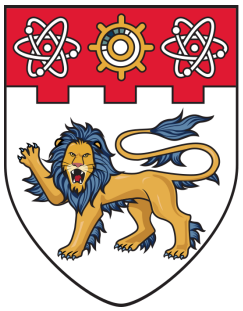


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

LEVERAGING DEEP LEARNING TECHNIQUES TO SECURE SOFTWARE DEVELOPMENT

SIOW JING KAI

School of Computer Science and Engineering

2022

LEVERAGING DEEP LEARNING TECHNIQUES TO SECURE SOFTWARE DEVELOPMENT

SIOW JING KAI

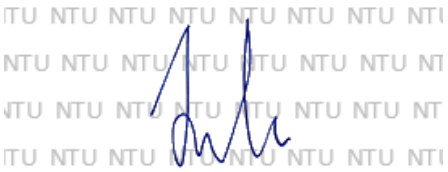
School of Computer Science and Engineering

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

2022

Statement of Originality

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



10/1/2022

. . . . . .
Date

. . . . . .
Jing Kai Siow

Supervisor Declaration Statement

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

10 Jan 2022

. . . . . .

Date

. . . . . .

Yang Liu

Authorship Attribution Statement

This thesis contains material from four paper(s) published in the following peer-reviewed journal(s) / from papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as **Jing Kai Siow**, Cuiyun Gao, Linging Fan, Sen Chen, and Yang Liu, "CORE: Automating Review Recommendation for Code Changes," 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 284-295, doi: 10.1109/SANER48275.2020.9054794.

The contributions of the co-authors are as follows:

- I co-designed and implemented the approach (e.g., neural networks, data curations), conducted the experiments, wrote the manuscripts, and conducted data analysis.
- Dr. Gao co-designed the approaches and wrote the manuscripts. She further gave valuable feedbacks
- Dr. Chen and Dr. Fan gave their valuable feedback, revised the manuscripts, and helped analyze the experiment results.
- Prof Liu Yang supported the research and revised the manuscript.

Chapter 4 is published as Yaqin Zhou, **Jing Kai Siow**, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. SPI: Automated Identification of Security Patches via Commits. ACM Trans. Softw. Eng. Methodol. 31, 1, Article 13 (January 2022), 27 pages. DOI:https://doi.org/10.1145/3468854

The contributions of the co-authors are as follows:

- I co-designed and co-implemented the approach (e.g., neural networks, data curation scripts), wrote the manuscript, conducted the experiments, and handle the publication communication with the journal committee.

- Dr. Zhou co-designed and co-implemented the approach and wrote the manuscripts.
- Dr. Wang prepared the keyword filtering process and data analysis.
- Mr. Liu Shangqing revised the manuscripts and gave valuable feedback on the approach and experiments.
- Prof Liu Yang co-designed the approach and supported the research.

Chapter 5 is *submitted* as **Jing Kai Siow**, Shangqing Liu, Kui Liu, Xiaofei Xie, and Yang Liu, Ratchet: Retrieval Augmented Transformer for Program Repair.

The contributions of the co-authors are as follows:

- I co-designed and implemented the approach (e.g., neural networks, data curations), conducted the experiments, wrote the manuscripts, and analyze the experiment results.
- Mr. Liu Shangqing co-designed the approach and help in conducting the experiments. He also wrote the manuscripts and participate actively in the discussion.
- Dr. Liu Kui provides his valuable feedbacks and wrote the manuscripts with the above authors.
- Dr. Xie provides his feedbacks and revised the manuscripts.
- Prof Liu Yang supported the research and co-designed the approach.

Chapter 6 is accepted as **Jing Kai Siow**, Shangqing Liu, Xiaofei Xie, and Guozhu Meng, Yang Liu, "Learning Program Semantics with Code Representations: An Empirical Study", 2022 IEEE 29th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022.

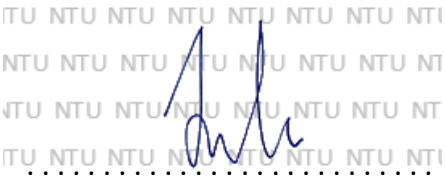
The contributions of the co-authors are as follows:

- I co-designed and implemented the approach (e.g., research questions, study design, source code implementation), conducted the experiments, wrote the manuscripts, and analyze the experiment results.

- Mr. Liu Shangqing co-designed the design and experiments. Furthermore, he helped in conducting the experiments. He also wrote the manuscripts and participate actively in the discussion.
- Dr. Xie and Dr. Meng provides their feedbacks and revised the manuscripts.
- Prof Liu Yang co-designed the approach of the study and further supported the research.

30/11/2021

. . . . . .
Date



. . . . . .
Jing Kai Siow

Acknowledgments

Firstly, I would like to express my sincere gratitude and appreciation to my supervisor, Prof. Liu Yang. Prof. Liu has given me the utmost support and assistance throughout my study at NTU. His guidance has helped me with countless problems in my research or my personal life. I had truly learned the most from Prof. Liu and he will always be a model role to me. In addition, I would like to thank the members of the Thesis Advisory Committee, Prof. Yan Xu and Prof. Yi Li, for their valuable time and advice in guiding me during my candidature.

I would like to thank my colleagues at NTU Cybersecurity Lab, Mr. Shangqing Liu, Dr. Yaqin Zhou, Prof. Guozhu Meng, Dr. Sen Chen, Dr. LingLing Fan, Dr. CuiYun Gao and Dr. Xiaofei Xie. They imparted me with valuable knowledge that enables me to explore deeper into my research. I am particularly grateful for the precious effort and guidance from Prof. Guozhu Meng and Dr. Yaqin Zhou. It is only through their guidance that I can grasp the direction of my research area. I also deeply appreciate the help and support of Mr. Shangqing Liu. His feedback and suggestions have been essential in raising the quality of our work.

Last, but not least, I would like to thank the support of my loved ones. I would also like to express my deepest appreciation to my friends and family for encouraging and inspiring me throughout these past four years. Their care and encouragement have been my mental support throughout my pursuit of my Ph.D. Finally, I would like to extend my special thanks to Jiang Yichu for her support and encouragement throughout these years.

Contents

List of Figures

List of Tables

Abstract

Adversarial threats have grown rapidly in recent years, resulting in the growing importance of software security. Many processes in the software development life cycle seek to reduce attack surfaces in the codebase, e.g., black box testing, code reviews, and static code analysis. The key idea is to reduce implementation bugs, such as out-of-bounds bugs and memory leaks, during the development phase. However, they often require an intensive amount of resources, further reducing the productivity of developers. Therefore, automation in software development processes is highly sought.

The vast amount of code-related data contributes massively to the domain of software security. With open-source information widely available, e.g., vulnerability databases, open-sourced codebase, and security patches, many data-driven approaches are employed to enhance the security of our cyberspace. This thesis presents my approach to enhancing software security throughout multiple development stages with code intelligent tasks. The main objective of this thesis is to increase the security in the codebase during software development by leveraging data-driven and deep learning techniques.

Vulnerability commonly occurs in the software development phase and might persist after the deployment. We propose a data-driven approach to increase the quality of the source code, reducing the number of security bugs and errors that might occur during the development phase. Specifically, we propose a deep learning approach, CORE, in automating the code review process. CORE employs a multi-level embedding layer in representing and learning the relevancy between the source code and their respective submitted reviews. During its inference phase, it suggests the most relevant reviews for given code submission. Our experiments further show that CORE achieves up to 0.234% in MRR and 0.482% in Recall@10 at suggesting reviews.

Patch management is a common process in software security. It ensures that all software is up-to-date and does not contain any exposures to vulnerabilities. However, the amount of officially published security patches is far from complete. Hence, we propose our work in Patch Curation and Security Patch Identification, SPI, which aims to collect unofficial security patches that lure silently in an open-source project. Due to the vast amount of necessary data in data-driven approaches, the dataset on security patches is still lacking. To enable an effective patching strategy, we propose our approach in finding security patches amidst open-source projects through a sophisticated deep-learning mining pipeline. We propose a three steps process in identifying security patches: Keyword Filtering, Manual Verification, and Deep Learning Patch Identification. Our experiments demonstrate the high performance of SPI, achieving up to 87.93% F1-score in identifying security patches. We further evaluate SPI in a production environment, showing that SPI can benefit both researchers and developers in future research and patch management.

Even though security measures are always in place, vulnerability still sneaks past them and appears in the published software. During the maintenance phase of the software development, developers resolve vulnerability and bugs, ensuring that the codebase is secured and correct. However, the time to resolve the vulnerability is crucial as this vulnerable period exposes the software to cyber-attacks and adversarial threats. Hence, to reduce the duration of this period, we present our approach in automated program repair, Ratchet. We employ a learning-based approach in repairing programs to ensure that real patches can be generated effectively and without manual effort. Specifically, we present our deep learning-based transformer model in learning and generating patches among open-source projects. Furthermore, we augmented our generation process with the retrieval information to enhance the patch generation process. Ratchet outperforms deep learning approaches on fault localization with 39.8-96.4% in accuracy and patch generation with 18.4-46.4% in repair accuracy.

Despite great performance in employing deep learning techniques in software engineering tasks, various code representations can be employed. Different representations inherently convey different meanings and semantics of the source code. To facilitate future works in software security and engineering code intelligent tasks, we conclude the thesis with an empirical study

of code representations across three code intelligent tasks: Code Classification, Vulnerability Detection, and Clone Detection. Our study shows that graph representations are superior to other forms of code representation, showing huge potential in representing source code with program graph. This work serves as a foundation for potential research directions, enabling us to investigate deeper into a better code representation.

Chapter 1

Introduction

1.1 Background

With the booming adoption of computers and software during the information era, the industry has become increasingly focused on securing its essential information and computer system. A well-crafted cyberattack can cause significant monetary losses and information leakages. For instance, a supply chain cyberattack, Sunburst, causes sensitive government and business information to be leaked [1]. The malicious actor gained access to one of Solarwind software, Orion, and installed malware on it. The malware then sends customer information to their remote server, effectively leaking information continuously to the malicious actor. Despite the attack being discovered in December 2020, it was reported that the malware existed in the system as early as February 2020 [2], months before the threat was discovered. Another classic example of a cyber attack is Code Red Worm and its variant [3] that was discovered in 2001. It specifically infects unpatched Microsoft Web servers and was reported to infect up to 2,000 hosts each minute. These attacks highlight the importance of cyber defenses, such as early vulnerability detection and patch management, as an essential component of an organization.

To this end, researchers and security experts seek ways to strengthen their system security by finding and patching vulnerabilities as early as possible. With the early discovery of vulnerabilities, countermeasures can be deployed at the earliest opportunity so that the damages of any attacks would be lessened. Despite all these efforts in cybersecurity, there is still an average of 50 Common Vulnerabilities and Exposures (CVE) reported daily in 2020. Furthermore,

over 57% of these reported CVEs are classified as high severity [4]. The increasing number of vulnerabilities demands an urgent need for better cybersecurity techniques and solutions.

Software development comprises several key processes, such as Requirement Engineering, System Design, Implementation, Deployment and Maintenance, etc. These processes are commonly conducted iteratively to ensure that the designed system is correct and secured. Among them, software developers are deeply involved in the implementation and maintenance of the codebase. Additional sub-processes, such as reviewing the submitted code of developers and vulnerability scanning, are often employed to raise the quality and security of the codebase. In spite of their effectiveness, they often took up precious manpower and time. For instance, software developers could spend up to 50% of their development time on discovering and fixing bugs [5]. Hence, it is intuitive for automation in security and software development-related processes to be sought after, as manual operations could delay in detecting and repairing vulnerable software, further increasing the risk of cyberattacks.

Researchers often seek deep learning and big code in assisting many software development processes. For instance, Allamanis et al. [6] proposed a framework, *NATURALIZE*, to suggest code convention and identifier names during the development phase, assisting developers in better coding practices. Another example would be a work by Liu et al [7] where they propose a solution through Abstract Syntax Tree and generative networks in automatically generating commit messages for code changes. These two works show the increasing trends of automating software development processes through big-code and deep-learning techniques. Likewise, software security can be improved through similar techniques and enhancement to current security-related processes.

In essence, software security can be enhanced throughout the software development stages, for instance, improving the quality of the codebase, preventing bugs and vulnerabilities from appearing in production source code, immediate remediation after discovering vulnerabilities, and ensuring software and codebase is updated to prevent exposures. There are processes in the development phase, such as code review, vulnerability remediation, and patch management, to increase the quality of the source code and decrease the attack surfaces for software. Despite extensive measures to maintain the quality of the codebase, there is still a possibility of implementing software function that contains security bugs. Detection and remediation of these

flaws can enable early patching of your software to prevent any exploitation and vulnerabilities. In fact, up to 50% of security flaws are not resolved for at least 216 days, exposing much software to potential cyber threats [8].

1.2 Motivation and Challenges

As highlighted previously, the need for cyber security has been increasing due to the uprising trend of cyber threats in recent years. Meanwhile, security measures are always in-demands due to the huge amount of effort and time that are required to safeguard their software. Although there is some form of automation in these domains, their performance and efficiency are still in question. Furthermore, the imminent age of big code impacts the development process in many ways, e.g., increasing the performance of a code-related task, or reducing the amount of manual labor in multiple software development processes.

Many software development processes can be secured or improved through data-driven approaches. Many methodologies employ an iterative approach in continuously improving and maintaining their software. For instance, the methodology, Continuous Integration/Continuous Deployment (CI/CD), iteratively develops and deploys software throughout the lifetime of the software. This displays the need for automated tools in facilitating efficiency and security throughout the development life cycle.

With the increasing trend of employing big code and deep neural networks in enabling great performance in code intelligent tasks, tedious software security and developments tasks can be also reduced, in terms of resources, through similar approaches. What follows is how to effectively employ these techniques to enhance security and aid developers in creating safer software. Furthermore, incorporating these deep-learning techniques into the software engineering or security domain is challenging and often requires domain knowledge.

Motivated by the constant demand for innovation in big code and software security, we propose deep-learning and data-driven approaches that aim to facilitate software security in software development processes with a data analytic perspective, instead of the traditional heuristic way. With the availability of big code, deep-learning approaches can be built up by code intelligence and aid developers in software development.

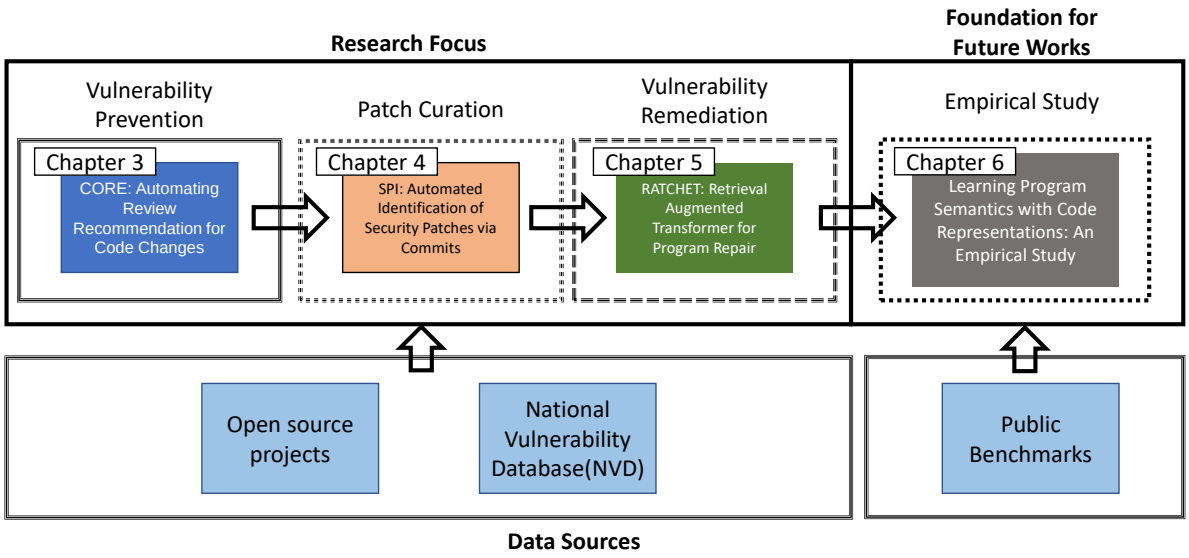


Figure 1.1: Roadmap of Thesis.

1.3 Main Work

Figure 1.1 shows the research focus of each chapter and overview of the thesis. CORE (Chapter 3) focuses on vulnerability prevention and enhancing the quality of the codebase, while our work on SPI (Chapter 4) focuses on curating high-quality patches. Furthermore, Ratchet (Chapter 5) focuses on remediating vulnerability at the earliest possible time, and Chapter 6 describes our empirical study to create a strong foundation for future works.

Despite the efforts in ensuring software security, their success is constantly undermined by implementation bugs and design flaws. Vulnerabilities are still introduced into the codebases and security measures, such as vulnerability remediation and patch management, are required to ensure that threats are neutralized properly and promptly. As mentioned previously, there are many key processes in the software development lifecycle (e.g., requirement engineering, system design, etc). Software developers are involved deeply in the iterative cycle of development and maintenance of the codebase, which is the focus of this thesis. Figure 1.2 shows how my works correlate to the cycle of development and maintenance of source code. In the software development phase, we employ a deep-learning technique in automating and strengthening the security of the codebase through our automated code review engine, CORE. On the other hand, bugs and vulnerabilities are often discovered and subsequently remediated in the maintenance

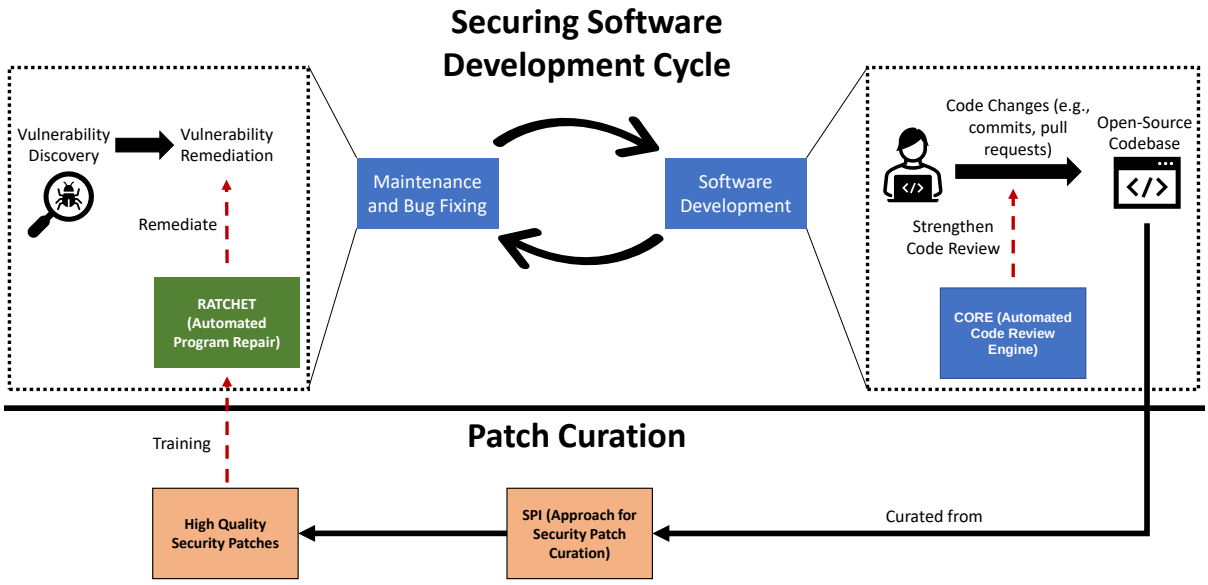


Figure 1.2: My works and their related SDLC stages.

phase. We propose our approach on Automated Program Repair (APR), Ratchet, to automatically fix and patch buggy programs. To gather high-quality patches for Ratchet, we further propose a patch curation approach, SPI, in curating security patches from open-source projects. The collected security patches and their curation approaches have high value in both research and commercial areas. Lastly, to construct a stronger foundation for future works, we conduct an empirical study on different code representations on various code intelligent tasks as part of this thesis.

1.3.1 Securing Software Development Phase

Vulnerabilities are often unintentionally introduced into the codebase during the development stage. Developers might overlook implementation design or neglect important sanity checks, allowing exploitation to occur after the software is pushed to production. To expose these security and implementation bugs before they are introduced into the production stage, we propose our solution to automated code reviewing, *CORE: Automated Code Review using Multi-Level Embedding and Relevancy Learning*. CORE targets the development phase in raising the quality of the codebase and ensuring that common implementation and security bugs can be exposed early with our automated code review engine.

Code review is a tedious and time-consuming process. Software developers often took much of their time in reviewing submitted code, reducing their productivity. CORE serves as an automated recommendation engine that enhances the quality of the codebase and further increases the efficiency of developers by automatically providing quality reviews. By suggesting a review for newly submitted code, developers can spend less time on the code review process and allocate more resources to other important parts of the development phase. Furthermore, our solution can learn to identify implementation bugs and notify the developers if similar implementation bugs occur in new submissions. By reducing the number of bugs and improving the quality of the source code, software security can be enhanced [9], directly increasing the productivity of the developers and improving the overall quality of the codebase.

1.3.2 High-Quality Security Patch Curation

Patch management is a crucial security process where software is kept updated and ensures that the exposure to vulnerability can be reduced. Online security database (e.g., National Vulnerability Database [10] and CVEDetails [11]) often publishes the security patches after developers resolve the vulnerability, (i.e., after the vulnerability remediation stage). However, developers often did not report the security patches due to neglect or simply do not wish to report them. This hinders the patch management process, where security experts could not locate all possible security patches in open-source software. To address this problem, we propose our solution on patch curation and security patch identification: *SPI: Security Patch Identification via Commits*. Our key objective in this task is to learn to identify security patches in open-source software. However, there are many challenges in getting patches that are in suitable condition for a data-driven approach. Firstly, gathering sufficient data for deep learning is challenging. Despite there being many online resources, the quantity of patches is far from enough. Secondly, the quality of the patches could also affect the learning of the model. For instance, if there is a large number of mislabeled samples in our training dataset, the model cannot learn effectively. Furthermore, manually collecting these patches can be resource-intensive.

Hence, we propose a large-scale security patch collection and identification from open-source projects. We employ neural networks to learn and identify security patches from their commits. Our work collected 40,523 security patches and non-security commits through a three

steps pipeline: Keyword Filtering, Manual Verification, and Model Learning. We further employ Long-Short Term Memory in learning the prominent features, e.g., commit messages and submitted code, that can help us distinguish security patches.

One of the major applications of SPI is to curate high-quality security patches that serve as a training resource for future relevant deep learning applications. The quality of the data directly impacts the performance of the underlying deep learning application. Therefore, our curated dataset assists us in curating high-quality security patches for our subsequent work in APR, Ratchet.

1.3.3 Facilitating Vulnerability Remediation

In spite of constant efforts for security measures, vulnerability still appears in published software due to human errors and other factors. With the assumption that vulnerabilities always exist, it is in the best interest of developers to neutralize them as soon as possible. Vulnerability remediation is a common workflow to conduct during the maintenance phase where vulnerabilities are first discovered in the codebase. Delay in patching vulnerabilities could exposure the software to threats. With early remediation to the discovered bugs, the system can be secured earlier, reducing the chances of cyberattacks.

To tackle this urgent need of vulnerability remediation, we propose our automated program repair solution: *Ratchet, an automated fault localization, and program repair through a retrieval-augmented transformer model*. With an automated approach, vulnerability can be resolved swiftly and without manual intervention, ensuring that the software is secure at the earliest possible time. Furthermore, we collect high-quality patches through our proposed curation process (i.e., SPI) and facilitate better learning of Ratchet. Ratchet consists of two key processes: localizing the fault/bugs and generating the corresponding Patches. Our fault localization model first locates the buggy statement among the buggy function. After which, our patch generation model augmented the source input of the transformer model with the closest patch as additional context for the learning.

1.3.4 Foundation for Future Work in Securing Software Development Phases

Although our proposed security-related code intelligent tasks achieve great performance through sequential code representation (i.e., representing source code using a sequence of tokens), sophisticated code representations have been proposed by researchers in recent years [12–16]. Code representations might perform differently across different code intelligent tasks, hence, there is a need for a study to investigate the impact of code representation on these tasks. Furthermore, finding an optimal and suitable code representation for different code intelligent tasks is also an important work that can inspire further research areas.

With the main objective of constructing a strong foundation for my future works in code intelligent tasks, we conduct an empirical study on different code representations through various software engineering tasks to investigate the advantages and disadvantages of different code representations. Our study spans four main categories of code representation: feature-based, sequence-based, tree-based, and graph-based code representation. We conduct our empirical study on three popular classification-based code intelligent tasks: code classification, vulnerability detection, and clone detection.

1.4 Major Contributions

The major contributions of this thesis are as follows:

1. We proposed an automated code review engine, CORE, that employs an information retrieval and deep learning approach in suggesting past reviews for newly submitted code. Our main objective is to raise the quality of the codebase and reduce the number of bugs before the software is ready for release. To this end, we collected code change and review data from GitHub open-source projects, ensuring that our data are as close to the real-life application as possible. CORE suggests relevant comments for code submissions based on training on past historical reviews. Our evaluation shows that CORE can suggest high-quality actions to increase the quality of the project.

2. We proposed a security patch identification pipeline, *SPI* that curates implicit and explicit security patches in open-source projects. Our solution aims to aid and facilitate better

patch management in software security. Similarly, we employ a data-driven and deep learning approach in identifying patches. We crafted a three-step data collection pipeline and curated a high-quality dataset of security patches, consisting of 40,523 samples. Our experimental result shows that SPI is effective in identifying security patches, and it is feasible and practical for a production setting.

3. We further propose an APR technique, Ratchet, in repairing buggy programs with generative neural networks. The key motivation of this work is to reduce the vulnerable time after a vulnerability is discovered. Ratchet consists of two different sub-models, Ratchet-FL and Ratchet-PG, which handle the localization of faults and generation of patches respectively. Ratchet-PG combines techniques in information retrieval and sequence generation by augmenting source input with the closest patches.

4. We build a foundation for future works by investigating different code representations, e.g., feature-based, sequence-based, tree-based, and graph-based program representation, that can be employed in code intelligent tasks. Our objective is to identify the optimal code representation for different tasks (Code Classification, Clone Detection, and Vulnerability Detection). To inspire future research directions in the program representation field, we provide insights and qualitative analysis in our study.

1.5 Thesis Organization.

We organized the remaining sections as follows:

Chapter 2 addresses the preliminary and related work on deep-learning approaches and security-relevant works.

In Chapter 3, we discuss the work on CORE, an automated code review engine that recommends reviews based on historical data. We further discuss how our work can help to improve security and increase the productivity of developers.

In Chapter 4, our work on identifying security patches in open-sourced projects is presented. We further conduct extensive experiments on the effectiveness of our approaches on real-life projects

In Chapter 5, we present our work on Automated Program Repair, Ratchet. It employs a data-driven approach to repairing programs and security bugs.

In Chapter 6, we introduce our work on an empirical study on different code representations. Lastly, Chapter 7 talks about the future works and potential extensions to our enhanced security pipeline.

Chapter 2

Literature Review

In this chapter, we discuss past prominent works and techniques that have been used in relevant areas. We further discuss some preliminaries for Software Engineering (SE) tasks and software security tasks to give the reader a better understanding of deep learning on code intelligent tasks studied in this thesis. We discuss the prior works and research in software engineering and security in Section 2.1.

2.1 Deep Learning in Software Engineering and Security

The success of deep learning in other domains has motivated researchers to explore artificial intelligence on SE tasks. In addition, the concept of "natural code" has driven researchers into applying natural languages neural networks onto learning source code [17]. State-of-the-arts NLP and Computer Vision (CV) techniques and approaches can achieve outstanding performance across the diverse software engineering tasks, e.g., summarizing source code into summary [18–22], repairing buggy statements [23, 24], searching for source code [25–27], detecting code clones [12, 28, 29], generating API sequences [30], and detecting vulnerable source code [31–33]. Their outstanding results inspire researchers to dive deeper into code intelligence.

There is a constant demand for better performance and efficiency in code intelligence as code-related artifacts increase rapidly. Efficient AI approaches have to be researched and investigated to ensure that they can keep up with a data-intensive world. Large-scale language models are a prominent example in demonstrating the growth of deep learning. For instance, the

recent GPT-3 [34] contains about 175 billion of parameters, while the neural language model proposed by Bengio et al. [35] in 2003 only contains up to millions of parameters. Similar trends can be observed in code intelligence. For example, the recent source code language model, CodeBERT [36], trains on a dataset of more than 6 million methods across six different programming languages.

However, learning a meaningful representation of source code is a challenging task. Firstly, the source code contains many different representations, such as graph representation [6] and tree representation [12]. Investigating the impact of different representations becomes a focus before applying them to specific tasks. Secondly, source code datasets are also an important factor as there are thousands of code repositories available publicly. Hence, consolidating a unified dataset and benchmark is a crucial step. For instance, CodeXGlue [37] highlights the impact of a benchmark dataset that greatly benefits this research area.

Another key challenge in learning code representation is out-of-vocabulary (OOV) words. They are common in source code and this results in a sparse vocabulary during the learning of the model. For instance, developers can give variables and functions unusual names, such as "foo()" and "f()". These names rarely repeat themselves among the dataset, hence, they hinders the learning of source code representation as they do not have any semantic meaning. Hence, to address these challenges, in-depth research and study has to be conducted.

On the other hand, researchers have been trying to employ deep-learning across different code intelligence tasks. Commonly, these downstream tasks employs a domain-specific representations or neural networks to handle their specific problem. To have an overview of these downstream tasks, we discuss recent works in software engineering and security domain that are relevant to the thesis.

2.1.1 Deep Learning in Code Review Process

Code review is a tedious process where developers manually review the submitted source code for implementation bugs or guidelines disregard. Reviewers often give suggestions and feedback on the submitted source code, either to improve the quality of the source code or correct the source code due to the wrong implementation. Although tedious and time-consuming, code

reviews have been proven to be effective in enhancing the security of the codebase and passing knowledge propagation among the development team [38, 39].

Various research has been conducted on code review, such as predicting acceptance of submitted code and suggesting reviewers. Furthermore, review tools in the commercial market (e.g., Gerrit [40], Review Board [41], Critique [38]) facilitates better reviewing process. However, to date, there are only a few works that employ deep learning in recommending and suggesting review automatically. One of the works by Gupta et al., DeepMem [42], employs LSTM in learning the relevance between reviews and submitted code reviews. Our work dives further into suggesting code reviews by learning a meaningful representation for the source code.

2.1.2 Vulnerability Detection via Neural Networks

Vulnerable programs contain certain patterns and features, such as misuse variables and dereferencing a null pointer. Previous works try to employ software metrics and machine learning algorithms into identifying vulnerabilities in the program. Shin et al. [43] employ logistic regression and software metrics in vulnerability prediction on Mozilla codebase and achieve low precision. Similarly, Du et al. [44] employ complexity metrics as a feature to identify vulnerabilities. These techniques lack the generalization abilities of deep neural networks, hence achieving relatively low performance.

To further make use of big code, deep learning is used to learn and generalize these patterns to detect vulnerabilities among projects. Li et al. [32] employ BiLSTM into learning whether code snippets are vulnerable. They further extend their research by employing a program dependency graph in finding the dependency between variables [45]. Duan et al. [46] further extend the research by applying graph neural networks onto graph representation of programs. However, all these works employ the SARD dataset, which is well-maintained academic security defects and vulnerabilities. This highlights the need for a real-world dataset that depicts vulnerabilities and security bugs naturally.

2.1.3 Data-Driven Approach in Identifying Software and Security Artifacts

Open-source codebases bring forth other sources of security artifacts, such as bug reports and security patches. End-users can report bugs and implementation errors through the help of bug-

reporting tools, such as Bugzilla [47] and GitHub Issues [48]. Furthermore, online databases (e.g., National Vulnerabilities Database (NVD) [10], CVEDetails [11], Bugzilla [47]) are created to promote open source initiatives and allow easy propagation of information among the community. These websites and tools serve as a reliable source of security information for developers, researchers, and end-users.

Despite these websites consolidating relevant information into one site, there are still much unstructured data among them. For instance, security patches are not available for all fixed vulnerabilities even though they are disclosed on NVD. Hence, the searching of correlation between vulnerability-fixing commits and their corresponding CVE entry is an active area of research. The curation of these security data can motivate research in these areas, hence, further increasing the knowledge base. Zhou et al. [49] propose an ensemble of multiple machine learning algorithms in learning the messages of commits. The objective is to identify security patches based on their submitted commit message. Furthermore, Yang et al. [50] proposes an iterative approach with a k-fold stacking ensemble model, including K-Nearest Neighbours, Gaussian Naive Bayes, etc, in mining security artifacts in open-sourced projects. Bhandari et al. [51] propose an automated pipeline in extracting and collecting patches from the official database, i.e., NVD. Similarly, Yang et al. [52] aims to identify the libraries that are associated with the CVE entries on NVD via FastXML [53] algorithm.

2.1.4 Automated Program Repair (APR)

Debugging bugs and managing vulnerability is crucial in software development, where the productivity of the developers is a major concern. As codebases increase in their size, i.e., more lines of code, the number of bugs and vulnerabilities potentially increase. [54]. Due to the complex and tedious nature of repairing buggy programs, automatically generating patches for buggy programs has always been a focus of researchers. This automation can reduce the amount of labor and increase the level of software security in their codebase. Furthermore, APR tools can potentially serve as an intelligent educator for novice programmers in teaching syntax of programming languages. [55]

Program repair can vary in their generation techniques, from random searching of mutation operator [23, 56, 57] to learning patches through a data-driven approach [24, 58]. Generally,

APR approaches can be broadly categorized into three main areas: Heuristic-based, Constraint-based, and Learning-based approaches. Heuristic-based approaches employ mutation rules in mutating a fixed set of operators in a buggy program [23, 56, 57]. With the mutated program, it conducts verification to search for an optimal program that can potentially fix the negative test cases. Le Goues et al. [23] proposes one of the pioneering works in program repair with mutating buggy programs with a set of operators and genetic programming. By defining a set of mutated operators, it can potentially find the fixed program that fulfills the test cases. Wen et al [57] and Fan et al. [59] employ historical information in prioritizing mutations and learning to rank candidate patches among the plausible patches. These techniques are computational-intensive and the generated programs can overfit to the test cases [60]. Moreover, they suffer from search space explosions depending on the number of mutations and operators. This greatly affects the quality of their patch validation process.

Constraint-based approaches aim to generate patches that can fulfill certain constraints that are set by intended programs. [61–63]. These constraints are solved by their proposed constraint solvers and specific patches or fixed program are crafted based on the solved constraints. Nguyen et al. [61] changes certain statements in the buggy program and infer the constraints through its execution path. Patches can be generated by adhering to these constraints. Xiong et al [62] propose to create patches based on the branch condition and variable ranks through a set of pre-defined heuristic rules, such as the ordering of variables. Koyuncu et al. [64] learns from historical patches and extracts a template for the patches. By creating patches based on the mined templates, fixed programs can be generated. Similar to heuristic-based approaches, these approaches require a validation process due to the absence of ground truth, which is resource-intensive [65].

Learning-based approaches learn from historical patches and generate a patch statement or program based on the buggy input. Commonly, these learning-based techniques employ NLP techniques, such as Neural Machine Translation (NMT) and encoder-decoder model, to further enhance their patch generation techniques. For instance, Lutellier et al. [58] employs two encoders, buggy statement encoder and context encoder, to represent the buggy programs. Furthermore, Jiang et al. [66] further extends their approach with a context-aware beam search for generating programs. To incorporate additional information, Yasunaga et al. [24] employ

compiler diagnostic information in learning the location of the bug. The compiler diagnostic information also aids in the generation of the patches. To further incorporate transfer learning and fault localization task into automated program repair, Meng et al. [67] propose two models, TRANSFER-FL and TRANSFER-PR, to improve fault localization and program repair.

Chapter 3

CORE: Automating Review Recommendation for Code Changes

3.1 Introduction

The process of manually inspecting source code that is submitted by fellow developers is known as Code Review. It is commonly conducted by software engineers and open-source project developers before the revisions are forwarded to quality assurance or production. There are various motivations behind the code review process, such as transfer of knowledge, source code maintenance, and guidelines checking. A formal and structured framework, known as Fagan Inspection [68], was proposed in 1976 to formalize the code reviewing process. The framework provides proper guidelines and best practices for developers to identify software defects in source code or formal specifications during the software development life cycle.

Even though software maintenance and security are still key motivations behind code review [69, 70], developers are deeply concerned with the secondary benefits, such as transferring of knowledge, understanding of source code, and code improvement [38, 71, 72]. There is much ongoing research on various aspects of code review, such as analyzing the relationship between

The work in this chapter has been published in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)

code changes and reviews [73], detecting overall review sentiments [74], predicting the outcome or reviewer [75, 76], matching identical code review [42], analyzing security-related reviews [77–79], and discussing the motivation and key objectives behind code reviews [38, 71].

Developers often have trouble allocating time for reviewing source code as it is time-consuming. Furthermore, developers cannot keep up with the number of reviews that are needed. Moreover, reviewers need to have substantial knowledge of the changed code and codebase before a fair evaluation can be performed [71]. These requirements further increase the amount of time and effort needed for the code review process. To ease the burden of developers. researchers have been working on tools to increase the efficiency of the review process [42, 80, 81]. One prominent work by Gupta et al. propose a neural network to learn the relation between code changes and reviews in searching of the recommendation reviews based on code changes.

In practice, developers often use code collaboration tools, such as Gerrit [40] and Review Board [41], to assist them in the process of code review. Their functionalities often include showing changed files, allowing reviewers to reject or accept changes, and searching the codebase for related information. Some research work, *e.g.,* ReviewBot [81], incorporates static analysis tools to publish reviews automatically. These static analyzers detect defect code and unconventional naming by the submitter and publish them as part of code review. Static analyzers require a comprehensive set of rules that allow them to detect defective source code. In comparison with human-generated reviews, reviews by static analyzers are more rigid and have difficulty in finding errors that are emerged outside of their heuristic rules [82]. Other work, like DeepMem [42], aims to recommend reviews by learning the relevance between source code and review. However, these methods often require a large number of additional sources, such as full or partial source code. This additional information might not be available at all times, for instance, submitting a pull request or commits.

3.2 Motivation and Challenges

In this chapter, we propose a novel deep learning model for recommending relevant reviews given a code change. We name our COde REview engine as CORE. CORE is built upon only code changes and reviews without external resources. We have two key motivations behind

building our code review engine: (1) We aim to reduce the workloads of developers and engineers by recommending high-quality review with zero humans intervention. To reduce the turnaround time between reviewing and code changes, developers can amend their code as soon as the review has been conducted. (2) By automatically recommending high-quality reviews, we aim to reduce the number of security bugs in the code, hence, *facilitating better software security in codebases*. As a majority of the implementation bugs are discovered in the code reviewing process, software security can be enhanced through a strict code review process [83].

The challenging point of automating code review through neural networks is to learn good semantic representations for both sources. Distributed representations, (e.g., word2vec [84]) have been proven useful in representing the semantics of words. One of their limitations is that they fail in capturing out-of-vocabulary (OOV) words. As new terms and tokens can be added as variable names and function names, OOV words can be found in the project along the development phase [85]. To alleviate this problem, we propose a multi-level embedding layer to effectively represents OOV words.

3.3 Core Review Engine: CORE

3.3.1 Main Contribution

To overcome this limitation of the OOV problem, we propose a two-level embedding method that combines both word-level embedding and character-level embedding [86] for representing source code. We predict the consistency (i.e., a relevancy score) between source code and their corresponding reviews with an attentional neural network, where the attention mechanism [87] learns to focus on the important parts of the two sources during prediction. Our experimental analysis based on 19 popular Java GitHub projects demonstrates the effectiveness of our proposed CORE. CORE can significantly outperform the state-of-the-art model by 131.03% in Recall@10 and 150.69% in Mean Reciprocal Rank. Qualitative analysis also shows that project developers in the industry are interested in our work and agreed that our work is effective for practical software development.

In summary, our contributions are as follows:

- We propose a novel deep learning model, CORE, for recommending reviews based on code changes. Our model, CORE, employs multi-level embedding, i.e., character-level and word-level embedding, to learn the relevancy between source code and reviews. Through our proposed neural network design, CORE can effectively learn representations of code changes and reviews without external resources.
- Our experiments demonstrate the effectiveness of CORE against state-of-the-art models. We also perform an ablation study on the different components of our model, answering questions, such as investigating the embedding that is more effective in our approach.
- We provide a benchmark dataset containing 57K pairs of <code change, review> collected from 19 popular Java projects. We release our dataset [2] for future research directions.

3.3.2 Problem Definition

In this section, we formally define our problem and introduce some of the necessary terms for understanding CORE. We define the comments, suggestions or reviews, by reviewers as "*reviews*" (denoted by **R**) and the submitted code changes as "*code changes*" (denoted by **C**). Our objective is to find the top-k most relevant reviews in our collected set of reviews when given a code snippet of changes. We model the problem as a recommendation and ranking problem i.e., each code snippet has a list of recommended reviews that are sorted based on their relevancy score. Formally, for each $<c_i, r_i> \in <\mathbf{C}, \mathbf{R}>$, a score (*i.e.*, $Rel(c_i, r_i)$) that indicates the relevancy of the code changes c_i and review r_i will be learned through our proposed deep learning model, CORE. Specifically, we can compute the relevancy score as follows:

$$Rel(c_i, r_i) = F(c_i, r_i, \theta) \tag{3.1}$$

where c_i and r_i represent the code changes features and review features respectively, and θ represents the learnt parameters of CORE. We approximate F through our code review engine, CORE.

[2] `https://sites.google.com/view/core2019/`

3.3.3 Background

3.3.3.1 Code Review

As mentioned in Section 3.3.2, code reviews consist of two main components: submitted source code changes and reviews. Traditionally, code reviews are contributed by developers and reviewers manually, i.e., looking at each submitted code change and reviewing whether the changes are acceptable. These reviews are commonly short sentences and are made up of natural languages. More often than not, these reviews serve as suggestions or comments for the authors, informing them if there are any implementation errors or security concerns with the newly submitted code. There are various commercial tools, such as Gerrit [40], Review Board [41] and Google's Critique [38] that facilitate a better reviewing process by providing a clearer view of the reviewing items, such as history and previous comments on the same file.

The code review process is widely adopted by both industry and open-source development teams. Their main objective is to increase the quality of the codebase and reduce the number of bugs that might be introduced. We focus on code reviews in open-source projects, for instance, projects that are hosted on GitHub and contributors can submit pull-request to the project. GitHub allows contributors to submit their changes to the project through a function known as Pull Request [88]. Reviewers are required to evaluate these submitted changes on whether they should be merged into the main branch of the open-source projects, which is commonly conducted by the main contributors. These main contributors are usually experienced developers that deeply involved in the project. Intuitively, submitted code that contains implementation bugs or does not adhere to the guidelines is rejected by the reviewers. This process ensures the integrity and security of the project.

3.3.3.2 Motivating Examples

Listing 3.1 shows a motivating example of the type of reviews that CORE aims to recommend. As observed from Listing 3.1, the reviewer suggested that the contributor should use the function, *java.util.Collections.addAll()*, instead of using *completions.add(completion)*. By suggesting a different function, the reviewer ensures that the codebase is consistent and could help to avoid regression bugs. With CORE automating such suggestions, the reviewer could save their time in reviewing commonly used functions, hence, reducing the turnaround time for the next iteration of the review.

Another example in Listing 3.2 shows that a better naming convention should be used instead of *tcpMd5Sig()*. This is a similar use case as to Listing 3.1, which suggests a change of naming convention. Hence, our motivating examples shows our main objective is to reduce the workload of developers and increase the security of the codebase by recommending practical and atomic reviews.

```
// Code changes
+      completion.add(completion);
+    }
+  }
+  for (OffsetCommitCompletion completion: completions){
-------------------------------------------------
// Review
    "Consider using java.util.Collection.addAll()"
```

Listing 3.1: Review regarding API replacement

```
// Code changes
+  private volatile Set<InetAddress> tcpMd5Sig = Collections.emptySet();
-------------------------------------------------
// Review
    "Could this field and the tcpMd5Sig() method have a better name? It
   does not contain any signature but a set of addresses only."
```

Listing 3.2: Review regarding naming convention

3.3.3.3 Distributed Representation

Distributed representations are popular in Natural Language Processing (NLP) tasks. The key idea behind them is to learn a unique representation on representing a single word in the vector space. Each word is mapped to a unique numerical vector. For instance, given a word "*int*", we embed it into a vector of $\mathbf{x_1} = [0.123, 0.45, ..., 0.415]$ where $\mathbf{x_1}$ is a vector of length n. Similar words are embedded closely in the learnt latent space. Many prominent works employ neural network in learning distribution representation for NLP tasks, such as Word2Vec [84] and ELMo [89]. Word2Vec uses fully connected layer(s) to learn the context around each word and outputs a vector for each word, while ELMo uses Bi-LSTM to learn deeper word

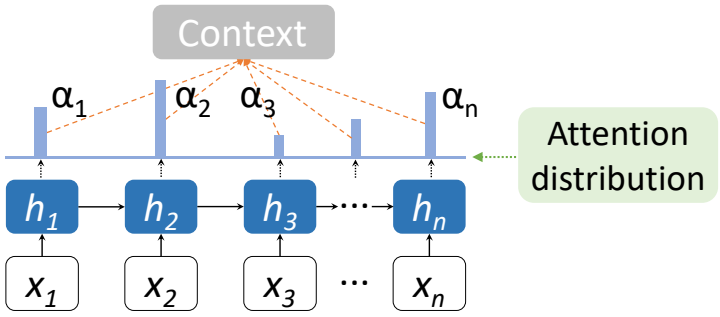


Figure 3.1: Attention mechanism

representations. Word embedding is often pre-trained to ensure that downstream tasks can be performed efficiently.

3.3.3.4 Long Short-Term Memory

Long Short-Term Memory (LSTM) [90] is a recurrent neural network that can learn long-term dependencies in sequential data, such as time-series or natural language. LSTM consists of three main gates, input gates, output gates and forget gates. Assume that the embedding sequence is in form of $X = [x_1, x_2, ..., x_2]$ where X represents the embedded sequence of tokens, LSTM computes the current output o_t based on its previous state and the memory cell state.

$$o_t = tanh(h_{t-1}, x_t) \tag{3.2}$$

where h_{t-1} represents the previous state and x_t represents the current input at time-step t. The $\tanh(\cdot)$ is an activation function for learning a non-linear adaption of the input.

3.3.3.5 Attention Mechanism

Attention mechanism was first proposed by Bahdanau et al. [87]. The main focus of the attention mechanism is to give different weights to the elements in a sequence, i.e., learning an importance score on each element. It is popular in the domain of NLP, specifically in Neural Machine Translation (NMT). Attention mechanism allows us to have better representation and provides a global context for each sentence that is beneficial to the relevant learning task in our work.

To employ an attention mechanism, we compute a context vector that is based on the tokens on the sequence and its importance score (i.e., attention weights). The computation of importance

score is based on the LSTM outputs, hence, giving them a higher-level representation. The context vector, shown in Fig. 3.1, is a weighted sum of the output of LSTMs by using attention weights, α_{ij}.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \tag{3.3}$$

Equation 3.3 shows the formula for computing the attention weights. An softmax layer are employed to ensure that the importance scores can be in range between 0 and 1. This give higher weight for variables that contribute more towards the success of the model. e_{ij} is learned using an activation function and linear layer computed on the output of LSTM.

$$e_{ij} = \tanh(W_s S_t + b) \tag{3.4}$$

where S_t is the output of LSTM, b and W_s is the bias and weight of the linear layer. After we achieved the attention weights, we can compute the context vector of the input sequence by the following equation:

$$\mathbf{a}_t = \sum_{i}^{n} S_t \alpha_{ij} \tag{3.5}$$

3.4 Approach

In this section, we present the overview of our proposed approach and design of our model, CORE. Furthermore, we present our design on extending the attention mechanism and LSTM model for code review recommendation. CORE learns a combined representation of both word-level embedding and character-level embedding, which is further enhanced through our multi-level attention layer. We consider code change and its corresponding reviews as two source sequences and use their relevancy label as the target for learning. Figure 3.2 shows the overall workflow of CORE, while Figure 3.3 shows the detailed architecture of our multi-embedding and multi-attention layer.

3.4.1 Overview

As observed in Figure 3.2, CORE consists of four crucial stages: (a) Data Preparation, (b) Data Parsing, (c) Model Training, and (d) Code Review. In the stage of data preparation and data

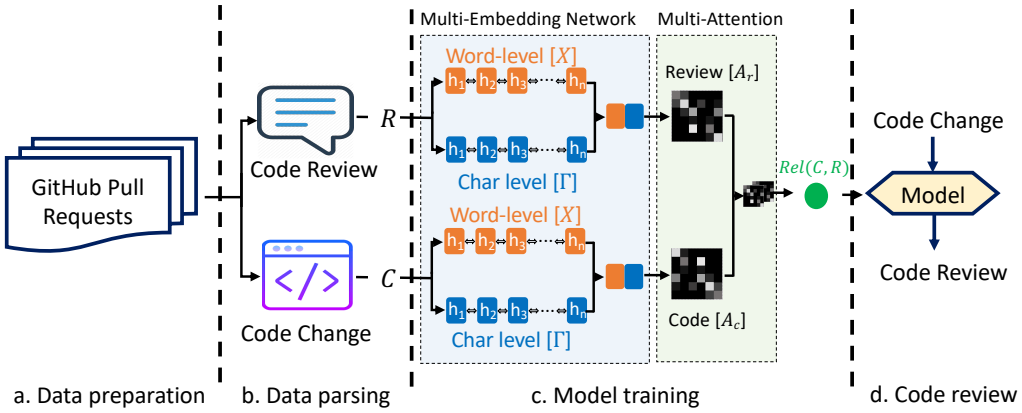


Figure 3.2: Overall architecture of CORE

parsing, we collect pull-requests from GitHub and preprocess the collected data. The data are curated into a corpus of code changes and their corresponding reviews, i.e., <**C**, **R**>. Our model training involves building our learning-to-rank neural network, CORE, based on this preprocessed corpus. Attention mechanism and LSTM are commonly used to solve natural language processing problems, such as text classification or neural machine translation. We apply both LSTM and attention mechanisms, in combination with word-level and character-level embedding. The major challenge during the training process lies in using limited information to well represent the two sources. Lastly, we deploy the trained CORE model for automated code reviews in the testing phase. We will explain the design and consideration of CORE in the following sections.

3.4.2 Multi-Level Embeddings

To better represent code changes and review texts, we propose to combine the two levels of embedding: **Word-level embedding** and **Character-level embedding**. Figure 3.3 shows the detailed structure of our multi-level embedding neural network.

3.4.2.1 Word-Level Embedding

Word embedding is widely adopted in representing the semantics of tokens through training a large corpus of text. One popular embedding technique is Word2Vec [84], a distributed representation of words that are pre-trained through a fully connected neural network. We employ word2vec [84] to represent tokens and words in CORE. Specifically, we pre-trained

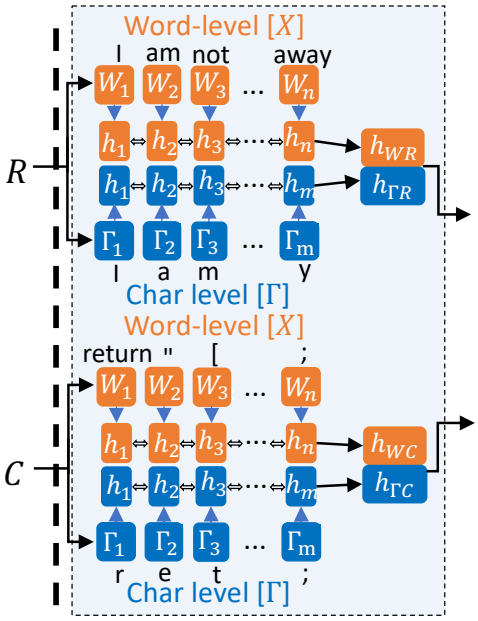


Figure 3.3: Detailed structure of CORE

two word2vec models, for code changes and reviews, and embed the tokens and words with their respective word2vec model.

3.4.2.2 Code Review Embedding

Code reviews are commonly mixed with texts and project-specific tokens. These project-specific tokens usually appear only a few times, including function names, variable names, version numbers of the projects, and commits hash ID. These low-frequency tokens cause the pre-trained word2vec embedding to be poorly learnt and may bring noise into the ultimate review text representation. For instance, as shown in our motivating example Listing 3.2, the variable, *'tcpMd5Sig"*, are only available in the project netty. We replace low-frequency words, (i.e., project-specific terms), with placeholders to alleviate the noises. Specifically, we convert hash IDs with "<HASHID>", numerical digits with "<NUM>", version numbers with "<VERSIONNUM>", and URLs with "<URL>". The preprocessed reviews are employed to pre-train the review-specific word embedding model.

3.4.2.3 Code Change Embedding

Source code and program representation are challenging tasks as they usually contains more low-frequency terms, such as temporary variables, string literals, and integers literals. Similar to code review, source code also contains project-specific words. Despite the low frequency of these terms, they can assist us in learning the semantics of the code changes. For example, a variable name, "SharedSparkSession", allows us to infer that the variable is closely related to a session and they are probably shared among multiple users. To parse the code changes into tokens, we employ a parser, *pygments* [91]. Pygments is a lexer that are mainly used to highlight keywords in programming languages. We employ their function of parsing sources into a list of their respective tokens. For example, given a Java statement, "*private final int shuffleId;*", we parse them using pygments and append each token to a list, such that we will receive a list of tokens, "["private", "final", "int", "shuffleId", ";"]". The preprocessed code changes are then fed into the word2vec trainer, from which we can obtain code-specific embedding.

3.4.2.4 Character-Level Embedding

Out-of-Vocabulary (OOV) and low-frequency code tokens are two common problems in code-related tasks [92–95]. As source code contains API methods and developer-named tokens, (e.g., variable names, function names, and class names), source code suffers from inadequate pre-training. Although word-level embedding can represent the semantics of the tokens in the source code, these issues still exist as low-frequency and OOV words are not included in the pre-training of the word2vec model. We propose to boost the performance of word embedding through an additional embedding layer, character-level embedding, to alleviate the OOV problem. Character-level embedding embeds each character with a unique representation. They are independent of tokens in the collection and can learn the dependency of each character through the usage of the LSTM layer. In this work, we embed each character into a one-hot vector. For example, given a review, "please fix this", we separate the sentence into a list of characters, ["p", "l", "e", "s", ..., "t", "h", "i", "s"]. We then embed the list of characters using one-hot embedding, such that each character has their unique one-hot representation.

3.4.3 Multi-Embedding Network

We propose to combine both embedding with multi-embedding networks that employ LSTM in learning the long-term dependency between the tokens and characters. Our proposed multi-embedding network jointly encodes the word-level and character-level representations for both code changes **C** and reviews **R**. The details of our multi-embedding networks are illustrated in Figure 3.3.

We denote the two-level embedding for both sources, denoted as $\mathbf{W}_C$ and $\mathbf{W}_R$ for the respective word-level embedding, and $\mathbf{\Gamma}_C$ and $\mathbf{\Gamma}_R$ for the respective character-level embeddings. We employ Bi-directional LSTMs (Bi-LSTMs) [96] to learn the sequence representations for word-level and character-level code changes, denoted as $\mathbf{h}_{WC}$ and $\mathbf{h}_{\Gamma C}$ respectively. $\mathbf{h}_{WC}$ and $\mathbf{h}_{\Gamma C}$ are the last hidden states produced by the corresponding Bi-LSTMs. The learnt representation for code changes are defined as:

$$\mathbf{h}_C = \tanh(\mathbf{W}^C[\mathbf{h}_{WC}; \mathbf{h}_{\Gamma C}]), \tag{3.6}$$

where $[\mathbf{h}_{WC}; \mathbf{h}_{\Gamma C}]$ is the concatenation of the two-level representations, $\mathbf{W}^C$ is the matrix of trainable parameters, and $\tanh(\cdot)$ is used as the activation function. Similarly, we obtain the two-level sequence representations $\mathbf{h}_{WR}$ and $\mathbf{h}_{\Gamma R}$ for code reviews **R**. The learnt representation representations for reviews can be calculated as:

$$\mathbf{h}_R = \tanh(\mathbf{W}^R[\mathbf{h}_{WR}; \mathbf{h}_{\Gamma R}]), \tag{3.7}$$

where $[\mathbf{h}_{WR}; \mathbf{h}_{\Gamma R}]$ is the direct concatenation of the word-level and character-level embedding for reviews R and $\mathbf{W}^R$ is the matrix of trainable parameters. For simplicity, we assume that the dimensions of the two-level embeddings, *i.e.*, $\mathbf{h}_{WR}$, $\mathbf{h}_{\Gamma R}$, $\mathbf{h}_{WC}$, and $\mathbf{h}_{\Gamma C}$, are the same.

3.4.4 Multi-Attention Mechanism

We further employ the attention mechanism [87] on the learnt representations of code changes and reviews, i.e., $\mathbf{h}_C$ and $\mathbf{h}_R$., to mitigate the influence of noisy input. The attention mechanism focus on the important words and characters that are representative of the code changes and

reviews. As observed in Figure 3.2, the representations of both word-level embedding and character-level embedding are further enhanced through our proposed attention layers:

$$\mathbf{a}_C = \sum_{i}^{n} \mathbf{H}_C \alpha_{ij}^{c} \tag{3.8}$$

$$\mathbf{a}_R = \sum_{i}^{n} \mathbf{H}_R \alpha_{ij}^{r} \tag{3.9}$$

where outputs, i.e., $\mathbf{a}_C$ and $\mathbf{a}_R$, are the learnt representations of code reviews and reviews after attending the important representations. The two attended vectors are concatenated into a *multi-attention* context vector, $\mathbf{a}_{\mathbf{C},\mathbf{R}}$. We further employ a linear layer for predicting of the relevancy score, $Rel(\mathbf{C}, \mathbf{R})$ between code changes $\mathbf{C}$ and reviews $\mathbf{R}$.

$$\mathbf{a}_{C,R} = [\mathbf{a}_C; \mathbf{a}_R], \tag{3.10}$$

$$Rel(C, R) = \tanh(\mathbf{w}^{\Lambda} \mathbf{a}_{C,R}), \tag{3.11}$$

where $\mathbf{W}^{\Lambda}$ is the learnable parameters of the linear layer and $Rel(C, R)$ indicates the predicted relevancy score between one code change and review.

3.4.5 Model Training and Testing

3.4.5.1 Training Setting

We determine the training goal as *Mean Square Error* loss function. The relevant code changes and review will have a score of 1 and the non-relevant code changes will have a score of 0. Therefore, we compute the loss function as follows:

$$Loss = \frac{1}{N} \sum_{i=1}^{N} (Rel(c_i, r_i) - \hat{Rel}(c_i, r_i))^2, \tag{3.12}$$

where $Rel(c_i, r_i)$ is the true relevancy label, $\hat{Rel}(c_i, r_i)$ is the predicted result of CORE, and N is the total number of code change-review pairs. We use ADAM [97] as our optimizer, with a learning rate of $1e-4$. The number of epochs is set to 50 and we use a dropout rate of 0.2.

The word embedding size and one-hot embedding size are set to 300 and 60. The number of hidden states for Bi-LSTMs is set to 400 and the dimension of the attention layer is set to 100. For training the neural networks, we limit the vocabularies of the two sources to the top 50,000 tokens that are most frequently used in code changes and reviews. For implementation, we use PyTorch [98], an open-source deep learning framework that is widely-used [99, 100]. We train our model in a server with one Tesla P40 GPU with 12GB memory and the training took around 35 hours.

3.4.5.2 Testing Setting

Similar to the training setup, we test our model using the setting as above. Each testing phase took about 10 minutes. Each review is ranked with a random set of 50 reviews which only one of which is the review with the true label. After which, we evaluate them using the metrics, Recall@K and MMR, which will be described in Section 3.5.2.

3.5 Evaluation Setup

3.5.1 Data Preparation

We curated our dataset from open-source projects that are available on GitHub, where communities frequently submit pull requests to update the projects. We selected our list of projects based on two criteria: (i) These projects are popular JAVA projects on GitHub. This criterion ensures that the high quality of crawled pull-request pairs. Furthermore, it filters away student projects and other projects that are of lower quality. (ii) These projects contain enough pull-request pairs, which necessitates an automated code review recommendation for code changes. As data-driven approaches require a high amount of data, we filter away projects with a low amount of pull-request. This criterion prevents us from crawling a large number of different projects to gather the pull-request.

To obtain projects that satisfy these two criteria, we inspect the projects ranked at the top 200 on GitHub in terms of the number of stars, and keep the ones with more than 400 pull requests. We selected 19 projects from this list of top-ranked projects. The full list of projects is as follows: mockito, swagger-core, selenium, karaf, spring-boot, okhttp, RxJava, spring-framework, kafka, neo4j, retrofit, nifi, deeplearning4j, orientdb, elasticsearch, hibernate-tools, netty, guava, junit4.

Table 3.1: Statistics of collected data

	Training Data	Validation Data	Testing Data
#Positive Samples	40,082	2,863	14,315
#Negative Samples	200,410	14,315	71,575
Total	240,492	17,178	85,890

We created a GitHub API crawler to collect code changes and corresponding reviews and ran our crawler in July 2019. In total, we crawled 85,423 reviews from these 19 projects.

To further ensure the quality of the experimental datasets, we conducted preprocessing to clean and organize our dataset. We filter out the reviews which are simply acknowledgment messages or feedback from the pull request author, i.e., the reviews or replies are authored by the pull request submitter. We then eliminate the reviews that are not written in English and convert all the remaining reviews into lowercase. We conduct word lemmatization using NLTK [101], where each word is converted into its base or dictionary form. After removing empty review texts, we obtained 57,260 <code change, review>pairs. We split the dataset into 70% of training set, 5% of validation set and , 25% of test sets.

However, the collected data are relevant pairs, i.e., they are code change and their corresponding reviews. In deep learning and data-driven approaches, negative samples, i.e., non-relevant pairs, are also needed for the training process to ensure that deep learning can learn from negative examples. To this end, we label the relevancy scores of all 57,260 change-review pairs as 1, i.e., these pairs are regarded as ground truth or positive samples. We follow the learning-based retrieval strategy [102] to generate negative samples of the dataset. Specifically, we randomly select m reviews corresponding to the other code changes as negative reviews of the current code change, i.e., $<c_i, r_j>$, where $i \neq j$ and the number of r_j equals m. We experimentally set $m = 5$ and generate a total of 343,560 <code changes, reviews>, including positive and negative samples. Detailed statistics of the experimental datasets are shown in Table 3.1.

3.5.2 Evaluation Metrics

We adopt two popular metrics for evaluating the effectiveness of our code review recommendation engine (CORE), namely, Recall@k [103, 104] and Mean Rank Reciprocal (MRR) [105,

106], which are widely used in information retrieval and the code review generation literature [107–109].

Recall@k measures the percentage of code changes for which more than one correct result exists in the top k ranked results [103, 104]. It can be computed as follows:

$$Recall@k = \frac{1}{|C|}\sum_{c \in C} \delta(Rank_c \leq k), \tag{3.13}$$

where C is a set of code changes, $\delta(\cdot)$ is a function which returns 1 if the condition is true and 0 otherwise. $Rank_c$ is the rank of the correct result of c in the recommended top k results. Following prior studies [42], we evaluate Recall@k when k is 1, 3, 5, and 10. A good code review recommendation should rank relevant reviews on the top of the recommended list, hence, the higher Recall@k is, the better the performance of CORE.

Mean Reciprocal Rank (MRR) is the average of the reciprocal ranks of results for a set of code changes C. It is a widely used metric to measure the quality of a retrieval system. The score shows the quality of a retrieval system by having an aggregation of the scores in the testing test. Similar to Recall@K, the higher the MRR is, the better the performance of code review recommendation is. The reciprocal rank of a code change is the inverse of the rank of the first hit result [105, 106]. MRR is defined as follows:

$$MRR = \frac{1}{|C|}\sum_{c \in C} \frac{1}{Rank_c}. \tag{3.14}$$

3.5.3 Baselines for Comparison

We compare the effectiveness of our approach with *TF-IDF+LR* (Logistic Regression with Term Frequency–Inverse Document Frequency) [110,111] and a deep-learning-based approach, *DeepMem* [42]. We use the same training and testing set for all three comparing models.

3.5.3.1 Term Frequency–Inverse Document Frequency (TF-IDF)

TF-IDF is a popular and conventional text retrieval engine. It computes the relevancy score between one code change and reviews text based on their TF-IDF [112] representations. The

training process is implemented by using the logistic regression (LR) method [111]. Specifically, we concatenate the TF-IDF representations of code changes and reviews as the input of the LR method.

3.5.3.2 DeepMem

DeepMem [42] is a state-of-the-art code review recommendation engine proposed recently by Microsoft. It recommends code reviews based on existing code reviews and their relevancy with the code changes. To learn the relevance between the review and code changes, the review, code changes, and the context, i.e. three statements before and after the code changes, are input into a deep learning network for learning. They employed LSTM for learning the relevance between code changes and reviews. We use similar settings according to their paper, such as LSTM dimensions and model components. Since the dataset is not publicly released by the authors [42] and our crawled data do not contain context information of code changes, we only take code changes and reviews as the input of DeepMem.

3.6 Evaluation

We conduct a quantitative analysis to evaluate the effectiveness of CORE. In particular, we evaluate our approach with the baselines with respect to several research questions:

- **RQ1**: What is the performance of CORE, in terms of Recall@K and MRR? Does CORE perform better than the comparing baselines?
- **RQ2**: Which module has a higher impact in CORE? The modules include word-level embedding, character-level embedding, and the attention mechanism. How effective is CORE as compared to character-level embedding only, word-level embedding, and CORE without attention?
- **RQ3**: What is the performance of CORE under different experiment settings? Do different parameters affect our models in terms of performance? We conduct our experiments with different hyperparameters to evaluate CORE.

Table 3.2: Comparison results with baseline models. $R@K$ indicates the metric $Recall@K$. Statistical significance results are indicated with * for $p - value < 0.01$.

Model	**MRR**	**R@1**	**R@3**	**R@5**	**R@10**
TF-IDF+LR	0.089*	0.019*	0.060*	0.100*	0.201*
DeepMem	0.093*	0.021*	0.065*	0.108*	0.208*
CORE	**0.234**	**0.113**	**0.247**	**0.333**	**0.482**

3.6.1 RQ1: What is the Performance of CORE?

The comparison results with the baseline approaches are shown in Table 3.2. We can see that our CORE model outperforms all baselines. As observed from the table, TF-IDF+LR achieved 0.089 in MRR score and 0.201 in Recall@10, while DeepMem achieved 0.093 in MRR score and 0.208 in Recall@10. In comparison, CORE achieve the best result of 0.234 in MRR and 0.482 in Recall@10. Among our comparing baselines, the TF-IDF+LR model achieves the lowest performance, with 0.019 in Recall@1, 0.201 in Recall@10, and 0.089 in MRR score. The result is consistent with the finding by Gupta and Sundaresan [42]. Our experimental results show that deep-learning models tend to better learn the semantic consistency between code changes and reviews.

As shown in Table 3.2, CORE outperforms our main comparing baseline, DeepMem, by 438.1% in Recall@1, 131.0% in Recall@10, and 150.7% in MRR score. To test the significance level, we further employ t-test and effect size measures for statistical significance test and Cliff's Delta (d) to measure the effect size [113]. Our significance test result ($p - value < 0.01$) and large effect size ($d > 1$) on five metrics confirm the superiority of CORE over TF-IDF+LR and DeepMem. The p-value shows that the baselines and CORE have a statistically significant difference between them. Hence, this implies that the recommended review by CORE is more relevant to the submitted code than our comparing baselines, i.e., TF-IDF+LR and DeepMem.

3.6.2 RQ2: Which Module Have a Higher Impact in CORE?

We perform an ablation study on the CORE and investigate the impact of three CORE modules, i.e., Character-level embedding, Word-level embedding, and Multi-Attention Network, by removing them individually from CORE. Specifically, one module is removed from CORE

Table 3.3: Comparison results with the different modules removed. The "CORE-WV", "CORE-CV", and "CORE-ATTEN" indicate the proposed CORE without considering word-level embedding, character-level embedding, the multi-attention network, respectively.

Model	**MRR**	**R@1**	**R@3**	**R@5**	**R@10**
CORE-WV	0.091	0.020	0.062	0.102	0.202
CORE-CV	0.103	0.026	0.077	0.123	0.233
CORE-ATTEN	0.233	0.107	0.246	**0.339**	**0.498**
CORE	**0.234**	**0.113**	**0.247**	0.333	0.482

in each experiment and we scrutinize the performance of CORE after the module is removed. Our experimental results are shown in Table 3.3. CORE-WV, CORE-CV, and CORE-ATTEN represent CORE without considering word-level embedding, character-level embedding, the multi-attention network respectively.

As observed in Table 3.3, CORE, with all three modules, achieve the highest performance among all other comparing variants of CORE. CORE-WV and CORE-CV achieve a low performance of 0.091-0.103 in MRR and 0.202-0.233 in Recall@10. As expected without both representation approaches, the model can only achieve comparable results as DeepMem. This demonstrates that CORE, with our two-level embedding approach, can combine the benefits of both representations to achieve better performance.

Compared with CORE-ATTEN, CORE only has slight advantages over CORE-ATTEN. The benefits of our proposed attention mechanism are not apparent. CORE-ATTEN achieves an MRR of 0.233, Recall@1 of 0.107, Recall@3 of 0.246, Recall@5 of 0.339, and Recall@10 of 0.498, outperforming CORE in both Recall@5 and Recall@10. This shows that without employing an attention mechanism, the model can perform slightly better than with attention in Recall@10 and Recall@5. Despite that, CORE has a higher performance in terms of MRR, Recall@1, and Recall@3. Generally, relevant reviews are preferred to rank as high as possible, hence, our proposed approach CORE can be regarded as performing better than CORE-ATTEN. With attention focusing on relevant keywords, CORE can learn much better in terms of relevancy, ranking related reviews with a higher score on MRR and Recall@1.

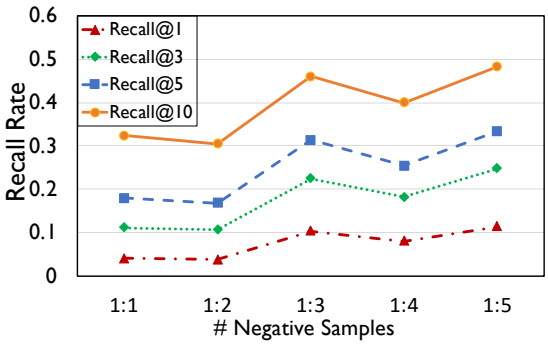


Figure 3.4: Impact on Recall

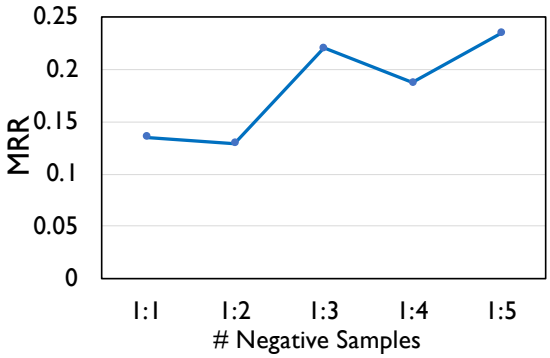


Figure 3.5: Impact on MRR

Figure 3.6: Impact of number of negative samples (m) on Recall@K and MRR

3.6.3 RQ3: What is the Performance of CORE Under Different Experiment Settings?

In this RQ, we evaluate the performance of CORE under three different parameter settings, e.g., number of negative samples (m) for each code change and review pair, the number of LSTM hidden units, and the word embedding size of word-level embedding.

We shows the experiments results when m varies on Figure 3.4 and Figure 3.5 in terms of MRR and Recall@K. We employ different values of m i.e., the number of negative samples per positive sample that are randomly selected for each <code, review> pair. To demonstrate the impact of m, we employ a range of m from 1 to 5, which is denoted as 1:1, 1:2, 1:3, 1:4, and 1:5. Generally, as observed in Figure 3.4 and Figure 3.5, CORE shows a general upward, but yet non-monotonic, trend. Our experiments show that when $m = 5$, CORE performs the best out of all other values of m. Although it is possible that CORE can perform even better when the value of m grows beyond 5, the processing and training cost increases exponentially when m increases. Hence, we set $m = 5$ to balance the training cost and performance of CORE.

The value of hidden units also impacts the performance of CORE. As observed in both Figure 3.7 and Figure 3.8, increasing the number of hidden units might not increase the performance of CORE after the value of 400. We observed that CORE performs the best when the number of hidden units is at 400. One of the possible reasons is that more hidden units might cause the model to overfit, causing lower performance during the testing phase.

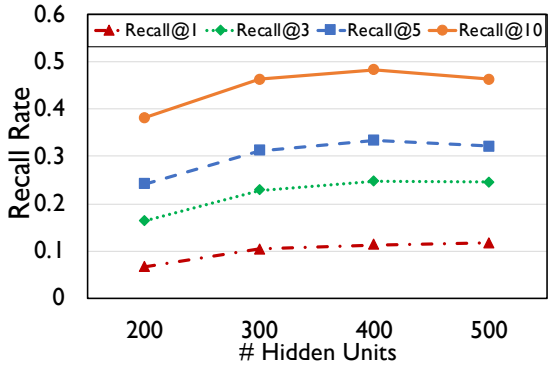


Figure 3.7: Impact on Recall

Figure 3.8: Impact on MRR

Figure 3.9: Impact of Hidden Units on Recall@K and MRR

Dimension of Word-level Embedding	MRR	Recall@1	Recall@3	Recall@5	Recall@10
100	0.154	0.057	0.143	0.204	0.338
200	0.219	0.105	0.224	0.308	0.452
300	**0.234**	**0.114**	**0.247**	**0.334**	**0.483**

Figure 3.10: Impact of different dimensions of word-level embedding

Table 3.4: Questions in the developer survey

Questions	#Participants
Q1. Do you think the recommended reviews are effective for practical development?	12
Q2. If you think the automated code review tool is useful, which parts make you think so? If not useful, which parts?	12

Figure 3.10 display the MRR and Recall@K when we vary the dimension of word-level embedding. We can observe from Figure 3.10 that more word dimension is beneficial to CORE as MRR and Recall@K increases proportionality to the dimension of word-level embedding. Hence, this shows higher-dimensional embedding can help in capturing better semantics of code changes and reviews. Despite the increase in performance, we also can observe that the performance increases minimally between 200 and 300, with an improvement of only 0.015 in MRR and 0.031 in Recall@10. Hence, we select 300 as our CORE parameter due to its optimum performance and cost balancing.

3.7 User Study

We conduct a user study with human evaluation to verify the effectiveness of CORE in automatically recommending reviews for developers. We consulted industrial developers from Alibaba, to prevent subjective viewpoint from the authors, in assessing the effectiveness of CORE as they are our industrial collaborator for this work. Alibaba is one of the largest e-commerce companies worldwide and its developers are proficient in software development and code reviews. We organize online meetings with the developers and approach their help in conducting the human evaluation. The user study involves 12 proficient Java developers that have at least 3 years of software development and experience in the code reviewing process. Specifically, we randomly selected 10 recommended reviews by CORE from each project on our collected 19 Java projects. We asked two key questions that are shown in Table 3.4 to the developers and collected their feedbacks.

Our collaborator developers showed great interest in CORE and they all agreed that CORE is useful and helps them in practical development. Furthermore, the developers provide feedbacks on the useful aspects and future improvements for our considerations, in response to Q2 in Table 3.4. Specifically, they find that CORE suggesting actions in our recommended reviews is useful in practical situations and can aid them during development phases. Moreover, they feel that there is room for improvement for CORE. For instance, there could be more indication on the target of the suggested actions. They also suggested that keywords should be highlighted to give more focus to the important actions.

It has always been in great demand to automatically generate code reviews, especially in the industry. Code review is a crucial, yet time-consuming, phase in software development that could greatly increase the security and quality of the codebase. However, the human cost of the process is substantial, hence, the demand for automated code reviewing is pressing. This is especially true for a big company, such as Alibaba, which has a large customer base and a large amount of developers that provide essential software development.

3.8 Discussion

To further understand why CORE works, we perform a qualitative analysis on CORE and discuss the limitation of CORE. Furthermore, we discuss the threats to validity of our work in dataset quality and evaluation.

3.8.1 Advantages of CORE

In this section, we explain the effectiveness of CORE recommendations by identifying two key advantages of CORE as compared to other baselines. Furthermore, we provide two case studies on the recommendation of CORE.

Advantage 1: CORE can well learn representations of source code and reviews. Our code review recommendation engine, CORE, employs multi-level embedding (i.e., character-level and word-level) in combination with a multi-attention network. Our proposed embedding layers can well-capture the semantics of both code changes and their corresponding reviews. Furthermore, the attention layer allows CORE to focus more on the keywords in each sentence, allowing it to encapsulate richer representations.

With attention mechanism providing a higher level context to the model, CORE can correlate the code changes and reviews based on the relevant topic in the sentences. CORE can better learn the representations of both sequences with our proposed approach. We highlight two real cases below to demonstrate the strengths of CORE as compared to the baselines.

```
// Code changes
-  return "[" + nodeId + "][" + taskId + "] failed, reason [" + getReason
   () + "]";
-------------------------------------------------
// Review by CORE
    "please add a message otherwise this just throws a
   nullpointerexception with no helpful message about what was null"
// Review by DeepMem
    "do we need to fully qualify this io netty handler codec http2
   http2stream state http2stream state"
// Review by TF-IDF+LR
    "do we need to fully qualify this io netty handler codec http2
   http2stream state http2stream state"
```

Listing 3.3: Case study on retrieving top review (return message)

Listing 3.3 shows the code changes and the top reviews recommended by CORE and other baseline models. As observed in Listing 3.3, CORE recommends review for the code change, providing actions such as adding a message or warning about an exception. CORE can learn the relevancy between semantically-related tokens, such as "*failed*" in the code and "*NullPointerException*" in the review. These two tokens are commonly used to express an exception or error that occurs unexpectedly. Hence, CORE can relate them together and focus on them during the learning phase. This shows that CORE can correlate the code change to the review better and hence, produce a higher relevancy score for such cases. In the case of the baseline model, both reviews recommended by the baselines do not capture the error-related token in the code changes and produce wrong results. We also discover that DeepMem and TF-IDF+LR prioritize the same review, which may be due to both models focusing on general words, such as "*return*" and "*nodeID*".

Listing 3.4 shows another example where CORE outperformed both DeepMem and TF-IDF+LR. The code change is mainly about adding a new test case that can be well captured by CORE (*e.g.*, the token "*tests*" in the review can be well-matched with "*test*" in the code). This semantic match cannot be observed in the suggested reviews of the baselines. For instance, the prioritized review of DeepMem discusses string manipulation and the top review of TF-IDF+LR talks about versioning, which in both cases are not relevant to testing in general. By contrast, the top retrieved reviews by CORE managed to learn that the source code relates to a test case and could retrieve reviews that are closely related to the test cases.

```
// Code changes
-package org.elasticsearch.node;
-import org.elasticsearch.common.collect.ImmutableOpenMap;
-import org.elasticsearch.common.settings.Settings;
-import org.elasticsearch.test.ESTestCase;
-import static org.hamcrest.Matchers.equalTo;
-public class NodeModuleTest extends ESTestCase{
- public void testIsNodeIngestEnabledSettings(){
-----------------------------------------------
// Review by CORE
    "can you split this into two different tests rather than using the
   randomness here i dont think it buys us anything"
// Review by DeepMem
    "i'd make the string here junk or not interpreted or something if you
    dont read the file carefully it looks like the script is run because
   that is a valid looking script"
// Review by TF-IDF+LR
    "we need the version check here as well for bw comp"
```

Listing 3.4: Case study on retrieving top review (test cases)

Advantage 2: CORE can better solve Out-of-Vocabulary (OOV) issues.

In this section, we present two case studies for illustrating the effectiveness of adding character-level embedding to CORE to alleviate OOV problems.

Listing 3.5 shows an example of changing the chain of function calls. For instance, the function, *"getSnapShot()"* are added to the submitted change code. We observed that CORE recommends a review suggesting adding comments for the handling and deletion. This recommendation review is strongly relevant to the code change which revolves around deletion. Without the character-level embedding considered, CORE-WV ranks a different and unrelated review at the

top. One possible reason is that CORE-WV could not well learn the semantic representation of "*deleteSnapshotListener*" and "*deleteSnapshot*" due to the low co-occurrence frequency[3] of the subwords (such as "*delete*" and "*snapshot*"). A similar case can be found in Listing 3.6, CORE-WV cannot well capture the semantics of the variable "*inSyncAllocationIds*" while CORE can learn that the token is semantically related to "*allocation*". Thus, with the addition of character-level embedding, CORE can better focus on the important characters that are not commonly represented in word-level embedding.

```
// Code changes
- deleteSnapshot(snapshot.snapshotId(), new DeleteSnapshotListener(){
+ deleteSnapshot(snapshot.snapshotId().getSnapshot(), new
    DeleteSnapshotListener() {
-----------------------------------------------
// Review by CORE
    "can you add a comment here as to why we have a special handling for
    external versions and deletes here"
// Review by CORE-WV
    "should values lower than versionnum be allowed here"
```

Listing 3.5: Case study on comparing CORE-WV & CORE

```
// Code changes
-   inSyncAllocationIds.contains(shardRouting.allocationId().getId()) ==
    false)
+   inSyncAllocationIds.contains(shardRouting.allocationId().getId()) ==
    false &&
+   (inSyncAllocationIds.contains(RecoverySource.
    ExistingStoreRecoverySource.FORCED_ALLOCATION_ID) == false
-----------------------------------------------
// Review by CORE
    "can we remove this allocation"
// Review by CORE-WV
    "that s just a minor thing but i think the recommended order in the
    java styleguide is static final"
```

Listing 3.6: Case study on comparing CORE-WV & CORE 2

[3]The token "*deleteSnapshotListener*" only appears 349 times among the whole GitHub repository [114].

3.8.2 Limitations of CORE

We show an example of CORE limitations in Listing 3.7. Our evaluation shows that CORE ranks the ground truth of this example at position 30. The low relevancy score can be attributed to the lack of related keywords between the submitted code and the review. CORE fails in capturing the relevancy between the two sources as there are no relevant keywords among them. Furthermore, the reviews are generic and only contains commonly used keywords, e.g., "*supposed*", "*were*") and it does not have keywords that are relevant to variables in the source code, e.g., "*asset*" or "*HashSet*".

Despite the poor performance of CORE, the top-recommended reviews by CORE show that our approach can capture the main semantic of the code changes. In the code changes, the function "*assetEquals*" is invoked to evaluate a condition. CORE infers that the submitted changes involve checks and assertions; hence it recommends a review that is closely related to assertion. This demonstrates the ability of CORE to focus on key information in the code changes and able to capture the general idea of the code changes. In addition, our design on CORE is flexible and extensible to integrate additional external information, such as code structure, code comments, compilation messages, to learn a better representation of the source code semantic and its corresponding review.

```
// Code changes
-  assertEquals(3, exec("def x = new HashSet(); x.add(2); x.add(3); x.add
   (-2); def y = x.iterator(); " +
-  "def total = 0; while (y.hasNext()) total += (int)y.next(); return
   total;"));
-----------------------------------------------
// Ground Truth
    "were these supposed to be removed"
// Review by CORE
    "that check should be reversed we assert that it s not null the else
   part dealing with the case when it is"
```

Listing 3.7: Case study on limitation of CORE

To further investigate the limitation of CORE, we conduct an experiment on the length of the code tokens and the performance of CORE. Table 3.5 shows the MRR and Recall@K for different code lengths, in the intervals of 25. Our experiments show that CORE performs better for

Table 3.5: Performance of CORE regarding different code token length

Code Length (l)	**MRR**	**R@1**	**R@3**	**R@5**	**R@10**
l <25	0.1954	0.0789	0.2171	0.2828	0.4342
25≤ l <50	0.2186	0.1029	0.2242	0.3010	0.4685
50≤ l <75	**0.2423**	**0.1246**	**0.2468**	**0.3441**	**0.5000**
l >75	0.2352	0.1138	0.2490	0.3353	0.4829

the code changes with lengths ranging from 50 to 75. Longer code sequences are trimmed to a fixed length, and only the beginning part of the sequence may not fully represent the semantics of the code changes. Hence, this might contribute to why CORE performs better in source code that is within the length of 50 to 75.

3.8.3 Threats to Validity

3.8.3.1 Subject Dataset

The quality of our dataset is one of the major threats to its validity. Overlap of data in the training and testing set should not occur as it greatly affects the training process and results. Specifically, for any pair of <code changes, review>, the negative sample of the pair exists in the training set and the positive sample of the pair exists in the testing set or vice versa. This provides the training model with side channels on the relevancy of the pair. To ensure that our training and testing set do not have any overlapping pairs of <code changes, review> we split our dataset into three distinct sets, train, validation, and test set, before we negative sampling them. Hence, these sets will have their own unique set of positive and negative pairs.

3.8.3.2 Comparison with DeepMem

The results of DeepMem can be another threat to our work validity as our implementation of DeepMem achieves a result lower than the original model. The result on DeepMem in the original paper [42] shows a Recall@10 of 0.227 and 0.2 MRR score, while our implementation only achieves an MRR of 0.093 and Recall@10 of 0.208. This can be attributed to the different code contexts that are employed. In the original paper, DeepMem employs additional code contexts, i.e., three statements of code between and after the code changes. We do not consider

these contexts as our crawled data do not contain such information. Furthermore, they did not release their data, hence we cannot compare with them on their dataset. We carefully review the technical part of DeepMem in the published paper and confirm our implementation with other co-authors of this work. Furthermore, we evaluated both CORE and our implementation of DeepMem with similar settings and on the same dataset for a fair comparison

3.8.3.3 User Study

Another possible threat to the validity of our work is the quality of the user study. As user study involves perspectives from developers, hence, this feedback might be subjective and varies from different developers. We mitigate this threat by seeking experienced developers that have at least 3-5 years of coding experience. To further ensure validity, we make sure that they have at least prior experience in code reviewing in the industry.

3.9 Conclusion

In this chapter, we propose our novel multi-level embedding with attentional relevancy neural network, CORE, for learning the relevancy between code changes and their corresponding reviews. Our novel embedding layer employing both word-level and character-level embedding aims to capture both the semantic information in source code and reviews. Furthermore, we propose a multi-layer attentional network to focus on the important tokens and characters for the sources and aim on recommending the best review for code changes. Our future work can be improved by considering the usage of neural machine translation and better representation for code changes.

Chapter 4

SPI: Automated Identification of Security Patches via Commits

4.1 Introduction

The open-source initiatives greatly impact the landscape of software developments. A diverse range of applications, from operating systems to encoding software, publicly release their code-base as Open-Source Software (OSS) in hopes of engaging the community in maintenance and debugging. Furthermore, the rapid increases in OSS also bring forth more reporting of vulnerabilities and exploitation due to the nature of public disclosure. It is reported that there is an 88% increase of reported vulnerabilities from 2017 to 2019 [115]. This problem is further worsened by a large number of unreported security issues. As reported by GitHub in 2019, there are 7.9 million security alerts remediated by software developers [116]. However, there are only 17,344 Common Vulnerabilities and Exposure (CVE) that are officially reported on NVD in 2019. These problems highlight that there is a mismatch reporting of a large number of security issues and vulnerabilities across OSS. Furthermore, these vulnerabilities are fixed without explicitly reporting them on official databases, i.e., silently patching the vulnerabilities.

Commonly, there are many online databases, e.g., NVD [10] and CVEDetails [11] that provide information on vulnerabilities and security patches. Furthermore, commercial companies,

The work in this chapter has been published in ACM Transactions on Software Engineering and Methodology Volume 31 Issue 1 January 2022

e.g., SNYK [115] and Whitesource [117], also provide an alternate source of security information to attract users in purchasing their products. Despite that, almost 70% of security issues are patched before they are public disclosed on these security information sites [118]. OSS developers neglect the importance of these patches or simply cannot keep up with the security management in their huge codebase. Hence, end-users cannot have first-hand knowledge of the updated security issues. To avoid delay of security updates, these patches should be identified and updated on the OSS as soon as possible.

4.2 Motivation

Our key objective is to *complement the mining of explicit and implicit security patches among OSS by employing deep learning techniques into identifying security patches amid a large amount of OSS commits*. Our work is valuable in both commercial application and research value. Commercially, vulnerability information, such as the source of vulnerability, security patches of the CVE, and details of the CVE, are periodically consolidated by security companies, in hopes of creating their valuable security database. Commercial companies, e.g., SNYK [115] and Veracode [119], curated these security databases and build security management software based on them. Dependency scan is one of the prominent examples of how this security information can utilize to enhance software security. By curating a comprehensive security database, developers can locate security patches easily and find existing fixes. Furthermore, commercial companies can complement their database based on our techniques and approaches. Our research aims to enhance these security knowledge bases and provide high-quality security data to developers, researchers, and commercial companies.

Besides commercial values and extending further security management application, our work can also provides great research value to the academic community. Our dataset can provide high-quality security patches at a large scale and employ in different vulnerability research. Several works [24, 120, 121], that focus on automated program repairs and vulnerability detection, can employ our dataset in conducting their code intelligent tasks. Higher quality security patches can potentially increase the performance of their approaches, hence improving the research area.

4.3 Challenges in Security Patch Curation

Despite previous works employing data-driven approaches in finding vulnerabilities and security artifacts in OSS [32, 122–124], they commonly employ handcrafted features, such as the number of operators and loop [122]. Vulnerabilities and exploitation can vary from projects and these handcrafted features might not be feasible for larger projects as they might require intensive resources.

We identified three challenges that can potentially aid us in crafting a deep-learning approach on curating high-quality security patches: (1) Since there is no publicly available dataset on security patches, we need to collect our own security patch dataset. Patches and security commits are rare in OSS as compared to other commits. Gathering such a dataset could require massive manual manpower and time. Furthermore, the information on commits, e.g., commit messages and code revision, are lengthy and contain non-natural languages information. (2) The second challenge is to build a neural network that can effectively learn from the commit message of the security patch. These commit messages can help us to identify key features on commit messages that can differentiate security patches from commits. (3) The last challenge involves learning from the code revision of the commits. Employing source code features in identifying vulnerable functions and code snippets has been an emerging research approach [32, 125]. Different from these approaches, code revision often contains less context and information as compared to file-level, program-level, and function-level source code. Furthermore, implicit distinctions between security-related and unrelated codes render distributed representation of code (e.g., Code2Vec [13]) inapplicable to obtain valid representations of vulnerabilities.

4.4 Security Patch Identifier: SPI

4.4.1 Main Contributions

We propose our data-driven approach in automatically identifying security patches in open-source software and codebases using deep neural networks, **S**ecurity **P**atch **I**dentifier (SPI). We address each aforementioned challenge in Section 4.3. To address the first challenge, we propose a sophisticated data curation process of two steps: Keyword filtering and Manual Verification. Our process curates a balanced dataset and we ensure data integrity and reliability

through a manual verification process. Furthermore, our curated dataset is diversified, i.e., collected from four different C open-sourced projects, that span from the operating system, emulators, multimedia-related libraries suit and network application. This could aid us in the second challenge in finding key features across different C Projects. We designed a statement-level Long-Short Term Memory (LSTM) code revision model to address our third challenge. Our model ensures that features from code revisions can be learned effectively, at the statement level, as opposed to learning function-level or file-level source code. SPI automatically captures semantics from both commit messages and code-revision of security patches to identify them among the commits.

Our contributions are summarized as follows:

1. We propose a filtered-based process/mechanism to build a security patch dataset. 344,519 unlabeled commits are collected from four open-source projects, Linux, Wireshark, FFmpeg, and Qemu. Our proposed process managed to curate 40,523 security patches and non-security commits out of the collected commits. The integrity and quality of the dataset are assured by our manual verification process, where four professional researchers spend up to 600 man-hours on the labeling of the dataset.

2. We designed and implemented Security Patch Identification system based on commits. We employ two deep neural networks in learning the commit-message and its code revision respectively to determine the nature of the commit, i.e., if it is a security patch or non-security-related commit. We designed two components: a commit-message deep neural network (SPI-CM) that learns from commit message and a code-revision deep neural network (SPI-CR) that automatically learn features from code changes in commits. Furthermore, we ensemble them to make use of both learnt features by using a weighted combination to form the final model, SPI.

3. We evaluate SPI and our experiment result shows that SPI achieves an 87.93% F1-score and 86.24% Precision in identifying security patches in our curated dataset. We achieved an average of 9.94% in improving the F1-score when compared to three baselines over five different datasets. We further conduct qualitative analysis on the security patches that have no clear indication of fixing the vulnerability in their commit messages to demonstrate the ability of our model to learn from code revision. The analysis shows

Table 4.1: Commit example from Linux kernel

Hash	98051872fd25077d3b106ab3d2b945fa7025c1ef
Date	Thu, 14 Dec 2017 13:03:17 +0100
Author	Lorenzo Bianconi <lorenzo.bianconi@redhat.com>
Message	Subject: [PATCH] mt76: fix possible NULL pointer dereferencing in mt76x2_mac_write_txwi(). Verify wcid is not NULL before dereferencing the pointer to initialize txwi ratepower info
Additive Changes	+ *if (wcid && (rate->idx < 0 || !rate->count)) {*
Subtractive Changes	− *if (rate->idx < 0 || !rate->count) {*

that 75% of implicit patches can be identified as security patches by SPI-CM solely by their code changes.

4. To further show the efficiency of SPI, we deployed SPI in the production cycle of our industry collaborator. 50.54% (151,080 of 298,917) of the production dataset are excluded from our production cycle as SPI identified them as non-security-related commits. This performance shows the effectiveness and practicality of SPI.

4.4.2 Problem Definition

We formally define the problems and terms in this section. A commit that fixes vulnerabilities in the source code are defined as ***security patch*** (SP), and otherwise ***non-security patch*** (NSP). Table 4.1 shows an example of Security Patch (SP) example from Linux Kernel [126], where the patch fixes *NULL pointer deference* [127]. As observed from the example, the commit message contains several projects and domain-specific words, e.g., *mt76, dereference, wcid, txwi*. Furthermore, each commit records the additive and subtractive changes of the source code. In our example in Table 4.1, the amended source file, *mt76x2_mac.c*, has one additive and one subtractive changes.

In this work, we focus on *the problem of identifying security patches, i.e, the commits that fix vulnerabilities, through the approach of deep learning.* We focus on general security patches

that fix any vulnerabilities in a open-source codebase. Our objective can formalize as finding function F and F can be defined as follows:

$$F(x_i, \theta) = p_i \tag{4.1}$$

where F takes in input commit x_i and outputs a probability p_i. Probability p_i indicate the likelihood of x being a security patch. In addition, θ represents the learnable parameters of the deep neural network models. Each commit is labeled as *positive* (denoted as 1) if it is a security patch or otherwise, *negative* (denoted as 0). We employ cross-entropy loss [128] as our objective loss function to deal with our binary classification problem.

$$L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)] \tag{4.2}$$

where n is the number of training commits; and y_i is the label of commit x_i.

4.4.3 Background

4.4.3.1 Embedding Techniques

Commit messages and code revision are represented by words and code tokens, both are in textual information. However, neural networks work on vectors where each vector uniquely represents a single unit of textual information. Hence, embedding techniques are required to transform discrete input objects into their unique numerical representations. Specifically, a *word embedding* $\mathbf{W}$: words $\rightarrow \mathbb{R}^n$ is a parameterized function mapping words or tokens to k-dimensional vectors. Code snippets and sentences can be represented by a matrix, where each word is denoted by a vector. e.g.,

$$\begin{aligned} Fix \rightarrow \quad & [\ 0.12, 0.05, -0.03, \ldots, 0.02; \\ a \rightarrow \quad & \ \ 0.01, 0.50, -0.10, \ldots, 0.11; \\ crash \rightarrow \quad & \ \ 0.06, 0.01, -0.04, \ldots, 0.01\ ] \end{aligned}$$

Various embedding methods, such as one-hot encoding, bag-of-words, term frequency-inverse document frequency [129], and word2vec (Continuous Bag Of Words (CBOW) and Skip-Gram) [84] can be employed to provide the model with alternative features. In this work, we employ word2vec [84], a feed-forward neural network that models the relationship between a word and its context.

4.4.3.2 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) uses three gates, input gate, output gate and forget gate, to model long-term dependencies between sequences, such as time-series and natural languages [130]. The forget gate remembers long-term dependency in the sequences, while the input gate and output gate control the information of input and retain the important information respectively.

$$h_i = tanh(x_i, h_{i-1}) \tag{4.3}$$

For a current output h_i, a hyperbolic tangent is performed on the current input, x_i and the previous state h_{i-1}. We employ LSTM in our commit message and code-revision learning to learn on the long-term dependencies between sequences of words/tokens.

4.4.3.3 Convolutional Neural Network

Convolutional Neural Network (CNN) has been proven its performance on text classification problem [131, 132]. It has been used to learn high-level features among words in a sentence through filters and convolutional operation [133].

$$c_i = \sigma(w \ldots x_{i:i+k} + b) \tag{4.4}$$

A feature c_i is learned by performing the convolutional operation over a window of K words, whereas w and b are the weights and bias of the learnt model respectively.

4.5 General Approach of SPI

4.5.1 Overview of SPI

Our design of SPI consists of three key milestones: (1) Extraction of a broad range of security patches from commits (2) Learning of features from commit messages and code revisions and selecting high-level features to obtain an informative latent semantic representation. (3) Efficiently ensemble the learnt unified representations to build a classifier that determines if a commit is a security patch. Fig. 4.1 highlights the framework of SPI, with Figure 4.1(a) depicting the learning phase and Figure 4.1(b) illustrating the prediction phase. During the learning phase, our approach employs neural networks and our curated dataset to learn the features of

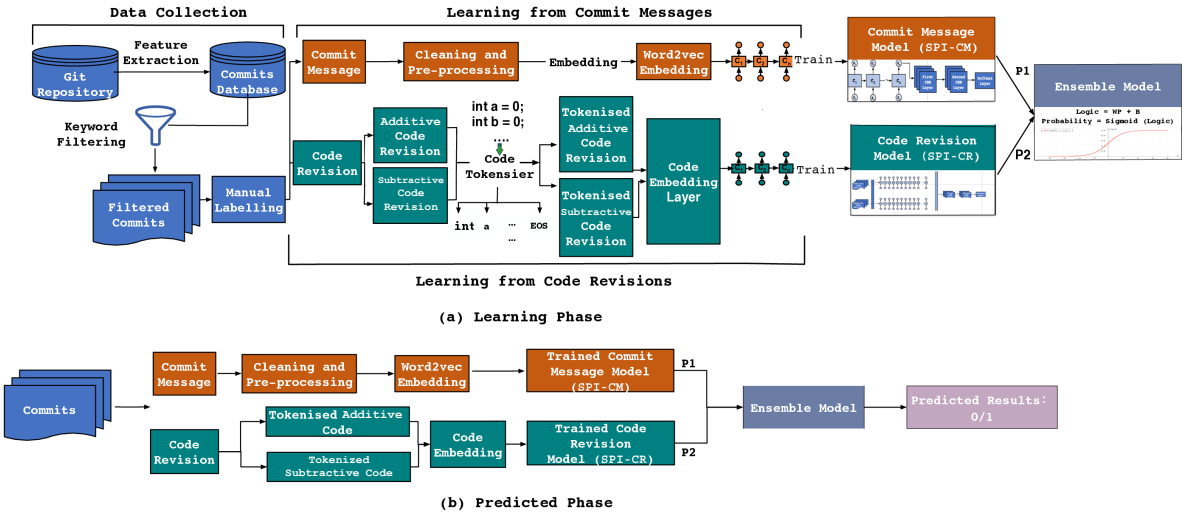


Figure 4.1: System Overview of SPI

commits. During the prediction phase, we employ the trained SPI to predict if a given commit patches a vulnerability.

Our learning phase consists of the four major building blocks:

- *Data Curation*. We design a keyword filtering and manual verification process to exclude the majority of security-unrelated commits as there is a notably low rate of security-related commits in OSS.
- *Learning from Commit Messages*. We propose a deep neural network, called SPI-CM, that is built upon the textual information of commit messages to identify security patches among commits.
- *Learning from Code Revisions*. Similar to SPI-CM, we design two-layers deep neural networks that utilize additive and subtractive code changes to identify security patches - SPI-CR.
- *Ensemble Learning*. To further utilize the learnt representation of SPI-CM and SPI-CR, we propose an ensemble layer to combine SPI-CM and SPI-CR into SPI.

4.5.2 Guiding Principles for Design of Deep Neural Networks

As there are neural networks for different applications, we propose a set of principled guidelines to design effective neural networks to learn from commit messages and code revisions in the

task of identifying security patches amidst commits.

Commit Message Learning. Sequence-based networks, such as LSTM and RNN, are effective in coping with sequential information (e.g., natural languages). Generally, commit messages are made up of words of natural languages, though including file paths, code snippets, and other noises. *Our proposed networks are required to understand the context of the commit messages and able to identify whether they describe a security patch.* To this end, we choose LSTM as our neural network in learning key features from commit messages. Furthermore, the average length of commit messages is 69, as opposed to Tweets which is around 10 [132]. This shows that we do not need to extend extra design to incorporate additional context learning and it is well within the learning capacity of LSTM networks.

Code Revision Learning. Source codes are sequential and contain structural and semantic knowledge. Learning from these data is challenging as they are not in natural languages and they do not explicitly define security patch characteristics. Previous work [32] shows LSTM is capable of capturing the context in the source code to a certain extent. However, *yhe neural networks have to distinctly identify the difference between the additive and subtractive code changes to identify security patches.* Hence, we cannot directly feed the whole changed code without highlighting the differences to our proposed networks. Keeping these concerns in mind. we propose two LSTMs model to learn the context from additive and subtractive changes, further followed by two layers of CNN to learn higher-level features of the code changes, statement by statement.

4.5.3 Data Curation

4.5.3.1 Sourcing of Data

We collected our security patches from two sources, *NVD* and *Manual Verification of Commits.* Although generally limited, the CVE patches on NVD can serve as labeled data of security patches as they are useful to our data collection. We managed to gather *only 1,045 security patches* from NVD, among our total dataset of 40,523 commits. Therefore, we need an alternative source for more security patches.

Table 4.2: Overview of datasets

Project	Total	Filtered	CVE	Filtered Ratio	SP Ratio
Linux	144601	10731	238	0.0742	0.6948
FFmpeg	80882	12123	249	0.1498	0.4578
Qemu	55070	10128	242	0.1839	0.4306
Wireshark	63966	7541	316	0.1178	0.4155
Total	344519	40523	1045	0.1176	0.5059

4.5.3.2 Keyword Filtering and Manual Verification

We extracted a wide range of security-related commits from selected libraries by following a two-step process: *Commits Filtering* and *Manual Verification*. We use four popular and diversified open-source libraries, i.e., Linux, FFmpeg, Qemu, and Wireshark. For instance, Linux is an operating system and Wireshark is a popular network packet analyzer. This diversified dataset allows us to generalize our model onto different C libraries. To collect the necessary information for our model learning, we collect their commit details, messages, and patches from these projects, except for Linux. Due to a large number of commits in the Linux repository (up to 750 thousand at the time of our curation process), only commits between 2016 and 2017 are collected. For the other three projects, we crawled their commits up to January 2018.

4.5.3.3 Commits Keyword Filtering

We show the statistics of the collected commits across different projects in Table 4.2. Each project has at least 55k commits and the majority of these commits are irrelevant to vulnerabilities or security issues. Hence, there is a need to filter irrelevant commits to curate a balanced security patch dataset. First, we process all the collected raw commits with a keyword filtering process. We derived a list of general and library-specific keywords by manually inspecting the security patches of CVEs. The common words in these patches are extracted and they form our list of security-related keywords. We then employ Regular Expression to exclude commits where commit messages are not matched with our pre-defined set of security keywords. We show our keywords in Table 4.3. The keywords can be broadly categorized into two categories: library-specific and general. Library-specific keywords, such as 'oops' or 'KASAN', are common in Linux. We also observed that memory-related vulnerabilities keywords are matched

Table 4.3: Keywords in Filtering Process

Types	Keywords List
General	*out of bound, use after free, double free, divide by zero, overflow, illegal, leak, disclosure, improper, unexpected, sanity check, uninitialize, fail, null pointer dereference, null function pointer, crash, corrupt, deadlock, race condition, denial of service, CVE, exploit, attack, vulnerable, fuzz, verify, security issue/problem/fix, privilege, malicious, undefined behavior, exposure, remote code execution, open redirect, OSVDB, ReDoS, NVD, clickjack, man-in-the-middle, hijack, advisory, insecure, cross-origin, unauthorized, infinite loop, authentication, brute force, bypass, crack, credential, hack, harden, injection, lockout, password, proof of concept, poison, privilege, spoof, compromise, valid, out of array, exhaust, off-by-one, privesc, bugzilla, limit, craft, overrun, overread, override, replay, constant time, mishandle, underflow, violation, recursion, snprintf, initialize, prevent, guard, protect*
Library-Specific	*KASAN, general protection fault (GPF), oops, panic, syzkaller, trinity, grsecurity, vsecurity, oss-security*

more often in our dataset. For instance, *null pointer dereference*, *uninitialize/initialize*, *overflow*, *corrupt*, appeared in the top matched keywords. They often appear along with memory-related vulnerabilities, such as null pointer exception [127], uninitialized variables [134], and buffer overflow [135].

Table 4.4 shows the top-matched keywords, we omitted the frequency count for generic keywords, such as *crash*, *check*, and *fail*. We further show our algorithm for keyword filtering in Algorithm 1. Given a single commit, the algorithm will return True if the commit contains keywords that belong to the list. Otherwise, it will return False. As shown in the summarized statistic in Table 4.2, in spite of Linux containing up to 144,601 commits, only 7.4% are included in our final dataset. For the other three projects, our keyword filtering phase also filters their commits to 11% to 18% of their total commits.

Table 4.4: Frequency of Keywords

Keywords	Count	Ratio
uninitialize	4,532	0.111
infinite loop	3221	0.0794
overflow	2,577	0.0635
race-condition	2274	0.0561
out-of-bound	1378	0.0340
null pointer dereference	1,294	0.0319
corrupt	1,228	0.0303
deadlock	805	0.0198
use-after-free	735	0.0181
sanity check	628	0.0154

Algorithm 1 Keyword Filtering Algorithm

```
Input: Commit Message c, Keyword List kws
for word in c do
  if word in kws then
    return True
  end if
end for
return False
```

4.5.3.4 Concerns with Keyword Filtering

To further ensure that our keyword filtering process is effective and sufficient, we validate our process with 238 Linux commit patches from 2016 and 2017. These patches belongs to official Linux CVEs that we curated from NVD in the previous step. They includes standard vulnerability patches, including both explicit and implicit security patches. Our validation process validate that our keywords can cover 89.49% of Linux CVE patches, showing that majority of the security patches can be captured by our keyword filtering process. In addition, we later show that, in Section 4.7.2, that our model can predict implicit security patches with high accuracy.

4.5.3.5 Manual Verification and Labelling

We further design a manual verification process to increase the reliability and integrity of our dataset. We engaged four experienced security researchers (3 Bachelors and 1 Ph.D.), that are hired by our industrial collaborator, to manually label the filtered commits. All security researchers have sufficient experience in reporting their respective CVEs to official databases (e.g., NVD). Each bachelor evaluator has 2 years of security-related experience and the Ph.D. evaluator has 3 years of security-related experience. It is notable that one of our authors is also involved in the manual verification process. It is difficult to distinguish security solely from their commit message, even so as many developers might not clearly indicate their intention of patching a security concern in their messages. We exercise a two steps manual verification process to increase the reliability of our dataset:

1. Initial Verification. Two independent evaluators examine and label the commits into one of the three following categories: Security-Patch, Non-Security Commits, and Unsure.
2. Final Confirmation. If the initial labels of a commit are different or either one of the labels contains an 'unsure" label, the commit will be forwarded to a third researcher for further investigation.

4.5.3.6 Ground Truth and Final Dataset

After our two key data curation steps (Keyword Filtering and Manual Verification), we curated our final dataset of ground truth that consists of both CVE patches and manually labeled security patches, i.e., commits that fix vulnerability. The total manpower costs for the data curation process are around **600 man-hours**, and finally, we gathered a total amount of filtered and verified commits with labels of **40,523**.

We also discovered that Linux has 69.4% commits that are security patches, a much higher rate compared to the other three libraries. The other three libraries, i.e., FFmpeg, Qemu, and Wireshark correspondingly have 45.7%, 43.0%, and 41.5% commits labeled as security patches. As we performed experiments within each project, each project dataset is randomly spilt into training (75%) and testing set (25%) for their respective evaluation. Furthermore, we conduct experiments on the combined dataset, i.e., the dataset that contains commits of all four projects. This complete dataset will also be split into training (75%) and testing set (25%) for training and evaluation of the baseline and models.

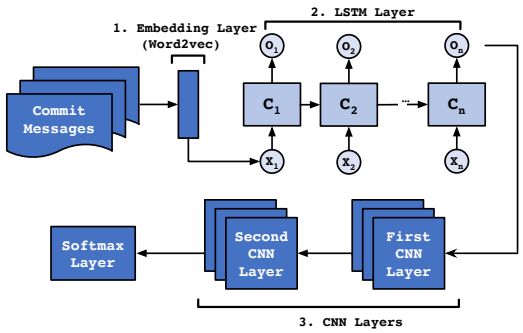


Figure 4.2: Deep neural network for commit messages

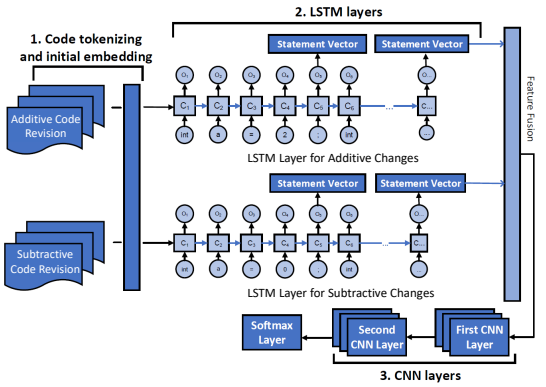


Figure 4.3: Deep neural network for code revisions

4.5.4 Learning from Commit Messages

Fig. 4.2 shows the network architecture for SPI-CM, our deep neural network for learning commit messages. SPI-CM comprises these four layers sequentially: Embedding Layer, LSTM Layer, CNN Layer, and Softmax Layer. Our four layers sequential design in SPI-CM allows us to identify security patches by (1) embeds commit messages into unique numeric vectors that are necessary for input of deep neural network and (2) learning the sequential relationship between tokens to identify features of security patches better. (3) enables a higher level of learning by connecting deep layers of convolutional networks (4) and a softmax layer to predict the likelihood of security patches.

4.5.4.1 Embedding Layer

To feed the commit messages to the LSTM layers for feature learning, we first use an embedding layer to represent the sequence of tokens with numerical vectors. We pre-trained a Word2Vec [84] model with all commit messages. Training a Word2Vec model over a large amount of data is proved to be effective in text classification [49]. The trained word2vec model consists of V embedding vectors, where V is the vocabulary size and each vector with k dimensions is a numerical representation of a word. We encode each message, using word2vec embedding, into a matrix with a dimension (L, k), where L is the maximum length of commit messages. The vocabulary size of our pre-trained word2vec model is $283,146$.

4.5.4.2 LSTM Layer

We employ an LSTM layer to learn and extract a representation of commit messages that can model the relationship between the tokens. The output of the embedding layer, embedded commit message, is passed to the LSTM layer to learn their middle-level semantic features. LSTM is designed to capture long-term dependency over sequences. We employ a single layer of LSTM for easier design and experiments. Let N be the number of units in the LSTM layer, the output of the layer for a single commit message becomes a (L, N) vector.

4.5.4.3 CNN Layer

We further employ CNN to learn higher-level representation. Our experimental result shows that by passing the learned features of LSTM to a dual CNN layer, instead of passing them directly to the softmax layer, our predictors perform better. Hence, we employ a two-layer CNN, each followed by an activation function, ReLU. Our CNN layer outputs a one-dimensional vector, which can be passed to the softmax layer for classification.

4.5.4.4 Softmax Layer

Lastly, we utilize the softmax layer to output the probabilities of a commit being a security patch. A two-unit softmax output layer is used to determine the likelihood that a commit message indicates that the commit fixes a vulnerability, i.e., a security patch.

4.5.5 Learning from Code Revisions

To circumvent introducing lengthy sequences of code and reflect the essential changes, we only consider the removed source code before revision (*i.e.*, the subtractive changes) and appended source code after revision (*i.e.*, the additive changes). We employ a more dedicated neural network, SPI-CR for learning on code revisions. Fig. 4.3 depicts the network architecture. SPI-CR consist of four layers of design that enables us to learn code revision effectively: Embedding Layer, Statement-Level LSTM, CNN Layer, and softmax layer. Similar to SPI-CM, our embedding layer embeds tokens into a numeric vector that is necessary for deep learning. Furthermore, our statement-level LSTM allows us to extract the statement representations through our selection. This lessens the input of the subsequent networks. Lastly, we employ CNN layer for deeper learning and softmax layer to predict the likelihood of the security patches.

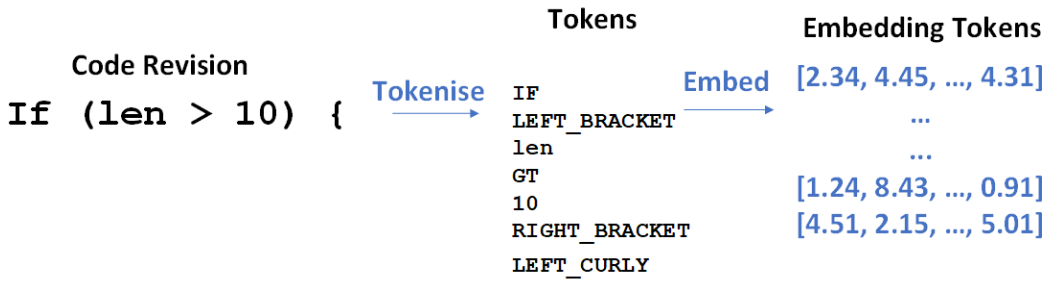


Figure 4.4: Example of code tokenizing and embedding

4.5.5.1 Tokenizing and Code Embedding

We employ an embedding layer that is similar to Section 4.5.4.1. Instead of training on commit messages, we trained a Word2Vec model on all the possible code revisions in our dataset. The code embedding vector is learned by training a word2vec [84] neural networks over the dataset. We split the code revision (both subtractive and additive changes) into individual statements of tokens and keywords before embedding them into their respective vectors. We use Pygments [91] to parse the code into the individual tokens. These tokens consist of variable names, function names, integer literals, logical operators, and programming keywords. Strings and literals are replaced with placeholders to avoid an excessive vocabulary set. For instance, strings, such as IP Address and Port Number, are replaced with a placeholder, $< STRING >$. Finally, the vocabulary size of all code tokens is $V = 111,987$. Fig. 4.4 shows our tokenizing and code embedding process of a single code revision.

4.5.5.2 Statement-Level LSTM Layer

Instead of a complete function in a single file, code revisions are scattered in form of statement changes across different files. For instance, a change in a global variable name can occur in many different files and across different functions. To tackle this dynamic nature of commit, we design a statement-level LSTM layer to learn the semantic meaning of code revision. Our statement-level LSTM layer incorporates two individual LSTM networks to learn on subtractive and additive code changes respectively. We output the representation of each statement in LSTM layers to learn the structural and semantic information of the code revisions. As each code revision is a sequence of tokens, we marked the end of each statement with a unique token End of Statement (EOS) token. During the training, the network learns to generate meaningful representations at the position of the EOS inputs. We simply sample the learned representation

yielded by the LSTM encoder at the time steps that are corresponding to the EOS token. Let M and P be the number of statements in additive and subtractive revision. The results of the LSTM encoders are M representation vectors output by the additive revision encoding LSTM, denoted by $\{v_j, j \in (1, ..., M)\}$, and P representation vectors output by the subtractive revision encoding LSTM, denoted by $\{u_i, i \in (1, ..., P)\}$. The statement representation vectors are then fused into a matrix.

4.5.6 Security Patch Prediction

After the learning phase, we obtained three models accordingly for prediction, SPI-CM, SPI-CR, and SPI.

Security Patch Identifier on Commit Messages (SPI-CM). We use a two-unit softmax layer to classify the learned features from Section 4.5.4. It helps us to justify if the deep learning approach performs better than the traditional machine learning algorithms at practically moderate-size datasets. The previous work [49] encoded the commit messages with a pretrained word2vec model and trained an ensemble learning algorithm that leverages several different basic classifiers to achieve the state-of-the-art result. The comparison with this work helps answer the question.

Security Patch Identifier on Code Revisions (SPI-CR). Similarly, the learned features from Section 4.5.5 are passed to a two-unit softmax output layer. Through this, we investigate if we can extract characteristics of meaningful patches from code revisions.

Security Patch Identifier on commit messages and code-revisions (SPI). We ensemble the results of both SPI-CR and SPI-CM to complement their disadvantages. We further evaluate their performance to see if they perform better.

Fig. 4.1(b) illustrates our prediction phase of a new commit. Given a commit raw features, commit messages, subtractive code changes, and additive code changes are extracted. The commit messages are classified via SPI-CM, while the subtractive and additive code changes are classified by SPI-CR. The upper part of Fig. 4.1 (b) shows the procedure of SPI-CM: 1) The commit message is first cleaned and tokenized into a list of words, 2) Each tokenized word is embedded by a pretrained word2vec model, so that the commit message is represented by a

matrix and suitable for input into the trained SPI-CM model, 3) A trained SPI-CM calculates based on the input embedded matrix and outputs a probability (P1 in the Fig. 4.1(b)). Similarly, we split the additive and subtractive changes of the commit into a sequence of tokens, and further embed them using a trained word2vec that is trained on code revision. We further input the embedded code revisions into the trained SPI-CR model. Lastly, SPI-CR predicts with an output probability P2. We propose an ensemble model through a weighted combination based on P1 and P2 to incorporate both SPI-CM and SPI-CR. The ensemble model, which contains SPI-CM and SPI-CR, is collectively known as SPI.

4.5.7 Evaluation Metric

In this section, we introduce three suitable evaluation metrics, *precision*, *recall*, and *F1-Score* for our experiments. Since we formulate our problem as binary classification, hence, they are effective in evaluating our model and approaches.

4.5.7.1 Precision

Precision measures the proportion of true positives among the predicted positives. It allows us to gauge the number of false positives that our model has predicted. Therefore, a high precision infer that our model performs well in identifying security patches. The formula of precision is shown in Equation 4.5

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{4.5}$$

4.5.7.2 Recall

In contrast, recall measures the total number of true positive over actual positive. If the recall of our model is high, it implies that our model is effective in determining non-security patches. Recall can be computed as shown in Equation 4.6:

$$Precision = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{4.6}$$

4.5.7.3 F1-Score

F1-Score is widely used in binary classification problems and previous works [49, 136] have also employed them. It is computed using a weighted combination of precision and recall. A high F1-score implies the model has a low number of false positives and false negatives. F1-score can be computed using an equation shown below:

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{4.7}$$

4.6 Evaluation

We conduct extensive experiments on our dataset and approach to assess the performance and effectiveness of SPI. We evaluate SPI on the four project datasets and one production dataset that are provided by our industrial collaborator. In summary, we structure our experiments into answering the following research questions (RQs).

1. *What is the performance, in terms of F1-score, recall, and precision, of SPI-CM?* Can SPI-CM learn from domain-specific commit messages, and outperform the state-of-the-art method [49]? (Section 4.6.3)

2. *What is the performance, in terms of F1-score, recall, and precision, of SPI-CR?* Can SPI-CR learn useful representations from code revisions for the identification of security patches? (Section 4.6.4)

3. *What is the performance, in terms of F1-score, recall, and precision, of SPI?* We evaluate SPI and observe if it can perform better than any of the single components. (Section 4.6.5)

4. *What is the performance, in terms of F1-score, recall, and precision, of SPI-CM, SPI-CR, and SPI in cross-project evaluation?* We evaluate SPI and test its generalization ability through a cross-project evaluation. (Section 4.6.6)

5. *Does SPI performs well in a production setting?* We evaluate if SPI performs equally well in an industrial production setting. (Section 4.6.7)

6. *Does word embedding dimensions and LSTM dimensions affect the performance of SPI?* We investigate if these hyperparameters can greatly affect the performance of SPI and explore the best value of these hyperparameters. (Section 4.6.8)

4.6.1 Implementation Details

We implemented SPI using Tensorflow [137], and Python 3.6. We conducted our experiments on a server with 36 2.30GHz Intel Xeon processors with an RTX 2080. All experiments are conducted with the patience of 200 epochs, Adaptive Moment Estimation (Adam) optimizer [97], batch size of 64, a learning rate of $=1e^{-4}$ and dropout of 0.2, word and code embedding dimension of 300. For SPI-CM and SPI-CR LSTM layer, we use 1 layer of LSTM and 64 LSTM units. We refer the reader to RQ6 for more hyperparameter tuning of word dimension and LSTM unit. For CNN layers, we use a two-layer CNN and max-pooling for all experiments. For SPI-CM, we have the first CNN with 64 filter, (3×64) kernel size, (1×1) stride, followed by a max-pooling with (2×2) pool size and (2) stride, and the second CNN with $32)$ filter, (3×64) kernel size, (1×1) stride, and a max-pooling with (2×2) pool size and (2) stride. For SPI-CR, both CNNs have (32) filter, (3×64) kernel size, and (2) stride.

We trim and pad the input sequence to be length of 100. If the length of the input sequence is more than 100, we trim the length of the sequence to 100. On the contrary, if the length of the input sequence is less than 100, we pad the sequences with zero until the length reaches 100. For the statement-level LSTM, we set the max number of statements to be the average number of statements in our dataset, i.e., 10.

4.6.2 Baselines

The most related works are VCCFinder [138] and one of the works by Zhou et al. [49] that specifically find vulnerability-introducing and vulnerability-fix commits, i.e, security patches. VCCFinder [138] utilized SVM while Zhou's work [49] designed a *K*-fold stacking algorithm that ensembles 6 different conventional classifiers for better performance and implemented in production. In our selection of baseline, we only consider works that identify security patches (e.g., Zhou et al. [49])

Consequently, we directly compare with the *K*-fold stacking algorithm (abbrev. *Kfs*). Furthermore, we compare SPI-CM and SPI-CR with *Kfs* baseline and base LSTM model. Code

Table 4.5: Evaluation results on Linux, FFmpeg, Qemu, and Wireshark datasets

Method	Linux			FFmpeg			Qemu			Wireshark		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
*K*fs (msg)-RQ1	0.8135	0.9200	**0.8635**	0.8729	0.8119	0.8413	0.7734	0.6422	0.7017	0.7932	0.6900	0.7380
LSTM (msg)-RQ1	0.8107	0.9171	0.8606	0.8867	0.9037	0.8951	0.8289	0.8041	0.8163	0.8091	0.8337	0.8212
SPI-CM-RQ1	0.8174	0.9066	0.8597	0.9266	0.9187	**0.9226**	0.8137	0.8277	**0.8206**	0.8174	0.8750	**0.8452**
*K*fs (code)-RQ2	0.7396	0.9479	**0.8309**	0.6597	0.4805	0.5560	0.6305	0.4009	0.4901	0.5950	0.3392	0.4321
LSTM (code)-RQ2	0.7966	0.7088	0.7502	0.5560	0.5267	0.5409	0.4896	0.5766	0.5295	0.4969	0.4951	**0.4960**
LSTM (LineReps)-RQ2	0.7598	0.7457	0.7527	0.5135	0.6343	0.5676	0.4319	0.4977	0.4625	0.4788	0.4538	0.4660
SPI-CR-RQ2	0.7720	0.7125	0.7410	0.5126	0.6764	**0.5832**	0.4992	0.6137	**0.5506**	0.4807	0.4842	0.4824
SPI-RQ3	0.8788	0.9298	**0.9036**	0.9652	0.9501	**0.9576**	0.9440	0.9211	**0.9324**	0.9016	0.8975	**0.8995**

Table 4.6: Evaluation results on Combined datasets

Method	*Combined*		
	Precision	Recall	F1
*K*fs (msg)	0.7988	0.8002	0.7995
LSTM (msg)	0.8403	0.9025	0.8703
SPI-CM	0.8685	0.8866	**0.8774**
*K*fs (code)	0.6681	0.6586	0.6633
LSTM (code)	0.5959	0.7504	0.6643
LSTM (LineReps)	0.5488	0.8513	0.6674
SPI-CR	0.5581	0.8656	**0.6787**
SPI	0.8624	0.8968	**0.8793**

revisions commonly consist of statement-level changes, hence, we test our performance of the model on statement-level without CNN to test the performance over basic LSTM. Specifically, we conduct an additional evaluation for SPI-CR to evaluate if the statement-level model, LSTM (LineReps), performs better than basic LSTM. We implemented the baselines and tuned them to the best parameters.

4.6.3 Performance of SPI-CM (RQ1)

We compare our commit message neural network, SPI-CM, with *K*-fold stacking algorithm, *i.e*, *K*fs (msg), and LSTM network, *i.e*, LSTM (msg). Table 4.5 and Table 4.6 present the three metric scores for learning on commit messages for the dataset on four projects and combined dataset respectively. Overall, it shows that SPI-CM outperforms the *K*-fold stacking algorithm and the basic LSTM network.

Compared with the K-fold stacking, the increment of the F1-score in SPI-CM ranges from **7.7%** to **11%** in all experiments, excluding Linux. We noticed that the F1-score for Kfs (msg) in Linux performs slightly better than SPI-CM, by **0.38%**. One possible explanation is that Linux has the most imbalanced data and hence, could affect the results on Kfs (msg). Compared with the LSTM network, the F1-score improves **2.7%** in FFmpeg, **0.43%** in Qemu, **2.4%** in Wireshark and **0.7%** in the combined dataset.

Our experiments prove the effectiveness of CNNs in our design to obtain higher-level and effective features. We also observe that the performance on the combined dataset is better than that of a single project, indicating that a large dataset can boost the performance of our model.

Answer to RQ1: Our proposed deep neural network SPI-CM proved advantages over the comparing baselines. With an additional CNN, SPI-CM can increase up to **2.75%** over the basic LSTM model.

4.6.4 Performance of SPI-CR (RQ2)

Similarly, we evaluate our SPI-CR with a K-fold stacking algorithm, Kfs (code), and basic LSTM model, LSTM (code). Furthermore, we added one evaluation against a statement-level LSTM without CNN, LSTM (LineReps). Table 4.5 and Table 4.6 shows the results on code revisions.

Our code revision model, SPI-CR, outperforms the Kfs (code) for FFmpeg, Qemu and the combined dataset, in terms of F1-Score, by **2.72%**, **6.05%** and **1.54%** respectively. Furthermore, SPI-CR also outperforms both LSTM (code) and LSTM (LineReps), in terms of F1-Score, by at least **11.3%** in these three projects. However, we observed that there are two exceptions to this evaluation. Firstly, Kfs (code) in Linux outperforms all other models. This aligns with our reasoning in the previous RQ as the data of the Linux dataset are imbalanced and, hence, Kfs (code) could predict more positive cases than negative, resulting in a higher F1-score. Secondly, for Wireshark, LSTM (code) outperforms SPI-CR by **1.36%** in terms of F1-score. Wireshark has a higher average line of code at 29.52, hence, possibly resulting in statement-level models do not perform as well as expected.

Note. We observed that results on commit messages are better, possibly due to that code revisions tend to be longer sequences and contain more complicated structure and logic. Partly

due to our network design where we drop all tokens after the first M tokens where M is the maximum length of tokens. This causes us to miss context on the code revisions when learning their representation. This is one of the limitations of sequential data and sequential-based neural networks. If we take a longer sequence, however, it will go beyond the capability of contemporary neural networks, which causes inferior performance in the other way.

Answer to RQ2: Our proposed neural network for code revision, SPI-CR, learns meaningful representations for identifying security patches. Although the performance of SPI-CR is comparatively lacking to SPI-CM, it outperforms the K-fold stacking method and basic LSTM models, showing its effectiveness when learning solely on source code.

4.6.5 Performance of SPI (RQ3)

We scrutinize the combined performance of the learned results from SPI-CM and SPI-CR. Table 4.5 and Table 4.6 shows the performance of SPI in the two experiment setting. We observed that weighted combination provides a great improvement in F1-score, up to **11.18%** for Qemu dataset, when compared to SPI-CM and SPI-CR. Our trained models on the combined dataset achieved **89.68%** in recall as well as precision of **86.24%**, and the highest F1-score of **87.93%**. This shows that the combination of features in code revision and commit messages can help identify security patches better.

Answer to RQ3: Despite the lacking performance in SPI-CR, SPI achieves better results in combining the features of commit messages and code revision. With an increment as high as 11.18% in F1-score, SPI effectively identifies undisclosed security patches based on the commit message and code revision.

4.6.6 Performance of SPI in Cross-Project Evaluation (RQ4)

As shown in the previous RQs and experiments, SPI effectively identifies security patches when the training and testing data belong to the same open-source projects. We further evaluate the generalization ability of SPI by conducting a cross-project evaluation, i.e., evaluating whether SPI can effectively identify security patches from projects that are different from the training dataset. Table 4.7 shows the result of cross-project evaluation. We isolate one of the projects as a testing dataset and used the remaining three projects as the training dataset. For example,

Table 4.7: Evaluation results on Cross Projects Evaluation

Method	LinuxTest			FFMpegTest			QemuTest			WiresharkTest		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
*K*fs (msg)	0.7657	0.7953	0.7802	0.7319	0.8146	0.7710	0.6696	0.6797	0.6746	0.7395	0.6579	0.6963
LSTM (msg)	0.7048	0.9867	**0.8222**	0.7834	0.8910	0.8337	0.6886	0.7255	0.7066	0.7987	0.8292	0.8137
SPI-CM	0.7109	0.9719	0.8211	0.8329	0.9146	**0.8718**	0.7008	0.7606	**0.7295**	0.8225	0.8063	**0.8143**
*K*fs (code)	0.7918	0.2943	0.4292	0.4988	0.6957	0.5810	0.4702	0.7248	0.5704	0.5064	0.4256	0.4625
LSTM (code)	0.7301	0.8672	0.7927	0.4844	0.9142	0.6333	0.4394	0.9630	0.6035	0.4208	0.9684	0.5867
LSTM (LineReps)	0.7213	0.8581	0.7838	0.4753	0.9383	0.6310	0.4507	0.9277	**0.6067**	0.4272	0.9160	0.5827
SPI-CR	0.7017	0.9902	**0.8213**	0.4721	0.9636	**0.6337**	0.4434	0.9585	0.6063	0.4214	0.9763	**0.588**7
SPI	0.7108	0.9814	**0.8245**	0.8304	0.9126	**0.8733**	0.6930	0.7714	**0.7301**	0.7933	0.8477	**0.8196**

in LinuxTest, we used Linux as the testing dataset, while we trained our model on FFmpeg, Qemu, and Wireshark.

SPI performs well under the setting of cross-project evaluation, achieving an F1-Score of 82.45%, 87.33%, 73.01%, and 81.96% for LinuxTest, FFMpegTest, QemuTest, and WiresharkTest respectively. On average, SPI performs better than *K*fs(msg) by 8.13% and *K*fs(code) by 30.11%. Comparing to LSTM(msg) and LSTM(LineReps), F1-Score increased, on average, by 1.78% and 15.80% across four experiments.

We observed that, despite a decrease in performance as compared to inter-project evaluation, SPI-CM achieves F1-score between 72.95-87.18% and SPI-CR achieves an F1-score in a range of 60.63-82.13%, while SPI achieves F1-Score between 73.01-87.33%. As compared to RQ3, the performance of SPI-CM and SPI-CR across four experiments decreases by 7.03% and increases by 7.32% on average. However, the overall performance of SPI decreases by 11.14% across all four experiments.

The decline in the performance of SPI is inevitable since we evaluate the model using different projects. However, SPI still can perform well, in the range of 73.01-86.96% in F1-Score. This demonstrates the ability to identify security patches from different domains despite the absence of the training dataset. Hence, we conclude that SPI can identify cross-domain security patches even with the lack of project data in the training set.

Answer to RQ4: Wide range of projects should be incorporated into the dataset to allow the performance of the SPI to generalize better to unseen security patches. Our model is shown to generalize well to unseen projects and identify security patches from projects that are not in the dataset.

4.6.7 Production Observation with SPI (RQ5)

We discuss the effectiveness and practicality of our model in a production setting, hence, we deployed our pipeline in a production setting of our industry collaborator. Our production dataset is crawled from 410 C Language open-source libraries and consists of 298,917 commits after the filtering process. The production dataset exhibits distinct data distributions and features, such as variable names that are unique to projects.

Production Prediction. We employ the same model that was trained using the combined dataset to predict the commits in the production dataset. We employed the same experiment setting, i.e., word embedding model, neural networks structure, and hyperparameters as in RQ1-3. Out of 298,917 commits in the production dataset, 136,466 (45.65%) were predicted as security patches and 162,451 (54.34%) as non-security commits.

Prediction Verification. We manually verified 1146 commits that were picked randomly from the predicted non-security commits. We discover that **93.63%** of the predictions were predicted correctly. The high precision implies that non-security commits are effectively filtered out from an unlabeled dataset. This implies that we can effectively confirm **151,080** out of the 162,451 predicted non-security commits. Hence, it reduces the workload for the manual verification process. To date, 43,543 out of the 136,466 predicted security patches had been verified manually, with an initial precision of **61.49%**. We further manually verified 1000 random commits from the production dataset and observed that there are 384 security patches among these commits, i.e., 38.4% of 1000 verified commits are security patches. As shown in Table 4.2, the four selected projects have security patches in ranges of 41% to 69%, hence, our sampling from the industrial dataset shows that security patches are even more scarce in the wild. This discovery highlights that our work is important and the gap in proportion between the production dataset and four selected projects dataset is caused by the diversity of the industrial datasets, which includes 410 C Language open-source libraries while the experimental datasets consist of only four projects. This explains why the F1-Score of production setting is lower than the test results, which is possibly caused by the distribution of the SP ratio and the diversity of the two datasets.

Iterative Model Training. It is challenging to get a sizable and high-quality labeled dataset at one time to train well-generalized models. In our case, the initial trained model has helped

significantly accelerate manual validation of new data, where the labeled new data can be fed back into the learning process. By iterating over training and manual verification, it forms a closed-loop for product development and improves performance gradually. The initial production test proves the usability and effectiveness of our system. Building on this success, we iterated and retrained the model with a dataset that is a combination of training dataset and production dataset. We then obtained an improved model with a precision of **77.99%**. This shows that the model can be improved iteratively in a production setting.

Answer to RQ5: The deployment of SPI in production validated its usability and effectiveness. Instead of manually verifying 298,917 commits, SPI reduces the amount of verifying commits by 50.54% by fulfilling as a pre-processing step. It saves manual workload effectively in building a security patch database and drives to build more robust and general prediction models with data and product iteration.

4.6.8 Hyperparameter Tuning (RQ6)

We explore two hyper-parameters tuning, word embedding dimensions and LSTM dimension, in this RQ. As previously mentioned, we employ word2vec [84] as our method for embedding textual data into their numerical representation. The dimension of the vector representation affects the performance of the neural network. If the dimension of the vector representation is too large, the training process might slow down significantly, failing to converge properly. On the other hand, if the dimension of the vector representation is too small, the information of the textual data cannot be represented effectively.

The dimensions of LSTM refer to the number of learned parameters in each LSTM cell. Similarly, it affects the learning capability of the model. If the dimension of LSTM is too small or too large, the performance of SPI might not be optimal. Therefore, we experiment on different values of these two hyperparameters.

Figure 4.5 shows the performance of SPI in different parameter settings. Figure 4.5(a) shows the result of different word embedding dimension, while Figure 4.5(b) shows the experiment results of different LSTM dimensions.

As observed from Figure 4.5(a), the performance of SPI increases proportionately as the dimensions of the word embedding increase. However, it decreases slightly after the dimension

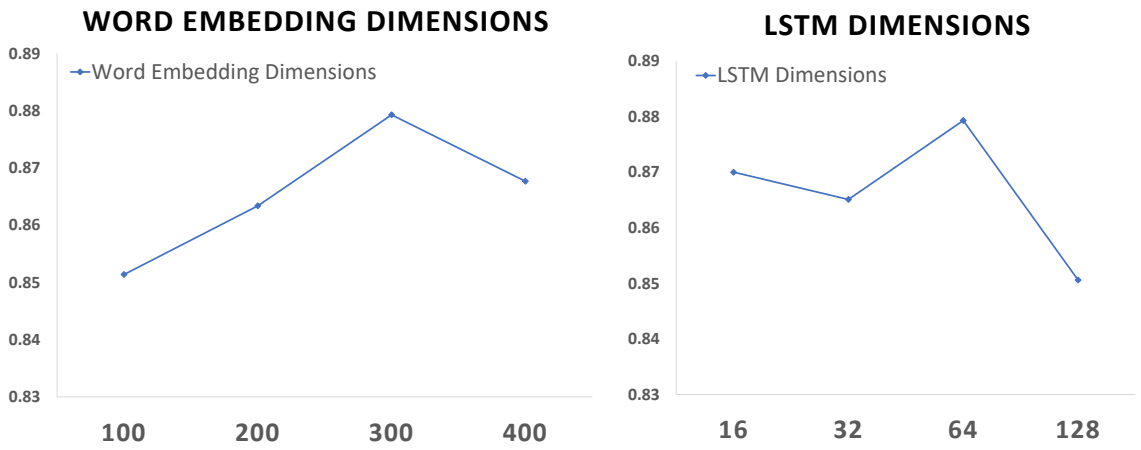


Figure 4.5: Hyper-Parameter Tuning

reaches 300. These experimental results are similar even in the case of the LSTM dimension. We can observe from our experiments that the performance of SPI shows an upward trend despite that the trend is non-monotonic. However, we discovered that our performance of SPI peaks when we select the value of the LSTM unit to be 64 and the value of word dimension to be 300.

Answer to RQ6: Deep neural networks are sensitive to hyperparameters as they affect their learning capability. We show the hyperparameter tuning on two main hyper-parameters: word embedding dimension and LSTM dimension, for exploration of the best values. Our experiments show that the performance of SPI peaks when the word embedding dimension is 300 and the LSTM unit is 64. We further show that the parameters that are used in the previous RQs allow us to achieve the best performance in SPI.

4.7 Discussion

In this section, we discuss two areas of SPI: Impact of Sequence Length on SPI and Prediction on Implicit and Explicit Patches. We aim to investigate if the performance of SPI is affected by the sequence length, i.e., length of code revision and commit messages. Furthermore, we examine if SPI can identify implicit and explicit security patches by qualitative analysis.

Table 4.8: Performance of SPI for different message length

Msg Length (l)	Total	SP	NSP	F1
l <50	5609	2558	3051	**0.9217**
50≤ l <100	2593	1385	1208	0.8435
100≤ l <150	936	520	416	0.8267
l >150	993	635	358	0.8085

Table 4.9: Performance of SPI for different code length

Code Length (l)	Total	SP	NSP	F1
l <50	3643	2052	1591	**0.8976**
50≤ l <100	1995	1069	926	0.8943
100≤ l <150	1072	527	545	0.8761
l >150	3421	1450	1971	0.8315

4.7.1 Impact of Sequence Length on SPI

In this discussion, we investigate the performance of SPI on various commit messages and code-revision length. Table 4.8 and Table 4.9 shows the results of SPI when we varying the length of sequence in the interval of 50. SPI can predict accurately on message with length less than 50 with an f1-score of **92.17%** and code with length less than 50 with an f1-score of **89.76%**. One of the possible reasons for better performance among shorter code revision and commit messages might be that security patches are often small and quick fixes to the code-base. Commonly, exploits and loopholes are patched with additional checkings, such as sanity checks. Furthermore, longer code revisions are more difficult to learn as they are complex and often span across different parts of the file.

4.7.2 Prediction on Implicit Security Patches

We are motivated to investigate the effectiveness of learning from code revisions alone on identifying security patches. We define implicit security patches as patches that fix a vulnerability but their commit messages do not convey their intention of fixing the vulnerability. We use official security patches of CVEs that are verified by vendors and security professionals to avoid

Table 4.10: Evaluation on the fix commit of CVE-2017-7187

Message	scsi: sg: check length passed to SG_NEXT_CMD_LEN The user can control the size of the next command passed along, but the value passed to the ioctl isn't checked against the usable max command size.
Code revision	+ *if* $(val > SG\_MAX\_CDB\_SIZE)$ + *return* $-ENOMEM$;
SPI-CM	0
SPI-CR	1
SPI	1

disputation in this discussion. We use four CVEs and their security patches for this evaluation of SPI.

CVE-2017-7187 [139]. This is a high-risk Denial of Service (DoS) from Linux Kernel, caused by a large-size command from user data. The commit message and code revision are shown in Table 4.10. Despite the message mentioning checking the length of SG_NEXT_C̃MD_LEN because a user can control the size of the command, it is unclear from the message if it is a patch. By looking at the code revision, it returned a token, -ENOMEM, if the input size is larger than the allowed size, indicating that before the fix the user can manipulate much larger size data in memory.

CVE-2017-8063 [140]. This high-risk DoS vulnerability was disclosed in the Openwall community [141] by pointing out that it was silently fixed by the Linux Kernel developers without mention of it at all in the commit message.

Similar examples can be found in CVE-2010-5329 [142] and CVE-2015-8952 [143] where commit messages of the patches do not mention that they were meant to fix vulnerabilities. Table 4.10 shows the predicted results, commit message and change revisions of CVE-2017-7187. The commit message has no security-related words shown in Table 4.3 or indication of fixing any vulnerability, thus SPI-CM predicted it as negative. On the other hand, as we analyzed previously, the code revision could be an indication for fixing a vulnerability, which is predicted as a security patch by SPI-CR. The combined result by SPI is positive as the final prediction. Out of the four CVE patches, we achieve about 75% accuracy, where patches of CVE-2017-7187,

Table 4.11: Evaluation on FFmpeg commit

Commit ID	61cd19b8bc32185c8caf64d89d1b0909877a0707
Message	vmnc: Port to bytestream2 Fix some buffer overreads.
SPI-CM	1
SPI-CR	0
SPI	1

CVE-2017-8063, and CVE-2010-5329 are predicted correctly as security patches though the message predictor failed. Only CVE-2015-8952 was predicted wrongly because of the long code revision in the patch.

As seen from these implicit security patches, our code-revision model provides meaningful insights to decide whether the commit is a security patch. The commit messages might not be able to provide sufficient information in deducing the type of commits. The code revision model allows us to draw insights on the commits based on the code revision. Therefore, the evaluation proves that the code revisions models are essential in identifying security patches.

4.7.3 Prediction on Explicit Security Patches

In this section, we discuss the performance of SPI on explicit security patches, i.e., security patch where the commit message clearly indicates a fix. We take a commit (61cd19) from FFMpeg as our example [144]. We perform predictions using three of our trained models, SPI-CR, SPI-CM, and SPI, to evaluate the different components. Table 4.11 shows the details of the commits and our prediction results. The code revision is omitted from the table due to the massive revision. hence, please refer to references [144] for more information. The shown example has a short message with evidence of fixing some buffer overreads. Buffer overread [145] is a common, yet serious, vulnerability that could lead to more severe consequences, such as privilege escalation and denial of service.

On the other hand, the code revisions are long and span over several different parts of the functions. It consists of changes of several function calls and complex equations. This prevents the neural networks to learn that it is a security patch. The prediction results of SPI-CM and SPI

show that our model can also effectively identify such security patches without any additional information.

4.8 Threats to Validity and Limitations

4.8.1 Threats to Data Quality

Our security researchers are experienced enough to be able to determine security patches. Furthermore, our dataset went through a stringent and rigid verification process, hence, we believe that our manual labels are fair.

4.8.1.1 Limitations of Dataset

Our observation in the data curation process has indicated that implicit security patches are often lacking in numbers. Furthermore, our dataset might contain false negatives due to human errors in the manual verification process. To increase the data quality and further increase the reliability of our dataset, we propose several measures: (1) We can collect ground truth from different security-related sources, i.e., robust ground truth, to ensure our model learnt from absolute patches. (2) We can employ deep and sophisticated neural models in code revision, such as Code2Vec [13] or Graph Neural Network [146], to better learn the semantic of the code revision. With a better learnt representation, the source code can convey its functionalities better, resulting in a clearer intention of patching in code revision.

4.8.2 Validity Threats on Language Generalization

Our approach does not use any programming language-specific techniques. Hence, it can be applied to other programming languages as well. Although some project-specific keywords might not work well in other projects, the list of keywords can be extended or iteratively enhanced to achieve better performance. Therefore, we argue that it is feasible to employ our approaches on other programming languages.

4.8.3 Limitations on Code Revision Learning

As observed in most of our RQs, the performance on commit messages is higher than when predicting solely on code revision. This could be due to multiple reasons: (1) Our dataset

is collected based on keywords and human comprehension of commit messages. Therefore, commit messages are more dominating and easier to learn when compared to code revision. To tackle this limitation, our future work can consider a better curation technique, where only code revisions are shown and commit messages are hidden from the security evaluator. This can alleviate our current limitations. (2) Subtractive and additive code changes have a minor difference, for instance, there could be only one token difference between the subtractive and additive code changes. Neural networks have difficulty in learning such minor changes, and their resulting learned representation can be very similar. (3) As mentioned previously, source code commonly has complicated structures and sequences of tokens cannot represent them sufficiently [13]. We can employ deep and complex representations, such as tree-based representation [12, 147] or graph-based representation [14, 31], for better performance and deeper program comprehension.

4.9 Conclusion

In this chapter, we propose our approach to enhancing and curating security patches through deep neural networks. Our neural network design employs several key features, such as commit message and code revision, in identifying and predicting security patches in open-source projects. By curating a high-quality set of underlying security patches, we can contribute to the security knowledge base. Our experimental result shows that our model, SPI, works well in both our experimental settings and in a production setting, displaying the practicality and feasibility of our work. By building future work on this approach and security dataset, we can facilitate better software security and maintenance.

Chapter 5

Ratchet: Retrieval Augmented Transformer for Program Repair

5.1 Introduction

Resolving bugs and errors is crucial in software development and maintenance. They are commonly introduced by developers unknowingly and often discovered after rigid reviewing processes or with external tools, such as the code review process or through static code analyser. As reported by LaToza et al. [5], developers spent 50% of their time in fixing and discovering bugs. Furthermore, the number of software defects has increased rapidly throughout the recent years [148]. These software defects are introduced by software developers with no intention. A rigid and stringent vulnerability and patch management process must be in place for discovering and fixing these bugs.

Despite the number of open-source projects increasing rapidly due to the open-source initiatives, there exists a large number of silent and hidden defects which has not been officially announced or exploited [148]. Hence, research has been exploring automated fault localization and repair techniques to alleviate the burden of manual debugging and program repairs. However, the problem still has not been resolved yet.

Identifying the location of bugs and fixing the located bugs can be time-consuming. Automated Program Repair (APR) aims to reduce the time and effort in resolving bugs during software developments. The main objective of APR is to generate patches and fixes for localized buggy

segments of source code. They often follow a two-step process: Fault Localization (localizing bugs) and Patch Generation (generating patches for localized bugs).

Generally, APR works often employ Spectrum-based Fault Localization (SBFL) techniques to locate bugs in a program during the fault localization phase. There are many variants of SBFL techniques, such as Ochiai [149] and Tarantula [150], that localize faults through exploring statements that cause failure or success of the corresponding test cases. These techniques compute a suspicious score based on the frequency of success and failure rates. SBFL techniques are heavily dependent on test suites as their suspicious score computation relies on the number of successful and failed test cases. Commonly, the statement with the highest suspicious score will be deemed as the buggy statement. After the buggy statement is localized, patches are generated automatically through various program synthesis techniques. The approaches in generating patches can be broadly categorized in three areas: (1) Heuristic-based Approaches, (2) Constraint-based Approaches, and (3) Learning-based Approaches.

5.1.1 Heuristic-Based APR

Heuristic-based approaches [23, 56, 57, 59] conduct program modifications through mutations of pre-defined operators. They iterate over a search space of these operators in searching for a mutated program that can successfully pass the test suites. Several works aim at employing prioritization in different mutations [57] or candidate patches [59] to find better ways of fixing the program. However, heuristic-based approaches suffer from search space explosion (i.e., producing far more patch candidates) as the size of the search space are directly proportional to the amount of mutation operators [57, 151]

5.1.2 Constraint-Based APR

Many works [61–63] employ constraint solving and program synthesis to generate patches that can fulfill pre-defined constraints. They commonly require execution paths to generate and infer constraints, which requires a large number of resources in the compilation of the target programs [151]. Furthermore, inadequate constraints increase the number of plausible patches which further hinders the validation process [65].

5.1.3 Learning-Based APR

In recent years, learning-based APR has performed outstandingly on generating patches through a data-driven and deep learning approach. These techniques often generate fewer patch candidates with fewer time costs, as compared to heuristic-based and constraint-based APR techniques. Several prominent works [24, 58, 66, 152] explore Neural Machine Translation (NMT) techniques (e.g., seq2seq [152] and attention [87]) in generating patches by learning on historical correct patches. Much deep learning works [58, 66, 152] assume a perfect fault localization scenario where buggy statements are already located [58], while other deep-learning works [121] employ SBFL techniques, similar to Generate-And-Validate approaches. These fault localization techniques are not feasible in real-world projects, where test cases and perfect fault localization are scarce.

The learning-based approaches [24, 58, 66, 121, 152–154] take as input the buggy function [152, 153] or buggy statements [58, 66, 154] for the encoder, and generate the patched statements or functions through the decoder. Various contexts have been explored as an additional feature to the encoder, in hopes to increase the quality of the generated patches. Lutellier et al. [58] and Jiang et al. [66] proposed to employ surrounding lines of localized bugs as its contexts.

They input these contexts into a separated encoder to perform contextual learning. Chen et al. [152] further propose to employ the entire function of the bug as its context and feed it into the encoder for patch inferences. These context learning approaches achieved promising performance, however, their underlying neural networks are simple and do not make full use of the contexts. Incorporating contexts for bug-related feature learning could introduce noises and hinder the learning process. Generating fixes for completed buggy functions can result in lower performance and hinder the learning of patch generation [152, 153].

5.1.4 Fault Localization

Although fault localization is commonly the first step of APR, some works [58, 66, 152] are evaluated with a perfect fault localization assumption [155]. Perfect fault localization refers to the buggy statement that has already been localized by a fault localization oracle. This ensures that their techniques ignore the impact of the imperfect fault localization tools. Some works further employ neural networks to localize buggy statements in the buggy program. DrRepair [24]

utilizes compiler messages to locate buggy statements and aids in generating better patches. However, their approach did not provide any hints for localizing functional and semantic bugs.

Spectrum-based and mutation-based fault localization techniques, such as Gzoltar [156], DStar [157], MUSE [158] and Metallaxis [159], provide practical usage on finding bug position, nevertheless, the accuracy of such techniques is still lacking [155]. Furthermore, they rely highly on bug-triggering test suites which are not always available in real-world settings [160]. Most importantly, the accuracy of fault localization methods could impact the quality of patches generated by these APR techniques [65].

5.2 Motivation

The redundancy assumption of program repair [161, 162] assumes that buggy codes occur multiple times in a big codebase and patches can be generated for these recurring bugs. In this era of "big code", retrieving similar artifacts for code intelligent tasks (e.g., commit message generation [7, 163] and source code summarization [164]) has been a success as duplicate codes are commonly found [85]. Building upon the success of retrieval techniques in code intelligent tasks and the redundancy assumption, we propose to augment the transformer model with code retrieval techniques to boost the fault localization and patch generation for program repairs. Specifically, we propose our dual deep learning-based program repair tool, Ratchet[1], with two different neural networks: Ratchet Fault Localization model (RatchetFL) and Ratchet Patch Generation model (RatchetPG).

Our model, RatchetFL, formulates fault localization as a classification problem that predicts the buggy statement via a Bi-Directional Long Short-Term Memory (BiLSTM) network, without any software artifacts (e.g., bug-triggering test cases or bug reports). RatchetPG employs our proposed retrieval augmented transformer networks to generate patches for the localized buggy statement. Motivated by the recent success [164] of employing retrieval information in code intelligent tasks, we propose to integrate additional contexts to the localized bugs to aid in patch generation. Specifically, we incorporate buggy statements with the closest retrieved patches, that are collected from the historical patches, via the retrieval augmented layer in the transformer

[1]Ratchet is a Chief Medical Officer of Autobot. It fixes robots that are a metaphor for our tool repairing programs.

to enhance our patch generation process. Our experimental results show that RatchetFL can outperform the fault localization baselines by up to 3.98% in Accuracy@Top1, while Ratchet can outperform APR baselines by 11.96% in Repair Accuracy.

5.3 Ratchet: Automated Program Repair Tool

5.3.1 Main Contributions

Our main contribution is our proposed dual neural networks that incorporate fault localization and patch generation, RatchetFL and RatchetPG. RatchetFL localizes bugs in a buggy statement, while RatchetPG employs our retrieval-augmented transformer to aid in patch generations. Our experimental results show that our proposed models can perform better in both tasks,

We like to highlight the following contribution in this chapter:

1. RatchetFL, a BiLSTM-based fault localization model that does not require additional artifacts, such as bug-triggering test cases or bug reports. Our experiments show that RatchetFL can identify buggy statements in the given buggy functions, which outperforms the state-of-the-art approaches in Acc@Top1, Acc@Top3, and Acc@Top5 metrics at 39.8-96.4% for both comparing datasets.

2. RatchetPG, a novel Retrieval Augmented Transformer model with the retrieval-augmented layer to integrate the closest retrieved patches with the buggy statements to generate correct patches. With the closest retrieved patches, RatchetPG can generate correct patches for more (16-236) bugs.

3. Ratchet, a dual deep learning-based program repair tool that integrates RatchetFL and RatchetPG. Our experiments show that Ratchet outperforms state-of-the-art learning-based APR tools on fixing both in-the-wild bugs with 19.5% repair accuracy and in-the-lab bugs with 46.4% repair accuracy. Furthermore, Ratchet is impervious to different fault localization settings when generating correct patches.

4. RatchetDS, a curated bug-patch pair dataset with 56,974 cases collected from 13 popular open-sourced C/C++ projects, of which bugs and patches are collected systematically.

5.3.2 Problem Formulation

We formalize the two sequential tasks: Fault Localization (RatchetFL) and Patch Generation (RatchetPG). Given a buggy function c where $c = \{s_1, s_2, ..., s_n\}$ and n is the total number of lines of c, the objective of RatchetFL is to find the buggy statement s_l by predicting the line number l through localization function f_{loc}:

$$l = f_{loc}(c) \tag{5.1}$$

where $l \in [1, n]$ is the line number of the buggy statement. After the buggy statement s_l is localized, RatchetPG aims to generate the patch statement s_p via a generation function f_{gen} where it is defined as follows:

$$s_p = f_{gen}(s_l) \tag{5.2}$$

Both function, f_{loc} and f_{gen} can be approximated through neural networks, RatchetFL and RatchetPG respectively.

5.4 Approach

We illustrate the workflow of Ratchet in Figure 5.1. It mainly consists of three phases: (1) Training RatchetFL model for finding and localizing which statement is bugged, (2) Training RatchetPG for generating candidate patches for the localized faulty statement. (3) Inference Phase, when given a buggy function, Ratchet first employ RatchetFL in localizing the bug position. Subsequently, we employ RatchetPG to generate candidate patches for the bug.

As depicted in Figure 5.1, our fault localization model RatchetFL and patch generation model RatchetPG of Ratchet is trained with buggy statements and patched statements from a well-curated dataset of open-sourced patches. Formally, given a dataset $D = \{f = (c, l, p) | c \in C, l \in L_c, p \in P_c\}$ where c is the buggy function in the dataset C, Ratchet first employ our fault localization model RatchetFL to locate the position of the buggy statement. This is done by predicting the line number l of the buggy statement s_l in the buggy function c. After the buggy statement s_l has been located by RatchetFL, Ratchet subsequently employs RatchetPG to generate candidate patches for the buggy statement. Our RatchetPG design are inspired

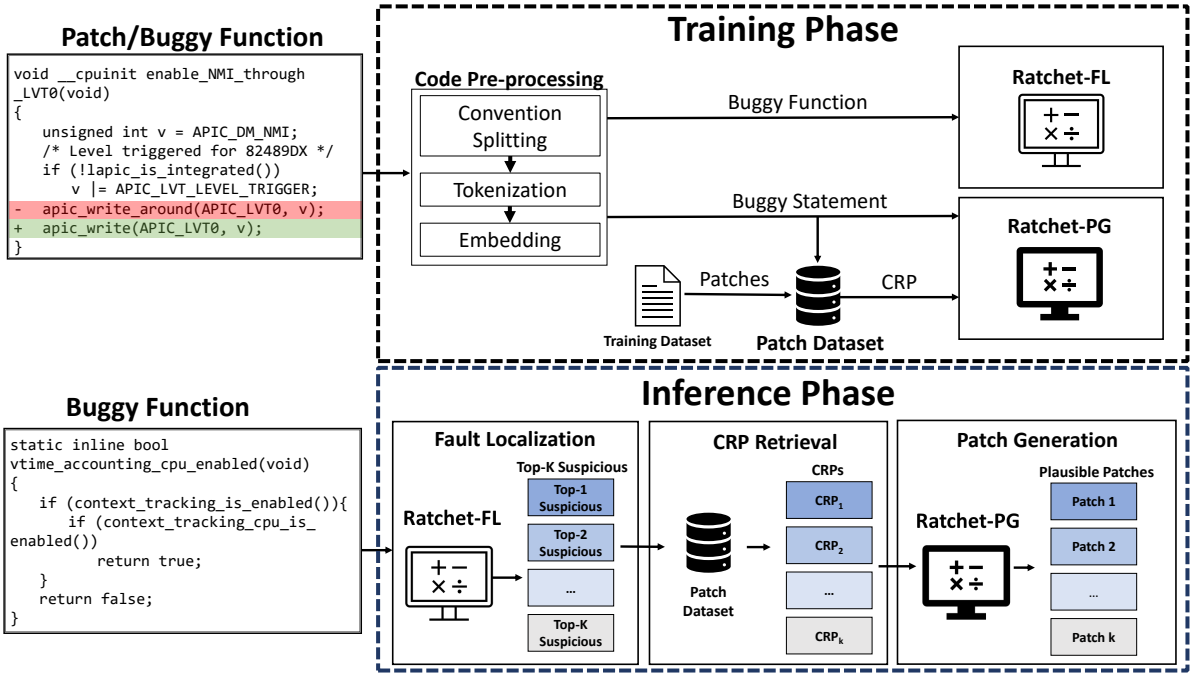


Figure 5.1: Overview of Ratchet.

by recent works in information retrieval [7, 163, 164]. The key idea behind RatchetPG is to augment the buggy statement s_l by retrieving the most similar patch from a patch dataset D' to enhance the patch generation process.

5.4.1 RatchetFL - Fault Localization Model

Inspired by the redundancy assumption [161, 165] that the bugs are widely recurrent in programs, we design our fault localization neural network RatchetFL that are trained with real-world bugs collected from open-source projects. Our main objective is to predict buggy statement of a given buggy function. RatchetFL takes as input a buggy function $c = \{s_1, s_2, ..., s_n\}$ to locate the single buggy statement s_l ($l \in [1, n]$) by predicting the line number l with the input c i.e., $l = f_{loc}(c)$. We formulate this problem as a multi-classification tasks. Figure 5.2 depicts the overall architecture of RatchetFL, which consists of two sequential layers: *Embedding Layer* and *Feature Learning Layer*

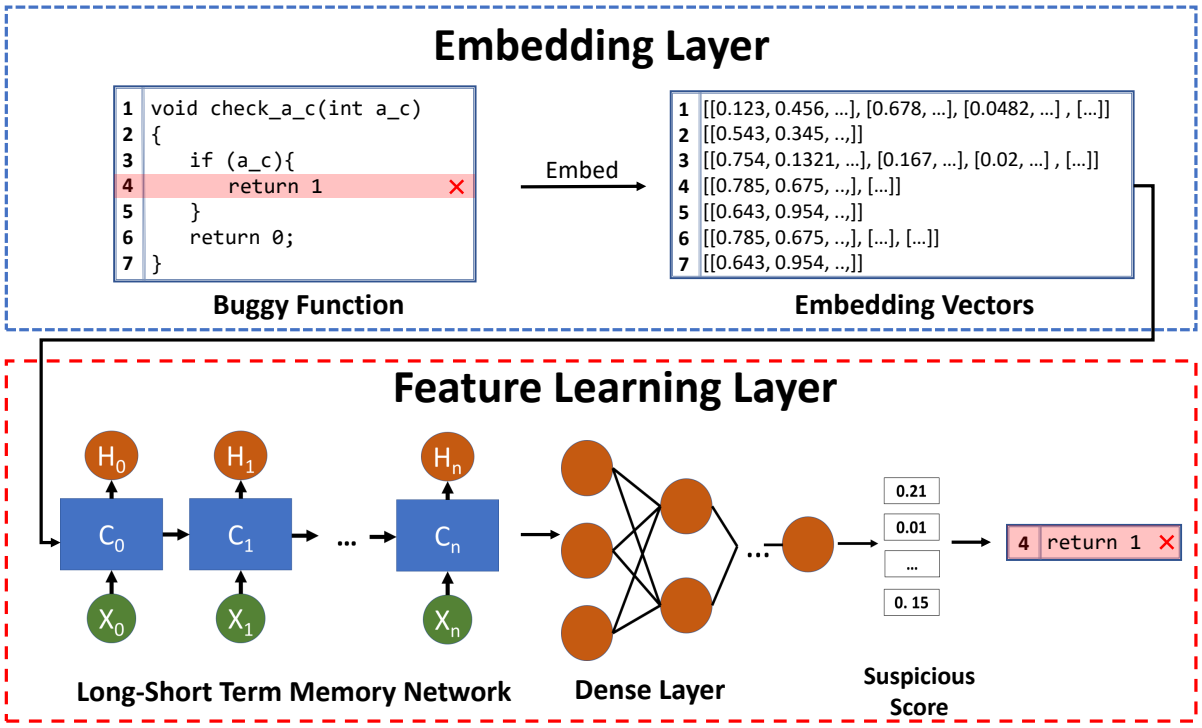


Figure 5.2: Architecture of RatchetFL model.

5.4.1.1 Embedding Layer

We use an embedding layer *Embed* to embed code tokens into their unique vector representation. The semantic and dependency of the code tokens are learnt through these vectors during the training phase. Code identifiers are named by developers and they are often combinations of natural language words/letters to represent a specific notion [166]. Furthermore, string literals and number literals are used to convey discrete values in the source code. These identifiers and literals can greatly increase the redundancy of vocabularies and cause noise during the learning process [167–169].

We employ subword splitting to ease the problem of a large vocabulary. Specifically, we split code identifiers based on camel-case and underscore (e.g., the function name "getPacket" is split into two separated tokens: "get" and "packet"). To further decrease the vocabulary size, we replace string literals and numerical literals into placeholders "<STRING>" and "<INT>" respectively. The subword splitting is commonly used in code intelligent tasks to address the large vocabulary of source code [31, 58].

Our embedding layer $embed$ can be formally defined as follows: For $\forall s_i \in c, s_i = \{t_1, t_2, ..., t_m\}$ where m is the number of tokens t in the i-th statement of the function c, the embedding layer can be represented with the following equation:

$$\mathbf{X} = \mathrm{Embed}(t_1, t_2, ..., t_m) \tag{5.3}$$

where $\mathbf{x_k} \in \mathbf{X}$ is the k-th token representation and $\boldsymbol{X} \in \mathbb{R}^{m \times d}$, d is the length of word embedding dimension.

5.4.1.2 Feature Learning Layer

We employ Bi-Directional LSTM (BiLSTM) [96] in learning the representation of each statement in a buggy function. Specifically, we compute a statement representation, $h_i \epsilon \mathbb{R}^{d^l}$ where d^l is the hyper-parameter, using BiLSTM and $\forall \mathbf{x_k} \in \mathbf{X}$ to achieve a higher-level statement representation. This representation encompasses the semantic and syntax representation as the BiLSTM learnt the implicit relationship of the token sequences. It can be computed using the following equations:

$$\begin{aligned} \mathbf{h_1}, \mathbf{h_2}, ..., \mathbf{h_m} &= \mathrm{BiLSTM}(\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_m}) \\ \mathbf{r_{s_i}} &= [\overrightarrow{\mathbf{h_m}}; \overleftarrow{\mathbf{h_1}}] \end{aligned} \tag{5.4}$$

With the learnt statement-level representations $\mathbf{R} = \{\mathbf{r_{s_1}}, \mathbf{r_{s_1}}, ..., \mathbf{r_{s_n}}\}$ for the buggy function, we use softmax with two fully connected layers to give the probability of each statement that would be faulty. The cross-entropy loss function is selected for the learning process since we model this problem as a multi-class classification task.

5.4.2 RatchetPG - Retrieval Augmented Transformer Model of Patch Generation

Retrieval techniques has achieved great success in the recent deep learning-based tasks [7, 163, 164, 170–172]. These techniques outperforms generation approaches (e.g., RNNs in code-to-text generation tasks like source code summarization [164, 170] and commit message generation [7, 163]) by complementing the generation process with retrieval capability. In this work,

we explore retrieval techniques with deep learning for automated patch generation. Our preliminary study shows that the BLEU-4 (a metric to measure the text similarity between the source and target input) score between the closest retrieved patch statement and the exact patch of a buggy statement is 0.6542 for retrieval-augmented transformer and 0.5547 for an LSTM-based Seq2Seq model. This shows that the generation process is augmented by the closest retrieved patches, further increasing the BLEU-4 score.

This study motivates us to design our retrieval-based patch generation model, RatchetPG, with two parts: (1) retrieving the similar patch for the buggy statement and (2) augmenting retrieved patch into transformer model [173] to generate candidate patches (cf. CRP Retrieval and Patch Generation illustrated in Figure 5.1).

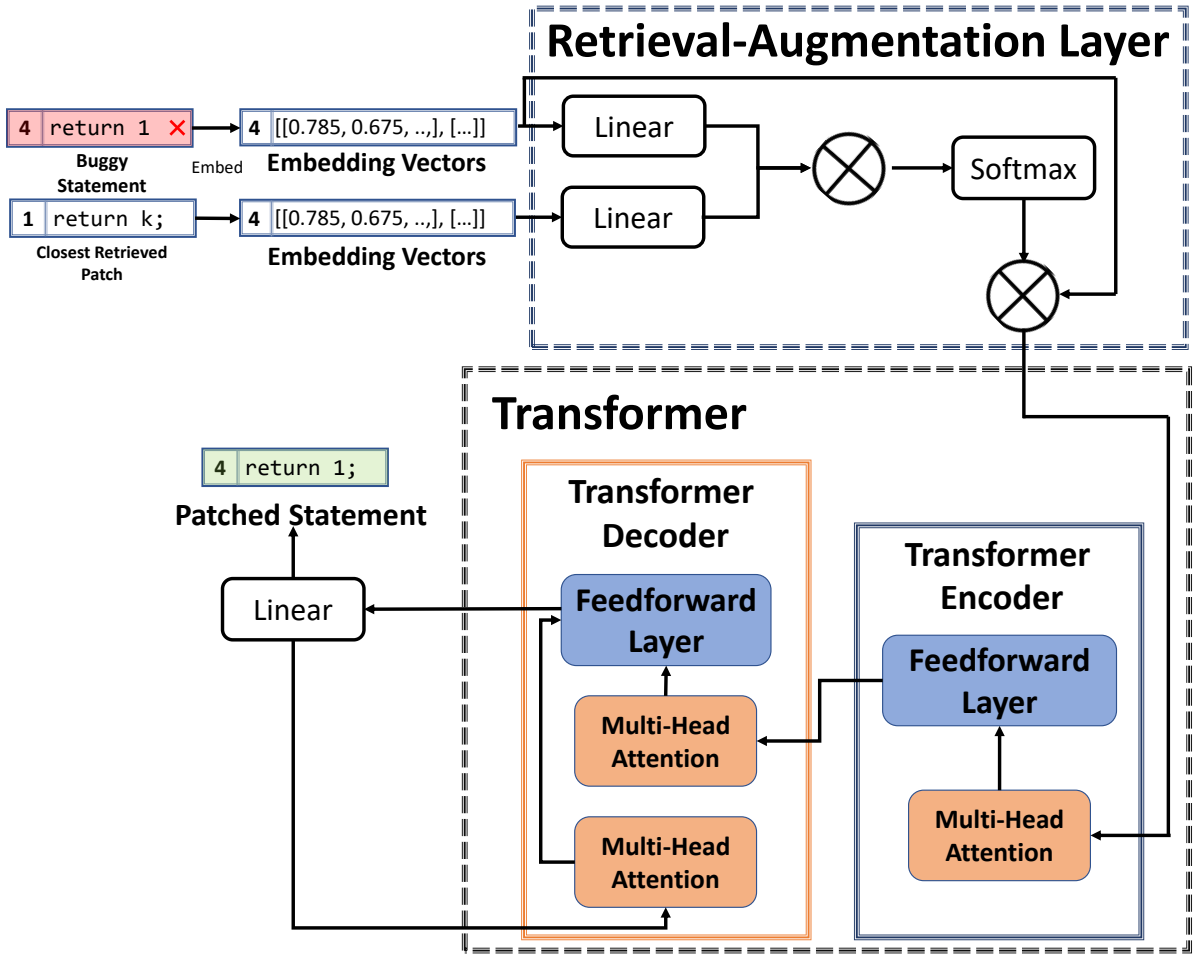


Figure 5.3: Architecture of RatchetPG model.

5.4.2.1 Retrieving Closest Patch for Buggy Statement

Ratchet aims to retrieve the closest retrieved patch (CRP) from the dataset D', a specialized dataset that is composed of all possible patches. Specifically, for a buggy statement s_l in the buggy function c in the dataset D, we seek to find CRP p' through Lucene [174]. We follow Liu et al's workaround [164] to use the patched statements in the training set as D' for Ratchet.

The retrieving process can be formulated as below:

$$p' = \text{argmax}_{p' \in D'} \text{score}(s_l, p') \tag{5.5}$$

where p' represents the CRP retrieved from D', and $p' \neq s_l$. argmax is used to select the p' that is more similar to s_l than other patches in D'.

In particular, we employ Lucene [174] to compute similarities between s_l and all other patches in D'. The similarity score $score(s_l, p')$ can be computed as below:

$$\text{score}(s_l, p') = \text{coord}(s_l, p') + \text{qb}(s_l) + \text{sim}(s_l, p') + \text{db}(p') \tag{5.6}$$

where the function $\text{coord}(*)$ computes the score of overlapping query terms, $\text{qb}(*)$ and $\text{db}(*)$ refers to the query-boost factor and document-boost factor of Lucene, and $\text{sim}(*)$ represents the cosine similarity between s_l and p'. Boost factors can be specified for the concrete documents and query terms, thus the concrete documents or terms can have higher weights.

5.4.2.2 Retrieval-Augmented Transformer

Transformer has been proved the effectiveness in many generation tasks e.g., source code summarization [21], we augmented additional patch information to the buggy statement to generate better patches. Specifically, RatchetPG takes as input the buggy statement s_l and generates the corresponding patch s_p by incorporating the closest retrieved patch p' into our patch generation process. The patch generation of RatchetPG can be expressed as follows:

$$s_p = f_{gen}(s_l, p') \tag{5.7}$$

where f_{gen} is approximated by RatchetPG. Figure 5.3 shows the overall architecture of RatchetPG, which consists of three parts: Retrieval-Augmentation Layer, Transformer Encoder and Transformer Decoder.

5.4.2.3 Retrieval-Augmentation Layer

We embed the buggy statement s_l and its CRP p' into the vector representations using an embedding layer and employ the same pre-processing method, as RatchetFL, to split code identifiers and abstract literals. An attention layer are employed to learn the attention vector $\boldsymbol{A}$ to compute a degree of relevance between s_l and p'. Specifically, for $s_l = \{t_1, t_2, ..., t_i\}$, $p' = \{\hat{t}_1, \hat{t}_2, ..., \hat{t}_j\}$ where i and j are the total number of tokens in s_l and p', X and $\hat{X}$ are the embedded vectors of s_l and p' respectively, the attention matrix $\boldsymbol{A}$ can be expressed as below:

$$\begin{aligned} \boldsymbol{X} &= \mathrm{Embed}(t_1, t_2, ..., t_i) \\ \hat{\boldsymbol{X}} &= \mathrm{Embed}(\hat{t}_1, \hat{t}_2, ..., \hat{t}_j) \\ \boldsymbol{M} &= \mathrm{ReLU}(\boldsymbol{W}^{Q}\boldsymbol{X}) \times \mathrm{ReLU}(\boldsymbol{W}^{R}\hat{\boldsymbol{X}}^{\boldsymbol{T}}) \\ \boldsymbol{A} &= \mathrm{softmax}(\boldsymbol{M}) \end{aligned} \tag{5.8}$$

where $\boldsymbol{W}^{Q} \in \mathbb{R}^{d \times d}$ and $\boldsymbol{W}^{R} \in \mathbb{R}^{d \times d}$ are learnable weights and ReLU is the rectified linear unit [175]. To imbue the retrieved patch feature with the feature of the buggy statement, we multiply the attention matrix $\boldsymbol{A}$ with the retrieved patch features. We then further sum up the original buggy statement feature and the attended retrieved patch feature. The summation can be expressed as below:

$$\boldsymbol{Z} = \boldsymbol{X} + \boldsymbol{A}\hat{\boldsymbol{X}} \tag{5.9}$$

where $\boldsymbol{Z} \in \mathbb{R}^{i \times d}$ represents the final learnt representations of s_l, i is the total token length of the buggy statement and d is a hyperparameter that correspond to the dimension size. We then further employ a Transformer [173] to proceed with the patch generation.

5.4.2.4 Transformer Encoder

The encoder takes as the input $\boldsymbol{Z}$ (the output of the Retrieval-Augmentation Layer) added with the positional encoding for learning. Positional encoding is used to imbue positional information to each token in relation to its position on the code sequences. Transformer Encoder is composed of a stack of N identical layers and each layer has two sub-layers. The first sub-layer is a multi-head attention layer and the second sub-layer is a fully connected feed-forward network. After each sub-layer, the output is further summed with residual connection and layer

normalization. The sub-layer can be expressed as follows:

$$\text{LayerNorm}(x + \text{Sublayer}(x)) \tag{5.10}$$

The multi-head attention layer uses different head h to attend to information from representation subspaces at different positions. Each head employs scaled dot-product attention, which can be computed as below:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}})\boldsymbol{V} \tag{5.11}$$

where $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ represent the key, values, and queries, respectively. Multi-head attention representation can be computed by concatenating the results of scaled dot-product attention as below:

$$\begin{aligned} \text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= \text{Concat}(\text{head}_\text{i}, ..., \text{head}_{\boldsymbol{h}})\boldsymbol{W}^O \\ \text{head}_i &= \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_\text{i}^\text{Q}, \boldsymbol{K}\boldsymbol{W}_\text{i}^\text{K}, \boldsymbol{V}\boldsymbol{W}_\text{i}^\text{V}) \end{aligned} \tag{5.12}$$

where $\boldsymbol{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}_i^K \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}_i^V \in \mathbb{R}^{d \times d_v}$ and $\boldsymbol{W}^O \in \mathbb{R}^{hd_v \times d}$ are model parameters and $d_k = d_v = d/h$. The feed-forward network sub-layer consists of several linear transformation. It can be computed as followed:

$$\text{FFN}(x) = \max(0, x\boldsymbol{W}_1 + b_1)\boldsymbol{W}_2 + b_2 \tag{5.13}$$

where $\boldsymbol{W}_1$, $\boldsymbol{W}_2$, $\boldsymbol{b}_1$, and $\boldsymbol{b}_2$ are weights and bias of the linear transformations.

5.4.2.5 Transformer Decoder

Similarly, the transformer decoder can be stacked up to N identical layers. Different from the encoder layer, the decoder layer includes a third sub-layer that computes multi-head attention over the final output of the encoder layers. This last sub-layer, uses the output of the encoder layers as $\boldsymbol{K}$ and $\boldsymbol{V}$, while $\boldsymbol{Q}$ is the output from the previous decoder layer. Similar to the transformer encoder, the decoder employs the same residual connection layer as the encoder.

Finally, the last layer of the decoder outputs a matrix. We then compute the probability of tokens by using a linear layer with softmax to generate the probability of each token, which can be expressed as below:

$$\mathrm{P(i)} = \mathrm{softmax}(\boldsymbol{HW}) \tag{5.14}$$

where $\boldsymbol{H} \in \mathbb{R}^{l \times d}$, $\boldsymbol{W} \in \mathbb{R}^{d \times l_{\mathrm{vocab}}}$, l is the input sequence length and l_{vocab} is the total vocabulary length.

5.4.3 Inference of Ratchet

With Ratchet that is trained with our collected dataset RatchetDS (i.e., trained RatchetFL and RatchetPG), we can then infer unseen patches through a bug-fixing inference pipeline. Specifically, our inference pipeline accepts a buggy function c, where it is first input into our fault localization model RatchetFL to compute the suspicious score for each statement. We then rank each statement by its score. Formally, for each statement s_i in c where $1 < i \leq n$ and n is number of statements, RatchetFL compute a suspicious score for each statement s_i and sort them by their score to return top-k suspicious statement.

$$S^{sus} = \{s_1^{sus}, s_2^{sus}, ..., s_k^{sus}\} \tag{5.15}$$

We then employ Lucene [174] to search the closest retrieval patch p_i' among all patches in the patch dataset D' for each suspicious statement $s_i^{sus} \in S^{sus}$. Thereafter, RatchetPG takes as input the k suspicious statement and their respective closest retrieval patch p' and generates k candidate patches. Specifically, RatchetPG can be formulated as follows:

$$s_p = f_{gen}(s_i^{sus}, p_i') \tag{5.16}$$

where s_i^{sus} and p_i' is a single suspicious statement with its corresponding closest retrieval patch, and f_{gen} is approximated with our RatchetPG. Finally, we get a set of candidate patches $S_p = \{s_{(p,1)}, s_{(p,2)}, ..., s_{(p,k)}\}$ for the buggy function c.

Table 5.1: Information of projects building RatchetDS.

Project	# Fix buggy Functions	Application Type
libxml2	285	Parser
tcpdump	155	Packet Analyzer
ImageMagick	23	Image Software
cURL	300	Networking/Data Transfer
Glibc	482	C Standard Library
openssl	662	Protocol
Asterisk	931	Communication Toolkit
PostgreSQL	1452	Database
Qemu	3422	Emulator
Wireshark	1196	Network/Packet Analyzer
FFmpeg	697	Multimedia Library
php-src	421	PHP Interpreter
Linux	33549	Operating System

5.4.4 Data Curation

Current existing datasets (e.g., DeepFix dataset [154] and CodeFlaws [176]) often have shortcomings in their design and quantity. For instance, Deepfix dataset consists of student programs with simple mutated errors, such as replacing operators (e.g., replacing `";"` with `","`). Furthermore, the number of unique tokens in the Deepfix dataset is far less than in real-world applications. The dataset of DeepFix consists of only 3,756 unique tokens, while our curated dataset has 56,974 functions with 50,314 unique tokens. This shows the diversity in tokens between real-world applications and student programs. On the other hand, CodeFlaws has 3,902 defects, which is higher than other datasets such as DBGBench [177], IntroClass [178], and ManyBugs [178]. However, this quantity is still far from enough in deep-learning approaches. To address the above challenges, we curate a large benchmark dataset RatchetDS from real-world projects that contain bug-patch pairs.

As shown in Table 5.1, we selected and curated 13 C/C++ open-source projects. We curate them based on two main criteria: (1) Each project contains sufficient commits (ranges from 5,058 to 951,181) that allow us to collect various data. (2) These projects range from a wide range of functionalities that provides our model data from different domains instead of focusing on a single area. For instance, we include codebase from an operating system (Linux), networking application (cURL), database application (PostgreSQL), emulator (QEMU), etc.

Table 5.2: Statistics of Dataset.

Dataset	Total	Training set	Validation set	Testing set
RatchetDS	56,974	45,575	5,695	5,704
DrRepair	57,344	45,875	5,734	5,735

We crawled all the commits of the projects at October 2020 and employed a keyword filtering process [165, 179, 180] to extract bug-fixing commits. Our keyword filtering process checks whether the commit message contains one of the bug-fixing related keywords (e.g., ”fix”, ”cve”, ”exploit”, ”fixes”, ”cves”, etc). We further include the variants of these keywords, such as ”fixes”, ”cves”, ”exploitation”, to capture more patches in our dataset. This process allows us to find patches amidst the raw collected commits. We extract their buggy functions and patched functions and excluded all patches that have more than one-lines of changes from our final dataset. Eventually, we curated the dataset RatchetDS with 56,974 buggy functions with associated patched functions. To ensure validity to our experiments, we randomly split RatchetDS into three sets with 80%, 10%, and 10% to build a training set, a validation set, and a testing set.

We further evaluated Ratchet on other benchmark dataset, such as DrRepair Dataset [24] to evaluate Ratchet on benchmark-overfitting problem [181]. Due to the different collecting criteria with the DrRepair dataset, we organize it with the following preprocessing steps: (1) We randomly selected two single-line bug-patch pairs for each program. (2) We remove any duplicated pairs to avoid over-fitting and redundancy in our experiments [85]. A summary of the RatchetDS and DrRepair dataset are shown in Table 5.2.

5.5 Evaluation Setup

5.5.1 Evaluation Baselines

We evaluate Ratchet in terms of the effectiveness (i.e., on Accuracy@K and Repair Accuracy) on both Fault Localization and Patch Generation. As Ratchet tackles both fault localization and patch generation, we compare against two sets of baselines (i.e., one set for fault localization and one set for patch generation.)

5.5.1.1 Baselines for Fault Localization

Locus [182] employs Vector Space Model (VSM) to localize bugs in source code and codebases. They employ two language models, Code Entity Model and Natural Language Model, to embed the tokens and compute the similarity score between the bug reports and the code elements. A high similarity score implies that the code elements are the bugs that the reports are referencing. We implemented a VSM model based on their approach [182] as Locus is not publicly available.

Similar to Locus, **iFixR** [160] employs TF-IDF with cosine similarity to locate the suspicious statements within a source file. They both employ information retrieval techniques in localizing faults with a file. We selected these two baselines as they are the state-of-the-art retrieval-based fault localization approaches without considering the bug-triggering test cases. As mentioned previously, bug-triggering test cases are uncommon in real-world applications. In the replication of Locus and iFixR, we replace bug reports with commit messages as bug report is not available for all patches.

Furthermore, we also build upon RatchetFL with other neural networks, **Transformer** by replacing our localization model with a Transformer Encoder [173]. This serves to assess the contribution of BiLSTM model in RatchetFL design on localizing fault positions. We further consider two other state-of-the-art learning-based fault localization techniques (DeepFL [183] and DeepRL4FL [184]). They are not considered as part of our comparing baselines since both works require bug-triggering test cases for their neural networks.

5.5.1.2 Baselines for Patch Generation

DeepFix [154] employs NMT approach to generate patches for C/C++ programs. It utilizes Seq2Seq and LSTM networks to generate patched statements based on each line of buggy code in the program. We implement an LSTM-based seq2seq network for generating single line patches for this baseline. We set our LSTM hidden dimension as 128 for this baseline.

SequenceR [152] uses Seq2Seq and Copy Mechanism to handle OOV words. It input the buggy function to generate a fixed line for the buggy function. For this baseline. we directly use the source code that is provided by the authors.

DrRepair [24] employs LSTM and Graph Neural Network [146] to capture the long-range dependencies that exist between the error location and the buggy statements. It then generates the correct fix based on the buggy statements. We use the source code provided by the authors in the necessary comparing experiments. These three APR tools are state-of-the-art learning-based approaches, which are selected as the baselines in this study.

We further consider other works (e.g., DLFix [121], CoCoNut [58], CuRe [66]) that we did not included in our comparing baselines. DLFix [121] only works for Java programs due to the limitation in AST generation. CoCoNut [58], and CuRe [66] are not replicable due to the limitation of their source code[2].

5.5.2 Evaluation Metrics

To evaluate our APR model, Ratchet, we employ Accuracy@TopK (Acc@TopK) as the evaluating metric for the fault localization model, and BLEU-4 and Repair Accuracy (RAcc) as the evaluating metrics for the patch generation model respectively.

5.5.2.1 Accuracy@TopK (Acc@TopK) for Fault Localization

We selected Accuracy@TopK as the evaluating metric for fault localization experiment following previous existing works [182, 185]. Specifically, given a set of localized suspicious statements $S^{sus} = \{s_1^{sus}, s_2^{sus}, ..., s_k^{sus}\}$, if the buggy statement s_l is within S^{sus}, we consider it as located.

$$Acc@TopK = \frac{1}{n}\sum_{i=1}^{n}\delta(\mathrm{FRank}_i \leq k) \tag{5.17}$$

where n is the size of the dataset, δ is a function that returns 1 if the condition is true, otherwise returns 0. FRank is the rank of the first hit result for the buggy statement s_l in the buggy function. We conduct an experiment where k is 1, 3, and 5. A fault localization model with a higher Acc@TopK value indicates a better precision at localizing bugs in buggy function.

[2]The peer practitioners also reported such issues in their repositories.

5.5.2.2 BLEU-4

A widely popular metric for NLP and source-code translation tasks [164,186] is BLEU score [187]. It evaluates the performance by computing text similarity between the model output and the ground truth. We select BLEU-4, a scoring metric based on 4-gram, in computing BLEU score to assess the similarity between the candidate patches and the ground truth. Similar to Accuracy@TopK, a higher BLEU-4 score indicates that the generated outputs are more similar to the ground truth.

5.5.2.3 Repair Accuracy (RAcc)

As previously mentioned in both Section 5.4.1 and Section 5.4.2, our model outputs a set of k plausible patches S_p. We further evaluate Repair Accuracy in experiments, where we check whether any patch within S_p is identical to the ground truth, and assess to what extent accurate patches can be generated by Ratchet. We define the Repair Accuracy as below:

$$RAcc = \frac{1}{n}\sum_{i=1}^{n}\delta(\mathrm{Hit_i}) \tag{5.18}$$

where δ is the function that returns 1 if the input is true, otherwise, returns 0. Hit is an oracle where return 1 if $y \in S_p$, where y is the ground-truth, otherwise return 0.

5.5.3 Experimental Settings

Fault Localization. We employ a 2-layer BiLSTM, word embedding size and LSTM hidden size of 128. We further set dropout [188] to 0.3, batch size to 16 and employ a learning rate of 0.001 and Adam [97] optimizer

Patch Generation. Our RatchetPG employs a Transformer-based model with a dimension size of 128 and feed-forward layer dimension size of 128. We set the following hyperparameters for the RatchetPG: Dropout [188] of 0.2, 4-layer transformer encoder and decoder, 4 heads for multi-head attentions. Similarly, we employ ADAM [97] optimizer and a learning rate of 0.001.

We trained RatchetFL and RatchetPG with our training set, and tune them with our validation set. Our reported performance of these two models is evaluated with the testing set to ensure

validity. For both models, we use patience of 10 epochs and conduct our experiments on three Tesla V100 graphic cards. The total training time for both RatchetFL and RatchetPG took an average of 7 hours and 11 minutes in training respectively.

5.6 Evaluation

We evaluate our approach by comparing against the baselines with two datasets, RatchetDS, and DrRepair Dataset. We structure our experiments by answering the following research questions (RQs):

- **RQ1:** What is the performance of RatchetFL, in terms of Accuracy@TopK? How does RatchetFL perform in Fault Localization compared to the baselines in terms of Accuracy@TopK?
- **RQ2:** What is the patch generation capability of RatchetPG when generating patches for real-world bugs that are localized by RatchetFL? How does RatchetPG perform in Patch Generation as compared to other baselines in terms of Repair Accuracy?
- **RQ3:** What is the context that RatchetPG can best utilize in patch generation, in terms of Closest Retrieval Patch(CRP), Closest Retrieval Buggy Line (CRBL), Closest Retrieval Buggy Function (CRBF), Closest Retrieval Patch Function (CRPF)? Would the patch generation ability of RatchetPG be affected by different contexts?
- **RQ4:** Will Ratchet be affected by the fault localization setting in repairing buggy programs? Is Ratchet sensitive to the fault localization setting for fixing bugs?

5.6.1 RQ1: Performance of RatchetFL on Fault Localization

In this RQ, we evaluate RatchetFL against four fault localization baselines with two datasets (our self-curated RatchetDS and DrRepair dataset). Our experimental results are shown in Table 5.3. We further show the distribution on source code length of buggy functions in Figure 5.4 for better evaluation of our fault localization experiments. As observed from Figure 5.4, the length of the functions in RatchetDS ranges from 1 to 188, while the length of the function in DrRepair dataset ranges from 7 to 46. The median for each dataset is 19. We omitted SBFL

Table 5.3: Results of Fault Localization.

Methods	RatchetDS			DrRepair Dataset		
	Acc@Top1	Acc@Top3	Acc@Top5	Acc@Top1	Acc@Top3	Acc@Top5
Locus	0.0689	0.2148	0.4504	0.0561	0.2396	0.4567
iFixR	0.0721	0.2205	0.4430	0.0572	0.2387	0.4619
DrRepair	0.3585	0.6615	0.8029	0.5500	0.7597	0.8777
Transformer	0.3462	0.6252	0.7614	0.7974	0.8936	0.9414
RatchetFL	**0.3983**	**0.6736**	**0.8075**	**0.8821**	**0.9376**	**0.9643**

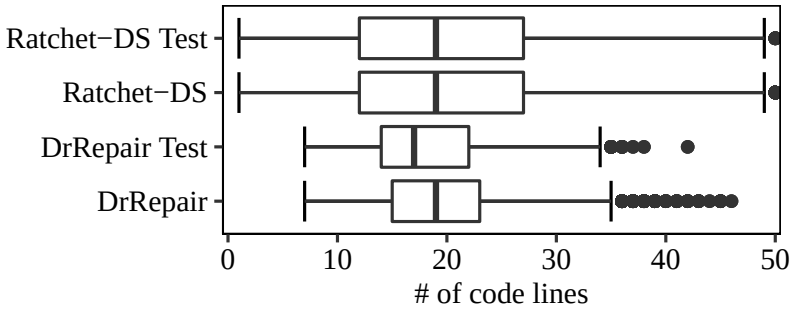


Figure 5.4: Distribution on the code lines of buggy functions.

techniques for comparison as both of our comparing datasets are lacking bug-triggering test cases.

Our RatchetFL outperform both information retrieval approaches (Locus and iFixR) by 32.62-45.90% in the RatchetDS and 50.20-82.60% in DrRepair Dataset in terms of Accuracy@Top1, Accuracy@Top3, and Accuracy@Top5. One possible reason for the low performance of Locus and iFixR could be the different retrieval indexes. Locus and iFixR originally employ bug reports for fault localization in the source file. However, it is infeasible to correspond each bug in our comparing datasets to a specific bug report, especially for a massive dataset. Therefore, we replace bug reports with commit messages. We argue that our approach is acceptable as many bugs are not associated with any bug reports [189]. Despite the lack of bug-triggering test cases, RatchetFL does not require any bug-related artifacts unlike other fault localization tools [160, 183, 184]. This allows RatchetFL to locate faults at the early software development stages, where they commonly do not have many software artifacts.

Our experiments show that DrRepair and RatchetFL achieve better results in identifying erroneous and buggy statements in both RatchetDS and DrRepair datasets at the three comparing

metrics. RatchetFL outperforms DrRepair in the range of 0.46-3.98% in the RatchetDS and 8.66-33.21% in the DrRepair dataset. In other words, RatchetFL can rank the buggy statements in better positions in the suspicious statement list than DrRepair. This implies that the simple BiLSTM model of RatchetFL outperforms DrRepair and Transformer Encoder in localizing buggy statements. RatchetFL effectively captures the feature from source code through our BiLSTM model, although DrRepair leverages the combination of LSTM and GNN to expose bug position with the source code and compiling message. One of the possible reasons might be that we employ the code-only model of DrRepair instead of the model that uses compiler messages. Compiler messages commonly present additional information, such as the line number of the last executed statements. However, it is infeasible to expect compiler messages of real-world bugs as compilation could take hours to complete.

Notably, RatchetFL is better at finding bugs in the DrRepair dataset than RatchetDS, with an improvement of 48.4% in Accuracy@Top1, 26.4% in Accuracy@Top3 and 15.7% in Accuracy@Top5. One key distinction between DrRepair dataset and RatchetDS is that DrRepair dataset contains mainly student programs. The buggy programs are created artificially through mutation of operators and variables, which are much simpler than the real-world bugs in RatchetDS. RatchetDS is curated with 56,974 functions from real-world programs, which shows a higher diversity than bugs from student programs. This implies that localizing bug position in in-the-lab bugs is much easier than in-the-wild bugs.

Answer to RQ1: Our fault localization model, RatchetFL achieves promising performance on localizing bugs in both in-the-lab and in-the-wild datasets. Furthermore, our approach does not require additional software artifacts (e.g., bug reports or bug-triggering tests), outperforming information retrieval-based and learning-based approaches. We also discover that performance on in-the-lab bugs might not be feasible for real-world applications.

5.6.2 RQ2: Performance of Ratchet on Program Repair

We evaluate the performance of Ratchet (i.e., RatchetFL and RatchetPG) at automated program repair and compare it with three other deep-learning baselines, DeepFix, SequenceR, and DrRepair. DeepFix [154] and SequenceR [152] assume a perfect fault localization setting where the location of the buggy statement must be known. To avoid a bias from different fault localization

Table 5.4: Results of Automated Repair.

Methods	RatchetDS		DrRepair Dataset	
	BLEU-4	RAcc	BLEU-4	RAcc
RatchetFL+DeepFix	0.1551	0.0755	0.2418	0.3703
RatchetFL+SequenceR	0.0642	0.0492	0.0764	0.1830
DrRepair	-	0.0002	-	0.2119
Ratchet	**0.1843**	**0.1951**	**0.2552**	**0.4638**

The BLEU-4 value of DrRepair is unavailable since DrRepair does not provide an interface for it.

settings, we employ RatchetFL in localizing the bug positions for them before generating the candidate patches. Since DrRepair employs both fault localization and patch generation into one single model, its fault localization setting is unchanged in this experiment.

Table 5.4 shows the experimental results of RQ2. DeepFix and DrRepair achieve better performance on the DrRepair dataset than RatchetDS. Both these learning-based APR tools were inspired by student programs. Furthermore, they were proposed and evaluated on the DrRepair dataset (i.e., in-the-lab bug datasets). We infer that learning-based APR models should be built and learnt from the in-the-wild dataset as the model built from the in-the-lab dataset can be impractical and their performance can be lacking when solving real-world bugs.

Our model, Ratchet, outperforms all learning-based APR baselines at generating patches for bugs in both RatchetDS and DrRepair datasets. Similarly, Ratchet has significant better performance in generating patches for RatchetDS than DrRepair dataset, having an improvement of 26.87% in Repair Accuracy. This is consistent with the performance of RatchetFL in Section 5.6.1 since mutated bugs from student programs are simpler and not as complicated as real-world bugs. We infer that Ratchet benefitted from our proposed retrieval-based patch generation model by incorporating semantically and syntactically correct code sequences. The employing contexts also enable our model to attend to tokens that are not within the source inputs (i.e., buggy statements or buggy functions)

Answer to RQ2: Ratchet outperforms all other baselines by an average of 21.49% across both datasets, demonstrating that our retrieval augmented approaches can generate better patches as compared to other approaches. Furthermore, Ratchet displays overwhelming performance in the in-the-wild dataset, achieving a Repair Accuracy of 19.51% in RatchetDS.

Table 5.5: Results of patch generation with various contexts.

Context	RatchetDS		DrRepair Dataset	
	BLEU-4	RAcc (Num)	BLEU-4	RAcc (Num)
w/o C	0.1815	0.1832 (1045)	0.2533	0.4397 (2522)
CRBL	**0.1846**	0.1921 (1096)	0.2509	0.4333 (2485)
CRBF	0.1737	0.1649 (941)	0.2480	0.4186 (2401)
CRPF	0.1821	0.1823 (1040)	0.2501	0.4292 (2462)
CRP	0.1843	**0.1949 (1112)**	**0.2552**	**0.4598 (2637)**

*"(Num)" denotes the number of bugs fixed by RatchetPG with the corresponding context setting.

5.6.3 RQ3: Impact of Retrieval Contexts on RatchetPG

As shown in previous RQs where Ratchet learns from correct patches (Closest Retrieval Patches), Ratchet fixes more bugs than the state-of-the-art learning-based APR baselines. In this RQ, we investigate the impact of different retrieval contexts on the performance of patch generation. Specifically, we selected four different retrieval contexts and evaluate their performances: Closest Retrieval Buggy Line (CRBL), Closest Retrieval Buggy Function (CRBF), Closest Retrieval Patch Function (CRPF), and one experiment without any context (w/o C). The experimental results of different contexts are presented in Table 5.5.

RatchetPG outperforms other context settings on both RatchetDS and DrRepair Dataset when CRP context is employed. Comparing on BLEU-4, RatchetPG achieves higher BLEU-4 in all context settings, except for CRBL. This further implies that a learning-based model can generate similar patches but the correctness of these patches cannot be ensured, highlighting the importance of correct patches.

We observed that when RatchetPG are trained on incorrect and negative example (e.g. buggy statements and buggy functions), there is a negative impact on the patch generation on RatchetPG. We also observed that RatchetPG performs worse when function-level contexts are employed. One possible reason is that closest buggy or patch functions have much more tokens than statement-level contexts. On average, there are 13 tokens on the statement-level contexts, while there are 122 tokens on the function-level contexts. The great differences in the number of tokens can cause a long-range dependency problem, such that the model cannot focus on the correct and important tokens.

Table 5.6: Impact of Fault Localization on Program Repair.

Fault Localization (FL) Settings	**RatchetDS**		**DrRepair Dataset**	
	BLEU-4	RAcc	BLEU-4	RAcc
No FL + RatchetPG	0.1938	0.0251	0.1910	0.0129
Perfect FL + RatchetPG	0.6573	0.1971	0.8573	**0.4656**
Ratchet	0.1857	0.1921	0.2552	0.4638

Our experiment shows that employing CRBF contexts causes the worst performance among all contexts. We infer that a longer sequence of contexts can lead to a decrease in performance in patch generation. Furthermore, learning from closest retrieval patches boosts RatchetPG's ability to generate patches that are syntactically correct. Our qualitative analysis in Section 5.7.1 further shows that RatchetPG alleviates the missing token problem by finding and attending to tokens that do not exist in the input source code.

Answer to RQ3: Choosing the correct context for patch generation is important. Our experiment shows that using CRP as context greatly contributes to generating correct patches for bugs. We also observed that employing closest retrieval buggy/patch functions will not improve, or even negative impact, the performance of RatchetPG. We attribute this problem to the long sequence of function-level contexts and long-range dependency problems. On the other hand, statement-level contexts are effective in providing additional contexts for the patch generation model to learn from.

5.6.4 RQ4: Significance of Fault Localization

The setting of fault localization has a significant impact on the performance of APR tools [155]. We further investigate the impact of fault localization on repairing bugs for Ratchet. Specifically, we conduct an experiment where instead of employing buggy statement as source input, we employ buggy functions. Our objective is to generate patches based on the buggy function, (i.e., without fault localization (No FL) in Table 5.6). This eliminates the need for localization as the whole buggy function will be used for the generation of the patches, including the buggy statement. We further proceed a perfect fault localization setting for RatchetPG the same as CoCoNut [58], CuRE [66], and SequenceR [152].

Table 5.6 presents the bug-fixing results of RatchetPG with different fault localization settings. As observed in the table, RatchetPG with a perfect fault localization setting performs the best on generating patches, outperforming RatchetPG with "No FL" and base Ratchet. This result is consistent with the result shown in other APR literature [65]. On the other hand, despite Ratchet does not outperform RatchetPG with perfect fault localization on BLEU-4 values, the repair accuracy of Ratchet on both datasets (RatchetDS and DrRepair dataset) are just slightly lower than the perfect fault localization setting. This result indicates that Ratchet is impervious to the fault localization setting as compared to other patch generation approaches.

Answer to RQ4 : As pointed by Liu et al. [155], the performance of fault localization plays an important role in APR. Our evaluation shows that different fault localization settings impact the performance of RatchetPG, but the impact is trivial on generating correct patches for Ratchet. Our research question also highlights the need for an accurate localization method in cooperation with an effective patch generation method to enable successful automated program repair.

5.7 Discussion

We conduct a qualitative analysis of the patches generated by Ratchet (RatchetPG) with two case studies. Furthermore, we investigate the impact of the selected k value on generating patches for Ratchet and discuss the threats to the validity.

5.7.1 Qualitative Case Studies

We present two high-quality case studies patches that are generated by Ratchet. Figure 5.5 and Figure 5.6 show two examples of patch generation via Ratchet from our test set, where the "Buggy Function" shows the buggy function (input of corresponding models) and the ground-truth patch, and "CRP" shows the closest retrieved patch identified by Lucene. Lastly, "DeepFix" and "Ratchet" denote the patches generated by themselves respectively.

Figure 5.5 shows a patch excerpted from commit 809fb76 in the open-source project, Wireshark. We observed that, in the ground truth patch, an additional argument "`pinto`" is added as argument for the invocation of function "`fVendorIdentifier`". As observed in the Figure,

Fixed/Buggy Function(Ground Truth)

```
static guint fIAmRequest  (tvbuff_t *tvb, proto_tree *tree,
guint offset)
{
     offset = fApplicationTypes (tvb, tree, offset, "BACnet
Object Identifier: ");
     offset = fApplicationTypes (tvb, tree, offset,
"Maximum ADPU Length Accepted: ");
     offset = fApplicationTypesEnumerated (tvb, tree,
offset,"Segmentation Supported: ", BACnetSegmentation);
-    return fVendorIdentifier (tvb, tree, offset);
+    return fVendorIdentifier (tvb, pinfo, tree, offset);
}
```

CRP:	`return faddress(tvb, pinfo, tree, offset);`
DeepFix:	`return fVendor.((tvb, tvb, tree, offset);`
Ratchet:	`return fVendorIdentifier (tvb, pinfo, tree, offset);`

Figure 5.5: Ground-truth and generated patches for bug Wireshark:809fb76.

DeepFix infers that this bug can be fixed by adding an argument into the statement. However, it cannot accurately determine the correct function name (e.g., omitted the sub-token "`Identifier`"") and arguments (inserted the variable "`tvb`" as argument instead of "`pinto`"). Despite the generated patch by DeepFix are syntactically similar to the ground truth, it does not correspond to the correct patch. Ratchet determines the function call and syntax of the return statement via our proposed retrieval-augmented model. As observed in CRP in Figure 5.5, CRP provides the correct parameters for the patch, hence, allowing Ratchet to generate the correct patch that is identical to the ground truth.

Figure 5.6 shows another example excerpted from commit 5489377 of Linux. The example depicts a modification to the function header, with a newly added specifier to the parameter "`blkif`". Similarly to the first example, DeepFix infers the correct specifier type ("`struct`") for the parameter. Despite the generated patch by DeepFix being syntactically similar to the ground truth, DeepFix still fails to predict the correct variable for the buggy statements. Our proposed model, Ratchet, first correctly located the bug position, which is the function header. This can be a challenge for spectrum-based fault localization techniques [155, 165].

Fixed/Buggy Function(Ground Truth)

```
- static void print_stats(blkif_t *blkif)
+ static void print_stats(struct blkif_st *blkif)
 {
      printk(KERN_DEBUG "%s: oo %3d  |  rd %4d  |  wr
%4d  |  br %4d\n",
             current->comm, blkif->st_oo_req,
             blkif->st_rd_req, blkif->st_wr_req, blkif-
>st_br_req);
      blkif->st_print = jiffies + msecs_to_jiffies(10 *
1000);
      blkif->st_rd_req = 0;
      blkif->st_wr_req = 0;
      blkif->st_oo_req = 0;
 }
```

CRP:	`static void blkif_notify_work(struct blkif_st *blkif)`
DeepFix:	`static void print_stats(struct blkif_blkif * blkif)`
Ratchet:	`static void print_stats(struct blkif_st * blkif)`

Figure 5.6: Ground-truth and generated patches for bug Linux:5489377.

After localizing the fault position, Ratchet retrieves the CRP with the correct structure from other functions. Despite CRP providing the correct contexts, Ratchet did not directly copy the CRP into the generated patch. Instead, Ratchet focuses on the argument list on the CRP and extracts it to augment our patch generation process. Hence, a correct patch is generated by our RatchetPG. This shows that the ability of RatchetPG to learn insights and adapt from CRP.

5.7.2 Selection of K's Value for Ratchet

Ratchet generates k candidate patches by using the top-k suspicious statements after localization by RatchetFL as previously mentioned in Section 5.4.3. The value of k directly impacts the effectiveness of fixing bugs as more candidate patches can increase the chances of generating the correct patches [65]. On the contrary, if the value of k is not large enough, the performance of Ratchet can be negatively impacted. Hence, we investigate the impact of different values of k on Ratchet. Specifically, we apply different values ({1, 3, 5, 7, 9, 11}) of k to assess the performance of Ratchet in localizing bugs and generating patches.

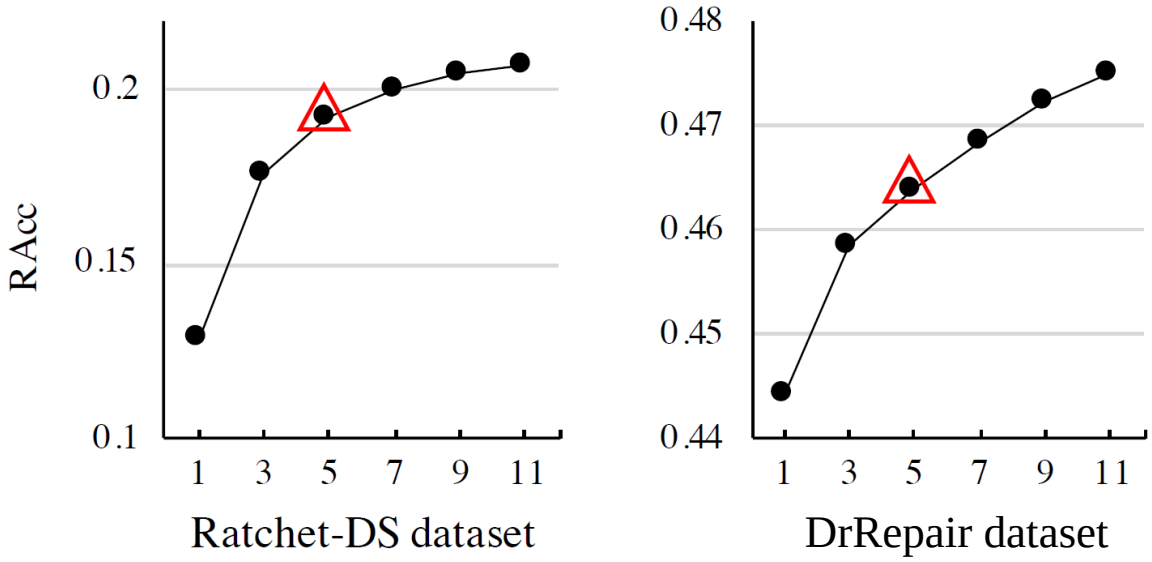


Figure 5.7: Impact of K's values on Ratchet.

Figure 5.7 shows the Repair Accuracy of Ratchet with different k values for both RatchetDS and DrRepair dataset. As observed from both graphs in Figure 5.7, the repair accuracy of Ratchet has minimal improvement after the value of k rises above 5. The repair accuracy of Ratchet on RatchetDS increases sharply before reaching a plateau of approximately 0.2 when k's value increases to 5. After the value of k increases above 5, the repair accuracy of Ratchet increases marginally, with an increment of less than 1%. A similar trend can be seen on the DrRepair dataset. With a large number of candidate patches, there is a higher possibility of capturing correct patches. Therefore, the value of k should be as high as possible. However, this causes an increased amount of plausible patches and hinders the generation and validation process [181]. Furthermore, this can also impact the efficiency of fixing candidates [65]. Hence, an optimal number of generated candidate patches should be sought, without a sharp decrease in Repair Accuracy. Most of the buggy functions have ~20 statements as illustrated in Figure 5.4. Hence, $k = 5$ is a reasonable setting for fault localization.

5.7.3 Threats to Validity

5.7.3.1 Threats to External Validity

RatchetFL and RatchetPG are not unique to C Programming Language as we do not employ any programming-specific techniques. The quality of our dataset can also be a threat to our work. We ensure validity by curating our dataset through a strict selection and filtering process. Furthermore, we publicly release RatchetDS to enable better transparency and replication. To ensure that our work is valid on other datasets, we further evaluate Ratchet on the DrRepair dataset.

5.7.3.2 Threats to Construct Validity

Our implementation on Locus and iFixR differs from their original paper as bug reports are not available widely and cannot be gathered on a large scale on both datasets. Furthermore, we presented a fault localization approach that localizes bugs without additional artifacts.

Although our reported results of DrRepair are lower than the reported result on their original paper, we employ the source code as given by the authors, hence, the result is supported by their source code. One possible reason for the low result is that we employ different metrics as compared to DrRepair (i.e., Localization Accuracy and Accuracy@TopK). The test set of DrRepair also contains overlapping data points with the training set. Out of 4,979 bug-patch pairs in the test set, 4,280 pairs exist in the training set. We refer readers to the source code [24] for more details.

5.8 Conclusion

We present our deep-learning-based approach in automated program repair, Ratchet, that comprises a fault localization model RatchetFL and a patch generation model RatchetPG. Ratchet first employ RatchetFL in localizing fault in given buggy functions by learning the semantic and syntax of the buggy statements. As compared to other fault localization techniques, RatchetFL does not require additional software artifacts for localization. After the bug position is located, Ratchet employs RatchetPG to generate candidate patches for the bugs. RatchetPG utilizes a retrieval-augmented transformer, which generates better patches for fixing software

bugs. Our experimental result also shows that Ratchet outperforms our comparing baselines in both comparing tasks. With a good learning-based program repair tool, software security can be enhanced by reducing the vulnerable time in patching vulnerabilities and security bugs, fitting into our proposed facilitation of better software security through a data-driven approach.

Chapter 6

Learning Program Semantics with Code Representations: An Empirical Study

6.1 Introduction

The large and growing body of successful, widely used, open-source software products (*e.g.,* Linux, Django, Ant) have contributed to a concept: *Big Code* in software engineering. *Big Code* means the scale of available data is massive: billions of tokens of code and corresponding meta-data, *e.g.,* changes, code reviews. With the booming development of open-source software, the amount of available code-related data has reached an unprecedented scale, which inspires researchers from both academia and the industry to explore employing data-driven approaches for diverse code-related problems such as type inference [190–192], clone detection [12,28,29], source code summarization [18–21], code search [25–27] and software vulnerability detection [31–33]. Most of the existing works attempt to understand the behavior of the program i.e., understand the program semantics by the well-designed approaches for different tasks and have achieved promising results. Typically, we can broadly categorize these data-driven code-related works into four major categories: Feature-based, Sequence-based, Tree-based, and Graph-based representation to learn the program semantics for different tasks.

The work in this chapter has been accepted in 2022 IEEE 29th International Conference on Software Analysis, Evolution and Reengineering (SANER)

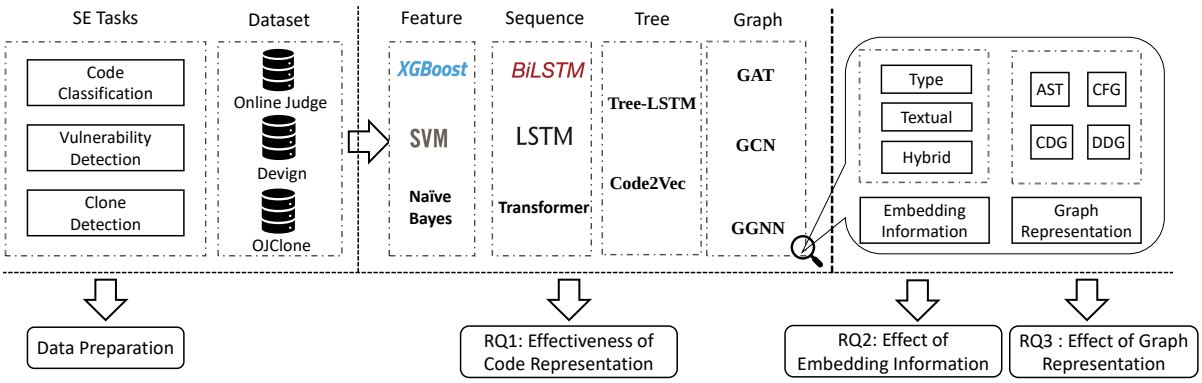


Figure 6.1: Overview of Our Study.

Feature-based approach requires the domain knowledge extracted from the program by the experts to represent the program semantics for different tasks. For instance, FLUCSS [193] is designed for fault localization. It incorporated Spectrum Based Fault Localisation [149] metric and source code related metrics to identify the fault in the software. It further performed learning-to-rank with Support Vector Machine [194] and Genetic Programming [195]. Their approach achieved a performance of 50% Mean Average Precision (MAP) in ranking software defects. Bhel et al. [196] proposed mining security bugs from bug reports using TF-IDF [197] and Naive Bayes [198] algorithm. They achieved high precision e.g., 92.56% for detecting the bug report.

Sequence-based representation treats code as a flat sequence of tokens [199, 200] and converts them into numerical vectors with the distributed representation [84] and further employs these vectors for diverse tasks. For example, VulDeepecker [32] predicted if a program is vulnerable by learning on code gadgets. Code gadgets are generated by slicing programs through the function calls. These code gadgets are then input into a bi-directional LSTM [96] for learning whether the program is vulnerable. They achieved a score of 85.4% in F1-score across multiple vulnerability types. Source code summarization is another popular application of the sequence-based approach. Hu et al. [201] aimed to translate source code into their respective summary. They employed API sequences and code tokens as the input for an encoder-decoder model to generate the summary. They achieved 41.98 in BLEU-4 [187] and 18.81 in METEOR [202].

Since programs are highly structured data, which can be converted into different structural representations such as the abstract syntax tree (AST). Many works attempt to explore this

information hidden in the text for different tasks [13,169,203,204]. For instance, code2vec [13] predicted method name through the path contexts extracted from the AST of the program and achieved an F1-score of 58.4%. CDLH [204] employed tree-based representation to detect code clones. It employed LSTM in learning an AST-based representation and hashed them to achieve a unique vector on each program. CDLH achieved promising results for Type-3 and Type-4 clone detection i.e., 81-94% F1-score.

The graph-based approaches attempt to embed more structural information such as data-flow, control-flow rather than the pure AST into a graph [14, 18, 19, 205] to represent the program semantics. For instance, Allamanis et al. [14] targeted at detecting variables that are incorrectly used in a project and predicting the correct variable. They enhanced AST with different data flow information by constructing diverse types of edges among the nodes, such as connecting variable nodes where the variable was last written to, to achieve a graph-based representation on the program. Their approach achieved a high accuracy of 85.5% and 78.2% in finding wrong variable names on the seen and unseen projects respectively.

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

- RQ1: Comparison of Feature-based, Sequence-based, Tree-based and Graph-based Representation.
- RQ2: Comparison of Node Embedding Information.

- RQ3: Efficacy of Different Graph Representation.

To answer these questions, we design our study on three popular and diverse code intelligent tasks: Code Classification, Vulnerability Detection and Clone Detection with typical approaches for these representations i.e., SVM, Naive Bayes, XGBoost for the feature-based representation; LSTM, BiLSTM, and Transformer for the sequence-based representation; Code2Vec, Tree-LSTM for the tree-based representation and Graph Convolution Neural Network (GCN), Graph Attention Network (GAT) and Gated Graph Neural Network (GGNN) for the graph-based representation. We further utilize the public released dataset for the evaluation and employ Joern [206] for the unification on code property graph (CPG) construction for the graph-based representation. Specifically, CPG contains abstract syntax tree (AST), control flow graph (CFG), control dependency graph (CDG), and data dependency graph (DDG) to facilitate investigating each component. By the extensive experiments (around **1000** GPU hours), we conclude that: (1) Since graph-based representation incorporates diverse program semantics, it outperforms other compared representation techniques by a significant margin among the selected tasks. (2) Both node type and node textual information are beneficial in capturing program semantics, however, the textual information is more critical. (3) Different task relies on the task-specific semantics to achieve the best performance, however, generally, a composite graph representation with the comprehensive program semantics can still produce promising results.

In summary, we make the following contributions:

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

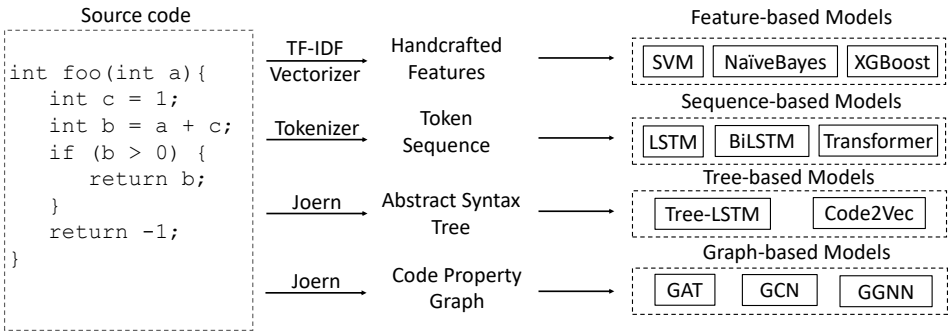


Figure 6.2: Techniques on Code Representation.

The remainder of this study is organized as follows: Section 6.2 describes the overview of our study, which contains the evaluation tasks and different code representation techniques. We elaborate on our experimental settings and results in Section 6.3. Section 6.4 provides more discussions about these code representation techniques. Lastly, we conclude this work in Section 6.6.

6.2 Overview

An overview of our study is shown in Fig. 6.1. In this study, we aim to answer the effectiveness of different code representations (Section 6.2.2) for different tasks (Section 6.2.1) on the public released benchmark. We categorize current state-of-the-art code representation techniques into feature-based, sequence-based, tree-based, and graph-based code representation and design RQ1-RQ3 (Section 6.3) for the comprehensive investigation.

6.2.1 Evaluation Tasks

We select three diverse tasks i.e., code classification [12], clone detection [12] and vulnerability detection [31] for our study. We selected these tasks based on two reasons: (1) These tasks can effectively evaluate the semantic, lexical, syntactic information that is on the comparing evaluations. For instance, control dependency and data dependency are crucial in detecting vulnerabilities as the data flow of variables can indicate wrongful usage of variables [32]. Lexical information is important for clone detection as programs that uses the same tokens might be more likely to be cloned. A code representation that performs well in all three evaluating tasks infers that the representation can learn better semantic, lexical, and syntactic information. (2)

Table 6.1: Statistics of Dataset.

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

The three evaluating tasks are popular in the software engineering domains. We collected the number of publications, in recent years in top leading conferences, e.g., ASE, ICSE, NeurIPS. We discovered that there are at least two, five, and four publications that are relevant to source code classification, vulnerability detection, and clone detection. This shows that our evaluating tasks are popular and important.

Code Classification. Code classification aims to classify the code fragments by their functionalities. which is vital for program understanding. Given a program, code classification aims to identify which category it belongs to from a category set. We employ OJ dataset [207], a C Programming Language dataset, for code classification. The dataset contains 52,000 programs that are classified into 104 classes and each class performs a high-level operation, such as reversing an integer or finding minimum/maximum words from an array of words.

Clone Detection. Clone detection detects whether two code fragments achieve the same functionality with different implementations. We follow the same approach as Zhang et al. [12] and Wei et al. [204] to generate the code clone dataset based on OJ dataset [207]. Two programs in the same class can be considered as at least a functionality clone as they fulfill the same functionality [204]. Therefore, we gather the clones by pairing the programs that are in the same class and non-clones by pairing the programs that are in different classes. We randomly selected 25,000 programs and pair each of them with a clone and non-clones. We specifically ensure that a program that appears in the training set will not appear in the testing to ensure a

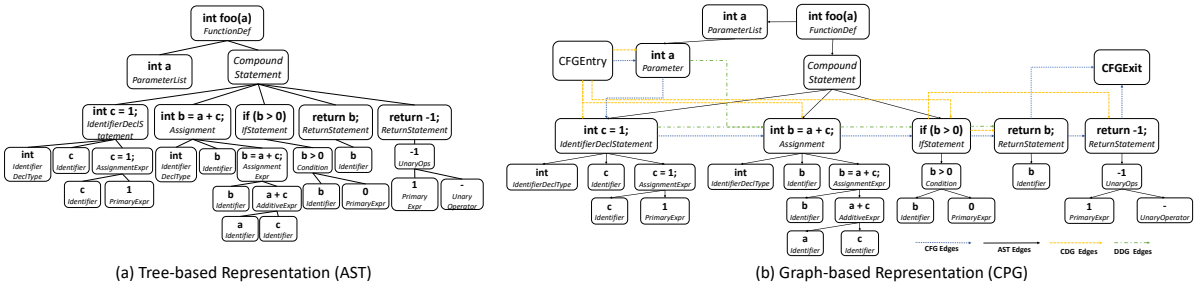


Figure 6.3: The Illustration of Tree and Graph Constructed by Joern.

fair evaluation. To differentiate from the dataset that we used in code classification, we refer to this dataset as OJClone.

Vulnerability Detection. The goal of vulnerability detection is to detect the vulnerable code fragments that may be attacked for cyber security. We utilize the completed [2] Devign dataset [31], which contains 66,067 labeled functions that are collected from four open-sourced C programming language projects, FFmpeg, Wireshark, Linux, and Qemu for the evaluation. These functions are classified as either vulnerable or non-vulnerable. Zhou. et al. verify security patches manually and extract the functions from these verified patches [31], hence, their reliability is ensured.

We remove the duplicate functions to ensure a fair comparison. Furthermore, we employ Joern [206] to extract the tree and the graph from the programs. However, due to some compilation errors, finally, there is a set of 35,831 programs in our OJ dataset and 48,158 programs in the Devign dataset and the statistics of the datasets are shown in Table 6.1. We split all our datasets into 80% for the training set, 10% for the validation set, and 10% for the testing set.

6.2.2 Code Representation

As shown in Fig. 6.2, to evaluate the efficacy of different code representation, we categorise the code representation into four major categories: Feature-based, Sequence-based, Tree-based and Graph-based Representation.

[2] We expressed our sincere thanks to the Devign Team to share the completed dataset with us for the experiments.

6.2.2.1 Feature-Based Representation

In our study, we used Term Frequency-Inverse Document Frequency (TF-IDF) to vectorize the code snippet. TF-IDF computes a vector for each program based on the term frequency and inverse document frequency, which reduces the importance for stopwords and increases the weightage for more relevant words. To evaluate feature-based representation, we selected three machine learning algorithms: Support-Vector Machine (SVM) [194], Naive-Bayes [198] and XGBoost [208]. We use TF-IDF as our feature vector and input these vectors to the models. SVM performs the classification by finding a hyperplane in the dimensional space that can distinctly identify samples in different classes, while Naive-Bayes computes the probability of a sample within a class with an assumption of independent features. XGBoost is an implementation of a gradient boosting machine that is known for scalability and performance. These models are commonly used in many SE tasks as the comparison baselines [12, 31].

6.2.2.2 Sequence-Based Representation

Since code snippets can be treated as a flat sequence of tokens, many works utilize this sequential information directly to learn program semantics. Following the previous works [14, 20, 31], we also employ token sequence directly in evaluating the sequence-based representation of the programs. We further employ sub-word splitting for efficient learning, where words are split by their camelcase and underscore. For instance, a function name, ”get_int”, is split into two subword tokens, ”get” and ”int”. The result of this sub-word splitting is a sequence of subwords tokens. Their purpose is to reduce the vocabulary size of the datasets. For evaluating sequence representation, we opt to use Long-Short Term Memory (LSTM) [90] and Bi-directional LSTM (BiLSTM) [96] network as they are suitable in training sequential data, i.e., sequence of tokens. For our experiment, we trained a LSTM and Bi-directional LSTM (BiLSTM) model to evaluate the performance on learning source code sequentially. We take the last hidden state of LSTM/-BiLSTM as the learnt representation of the sequence representation. Furthermore, we added in transformer encoder [173] as part of our baseline to investigate the impact of the attention-based sequence approach. Transformer employs multi-head attention to learn a representation for a sequence and we utilize the contextual representation produced by the transformer encoder as the learnt program representation.

6.2.2.3 Tree-Based Representation

The program is a highly structured data compared to the sequential data, hence many code-related works attempt to extract the structure information e.g., abstract syntax tree (AST) [12–14,204] behind the text to capture the semantics. An example of AST produced by Joern can be seen in Fig. 6.3(a). We selected two approaches for evaluating tree-based representations, Tree-LSTM [209] and Code2Vec [13]. Tree-LSTM employs LSTM in learning the network topology of the input tree structure, or in our case, the AST. It computes the hidden states based on its successors. As opposed to the single forget gate used in LSTM, it uses one forget gate for each child to focus on important information and outperforms several sequence-based approaches. Furthermore, several works [169,210] extend or employ Tree-LSTM as its baselines in software engineering. In our study, we employ the Child-Sum Tree-LSTM where the network learns the hidden states based on the summation of the children states. We employed max-pooling over all node representations to achieve the tree-based representation vector. Code2Vec [13] is a state-of-the-art technique in code representation. It represents programs in a bag of path context, where path context represents a path between terminal nodes across the AST. Code representation is learnt through focusing on these path contexts. We extracted the path contexts from the tool ASTMiner [211] and adapted the source code given by Alon et al. [13] for the evaluating tasks.

6.2.2.4 Graph-Based Representation

Many types of graphs are associated with programs, such as control flow graph (CFG), control dependency graph (CDG), data dependency graph (DDG). Existing works utilized these information for various tasks [18–20, 31]. To investigate the impact of different types of graphs, we employ Code Property Graph (CPG) which is proposed by Yamaguchi et al. [212] by combining several graph representations, e.g., AST, CFG, CDG, and DDG, of source code into a graph structure. We selected three popular and widely used GNN variants i.e., Graph Convolutional Network (GCN), Graph Attention Network (GAT), and Gated Graph Neural Network (GGNN) to be our evaluating GNNs. These networks differ in their node message propagation, i.e., they aggregate and propagate information across the graph differently. GCN [213] uses first-order approximation of ChebNet [214]. The neighboring information of a node is aggregated using convolutional operation and layers of networks can be stacked to enhance the

learning of node representation. Velickovic et al. [215] propose GAT to use an attention mechanism in GNNs to attend over the neighborhood of a node to capture the local neighborhood information. GGNN [216] aggregates node information by the Gated Recurrent Unit [217] at every iteration to learn the node representations. For the above GNN variants, to obtain the graph-level representation, which can be considered as the program representation for different tasks, we employ the max-pooling operation over the learnt node representations.

6.2.2.5 Node Representation in Tree and Graph

To obtain the tree structure or the graph structure for a program, we use Joern [206], a tool that is widely used academically [212] and commercially [206], to transform a function. A simple example of the source code in Fig. 6.2 is presented in Fig. 6.3, where Fig. 6.3 (a) is the tree representation and Fig. 6.3 (b) is the graph representation. We can find that each node has its node type, which distinguishes the node from the others (the second line in each node), and code textual information, which is a small fragment code snippet from the original program (the first line in each node). To get the initial node representation of the graph and tree, which is a vector uniquely to represent each node for the model learning, we employ three different embedding methods: type embedding, textual embedding and hybrid embedding. In type embedding, we embed each node solely by its node type, i.e., embed the node by the unique vectors that represent each type of node. Each node type is input into a linear layer to learn a unique representation. Textual embedding learns an intermediate representation of the node by inputting the textual information of the node into a BiLSTM. We use the last hidden state of the BiLSTM as our initial node representation for the embedding. Hybrid embedding employs a linear layer to learn the concatenated representation of textual embedding and type embedding. Then these embedded node representations are utilized with Tree-LSTM or GNN variants respectively to learn the program representations.

6.3 Empirical Study

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

6.3.1 Experiment Settings

6.3.1.1 Experimental Setup

We embed each function into TF-IDF feature vector and input these vectors into the SVM, Naive Bayes, and XGBoost, resulting in a list of class probabilities. The class with the highest probability will be our prediction for the model. We employ Multinomial Naive Bayes [198] and Radial Basis Function Kernel for SVM [194]. For XGBoost [208], we trained our model with 40 rounds with a tree-depth of 32. We used the following hyper-parameters for LSTM, BiLSTM, and Transformer network: 128 for word embedding dimension and LSTM hidden size, 4 layers of LSTM/BiLSTM and transformer encoder, 4 heads for transformer attention layer. We used DGL [218] as our implementation for Tree-based and Graph-based approaches. Similarly, we used a hidden dimension of 128. For GCN, GAT, and GGNN, the following hyperparameters are used: 128 for word embedding and initial node representation, 4 layers for GAT/GCN/GGNN, and 4 attention heads for GAT. We train all our baselines with training data and tune the models based on the validation data. We then report the performance of the models using the testing data. To ensure fairness in the evaluation, we fixed all the dimensions to be 128 and uses a learning rate between 0.01 to 0.0001 for different tasks, batch size of 128, and a dropout rate of 0.2. All models are trained until 50 epoch and early stopping of 10 epoch. We limit the number of tokens in the function to be 150 and restrict the size of the graph to be within 250 nodes. For all deep learning experiments, we employ an additional linear classifier to learn on classifying the learnt representation into their respective classes. All experiments are conducted on an Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz Linux 16.04 server with 128GB RAM and equipped with three Tesla V100-SXM2-32GB graphic cards.

6.3.1.2 Evaluation Metrics

We adopt *accuracy* and *F1-score* as the evaluating metrics. Accuracy computes the correct prediction of each class and averages it with the total number of samples. F1-Score is commonly used in binary classification [12, 31, 204]. It is computed using a weighted combination of precision and recall. A high F1-score implies the model has a low number of false positives and false negatives. The higher values of these metrics, the better performance the approach achieves.

Table 6.2: Results of Code Classification, Vulnerability Detection and Clone Detection.

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

6.3.2 RQ1: Comparison of Different Code Representation

Table 6.2 shows the comparison results of our experiments. For this RQ, we employed hybrid embedding for tree representation and graph representation to learn the programs. We observed that among all the feature-based models, XGBoost performs the best across all three tasks, in contrast, Naive Bayes has the worst performance since the dependencies among the tokens in the program cannot be captured.

From the third row of Table 6.2, we can find that the sequence-based models have a better performance as compared to the feature-based approaches and BiLSTM performs the best across all three tasks, having accuracy in 83.82% in Code Classification, and F1-score of 71.62% in Vulnerability Detection and accuracy of 85.44% in Clone Detection. This indicates that learning dependency among the tokens in the program is important for learning program semantics and these tokens have the rich semantic information to represent the programs, which yields better performance compared with the feature-based approaches.

The tree-based approaches such as Tree-LSTM has the better performance over these tasks compared with the sequence-based approaches. Specifically, Tree-LSTM has an improvement of 2.18%, 0.47%, and 5.11% in accuracy and F1-score over BiLSTM, which shows that the

semantic and syntactic information on AST can be useful in learning the representation for the program, An interesting finding is that Code2Vec performs better than BiLSTM in both code classification and vulnerability detection, with an increase of 5.91% in code classification accuracy and 0.3% in f1-score of vulnerability detection. However, the performance in clone detection is lacking. We attribute the low performance to the different purposes of the code representation. The original purpose of code2vec is to predict method names in Java programs [13]. Hence, path contexts might be better for predicting function names but lacks in detecting clones.

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

Answer to RQ1: Graph-based representations are best in representing program semantic among all our comparing representations. We achieve improvements up to 64.42% accuracy in code classification, 8.62% F1-score in clone detection, and 30.37% F1-score in vulnerability detection when graph-based representations are used.

Insights and Directions: Graph-based representation is superior to the sequence-based or tree-based representation for many tasks. However, the construction of the graph for the program is non-trivial which requires extra efforts and this limits the usage of graph-based representation. It is crucial to have better tools to facilitate the code property graph construction for other programming languages such as Java and Python.

6.3.3 RQ2: Comparison of Different Node Embedding Information

In this RQ, we want to investigate the impact of different information that is embedded into the node representation for tree-based or graph-based representation. Specifically, we ablate the

Table 6.3: Results of Embedding Information.

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

performance that is embedded with type, textual, and hybrid embedding. Table 6.3 shows the results.

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

Furthermore, we can see that combining textual and type embedding i.e., hybrid embedding, will further improve the performance over code classification and clone detection tasks. For

instance, the performance of GGNN and Tree-LSTM increases by 3.65%, 2.20% when hybrid embedding is used on the code classification. We infer that incorporating both node type and node textual information can improve the model capacity and brings better performance. However, on the vulnerability detection, using type embedding has a negative impact. We conjecture that it is due to the way of combining embeddings. In the hybrid embedding, we just employ a linear layer to concatenate the representation of the textual embedding and type embedding, which is simple and straightforward. Vulnerability detection is a much-complicated task, especially for the real vulnerabilities. Although hybrid embedding obtains sub-optimal performance on the vulnerability detection, however, on the other tasks, it still gets the best performance. We will explore a more effective combining way and leave it as our future work.

Answer to RQ2: Textual information is more critical to learning the program semantics for the tasks as compared to the node type for the tree-based and graph-based representation. Furthermore, combining both with a simple linear layer, we can obtain the optimal performance on code classification and clone detection and sub-optimal performance on vulnerability detection.

Insights and Directions: The combination way i.e., a single linear layer is simple and straightforward. Although it generates a promising performance on code classification and clone detection, more tasks need to evaluate its generalization. Furthermore, it is also valuable to explore some other combination ways.

6.3.4 RQ3: Efficacy of Different Graph Representation

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

The results are shown in Table 6.4. We observe that CFG performs better than other graph representations in code classification and clone detection, however on the vulnerability detection, DDG can achieve higher performance. We believe it is reasonable, as data dependency is easy to trigger the vulnerabilities since vulnerable functions tend to have complex dependencies among the statements, e.g., memory de-referencing, and buffer overflows. Furthermore,

Table 6.4: Results of Graph Representation Analysis with GGNN.

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

Figure 6.4: Vulnerable function that required data dependency for the detection.

```
static void fix_bitshift(ShortenContext *s, int32_t *buffer)
{
    int i;
    if (s->bitshift != 0)
        for (i = 0; i < s->blocksize; i++)
            buffer[s->nwrap + i] <<= s->bitshift;
}
```

according to Fabian et al. [219], it is inherent that DDG has a bigger influence on finding vulnerable functions. We show an example for the illustration.

Fig 6.4 shows an example of a function [3] from FFmpeg, which corresponds to a out-of-bound (OOB) bug. GGNN predicted the correct label for this function with the DDG representation, which is shown in Fig 6.5. Specifically, we can see that the "buffer" array keeps increasing by adding the variable "i" without OOB guard. This results in the indexing of the array possibly growing out of the array size. In DDG, the relationship between the increment of "i" and the "buffer" indexing can be directly observed and there is a dependency between the nodes "ArrayIndexing" and "IncDecOp". However, this relationship is missed in other representations such as AST and CFG.

Furthermore, when all graph representations are combined i.e., CPG, it achieves better performance on code classification and clone detection and yields the sub-optimal results on vulnerability detection i.e., the accuracy of CPG (0.7158) is lower than DDG (0.7235), but F1 is still the highest (0.7344). It indicates that due to the different characteristics of the task, there might

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

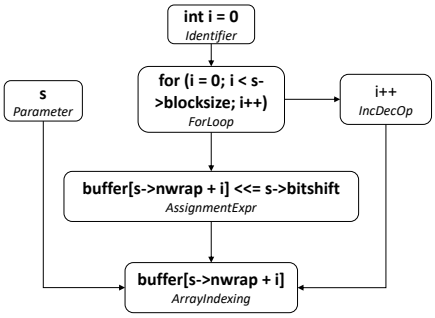


Figure 6.5: Data Dependency Graph of the function in Fig 6.4.

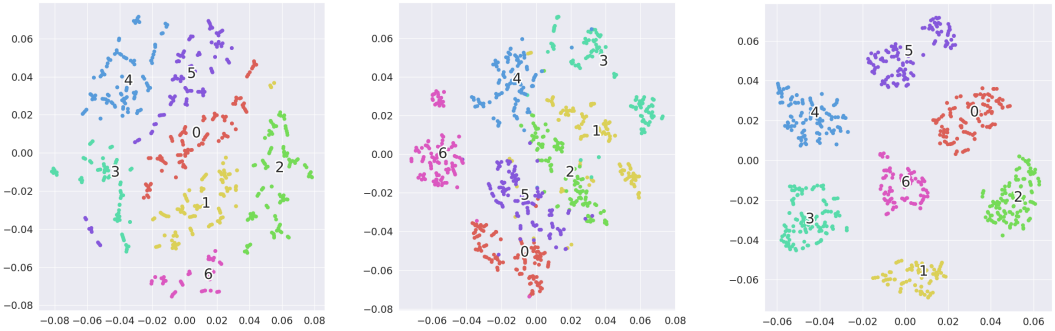
be some specific semantics that are particularly suitable for this task e.g., the data dependency for vulnerability detection and produce the best performance. However, a composite graph with immense semantic and syntactic information can still achieve promising results.

Answer to RQ3: Different task relies on the task-specific semantics to achieve the best performance, however, generally speaking, a composite graph with the comprehensive program semantics can yield the promising results.

Insights and Directions: Combining diverse dimensional code semantics, e.g., AST, CFG, CDG, DDG are beneficial for the neural networks to capture the program semantics, However, from one perspective, how to capture the task-specific semantics for a task to achieve the best results is still an open-question, from another perspective, CPG only contains syntactic information (AST), data flow information (CDG, DDG), control flow information (CFG), hence, is it sufficient to represent program semantics?

6.4 Discussion

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



(a) t-SNE Plot of BiLSTM Learnt Representation

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

(c) t-SNE Plot of GGNN Learnt Representation

Figure 6.6: t-SNE plot of Learnt Representation Space.

6.4.1 Learnt Representation Space by Neural Network

To demonstrate the learning capability of GGNN over BiLSTM and Tree-LSTM, we employ t-SNE [220] to visualize the learnt representation space of code classification. Specifically, we randomly picked 7 classes in the testing dataset. The learnt space is shown in Figure 6.6, where the class is labeled along with the color in the plot.

We can observe that the graph representation performs the best. As shown in Figure 6.6(a) and Figure 6.6(b), the representations learnt by BiLSTM are more scattered across the plot than the learnt space by Tree-LSTM, which means that the distances of any samples from the same class are greater in BiLSTM. Hence, compared with BiLST, Tree-LSTM produces a better learning space. The learnt space of GGNN is shown in Figure 6.6(c), we can easily find that the boundary for each class is more clear and the aggregated cluster is more condensed compared with Tree-LSTM. This indicates that GGNN has a more powerful learning capacity compared with BiLSTM and Tree-LSTM.

6.4.2 Semantic-Preserving Transformation on Code Representation

We further explore the capacity of these models on semantic-preserving operations. Here the semantic-preserving operation is defined to transform the code snippet with simple operations such as renaming the identifiers or swapping two independent statements while keeping the original program semantics.

Figure 6.7: Example 1 of Vulnerable Function with its transformed version.

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

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

Figure 6.8: Example 2 of Vulnerable Function with its transformed version.

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


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

Specifically, we randomly select two vulnerable functions [4] that BiLSTM, Tree-LSTM and GGNN predict correctly from the Devign test set and both examples are shown in Figure 6.7 and Figure 6.8 respectively where the top section shows the original function, while the bottom section are the transformed version. The original function in the top section of Figure 6.7 lacks a buffer overwrite protection, hence is exposed to buffer overflow vulnerability. For the function in the bottom section of Figure 6.8, a deprecated variable, "dc->no_user" is used. This might cause a regression bug where a previously working function stops working. We conduct two simple transformation operations: (1) We swap the location of two statements, that are independent with others. For instance, in Figure 6.7, we swap Line 4 and Line 5 in the original function. This does not affect any dependency as they are independent assertion statements and maintain the original program semantics. (2) We replace all occurrences of a variable name into a meaningless placeholder. For instance, in Figure 6.8, we replace all occurrence of "klass" with "k1". This operation also keeps the original semantics for the programs.

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

6.4.3 Analysis of Prediction Results

By Table 6.2, we have proved that the graph-based representation has the best performance across three tasks, to further analyze its capacity, we conduct the statistical analysis on the specific complex features in the program that GGNN cannot learn well for the code classification. Specifically, we manually examine the correct and incorrect predicted programs on the test set in code classification. We utilize the correct predicted programs (total 149) from the top-three best performing classes (Class 29, Class 66, Class 83) and incorrectly predicted programs (total

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

Table 6.5: Complex Features in Classified Programs.

Complex Features	Incorrect		Correct	
	Num	Ratio	Num	Ratio
Do-While	0	0%	3	1.84%
Multiple Loops	29	34.11%	58	35.58%
Nested Loops	25	29.41%	18	11.04%
Pointer Operation	17	20.00%	4	2.45%
Others/Basic	14	16.47%	80	49.07%
Total Features	85	-	163	-
Total Samples	63	-	149	-

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

The statistical results are shown in Table 6.5. Note that a program can have multiple defined categories, for example, a program can have both multiple loops and nested loops associated with it, hence the total number of features is greater than the total samples. We can find that the values for Nested Loops and Pointer Operation in the incorrect samples are higher than these values in the correct samples, which proves that these program features are not learnt well by the neural network. Furthermore, the number of samples with the simple structure i.e., Others/Basic category in the incorrect samples are far less than the samples in the correct samples, which illustrates that complex program structure is still a challenge for the neural network to learn semantics. Lastly, we cannot claim the learning capacity of the neural network on Do-While and Multiple Loops categories, since the values of the correct and incorrect samples are near.

6.5 Threats to Validity

6.5.1 Programming Language Limitation

Despite that we evaluate our empirical study on C Programm languages, the study can be also applied to other programming languages. C Programming language generally is difficult to learn, as compared to Java and Python, due to their memory operators and complex data types. This limits the neural network capabilities to learn semantics. However, our study can extend to other programming languages as long as similar graph representations can be built. Furthermore, our study does not includes languages-specific notions.

6.5.2 Evaluation Tasks

There are many other code-related tasks such as code translation [221], source code summarization [18], code search [25]. We only consider the classification tasks since these classification tasks are more suitable for quantitative analysis as compared to the generation tasks such as code summarization [18–20]. Our study may provide some ideas on how to evaluate the program semantics on complicated tasks.

6.5.3 State-of-the-Art Results

We did not compare with the state-of-the-art results for each task, since the approaches that achieve the highest performance tend to design more complicated architecture than these basic networks. For example, to achieve the best performance, devign [31] utilized a convolution module to further improve the learnt representations by GGNN. To reduce the complexity and difficulty of the analysis, we target the basic model and conduct a comprehensive and systematic study to explore different categories of code representations on code intelligent tasks, which we believe is fundamental and meaningful.

6.5.4 Hyper-Parameter Tuning

Hyper-parameters affect the performance of each model. For fairness in our study, we tune the hyper-parameters such as embedding dimension, batch size to obtain the best result for each evaluating task. We tune code classification based on the best accuracy, vulnerability detection, and clone detection based on the best f1-scores.

6.6 Conclusion

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

Chapter 7

Summary

7.1 Security Pipeline

With the ever-increasing size of open-source codebases, various code intelligent tasks have been proposed in an attempt to draw on this invaluable and extensive resource to solve various challenges, such as increasing automation or reducing manual labor. One prominent area that can be tackled by code intelligent tasks is Software Security. My thesis presents three solutions to code intelligent tasks: SPI, CORE, and Ratchet, which aims on enhancing software security through a data-driven approach. Each proposed solution can be employed sequentially to further enhance software security.

Our proposed solution on security patch identification, SPI, can be deployed outside of software development, continuously curate and enhance the security knowledge bases by gathering high-quality patches information. These patches and their corresponding information are useful for patch management and vulnerability management. Furthermore, these curated data could also aid the research community in future works on software security artifacts.

The code review process is conducted iteratively throughout software development. When there is a newly submitted code by a member of the development team, reviewers are required to review the code submission and approve or reject the submitted code. Hence, CORE can employ throughout the development process to enhance the quality of the codebase through automated code reviewing. Furthermore, by constantly giving early feedback to developers, it reduces the turnaround time for acceptance and subsequent revision of the submitted code.

Lastly, our proposed solution, Ratchet, can be utilized in two different ways: during vulnerability scanning or incorporated with CORE. Source codes are commonly scanned for vulnerability before major releases. This process ensures a minimum attack surface for the released software. During the vulnerability scanning process, Ratchet can repair any discovered bugs and faults, further patching up any vulnerability before releasing the software to their users. Another possible way of employing Ratchet is utilizing Ratchet together with CORE. With each code submission, Ratchet can be deployed to detect buggy statements and attempt to fix the buggy code submission before code reviewing is conducted.

In summary, my proposed solutions in this thesis complement each other, enhancing the security along the development pipelines. Security cannot be ensured by a single process but it can be enhanced and facilitated through consistent measures and defenses to ensure that your software is free from common implementation bugs and errors. Despite that my thesis covers many different areas of the software developments process, there are still many code intelligent tasks that can be implemented and added to the security pipeline. For instance, we can employ deep-learning in a vulnerability detection stage, where we employ deep neural network in finding security bugs and vulnerabilities in open-source project.

7.2 Future Works

There is great potential in applying deep learning techniques in software engineering and security. In this section, we discuss the possible future works that can stem from our research in this thesis. Specifically, we discuss the prospective work of code review recommendation, security patch identification, automated program repair, and source code representations

7.2.1 Code Review Recommendation

As mentioned in Chapter 3, CORE employs bi-directional LSTM and multi-level embedding network in learning representation of the source code and review. We can extend the neural network that learns the relevancy between source code and review. For instance, large-scale language models are famous for their few-shots and zero-shot learning capability. Extensions to the neural network design, such as incorporating large-scale language models (e.g., CodeBERT [36] and GraphCodeBERT [186]), can be considered possible future works. Further-

more, we can formulate code review recommendation into a generative problem through large-scale language model.

On the other hand, supplementary contexts of the review can be provided as input to our neural networks. For instance, a subsequent work by Tufano et al [222] employed the changed functions as input to neural network, providing more useful context to generating code reviews. Another possible source of information would be information on the pull-request thread can be used as additional contexts for learning the relevancy of source code and review. Furthermore, reviews are commonly posted sequentially. This infers that the previous review could provide useful information in predicting the subsequent review, hence, we can employ this information in future designs.

7.2.2 Security Patch Identification

Our work, SPI, employs LSTM and CNN in learning to identify security patches. Furthermore, SPI employs sequential representation (i.e., token-based representation) in its design. SPI can be further extended by replacing the sequential representation with graph representation. Specifically, we can employ graph representation to represent the source code instead of sequential representation as described in Chapter 4. Using graph representation can potentially increase our performance as our empirical study shows that it is better at representing source code. To further extend the work, we can explore the different representations of commit messages (e.g., parse tree of sentences) and design novel neural networks to ensure that both commit messages and source code can contribute equally to identifying security patches. One prominent example would be VulFixMiner by Zhou et al. [223] where they employed pre-trained language models to identify silent vulnerability fix. Similarly, Ghadhab et al. [224] fine-tuned BERT on the commit classification. Both works show the potential in employing pre-trained language models in finding security patches in open-source projects.

Many subsequent works [225, 226] supports the notions that security artifacts are valuable to the domain and they can be beneficial to researchers. Therefore, we can further extend our work to tackle other security artifacts (e.g., security-critical commits and security keywords). Sabetta et al. [227] propose an approach that mines security-relevant commits through machine learning techniques. We can further apply our deep learning models and approaches to perform the same security-relevant task.

7.2.3 Automated Program Repair

There are several issues in the domain of program repair [181], such as differentiating between syntax bugs and logic bugs, having different evaluating metrics for repair models, and repair models over-fitting to test cases. For instance, several works [24, 154] evaluate their models based on the rate of successful compilations, while some works [23, 61] evaluate themselves based on the ratio of successful passing of test-case. Furthermore, there are different approaches to program repair, such as heuristic-based approaches and constraint-based approaches. To further extend my study in this field, we can first conducts a study on evaluating the benefits and limitations of these approaches. Eventually, we can build an evaluation framework to eliminate unfair comparisons and standardize evaluation for program repair.

7.2.4 Source Code Representation

As mentioned previously, our empirical study on code representations (Section 6) serves as a foundation for future works. With meaningful code representation, the semantics of the source code can be learnt effectively, allowing better performance across code intelligent tasks. Our empirical study concludes that graph representation is an effective representation that comprises of strong semantic property of source code. Potential future works can be built upon tree or graph representation, allowing downstream tasks to be performed effectively.

One great example would be GraphCodeBERT [186] where Guo et al. employs the data flow feature in learning a large-scale language model for source code representation. However, there are other possible ways of extending the work. For instance, one potential work would be incorporating a composite graph representation, instead of employing only a data-flow graph, into learning the graph language model.

7.3 Conclusion

In this thesis, we propose three solutions to common code intelligent tasks: CORE, Ratchet, and SPI. We designed our solution for automated code review, CORE, which aims on raising the quality of source code and increasing the productivity of developers. CORE employs a deep-learning model to learn the relevance between source code and reviews. It further suggests relevant reviews based on the submitted source code during the inference phase. Our

work benefits developers in reducing the manpower required for code review and increasing the quality of the codebase.

To enhance the mining and curation of security patches, we designed and implemented our Security Patch Identifier (SPI) that employs a data-driven approach and deep learning neural network. The main objective of SPI is to identify security patches amidst large open-source projects, facilitating better curation and management of security patches. Our work benefits both the research community and the software developers community. For the research community, our work extends previous studies on identifying security patches and provides an extensive dataset of security patches that can be employed for future research works. Our work also provides software developers with a patch management support that can deploy for the software development life cycle.

Ratchet employs a transformer model in learning to generate patches for buggy statements. It consists of two main components, RatchetFL for fault localization and RatchetPG for patch generation. With our proposed retrieval-augmented transformer model, Ratchet produces better patches and fixes more security bugs than comparing baselines.

To inspire potential research directions and build the foundation for my future works, I perform an empirical study on various code representations and neural networks. This empirical study demonstrates the strong capability of graph neural networks and graph representation for code intelligent tasks, further highlighting that new and innovative representations are emerging due to the trend of big code.

Software developments are closely tied with software security, hence, my work presents an complete solution to the three stages of vulnerability during development phases, Vulnerability Prevention, Vulnerability Remediation and Patch Management, working towards the common goal of enhancing and facilitating software security through data-driven approaches.

Bibliography

[1] FIREEYE. Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor. [Online]. Available: https://www.mandiant.com/resources/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor

[2] Solarwinds. New findings from our investigation of sunburst. [Online]. Available: https://orangematter.solarwinds.com/2021/01/11/new-findings-from-our-investigation-of-sunburst/

[3] C. for Applied Internet Data Analysis. (2020) Caida analysis of code-red. [Online]. Available: https://www.caida.org/archive/code-red/

[4] Redscan, "A redscan report nist security vulnerability trends in 2020: an analysis," 2020.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 492–501. [Online]. Available: https://doi.org/10.1145/1134285.1134355

[6] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 281–293. [Online]. Available: https://doi.org/10.1145/2635868.2635883

[7] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, 2020.

[8] VeraCode, "State of software security," 2020.

[9] G. McGraw, "Software security: Building security in," in *2006 17th International Symposium on Software Reliability Engineering*, 2006, pp. 6–6.

[10] Nvd - home. [Online]. Available: https://nvd.nist.gov/

[11] Cve security vulnerability database. security vulnerabilities, exploits, references and more. [Online]. Available: https://www.cvedetails.com/

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

[13] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018.

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

[15] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin, "Integrating tree path in transformer for code representation," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=70Q_NeHImB3

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

[17] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

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

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

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

[21] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *ACL (short)*, 2020.

[22] W. Tao, Y. Wang, E. Shi, L. Du, S. Han, H. Zhang, D. Zhang, and W. Zhang, "On the evaluation of commit message generation models: An experimental study," *CoRR*, vol. abs/2107.05373, 2021. [Online]. Available: https://arxiv.org/abs/2107.05373

[23] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 3–13.

[24] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 10 799–10 808. [Online]. Available: http://proceedings.mlr.press/v119/yasunaga20a.html

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

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

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

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

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

[30] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," *CoRR*, vol. abs/1605.08535, 2016. [Online]. Available: http://arxiv.org/abs/1605.08535

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

[32] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *25th Annual Network and Distributed System Security Symposium (NDSS 2018)*, 2018.

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

[34] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[35] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, no. null, p. 1137–1155, mar 2003.

[36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[37] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[38] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

[39] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 202–211. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597082

[40] Gerrit open-source code review tool. [Online]. Available: https://www.gerritcodereview.com/

[41] Review Board code review tool review bot. [Online]. Available: https://www.reviewboard.org/

[42] A. Gupta and N. Sundaresan, "Intelligent code reviews using deep learning," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[43] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, 02 2011.

[44] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: identifying vulnerable code for vulnerability assessment through program metrics," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 60–71. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00024

[45] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *CoRR*, vol. abs/1807.06756, 2018. [Online]. Available: http://arxiv.org/abs/1807.06756

[46] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, ser. IJCAI'19. AAAI Press, 2019, p. 4665–4671.

[47] Mozilla. Bugzilla main page. [Online]. Available: https://bugzilla.mozilla.org/home

[48] Github. [Online]. Available: https://github.com/

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

[50] Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo, "A machine learning approach for vulnerability curation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 32–42. [Online]. Available: https://doi.org/10.1145/3379597.3387461

[51] G. Bhandari, A. Naseer, and L. Moonen, *CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software*. New York, NY, USA: Association for Computing Machinery, 2021, p. 30–39. [Online]. Available: https://doi.org/10.1145/3475960.3475985

[52] Y. Chen, A. E. Santosa, A. Sharma, and D. Lo, "Automated identification of libraries from vulnerability data," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 90–99. [Online]. Available: https://doi.org/10.1145/3377813.3381360

[53] Y. Prabhu and M. Varma, "Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 263–272. [Online]. Available: https://doi.org/10.1145/2623330.2623651

[54] C. Perrin. The danger of complexity: More code, more bugs. [Online]. Available: https://www.techrepublic.com/blog/it-security/the-danger-of-complexity-more-code-more-bugs/

[55] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 60–70. [Online]. Available: https://doi.org/10.1145/3180155.3180219

[56] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[57] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

[58] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[59] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 298–312. [Online]. Available: https://doi.org/10.1145/2837614.2837617

[60] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 532–543. [Online]. Available: https://doi.org/10.1145/2786805.2786825

[61] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.

[62] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2017.45

[63] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3241980

[64] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *CoRR*, vol. abs/1810.01791, 2018. [Online]. Available: http://arxiv.org/abs/1810.01791

[65] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.

[66] N. Jiang, T. Lutellier, and L. Tan, "CURE: code-aware neural machine translation for automatic program repair," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1161–1173. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00107

[67] M. Xiangxin, W. Xu, Z. Hongyu, S. Hailong, and L. Xudong, "Improving fault localization and program repair with deep semantic features and transferred knowledge," ser. ICSE '22, 2022.

[68] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[69] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 408–419.

[70] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 486–497.

[71] C. Bird and A. Bacchelli, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering*. IEEE, May 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/expectations-outcomes-and-challenges-of-modern-code-review/

[72] M. Greiler, C. Bird, M.-A. Storey, L. MacLeod, and J. Czerwonka, "Code reviewing in the trenches: Understanding challenges, best practices and tool needs," Tech. Rep. MSR-TR-2016-27, May 2016.

[73] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "The review linkage graph for code review analytics: A recovery approach and empirical study," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 578–589. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338949

[74] T. Ahmed, A. Bosu, A. Iqbal, and S. Rahimi, "Senticr: A customized sentiment analysis tool for code review interactions," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 106–111.

[75] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," in *AAAI*, 2019.

[76] M. Madera and R. Tomoń, "A case study on machine learning model for code review expert system in software engineering," in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 2017, pp. 1357–1363.

[77] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, "An empirical assessment of security risks of global android banking apps," in *Proceedings of the 42st International Conference on Software Engineering*. IEEE Press, 2020, pp. 596–607.

[78] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu, "Are mobile banking apps secure? what can be improved?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 797–802.

[79] S. Chen, G. Meng, T. Su, L. Fan, M. Xue, Y. Xue, Y. Liu, and L. Xu, "Ausera: Large-scale automated security risk assessment of global mobile banking apps," *arXiv preprint arXiv:1805.05236*, 2018.

[80] C. Hannebauer, M. Patalas, S. Stünkel, and V. Gruhn, "Automatically recommending code reviewers based on their expertise: An empirical comparison," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970306

[81] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 931–940.

[82] P. Anderson, "The use and limitations of static-analysis tools to improve software quality," *CrossTalk-Journal of Defense Software Engineering*, vol. 21, 06 2008.

[83] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Proceedings of the 8th International Symposium on Software Metrics*, ser. METRICS '02. USA: IEEE Computer Society, 2002, p. 249.

[84] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[85] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[86] X. Zhang and Y. LeCun, "Text understanding from scratch," *CoRR*, vol. abs/1502.01710, 2015. [Online]. Available: http://arxiv.org/abs/1502.01710

[87] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[88] "Pull Request github pull request," https://help.github.com/en/articles/about-pull-requests, accessed: 2019-10-05.

[89] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *CoRR*, vol. abs/1802.05365, 2018. [Online]. Available: http://arxiv.org/abs/1802.05365

[90] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[91] Pygments pygments parser. [Online]. Available: http://pygments.org/

[92] J. Li, Y. Wang, I. King, and M. R. Lyu, "Code completion with neural attention and pointer networks," *CoRR*, vol. abs/1711.09573, 2017. [Online]. Available: http://arxiv.org/abs/1711.09573

[93] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 3975–3981. [Online]. Available: https://doi.org/10.24963/ijcai.2019/552

[94] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: ACM, 2019, pp. 2727–2735. [Online]. Available: http://doi.acm.org/10.1145/3292500.3330699

[95] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 200–210. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196334

[96] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, Nov 1997.

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

[98] Solarwinds. Pytorch pytorch - from research to production. [Online]. Available: https://pytorch.org/

[99] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," *arXiv preprint arXiv:1909.06727*, 2019.

[100] R. Feng, S. Chen, X. Xie, L. Ma, G. Meng, Y. Liu, and S.-W. Lin, "Mobidroid: A performance-sensitive malware detection system on mobile platform," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 61–70.

[101] nltk nltk. [Online]. Available: https://www.nltk.org

[102] B. Hu, Z. Lu, H. Li, and Q. Chen, "Convolutional neural network architectures for matching natural language sentences," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2042–2050. [Online]. Available: http://papers.nips.cc/paper/5550-convolutional-neural-network-architectures-for-matching-natural\-language-sentences.pdf

[103] J. Weston, C. Wang, R. J. Weiss, and A. Berenzweig, "Latent collaborative retrieval," *CoRR*, vol. abs/1206.4603, 2012. [Online]. Available: http://arxiv.org/abs/1206.4603

[104] D. Sharma and D. Garg, "Information retrieval on the web and its evaluation," *CoRR*, vol. abs/1209.6492, 2012. [Online]. Available: http://arxiv.org/abs/1209.6492

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

[106] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," 11 2014.

[107] D. R. Radev, H. Qi, H. Wu, and W. Fan, "Evaluating web-based question answering systems," in *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC'02)*. Las Palmas, Canary Islands - Spain: European Language Resources Association (ELRA), May 2002. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2002/pdf/301.pdf

[108] E. Voorhees and D. Tice, "The trec-8 question answering track evaluation," *Proceedings of the 8th Text Retrieval Conference*, 11 2000.

[109] A. Severyn and A. Moschitti, "Learning to rank short text pairs with convolutional deep neural networks," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '15. New York, NY, USA: ACM, 2015, pp. 373–382. [Online]. Available: http://doi.acm.org/10.1145/2766462.2767738

[110] J. Ramos, "Using tf-idf to determine word relevance in document queries," 01 2003.

[111] W. P. Ramadhan, S. T. M. T. A. Novianty, and S. T. M. T. C. Setianingsih, "Sentiment analysis using multinomial logistic regression," in *2017 International Conference on Control, Electronics, Renewable Energy and Communications (ICCREC)*, Sep. 2017, pp. 46–49.

[112] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing & Management*, vol. 24, no. 5, pp. 513 – 523, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0306457388900210

[113] S. E. Ahmed, "Effect sizes for research: A broad application approach," *Technometrics*, vol. 48, no. 4, p. 573, 2006.

[114] Frequency of token deletesnapshot. [Online]. Available: https://github.com/search?q=DeleteSnapshotListener&type=Code

[115] SNYK. The state of open source security - 2019. [Online]. Available: https://snyk.io/opensourcesecurity-2019/

[116] GitHub. The state of octoverse. [Online]. Available: https://octoverse.github.com/

[117] Open Sourced Security and License Management - 2020. [Online]. Available: https://www.whitesourcesoftware.com/

[118] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2017, pp. 2201–2215.

[119] Veracode. TConfidently secure apps you build and manage with Veracode. [Online]. Available: https://www.veracode.com/

[120] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=SJeqs6EFvB

[121] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[122] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting "0-day" vulnerability: An empirical study of secret security patch in oss," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019, pp. 485–492.

[123] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

[124] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. AAAI Press, 2016, pp. 1606–1612. [Online]. Available: http://dl.acm.org/citation.cfm?id=3060832.3060845

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

[126] L. Bianconi. mt76: fix possible null pointer dereferencing in mt76x2_mac_write_txwi(). [Online]. Available: https://github.com/torvalds/linux/commit/98051872fd25077d3b106ab3d2b945fa7025c1ef

[127] Null Pointer Dereference. [Online]. Available: https://www.owasp.org/index.php/Null_Dereference

[128] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[129] J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.

[130] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," 2015.

[131] Y. Kim, "Convolutional neural networks for sentence classification," 2014.

[132] J. Wang, Z. Wang, D. Zhang, and J. Yan, "Combining knowledge with deep convolutional neural networks for short text classification." in *IJCAI*, 2017, pp. 2915–2921.

[133] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*. Valencia, Spain: Association for Computational Linguistics, Apr. 2017, pp. 1107–1116. [Online]. Available: https://www.aclweb.org/anthology/E17-1104

[134] Uninitialized Variable. [Online]. Available: https://www.owasp.org/index.php/Uninitialized_variable

[135] Buffer Overflow. [Online]. Available: https://www.owasp.org/index.php/Buffer_Overflows

[136] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *CoRR*, vol. abs/1807.06756, 2018. [Online]. Available: http://arxiv.org/abs/1807.06756

[137] M. Abadi, A. Agarwal, P. Barham, and et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[138] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 426–437.

[139] CVE-2017-7187. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-7187

[140] CVE-2017-8063. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-8063

[141] (2017) Silently (or obliviously) partially-fixed CONFIG_STRICT_DEVMEM bypass. [Online]. Available: http://www.openwall.com/lists/oss-security/2017/04/16/4

[142] CVE-2010-5329. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2010-5329

[143] CVE-2015-8952. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2015-8952

[144] lu zero. FFMpeg Security Patch Example. [Online]. Available: https://github.com/FFmpeg/FFmpeg/commit/61cd19b8bc32185c8caf64d89d1b0909877a0707

[145] Buffer Overreads. [Online]. Available: https://cwe.mitre.org/data/definitions/126.html

[146] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

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

[148] Github, "The 2020 state of the octoverse," Github, Report P-19, 2020.

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

[150] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 467–477. [Online]. Available: https://doi.org.remotexs.ntu.edu.sg/10.1145/581339.581397

[151] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 702–713.

[152] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[153] M. Tufano, C. Watson, G. Bavota, M. di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 832–837.

[154] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 1345–1351.

[155] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 102–113.

[156] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 378–381.

[157] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[158] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 153–162.

[159] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Softw. Test. Verif. Reliab.*, vol. 25, no. 5–7, p. 605–628, aug 2015. [Online]. Available: https://doi.org/10.1002/stvr.1509

[160] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "Ifixr: Bug report driven program repair," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 314–325. [Online]. Available: https://doi.org/10.1145/3338906.3338935

[161] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 306–317. [Online]. Available: https://doi.org/10.1145/2635868.2635898

[162] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*. ACM, 2014, pp. 492–495. [Online]. Available: https://doi.org/10.1145/2591062.2591114

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

[164] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid GNN," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=zv-typ1gPxA

[165] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. L. Traon, "A closer look at real-world patches," in *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2018, pp. 275–286. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00037

[166] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 2019, pp. 1–12.

[167] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[168] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1073–1085. [Online]. Available: https://doi.org/10.1145/3377811.3380342

[169] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[170] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.

[171] F. F. Xu, Z. Jiang, P. Yin, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," in *Annual Conference of the Association for Computational Linguistics*, 2020.

[172] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[173] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[174] A. Lucene. Apache - welcome to apache lucene. [Online]. Available: https://lucene.apache.org/

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

[176] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 180–182.

[177] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 117–128. [Online]. Available: https://doi.org/10.1145/3106237.3106255

[178] C. L. Goues, N. J. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of C programs," *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2015.2454513

[179] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance*. San Jose, California, USA: IEEE, 2000, pp. 120–130.

[180] K. Pan, S. Kim, and E. J. W. Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[181] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302156

[182] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 262–273. [Online]. Available: https://doi.org/10.1145/2970276.2970359

[183] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[184] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 2021, pp. 661–673.

[185] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 14–24.

[186] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," *CoRR*, vol. abs/2009.08366, 2020. [Online]. Available: https://arxiv.org/abs/2009.08366

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

[188] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[189] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 913–923.

[190] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 91–105. [Online]. Available: https://doi.org/10.1145/3385412.3385997

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

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

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

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

[195] L. Vanneschi and R. Poli, *Genetic Programming — Introduction, Applications, Theory and Open Issues*, G. Rozenberg, T. Bäck, and J. N. Kok, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. [Online]. Available: https://doi.org/10.1007/978-3-540-92910-9_24

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[222] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 163–174.

[223] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 705–716.

[224] L. Ghadhab, I. Jenhani, M. W. Mkaouer, and M. Ben Messaoud, "Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model," *Information and Software Technology*, vol. 135, p. 106566, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000495

[225] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, *CrossVul: A Cross-Language Vulnerability Dataset with Commit Data*. New York, NY, USA: Association for Computing Machinery, 2021, p. 1565–1569. [Online]. Available: https://doi.org/10.1145/3468264.3473122

[226] X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia, "Patchdb: A large-scale security patch dataset," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 149–160.

[227] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2018.