The input of Multi-Statement Expansion algorithm is the buggy statement *buggyS*, i.e., the seed statement of a hunk. The output is a buggy hunk of consecutive statements.

First, it produces a candidate list of buggy statements by including N statements before and N statements after *buggyS* (*Expand2NCandidatesList* at line 2). In the current implementation, N=5. Then, it uses the RNN model to act as a classifier to predict whether each statement (except *buggyS*) in the candidate list is buggy or not (*RNNClassifier* at line 3). To train that RNN model, we use the buggy statements in the buggy hunks in the training data (see Sub-Section 4.3.6.2.3). TreeCaps [50] is used to encode the statements.

In *DataDepAnalysis* (line 4), to adjust the results from the RNN model, we obtain *buggyHunk* surrounding the buggy statement *buggyS*, consisting of the statements before and after *buggyS*, that were predicted as buggy by the RNN model (line 7). We then examine statement-by-statement in the upward direction from the center buggy statement in the candidate list (line 8, via *TopHalf*) and in the downward direction (line 9, via *BotHalf*). In *DDExpandHunk*, we continue to expand (upward or downward) the current buggy hunk *buggyHunk* to include a statement that is deemed as buggy by the RNN model or has a data dependency with the center buggy statement *buggyS* (lines 13–14). We stop the process (upward or downward),

if we encounter a non-buggy statement without data dependency with *buggyS* or we exhaust the list (line 15). Finally, the buggy hunk containing consecutive buggy statements is returned.

Algorithm 2 Multiple-Statement Expansion Algorithm

```
1: function MultiStatementsExpansion(buggyS)
2:     candStmts = Expand2NCandidatesList(buggyS)
3:     predResult = RNNClassifier(candStmts)
4:     expandResult = DataDepAnalysis(candStmts, predResult)
5:     return expandResult
6: function DataDepAnalysis(buggyS,candStmts, predResult)
7:     buggyHunk = GetCenterBuggyHunk(predResult)
8:     DDExpandHunk(buggyS, buggyHunk, TopHalf(candStmts))
9:     DDExpandHunk(buggyS, buggyHunk, BotHalf(candStmts))
10:    return buggyBlock
11: function DDExpandHunk(buggyS, buggyHunk, candStmts)
12:    for each (stmt ∈ candStmts) & (stmt ∉ buggyHunk) do
13:        if HasDataDep(stmt, buggyS) then
14:            buggyHunk = buggyHunk ∪ stmt
15:        else break
16:    return buggyHunk
```

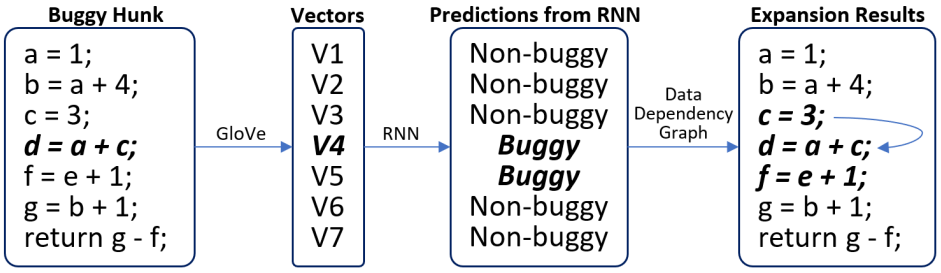


Figure 4.17 Automated Program Repair: Multiple-Statement expansion example.

In Figure 4.17, the SBFL tool returns the buggy statement at line 4. All statements are encoded into the sets of vectors via GloVe [122] and classified by the RNN model. We expand from the statement at line 4 upward to include line 3 (even though the RNN model predicted it as non-buggy), since line 3 has a data dependency with the buggy statement at line 4 via the variable *c*. We include line 5 because the RNN model predicts the line 5 as buggy. At this time, we stop the upward and downward directions because we encounter the non-buggy statements at lines 2 and 6 that do not have data dependency with line 4. That is, lines 1–2 and 6–7 are excluded. The final result includes the statements at lines 3–5 as the buggy hunk.

4.3.6.2.3 Buggy statement prediction with RNN. We present how to use an GRU-based RNN model [25] to predict a buggy statement.

4.3.6.2.4 Training. To train the RNN model, we use the buggy/non-buggy statements in all the hunks in the training dataset. We use GloVe [122] to encode each token in a statement so that a statement is represented by a sequence of token vectors. We use the neural architecture of the GRU-based RNN model [25] to consume the GloVe vectors of statements associated with the buggy/non-buggy labels.

The RNN model operates in the time steps. At the time step k (k¿=1), at the input layer, GRU consumes the GloVe vectors of the k^{th} statement S_k. At the output layer, S_k is labeled as 1 if it is a buggy statement and as 0 otherwise. In addition to the input at the time step $k+1$, we feed the output of the time step k to the GRU.

4.3.6.2.5 Prediction. The trained GRU-based RNN model is used in Expansion Algorithm at line 3 to predict if a statement in the hunk is buggy or not. The model takes a statement in the form of GloVe token vectors. It takes the vectors of all the statements in a hunk and labels them as buggy or non-buggy in multiple-time-step manner.

4.3.6.2.6 Tree-Based code repair. Figure 4.18 illustrates this process. After deriving the buggy hunk(s) in the method(s), DEAR performs code repairing for all

buggy statements in all the hunks at once using the trained LSTM models. The tree-based code repair is conducted in the following steps:

Step 1. Identifying buggy S-subtrees For each hunk, we parse the code into ABT, and identify the buggy S-subtrees corresponding to the derived buggy statements. In Figure 4.18, the S-subtrees T_1 and T_2 are identified as buggy. If a buggy S-subtree is part of another larger buggy S-subtree, we just need to perform fixing on the larger S-subtree since that fix also fixes the smaller S-subtree.

Step 2. Embedding and Summarization We perform word embedding using GloVe [122] and tree summarization using TreeCaps [50] on all the buggy subtrees to obtain the contexts. For example, in Figure 4.18, T_1 and T_2 are summarized into two vectors V_1 and V_2.

Step 3. Predict Context We use the trained context learning model (CTL) to run on the context with the AST nodes including also the summarized nodes to predict the context. In the resulting AST, the structure is the same as the AST for the input context, except that the summarized nodes become the new ones. For example, V_1 and V_2 in the context becomes V_1' and V_2' after Step 3.

Step 4. Adding Weights The weights V_1 and V_2 from Step 2 are used in a product with the vectors in the buggy subtrees T_1 and T_2. Each node in T_1 and T_2 is represented by a multiplication vector between the original vector of the node and the weight vector V_1' or V_2'.

Step 5. Predict Transformations We use the trained tree transformation learning model to predict the subtrees T_1' and T_2' of the fix.

Step 6. Removing Weights We remove the weight from Step 4 to obtain the candidate fixed subtree for a buggy one. For example, we remove V_1' and V_2' to obtain the candidate fixed subtrees T_{1f} and T_{2f}. However, because we know the cross product and a vector, we can get the unlimited number of solutions. Thus, to produce a single solution T_{1f} and T_{2f}, for each node in T_1' and T_2', we assume that V_1'

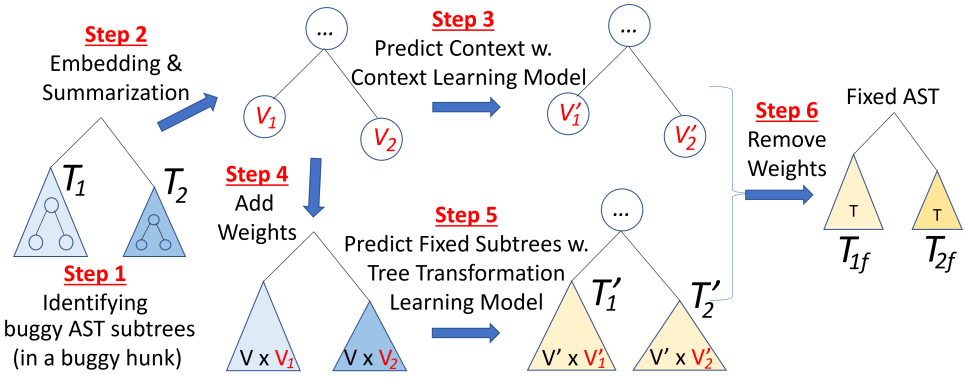


Figure 4.18 Automated Program Repair: Tree-Based code repair.

and V_2' are vertical with the node vector V_{n1} in T_1' and V_{n2} in T_2'. Then, we can get the unweighted node vector V_{n1}' in T_{1f} as follows:

$$V_{n1}' = \frac{V_1' \times V_{n1}}{V_1' \dot{V}_1'} \tag{4.10}$$

After having the vector for each node in a fixed S-subtree T_{1f}, we generated candidate patches based on word embedding. For each node, we calculated the cosine similarity score between its vector V_{n1}' and each vector in the vector list for all tokens. To generate candidate patches, we select the token t in the list. Token t has its similarity score $Score_t$ for one node in the fixed S-subtree. By adding all $Score_t$ for all the tokens, we have the total score, $Score_{sum}$, for a candidate. We select the top-5 candidates for each node to generate the candidates and sort them based on $Score_{sum}$.

4.3.6.3 Post-Processing. A naive approach would face combinatorial explosion in forming the candidate fix(es) because for each node in the sub-tree, we maintain top-5 candidates. However, when we combine the candidates for all the nodes in the fixed sub-tree, many candidates are not valid for the current method in the project. Therefore, when we form a candidate by combining all the candidates for the nodes, we apply a set of filters to verify the program semantics in the same manner as in DLFix [74]. This allows us to eliminate invalid candidates immediately. Specifically, we use the alpha-renaming filter to change the names back to the normal Java code

using a dictionary containing all the valid names in the scope, the syntax-checking filter to remove the candidates with syntax errors, and the name validation filter to check the validity of the variables, methods, and classes. Moreover, for further improvement, we use beam search to maintain only the top-ranked candidate fixes. Thus, we do not exhaust all compositions in forming the statements. This helps maintain a manageable number of candidates.

After applying all the filters, we also used DLFix [74]'s re-ranking scheme on the candidate patches. We then used test cases to conduct patch validation on those candidates. We verify each patch from the top to the bottom until a correct patch is identified and the patch validation ends. If all candidates for fixing a location cannot pass all the test cases, we select the next location to repeat the process.

4.3.7 Empirical evaluation

4.3.7.1 Research questions. To evaluate DEAR, we seek to answer the following questions:

RQ1. Comparative Study with Deep Learning-Based APR Models on Defects4J Benchmark. How well does DEAR perform in comparison with existing *DL-based* APR models on *Defects4J*?

RQ2. Comparative Study with Deep Learning-Based APR Models on Large Bug Datasets. How well does DEAR perform in comparison with *DL-based* APR models on *large-scale bug datasets*?

RQ3. Comparative Study with Pattern-Based APR Approaches on Defects4J. How well does DEAR perform in comparison with the state-of-the-art, *pattern-based* APR approaches?

RQ4. Sensitivity Analysis of DEAR. How do various factors affect the overall performance of DEAR in APR?

RQ5. Time Complexity and Model's Training Parameters. What is time complexity and the numbers of training parameters?

4.3.7.2 Data collection.

We have conducted our empirical evaluation on three datasets:

1) *Defects4J* v1.2.0 [32] with 395 bugs with test cases;

2) *BigFix* [74] with +26k bugs in +1.8 million buggy methods;

3) *CPatMiner* [110] with +44k bugs from 5,832 Java projects.

All experiments were conducted on a workstation with a 8-core Intel CPU and a single GTX Titan GPU.

4.3.7.3 Experimental methodology.

4.3.7.3.1 RQ1. Comparison with DL-Based APR on Defects4J.

Comparative Baselines. We compare DEAR with five state-of-the-art DL-based APR models: **DLFix** [74], **CoCoNuT** [94], **SequenceR** [24], **Tufano19** [156], **CODIT** [21], and **CURE** [54].

Procedure and Settings. We replicated all DL-based APRs except CURE, which is unavailable. We re-implemented CURE following the details in their paper. We trained all DL approaches on the bugs and fixes in CPatMiner dataset and tested them on all 395 bugs in Defects4J (no overlap between the two datasets). All DL approaches were applied with the same fault localization tool, Ochiai [2], and patch validation with the test cases in Defects4J. Following prior experiments [53, 74], we set a 5-hour running-time limit for a tool for patch generation and validation.

We tuned DEAR with the following key hyper-parameters using the beam-search: (1) BERT for hunk detection: epoch size (e-size) $(2, 3, 4, 5)$, batch size (b-size) $(8, 16, 32, 64)$, and learning rate (l-rate) $(3e^{-4}, 1e^{-4}, 5e^{-5}, 3e^{-5}, 1e^{-5})$; (2) LSTM for Multi-Statement Expansion and code repair: e-size $(100, 150, 200, 250)$, b-size $(32, 64, 128, 256)$, and l-rate $(0.0001, 0.0005, 0.001, 0.003, 0.005)$; (3) GloVe for representation vectors: vector size (v-size) $(100, 150, 200, 250)$, l-rate $(0.001, 0.003, 0.005, 0.01)$, b-size $(32, 64, 128, 256)$, and e-size $(100, 150, 200, 250)$. The other default parameters were used.

The best setting for DEAR is (1) e-size=4, b-size=32, l-rate=$1e^{-4}$ for BERT; (2) e-size=200, l-rate=0.003, b-size=128 for LSTM; (3) v-size=200, l-rate=0.001, b-size=64, e-size=200 for GloVe. For other models, we tuned with the parameters in their papers, e.g., the vector length of word2vec, learning rate, and epoch size to find the best parameters for each dataset. We tuned all approaches with the aforementioned parameters on the same *CPatMiner* dataset to obtain the best performance. Once we obtained the best parameters for each model, we used them for later experiments.

Quantitative Analysis. We report the numbers of bugs that a model can auto-fix for the following bug-location types:

Type-1. One-Hunk, One-Statement: A bug with the fix involving only one hunk with one single statement.

Type-2. One-Hunk, Multi-Statements: A bug with the fix involving only one hunk with multiple statements.

Type-3. Multi-Hunks, One-Statement: A bug with the fix involving multiple hunks; each hunk with one fixed statement.

Type-4. Multi-Hunks, Multi-Statements: A bug with the fix involving multi-hunks; each hunk has multiple statements.

Type-5. Multi-Hunks, Mix-Statements: A bug with the fix involving multiple hunks, and some hunks have one statement and other hunks have multiple statements.

Evaluation Metrics. We report the *number of bugs* that can be correctly fixed and the number of plausible patches (i.e., passing all test cases, but not the actual fixes) using the *top candidate patches*.

4.3.7.3.2 RQ2. Comparison with DL-Based APR on large datasets.

Comparative Baselines. We compare DEAR with the same baselines as in RQ1 on two large datasets: BigFix and CPatMiner.

Procedure and Settings. First, we evaluated all DL-based APR models on BigFix and CPatMiner. Following DLFix and Sequencer, we randomly split data into 80%/10%/10% for training, tuning, and testing. Second, we have cross-dataset evaluation: training DL-based approaches on CPatMiner and testing on BigFix, and vice versa. Unlike Defects4J, BigFix and CPatMiner datasets do not have test cases. Without test cases, we cannot use fault localization and patch validation for all DL approaches. Thus, we fed the actual bug locations into the DL models, including locations on buggy hunks and statements. The DL-based baselines do not distinguish hunks, instead *process each buggy statement at a time*. We use developers' actual fixes as the ground truth to evaluate the DL-based approaches.

Evaluation Metrics. We use the *top-K metric*, defined as the ratio between the number of times that a correct patch is in a ranked list of the top K candidates over the total number of bugs.

4.3.7.3.3 RQ3. Comparison with pattern-based APR on Defects4J.

Comparative Baselines. We compare DEAR with the state-of-the-art, pattern-based APR tools on Defects4J: **Elixir** [134], **ssFix** [178], **CapGen** [167], **FixMiner** [64], **Avatar** [86], **Hercules** [136], **SimFix** [53], and **Tbar** [87]. We were able to replicate the following pattern-based baselines: **Elixir, ssFix, FixMiner, SimFix, TBar** under the same computing environments. We set the time limit to 5 hours for the tools. For the other baselines, due to unavailable code, we use the results reported in their papers as they were run on the same dataset. We used the same setting and evaluation metric.

4.3.7.3.4 RQ4. Sensitivity analysis. We evaluate the impacts of different factors on DEAR's performance. We consider the following: (1) hunk detection (Hunk); (2) multi-statement expansion (Expansion); (3) multi-statement tree model and cycle training; and (4) data splitting scheme. We use the left-one-out strategy

for each factor. We evaluate the first three factors on Defects4J and the last one on CPatMiner since we need a larger dataset for various splitting.

4.3.7.3.5 Time complexity and numbers of parameters in model training. We measure the training and fixing time for a model and its number of parameters for model training on the datasets.

4.3.8 Empirical results

4.3.8.1 RQ1. Comparison results with DL-Based APR models on Defects4J.

4.3.8.1.1 With fault localization. We first evaluate the APR models when using with the fault localization tool Ochiai [2]. Tables 4.5 and 4.6 show the comparison results among DEAR and the baseline models.

As seen in Table 4.5, DEAR can auto-fix the most number of bugs (47) and generate the most number of plausible patches (91) that pass all test cases on Defects4J. Particularly, DEAR can auto-fix 32, 41, 33, 17, 14, and 11 more bugs than Sequencer, CODIT, Tufano19, DLFix, CoCoNuT, and CURE, respectively (i.e., 213%, 683%, 236%, 57%, 42%, and 31% relative improvements). Compared with those tools in that order on Defects4J, DEAR can auto-fix 35, 34, 41, 18, 31, and 18 bugs that those tools missed, respectively. Via the overlapping analysis between the result of DEAR and those of the baselines combined, DEAR can fix 18 unique bugs that they missed.

Table 4.6 shows the comparison between DEAR and the top DL-based baselines (DLFix, CoCoNuT, CURE) w.r.t. different bug types.

For single-hunk bugs (Types 1-2), DEAR fixes 33 bugs including 4 unique single hunk bugs that the other tools missed.

Table 4.5 Automated Program Repair: RQ1. Comparison with DL-Based APR Models on Defects4J with Fault Localization.

Projects	Chart	Closure	Lang	Math	Mockito	Time	Total
Sequencer	3/3	4/5	2/2	6/9	0/0	0/0	15/19
CODIT	1/2	2/5	0/0	3/5	0/0	0/0	6/12
Tufano19	3/4	3/5	1/1	6/8	0/0	0/0	14/18
DLFix	5/12	6/10	5/12	12/18	1/1	1/2	30/55
CoCoNuT	6/11	6/9	5/13	13/21	2/2	1/1	33/57
CURE	6/13	6/10	5/14	16/23	2/2	1/2	36/71
DEAR	8/16	7/11	8/15	20/33	1/2	3/6	**47/91**

Notes: X/Y: are the numbers of correct and plausible patches, respectively.

Table 4.6 Automated Program Repair: RQ1. Detailed Comparison with DL-Based APR Models on Defects4J with Fault Localization.

Bug Types	DLFix	CoCoNuT	CURE	DEAR
Type 1. One-Hunk One-Stmt	30	33	36	29
Type 2. One-Hunk Multi-Stmts	0	0	0	4
Type 3. Multi-Hunks One-Stmt	0	0	0	11
Type 4. Multi-Hunks Multi-Stmts	0	0	0	1
Type 5. Multi-Hunks Mix-Stmts	0	0	0	2
Total	30	33	**36**	**47**

For multi-hunk bugs (Types 3–5), DEAR can fix 14 bugs that cannot be fixed by DLFix, CoCoNuT, and CURE. Existing DL-based APR models cannot fix those bugs since the mechanism of fixing one statement at a time does not work on the bugs that require the fixes with dependent changes to multiple statements at once. Thus, they do not produce correct patches for those cases.

For multi-hunk or multi-statement bugs (Types 2–5), DEAR fixes 18 of them (out of 47 fixed bugs, i.e., 38.3% of total fixed bugs).

4.3.8.1.2 Without fault localization. We also compared DEAR with other tools in the fixing capabilities without the impact of a third-party FL tool. All the tools under comparison (Table 4.7) were pointed to the correct fixing locations and performed the fixes. As seen, if the fixing locations are known, DEAR's fixing capability is also higher than those baselines (53 bugs versus 44, 40, and 48). Importantly, it can fix 20 multi-hunk/multi-statement bugs (37.7% of a total of 53 fixed bugs), while CoCoNuT, DLFix, and CURE can fix only 7, 5, and 10 such bugs.

Table 4.7 Automated Program Repair: RQ1. Comparison with DL-Based APR Models on Defects4J without Fault Localization (i.e., Correct Location)

Bug Types	DLFix	CoCoNuT	CURE	DEAR
Type 1. One-Hunk One-Stmt	35	37	38	33
Type 2. One-Hunk Multi-Stmts	1	3	3	4
Type 3. Multi-Hunks One-Stmt	4	3	6	13
Type 4. Multi-Hunks Multi-Stmts	0	0	0	1
Type 5. Multi-Hunks Mix-Stmts	0	1	1	2
Total	40	44	48	**53**

DEAR is more general than existing DL-based models because it can support dependent fixes with multi-hunks or multi-statements. Importantly, *it significantly improves these DL-based models and raises the DL direction to the same level as the other APR directions (search-based and pattern-based)*, which can handle multi-statement bugs. Moreover, DEAR is fully data-driven and does not require the defined fixing patterns as in the pattern-based APR models.

4.3.8.2 RQ2. Comparison results with DL-Based APR models on large datasets.

Table 4.8 shows that DEAR can fix more bugs than any DL-based APR baselines on the two large datasets. Using the top-1 patches, DEAR can fix 15.1% of the total 4,415 bugs in CPatMiner. It fixes 40–322 more bugs than the baselines with top-1 patches. On BigFix, it can fix 14.1% of the total 2,594 bugs with the top-1 patches. It can fix 31–145 more bugs than those baselines with the top-1 patches.

Table 4.9 shows that DEAR also outperformed the baselines in the cross-dataset setting in which we trained the models on CPatMiner and tested them on BigFix and vice versa.

Table 4.8 Automated Program Repair: RQ2. Comparison with DL APRs on Large Datasets.

	CPatMiner (4,415 tested bugs)			BigFix (2,594 tested bugs)		
Tool/Dataset	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Sequencer	7.8%	8.9%	10.3%	8.5%	9.1%	10.8%
CODIT	4.5%	7.4%	9.2%	3.9%	6.3%	9.1%
Tufano19	8.6%	9.3%	11.2%	7.7%	8.8%	9.6%
DLFix	11.4%	12.3%	13.1%	11.2%	11.9%	12.5%
CoCoNuT	13.5%	14.7%	15.3%	12.2%	13.6%	14.3%
CURE	14.2%	15.1%	15.5%	12.9%	14.2%	14.1%
DEAR	**15.1%**	**15.6%**	**16.8%**	**14.1%**	**15.4%**	**16.3%**

Table 4.10 shows the detailed comparative results on CPatMiner w.r.t. different bug types. As seen, DEAR can *auto-fix more bugs on every type of bug locations on the two large datasets*. Among 667 fixed bugs, DEAR has fixed 169 multi-hunk or multi-stmt bugs of Types 2–5 (i.e., 25.33% of the total fixed bugs). DEAR fixes more

bugs (71, 164, and 41 more), and fixes more bugs in each bug type than the baselines CoCoNuT, DLFix, and CURE, respectively.

Table 4.9 Automated Program Repair: RQ2. Comparison with DL APRs on Cross-Datasets.

Tool/Dataset	CPatMiner(Train)/BigFix			BigFix(Train)/CPatMiner		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Sequencer	5.4%	5.8%	6.2%	5.3%	6.1%	7.2%
CODIT	2.5%	4.0%	4.4%	3.2%	5.2%	6.4%
Tufano19	4.5%	5.4%	5.7%	5.9%	6.3%	7.6%
DLFix	6.3%	6.9%	7.3%	8.2%	8.7%	9.2%
CoCoNuT	6.7%	7.4%	8.1%	8.3%	9.6%	10.7%
CURE	7.1%	7.7%	8.2%	8.7%	9.9%	10.9%
DEAR	**7.5%**	**8.1%**	**8.6%**	**9.6%**	**10.2%**	**11.3%**

Table 4.10 Automated Program Repair: RQ2. Detailed Analysis: Top-1 Result Comparison with DL-Based APR Models on CPatMiner Dataset.

Types (#bugs)	CoCoNuT	CURE	DLFix	DEAR
	#Fixed	#Fixed	#Fixed	#Fixed
Type-1 (1,668)	28.7% (479)	29.8%(497)	23.7% (395)	29.9% (499)
Type-2 (530)	1.3% (7)	2.1% (11)	0.6% (3)	4.2% (22)
Type-3 (879)	12.4% (109)	13.2% (116)	11.8% (104)	13.7% (120)
Type-4 (1,089)	0% (0)	0% (0)	0% (0)	2.0% (22)
Type-5 (249)	0.4% (1)	0.8% (2)	0.4% (1)	2.0% (5)
Total (4,415)	13.5% (596)	14.2% (626)	11.4% (503)	**15.1% (667)**

DEAR fixes 52, 61, and 40 more multi-hunk/multi-stmt bugs, and 20, 104, and 2 more one-hunk/one-stmt bugs than CoCoNuT, DLFix, and CURE. For the multi-statement bugs (Types 2 and 5) that the other tools fixed, the fixed statements are independent. This result shows that fixing each individual statement at a time does not work.

4.3.8.3 RQ3. Comparison results with Pattern-Based APR models. As seen in Table 4.11, DEAR performs at the same level in terms of the number of bugs as the top pattern-based tools Hercules and Tbar.

Table 4.11 Automated Program Repair: RQ3. Comparison with Pattern-Based APR Models.

Projects	Chart	Closure	Lang	Math	Mockito	Time	Total
ssFix	3/7	2/11	5/12	10/26	0/0	0/4	20/60
CapGen	4/4	0/0	5/5	12/16	0/0	0/0	21/25
FixMiner	5/8	5/5	2/3	12/14	0/0	1/1	25/31
ELIXIR	4/7	0/0	8/12	12/19	0/0	2/3	26/41
AVATAR	5/12	8/12	5/11	6/13	2/2	1/3	27/53
SimFix	4/8	6/8	9/13	14/26	0/0	1/1	34/56
Tbar	9/14	8/12	5/13	19/36	1/2	1/3	43/81
Hercules	8/10	8/13	10/15	20/29	0/0	3/5	49/72
DEAR	8/16	7/11	8/15	20/41	1/2	3/6	**47/91**

Notes: X/Y: are the numbers of correct and plausible patches; Dataset: Defects4J

Table 4.12 displays the details of the comparison w.r.t. different bug types. As seen, DEAR fixes 7 Multi/Mix-Statement bugs (Types 2, 4–5) that Hercules missed. Investigating further, we found that Hercules is designed to fix *replicated bugs*, i.e., the hunks must have similar statements. Those 7 bugs are non-replicated, i.e., the buggy hunks have different buggy statements or a buggy hunk has multiple non-similar

buggy statements. For Types 1 and 3, DEAR fixes 9 less one-statement bugs than Hercules due to its incorrect fixes. In total, DEAR fixes *12* bugs that Hercules misses: *Chart-7,16,20,24; Time-7; Closure-6,10,40; Lang-10; Math-41,50,91.*

Compared to Tbar, DEAR fixes 15 more multi-hunk/multi-stmt bugs. Tbar is not designed to fix multi-statements at once as DEAR. Instead, it fixes one statement at a time, thus, does not work well when those 15 bugs require dependent fixes to multiple statements. The 3 bugs of Type 2 that Tbar can fix are the ones that the fixes to individual statements are independent. The same reason is applied to SimFix. Tbar fixes 11 more correct one-hunk/one-statement bugs.

Table 4.12 Automated Program Repair: RQ3. Detailed Comparison with Pattern-Based APRs.

Bug Types	SimFix	Tbar	Hercules	DEAR
Type 1. One-Hunk One-Stmt	30	40	34	29
Type 2. One-Hunk Multi-Stmts	1	3	0	4
Type 3. Multi-Hunks One-Stmt	3	0	15	11
Type 4. Multi-Hunks Multi-Stmts	0	0	0	1
Type 5. Multi-Hunks Mix-Stmts	0	0	0	2
Total	34	43	49	47

In brief, we raise DEAR, a DL-based model, to the *comparable and complementary* level with those pattern-based APR models.

4.3.8.4 RQ4. Sensitivity analysis.

4.3.8.4.1 Impact of fixing-together hunk detection. As seen in Table 4.13, without hunk detection, DEAR can auto-fix 35 bugs. With hunk detection, DEAR can fix 14 more multi-hunk bugs (Types 3-5). It fixes two less Type-1 bugs due

to the incorrect hunk detection. In brief, hunk detection is useful since the multi-hunk/multi-statement bugs require dependent fixes to multiple hunks at once.

Table 4.13 Automated Program Repair: RQ4. Sensitivity Analysis on Defects4J.

Variant	Without Hunk-Det	Without Expansion	Without Attention-Cycle	DEAR
Type-1	31	30	26	29
Type-2	4	0	2	4
Type-3	0	13	9	11
Type-4	0	0	1	1
Type-5	0	0	2	2
Total	35	43	40	47

4.3.8.4.2 Impact of multi-statement expansion. As seen in Table 4.13, without expansion, DEAR fixes *43* bugs in Defects4J. With expansion, it fixes *7* more multi-stmt bugs in Types 2,4,5 while it fixes two less Type-3 bugs and one less Type-1 bug. The reason of fixing less bugs in these two types is that the multi-statement expansion may expand the buggy hunk incorrectly by regarding a single-statement bug as a multi-statement bug. Even so, DEAR still can fix more bugs, showing the usefulness of the multi-statement expansion.

To compare the impact of Hunk Detection and Multi-Statement Expansion, let us note that the variant of DEAR without Hunk-Detection missed all 14 multi-hunk bugs (Types 3,4,5). The variant without Expansion missed all 7 multi-statement bugs (Types 2,4,5). However, let us consider how challenging it is to fix them. Among 14 multi-hunk bugs fixed with Hunk-Detection, 11 bugs are of Type-3 (multi-hunk/one-statement), in which some approaches (e.g., Hercules) can handle by fixing one statement at a time. Only 3 bugs are of Types 4–5. In contrast, all

Table 4.14 Automated Program Repair: Impact of the Size of Training Data.

Splitting Scheme on CPatMiner dataset	90%/10%	80%/20%	70%/30%
% Total Bugs at Top-1	15.1%	13.8%	11.7%

7 bugs fixed with Expansion are multi-statement bugs (Types 2,4,5), which cannot be fixed by existing DL-based APR approaches. Thus, Expansion contributes to handling more challenging bugs than Hunk-Detection.

4.3.8.4.3 Impact of Tree-Based LSTM model with attention and cycle training. (Attention-Cycle) To measure the impact of Attention and Cycle Training, we removed those two mechanisms from DEAR to produce a baseline. Our results show that in Defects4J, DEAR fixes 7 more bugs on all bug types than the baseline (17.5% increase). This result indicates the usefulness of the two mechanisms.

4.3.8.4.4 Impact of training data's size. Table 4.14 shows that the size of training data has impact on DEAR's performance. As seen in Table 4.14, the more training data, the higher the DEAR's accuracy. This is expected as DEAR is a data-driven approach. But even with less training data (70%/30%), DEAR achieves 11.7% for top-1 result, which is still higher than DLFix (11.4% in top-1) and Sequencer (7.7% in top-1); both are with more training data (90%/10% splitting).

4.3.8.5 RQ5. Time complexity and parameters. Training time of DEAR on CPatMiner was +22 hours and predicting on CPatMiner took 2.4-3.1 seconds for each candidate patch. Training of DEAR on BigFix took 18-19 hours and predicting on BigFix took 3.6-4.2 seconds for each candidate. Predicting on Defects4J took only 2.1 seconds for a candidate due to a much smaller dataset. Test execution time was +1 second per test case. Test validation took 2–20 minutes for all the test cases for a bug fix.

The best baseline, CURE [54], fixes fewer bugs than DEAR (RQ1 and RQ2), and requires *7 and 7.3 times more* training parameters than DEAR on CPatMiner and BigFix, respectively. Specifically, DEAR and CURE require 0.39M and 3.1M training parameters on CPatMiner, and 0.42M and 3.5M parameters on BigFix. Thus, DEAR is less complex than CURE, while achieving better results.

4.3.9 Illustrative example

```
1  public void excludeRoot(String path) {
2  -    String url = toUrl(path);
3  -    findOrCreateContentRoot(url).addExcludeFolder(url);
4  +     Url url = toUrl(path);
5  +     findOrCreateContentRoot(url).addExcludeFolder(url.getUrl());
6  }
7  public void useModuleOutput(String production, String test) {
8       modifiableRootModel.inheritCompilerOutputPath(false);
9  -    modifiableRootModel.setCompilerOutputPath(toUrl(production));
10 -    modifiableRootModel.setCompilerOutputPathForTests(toUrl(test));
11 +     modifiableRootModel.setCompilerOutputPath(toUrl(production).getUrl());
12 +     modifiableRootModel.setCompilerOutputPathForTests(toUrl(test).getUrl());
13 }
```

Figure 4.19 Automated Program Repair: A multi-hunk/multi-statement fixing in CPatMiner.

Figure 4.19 shows a correct fix from DEAR. It correctly detects two buggy hunks; each with multiple statements. DEAR leverages the variable names existing in the same method (*modifiableRootModel* at line 8) in composing the fixed code at lines 11–12. The DL-based baselines, Sequencer [24] and CoCoNuT [94], treat code as sequences, and does not derive well the structural changes for this fix. DLFix fixes one statement at a time, thus, does not work (the fixes at line 2 and line 3 depend

on each other). For pattern-based APRs [136, 53], there is no fixing template for this bug.

4.3.9.1 Limitations. DEAR has the following limitations. First, as with ML approaches, fixes with rare or out-of-vocabulary names are challenging. With more training data, DEAR has higher chance to encounter the ingredients to generate a new name. Second, we focus only on the bugs that cause failing tests. Security, vulnerabilities, and non-failing-test bugs are still its limitations. Third, we cannot generate fixes with several new statements added or arbitrarily large sizes of dependent fixed statements. Fourth, the expansion algorithm produces incorrect hunks to be fixed, leading to fixing incorrect statements. Finally, we currently focus on Java, however, the basic representations used in DEAR, e.g., token, AST, dependency, are universal to any program language. Only third-party FL and post-processing with semantic checkers are language-dependent.

4.3.10 Threats to validity

We tested on Java code. The key modules in DEAR are language-independent, except for the third-party FL and post-processing with program analysis. Pattern-based APR tools require a dataset with test cases, thus, we compared them on Defects4J only. We tried our best to re-implement the pattern-based APR baselines and CURE for a fair comparison.

4.4 Conclusion

In this chapter, we designed our approaches with suitable learning code representations on two research works on automated program repair. One is for single-line bug fixing, while the other is dealing with multi-hunk/multi-statement bugs.

For single-line bug auto repair, we propose a new deep learning (DL) based automated program repair (APR) approach, namely DLFix, to improve and complement the existing state-of-the-art APR approaches. The key ideas that enable our

approach are (1) using tree-based RNN to directly model code and learning tree-based structural code transformations from previous bug fixes; (2) learning the context of the code surrounding a fix, namely local context learning. In DLFix, we propose a two-layer tree-based RNN encoder-decoder model to learn local contexts and code transformations from previous fixes. In addition, we build a CNN-based classification approach to re-rank possible patches.

For multi-hunk/multi-statement bug auto repair, we make three key contributions: (1) a novel FL technique for multi-hunk, multi-statement fixes combining traditional SBFL with deep learning and data-flow analysis; (2) a compositional approach to generate multi-hunk, multi-statement fixes with a divide-and-conquer strategy; and (3) enhancements and orchestration of a two-layer LSTM model with the attention layer and cycle training.

All the empirical results show that our approaches can outperform all the studied state-of-the-art deep learning-based automated program repair approaches and have complementary results against pattern-based APR approaches. Our notion's effectiveness in utilizing appropriate code representations for enhancing the performance of automated program repair tasks is substantiated. Once the software has undergone testing, it can be released to the market, marking the transition to the crucial maintenance stage that requires developers' attention. Detecting vulnerabilities during this stage holds significant importance. In Chapter 5, we will go deeper into the topic of selecting appropriate code representations for effectively addressing vulnerabilities, providing a comprehensive explanation.

CHAPTER 5

MODEL AGNOSTIC EXPLANATION

5.1 Introduction

In this chapter, we present our research focusing on a specific aspect of model-agnostic explanation: providing explanations for graph-based deep learning models in the context of vulnerability detection tasks. The structure of this chapter, along with the corresponding code representations used for each research topic, can be visualized in Figure 5.1. Our work on this research topic has already been published as a conference paper, which will be discussed in detail in the subsequent sections.

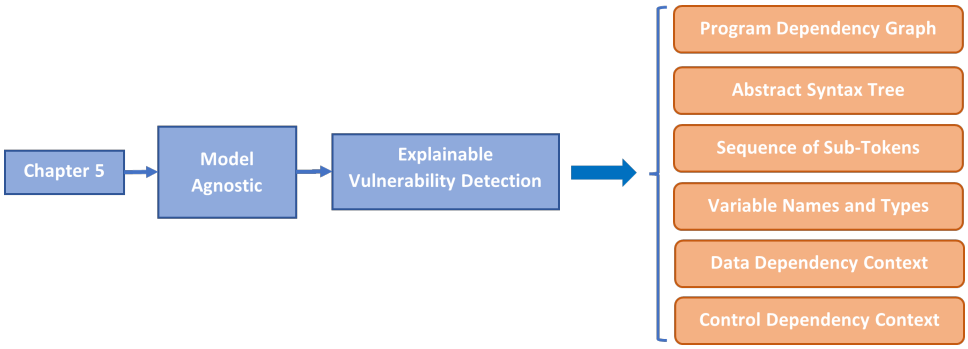


Figure 5.1 Model Agnostic Explanation: Roadmap for Chapter 5.

5.2 Model Agnostic Explanation for Graph-Based Vulnerability Detection and Bug Detection

5.2.1 Introduction

Software vulnerabilities have caused substantial damage to our society's software infrastructures. To address the problem, several automated vulnerability detection (VD) approaches have been proposed. They can be broadly classified into two categories: *program analysis* (PA)-based [38, 127, 161, 23, 49, 30] and *machine learning* (ML)-based [137, 109, 140]. The PA-based VD techniques have often

focused on solving the *specific types of vulnerabilities* such as BufferOverflow [18], SQL Injection [149], Cross-Site Scripting [142], Authentication Bypass [148], *etc.* In addition to those types, the more general software vulnerabilities, *e.g.*, the software vulnerabilities occurring in API usages of libraries/frameworks, have manifested in various forms. To detect them, machine learning (ML) and deep learning (DL) have been leveraged to implicitly learn the patterns of vulnerabilities from prior vulnerable code [79, 198, 45].

Despite several advantages, the ML/DL-based VD approaches are still limited to providing only coarse-grained detection results on whether an entire given method is vulnerable or not. In comparison with the PA-based approaches, they fall short in the ability to elaborate on the *fine-grained details* of the lines of code with specific statements that might be involved in the detected vulnerability. One could use fault localization (FL) techniques [58] to locate the vulnerable statements, however they require large, effective test suites. Due to such feedback at the coarse granularity from the existing ML/DL-based VD tools, developers would not know where and what to look for and to fix the vulnerability in their code. This hinders them in investigating the potential vulnerabilities.

To raise the level of ML/DL-based VD, we present IVDetect, an *interpretable VD* with the philosophy of using *Artificial Intelligence* to detect coarse-grained vulnerability, while leveraging *Intelligence Assistant* via interpretable ML to provide fine-grained interpretations in term of vulnerable statements relevant to the vulnerability.

For *coarse-grained vulnerability detection*, our novelty is the *context-aware representation learning of the vulnerable code*. During training, the existing ML/DL-based VD approaches [79, 198] take the entire vulnerable code in a method as the input without distinguishing the vulnerable statements from the surrounding contextual code. Such distinction from vulnerable code and the contexts during training enables

IVDetect to better learn to discriminate the vulnerable code and benign ones. We represent source code via program dependence graph (PDG) and we treat the vulnerability detection problem as graph-based classification via Graph Convolution Network (GCN) [63] with feature-attention (FA), namely FA-GCN. The vulnerable statements, along with surrounding code, are encoded during the code representation learning.

For *fine-grained interpretation*, as the given method is deemed as vulnerable by IVDetect, our novelty is to *leverage interpretable ML [184] to provide the interpretation in term of the vulnerable statements as part of the PDG that are involved to the detected vulnerability.* The rationale for choosing PDG sub-graph as an interpretation is that a vulnerability often involves the data and control dependencies among the statements [123].

To derive the vulnerable statements as the interpretation, we leverage the interpretable ML model, GNNExplainer [184], that *"explains" on why a model has arrived at its decision.* Specifically, after vulnerability detection, to produce interpretation, IVDetect takes as input the FA-GCN model along with its decision (vulnerable or not), and the input PDG G_M of the given method M. The goal is to find the interpretation subgraph, which is defined as a minimal sub-graph $\mathcal{G}$ in the PDG of M that *minimizes the prediction scores between using the entire* G_M *and using* $\mathcal{G}$. To that end, we leverage GNNExplainer [184] in which the searching for $\mathcal{G}$ is formulated as the learning of the edge-mask set EM. The idea is that if an edge belongs to EM (*i.e.*, if it is *removed* from G_M), and *the decision of the model is affected, then the edge is crucial and must be included in the interpretation for the detection result.* Thus, the minimal sub-graph $\mathcal{G}$ in PDG contains the nodes and edges, *i.e.*, the *crucial statements and program dependencies, that are most decisive/relevant to the detected vulnerability* when the decision is vulnerable.

Using IVDetect's results, a practitioner could 1) examine the ranked list of potentially vulnerable methods, and 2) use the interpretation to further investigate what statements in the code that cause the model to predict that vulnerability.

We conducted several experiments to evaluate IVDetect in both vulnerability detection at the method level and interpretation in term of vulnerable statements. We use three large C/C++ vulnerability datasets: Fan [35], Reveal [22] and FFMPeg+Qemu [198]. For the method-level VD, our results show that IVDetect outperforms the existing ML/DL-based approaches [79, 198, 78, 132, 22] by 43%–84% and 105%–255% at the top-10 list for two ranking scores nDCG and MAP, respectively. For the statement-level interpretation, IVDetect correctly points out the vulnerable statements relevant to the vulnerability in 67% of the cases with a top-5 ranked list. It improves over the baseline ATT [184] and GRAD [184] interpretation models by 12.3%–400% and 9%–400% in accuracy, respectively.

The contributions of this research topic include:

A. Interpretable VD with Fine-Grained Interpretations

a. Vulnerability Detection with Fine-Grained Interpretations: IVDetect is the first approach to leverage interpretable ML to enhance VD with *fine-grained* details on PDG sub-graphs, statements, and dependencies relevant to the detected vulnerability.

b. Context-Aware Representation Learning of vulnerable code: The novelty of our representation learning of vulnerable code is *the consideration of the contextual code surrounding the vulnerable statements and fixes* to better train the VD model.

B. Empirical Evaluation. Our results show IVDetect's high accuracy in both detection and interpretation.

5.2.2 Motivation

5.2.2.1 Motivating example. Figure 5.2 shows the method *ec_device_ioctl_xcmd* in Linux 4.6, which constructs the I/O control command for the CromeOS devices. This is listed as a vulnerable code within Common Vulnerabilities and Exposures (CVE-2016-6156) in the National Vulnerability Database. The commit log of the corresponding fix stated that

"At line 6 and line 13, the driver fetches user space data by pointer arg via copy_from_user(). The first fetched value (stored in u_cmd) (line 6) is used to get the in_size and out_size elements and allocation a buffer (s_cmd) at line 10 so as to copy the whole message to driver later at line 13, which means the copy size of the whole message (s_cmd) is based on the old value (u_cmd.outsize) from the first fetch. Besides, the whole message copied at the second fetch also contains the elements of in_size and out_size, which are the new values. The new values from the second fetch might be changed by another user thread under race condition, which will result in a double-fetch bug when the inconsistent values are used."

Thus, to fix this bug, a developer added the code at lines 17–21 to make sure that *u_cmd.outsize* and *u_cmd.insize* have not changed due to race condition between the two fetching calls. Moreover, memory access might be also beyond the array boundary, causing a buffer overflow within the method call *cros_ec_cmd_xfer(...)*, when the command is transferred to the ChromeOS device at line 23. Another issue is at line 27 with *copy_to_user*. The method call *cros_ec_cmd _xfer(...)* can set *s_cmd->insize* to a lower value. Thus, the new smaller value must be used to avoid copying too much data to the user: *u_cmd.insize* at line 27 is changed into *s_cmd->insize*.

This vulnerable code could potentially cause the damages such as denial of service, buffer overflow, program crash, *etc.* Deep learning (DL) advances enable several approaches [198, 79] to *implicitly learn* from the history the patterns of vulnerable code, and to detect *more general vulnerabilities*.

```
1  static long ec_device_ioctl_xcmd(struct cros_ec_dev *ec, void __user *arg)
2  {
3      long ret;
4      struct cros_ec_command u_cmd;
5      struct cros_ec_command *s_cmd;
6      if (copy_from_user(&u_cmd, arg, sizeof(u_cmd)))
7          return -EFAULT;
8      if ((u_cmd.outsize > EC_MAX_MSG_BYTES) || (u_cmd.insize > EC_MAX_...))
9          return -EINVAL;
10     s_cmd = kmalloc(sizeof(*s_cmd) + max(u_cmd.outsize, u_cmd.insize),.);
11     if (!s_cmd)
12         return -ENOMEM;
13     if (copy_from_user(s_cmd, arg, sizeof(*s_cmd) + u_cmd.outsize)) {
14         ret = -EFAULT;
15         goto exit;
16     }
17 +       if (u_cmd.outsize != s_cmd->outsize ||
18 +               u_cmd.insize != s_cmd->insize) {
19 +               ret = -EINVAL;
20 +               goto exit;
21 +       }
22     s_cmd->command += ec->cmd_offset;
23     ret = cros_ec_cmd_xfer(ec->ec_dev, s_cmd);
24         /* Only copy data to userland if data was received. */
25     if (ret < 0)
26     goto exit;
27 -       if (copy_to_user(arg, s_cmd, sizeof(*s_cmd) + u_cmd.insize))
28 +       if (copy_to_user(arg, s_cmd, sizeof(*s_cmd) + s_cmd->insize))
29         ret = -EFAULT;
30         exit:
31         kfree(s_cmd);
32         return ret;
33 }
```

Figure 5.2 Model Agnostic Explanation: CVE-2016-6156 vulnerability in Linux 4.6.

However, they are still limited in comparison with program analysis-based approaches in the ability to provide any detail on the *fine-grained* level of the vulnerable statements, and on why the model has decided on the vulnerability. For example, the PA-based approaches, *e.g.*, a race detection technique, could potentially detect the involvement of the two fetching statements at line 6 and line 13. The method in Figure 5.2 might be deemed as vulnerable by a DL-based model. But without any fine-grained details, a developer would not know where and what to investigate next. This would make the output of a DL model less constructive in VD. Moreover, a fault localization technique [58], which locates buggy statements, does not solve the problem because it would need a large and effective test suite.

Regarding detection, the existing DL-based approaches [198, 79] do not fully exploit all the available information on the vulnerable code during training. For example, during training, we know that lines 23 and 27 are vulnerable/buggy, and other *relevant statements via data/control dependencies provide contextual information for the vulnerable ones*. However, the existing approaches [198, 79] do not consider the vulnerable statements and do not use the contextual code to help a model discriminate the vulnerable and non-vulnerable ones. The entire method would be fed to a DL model.

5.2.2.2 Approach overview and key ideas. We introduce IVDetect, an DL-based, *interpretable vulnerability detection* approach that goes beyond the decision of vulnerability by providing the *fine-grained* interpretation in term of the vulnerable statements. Specifically, as the method is deemed as vulnerable by IVDetect, it will provide a *list of important statements as part of the program dependence graph (PDG) that are relevant to the detected vulnerability*. For example, it provides the partial sub-graph of the PDG including the statements at the lines 13–15, 22–23, and 25–27 in Figure 5.3 for the vulnerable code at line 23 and line 27. We use the PDG sub-graph including important statements for fine-grained VD since they will give

a developer the hints on the program dependencies relevant to the vulnerability for further investigation. Moreover, if our model determines the code as non-vulnerable, it can also produce the key sub-graph of the PDG with the key statements that are deemed to be safe.

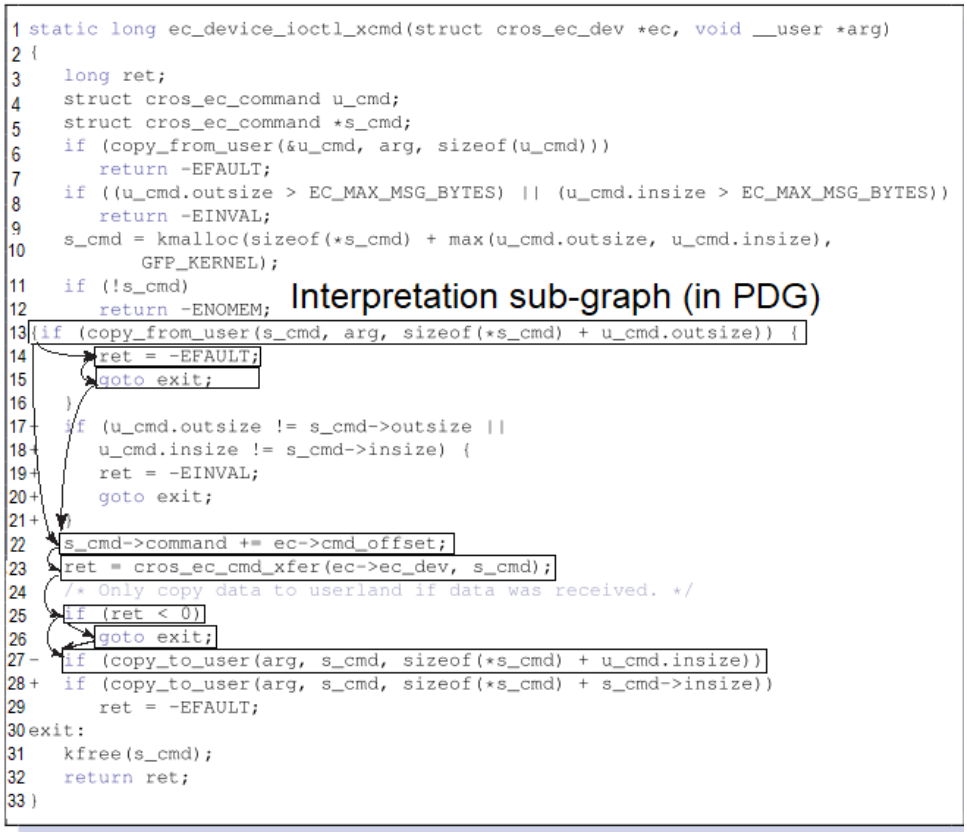


Figure 5.3 Model Agnostic Explanation: Interpretation sub-graph for Figure 5.2.

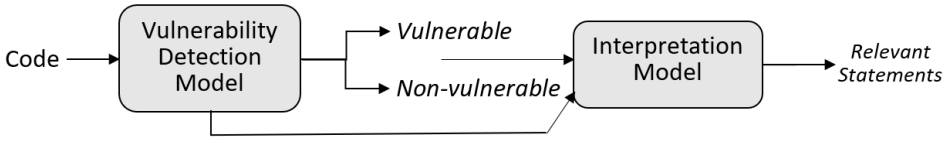


Figure 5.4 Model Agnostic Explanation: Overview of IVDetect.

IVDetect has two main modules (Figure 5.4): graph-based vulnerability detection model, and graph-based interpretation model. The input is the source code of all methods in a project. The output is the ranked list of methods with the

detection result/score and the interpretation (PDG sub-graph). Let us explain our key ideas.

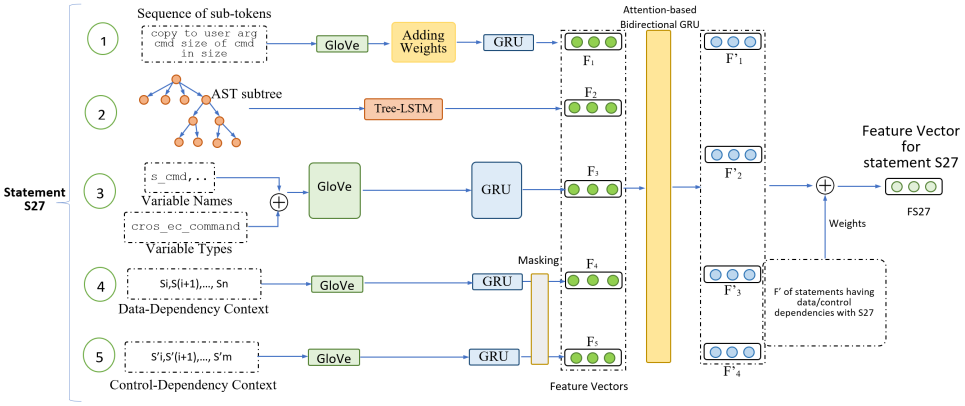


Figure 5.5 Model Agnostic Explanation: Context-Aware vulnerable code representation learning for statement S27 in graph-based vulnerability detection.

5.2.2.2.1 Graph-Based vulnerability detection model (Sub-Section 5.2.3).

As seen in Sub-Section 5.2.2.1, a vulnerability is usually exhibited as multiple statements being exploited, thus, it is natural to capture the vulnerable code as a sub-graph in the PDG with the data and control flows. This also helps developers further investigate the detected vulnerability with those flows. Toward that goal, we model the vulnerability detection via the Graph Convolutional Network (GCN) [63] as follows. The PDG of a method M is represented as a graph $G_N = (V, E)$ in which V is a set of nodes representing the statements, and E is a set of edges representing the data/control dependencies. A feature description x_V is for every node v, which represents a property of a node, *e.g.*, variable name, *etc.* Features are summarized in a $N \times D$ feature matrix X_M (N: number of nodes and D is the number of input features). Let f be a label function on the statements and methods $f : V \rightarrow \{1, ..., C\}$ that maps a node in V and an entire method to one of the C classes. In IVDetect, C=2 for vulnerable ($\mathcal{V}$) and non-vulnerable ($\mathcal{NV}$).

For training on (non-)vulnerable code in the training set, GCN performs similar operations as CNN where it learns the features with a small filter/window sliding

over PDG sub-structure. Differing from image data with CNN, the neighbors of a node in GCN are unordered and variable in size. To predict if a method M is vulnerable, its PDG G_M with the associated feature set $X_M = \{x_j | v_j \in G_M\}$ are built. GCN learns a conditional distribution $P(Y|G_M, X_M)$, where Y is a random variable representing the labels $\{1, ..., C\}$. That distribution indicates the probability of the graph G_M belonging to each of the classes $\{1, ..., C\}$, *i.e.*, M is vulnerable or not (Sub-Section 5.2.3).

5.2.2.2.2 Distinction between vulnerable statements and surrounding contexts. During training, for each vulnerable statement s in a method in the training dataset, we distinguish s and the surrounding contextual statements for s. A context consists of the statements with data and/or control dependencies with s. This is expected to help our model recognize better the vulnerable code appearing in specific surrounding contexts, and discriminate better the vulnerable code from the benign one. For example, the existing approaches feed the entire PDG of the method in Figure 5.3 into a model. IVDetect distinguishes and learns the vector representation for the vulnerable statement at line 27 while considering as contexts the statements with data/control dependencies with line 27: the data-dependency context (lines 31, 22, 13, 10, and 6), and the control-dependency context (lines 29, 25, 23, and 13).

5.2.2.2.3 Graph-Based interpretation model for vulnerability detection (Sub-Section 5.2.4). After prediction, IVDetect performs fine-grained interpretation. It uses both the PDG G_M of the method M and the GCN model as the input to obtain the interpretation. To that end, we leverage the interpretable ML technique *GNNExplainer* [184]. Its goal is to take the GCN and a specific input graph G_M, and produce the *crucial sub-graph structures and features* in G_M that affect the decision of the model. GNNExplainer's idea is that *if removing or altering a node/feature does affect the prediction outcome, the node/feature is considered as*

essential and thus must be included in the crucial set (let us call it the *interpretation* set). GNNExplainer searches for a sub-graph $\mathcal{G}_M$ in G_M that minimizes the difference in the prediction scores between using the whole graph G_M and using the minimal graph $\mathcal{G}_M$ (Sub-Section 5.2.4). Because without that subgraph $\mathcal{G}_M$ in the input PDG G_M, GCN model would not decide G_M as vulnerable, $\mathcal{G}_M$ is considered as crucial **PDG sub-graph** consisting of **crucial statements** and data/control dependencies relevant to the detected vulnerability (if the outcome is $\mathcal{V}$). If the outcome is non-vulnerability, $\mathcal{G}_M$ can be considered as the safe statements in PDG for the model to decide the input method M as benign code.

5.2.3 Graph-Based vulnerability detection model

This sub-section describes our graph-based vulnerability detection model. We first explain how we build the context-aware representation learning for vulnerable code, and then how we use such learned vectors for vulnerability detection using FA-GCN [139].

5.2.3.1 Context-Aware representation learning. Let us present how we build the vector representations for code features. For a *statement*, we extract the following types of **features**:

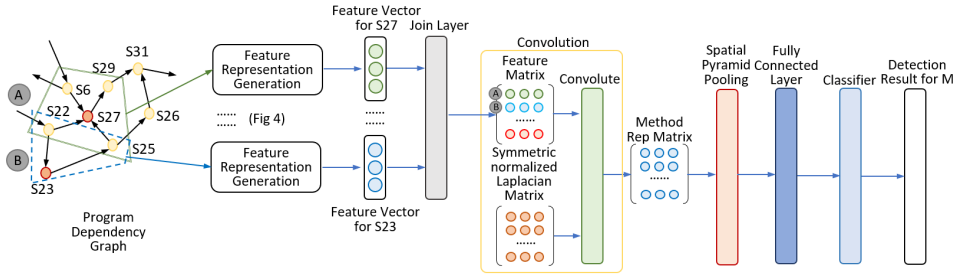


Figure 5.6 Model Agnostic Explanation: Vulnerability detection with FA-GCN.

5.2.3.1.1 Sequence of sub-tokens of a statement. At the lexical level, we capture the content of a statement in term of the sequence of sub-tokens. We choose

the sub-token granularity because the sub-tokens are more likely to be repeated than the entire lexical tokens in source code [113].

We tokenize each statement and keep only the variables, method and class names. The names are broken into sub-tokens using CamelCase or Hungarian convention. We remove the sub-tokens with one character to avoid the influence of noises. For example, in Figure 5.5, the tokens of S_{27} are collected and broken down into the sequence: *copy*, *to*, *user*, *arg*, *etc.* Then, we use GloVe [122], to build the vectors for tokens, together with Gate Recurrent Unit (GRU) [28] to build the feature vector for the sequence of sub-tokens for S_{27}. GloVe is known to capture well semantic similarity among tokens. GRU is chosen to summarize the sequence of vectors into one feature vector for the next step.

5.2.3.1.2 Code structure of a statement. We capture code structure via the AST sub-tree. In Figure 5.5, the AST sub-tree for S_{27} is extracted and fed to Tree-LSTM [152] to capture the structure into a vector F_2.

5.2.3.1.3 Variables and types. For each node (*i.e.*, a statement), we collect the names of the variables and their static types at their locations, break them into the sub-tokens. For example, we collect the variable *s_cmd* and its static type *cross_ec_command*. We use the same vector building techniques as for the sub-token sequences as in the feature 1, including GloVe and GRU, to apply on the sequences of the sub-tokens built from the variables' names (*e.g.*, *s_cmd*) and those from the variables' types (*e.g.*, *cross_ec_command*).

5.2.3.1.4 Surrounding contexts. During training, for a statement s, we also encode the statements surrounding s, which we refer to as *context*. We have two contexts. Data- and control-dependency contexts contain the statements having such dependencies with the current statement. For example, the data-dependency context for S_{27} includes the statements at the lines 31, 22, 13, 10, and 6. If the control dependencies are considered, the statements with control dependencies with S_{27} at

the lines 29, 25, 23, and 13 are included. The vectors for the statements in the context are calculated via GloVe and GRU as described earlier. Because the number of dependencies could be different, the lengths of the GRU model inputs could be different. Thus, we apply zero padding with a masking layer, allowing the model to skip the zeros at the end of the sequence of sub-tokens. Those zeros will not be included in training.

5.2.3.1.5 Attention-Based Bidirectional GRU. After having all vectors for the features F_1, F_2, ..., we use a bi-directional GRU and an attention layer to learn the weight vector W_i for each feature F_i, based on the hidden states from that model. Then, we compute the weighted vector for each feature by multiplying the original vector for the feature by the weight, that is, we have $F_i' = W_i.F_i$.

Finally, we need to consider the impacts from the *dependent statements to the current statement in the PDG*. The rationale is that those neighboring statements in the PDG must have the influence on the current statement if one of them is vulnerable. For example, the neighboring statements for S_{27} in the PDG include the statements at lines 6, 22, 25, and 29. Thus, we combine and summarize them into the final feature vector F_{S27} for the statement S_{27} as follows:

$$F_{S27} = \sum_i W_i Concat(h(F_i', j)) \tag{5.1}$$

W_i is the trainable weight for combination; $Concat$ is the concatenate layer to link all values into one vector; h is the hidden layer to summarize vector into a value; i = S6, S22, S25, S27, S29; j is feature index. F_{27} is used in the next step with GCN model for detection.

5.2.3.2 Vulnerability detection with FA-GCN. Figure 5.6 presents how we use Feature-Attention GCN model (FA-GCN) [139] for detection. The rationale is that FA-GCN can deal well with the graphs with sparse features (not all the statements share the same properties), and potentially noisy features in a PDG. First,

we parse the method M into PDG. Similar to CNN using the filter on an image, FA-GCN performs sliding a small window along all the nodes (statements) of the PDG. For example, in Figure 5.6, the window marked with Ⓐ for the node $S27$ consists of itself and the neighboring statements/nodes $S6$, $S22$, $S25$, and $S29$. Another window (marked with Ⓑ) is for the node $S23$, including itself and the neighboring nodes: $S22$ and $S25$. For each window, FA-GCN generates the feature representation matrix for the statement at the center. For example, for the window centered at $S27$, it generates the feature vector F_{S27} for $S27$, using the process explained in Figure 5.5. From the representation vectors for all statements, FA-GCN uses a join layer to link all these vectors into the Feature Matrix $\mathcal{F}_m$ for method M. A row in $\mathcal{F}_m$ corresponds to a window in PDG.

Next, FA-GCN performs the convolution operation by first calculating the symmetric normalized Laplacian matrix $\tilde{A}$ [63], and then calculating the convolution to generate the representation matrix M_m for the method m. After that, we use the traditional steps as in a CNN model: using a spatial pyramid pooling layer (to normalize the method representation matrix into a uniform size, and reduce its total size), and connecting its output to a fully connected layer to transform the matrix into a vector V_m to represent m. With V_m, we perform classification by using two hidden layers (controlling the length of vectors and output) and a softmax function to produce a prediction score for m. We use those scores as *vulnerability scores to rank the methods* in a project. The decision for m as $\mathcal{V}$ or $\mathcal{NV}$ is done via a trainable threshold on the prediction score [79, 77].

5.2.4 Graph-Based interpretation model

Let us explain how we use GNNExplainer [184] to build our graph-based interpretation. The input includes the trained FA-GCN model, the PDG (G_M) of the method M, and the detection result $\mathcal{V}$ or $\mathcal{NV}$, and prediction score. Figure 5.7 illustrates our process for the case of $\mathcal{V}$ (Vulnerable) (the case of $\mathcal{NV}$ is done similarly).

To derive the interpretations, the key goal is to find a sub-graph $\mathcal{G}_M$ in the PDG G_M of the method M that minimizes the difference in the prediction scores between using the entire graph G_M and using the minimal graph $\mathcal{G}_M$. To do so, we use GNNExplainer with the *masking technique* [184], which treats the searching for the minimal graph $\mathcal{G}_M$ as a learning problem of the *edge-mask* set EM of the edges. The idea is that learning EM helps IVDetect derive the interpretation sub-graph $\mathcal{G}_M$ by masking-out the edges in EM from G_M ("masked-out" is denoted by $\odot$):

$$\mathcal{G}_M = G_M \odot EM \tag{5.2}$$

Figure 5.7 illustrates GNNExplainer's principle. As an edge-mask set is applied, GNNEXplainer checks if the FA-GCN model produces the same result (in this case the result is $\mathcal{V}$). If yes, the edge in the edge-mask is not important and is not included in $\mathcal{G}_M$. Otherwise, the edge is important and included in $\mathcal{G}_M$. Because the numbers of possible sub-graphs and the edge-mask sets are untractable, GNNExplainer uses a learning approach for the edge-mask EM.

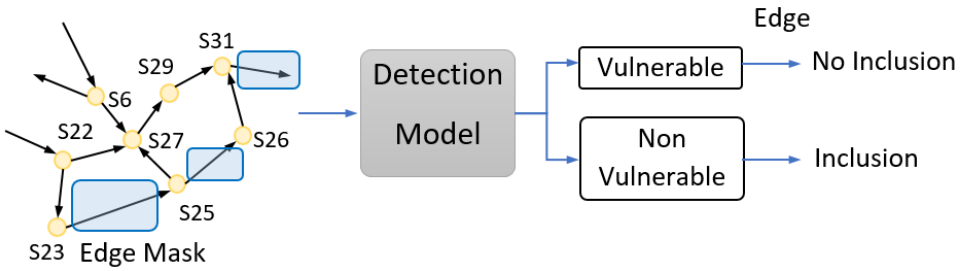


Figure 5.7 Model Agnostic Explanation: Masking to derive interpretation sub-graphs.

Let us formally explain how GNNEXplainer [184] works. It formulates the problem by maximizing the mutual information (MI) between the minimal graph $\mathcal{G}_M$ and the input PDG G_M:

$$\max_{\mathcal{G}_M} MI(Y, \mathcal{G}_M) = H(Y) - H(Y|G = \mathcal{G}_M) \tag{5.3}$$

Y is the outcome decision by the FA-GCN model. Thus, the entropy term $H(Y)$ is constant for the trained FA-GCN model. Maximizing the MI value for all $\mathcal{G}_M$ is equivalent to minimizing conditional entropy $H(Y|G = \mathcal{G}_M)$, which by definition of conditional entropy can be expressed as

$$-\mathbb{E}_{Y|\mathcal{G}_M}[logP_{FA-GCN}(Y|G = \mathcal{G}_M] \tag{5.4}$$

The meaning of this conditional entropy formula is a measure of how much uncertainty remains about the outcome Y when we know $G = \mathcal{G}_M$. GNNEXplainer also limits the size of $\mathcal{G}_M$ by K_M, *i.e.*, taking K_M edges that give the highest mutual information with the prediction outcome Y. Direct optimization of the formula 5.4 is not tractable, thus, GNNExplainer treats $\mathcal{G}_M$ as a random graph variable $\mathcal{G}$. The objective in Equation 5.4 becomes:

$$\min_{\mathcal{G}} \mathbb{E}_{\mathcal{G}_M \sim \mathcal{G}}\, H(Y|G = \mathcal{G}_M) \tag{5.5}$$

$$\min_{\mathcal{G}} H(Y|G = \mathbb{E}_{\mathcal{G}}[\mathcal{G}_M]) \tag{5.6}$$

From Equation 5.5, we obtain Equation 5.6 with Jensen's inequality. The conditional entropy in Equation 5.6 can be optimized by replacing $\mathbb{E}_{\mathcal{G}}[\mathcal{G}_M]$ to be optimized by masking with EM on the input graph G_M. Now, we can reduce the problem to learning the mask EM. Details on training can be found in [184]. The resulting sub-graph $\mathcal{G}_M$ is directly used as an interpretation. We can similarly produce the interpretations for the cases of non-vulnerability result.

5.2.5 Empirical evaluation

5.2.5.1 Research questions. To evaluate IVDetect, we seek to answer the following questions:

RQ1. Comparison on Method-Level Vulnerability Detection (VD). How well does IVDetect perform in comparison with the state-of-the-art method-level Deep Learning VD approaches?

RQ2. Comparison with other Interpretation Models for Fine-Grained VD Interpretation. How well does IVDetect perform in comparison with the state-of-the-art interpretation models for fine-grained VD interpretation to point out vulnerable statements?

RQ3. Vulnerable Code Patterns and Fixing Patterns. Is IVDetect useful in detecting vulnerable code patterns and fixes?

RQ4. Sensitivity Analysis for Internal Features. How do internal features affect the overall performance of IVDetect?

RQ5. Sensitivity Analysis on Training Data. How do different data splitting schemes affect IVDetect's performance?

RQ6. Time Complexity. What is time complexity of IVDetect?

5.2.5.2 Datasets. To empirically evaluate IVDetect, we have conducted several experiments on three public vulnerability datasets including Fan *et al.*'s [35], Reveal [22], and FFMPeg+Qemu [198] (Table 5.1). Fan *et al.* [35] dataset covers the CWEs from 2002 to 2019 with 21 features for each vulnerability. At the method level, the dataset contains +10K vulnerable methods and fixed code. The Reveal dataset [22] contains +18K methods with 9.16% of them being vulnerable ones. The FFMPeg+Qemu dataset has been used in Devign study [198] with +22K methods, and 45.0% of the entries are vulnerable.

Table 5.1 Model Agnostic Explanation: Experimental Datasets.

Dataset	Fan	Reveal	Devign
Vulnerabilities	10,547	1,664	10,067
Non-Vulnerabilities	168752	16505	12,294
Ratio (Vul:Non-Vul)	1:16	1:9.9	1:1.2

5.2.5.3 Experimental methodology.

5.2.5.3.1 RQ1. Comparison on method-level DL-Based VD approaches.

Baselines. We compare IVDetect with the state-of-the-art DL-based vulnerability detection approaches: 1) **VulDeePecker** [79]: a DL-based approach using Bidirectional LSTM on the statements and their data/control dependencies. 2) **Devign** [198]: an DL-based approach that uses GGCN model with Gated Graph Recurrent Layers on the AST, CFG, DFG, and code sequences for graph classification. 3) **SySeVR** [78]: in addition to statements and program dependencies, this approach also uses program slicing and leverages several DL models (LR, MLP, DBN, CNN, LSTM, *etc.*). 4) **Russell** *et al.* [132]: this DL approach encodes source code as matrices of code tokens and leverages convolution model with random forest (RF) via ensemble classifier. 5) **Reveal** [22]: this approach uses GGNN, MLP, and with Triplet Loss on graph representations of source code.

Procedure. A dataset contains a number of vulnerable and non-vulnerable methods. We randomly split all of its vulnerable methods into 80%, 10%, and 10% to be used for training, tuning, and testing, respectively. For training, we added to that 80% part the same number of non-vulnerable methods as the vulnerable ones to obtain a balanced training data. For tuning and testing, we also added the non-vulnerable methods but we used the real ratio between vulnerable and non-vulnerable methods in the original dataset to build the tuning/testing data. We used AutoML [98] on all models to automatically tune hyper-parameters on the tuning dataset.

We also performed the evaluation across the datasets. We first trained our model on the combination of two datasets *Reveal* and *FFMPeg+Qemu*, which has a balanced number of vulnerable methods and non-vulnerable ones. We then tested the model on *Fan* dataset, which has a more realistic ratio of vulnerable and non-vulnerable methods. To ensure the model suitable for cross-data evaluation, we also used 20%

of *Fan* dataset for tuning the parameters and performed prediction on the remaining 80%.

Evaluation Metrics. We use the following evaluation metrics.

Mean Average Precision $MAP = \frac{\sum_{q=1}^{Q} AvgP(q)}{Q}$, with *Average Precision* $AvgP = \sum_{k=1}^{n} P(k)rel(k)$, where n is the total number of results, k is the current rank in the list, $rel(k)$ is an indicator function equaling to 1 if the item at rank k is actually vulnerable, and to zero otherwise. Q is the total number of classification types. It is 1 because we only have two types including vulnerable and non-vulnerable classes, however, we rank all the methods based on their scores (1 indicates vulnerable, and 0 otherwise).

Normalized DCG at k: $nDCG_k = \frac{DCG_k}{IDCG_k}$, with *Discounted Cumulative Gain* at rank k, $DCG_k = \sum_{i=1}^{k} \frac{r_i}{log_2(i+1)}$; and *Ideal DCG* at k $IDCG_k = \sum_{i=1}^{|R_k|} \frac{2^{r_i}-1}{log_2(i+1)}$; where r_i is the score of the result at position i, and R_k the rank of the actual vulnerable methods (ordered by their scores) in the resulting list up to the position k.

First Ranking (FR) is the rank of the first correctly predicted vulnerable method. ***Average ranking*** (AR) is the average rank of the correctly predicted vulnerable methods in the top-ranked list.

Accuracy under curve (AUC) is defined as $AUC = P(d(m_1) > d(m_2))$ in which P is the probability, d is the detection model (can be regarded as a binary classifier), m_1 is a randomly chosen positive instance, and m_2 is a randomly chosen negative instance.

Precision (P) is the ratio of relevant instances among the retrieved ones. It is calculated as $Precision = \frac{TP}{TP+FP}$ where TP is the number of true positives and the FP is the number of false positives.

Recall (R) is the ratio of relevant instances that were retrieved. It is calculated as $Recall = \frac{TP}{TP+FN}$ where TP is the number of true positives and the FN is the number of false negatives.

F-score (F) is the harmonic mean of precision and recall. It is calculated as $Fscore = 2\frac{Precision*Recall}{Precision+Recall}$.

5.2.5.3.2 RQ2. Comparison with other interpretation models for fine–grained interpretation.

Baselines. We compare IVDetect with the following interpretation models: 1) **ATT** [184]: this model is a graph attention network that uses the attention mechanism to evaluate the weights (importance degrees) of the edges in the input graph; 2) **GRAD** [184]: this approach uses a gradient-based method that computes the gradient of the GNN's loss function *w.r.t.* the adjacency matrix.

Procedure. Our goal here is to evaluate how well IVDetect produces the fine-grained interpretations pointing to vulnerable statements. Thus, to train/test the interpretation model, we need to use the *Fan* dataset because it contains the vulnerable statements and respective fixes. The other two datasets contain only the vulnerabilities at the method level and no fixes. Therefore, in this RQ2, for the vulnerability prediction part, we used the FA-GCN model that was trained on *Reveal* and *FFMPeg+Qemu* and predicted on the *Fan* dataset. For the methods that are vulnerable, but predicted as non-vulnerable, we considered those cases as incorrect because the resulting interpretations do not make sense for incorrect detection. For the methods that are actually non-vulnerable (regardless of the prediction results of vulnerable or non-vulnerable), we could not use them because the non-vulnerable methods do not have the fixed statements as the ground truth for measuring the correctness of interpretations. Thus, we use the set of methods that are vulnerable and correctly detected as vulnerable for the evaluation of the interpretation model. Let us use D to denote this set.

For the interpretation, we randomly split D into 80%, 10%, and 10% for training, tuning, and testing. For training, we used *the fixed statements* as the labels for interpretation because those fixed statements were vulnerable. For testing, we

compared the relevant statements from the interpretation model against the actual fixed statements. Each method in the testing set and the trained FA-GCN model are the input of the interpretation model in this RQ2.

Evaluation Metrics. Given an interpretation sub-graph $\mathcal{G}_M$ generated from the graph-based interpretation model, we evaluate the accuracy of the interpretation for a model as follows. For a method, if $\mathcal{G}_M$ has an overlap with any statement in the code changes that fix the vulnerability, $\mathcal{G}_M$ is considered as a correct interpretation, *i.e.*, relevant to that detected vulnerability. We then calculate *Accuracy* as the ratio between the number of correct interpretations over the total number of interpretations. Because code changes could include addition, deletion, and modification, we further define such overlap as follows.

If one of the statements S in the vulnerable version was *deleted* or *modified* for fixing, and if $\mathcal{G}_M \ni S$, then we consider the interpretation sub-graph $\mathcal{G}_M$ as correct, otherwise as incorrect. If one of the statements S' was *added* to the vulnerable version for fixing, we check on the fixed version whether $\mathcal{G}_M$ contains any statement with data or control dependencies with S'. If yes, we consider it as correct, otherwise, as incorrect. In Figure 5.3, $\mathcal{G}_M$ contains the statement S23 having the data/control dependencies with one of the added lines from 17–21. Thus, $\mathcal{G}_M$ is correct. The rationale is that if the interpretation sub-graph $\mathcal{G}_M$ contains some statement relevant to the added statement to fix the vulnerability, that interpretation is useful in pointing out the code relevant to the vulnerability.

We also use ***Mean First Ranking*** (MFR), *i.e.*, the mean of the rankings for the first statement that needs to be fixed in the interpretation statements, and ***Mean Average Ranking*** (MAR), *i.e.*, the mean of the rankings for all statements to be fixed in the interpretation statements. If a statement to be fixed has not been selected as interpretation, we do not consider it when calculating MFR/MAR.

5.2.5.3.3 RQ3. Vulnerable code patterns and fixing patterns.

Procedure. We use a mining algorithm on the set of interpretation sub-graphs to mine patterns of vulnerable code. We also mine fixing patterns for those vulnerabilities. See details in Sub-Section 5.2.6.3.

Evaluation Metrics. We counted the identified patterns.

5.2.5.3.4 RQ4. Sensitivity analysis for features.

Procedure. We first built a base model with only the feature that represents the code as the sequence of tokens. We then built other variants of our model by gradually adding one more feature in Sub-Section 5.2.3.1 to the base model including the sequence of sub-tokens, AST subtree, variable names, data dependencies, and control dependencies. We measured the accuracy for each variant. We used the *Fan* dataset and the same experiment setting as in RQ1.

Evaluation Metrics. We use the same metrics as in RQ1.

5.2.5.3.5 RQ5. Sensitivity analysis on training data. We used different ratios in data splitting for training, tuning, and testing: (80%, 10%, 10%), (70%, 15%, 15%), (60%, 20%, 20%), and (50%, 25%, 25%). We used the *Fan* dataset and the same setting as in RQ1.

Evaluation Metrics. We use the same metrics as in RQ1.

5.2.5.3.6 RQ6. Time complexity analysis. We measure the actual training and predicting time.

5.2.6 Experimental results

5.2.6.1 RQ1. Comparison on method-level VD. In Table 5.2, among the top 10 prediction results, IVDetect has the most correct predictions (6 vulnerable methods). The vulnerable methods correctly detected by IVDetect are also pushed higher in the top-10 ranked list with 4 correct results out of top-5 results. All other baselines have only 0–1 correct detection in the top-5 list. Importantly, the first rank

for IVDetect (*i.e.*, the rank of the first correctly detected vulnerable methods) is 1^{st}, while those of the baselines are 4^{th}, 5^{th}, 5^{th}, 6^{th}, and 7^{th} (the bold values in Table 5.2). Moreover, IVDetect can detect 14, 35, and 64 vulnerabilities among top-20, top-50, and top-100 prediction results.

Table 5.2 Model Agnostic Explanation: RQ1. Top-10 Vulnerability Detection Results on FFMPeg+Qemu Dataset.

Top-10 Rank	1	2	3	4	5	6	7	8	9	10	Total
VulDeePecker	0	0	0	0	0	0	**1**	0	1	1	3
SySeVR	0	0	0	0	0	**1**	1	1	0	1	4
Russell *et al.*	0	0	0	0	**1**	0	1	0	1	1	4
Devign	0	0	0	0	**1**	0	1	1	1	0	4
Reveal	0	0	0	**1**	0	1	0	1	1	1	5
IVDetect	**1**	0	1	1	1	0	1	1	0	0	6

Notes: 0: incorrect, 1: correct.

Tables 5.3, 5.4, and 5.5 show the comparison among the approaches on three datasets. IVDetect consistently performs better in all the metrics (Table 5.3). For nDCG@{1,3}, all the baselines get zeros because they did not have correct detections in top-3 results. IVDetect can improve nDCG@10 from 43%–84% and nDCG@20 from 37%–71% as compared to the baselines. Higher nDCG indicates that IVDetect achieves the ranking closer to the perfect ranking and the correct vulnerable methods appear higher in the top list.

For MAP scores, IVDetect relatively improves over the baselines from 105%–255% for top-10 and from 53%–116% for top-20 accuracy. With higher MAP, IVDetect has higher precision on average for all the top-ranked positions in the top list. That is, the top-ranked result is highly precise in detecting the vulnerable methods.

Table 5.3 Model Agnostic Explanation: RQ1. Method-Level VD on FFMPeg + Qemu Dataset.

	VulDee--Pecker	SySeVR	Russell *et al.*	Devign	Reveal	IVDetect
nDCG@1	0	0	0	0	0	1
nDCG@3	0	0	0	0	0	0.63
nDCG@5	0	0	0.43	0.45	0.5	0.65
nDCG@10	0.37	0.44	0.45	0.46	0.5	0.68
nDCG@15	0.45	0.48	0.49	0.52	0.55	0.75
nDCG@20	0.48	0.51	0.54	0.56	0.6	0.82
MAP@1	0	0	0	0	0	1
MAP@3	0	0	0	0	0	0.83
MAP@5	0	0	0.20	0.20	0.25	0.80
MAP@10	0.22	0.31	0.30	0.32	0.38	0.78
MAP@15	0.29	0.33	0.34	0.37	0.41	0.72
MAP@20	0.32	0.35	0.37	0.42	0.45	0.69
FR@1	n/a	n/a	n/a	n/a	n/a	1
FR@3	n/a	n/a	n/a	n/a	n/a	1
FR@5	7	6	5	5	4	1
FR@10	7	6	5	5	4	1
FR@15	7	6	5	5	4	1
FR@20	7	6	5	5	4	1
AR@1	n/a	n/a	n/a	n/a	n/a	1
AR@3	n/a	n/a	n/a	n/a	n/a	2
AR@5	n/a	n/a	5	5	4	3.3
AR@10	8.7	7.8	7.8	7.4	7.4	4.7
AR@15	11.2	10	9.5	10	9.1	7.6
AR@20	13.3	12.1	12.6	12.1	12.4	10.3
AUC	0.68	0.72	0.79	0.77	0.79	0.84

Table 5.4 Model Agnostic Explanation: RQ1. Method-Level VD on Fan Dataset.

	VulDee--Pecker	SySeVR	Russell *et al.*	Devign	Reveal	IVDetect
nDCG@1	0	0	0	0	0	0
nDCG@5	0	0	0	0	0	0.5
nDCG@10	0	0	0.30	0.33	0.34	0.43
nDCG@15	0	0	0.28	0.30	0.37	0.45
nDCG@20	0.08	0.23	0.31	0.32	0.38	0.46
MAP@1	0	0	0	0	0	0
MAP@5	0	0	0	0	0	0.25
MAP@10	0	0	0.1	0.13	0.18	0.27
MAP@15	0	0	0.12	0.14	0.21	0.28
MAP@20	0.08	0.24	0.14	0.15	0.20	0.28
FR@1	n/a	n/a	n/a	n/a	n/a	n/a
FR@5	n/a	n/a	n/a	n/a	n/a	4
FR@10	n/a	n/a	10	8	6	4
FR@15	n/a	n/a	10	8	6	4
FR@20	19	16	10	8	6	4
AR@1	n/a	n/a	n/a	n/a	n/a	n/a
AR@5	n/a	n/a	n/a	n/a	n/a	4
AR@10	n/a	n/a	10	8	8	7.3
AR@15	n/a	n/a	12	10.5	9.3	8.5
AR@20	19.5	18	13.3	13.3	12	10.4
AUC	0.72	0.81	0.82	0.75	0.82	0.9

Table 5.5 Model Agnostic Explanation: RQ1. Method-Level VD on Reveal Dataset.

	VulDee--Pecker	SySeVR	Russell *et al.*	Devign	Reveal	IVDetect
nDCG@1	0	0	0	0	0	0
nDCG@3	0	0	0	0	0	0.63
nDCG@5	0	0	0	0	0.43	0.53
nDCG@10	0	0.30	0.32	0.34	0.39	0.52
nDCG@15	0.26	0.28	0.32	0.39	0.42	0.55
nDCG@20	0.27	0.33	0.35	0.43	0.48	0.57
MAP@1	0	0	0	0	0	0
MAP@3	0	0	0	0	0	0.33
MAP@5	0	0	0	0	0.2	0.37
MAP@10	0	0.11	0.11	0.18	0.24	0.34
MAP@15	0.07	0.12	0.16	0.23	0.25	0.36
MAP@20	0.11	0.15	0.18	0.36	0.29	0.37
FR@1	n/a	n/a	n/a	n/a	n/a	n/a
FR@3	n/a	n/a	n/a	n/a	n/a	3
FR@5	n/a	n/a	n/a	n/a	5	3
FR@10	n/a	10	9	7	5	3
FR@15	15	10	9	7	5	3
FR@20	15	10	9	7	5	3
AR@1	n/a	n/a	n/a	n/a	n/a	n/a
AR@3	n/a	n/a	n/a	n/a	n/a	3
AR@5	n/a	n/a	n/a	n/a	5	4
AR@10	n/a	10	9	8	6	6
AR@15	15	12.5	12	10.5	9.8	9.5
AR@20	18	15.5	13.3	12.7	13	11.8
AUC	0.65	0.76	0.75	0.72	0.74	0.81

IVDetect also achieves better first ranking (FR) and average ranking (AR). While its best FR is 1, that of next best performer is 4. For AR@10, a correct vulnerable method is on average ranked by IVDetect 2.7–4.0 positions higher in the ranked list than by the baselines. Our tool also has relatively higher AUC from 6%–24%.

The comparative results on *Fan* and *Reveal* datasets are similar (Tables 5.4 and 5.5). In *Fan* dataset, IVDetect can improve the nDCG and MAP scores over the baselines by 26%–43%, 50%–170% for top-10, and 21%–475%, 40%–250% for top-20. IVDetect's FRs and ARs are better from 2–6 positions and 0.7–2.7 positions for top 10, and 2–13 positions and 1.6–9.1 positions for top 20. In *Reveal* dataset, the improvements in nDCG, MAP, FR, and AR are 33%–73%, 42%–209%, 2–7 positions, and 0–4 positions for top 10, and 19%–111%, 28%–236%, 2–12 positions, and 1.2–6.2 positions for top 20.

Table 5.6 Model Agnostic Explanation: RQ1. Precision and Recall Results of Method-Level VD on Three Datasets.

	FFMPeg+Qemu			Fan			Reveal		
	P	R	F	P	R	F	P	R	F
VulDeePecker	0.49	0.27	0.35	0.12	0.49	0.19	0.19	0.14	0.17
SySeVR	0.50	0.66	0.56	0.15	**0.74**	0.27	0.24	0.42	0.31
Russell *et al.*	0.55	0.41	0.45	0.16	0.48	0.24	0.26	0.12	0.16
Devign	0.52	0.63	0.57	0.18	0.52	0.26	0.33	0.32	0.32
Reveal	0.55	**0.73**	0.62	0.19	**0.74**	0.30	0.31	**0.58**	0.40
IVDetect	**0.60**	0.72	**0.65**	**0.23**	0.72	**0.35**	**0.39**	0.52	**0.45**

Notes: P: Precision; R: Recall; F: F score.

The results on three datasets are different due to the ratio between the vulnerable and non-vulnerable methods. That ratio is 1:16 and 1:9.9 in *Fan* and

Reveal datasets. That number is 1:1.2 in *FFMPeg+Qemu* dataset, thus, there are more vulnerable methods, and the results are consistently higher across all the models.

Table 5.6 shows the Precision and Recall results of IVDetect and the baselines. Specifically, IVDetect has higher precision than all the baselines across all three datasets. It can improve Precision by 2.6%-105%. For Recall, it is marginally worse than Reveal on Fan and FFMPeg+Qemu datasets (1.4% and 2.7%), and worse than SySeVR on Fan dataset (2.7%). On the Reveal Dataset, IVDetect outperforms Reveal by 25.8% in terms of Precision, but has lower Recall by 10.3%. However, in terms of F-score, IVDetect can outperform the best performing baseline, Reveal, by 4.8% on FFMPeg+Qemu dataset, 16.7% on Fan dataset, and 12.5% on Reveal Dataset.

Figure 5.8 shows that IVDetect consistently has better MAP and nDCG scores when considering top-1 to top-100 ranked lists.

For cross-dataset validation, as seen in Figure 5.9, the results for MAP and nDCG in the within-dataset setting are better than those in the cross-dataset setting. This is expected because the model might see similar vulnerable code before in the same projects in the same dataset. The FR and AR values for the cross-dataset setting are one rank higher than those of the within-dataset setting.

(a) MAP Scores. **(b)** nDCG Scores.

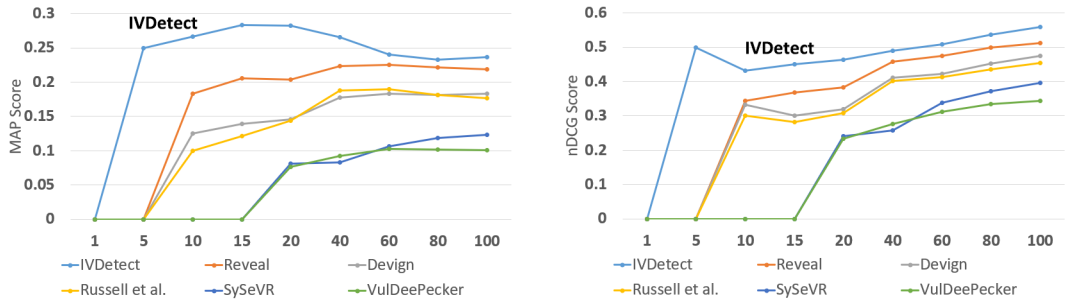


Figure 5.8 Model Agnostic Explanation: Scores from top 1 to top 100 on Fan dataset.

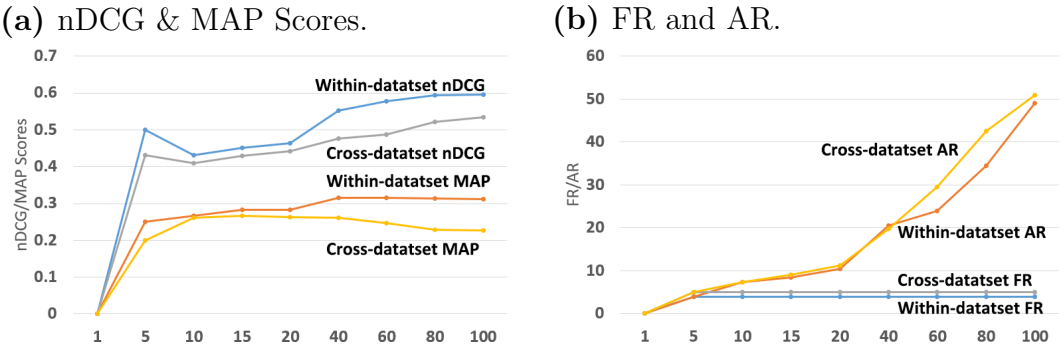


Figure 5.9 Model Agnostic Explanation: RQ1. Cross-Dataset validation: training on Reveal and FFMPeg+Qemu datasets, testing on Fan dataset.

Figure 5.10 shows our analysis on the overlapping results between IVDetect and the baselines on *Fan* dataset for top-100. As seen, IVDetect can detect 17, 13, 13, 11, and 10 vulnerable methods that VulDeePecker, SySeVR, Russell, Devign, and Reveal missed, respectively, while they can detect only 2,3, 4, 5, and 5 vulnerable methods that IVDetect missed. In summary, IVDetect can detect 15, 10, 9, 6, and 5 more vulnerable methods than the baselines.

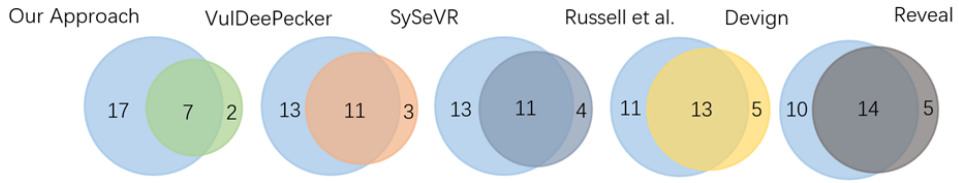


Figure 5.10 Model Agnostic Explanation: Overlapping analysis.

5.2.6.2 RQ2. Comparison with interpretation models for fine-grained VD interpretation.

Table 5.7 shows the accuracy of different interpretation models. As seen, using GNNExplainer improves over ATT and GRAD from 12.3%–400% and 9.0%–400% in accuracy, respectively, as we vary the size of interpretation sub-graphs (*i.e.*, the number of statements) from 1–10. Higher accuracy indicates that IVDetect can provide better fine-grained vulnerability detection interpretation at the statement level. That is, *in more cases, if IVDetect detects correctly vulnerable methods, it can point out more precisely the vulnerable statements* relevant to the vulnerabilities. For

ranking vulnerable statements, using GNNExplainer improves MFR by 0.7 and 1.3 ranks, and improves MAR by 0.6 and 1.3 ranks over ATT and GRAD.

Table 5.7 Model Agnostic Explanation: RQ2. Fine-Grained VD Interpretation Comparison.

Interp.	Accuracy										MFR	MAR
Model	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10		
ATT	0.01	0.16	0.41	0.54	0.59	0.60	0.62	0.63	0.64	0.65	4.8	6.3
GRAD	0.01	0.19	0.43	0.54	0.59	0.62	0.63	0.65	0.66	0.67	4.2	5.6
GE	0.05	0.30	0.54	0.63	0.67	0.68	0.70	0.72	0.72	0.73	3.5	5.0

Notes: GE: GNNExplainer; Nx: x is the number of nodes in the interpretation.

ATT uses the edge attention in the Graph Attention Network to assign the weights for the edges, while GNNExplainer directly gives a score for the subgraph after masking. Thus, for the case in which there are more than one path from a node to another, the weight for an edge is the average weight of the weights through multiple paths, *i.e.*, ATT might be less precise than GNNExplainer. GRAD computes the gradient of the loss function with respect to the input for computing the weight of an edge. However, such gradient-based approach may not perform well with respect to the discrete inputs (an input graph is represented as an adjacency matrix).

As the number of nodes in $\mathcal{G}_M$ increases, the number of statements covered also increases, accuracy is higher. However, the computation time is higher and developers need to investigate more statements. As seen, when the number of statements is higher than 5, accuracy increases more slowly. Thus, we chose 5 as a default.

5.2.6.3 RQ3. Vulnerable code pattern analysis. This sub-section describes another experiment that we exploit IVDetect's capability of providing the interpretation sub-graphs to mine the patterns of vulnerable code and fixes. A vulnerable code pattern is a fragment of vulnerable code that repeats frequently, *i.e.*, more than

a certain threshold. The detected vulnerability patterns and corresponding fixes are the good resources for developers to learn about the vulnerable code that others have frequently made, and learn to fix the vulnerable code in the same patterns.

From the results in RQ2, we first collected into a set $\mathcal{G}$ the interpretation sub-graphs $\mathcal{G}_M$s with the correctly detected statements as relevant to the vulnerability in the methods. In total, we obtain +700 $\mathcal{G}_M$s. Note that $\mathcal{G}_M$ is a sub-graph of PDG. For each $\mathcal{G}_M$, we abstract out the variables' names with a keyword *VAR*, and the literals with their data types. We then ran the sub-graph pattern mining algorithm [116] on $\mathcal{G}$ with different thresholds of frequencies and collected different sizes of the sub-graph patterns. The outputs are the frequent isomorphic sub-graphs within $\mathcal{G}_M$s, which are considered as vulnerable code patterns because we chose $\mathcal{G}_M$ that contains correct interpretation statements relevant to the correctly detected vulnerabilities. After manual verification, we obtain a number of correct patterns (Table 5.8). As seen, as the frequency threshold or the size of pattern is larger, the number of patterns decreases as expected. When they are both larger than 5, we found no pattern. Let us explain a few examples.

Table 5.8 Model Agnostic Explanation: RQ3. Numbers of Vulnerable Code Patterns.

	thresh=2	thresh=3	thresh=4	thresh=5
size=2	47	36	22	7
size=3	25	27	19	6
size=4	23	22	16	5
size=5	22	21	11	2
Total	117	102	68	21

```
1 // ======================== PATTERN 1 ================================
2 if (is_link(STRINGLITERAL)) {
3      fprintf(stderr, "Error:␣invalid␣/etc/skel/.zshrc␣file\n"); //∉ patt
4     exit(INTLITERAL);
5 }
6 if (copy_file(STRINGLITERAL, VAR) == INTLITERAL) { ...
7 // ======================== PATTERN 2 ================================
8 VAR = udf_get_filename(VAR, VAR, VAR, VAR);
9 if (VAR && ...)  goto LABEL;
```

Figure 5.11 Model Agnostic Explanation: Vulnerable code patterns.

Figure 5.11 shows two examples of vulnerable code patterns. The first pattern (lines 2,4, and 6) shows an API misuse in the project *firejail* involving *is_link(...)*, *exit*, and *copy_file(...)*. The usage is to check the validity of a link, and if yes to copy the file, or otherwise to stop the execution. This pattern appeared three times with different string literals and was fixed by developers to replace the statements. An interesting observation is that IVDetect is able to eliminate the *fprintf* statement at line 2 from the interpretation sub-graph, thus, eliminating it from the pattern, even though the *fprintf* statement appears with the other statements three times in the project. This shows a benefit of IVDetect because if a tool does not have statement-level VD interpretation and it mines pattern from the entire methods, it will incorrectly include *fprintf* in the pattern. The second pattern (lines 8–9) shows a pattern involving a vulnerable method call *udf_get_filename*, and the checking on its return value. The method was later fixed to add the 5^{th} parameter.

```
1  // ====================== FIXING PATTERN 1 ========================
2  - VAR = fl6_update_dst(VAR, VAR, VAR);
3  + rcu_read_lock();
4  + final_p = fl6_update_dst(VAR, rcu_dereference(VAR), VAR);
5  + rcu_read_unlock();
6  // ====================== FIXING PATTERN 2 ========================
7  - char VAR = malloc (VAR);
8  + char VAR;
9  + if (VAR < 0 || VAR > LITCONST) {
10 +     error_line (STRINGLITERAL, VAR);
11 +     return LITCONST;
12 + }
13 + VAR = malloc (VAR);
```

Figure 5.12 Model Agnostic Explanation: Fixing patterns.
Notes: -: removal, +: addition.

Another interesting finding is that IVDetect enables the discovery of not only vulnerable code patterns but also the fixing patterns for them. Figure 5.12 shows two fixing patterns for vulnerable code. The first vulnerability (from Linux kernel), lines 2–5, is about the method *f16_update_dst*(...). According to the commit log, to avoid another thread changing a data record concurrently, developers need to provide mutual exclusion access and deferencing. This fixing pattern was repeated 3 times in the methods *dccp_v6_send_response*, *inet6_csk_route_req*, and *net6_csk_route_socket*. This fixing pattern would be useful for a developer to learn the fix from one method and apply to the other two methods. The second pattern (lines 7–13) shows a fixing pattern to a vulnerability on buffer overflow with the *malloc* call in *ParseDsdiffHeaderConfig* method of WavPack 5.0. According to CVE-2018-7253, this problem *"allows a remote attacker to cause a denial-of-service (heap-based buffer over-read) or possibly overwrite the heap via a maliciously crafted DSDIFF file"*. This fixing pattern occurred three times in the same project.

5.2.6.4 RQ4. Sensitivity analysis for features. Table 5.9 shows the changes to the metrics as we incrementally added each internal feature into our model in Figure 5.5. Generally, each internal feature contributes positively to the better performance of IVDetect, as both the score metrics (nDCG, MAP, and AUC) and the ranking metrics (FR and AR) are improved.

When IVDetect considers only the sequence of tokens (ST) in the code, the first correct detection (FR) is at the position 14, thus, nDCG@{1,5,10}=0 and MAP@{1,5,10}=0 (not shown). When considering the code as the sequence of sub-tokens (SST), IVDetect deals with the unique tokens better because the sub-tokens appear more frequently than the tokens [113]. At top-20, FR improves 2 positions, AR improves 4.5 positions, and nDCG and MAP relatively improve 3.8% and 22.2%. When AST is additionally considered, the model can distinguish vulnerable code structures and statements. At top-20, FR and AR improve 1 and 1.5 positions, and nDCG and MAP improve 7.4% and 18.1%. However, FR is still 11 and nDCG@{1,5,10}=0 and MAP@{1,5,10}=0 (not shown), because tokens and AST do not help much discriminate the vulnerable statements.

The feature on variables also helps improve FR and AR from 11 to 7 and 13.5 to 12.5, and nDCG and MAP relatively improve 27.6% and 46.2% at top 20. nDCG@10 and MAP@10 improve from 0 to 0.33 and to 0.18, respectively (not shown). This feature allows the model to detect similar incorrect variable usages. By additionally integrating control dependencies (CD), FR and AR improve from 7 down to 5 and 12.5 down to 11.2, and nDCG and MAP relatively improve 18.9% and 36.8%. By adding data dependencies (DD), FR and AR improve from 5 to 4 and 11.2 to 10.4. nDCG and MAP improve 4.5% and 7.7% for top 20. This result confirms that vulnerable code often involves the statements with control and/or data dependencies [198, 22].

Table 5.9 Model Agnostic Explanation: RQ4. Evaluation for the Impact of Internal Features.

	ST (A)	(A)+**SST** (B)	(B)+**AST** (C)	(C)+**Var** (D)	(D)+**CD** (E)	(E)+**DD** (F)
nDCG@15	0.25	0.27	0.29	0.35	0.42	0.45
nDCG@20	0.26	0.27	0.29	0.37	0.44	0.46
MAP@15	0.07	0.11	0.12	0.19	0.26	0.28
MAP@20	0.09	0.11	0.13	0.19	0.26	0.28
FR@15	14	12	11	7	5	4
FR@20	14	12	11	7	5	4
AR@15	14	13.5	11	10.3	9	8.5
AR@20	19.5	15	13.5	12.5	11.2	10.4
AUC	0.75	0.76	0.77	0.83	0.85	0.9

Notes: **ST**: sequence of tokens; **SST**: sequence of sub-tokens; **AST**: sub-AST; **Var**: variables; **CD**: control dependencies; **DD**: data dependencies; **F = IVDetect**.

Figure 5.13 shows a detected vulnerable method: *validate_event(...)* was vulnerable and replaced with a new version with an additional parameter. We used the models (A)–(F) for detection, and observed that the rank for *validate_event(...)* in the candidate list improves from 140 (A), to 121 (B), 99 (C), 71 (D), 48 (E), and 19 (F). While the features on tokens, sub-tokens, and AST are contributing, they do not help much because the model did not see them in vulnerable methods before. However, the variable/method names, especially control/data dependencies between the surrounding statements and *validate_event(...)* help discriminate this vulnerability, and push it to the top-20 list. Control dependencies (*e.g.*, between *validate_event(...)* and *return -EINVAL*) help improve 29 ranks. Generally, the improvement in ranking shows the positive contributions of all the features.

```
static int validate_group(struct perf_event *event)
{       ...
-   if (!validate_event(&fake_pmu, leader))
+   if (!validate_event(event->pmu, &fake_pmu, leader))
        return -EINVAL;

    list_for_each_entry(sibling, &leader->sibling_list, group_entry) {
-       if (!validate_event(&fake_pmu, sibling))
+       if (!validate_event(event->pmu, &fake_pmu, sibling))
            return -EINVAL;
    }

-   if (!validate_event(&fake_pmu, event))
+   if (!validate_event(event->pmu, &fake_pmu, event))
        return -EINVAL; ...
}
```

Figure 5.13 Model Agnostic Explanation: A detected vulnerable method in android kernel.

This example also shows a fixing pattern appearing three times with different variables *leader*, *sibling*, and *event*.

5.2.6.5 RQ5. Sensitivity analysis on training data. As seen in Table 5.10, with more training data, the performance is better as expected. Even with 60%/20%/20%, IVDetect still achieves nCDG of 0.43 and MAP of 0.25, which are still higher than those of the other baselines for top 20 (highest nDCG and MAP of the baselines are 0.38 and 0.20). With 20% less training data (60% vs 80%), IVDetect drops AUC only by 5.5%.

Table 5.10 Model Agnostic Explanation: RQ5. Sensitivity Analysis on Training Data.

Train/Tune/Test	nDCG@20	MAP@20	FR@20	AR@20	AUC
40%/30%/30%	0.26	0.09	12	15.5	0.69
50%/25%/25%	0.33	0.16	8	12.3	0.74
60%/20%/20%	0.43	0.25	5	11.6	0.85
70%/15%/15%	0.44	0.26	5	11.2	0.87
80%/10%/10%	0.46	0.28	4	10.4	0.9

5.2.6.6 RQ6. Time complexity. To generate the interpretation sub-graphs for all methods, it takes about 9 days, 2 days, and 3 days to finish on Fan, Reveal, and FFMPeg+Qemu datasets, respectively. It took 23, 7, 10 hours to train IVDetect on Fan, Reveal, and FFMPeg+Qemu datasets. For VD prediction, it takes only 1-2s per method.

5.2.7 Threats to validity

We only tested on the vulnerabilities in C and C++ code. In principle, IVDetect can apply to other programming languages. We tried our best to tune the baselines on same dataset for fair comparisons. We focus only on DL-based VD models.

5.3 Conclusion

In this chapter, we analyze how to use suitable learning code representation to do vulnerability detection and provide a reasonable explanation. To achieve it, We present IVDetect, a novel DL-based approach to provide sub-graphs in PDG, which explains the prediction results of graph-based vulnerability detection. The key ideas that enable our approach are (1) modeling and analyzing the source code using graphs to do the vulnerability detection; (2) analyzing the distinction between

vulnerable statements and surrounding contexts; (3) using sub-graphs to explain the vulnerability detection results.

Through our empirical evaluation on vulnerability databases, we have demonstrated that IVDetect surpasses existing DL-based approaches, providing evidence for the effectiveness of our concept of utilizing appropriate learning code representations in vulnerability detection, accompanied by reasonable explanations.

CHAPTER 6

RELATED WORK

6.1 Bug Detection

Bug Detection Approaches. Many techniques have been developed for rule-based and mining-based bug detection. Some existing rule-based bug detection approaches, such as [13, 55, 117, 34, 29, 155], are unsupervised and very efficient. However, new rules are needed to define to detect new types of bugs, for example, in FindBugs [48]. The mining-based approach in NAR-miner [13] extracts negative rules to detect bugs and outperforms rule-based approaches that are based on mining positive code rules. Our experiment results show that it only costs NAR-miner 1 minute to detect bugs on a project. However, the mining-based approaches mainly suffer from the problem of high false positive (FP) rates, such as the NAR-miner has a high FP rate, i.e., 52% in the cross-project setting, which makes them impractical for daily use. When comparing our approach in the bug detection chapter with the existing state-of-the-art rule-based and mining approaches, the main differences are as follows. First, we consider the relations among paths from different methods for detecting cross-method bugs, while the state-of-the-art approaches normally work on individual methods and cannot work well on cross-method bugs. Second, our approach covers path information in an AST in order to detect very detailed bugs in each method, while rule-based and mining-based approaches often consider the important rules and may miss some information outside of their rules.

There exist machine learning-based bug detection approaches, including the deep learning techniques [125] and traditional machine learning techniques [34, 80, 166, 165, 164, 82]. For example, the Bugram [164] uses n-gram models to rank the methods and then picks the top-ranked methods as buggy methods. DeepBugs [125]

uses deep learning techniques to propose a name-based bug detection approach. In the bug detection chapter, our approach also uses deep learning techniques to train the models and classify methods into buggy or non-buggy. Our approach is different from the existing learning-based approaches in the following ways. First, like the rule-based approaches, the existing learning-based approaches do not consider the relations among paths across multiple methods. In our code representation learning step, we model the relations among paths from different methods using the dependencies of entities in the PDG and DFG, in addition to the AST nodes of a path. Second, our approach uses long paths of an AST to cover all of the AST nodes for representing local context. In contrast, other approaches often use part of method information to detect bugs, such as name-based identifier representation and frequent n-grams. Our results show that our approach can outperform all of the studied baselines.

Learning Code Representation in Bug Detection. The recent success in machine learning has led to strong interest in applying machine learning techniques, especially deep learning, to program language (PL) analysis and software engineering (SE) tasks, such as automated correction for syntax errors [12], fuzz testing with probabilistic, generative models [120], program synthesis [8], code clones [170, 144, 71], program classification and summarization [5, 104], and so on.

All approaches learn code representations using different program properties in the above PL and SE tasks. Although the learned code representations are not proposed for detecting bugs, they are still very relevant to our study, as one important step of our approach is to learn bug detection specialized code representation. Different from the existing code representation learning approaches, in our bug detection chapter, we learn code representation using the AST, Program Dependency Graph and Data Flow Graph, and different attention-based neural networks. More importantly, using an attention mechanism, we incorporate the previous bug fixes into our code representation. Our results show that our code representation is more

suitable for detecting bugs than the studied baselines, such as the baseline code representations using tokens and identifiers.

Fault Localization Approaches. The Spectrum-Based Fault Localization (SBFL), e.g., [189, 3, 56, 1, 106, 176, 83, 93, 58], has been intensively studied in the literature, e.g., Tarantula [57], SBI [83], Ochiai [1] and Jaccard [3], they share the same basic insight, i.e., code elements mainly executed by failed tests are more suspicious. The Mutation-Based Fault Localization (MBFL), e.g., [103, 191, 17, 190, 105], aims to additionally consider impact information for fault localization since code elements covered by failed/passed tests may not have an impact on the corresponding test outcomes. The examples of MBFL are Metallaxis [118, 119] and MUSE [103]. Learning-to-Rank (LtR) has been used to improve fault localization [10, 180, 73, 146]. MULTRIC [180] combines different suspiciousness values from SBFL. Some work combines SBFL suspiciousness values with other information, e.g., program invariant [10] and source code complexity information [146], for more effective LtR fault localization. TraPT [73] combines suspiciousness values from both SBFL and MBFL. Neural networks have been applied to fault localization [197, 15, 194, 175]. However, they mainly work on the test coverage summarization scores, which have clear limitations (e.g., it cannot distinguish elements covered by failing and passing test cases) [73], and are usually studied on artificial faults or small programs. Recently DeepFL [72] was proposed to improve method-level FL, and it improves the most state-of-the-art LtR FL approach, TraPT [73], by 36.54% in terms of Recall at Top-1.

Learning Code Representation in Fault Localization. The recent success in machine learning has led to much interest in applying machine learning, especially deep learning, to program language (PL) analysis and software engineering (SE) tasks, such as automated correction for syntax errors [12], spreadsheet errors detection [141, 11] fuzz testing [120], program synthesis [8], code clones [170, 144, 71], program summarization [5, 104], code similarity [195, 7], probabilistic model for code [14], and

path-based code representation, e.g., Code2Vec [7] and Code2Seq [6]. All approaches learn code representations using different program properties. However, none of the existing fault localization techniques have performed direct code modeling and learning on code coverage information of the test cases for the FL purpose, as in DeepRL4FL, which is our approach mentioned in the fault localization chapter. Moreover, all the existing fault localization techniques cannot simultaneously locate the relevant faulty statements based on their relationship. But our co-change fault localization technique Fixlocator in the fault localization chapter somehow solved this problem.

6.2 Automated Program Repair

In the earlier stage, the APR approaches aimed to automatically derive *the fixes for similar code* cloned from one place to another [114], or similar code due to porting or branching [129]. Ray and Kim [129] automatically detect similar fixes for similar code that are ported or branched out. The code in different branches is almost similar. Thus, the fix can be reused. FixWizard [114] automatically derives *the fixes for similar code* that were cloned from one place to another. It can also work for the *code peers*, which are the code having similar internal and external usages.

A large group of APR approaches has explored *search-based software engineering* to tackle more general types of bugs [70, 126, 68, 96]. First, a search strategy is performed in the space of potential solutions produced by several operators that mutate the buggy code. Then, the test cases and/or program verification are applied to select the better candidate fixes [143]. GenProg [70] uses genetic search on repair mutations and works at the statement level by inserting, removing, or replacing a statement taken from other parts of the same program. RSRepair [126] fixes buggy programs with random search to guide the patch generation process. MutRepair [96] attempts to generate patches by applying mutation operators on suspicious if-condition statements. Smith *et al.* [143] showed that these approaches

tend to overfit test cases by generating incorrect patches that pass the test cases, mostly by deleting functionalities.

PAR [60] is an APR approach that is based on fixing templates that were manually extracted from 60,000 human-written patches. Later studies (e.g., Le *et al.* [67]) have shown that the six templates used by PAR could fix only a few bugs in Defects4J. Therefore, anti-patterns were integrated within existing search-based APR tools [153] (namely, GenProg [70] and SPR [91]) to help alleviate the problem of incorrect or incomplete fixes. However, a key limitation of those search-based approaches is their reliance on the quality of mutation operations and the fixing patterns.

In contrast to the search-based approaches, other approaches have aimed to *mine and learn fixing patterns* from prior bug fixes [112, 67, 86, 60]. The fixing patterns, also called fixing templates, could be automatically or semi-automatically mined [67, 112, 86, 87]. SemFix [112] instead uses symbolic execution and constraint solving to synthesize a patch by replacing only the right-hand side of assignments or branch predicates. Long and Rinard proposed Prophet [92], which learns code correctness models from a set of successful human patches. Prophet learns a patch ranking model using a machine-learning algorithm based on existing patches. Genesis [90] can automatically infer patch generation transformed from developers' submitted patches for automated program repair. HDRepair [67] was proposed to repair bugs by mining closed frequent bug fix patterns from graph-based representations of real bug fixes. ELIXIR [134] uses method call-related templates from PAR with local variables, fields, or constants, to construct more expressive repair expressions that go into synthesizing patches. CapGen [167], SimFix [52], FixMiner [64] is based on the frequently occurring code change operations (e.g., Insert If- Statement) that are extracted from the patches in code change histories. Avatar [86] exploits fixed patterns of static analysis violations as ingredients for patch

generation. Tbar [87] is a template-based APR tool with the collected fix patterns. Angelix [97] catches semantic information to repair methods. ARJA [186] generates lower-granularity patch representation that enables more efficient searching.

Deep Learning-Based APR approaches. Recently, deep learning (DL) has been applied to APR for directly generating patches. The first group of DL-based APR approaches to leverage the capability of DL models in *learning similar source code for similar fixes*. DeepRepair leverages learned code similarities, captured with recursive auto-encoders [169], to select the repair ingredients from code fragments that are similar to the buggy code. DeepFix [41] learns the syntax rules and is evaluated on syntax errors.

The second group of approaches treats APR as a *statistical machine translation* that translates the buggy code to the fixed code. Ratchet [46] and Tufano *et al.* [158] use sequence-to-sequence translation. They use neural network machine translation (NMT) with attention-based Encoder-Decoder, and different code abstractions to generate patches, while SequenceR [24] uses sequence-to-sequence NMT with a copy mechanism [138]. CODIT [21] learns code edits with encoding code structures in an NMT model to recommend fixes. A comparison with these NMT-based APR approaches is provided in the introduction. Recently, Tufano *et al.* [156] learn code changes using sequence-to-sequence NMT with simple code abstractions and keyword replacing. Despite treating the APR as a code transformation learning problem, their approach takes the entire method as the context for a bug. Thus, compared with our work in the automated program repair chapter, it has too much noise, leading to lower effectiveness than DLFix. In other words, the treatment of context from DLFix helps improve their model. Moreover, compared with our second idea about fixing multi hunks multi statements bugs from DEAR, the existing DL-based APRs fix individual statements at a time, thus, cannot work on the bugs with dependent fixes to multiple statements.

6.3 Model Agnostic Explanation

Interpretable AI. Many interpretable models just create simple proxy models for full neural networks. This kind of proxy model can often be generated by learning local faithful approximation. Lakkaraju *et al.* [66] represents sufficient conditions to get the local faithful approximation, while Ribeiro *et al.* [131] uses the linear model to do so. Some models try to find out the key features during the computation. Some researchers find key features through feature gradients, such as the research from Zeiler *et al.* [187]. And some others focus on the backpropagation of neurons' like the research work from Sundararajan *et al.* [150]. All of these approaches cannot handle the graph structure when dealing with the problem, while our explanation model, like IVDetect, can do.

Also, the post-hoc interpretability method is the other kind of common approach. This kind of approach regards the model as a black box and interprets them based on the influence of relevant information. Fisher *et al.* [37] analyzes the relevant information from variables and model class reliance to generate the interpretation. Adadi *et al.* [4] generates the interpretation for the model by using a survey. But all of these approaches have a similar problem as the previous type. None of them can deal with the tasks based on graphs.

Finally, there are some interpretation models that found the problems we mentioned and tried to use a new way to analyze the information within the model and generate the interpretation. Ying *et al.* [184] builds an explainer for the GNN-related models by generating the sub-graph from the whole graph to point out which part of the graph is important. We have also used this approach in interpretable vulnerability detection and bug detection, and we build better ways of learning code representations that make the model more suitable for specific tasks.

Vulnerability Detection Approaches. Various techniques have been developed to detect vulnerabilities. The rule-based approaches were developed to leverage known

vulnerability patterns to discover possible vulnerable code, such as FlawFinder [38], RATS [127], ITS4 [161], Checkmarx [23], Fortify [49] and Coverity [30]. Typically, the patterns are manually defined by human experts. The state-of-the-art vulnerability detection tools using static analysis provide the corresponding rules for each vulnerability type.

Another type is machine learning (ML) based or metric-based. Typically, these approaches require the human-crafted or summarized metrics as features to characterize vulnerabilities and train machine learning models on the defined features to predict whether a given code is vulnerable or not. For example, various ML-based approaches have been built on top of distinct metrics, such as terms and their occurrence frequencies [137], imports and function calls [109], complexity, code churn, and developer activity [140], dependency relation [108], API symbols and subtrees [182, 181]. Therefore, these approaches rely on human experts to define features manually and cannot pin down the precise locations of vulnerabilities because programs are represented in some coarse-grained granularities.

Recently, deep learning has been applied to detect vulnerabilities. For example, some DL approaches train a deep learning model on different code representations to detect vulnerabilities, such as the lexed representations of functions in a synthetic codebase [44], code snippets related to library/API function calls to detect two types of vulnerabilities [79], syntax-based, semantics-based, and vector representations [78], graph-based representations [198].

Nevertheless, none of the above approaches are designed to explain the detection results. Our IVDetect idea in the interpretable vulnerability detection chapter is different from all of the above approaches. The main goal of IVDetect is to add more intelligence assistance (IA) using a small graph of code statements, key variables, and CWE description to explain why a detection model reaches a prediction.

CHAPTER 7

CONCLUSION

As a summary of this dissertation, it highlights the importance of code representation learning for software quality control downstream tasks. The exploration of intelligent and explainable software quality control has shown significant potential, particularly in bug detection and vulnerability detection. The inclusion of explainability is crucial for enhancing the effectiveness of these tasks.

The research conducted in this dissertation focused on utilizing various learning code representation solutions to improve bug detection, automated program repair, and model agnostic explanation. Bug detection benefited from code context, code relationship, and code coverage matrix representations, while automated program repair employed a tree-based structure to analyze the surrounding context of buggy code and facilitate code transformation for fixing. Additionally, the program dependency graph was used for model agnostic explanation, generating explanations for graph-based deep learning models.

The results obtained through these efforts demonstrated that employing diverse approaches to learning code representation can enhance existing work in software quality control. It signifies that selecting an appropriate method for learning code representation contributes to the development of effective and explainable bug detection and fixing, which is the principal objective of this dissertation.

In the future, exploring innovative methods in code representation learning that can effectively meet the diverse needs of various research tasks and harness the advancements in cutting-edge techniques would be intriguing. An interesting avenue for future work lies in integrating robust deep learning models like ChatGPT into the realm of software quality control, offering significant potential. Furthermore,

an exciting research direction would involve the development of more sophisticated and nuanced explainable models that aid developers in comprehending the underlying rationale behind software quality control issues.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 39–46, 2006.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, number 12, pages 39–46, 2006.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION)*, pages 89–98, 2007.

[4] Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018.

[5] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (PMLR)*, pages 2091–2100, 2016.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[8] Matthew Amodio, Swarat Chaudhuri, and Thomas W Reps. Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231*, 2017.

[9] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM Special Interest Group on Programming Languages-Special Interest Group on Software Engineering (SIGPLAN-SIGSOFT) Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 1–8, 2007.

[10] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–188, 2016.

[11] Daniel W Barowy, Emery D Berger, and Benjamin Zorn. Excelint: Automatically finding spreadsheet formula errors. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, 2018.

[12] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*, 2016.

[13] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. Nar-miner: Discovering negative association rules from code for bug detection. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 411–422, 2018.

[14] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code. In *Proceedings of the 33rd International Conference on Machine Learning (PMLR)*, volume 48, pages 2933–2942, 2016.

[15] Lionel C Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE)*, pages 137–146, 2007.

[16] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*, 2017.

[17] Timothy Alan Budd. *Mutation analysis of program test data*. Yale University, 1980.

[18] BufferOverFlow. Cwe-120: Buffer overflow. https://cwe.mitre.org/data/definitions/120.html. Accessed 2022.

[19] BugsInPy. The BugsInPy data set. https://github.com/soarsmu/BugsInPy. Accessed 2020.

[20] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Treecaps: Tree-based capsule networks for source code processing. In *Proceedings of the Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence (AAAI)*, volume 35, pages 30–38, 2021.

[21] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural machine translation. *arXiv preprint arXiv:1810.00314*, 2018.

[22] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *arXiv preprint arXiv:2009.07235*, 2020.

[23] Checkmarx. Checkmarx. https://www.checkmarx.com/. Accessed 2022.

[24] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering (TSE)*, 47(9):1943–1959, 2019.

[25] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[26] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[27] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[28] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[29] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Proceedings of the 21st ACM Special Interest Group on Programming Languages (SIGPLAN) International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 673–674, 2006.

[30] Coverity. Coverity scan. https://scan.coverity.com/. Accessed 2022.

[31] Yann Le Cun, Conrad C. Galland, and Geoffrey E. Hinton. Gemini: Gradient estimation through matrix inversion after noise injection. *Advances in Neural Information Processing Systems (NIPS)*, 1, 1988.

[32] Defects4j. Defects4j. https://github.com/rjust/defects4j. Accessed 2023.

[33] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[34] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM Special Interest Group on Operating Systems (SIGOPS) Operating Systems Review (OSR)*, 35(5):57–72, 2001.

[35] Jiahao Fan, Yi Li, Shaohua Wang, and Tien Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 508–512, 2020.

[36] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[37] Aaron Fisher, Cynthia Rudin, and Francesca Dominici. All models are wrong but many are useful: Variable importance for black-box, proprietary, or misspecified prediction models, using model class reliance. *arXiv preprint arXiv:1801.01489*, 2018.

[38] FlawFinder. Flawfinder. http://www.dwheeler.com/FlawFinder. Accessed 2021.

[39] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *roceedings of the 22nd ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 855–864, 2016.

[40] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proceedings of the 19th ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–130, 2010.

[41] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence (AAAI)*, volume 31, 2017.

[42] GZoltar. Gzoltar. http://www.gzoltar.com/. Accessed 2021.

[43] Hadamard. Hadamard product. https://en.wikipedia.org/wiki/Hadamard_product_(matrices). Accessed July 11, 2019.

[44] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, et al. Learning to repair software vulnerabilities with generative adversarial networks. In *Advances in Neural Information Processing Systems (NIPS)*, volume 31, 2018.

[45] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[46] Hideaki Hata, Emad Shihab, and Graham Neubig. Learning to generate corrective patches using neural machine translation. *arXiv preprint arXiv:1812.07170*, 2018.

[47] Jordan Henkel, Shuvendu Lahiri, Ben Liblit, and Thomas W. Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 163–174, 2018.

[48] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM Special Interest Group on Programming Languages-Special Interest Group on Software Engineering (SIGPLAN-SIGSOFT) Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 9–14, 2007.

[49] HPFortify. Hp fortify. https://www.hpfod.com/. Accessed 2022.

[50] Vinoj Jayasundara, Nghi Duy Quoc Bui, Lingxiao Jiang, and David Lo. Treecaps: Tree-structured capsule networks for program source code processing. *arXiv preprint arXiv:1910.12306*, 2019.

[51] JDT. Jdt package. https://www.eclipse.org/jdt/core/tools/jdtcoretools/index.php. Accessed 2021.

[52] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Software Testing and Analysis (ISSTA)*, pages 298–309, 2018.

[53] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Software Testing and Analysis (ISSTA)*, page 298–309, New York, NY, USA, 2018.

[54] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *Proceedings of the 43th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1161–1173, 2021.

[55] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, 47(6):77–88, 2012.

[56] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 273–282. ACM, 2005.

[57] James A Jones, Mary Jean Harrold, and John T Stasko. Visualization for fault localization. In *Proceedings of ICSE Workshop on Software Visualization*, 2001.

[58] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125, 2017.

[59] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM Special Interest Group on Algorithms and Computation Theory- Special Interest Group on Programming Languages (SIGACT-SIGPLAN) Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[60] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 802–811, 2013.

[61] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[62] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[63] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[64] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering (ESE)*, (25):1980–2024, 2020.

[65] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, volume 25, 2012.

[66] Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Jure Leskovec. Interpretable & explorable approximations of black box models. *arXiv preprint arXiv:1707.01154*, 2017.

[67] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224, 2016.

[68] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 3–13, 2012.

[69] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, 2015.

[70] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38(1):54–72, 2011.

[71] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, number 9, pages 249–260, 2017.

[72] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Software Testing and Analysis (ISSTA)*, pages 169–180, 2019.

[73] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[74] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the 42th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 602–614, 2020.

[75] Yi Li, Shaohua Wang, and Tien N. Nguyen. Fault localization with code coverage representation learning. In *Proceedings of the 43th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 661–673, 2021.

[76] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 511–523, 2022.

[77] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[78] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756*, 2018.

[79] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[80] Zhenmin Li and Yuanyuan Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. *Special Interest Group on Software Engineering (SIGSOFT) Software Engineering Notes (SEN)*, 30(5):306–315, 2005.

[81] Ziyao Li, Liang Zhang, and Guojie Song. Gcn-lase: Towards adequately incorporating link attributes in graph convolutional networks. *arXiv preprint arXiv:1902.09817*, 2019.

[82] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. Antminer: Mining more bugs by reducing noise interference. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 333–344, 2016.

[83] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. In *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 40, pages 15–26, 2005.

[84] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM Special Interest Group on Programming Languages (SIGPLAN) Conference on Programming Language Design and Implementation (PLDI)*, pages 15—-26, New York, NY, USA, 2005.

[85] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 102–113, 2019.

[86] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12, 2019.

[87] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Software Testing and Analysis (ISSTA)*, pages 31–42, 2019.

[88] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662, 2018.

[89] Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. *ACM Special Interest Group on Software Engineering (SIGSOFT) Software Engineering Notes (SEN)*, 30(5):296–305, 2005.

[90] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 727–739, 2017.

[91] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 166–178, 2015.

[92] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 51, pages 298–312, 2016.

[93] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26(2):172–219, 2014.

[94] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Software Testing and Analysis (ISSTA)*, pages 101—-114, New York, NY, USA, 2020.

[95] ManyBugs. The manybugs data set. https://repairbenchmarks.cs.umass.edu/. Accessed 2019.

[96] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 441–444, 2016.

[97] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.

[98] Microsoft. Neural network intelligence. https://github.com/microsoft/nni. Accessed August 28th, 2020.

[99] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS)*, page 3111–3119, 2013.

[100] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3111–3119, 2013.

[101] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3994–4003, 2016.

[102] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSE)*, pages 120–130, 2000.

[103] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *IEEE 7th International Conference on Software Testing, Verification and Validation (ICST)*, pages 153–162, 2014.

[104] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. Tbcnn: A tree-based convolutional neural network for programming language processing. *arXiv preprint arXiv:1409.5718*, 2014.

[105] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A large-scale study of call graph-based impact prediction using mutation testing. *Software Quality Journal*, 25(3):921–950, 2017.

[106] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):11, 2011.

[107] Jaechang Nam and Sunghun Kim. Heterogeneous defect prediction. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 508–519, New York, NY, USA, 2015.

[108] Stephan Neuhaus and Thomas Zimmermann. The beauty and the beast: Vulnerabilities in red hat's packages. In *Proceedings of the USENIX Annual Technical Conference*, pages 383–396, 2009.

[109] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM*

Conference on Computer and Communications Security (CCS), pages 529–540, 2007.

[110] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proceedings of the 41th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 819—830, 2019.

[111] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, pages 440–455, 2009.

[112] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.

[113] Son Nguyen, Hung Dang Phan, Trinh Le, and Tien N. Nguyen. Suggesting natural method names to check name consistencies. In *Proceedings of the 42th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1372–1384, 2020.

[114] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 315–324, 2010.

[115] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM Special Interest Group on Software Engineering (SIGSOFT) Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 383–392, 2009.

[116] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM Special Interest Group on Software Engineering (SIGSOFT) Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 383—-392, 2009.

[117] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, 50(6):369–378, 2015.

[118] Mike Papadakis and Yves Le Traon. Using mutants to locate" unknown" faults. In *IEEE 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 691–700, 2012.

[119] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability (STVR)*, 25(5-7):605–628, 2015.

[120] Jibesh Patra and Michael Pradel. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[121] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 609–620, 2017.

[122] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[123] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the 25st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 447–456, 2010.

[124] PIT. Pit. https://pitest.org/. Accessed 2021.

[125] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[126] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 254–265, 2014.

[127] RATS. Rats: Rough audit tool for security. https://code.google.com/archive/p/rough-auditing-tool-for-security/. Accessed 2021.

[128] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the" naturalness" of buggy code. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 428–439, 2016.

[129] Baishakhi Ray and Miryung Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM Special Interest Group on Software*

Engineering (SIGSOFT) 20th International Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 53:1–53:11, 2012.

[130] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 155–165, 2014.

[131] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1135–1144, 2016.

[132] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, 2018.

[133] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 10–13, 2018.

[134] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *Proceedings of the 32st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659, 2017.

[135] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 13–24, 2019.

[136] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. Harnessing evolution for multi-hunk program repair. pages 13–24. Proceedings of the 41th ACM/IEEE International Conference on Software Engineering (ICSE), 2019.

[137] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering (TSE)*, 40(10):993–1006, 2014.

[138] Abigail See, Peter J Liu, and Christopher D Manning. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*, 2017.

[139] Min Shi, Yufei Tang, Xingquan Zhu, and Jianxun Liu. Feature-attention graph convolutional networks for noise resilient learning. *arXiv preprint arXiv:1912.11755*, 2019.

[140] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering (TSE)*, 37(6):772–787, 2010.

[141] Rishabh Singh, Benjamin Livshits, and Benjamin Zorn. Melford: Using neural networks to find spreadsheet errors. *Microsoft Tech Report Number MSR-TR-2017-5*.

[142] Cross site Scripting. Cwe-79: Cross-site scripting. http://cwe.mitre.org/data/definitions/79.html. Accessed 2022.

[143] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 532–543, 2015.

[144] Randy Smith and Susan Horwitz. Detecting and measuring similarity in code clones. In *Proceedings of the International workshop on Software Clones (IWSC)*, volume 24, 2009.

[145] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pages 129–136, 2011.

[146] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Software Testing and Analysis (ISSTA)*, pages 273–283, 2017.

[147] Soot. Soot introduction. https://sable.github.io/soot/. Accessed July 11, 2019.

[148] Auth Bypass Spoofing. Cwe-290: Authentication bypass by spoofing. https://cwe.mitre.org/data/definitions/290.html. Accessed 2022.

[149] SQLInj. Cwe-89: Sql injection. https://cwe.mitre.org/data/definitions/89.html. Accessed 2022.

[150] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *Proceedings of the 34th International Conference on Machine Learning (ICML)*, page 3319–3328, 2017.

[151] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[152] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[153] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 24th ACM Special Interest Group on Software Engineering (SIGSOFT) International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 727–738, 2016.

[154] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1422–1432, 2015.

[155] John Toman and Dan Grossman. Taming the static analysis beast. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL)*, pages 18:1–18:14, 2017.

[156] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 25–36, 2019.

[157] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 542–553, 2018.

[158] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[159] Unknown. Bilinear interpolation. https://en.wikipedia.org/wiki/Bilinear_interpolation. Accessed 2022.

[160] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NIPS)*, 30, 2017.

[161] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC)*, pages 257–267, 2000.

[162] WALA. Wala documentation. http://wala.sourceforge.net/wiki/index.php/Main_Page. Accessed July 11, 2019.

[163] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 397–407, 2018.

[164] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 708–719, 2016.

[165] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 297–308, 2016.

[166] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM Special Interest Group on Software Engineering (SIGSOFT) Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 35–44, 2007.

[167] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1–11, 2018.

[168] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490, 2019.

[169] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016.

[170] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016.

[171] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1556–1560, 2020.

[172] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2011.

[173] W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. Software fault localization using dstar (d*). In *IEEE 6th International Conference on Software Security and Reliability (SSIRI-C)*, pages 21–30, 2012.

[174] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering (TSE)*, 42(8):707–740, 2016.

[175] W Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 19(04):573–597, 2009.

[176] W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 449–456, 2007.

[177] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 204–214, 2014.

[178] Qi Xin and Steven P Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670, 2017.

[179] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 416–426, 2017.

[180] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 191–200, 2014.

[181] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Workshop on Offensive Technologies (WOOT)*, pages 13–13, 2011.

[182] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 359–368, 2012.

[183] Wenpeng Yin, Hinrich Schütze, Bing Xiang, and Bowen Zhou. ABCNN: attention-based convolutional neural network for modeling sentence pairs. *Transactions of the Association for Computational Linguistics (TACL)*, 4:259–272, 2016.

[184] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, volume 32, 2019.

[185] Edward Yourdon. Structured programming and structured design as art forms. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, pages 277–277, 1975.

[186] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering (TSE)*, 46(10):1040–1067, 2018.

[187] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings (ECCV)*, pages 818–833, 2014.

[188] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 783–794, 2019.

[189] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 23–32, 2011.

[190] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[191] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 48, pages 765–784, 2013.

[192] Zhenyu Zhang, Wing Kwong Chan, TH Tse, Bo Jiang, and Xinming Wang. Capturing propagation of infected program states. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM Special Interest Group on Software Engineering (SIGSOFT) Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 43–52, 2009.

[193] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. Cnn-fl: An effective approach for localizing faults using convolutional neural networks. In *Proceedings of*

the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 445–455, 2019.

[194] Zhuo Zhang, Yan Lei, Qingping Tan, Xiaoguang Mao, Ping Zeng, and Xi Chang. Deep learning-based fault localization with contextual information. *IEICE Transactions on Information and Systems*, 100(12):3027–3031, 2017.

[195] Gang Zhao and Jeff Huang. Deepsim: Deep learning code functional similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 141–151, New York, NY, USA, 2018.

[196] Lei Zhao, Lina Wang, Zuoting Xiong, and Dongming Gao. Execution-aware fault localization based on the control flow analysis. In *International Conference on Information Computing and Applications (ICICA)*, pages 158–165, 2010.

[197] Wei Zheng, Desheng Hu, and Jing Wang. Fault localization analysis based on deep neural network. *Mathematical Problems in Engineering*, 2016.

[198] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 10197–10207, 2019.

[199] J. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *IEEE International Conference on Computer Vision (ICCV)*, number 10, pages 2242–2251, 2017.