AN AUTOMATED SOFTWARE VULNERABILITY DETECTION TECHNIQUE USING VULNERABILITY REPORTS

RABAYA SULTANA MIM
Class Roll: MSSE 1730
Exam Roll: 2323203
Registration Number: 2021-116-287
Session: 2022-2023

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

© RABAYA SULTANA MIM, 2023

AN AUTOMATED SOFTWARE VULNERABILITY DETECTION TECHNIQUE USING VULNERABILITY REPORTS

RABAYA SULTANA MIM

Approved:

Signature *Date*

Supervisor: Toukir Ahammed

To *my beloved parents,*
who have always been there for me and inspired me with all the support, sacrifice, and blessings.

Abstract

As software programs increase in size and complexity, the occurrence of vulnerabilities has become a significant security threat. Manual vulnerability detection is time consuming and challenging because vulnerabilities can potentially be dispersed throughout the source code. To reduce manual effort, automatic detection techniques have been adopted which often suggest potentially vulnerable source code files or functions. However, these techniques often underutilize vulnerability reports that provide crucial contextual information for identifying specific vulnerable statements within the source code. As a result, a fine-grained approach, such as statement-level vulnerability detection is necessary for accurately identifying and fixing new vulnerabilities.

Recent vulnerability detection approaches utilize either text-based or graph-based analysis on source code. However, combining these methods and incorporating vulnerability reports for detection has not been fully explored. This thesis proposes a statement-level vulnerability detection technique where statements are extracted from function-level detection. The research focuses on two groups: one integrates text and graph analysis to identify vulnerable functions and the other minimizes the source code search space using vulnerability reports to rank probable vulnerable functions, subsequently ranking vulnerable statements.

Although Deep Learning (DL) methods are efficient,previous studies have faced difficulties in preserving both detection accuracy and scalability using either text or graph methods. To address this challenge, an automated software **Vul**nerability

detection technique using pre-trained Code**BERT** model and **C**onvolutional **N**eural **N**etwork named VulBertCNN is proposed which explores the potential of integrating both approaches. VulBertCNN employs a pre-trained CodeBERT model for textual analysis and converts the source code into images for graphical analysis using a convolutional neural network. Comparative analyses show that VulBertCNN significantly outperforms state-of-the-art techniques, improving accuracy from 78.6% to 95.7% and F1 measure from 62.6% to 89%. This demontrates the effectiveness of our integrated approach in identifying vulnerabilities within large-scale source code.

Based on the result of the function level detection, to more narrow down the detection in statement level vulnerability reports are used. Existing studies have underutilized vulnerability reports which can provide crucial contextual information for identifying vulnerable functions and statements. To address this, an information retrieval-based approach named **V**ulnerable **F**unctions and **S**tatements **Detector** (VFSDetector) is proposed aimed at detecting vulnerable functions and particular vulnerable statements within source code which need fixes. VFSDetector first ranks vulnerable functions by calculating similarity scores between vulnerability reports and source code using a tweaked Vector Space Model (tVSM). Then, it constructs a Function Statement Dependency Graph (FSDG) for each suggested vulnerable functions. A relationship is established between each node and its predecessor nodes of FSDG by constructing a Predecessor Node Dependency Graph (PNDG) to rank vulnerable statements based on similarity scores. The effectiveness of this methodology is evaluated by testing 1200 vulnerability reports from 14 open-source projects, VFSDetector accurately ranks genuine vulnerable statements in 1^{st} position in 40% of cases and within the top 20 up to 100% of cases, outperforming baseline methods with maximum accuracy, precision, and recall of 70.43%, 42.37%, and 95.81%. Hence, developers can utilize these findings to promptly implement effective patches on vulnerable statements.

Acknowledgments

"All praises are due to Allah"

Firstly, I express my gratitude to Allah for giving me the opportunity and granting me the ability to continue my research work properly.

I want to express my appreciation and utmost respect to my supervisor, Lecturer Toukir Ahammed and co-supervisor Dr. Kazi Muheymin-Us-Sakib, Institute of Information Technology, University of Dhaka. Without their motivation, support, effort and guideline, this thesis could not be successful.

I would like to appreciate the faculty and thesis committee members of Institute of Information Technology, University of Dhaka for their valuable feedback of every thesis presentation, which improved my thesis from scratch till date.

I am greatly indebted to my parents who have always supported and motivated me to reach my goal. I would also like to appreciate my classmates for their support. I extend my gratefulness towards the members of Distributed Systems and Software Engineering (DSSE) research group for their knowledgeable insights and constructive criticism on my work.

I am also thankful to the Ministry of Posts, Telecommunications and Information Technology, Government of the Peoples Republic of Bangladesh for granting me ICT Fellowship No - 56.00.0000.052.33.001.23-09; Dated: 04.02.2024. Last but not least, my special thanks to the Bangladesh Research and Education Network (BdREN) for providing me with virtual machine to carry out my thesis.

List of Publications

1. "Impact of Centrality on Automated Vulnerability detection using Convolutional Neural Network." in *Proceedings of the 2nd International Conference on Information and Communication Technology for Sustainable Development (ICICT4SD)*, pp. 331-335, 2023.

2. "Identifying Vulnerable Functions from Source Code using Vulnerability Reports" in *Proceedings of the 11th International Workshop on Quantitative Approaches to Software Quality (QuASoQ) co-located with 30th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 66-73, 2023.

3. "Automated Software Vulnerability Detection Using CodeBERT and Convolutional Neural Network" in *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pp. 156-167, 2024.

4. "Automated Software Vulnerability Detection in Statement Level using Vulnerability Reports" in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pp. 454-455, 2024.

Contents

Approval ii

Dedication iii

Abstract iv

Acknowledgements vi

List of Publications vii

Table of Contents viii

List of Tables xi

List of Figures xii

Listings xiii

1 Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Research Questions . . . 6
- 1.3 Contribution and Achievement . . . 10
- 1.4 Organization of the Thesis . . . 13

2 Background Study 16
- 2.1 Concepts Related to Software Vulnerability . . . 17
  - 2.1.1 Software Vulnerability . . . 17
  - 2.1.2 Causes of Software Vulnerabilities . . . 18
  - 2.1.3 Impact of Software Vulnerabilities . . . 20
  - 2.1.4 Vulnerability Report . . . 23
  - 2.1.5 Software Vulnerability Detection . . . 26
  - 2.1.6 Vulnerability Fixing . . . 26
  - 2.1.7 Program Syntax . . . 26
  - 2.1.8 Program Semantics . . . 27
  - 2.1.9 Abstract Syntax Tree (AST) . . . 28
  - 2.1.10 Program Dependency Graph from Source Code . . . 29
- 2.2 Source Code Analysis Concepts . . . 32
  - 2.2.1 Static Analysis of Source Code . . . 32

2.2.2 Dynamic Analysis of Source Code . . . 33
2.3 Steps of Text Retrieval Based Vulnerability Detection . . . 35
2.3.1 Source Code Corpora Generation . . . 36
2.3.2 Query Generation . . . 38
2.3.3 Indexing . . . 38
2.3.4 Ranking and Retrieval . . . 39
2.4 Summary . . . 43

3 Literature Review 44
3.1 Existing Approaches . . . 45
3.2 Text Based Approach . . . 46
3.2.1 VulDeePecker . . . 47
3.2.2 μVulDeePecker . . . 48
3.2.3 TokenCNN . . . 50
3.2.4 LineVul . . . 51
3.2.5 COSTA . . . 52
3.3 Graph Based Approach . . . 54
3.3.1 VulCNN . . . 55
3.3.2 Devign . . . 56
3.3.3 IVDetect . . . 57
3.3.4 LineVD . . . 58
3.4 Summary . . . 60

4 Combination of Text and Graph Based Approach in Function Level Vulnerability Detection 62
4.1 Introduction . . . 63
4.2 Methodology of the Proposed System . . . 65
4.2.1 Graph Generation . . . 65
4.2.2 Feature Extraction . . . 68
4.2.3 Image Construction . . . 70
4.2.4 Vulnerability Classification . . . 76
4.3 Experiment and Result Analysis . . . 79
4.3.1 Implementation . . . 80
4.3.2 Datasets . . . 80
4.3.3 Evaluation Metrics . . . 81
4.3.4 Detection Performance Evaluation . . . 82
4.3.5 Baseline Methods Comparison . . . 85
4.4 Threats to Validity . . . 87
4.5 Summary . . . 88

5 A Statement Level Vulnerability Detection Technique using Vulnerability Reports 90
5.1 Introduction . . . 91
5.2 Methodology . . . 93
5.2.1 Function Level Vulnerability Detection by Minimized Search Space using Vulnerability Reports . . . 94
5.2.1.1 Source Code Corpora Generation . . . 95

5.2.1.2 Vulnerability Report Corpora Generation 97
5.2.1.3 Ranking Vulnerable Functions 97
5.2.2 Statement Level Vulnerability Detection using Vulnerability Reports 100
5.2.2.1 Function Statement Dependency Graph (FSDG) Generation 102
5.2.2.2 Node Predecessor Dependency Graph (PNDG) Generation 105
5.3 Experiment and Result Analysis 108
5.3.1 Dataset 108
5.3.2 Implementation 108
5.3.3 Evaluation Metrics 109
5.3.4 Result Analysis in Function Level 111
5.3.5 Result Analysis in Statement Level 113
5.3.6 Baseline Methods Comparison 118
5.4 Threats to Validity 120
5.5 Summary 121

6 Conclusion **122**
6.1 Impact of Combining Text and Graph Based Approach in Function Level Vulnerability Detection 123
6.2 Impact of using Vulnerability Reports in Function and Statement Level Vulnerability Detection 125
6.3 Future Directions Related to this Thesis 127
6.4 Future Directions Related to the Domain 129

A Appendix **130**
A.1 Stop Words 130
A.2 Programme Language Specific Keywords 132

Bibliography **133**

List of Tables

4.1 Hyper-Parameter Settings In VulBertCNN 78
4.2 Details of Datasets 80
4.3 Evaluation of VulBertCNN using CodeBert Embedding 86

5.1 List of Initially Analyzed Open Source Projects in Function Level . 112
5.2 Summary of Tested Projects in Function Level 113
5.3 Initially Tested Software Systems Tested for Statement Level Detection 114
5.4 The performance of VFSDetector Tested on 1200 Vulnerabilities . . 115
5.5 Vulnerability Detection Performance Comparison with State-of-the-art Approaches 118

List of Figures

2.1 Vulnerability Report of CVE-2021-3456 [1] . . . 25
2.2 Example of Progran Syntax . . . 27
2.3 Syntax Error . . . 27
2.4 Symantic Error in Source Code . . . 28
2.5 Abstract Syntax Tree . . . 29
2.6 Dynamic Analysis of Source Code by Function Calling . . . 34
2.7 Latent Semantic Indexing [2] . . . 39
2.8 Vector Space Model [3] . . . 40
2.9 Latenet Dirichlet Allocation Model [4] . . . 41
2.10 Representation of using LDA Model [5] . . . 42

3.1 Generic workflow of Vulnerability Detection Model . . . 45

4.1 Overview of the Proposed Automated Vulnerability Detection System in Function Level . . . 66
4.2 Steps of Source Code Normalization . . . 67
4.3 The Program Dependency Graph (PDG) and corresponding adjacency matrix of a vulnerable function . . . 71
4.4 The Program Dependency Graph (PDG) and corresponding adjacency matrix of a vulnerable function . . . 72
4.5 Cumulative Distribution Function of total number of lines of code of a function . . . 77
4.6 CNN Classification of VulBertCNN . . . 78
4.7 Loss Curve with respect to Number of Epochs . . . 79
4.8 Comparison of the detection accuracy for different centralities using CodeBert embedding and CNN on SARD dataset . . . 83
4.9 Maximum vulnerability detection performance in VulBertCNN with CodeBert embedding and different centralities . . . 84

5.1 Overview of VFSDetector in Function Level . . . 94
5.2 Overview of VFSDetector in Statement Level . . . 101
5.3 Function Statement Dependency Graph of *determineProfit* Function 104
5.4 Predecessor Node Dependency Graph of FSDG in Figure 5.3 . . . 106
5.5 Detection Performance of VFSDetector Across Different Projects . . 116
5.6 Comparison of Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) Across Various Software Projects . . . 117
5.7 Performance Comparison of Top 20 Vulnerability Detection Metrics for Different Baseline Approaches . . . 119

Listings

2.1 Vulnerable Source Code of CVE-2019-10092 18
5.1 Source Code for Generating FSDG 103

Chapter 1

Introduction

A software vulnerability refers to a flaw within a program that attackers exploit to compromise security, raising concerns regarding integrity, availability, and confidentiality [6]. Such vulnerabilities render software systems increasingly susceptible to attacks and damage which intensifies security concerns [7]. In 2023, the Open Source Security and Risk Analysis (OSSRA) conducted an extensive study encompassing 1703 codebases with audit data, revealing that 76% of the codes were open source. Notably, 48% of the codebases displayed high-risk vulnerabilities, with 84% of these vulnerabilities linked to open source security flaws. Consequently, to enhance software security, it is crucial to employ advanced techniques for detecting vulnerabilities on a large scale. Automatic software vulnerability detection serves as a method to identify vulnerable locations within the source code with minimal manual intervention. Developers often invest significant time in finding vulnerable statements within extensive codebases when new vulnerabilities arise. Effectively identifying these functions and statements is essential for ensuring software security. Accurate identification of these vulnerabilities can significantly reduce developer effort by pinpointing the precise locations along with providing information about the vulnerability which requires fixes. Existing techniques either use text or graph based approach to suggest files or functions as

vulnerable which need lots of developer effort to find the actual vulnerable statements. Hence, an accurate statement level vulnerability detection technique is needed.

In the realm of software development, errors and anomalies frequently arise as a consequence of programming flaws leading to unexpected outcomes. Upon encountering such unexpected behavior users typically reports detailing the vulnerability, facilitating its prompt resolution. These reports are channeled to developers through vulnerability tracking systems and are available in vulnerability database like [8] and [9]. Subsequently, developers engage in the task of pinpointing the vulnerable statements within the source code from these vulnerability reports. Existing studies have overlooked the potential of vulnerability reports to provide contextual insights for detecting software vulnerabilities. This study investigates the role of these reports in identifying vulnerable functions and statements by analyzing shared keywords between the statements and the report. Manual localization of vulnerabilities demands a significant investment of time. Consequently, the duration required for vulnerability rectification is substantially prolonged. To address this issue, an automatic technique is necessary to quickly identify where the vulnerable statements are in the code, aimed at diminishing the overall effort required in vulnerability resolution by swiftly identifying the faulty statements within the codebase.

Vulnerability detection can be categorized into two types according to granularity level: coarse-grained and fine-grained. Coarse-grained vulnerability detection examples include identifying vulnerabilities at the file or function levels, while fine-grained detection involves finding particular vulnerabilities at the statement level. The primary aim of statement level vulnerability detection is to locate vulnerabilities at the most granular level possible, thereby minimizing the time required for localization. Based on this concept, this chapter initially presents the rationale behind the need for granular-level vulnerability detection in software.

Subsequently, it outlines the research questions and provides a brief overview of the contributions. Finally, the chapter concludes with an outline of the thesis organization.

1.1 Motivation

Software vulnerabilities have long been an inherent challenge for programmers, emerging from the inception of a program. When a vulnerability arises, developers address it by identifying and rectifying the affected areas during the vulnerability remediation phase. However, pinpointing these vulnerable spots within the source code is a formidable task, given that a program typically comprises millions of lines of code. For instance, one vulnerability report CVE-2013-6763 of Linux Kernel version 3.12.1 contains total 273,898 functions. Detecting software vulnerabilities at the statement level serves several key motivations:

- **Enhancing Security:** The primary motivation is to enhance the security of software systems. By identifying vulnerabilities at the statement level, developers can pinpoint specific lines of code that are susceptible to exploitation. This allows for targeted fixes to mitigate security risks, reducing the likelihood of successful attacks.

- **Reducing Attack Surface:** Vulnerability detection at the statement level helps in reducing the overall attack surface of software applications. By identifying and fixing vulnerabilities at a granular level developers can eliminate potential entry points for attackers making it more challenging for them to exploit the system.

- **Compliance Requirements:** Many industries and regulatory bodies have strict compliance requirements regarding software security. Detecting vulnerabilities at the statement level enables organizations to demonstrate com-

pliance with security standards and regulations, such as PCI DSS, HIPAA, or GDPR, which mandate the implementation of security measures to protect sensitive data.

- **Improving Code Quality:** Identifying vulnerabilities at the statement level also contributes to improving the overall quality of the codebase. By addressing vulnerabilities early in the development process, developers can adopt best practices for secure coding and reduce technical debt, resulting in more robust and maintainable software.

- **Preventing Financial Loss:** Vulnerabilities in software can lead to financial losses due to data breaches, system downtime, or regulatory fines. Detecting and fixing vulnerabilities proactively at the statement level helps prevent potential financial losses associated with security incidents, as organizations can avoid the costs associated with remediating breaches and reputational damage.

- **Maintaining Trust:** Maintaining trust with customers, partners, and stakeholders is crucial for businesses. By demonstrating a commitment to security through proactive vulnerability detection and mitigation at the statement level, organizations can instill confidence in their products and services, thereby preserving trust and credibility in the market.

- **Staying Ahead of Threats:** The threat landscape is constantly evolving, with new attack vectors and techniques emerging regularly. Detecting vulnerabilities at the statement level allows organizations to stay ahead of evolving threats by continuously monitoring and updating their software to address newly discovered vulnerabilities and emerging security risks.

- **Software Aging:** Software Aging, similar to human aging, refers to the process whereby software becomes outdated. Aging software is typically built

using older technologies, which may become obsolete when new advancements emerge. This aging process poses difficulties in manually identifying precise vulnerabilities locations, consequently inflating the software's cost [10].

- **Insufficient Documentation:** In numerous software industries, comprehensive documentation is not consistently upheld and they don't always keep good records. So, developers might not know exactly what the vulnerabilities are, making it harder to fix them and taking more time to maintain the software [11].

- **Developer Transition:** Developers might switch jobs between different industries. This means that new developers might join a company without knowing everything about how the software works. So, they might not be able to handle any vulnerability properly because they don't have enough knowledge about them [12].

- **Others:** Sometimes, the Quality Assurance team (called QA) might not have enough skills. This can lead to really poor quality bug reports, which might confuse developers trying to find where the software vulnerabilities are [13]. Another reason is the source code having unstructured and inappropriate names. In real life, new developers might use wrong names in the source code, breaking the right design standards and making it hard to find the real vulnerable spots in the big source code [13]. (This can also make automatic vulnerability finding techniques tricky.)

Overall, vulnerability detection at the statement level is essential for enhancing security, reducing risk, ensuring compliance, improving code quality, and maintaining trust in software systems. According to statistics, finding where software vulnerabilities are located takes up 60-70% of the effort needed to fix them [13]. This can lead to higher costs for the software. However, using an automated

vulnerability detection technique can reduce these costs by cutting down on the amount of work developers need to do to fix vulnerabilities.

Automatic vulnerability detection finds where the vulnerabilities are without needing any manual input. In research, vulnerabilities are usually found at either the file level, class level or function level. File level detection gives a list of files where vulnerabilities might be. This means there's a big list of possible places for vulnerabilities, and developers have to go through each file to find the vulnerable statements. This takes a lot of time because files often have many functions and many statements. For example, if 10 files are suggested as vulnerable and each file has around 900 statements, and the actual vulnerable statement is in the 7th file on the list, developers would need to look through about 900x7 = 6300 statements to find it.

Function level vulnerability localization finds a list of functions where vulnerabilities might be, but again manual searching through the source code is needed to find the actual vulnerable statements. This means more work for developers.

A more precise way to find vulnerable spots is by statement level detection starting from the reported vulnerability. This can save money compared to the other two levels (file and function) because developers only need to look through the suggested statements to find the vulnerable ones as described in vulnerability reports. Unfortunately, there's currently no technique for statement level vulnerability detection using vulnerability reports. So, we still need to develop an accurate way to find vulnerabilities at this level using the reported vulnerability description.

1.2 Research Questions

Taking into account the issues mentioned earlier, the goal of this research is to create a detailed vulnerability-finding method, like finding vulnerabilities at the

statement level. To achieve this, we need to start with broader techniques, like finding vulnerabilities at the file or function level. However, current techniques for file or function level vulnerability finding look at the entire source code, which can make them less accurate because they include unnecessary information. So, in this research, we'll first make function-level vulnerability-finding more accurate by suggesting precise functions and filtering out irrelevant code. Then, we'll use these more accurate functions to find vulnerability at the statement level. To do this, we need to answer the following research questions:

RQ: How does the combination of vulnerability report and source code impact in statement level vulnerability detection?

The following Sub Questions (SQs) will be used to answer the research question.

1. **SQ1: What is the impact of combining text-based and graph-based methods on function-level vulnerability detection?**

   Accurate statement level vulnerability detection is possible if proper function level vulnerability detection can be ensured. Existing approaches have either used text or graph based approaches between vulnerable code and potential fixes. By integrating text and graph based approaches, the detection of vulnerabilities may be enhanced because it incorporates both syntactic and semantic information of source code, thereby elevating the detection of correct vulnerabilities. Hence in our proposed method VulBertCNN both text based and graph based approach is integrated to improve function level vulnerability detection

2. **SQ2: What is the impact of combining vulnerability report with source code in minimizing solution search space?**

   Existing software vulnerability detection technique commonly follow either source code static analysis or dynamic analysis or the combination of both. However, none of those minimize the search space rather these consider the

irrelevant information of the source code during static analysis. Hence, here in our proposed method VFSDetector search space reduction is proposed which introduces two further research questions.

- How can we filter out unnecessary details from the source code?

  To narrow down the search area, we can focus only on the parts of the code related to the specific vulnerability. This involves looking at how the code runs and using that information to identify which parts are relevant. For removing irrelevant information from the large solution search space, only vulnerability specific source code can be taken by considering source code execution traces. For example: Code comments are irrelevant with the vulnerability because code comments do not contains any relevant information from which vulnerability can be detected.

- How do we compare vulnerability reports with the source code to see if they match?

  To pinpoint where the vulnerable code are, we use a tweaked Vector Space Model (tVSM) to measure how similar vulnerability reports are to the source code. We do this by processing the vulnerability reports and source code texts using various techniques, like removing common words, getting rid of programming language-specific terms, identifying multi-word phrases, understanding the meaning of the words, and reducing words to their root form.

3. **SQ3: What is the impact of combining vulnerability report with source code in improving performance and developing an effective statement level vulnerability detection technique?**

   In this phase, the suggested vulnerable functions returned by answering the previous question can be used. Since a statement contains very few infor-

mation about a vulnerability, the information of the probable vulnerable methods can be taken from vulnerability reports. Here, the way of increasing valid information of a statement is another question that needs to be answered. For this, a vulnerable statements ranking approach that integrates syntactic and semantic similarities with source code and report is proposed. This also brings up two more research questions, which are as follows:

- How can we get more useful and valid information from a single statement?

  A single line of code often doesn't tell us much about a bug. For example, a vulnerability report CVE-2023-52286 might be related to line 35[1], but that line alone might not give us enough information. So, to get more useful info, we can look at the lines of code before it (the "predecessor nodes") that might affect it. For example, if we have a = 7; b = 1; p = "a + b", we can see that "p = a + b" depends on both "a = 7" and "b = 1". So, combining these predecessor lines helps us understand "p = a + b" better because more relevant information significantly enhance the understanding of the vulnerability's root cause and potential scenarios.

- How do we compare statements with vulnerability reports?

  To see how similar each statement in the code is to the vulnerability reports, we can use something called the Vector Space Model (VSM). Then, we measure how much each statement is like each vulnerability report and we rank the statements based on the similarity score which indicates how closely each statements match the reports. Thus to understand the impact of combining report and code similarities in improving performance, the proposed approach can be compared with

[1]https://github.com/Narrator21/tdsql/blob/main/20230927.md

other vulnerability detection techniques that use either semantic or syntactic similarity. For comparison, these approaches can be executed on a well-known vulnerability dataset that contains diverse projects. Since the goal is to rank the correct vulnerabilities higher among all the generated vulnerabilities, the results can be inspected using vulnerability ranking related metrics. Finally, this SQ aims to find a ranked list of vulnerable statements.

1.3 Contribution and Achievement

This thesis introduces two automated vulnerability detection technique one is in function level and another in statement level. Existing techniques either use text or graph based approach to suggest files or functions as vulnerable. Our approach integrates both text and graph based relationship of source code and vulnerability reports similarities to tackle the research question discussed earlier. Initially, an empirical investigation is conducted to assess the effects of incorporating both types of similarities. Drawing insights from this empirical study, the proposed approaches integrate both text and graph analysis of source code in function level and vulnerability report in statement level to effectively navigate the correct statement from potentially incorrect ones. In summary, the key contributions of this thesis are as follows:

- **Analyzing the impact of a hybrid approach combining both text and graph based analysis in function level vulnerability detection:** To the best of our knowledge, the impact of combining syntactic and semantic information and creating a hybrid approach with both text based and graph based approach has not been explored yet. As a result this thesis proposes an enhanced automated vulnerability detection method using CodeBERT and Convolutional Neural Network called VulBertCNN. It uses

CodeBERT [14] embedding model leveraging the advantages of text based approach by large-scale pre-training which captures both syntactic and semantic information of source code. Our method consist of three phases. First, using the source code as input, we create a Program Dependency Graph (PDG) containing data flow and control flow information. In the second phase, CodeBERT embedding is applied to each node of the generated PDG. In the third phase, we utilize different combinations of five centralities on each node, with each centrality corresponding to a channel, to create an image. Finally, a Convolutional Neural Network (CNN) [15] model is trained on the produced images to identify functions as either vulnerable or non-vulnerable.

To ensure the model's effectiveness in large scale vulnerability scanning, pre-trained CodeBERT embedding model with various centrality combinations are explored and we applied the proposed model on two benchmark datasets. Our experimental results demonstrate a noteworthy improvement in accuracy from 78.6% to 95.7% and F1 measure increasing from 62.6% to 89% which is a significant increase of 21.7% and 26.3%. This underscores the effectiveness of our approach in detecting vulnerabilities in large-scale source code. Hence, developers can employ these findings to promptly apply effective patches on vulnerable functions.

- **Analyzing the impact of using vulnerability reports in function and statement level vulnerability detection:** In this thesis, initially this study proposes a technique of automatic software vulnerable function identification namely VFSDetector. It takes all source code files of a system as input. First, it extracts all source code functions of that system. Then static analysis is performed to extract the contents of those functions. Several text preprocessing analysis such as tokenization, stopwords removal,

multiword splitting, semantic meaning extraction and lemmatization are applied on these source code along with the vulnerability report to produce code and report corpora. In addition, programming language specific keywords are removed for generating code corpora. Finally, to rank the vulnerable functions, similarity scores are measured between the code corpora of the functions and report corpora by a modified version of Vector Space Model (tVSM) where larger methods get more weight while ranking. In our experiments, we choose Common Vulnerabilities and Exposures (CVE) reports from 14 open source projects (Openssl, Wireshark, FFmpeg, Kuberos, Radare2, Curl, Php Interpreter, Libarchive, LibRaw, Tcpdump, ImageMagick, Chrome, Android, Linux) which are available in GitHub repositories. Based on the commit link available in reports we crawled the corresponding projects before the vulnerability was patched. Experimental results show that our proposed technique identifies 40% of vulnerabilities at 1st position and 50% in Top 5 rank and 90% reports are ranked in Top 7.

In the next phase, we proposed a statement-level vulnerability detection technique. Here, we used the suggested probable vulnerable functions to minimize the search space. For each suggested vulnerable function, we prepared a Function Statement Dependency Graph (FSDG). Following this, each statement and its immediate predecessor nodes are combined to enhance the valid and relevant information for each node. Consequently, a Node Predecessor Dependency Graph (NPDG) was generated. Each node's text was then processed using various techniques, and the text was compared with the vulnerability report using the Vector Space Model (VSM) to measure similarity. Finally, a ranked list of probable vulnerable statements was produced based on similarity scores.

To demonstrate the results of the statement-level vulnerability detection technique, we used those 14 projects which were used in function level. The

obtained results show that our proposed system achieves minimum 22% Mean Reciprocal Rank (MRR) and 20% Mean Average Precision (MAP) for Linux, and a maximum of 66% MRR and 61% MAP for Tcpdump respectively. The details of the proposed approaches and their evaluation are presented in Chapter 5.

1.4 Organization of the Thesis

This section provides an overview of the subsequent chapters. The chapters are organized in the following way:

- **Chapter 2 Background Study:** This chapter establishes the foundation for comprehending automated software vulnerability detection. It begins by introducing key concepts related to software vulnerabilities and associated terms, such as vulnerable source code, patches, commits etc. Subsequently, it delves into terminologies associated with source code, including statements, expressions, and abstract syntax trees,their classification and the various types of program analysis employed for detecting software vulnerabilities. Additionally, it will delve into the general procedures involved in text retrieval-based vulnerability detection. Finally, the chapter concludes by discussing the significance, steps, and challenges of automated vulnerability detection, offering insights into this critical aspect of software development.

- **Chapter 3 Literature Review on Automated Software Vulnerability Detection:** In this chapter, existing vulnerability detection techniques are categorized into two main groups: function level and statement level approaches. This classification serves as a framework for discussing the methodology, strengths, and weaknesses of current automated vulnerability detection methods. An extensive review of existing state-of-the-art vulnerability detection techniques in both function and statement level are

presented. The chapter also includes the limitations of current vulnerability detection approaches, which motivates the development of a novel bug localization approach.

- **Chapter 4 Combination of Text and Graph Based Approach in Function Level Vulnerability Detection:** This chapter presents the impact of integrating text and graph based information of source code in a hybrid approach and we aim to identify the vulnerable functions. By integrating text and graph based approaches, the detection of vulnerabilities is enhanced because it incorporates both syntactic and semantic information of source code. Initially, the chapter outlines the methodology employed in the study. Subsequently, it delves into the implementation specifics, evaluation criteria, and analysis of results. Following this analysis, potential threats to the validity of the study are identified and discussed. Finally, the chapter concludes with a summary of the study's findings.

- **Chapter 5 A Statement Level Vulnerability Detection Technique using Vulnerability Reports:** In this chapter, a novel statement-level vulnerability detection method is proposed. This approach exclusively focus on the probable vulnerable functions identified in Chapter 4. This chapter begins by outlining the methodology of the approaches. Subsequently, it delves into implementation specifics, detailing experimentation datasets and evaluation metrics. The discussion then moves on to examine the results from the study. After this analysis, possible threats to the study's validity are identified and explored. The chapter wraps up with a summary of the study's main findings.

- **Chapter 6 Conclusion:** This chapter provides a comprehensive summary of the entire thesis, encapsulating its key findings and contributions. Additionally, future directions of research are presented, offering potential direc-

tions for further exploration and development.

Chapter 2

Background Study

In software development, encountering a series of errors is common, but it requires significant effort to address these errors and also to overcome the vulnerability. Initially, developers manually identify the locations of these vulnerabilities, a process that consumes considerable time. It's important to note that addressing vulnerabilities usually takes up 50% to 80% of the overall development and maintenance effort with over half of this effort dedicated to manually locating the vulnerable areas in the source code [16]. However, automatic software vulnerability localization techniques offer a promising solution by providing vulnerable locations without the need for human intervention, thereby reducing maintenance effort [17], [18].

To provide a comprehensive understanding of vulnerability localization, this chapter begins by introducing an overview and classification of software vulnerabilities. Subsequently, various categories of source code analysis are explored, as program analysis plays a crucial role in automatic vulnerability detection. Finally, a general procedure for software vulnerability detection is presented, outlining the steps involved in this important process.

2.1 Concepts Related to Software Vulnerability

Vulnerabilities are very common in software evolution which are the defects that can be caused by errors in the software's coding or the way it was created. To detect the system is first scanned by the attacker to determine if it contains any software vulnerabilities. By scanning the system, an attacker can gather information about the installed software, determine whether it is up-to-date, and identify any vulnerabilities in the software packages. This information enables attackers to better understand the potential types of attacks they can initiate against the system. If an attack is successful, the attacker can execute malicious commands on the target system, potentially rendering the software inoperable due to defects. The details of the vulnerabilities are discussed below, followed by an observation of its impact. Additionally, a brief description of the methods for detecting software vulnerabilities is provided

2.1.1 Software Vulnerability

Software vulnerabilities pose an increasing risk to software systems making them susceptible to attacks and potential damage, thereby raising security concerns[6]. Gaps in software that can give hackers access to a system are called software vulnerabilities. Software vulnerabilities refer to weaknesses or flaws present in code. These vulnerabilities typically arise from issues, defects, or weaknesses within the software itself [19], [20]. Attackers exploit these vulnerabilities to carry out various malicious activities such as stealing or tampering with sensitive data, connecting systems to botnets, installing backdoors, or injecting other forms of malware. Moreover, once attackers gain access to a network host, they can leverage it to infiltrate other hosts within the same network. Most security vulnerabilities are caused by errors or mistakes made by developers in program sources code. To fix these vulnerabilities, software costs increases which also involves a large scale

efforts of the developers. Listing 2.1 is an example of a software vulnerability.

Listing 2.1: Vulnerable Source Code of CVE-2019-10092

```
1 import os
2 def delete_file(file_name):
3     os.system("rm " + file_name) //Vulnerable statement
4 file_to_delete = input("Enter the file name to delete: ")
5 delete_file(file_to_delete)
```

In this code, the vulnerability exists in line 5 in the delete_file function. The os.system function is used to execute shell commands, and the file_name parameter is directly concatenated with the command string without proper validation. This can lead to command injection vulnerability, allowing an attacker to execute arbitrary commands on the system.

For example, if an attacker inputs file.txt; ls as the file_to_delete, the resulting shell command would be rm file.txt; ls, which would not only delete file.txt but also list the contents of the directory, potentially exposing sensitive information. This vulnerability is commonly known as Command Injection [21].

2.1.2 Causes of Software Vulnerabilities

Vulnerabilities Software vulnerabilities are caused mainly by two reasons: programming error or design flaws [22]. Flaws in program design, such as those found in login functions, can signal potential vulnerabilities. However, even with flawless design, vulnerabilities can still exist if there are bugs present in the program's source code. Coding errors have the potential to introduce various types of vulnerabilities, including:

- **Buffer overflows:** This permits exceeding the designated capacity of an input field, enabling the inclusion of more data than the field is intended to

accommodate. An attacker can capitalize on this by inserting malevolent commands into the overflow data field, which are subsequently executed.

- **SQL injection:** Through SQL injection, attackers can insert harmful code into a web application's database. They achieve this by inputting carefully crafted SQL commands into the data fields of web forms or directly into the URLs of the web application. If successful, unauthorized individuals can access and potentially compromise or destroy data stored in the database.

- **Third Party Libraries:** Lots of programmers opt for third-party code libraries instead of building everything from the ground up. It's a big time-saver, but it can be risky if the library has vulnerabilities. Developers need to make sure these libraries are free from vulnerabilities before integrating them into their projects.

- **Application Programming Interfaces:** APIs, which enable software programs to communicate with each other, can also introduce software vulnerabilities. Numerous application programming interfaces (APIs) lack stringent security measures, potentially enabling unauthorized access for hackers to breach systems.

- **Security Misconfiguration:** Security misconfiguration is one common problem in software development which results from incomplete configurations and insecure default configurations. For example, there are misconfigurations in open cloud storage. To avoid software security vulnerabilities of this kind, we need to ensure that we have configured our operating system, frameworks, and applications correctly. Likewise, all of these should be updated as needed.

- **Broken Access Control:** Broken user restrictions can be caused due to severe software weaknesses. For example, if our website has an admin panel

then we will want to restrict the accessibility of that particular area. If these restrictions are not implemented properly, unauthorized people and hackers can easily exploit these vulnerabilities and access sensitive data or even control our system.

- **Exposure of Sensitive Data:** Sensitive data includes usernames, account numbers, financial information, addresses, health information, and passwords. All these data needs to be secured from going to the wrong hands. Encryption should be done to prevent the access of unauthorized people. Hackers can gain access to these sensitive data if the development of the software contains any security vulnerabilities [23].

Overall, the impact of the vulnerability depends on various factors such as the attacker's intentions, the environment in which the vulnerable application operates, and the security measures in place to mitigate the risk. However, given the nature of buffer overflow vulnerabilities, they are typically considered severe and require immediate attention and remediation to prevent exploitation and mitigate potential damage.

2.1.3 Impact of Software Vulnerabilities

The impact of software vulnerabilities can vary widely depending on several factors, including the nature of the vulnerability, the context in which it is exploited, and the security measures in place to mitigate its effects. Here's a breakdown of high impact and low impact vulnerabilities:

1. **High Impact Vulnerabilities:** High impact vulnerability refers to a security flaw or weakness in software or systems that, if exploited, can lead to significant adverse consequences such as data breaches, system compromise, denial of service (DoS) attacks, regulatory compliance violations, or cyber espionage. These vulnerabilities pose a severe risk to the confidentiality,

integrity, and availability of data and systems, as well as the overall security posture of organizations. They often require immediate attention and mitigation efforts to prevent exploitation and mitigate potential damage.

- **Data Breaches**: Vulnerabilities that allow unauthorized access to sensitive data, such as personally identifiable information (PII), financial records, or trade secrets, can result in significant data breaches. The impact can include financial losses, damage to reputation, legal consequences, and loss of customer trust.
- **System Compromise**: Vulnerabilities that enable attackers to gain unauthorized access to systems or execute arbitrary code with elevated privileges can lead to complete system compromise. This can result in the theft or destruction of data, disruption of critical services, and significant downtime for affected organizations.
- **Denial of Service (DoS)**: Vulnerabilities that allow attackers to exploit resource exhaustion or amplification techniques can lead to widespread service disruptions, rendering systems unavailable to legitimate users. High-profile DoS attacks can result in significant financial losses, damage to reputation, and disruption of business operations.
- **Regulatory Compliance Violations**: Vulnerabilities that result in data breaches or the compromise of sensitive information can lead to violations of regulatory requirements like the General Data Protection Regulation (GDPR), Health Insurance Portability and Accountability Act (HIPAA), or Payment Card Industry Data Security Standard (PCI DSS). Such violations can incur fines, penalties, and legal consequences for organizations found to be non-compliant.
- **Cyber Espionage and Sabotage**: Vulnerabilities exploited by sophisticated threat actors, such as nation-state actors or advanced per-

sistent threats (APTs), can lead to cyber espionage or sabotage. This may involve theft of intellectual property, disruption of critical infrastructure, or manipulation of data for political or economic gain.

2. **Low Impact Vulnerabilities:** Low impact vulnerability, on the other hand, refers to a security flaw or weakness in software or systems that, if exploited, may result in relatively minor or localized adverse consequences with limited impact on operations, data security, or organizational reputation. While these vulnerabilities may still pose security risks and require remediation, their potential impact is generally less severe compared to high impact vulnerabilities. Examples include minor data leakage, limited service disruptions, reputation damage, or opportunistic attacks by less sophisticated threat actors.

    - **Minor Data Leakage**: Vulnerabilities that result in the exposure of non-sensitive or low-value information, such as system configuration details or non-critical application logs, may have minimal impact on affected organizations. While data leakage can still pose privacy concerns and undermine trust, the overall impact may be relatively low compared to high-profile data breaches.
    - **Limited Service Disruption**: Vulnerabilities that result in temporary service disruptions or degraded performance, particularly for non-critical systems or services, may have a limited impact on business operations. While such disruptions can still cause inconvenience and potential financial losses, the overall impact may be manageable with appropriate incident response measures.
    - **Reputation Damage**: Even vulnerabilities with relatively low direct impact on operations or data security can still result in reputational damage for affected organizations. Perception of poor security prac-

tices or failure to address vulnerabilities promptly can erode customer trust and confidence in products or services, leading to long-term consequences for business viability.

- **Opportunistic Attacks**: Vulnerabilities that are actively exploited by opportunistic attackers, such as script kiddies or automated malware, may result in isolated incidents of compromise or exploitation. While the impact on individual victims may be low, the cumulative effect of widespread exploitation can still pose a significant risk to overall cybersecurity posture.

2.1.4 Vulnerability Report

A vulnerability report is a structured document that outlines the details of a security vulnerability discovered in a software application, system, or network. It typically includes information such as the vulnerability name, affected system components, severity level, potential impact, recommendations for mitigation, references to relevant resources, and a disclosure timeline indicating when the vulnerability was discovered and reported. Vulnerability reports play a crucial role in cybersecurity by providing actionable insights to software developers, system administrators, and security professionals, enabling them to address and mitigate security risks effectively. Here is an example of vulnerability report which is given below:

1. Vulnerability Details:

    - **Vulnerability Name:** CVE-2022-3456: Heap-Based Buffer Overflow in Linux Kernel Bluetooth Stack
    - **Affected System:** Linux Kernel
    - **Affected Component:** Bluetooth Stack
    - **Affected Versions:** 3.3 - 5.11

- **Severity:** High
- **CVSS Score:** 7.8
- **CVE ID:** CVE-2022-3456

2. **Description:** A heap-based buffer overflow vulnerability was discovered in the Linux kernel's Bluetooth stack. This vulnerability can be exploited by an attacker within Bluetooth range to send a specially crafted Bluetooth Low Energy (BLE) packet, leading to the execution of arbitrary code or a denial of service condition.

3. **Impact:** Successful exploitation of this vulnerability may allow an attacker to:

   - Execute arbitrary code with kernel privileges.
   - Crash the affected system, leading to a denial of service condition.
   - Potentially escalate privileges and gain control over the affected system.

4. **Recommendations:** To mitigate this vulnerability, it is recommended to:

   - Apply the vendor-supplied patch or update to the latest kernel version.
   - Disable Bluetooth functionality if not needed.
   - Implement network segmentation to limit exposure to potential attackers.
   - Monitor Bluetooth traffic for signs of exploitation.

5. **References:**

   - Linux Kernel Security Mailing List: `https://lkml.org/`
   - CVE-2022-3456: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-3456`

6. **Disclosure Timeline:**

- **Discovery Date:** September 30, 2022
- **Reported Date:** October 5, 2022
- **Vendor Contacted:** October 5, 2022
- **Vendor Response:** Patch released on October 15, 2022

This report details a high-severity vulnerability in the Linux kernel's Bluetooth stack, provides recommendations for mitigation, and includes references for further information. Additionally, it outlines the disclosure timeline indicating when the vulnerability was discovered, reported, and patched. Figure 2.1 is an example of a software vulnerability.

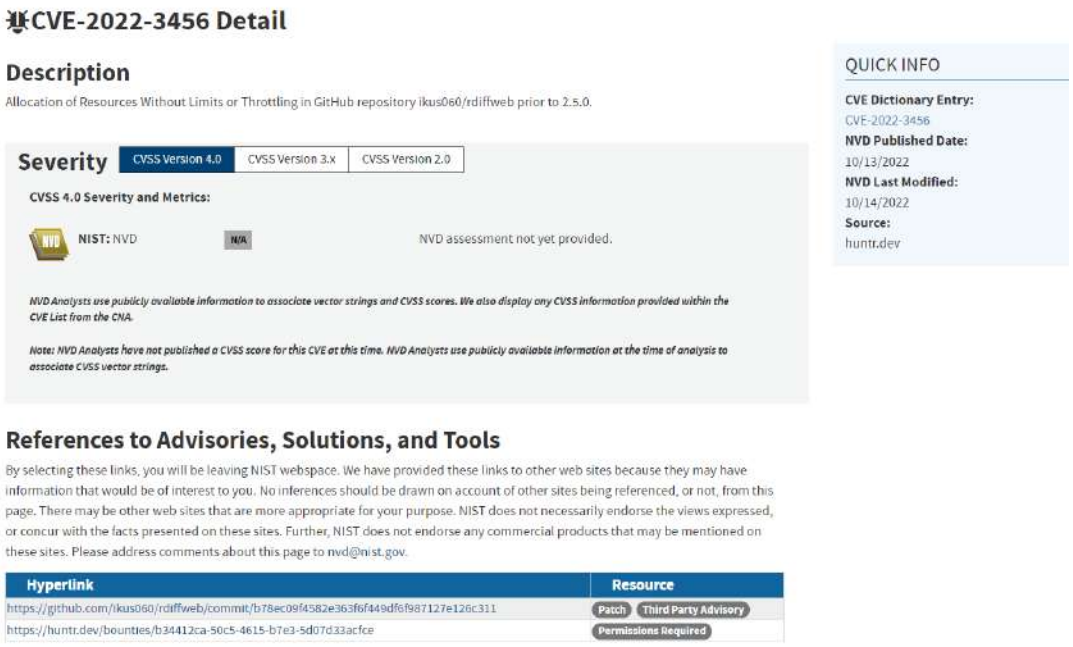


Figure 2.1: Vulnerability Report of CVE-2021-3456 [1]

Figure 2.1 shows that a report contains description, severity rating,references to advisories, solutions and tools and also the reference links which the developers can use to fix the vulnerability.

2.1.5 Software Vulnerability Detection

Software Vulnerability detection is the process of finding particular vulnerable statements in software rather than just pinpointing in which file specific vulnerability are. It's quite challenging because software can be really big, with lots of code, features and statements [24]. Plus, over time, software can become harder to understand due to things like outdated documentation, developers leaving, or not having enough information about reported vulnerabilities. Also, if the code isn't named and organized well, it makes finding vulnerabilities even harder[17], [25].

Automatic vulnerability detection helps reduce the significant time spent during the software maintenance phase, resulting in faster vulnerability fixing. In many cases, locating these vulnerabilities takes up at least half of the total maintenance effort. More information about automated software vulnerability detection will be discussed in later sections.

2.1.6 Vulnerability Fixing

Vulnerability fixing involves making changes to a software product to permanently resolve a vulnerability. Typically, it begins with either QA teams or users reporting issues encountered during software use. Developers then utilize their domain knowledge and information from vulnerability reports to address the reported vulnerability. This process generally consists of two steps: identifying the vulnerable areas and implementing fixes.

2.1.7 Program Syntax

In practical syntax refers to the set of rules which defines the arrangement of characters for making a statement valid [26]. In programming language syntax holds the rules of that particular programming language structure, punctuations,

brackets etc. In programming language whether it is C/C++, Java, Python each have specific schemas which need to be followed for developing code. A syntax error occurs when an element in a statement is misplaced preventing successful communication [27]. Example- C language have a set of rules for writing code , if we do not follow the rules and miss any punctuation or commands the program will not compile and we will get error. After running the command in Figure 2.2 is an example of a software vulnerability.

```
#include <stdio.h>

main()
{
  printf("Syntax Error!")
}
```

Figure 2.2: Example of Progran Syntax

We will get the following output as in Figure 2.3 we get an error which is an example of a syntactic error.

```
main.c: In function 'main':

main.c:5:27: error: expected ';' before '}' token
```

Figure 2.3: Syntax Error

2.1.8 Program Semantics

Semantics means the meaning of the statement written in any programming language [28], [29]. It roughly refers to the meaning of different symbols +, - ,*, >, < in code statement. Suppose if we want to express syntactic relation between 3 and 4 the we must maintain semantics between 3 and 4. It could be arithmetic + , we can put it in between which gives the semantic meaning of adding the two

numbers like 3+4. Another example of semantic error is given below in Figure 2.4–

```
// / A C# program that uses System to show a semantic error;
using System;

class CodePropertyGraph
{

// Driver Code
public static void Main(String[] args)
{

   // Before writing Environment, using the exit() statement.
   Environment.Exit(0);

   // / Print the Console value.
   Console.Write("Code Property Graph!");
}
}
```

Figure 2.4: Symantic Error in Source Code

2.1.9 Abstract Syntax Tree (AST)

AST is a tree where each statement prepares a branch of the tree and the parent node is the program super node such as package name [30]. Program has been written as a parent of the tree because this figure comprises a section of source code. For each statement in AST, a new branch has been created, and with that branch, each component of the statement has been asserted. Each node in AST represents a construction that occurs in the source code. The AST results compiler's parsing phase. It is the intermediate representation of the program step by step which has a significant role in final output. Before moving on to the implementation part, let's discuss how to use the AST. Compilers mostly employ AST to verify the accuracy of the code. The compiler outputs an error message if the generated tree has errors. As an example, a statement containing 'x := a + b', creates a branch under the program node. For each operator, a new branch is

generated and the operands are inserted. That is why '=' creates a branch where one side contains 'x ' and another side contains 'a + b'. Due to obtaining an operator '+' in 'a + b', a new branch is created where one side contains 'a' and the other side contains 'b'. Figure 2.5 is an example of Abstract Syntax Tree

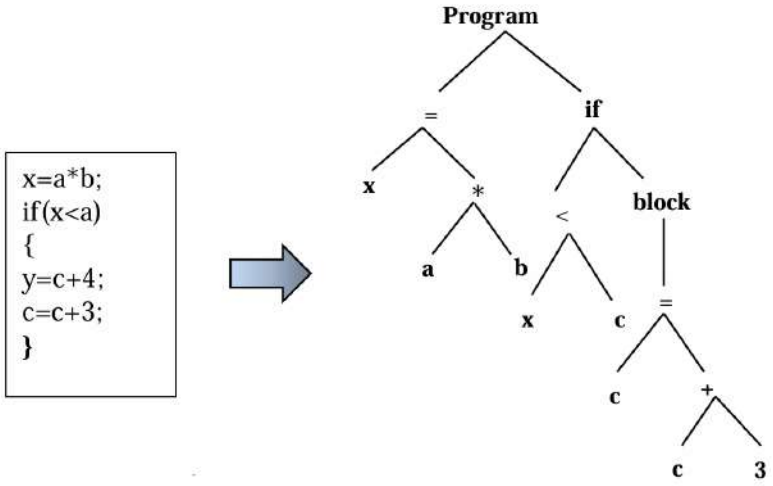


Figure 2.5: Abstract Syntax Tree

2.1.10 Program Dependency Graph from Source Code

In modern software development, ensuring the security of software systems is of utmost importance. Vulnerabilities present in software applications can result in serious consequences, such as data breaches, financial losses, and harm to reputation. Therefore, employing robust techniques to detect vulnerabilities is crucial in order to mitigate these risks. One such technique gaining traction is the utilization of Program Dependency Graphs (PDGs) in vulnerability detection [31]. A Program Dependency Graph serves as a depiction of the interconnections among different elements within a software program. These elements typically comprise variables, functions, statements, and other constructs within the program. In a PDG, nodes symbolize program entities, while edges signify the dependencies between these entities. These dependencies can manifest in various forms, including data dependencies, control dependencies, and call dependencies [32]. Program De-

pendency Graphs offer a powerful framework for analyzing software code and identifying potential security vulnerabilities. By examining the dependencies between program elements, PDGs enable the detection of various types of vulnerabilities, including but not limited to:

1. **Data Flow Analysis:** PDGs facilitate data flow analysis, which is crucial for identifying security vulnerabilities such as information leaks, injection attacks, and data tampering. By tracing data dependencies within the graph, analysts can pinpoint sensitive data flows and potential security risks associated with them [33], [34].

2. **Control Flow Analysis:** Control flow analysis using PDGs helps identify vulnerabilities related to improper control flow management, such as buffer overflows, code injection, and authentication bypasses. By analyzing control dependencies within the graph, vulnerabilities stemming from flawed control structures can be detected [35].

3. **Call Graph Analysis:** PDGs can also be used to construct call graphs, which represent the relationships between different functions or methods within a program. Call graph analysis aids in identifying vulnerabilities related to insecure function calls, including cross-site scripting (XSS), remote code execution (RCE), and command injection [36].

4. **Comprehensive Analysis:** PDGs provide a holistic view of the software codebase, enabling comprehensive analysis of dependencies and interactions between program elements. By considering the interrelationships between different parts of the code, PDGs allow for a thorough examination of potential security vulnerabilities across the entire software system. This comprehensive analysis helps in identifying vulnerabilities that may not be apparent when examining individual code segments in isolation.

5. **Precise Vulnerability Identification:** By leveraging PDGs, analysts can accurately identify vulnerabilities by tracing their impact through the graph. PDGs capture various types of dependencies, including data flow, control flow, and call dependencies, allowing for a detailed understanding of how vulnerabilities propagate within the software. This precise identification enables developers to focus their efforts on addressing the root causes of vulnerabilities, leading to more effective mitigation strategies.[37].

6. **Scalability and Automation:** PDG-based vulnerability detection techniques can be automated and scaled to analyze large and complex software systems efficiently. Automation reduces the manual effort required for vulnerability detection, enabling organizations to scan their codebases regularly for security issues. By leveraging automated PDG analysis, developers can identify vulnerabilities early in the software development lifecycle, reducing the likelihood of security breaches in production.

In summary Program Dependency Graphs represent a valuable technique for vulnerability detection. By leveraging the rich dependency information encoded in PDGs, analysts can effectively identify and mitigate security risks in software applications. As software systems continue to evolve, the role of PDGs in vulnerability detection is expected to grow, contributing to enhanced software security and resilience against cyber threats.

It's evident from the above that each aspect of vulnerability fixing, including vulnerability reporting, pinpointing vulnerabilities in the source code, and resolving them, is interconnected. Locating software vulnerabilities, in particular statements, is crucial during the software maintenance phase and often demands significant developer effort. Therefore, there's a need for automatic identification of vulnerable locations, functions or statements. To automate vulnerability detection, program analysis is commonly employed. This method aids in easily

identifying vulnerable functions and statements within the larger scale codebase. The later sections outlines the typical procedures involved in program analysis.

2.2 Source Code Analysis Concepts

There are various terminologies in software vulnerability detection field and also for understanding the source code analysis such as Program Syntax, Program semantics, Control Flow Graph, Abstract Syntax Tree (AST), Generation of Adjacency Matrix from Program Dependency Graph (PDG), Code Normalization, Centrality Analysis and Word2Vec.These are described below in the following subsections:

2.2.1 Static Analysis of Source Code

Static analysis of source code is a method that examines all potential paths of the code without executing it, relying solely on syntactic and semantic information of the program [38]. This approach stands in contrast to dynamic analysis, which considers runtime behavior [39], [40], [41]. Static analysis methods encompass various techniques such as Model Checking [42], Data-Flow Analysis, and Abstract Interpretation [43], which are facilitated by tools like BLAST [44], Clang [45], Microsoft's SLAM Toolkit [46], and Abstract Syntax Tree (AST).

Source code static analysis identifies the structure and details of the program, preserving the hierarchy of classes, methods, and statements using Abstract Syntax Trees (AST). Each statement generates a branch in the tree, with the program node as the root. This structure facilitates the analysis of code elements and relationships. Static analysis provides a list of source code words to identify potentially vulnerable locations. By measuring textual similarity between the source code and vulnerability reports using Information Retrieval (IR) techniques, relationships between them are established. Text processing methods help identify similarities between vulnerability reports and source code, aiding in ranking vul-

nerable functions or statements based on similarity scores. Example: Vulnerability reports and corresponding source code snippets can be analyzed for textual similarity, as shown in listing 2.1. Keywords in vulnerability reports and source code are compared, and similarity scores determine the ranking of potential vulnerable locations.

Static analysis may include unnecessary details, such as entire source code textual information, leading to decreased accuracy in vulnerability detection. Biasness may arise due to the probabilistic nature of bug localization techniques. Static analysis solely relies on static information, making it unable to identify runtime vulnerabilities. Since it does not consider dynamic behavior, certain types of vulnerabilities may go undetected.

2.2.2 Dynamic Analysis of Source Code

Another technique for vulnerability detection is source code dynamic analysis [47]. This method identifies vulnerable locations by analyzing system runtime information alongside appropriate test suites. By running the software project, dynamic analysis reduces unnecessary search space and identifies relevant source code. It typically generates a call graph showing how methods and classes interact during runtime. For example, in Figure 2.6, the function call graph illustrates the execution flow from the main function to various functions within different classes.

Dynamic analysis is also integral to statistical debugging approaches, where runtime information is collected to predict faulty code. This method helps developers identify suspicious statements by comparing program states using passing and failing test cases. However, the effectiveness of dynamic analysis depends on the quality of the test suites. While dynamic analysis can improve vulnerability detection accuracy by minimizing irrelevant source code, it is more complex compared to static analysis and may not capture the contents of classes or methods, as it primarily focuses on call sequences. Software dynamic analysis involves analyz-

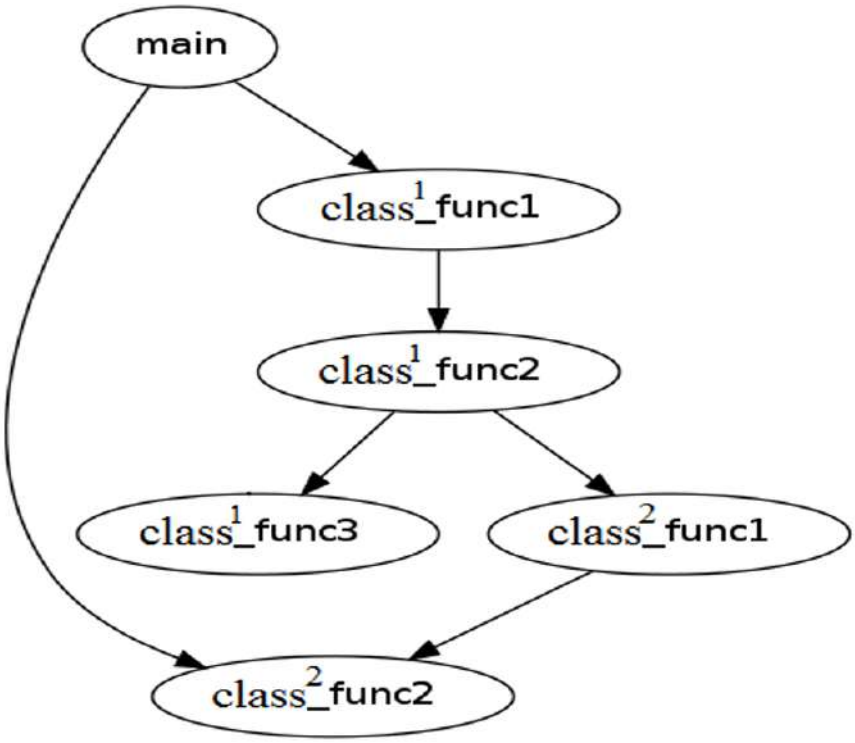


Figure 2.6: Dynamic Analysis of Source Code by Function Calling

ing a program's behavior during execution to identify vulnerabilities and potential security threats. In vulnerability detection, dynamic analysis of source code focuses on monitoring the runtime behavior, inputs, and interactions of the software to uncover security weaknesses. Key points of dynamic analysis in vulnerability detcetion are as follows:

- **Real-world Simulation:** Dynamic analysis simulates real-world scenarios by executing the software under various conditions. This allows for the identification of vulnerabilities that may only manifest during runtime.
- **Monitoring Runtime Behavior:** Dynamic analysis tools instrument the code to monitor its execution, data flows, and interactions with external entities. By observing the program's behavior during execution, vulnerabilities such as injection attacks, memory leaks, and denial-of-service (DoS) vulnerabilities can be detected.

- **Detection of Complex Vulnerabilities:** Dynamic analysis is effective in detecting complex vulnerabilities that may be difficult to identify using static analysis alone. It can capture runtime behaviors that may depend on specific conditions or inputs, providing insights into security risks that may otherwise go unnoticed.

- **Challenges and Limitations:** Dynamic analysis may introduce overhead during runtime and may not exercise all code paths or scenarios, leading to incomplete vulnerability detection. Additionally, it relies on the quality of test suites and may not handle detailed contents of classes or functions.

Dynamic analysis complements static analysis techniques by providing insights into the runtime behavior of the software. By leveraging dynamic analysis in vulnerability detection, developers can gain a deeper understanding of the software's security posture and mitigate potential risks effectively.

2.3 Steps of Text Retrieval Based Vulnerability Detection

In software vulnerability detection, relevant information is gathered from both the source code and vulnerability reports. Once this text is retrieved, it needs to be processed. The texts are then compared with each other, and based on this comparison, their similarity score is measured. This process helps in pinpointing the vulnerable statements within the software. Text retrieval-based vulnerability detection is a method used to extract text from both the source code and vulnerability reports [48]. These techniques are commonly employed in localizing software vulnerabilities. During vulnerability detection, developers often create queries using information from vulnerability reports. The underlying assumption is that vulnerability descriptions and relevant sections of the source code contain impor-

tant keywords. Moreno et al. described the relationship between vulnerability reports and source code in their empirical study [49]. They found that vulnerability reports tend to share more terms with patched classes than with other classes in the system. Additionally, they observed that class names are more likely to share terms with vulnerability descriptions than other parts of the code. Their findings suggest that shared terms may serve as better indicators for identifying vulnerable locations. Vulnerability detection typically involves four phases:

1. **Source Code Corpora Generation**
2. **Query Generation**
3. **Indexing**
4. **Ranking and Retrieval**

Each of these phases comprises a series of steps, and the details of each step are provided below.

2.3.1 Source Code Corpora Generation

Code corpora are collections of words generated from the source code. In this step, lexical analysis is performed for each source code file, creating a vector of lexical tokens. Various pre-processing techniques are applied during this step.The pre-processing involves implementing various techniques as follows:

- Removing stop words
- Eliminating programming language specific keywords
- Handling multi-word identifiers
- Applying Stemming

Each of these techniques is described as follows:

- **Removing stop words:** Stop words are common words in a language that don't convey specific meaning in a given context. Instead, they add unnecessary words that can significantly affect ranking. Removing stop words supports phrase search, a key aspect of bug localization which relies on textual similarity. Common stop words include "a", "as", "able", "across", etc.

- **Eliminating programming language specific keywords:** During software development, programmers commonly use programming language specific keywords like "public", "static", "private", etc., to support the program's functionality. However, these keywords don't provide bug-specific information and are removed from consideration in bug localization.

- **Handling multi-word identifiers:** In software development, variables often consist of multiple words concatenated together. For example, "startDate" comprises "start" and "Date". These multi-word identifiers are split into individual tokens to enhance the relevance of keywords. Conventionally, variable names in object-oriented languages follow patterns like lower Camel case, helping to identify and split words.

- **Applying Stemming:** Stemming is a pre-processing step used to extract the base or root form of words, reducing inflectional forms [50]. Its goal is to standardize words to improve search accuracy. However, stemming can lead to errors such as understemming and overstemming. To address this, various stemming algorithms like Lovins [51], Porter [50], and Paice [52] are available. Porter's algorithm, popular for its effectiveness, applies rules to reduce word extensions in multiple phases. For example, the words "running", "runs", and "ran" all have the same root word "run" after stemming. This helps to standardize words and improve search accuracy.

By removing stop words and programming language specific keywords, the focus

of the vulnerability detection process is refined to identify relevant information for detection and resolution.

2.3.2 Query Generation

Query Generation in vulnerability detection involves utilizing vulnerability reports to form a query for searching the source code and identifying vulnerable locations. The accuracy of vulnerability detection heavily relies on proper query generation. To construct a query from a vulnerability report, text processing techniques are applied, including the removal of stop words, programming language-specific keywords, identification of multi-word identifiers, stemming, and more.

After applying these text processing techniques, a corpus is generated, which serves as the query for vulnerability detection. This corpus generates a bag of words known as the vulnerability report corpus.

2.3.3 Indexing

Following the creation of the code corpus, it's essential to index these corpora. During the indexing phase, the code corpus is stored within the respective source code classes and methods, preserving the structure of the source code. Indexing facilitates the easy identification of the actual buggy classes or methods within the source code. Various types of indexing methods have been established and Latent Semantic Indexing is one of them. The specifics of this indexing method are elaborated in the following:

Latent Semantic Indexing

Latent Semantic Indexing (LSI) is a method for organizing and retrieving information from unstructured documents. It employs Singular Value Decomposition (SVD) to recognize relationships between terms and concepts within these documents. LSI focuses on identifying words with similar meanings, particularly those used in similar contexts. Its primary benefit lies in its capacity to extract the un-

derlying concepts of a document by establishing connections between terms found in similar contexts. LSI takes into account both synonymous and polysemous words. The process involves several steps as shown in Figure 2.7:

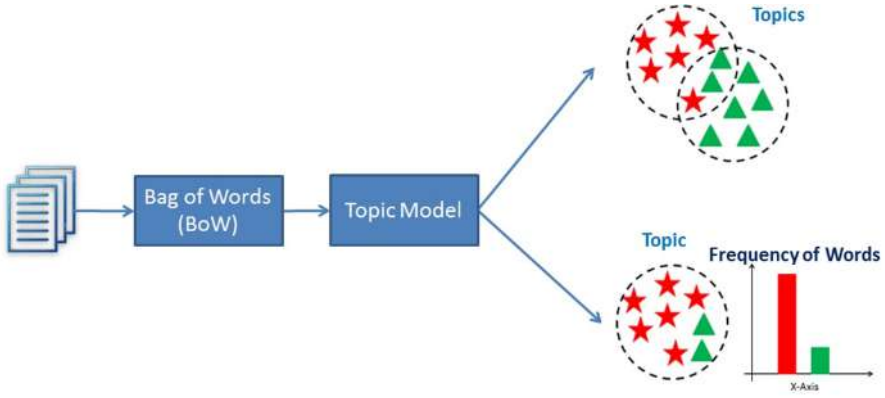


Figure 2.7: Latent Semantic Indexing [2]

- Converting each document into a word vector, with the removal of stop words and ensuring the number of dimensions matches the unique word count.
- Calculating the frequency of each term, considering terms with similar meanings during frequency measurement.
- Creating a term-document matrix to store the frequency of each term within a document.
- Applying Singular Value Decomposition (SVD), where all but the k highest singular values are set to zero.
- Reconstructing the original matrix by recombining the terms.
- Breaking down this reduced rank term-document matrix into column vectors, resulting in the latent semantic index.

2.3.4 Ranking and Retrieval

Ranking and retrieval are vital stages in assessing the similarity between source code corpora and vulnerability report corpora generated during the query genera-

tion phase. Numerous ranking algorithms exist for this purpose, including Vector Space Model (VSM), Latent Semantic Analysis (LSA), and Latent Dirichlet Allocation (LDA), among others. Among these, VSM is extensively employed for retrieval and ranking purposes. Here is an outline of the Vector Space Model:

Vector Space Model

The Vector Space Model (VSM) is an algebraic model employed to represent text documents as vectors of identifiers, typically index terms.

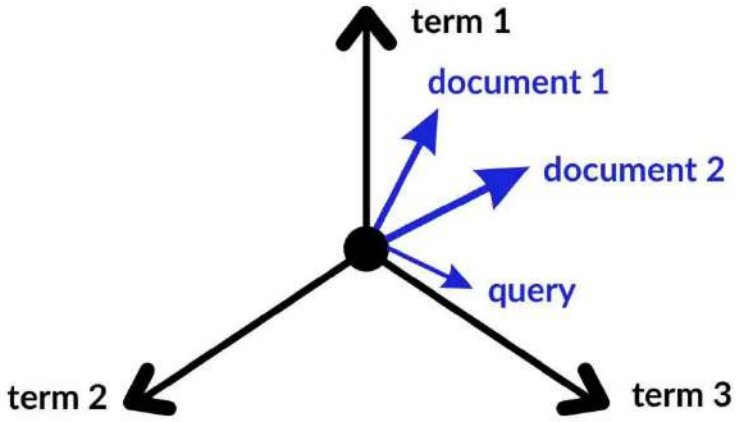


Figure 2.8: Vector Space Model [3]

Let's denote d and q as the document and query vectors, respectively. The similarity between these two vectors can be measured using cosine similarity in VSM, where the angle between the vectors indicates their similarity. A small angle signifies high similarity between the document and query, while a large angle suggests limited similarity between the source code and vulnerability report.The cosine similarity between the vulnerability report and source code is calculated using Equation 2.3.4:

$$\cos = \frac{d \cdot q}{|d| \times |q|}$$

Here, $d \cdot q$ represents the dot product of the document (source code) and query (vulnerability report) vectors, while $|d|$ and $|q|$ are the norms of vectors d and q, respectively. In Figure 2.8, a graphical representation of the query and document vectors is shown. If a document is highly relevant to a query, the angle between

the query and document vectors will be small. Conversely, a larger angle indicates greater dissimilarity between them.

The Vector Space Model (VSM) is utilized for detecting vulnerabilities in source code by representing both the source code and vulnerability reports as high-dimensional vectors of terms or identifiers. In our proposed model of chapter 5, each document and query are converted into vectors within a shared vector space, where each dimension corresponds to a unique term from the corpus. The primary technique for assessing the similarity between the source code (document) and the vulnerability report (query) is cosine similarity. Cosine similarity measures the cosine of the angle between two vectors, providing a value between -1 and 1. A smaller angle indicates a higher similarity, meaning that the source code closely matches the vulnerability report in terms of term distribution and frequency. By calculating this cosine similarity, VSM allows for a straightforward and computationally efficient way to compare textual data, making it suitable for identifying relevant sections of code that may contain vulnerabilities described in the reports.

Latent Dirichlet Allocation (LDA)

LDA is a statistical model that aims to explain why certain parts of data are similar. It's commonly applied in natural language processing to identify text similarities. In LDA, each document is seen as a blend of various topics, such as

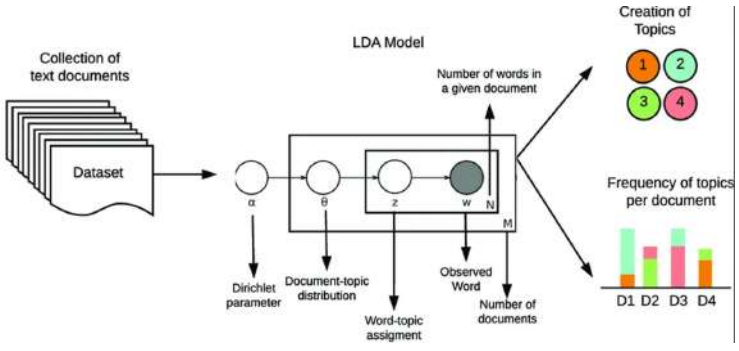


Figure 2.9: Latenet Dirichlet Allocation Model [4]

"Technology" or "Sports." Each topic has probabilities associated with generating

different words, like “computer” for “Technology” and “football” for “Sports.” In Figure 2.9 LDA [53], documents are represented as mixtures of topics, with each topic having its unique set of words. The model assigns probabilities to words under each topic, aiding in understanding the underlying themes within a document collection. Refering to Figure 2.10 for a visual depiction of an LDA model, where D represents the documents, K represents the topics, and W represents the words. This approach is effective in calculating the similarity between a newly reported

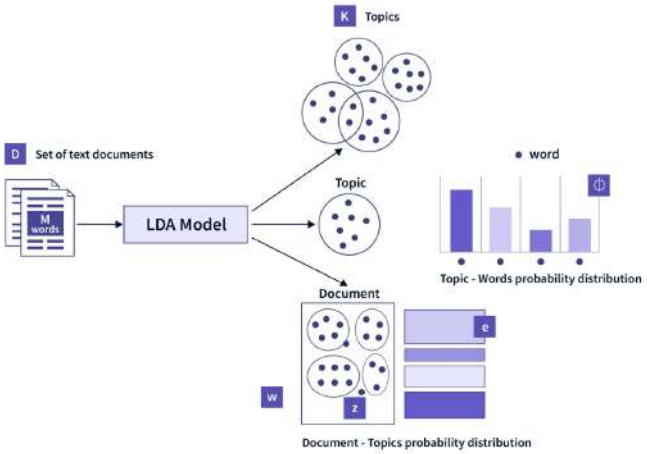


Figure 2.10: Representation of using LDA Model [5]

vulnerability and previously established sets of files or documents. Based on this similarity assessment, we can pinpoint potential vulnerable statements.

LDA is utilized in our proposed approach VFSDetector in chapter 5 to rank vulnerable statements by calculating the similarity score between a vulnerability report and source code, employing concepts of term frequency and inverse function frequency (IFF) from LDA. In LDA, documents are represented as mixtures of topics, where each topic is characterized by a specific distribution of words. This statistical model enables the identification of underlying themes within the text. By analyzing the distribution of terms across topics, LDA assigns similarity scores to different sections of the source code based on their thematic alignment with the vulnerability report. This method effectively highlights code segments that are most likely to contain vulnerabilities, based on their conceptual resemblance

to the reported vulnerabilities, thereby enhancing the accuracy of vulnerability detection and prioritization.

2.4 Summary

This chapter serves as the foundation for understanding automated vulnerability detection, covering essential terminologies such as vulnerability, along with discussing the significance, processes, and hurdles associated with automated vulnerability detection. A vulnerability refers to a discrepancy between the expected and actual behavior of the software. Automated vulnerability detection aims to identify the vulnerable statements to reduce both time and cost.

To localize software vulnerability effectively, the chapter also discusses various types of program analysis. It outlines the standard procedures involved in vulnerability detection, including text processing, indexing, retrieval, and ranking. These discussions are crucial for understanding both function-level and statement-level vulnerability detection methods proposed. Understanding these topics provides a solid foundation for the proposed vulnerability detection. The following chapter will explore existing automatic software vulnerability detection techniques.

Chapter 3

Literature Review

Automatic software vulnerability detection is a process aimed at identifying vulnerable locations within the source code [6]. This automated identification of vulnerable areas can significantly reduce vulnerability fixing time and consequently, decrease software maintenance efforts. Current vulnerability detection techniques typically pinpoint vulnerable statements at either the Function Level or Statement Level. Therefore, this chapter focuses on recent research endeavors that aim to localize buggy locations at these granular levels.

In the preceding chapter, we explored the background knowledge of vulnerability detection, an overview of software vulnerabilities and the general procedures of vulnerability detection techniques. In this chapter, we explore existing research conducted in automated vulnerability detection at the function level or statement level. The objective of such research is to identify vulnerable locations at as granular a level as possible (e.g., files, functions, statements within the source code). Accordingly, this chapter is structured according to the granularity of vulnerability detection, starting with function-level techniques and then proceeding to statement-level techniques. Finally, we conclude this chapter by discussing the limitations of existing vulnerability detection techniques and offering necessary recommendations.

3.1 Existing Approaches

This section focuses on researches that are conducted to detect vulnerabilities. Vulnerability detection techniques vary in their degree of automation, typically falling into three main categories:

- Manual Methods
- Semi-automatic Methods
- Full-automatic Methods

Manual techniques rely on human experts to create vulnerability patterns, but they may not cover all potential vulnerabilities, resulting in lower detection efficiency in real world scenarios for tools like Checkmarx, FlawFinder and RATS. Semi automatic techniques [54, 55, 56] involve human experts extracting specific features, such as API symbols [57] and subtrees, import and function calls [58] which are then fed into traditional machine learning models to identify vulnerabilities. In contrast, full-automatic techniques [59, 60, 61, 62] leverage deep learning to automatically extract features and generate vulnerability patterns without the need for manual expert input. Deep learning based techniques [59, 63, 64, 65, 66] can be further categorized as follows:

- Text-based approaches
- Graph-based approaches.

Figure 3.1 represents the generic workflow of all the detection techniques. Each of these approaches will be detailed extensively in the following sections.

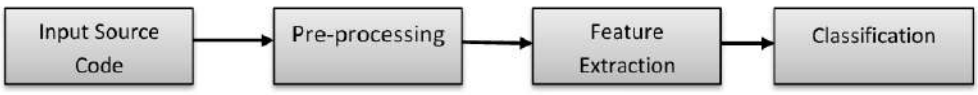


Figure 3.1: Generic workflow of Vulnerability Detection Model

3.2 Text Based Approach

Text-based vulnerability detection approaches focus on identifying security vulnerabilities in software through the analysis of textual information, such as source code, documentation, or vulnerability reports. These approaches leverage natural language processing (NLP) [67] techniques, machine learning models, and other text analysis methods to automatically detect potential vulnerabilities. One common approach is to analyze code comments, commit messages, and documentation for indicators of vulnerabilities, such as mentions of security flaws, error handling or input validation. NLP techniques can be used to extract relevant information from these textual sources and identify patterns associated with known vulnerabilities. Another approach involves analyzing vulnerability reports and security advisories to identify common vulnerabilities reported in software projects. By analyzing the language used in these reports and advisories, text-based vulnerability detection systems can identify similarities with code patterns or configurations that may lead to vulnerabilities. Machine learning models can also be trained on large datasets of code and vulnerability information to automatically detect vulnerabilities based on textual features. These models can learn to recognize patterns associated with vulnerabilities and flag potential issues for further investigation.

Overall, text-based vulnerability detection approaches offer a powerful means of automatically identifying potential security risks in software by analyzing textual information associated with code, documentation, and vulnerability reports.This category includes notable approaches like TokenCNN [64], VulDeePecker [65], muvuldeepecker [66], COSTA [68], LineVul [69] which will be discussed in the following subsections. These approaches can help developers identify and mitigate vulnerabilities early in the development process, improving the overall security of software systems.

3.2.1 VulDeePecker

Li et al. proposed VulDeePecker [65], a pioneering deep learning-based vulnerability detection system designed to alleviate the reliance on manual feature definition and mitigate false negatives. The automatic detection of software vulnerabilities has long been a pivotal concern in cybersecurity research, given the persistent threat landscape posed by malicious actors. Traditional methods of vulnerability detection include pattern-based and code similarity-based approaches. Pattern-based systems, such as Flawfinder [70] and RATS [71], rely on predefined rules to identify vulnerabilities, while code similarity-based systems, like VUDDY [72] and VulPecker [73] detect vulnerabilities by comparing code snippets for similarity. Traditional methods for vulnerability detection have often relied on human experts to manually define features, resulting in high false negative rates and significant labor intensity. The study sets out to address fundamental limitations in existing vulnerability detection approaches by leveraging the potential of deep learning. While acknowledging that deep learning was not originally conceived for vulnerability detection, the paper outlines guiding principles for applying this technology to the domain. Central to this approach is the use of "code gadgets" [74] non-consecutive lines of code that are semantically related and can be transformed into vectors suitable for deep learning analysis.

The contribution of VulDeePecker lies in its design and implementation as a deep learning-based vulnerability detection system. The effectiveness of VulDeePecker is evaluated across multiple dimensions, including its ability to detect multiple types of vulnerabilities simultaneously, the potential enhancement through the incorporation of human expertise, and its performance compared to other vulnerability detection approaches. Experimental results demonstrate VulDeePecker's superiority in achieving lower false negative rates, thereby highlighting its potential to significantly improve vulnerability detection capabilities. Furthermore, VulDeePecker is applied to real-world scenarios involving three software products,

where it successfully identifies vulnerabilities not reported in the National Vulnerability Database (NVD) [75]. This underscores the system's ability to uncover previously undetected vulnerabilities, thereby enhancing overall software security.

Despite its advancements, VulDeePecker is not without limitations. The study identifies several areas for future research, including the extension of the system to detect vulnerabilities in executables, adaptation to support diverse programming languages beyond C/C++, and the exploration of additional vulnerability types beyond library/API function calls [76]. Moreover, the study acknowledges the need to refine the system's heuristic-based approaches and expand its evaluation to encompass a broader range of vulnerability types and software products.

3.2.2 μVulDeePecker

Zou et al. introduced μVulDeePecker [66], the first deep learning-based system designed for multiclass vulnerability detection. Traditional methods in vulnerability detection often struggle with pinpointing the specific type of vulnerabilities present in software code, while existing deep learning-based solutions excel in binary classification, they fall short in multiclass classification, where the system must identify the precise type of vulnerability. To address this gap, μVulDeePecker leverages the concept of "code attention" to capture information crucial for identifying vulnerability types, even with limited data samples. The study presents a new dataset and systematic experiments to evaluate the effectiveness of µVulDeePecker, highlighting its ability to detect and classify vulnerabilities accurately.

The primary contribution of the paper lies in pioneering the domain of multiclass vulnerability detection using deep learning. The system, μVulDeePecker, introduces several conceptual and methodological innovations. Code attention refines the existing concept of code gadgets, enabling the system to capture local features crucial for identifying vulnerability types. Incorporating Control-Dependence in addition to Data-Dependence, μVulDeePecker achieves higher ef-

fectiveness in multiclass vulnerability detection. A novel Neural Network Architecture is proposed, incorporating building-block BLSTM networks and fusion of features from code gadgets and code attention, offering a more effective approach to leveraging different types of code features. Additionally, the paper provides a new dataset specifically designed for evaluating multiclass vulnerability detection systems, facilitating further research in this area.

μVulDeePecker's methodology involves three key components: Conceptual Framework introduces code attention and refines code gadgets to accommodate both data and control dependence. Dataset creation involves the development of a new dataset containing code gadgets from various programs, with annotations for vulnerability types. System evaluation entails systematic experiments conducted to evaluate the effectiveness of μVulDeePecker in multiclass vulnerability detection, comparing different variants and architectures.

The systematic experiments demonstrate the effectiveness of μVulDeePecker in multiclass vulnerability detection. μVulDeePecker accurately detects and classifies vulnerability types, even with limited data samples, which credit to the incorporation of code attention. Accommodating control-dependence significantly improves the system's detection capabilities. The proposed neural network architecture outperforms traditional approaches, demonstrating the efficacy of feature fusion.

While groundbreaking, μVulDeePecker has some limitations that warrant further investigation. The system can identify vulnerability types but cannot pinpoint the precise location of vulnerabilities beyond the granularity of code gadgets. Current implementation focuses on programs written in C/C++, necessitating future research to support other programming languages. Vulnerabilities related to library/API function calls are primarily addressed, leaving out other vulnerability types for consideration in future studies.

3.2.3 TokenCNN

Russell et al. present a pioneering approach TokenCNN [64] to automated vulnerability detection leveraging machine learning (ML) techniques directly from source code in response to the escalating challenges posed by software vulnerabilities. The paper encapsulates a comprehensive exploration of ML methodologies applied to function-level vulnerability detection in C and C++ codebases.

The work begins by addressing the foundational necessity of large, diverse datasets for effective ML model training. TokenCNN meticulously curate a vast dataset comprising millions of function-level examples extracted from multiple sources, including synthetic code from the SATE IV Juliet Test Suite, Debian Linux distribution [77], and public Git repositories [78]. This diverse dataset encapsulates the vulnerabilities of real-world code, essential for training robust vulnerability detection models.Central to the success of the ML framework is the custom C/C++ lexer, engineered to distill critical tokens while maintaining a generic representation. This standardized lexed representation, with a modest vocabulary size of 156 tokens, facilitates transfer learning efficacy across diverse datasets while mitigating overfitting risks through stringent duplicate removal processes.Labeling functions for vulnerability analysis proves challenging due to the absence of ground truth in mined open-source code. TokenCNN address this challenge by employing a tripartite labeling approach encompassing static analysis, dynamic analysis, and commit-message/bug-report tagging. The integration of multiple static analyzers— Clang [45], Cppcheck [79], and Flawfinder [70]—facilitates robust label generation, augmented by expert mapping of findings to Common Weakness Enumeration (CWE) categories [80].

Building upon the data curation and labeling foundations, Russell et al. explore the development of ML models for vulnerability detection. The authors employ convolutional neural networks (CNNs) [81] and recurrent neural networks (RNNs) for feature extraction, pooling, and dense layering to ensure the genera-

tion of fixed-size representations conducive to effective vulnerability classification. Training the models involves careful consideration of data batching, optimization algorithms, loss functions, and class imbalance handling. Leveraging techniques such as Adam optimization with weighted cross-entropy loss addresses the dataset's strong class imbalance, enhancing the models' sensitivity to vulnerable functions. While the neural network approaches exhibit promise, their classification performance on the full dataset remains suboptimal. Russell et al. overcome this challenge by integrating ensemble classifiers, such as random forest (RF), with neural features, yielding superior vulnerability detection results compared to traditional bag-of-words representations. The efficacy of the ML framework is underscored by comprehensive evaluation metrics across various test datasets. The CNN models outperform RNN counterparts and traditional bag-of-words representations across metrics such as precision-recall area under curve (PRAUC), receiver operating characteristic area under curve (ROCAUC), Matthews Correlation Coefficient (MCC), and F1 score.

3.2.4 LineVul

Fu et al. proposed a statement level vulnerability detection method named LineVul [69] which addresses the critical need for fine-grained vulnerability prediction in software systems by introducing a Transformer-based line-level approach. Traditional methods often provide coarse-grained predictions at the file or function level, limiting their effectiveness in identifying vulnerabilities accurately. Leveraging a large-scale real-world dataset comprising over 188k C/C++ functions, LineVul aims to significantly enhance the accuracy and granularity of vulnerability predictions, thereby mitigating potential risks associated with software vulnerabilities.

The primary contribution of LineVul lies in its innovative approach to vulnerability prediction, particularly at the line level. By leveraging Transformer architecture with self-attention layers, LineVul effectively captures long-term de-

pendencies within code sequences, improving the representation learning process. Additionally, LineVul utilizes a pre-trained language model (CodeBERT) to generate vector representations of source code, enabling more robust and generalized predictions. Furthermore, LineVul introduces a novel attention mechanism to locate vulnerable lines, offering finer-grained predictions compared to existing methods.

Empirical evaluation of LineVul on the large-scale real-world dataset reveals substantial improvements in accuracy and cost-effectiveness compared to baseline approaches. LineVul achieves remarkable performance metrics, including 160%-379% higher F1-measure for function-level predictions, 12%-25% higher Top-10 Accuracy for line-level predictions, and 29%-53% less Effort@20%Recall. These results underscore the significant advancement of LineVul towards more accurate and finer-grained vulnerability predictions, with potential implications for real-world security scenarios.

While LineVul demonstrates promising results, it is not without limitations. The window size limitation of 512 tokens may restrict its ability to learn longer functions fully, potentially affecting prediction accuracy for complex code segments. Moreover, LineVul's reliance on specific datasets and hyperparameter settings raises concerns about its generalizability and internal validity across diverse software environments. Addressing these limitations will be crucial to enhancing the robustness and applicability of LineVul in real-world software systems.

3.2.5 COSTA

Nguyen et al. proposed a statement level approach named COSTA [68] which presents a context-based approach to localize vulnerable statements within software functions. Traditional detection methods often operate at a coarse-grained level [65], [61], necessitating significant developer effort to pinpoint vulnerable code segments. COSTA addresses this challenge by assessing the suspiciousness

of individual statements within functions, leveraging four contexts: operation, dependence, surrounding, and vulnerability type. Through extensive experimentation on a large dataset, COSTA demonstrates significant improvements over existing approaches, achieving higher F1-score and higher accuracy.

COSTA contributes a novel methodology for fine-grained vulnerability localization, focusing on identifying vulnerable statements within functions. By considering multiple contextual factors, including operation semantics, dependence relationships, surrounding code and vulnerability type, COSTA offers a comprehensive approach to assessing the likelihood of vulnerability at the statement level. This context-based strategy enables developers to more efficiently pinpoint vulnerable code segments, reducing investigation efforts and enhancing overall software security. Experimental evaluation of COSTA on a large-scale vulnerability dataset Big-Vul[82] which reveals substantial performance gains compared to state-of-the-art approaches. COSTA achieves up to 96% higher F1-score and 167% higher Accuracy, demonstrating its superior effectiveness in localizing vulnerable statements within software functions. Notably, COSTA also outperforms existing methods [69], [83], [84] by up to two times in Top-1 Accuracy and achieves approximately 80% recall at the Top-3 level, enabling developers to identify a significant portion of vulnerable statements with minimal investigation effort.

Despite its promising results, COSTA is subject to several limitations. The approach relies on the accurate extraction of contextual information, which may be challenging in complex codebases. Furthermore, while COSTA demonstrates effectiveness for C/C++ programs, its applicability to other programming languages remains unexplored. Additionally, the reliance on neural network models for context representation may introduce scalability challenges and require further exploration to optimize performance across diverse software environments. Addressing these limitations will be essential for enhancing the robustness and generalizability of COSTA in real-world scenarios.

3.3 Graph Based Approach

Graph-based vulnerability detection is a sophisticated methodology that employs graph representations, such as Abstract Syntax Trees (ASTs) [30], Control Flow Graphs (CFGs) [35], and Data Flow Graphs (DFGs) [33] to identify security vulnerabilities in software systems. To overcome the limitations of text-based approaches, researchers have turned to dynamic program analysis, converting source code semantics into a graph and employing graph analysis for vulnerability detection. By constructing graphs that capture the structural and behavioral aspects of the code, this approach offers a comprehensive view of the software artifacts, facilitating the detection of complex vulnerabilities that may involve intricate code interactions. These graphs provide a contextual understanding of the codebase, enabling accurate identification of vulnerabilities and reducing false positives. Moreover, graph-based techniques are scalable and can effectively analyze large codebases, offering modular analysis at different granularities, from individual functions to entire applications. The methodology involves graph construction, feature extraction, and vulnerability detection, where features extracted from the graphs are utilized by algorithms to classify code segments as vulnerable or benign. This approach integrates with existing techniques in software security research, as demonstrated by works like - Zhou et al. [63] and Wu et al. [85] which leverage Code Property Graphs (CPGs) to represent code functions for vulnerability detection. IVDetect [86] and LineVD [83] extends this approach to detect vulnerabilities at a finer-grained level, while VELVET [84] employs graph-based techniques to identify vulnerable statements within the codebase. In summary, graph-based vulnerability detection offers a powerful and scalable approach to enhancing software security by providing a comprehensive, context-aware representation of code and enabling accurate detection of vulnerabilities across various levels of abstraction and each of these approaches are described in the following

subsections.

3.3.1 VulCNN

Wu et al. introduced VulCNN [85], a novel approach for scalable and accurate detection of source code vulnerabilities. Traditional text-based techniques lack accuracy, while graph-based methods suffer from scalability issues. VulCNN addresses these challenges by converting source code into images, preserving program details. Through experimentation, VulCNN demonstrates superior accuracy and scalability compared to state-of-the-art detectors.

VulCNN method converts source code into images while preserving program details, enabling accurate and scalable vulnerability detection. It introduces a novel approach inspired by DL-based image classification, achieving better accuracy than existing detectors. VulCNN also presents a prototype system for graph-based vulnerability detection and conducts comprehensive evaluations on a dataset of vulnerable and non-vulnerable functions. Different thresholds for lines of code in VulCNN's image generation process were tested, with 100 lines chosen for optimal accuracy and memory efficiency. Comparative analysis showed commercial and open-source static analysis tools had sub optimal detection rates due to reliance on human-defined rules. The study leverages deep learning (DL) techniques for source code vulnerability detection. It begins by distilling program semantics into a program dependency graph (PDG) and applying centrality analysis to attach structural information to each line of code. Three centralities (degree, katz, and closeness centrality) are utilized to transform PDGs into images. These images are then fed into a Convolutional Neural Network (CNN) model for vulnerability detection. Class Activation Maps (CAM) [87], [88] are employed for pinpointing vulnerable code lines. Evaluation is conducted on a dataset of 40,657 functions, demonstrating VulCNN's superiority over eight comparative detectors and its scalability in large-scale vulnerability scanning. Additionally, a case study

on over 25 million lines of code uncovers 73 previously unreported vulnerabilities.

While VulCNN shows promising results, there are limitations to consider. The dataset used for evaluation may not fully represent all program types, potentially affecting the generalizability of results. Additionally, false positives may occur, especially when analyzing open-source products. However, these limitations are mitigated through careful dataset curation and comparison with real-world vulnerabilities.

3.3.2 Devign

Zhou et al. proposed Devign [63], a novel approach to vulnerability identification in source code, leveraging graph neural networks and multiple code representations. The model is tailored for detecting vulnerable functions and operates by encoding functions into joint graph structures from various syntax and semantic representations. Through comprehensive graph embeddings and a Conv module, Devign achieves state-of-the-art performance in vulnerability discovery across real open-source projects. Devign builds upon prior research in vulnerability discovery and graph neural networks. While earlier works focused on natural language processing techniques for vulnerability detection, Devign distinguishes itself by incorporating graph-based representations of code. The model draws inspiration from gated graph recurrent networks and employs novel techniques for graph-level classification. Additionally, Devign contributes to the growing body of literature on automated vulnerability identification, offering advancements in both methodology and performance.

The primary contribution of Devign lies in its ability to integrate diverse code representations into a unified graph structure, enabling effective vulnerability identification. By leveraging gated graph recurrent networks and a Conv module, Devign surpasses existing machine-learning-based methods, providing a significant advancement in automated vulnerability detection. Additionally, the paper

introduces a meticulously curated dataset of manually labeled vulnerable functions, enhancing the reliability and accuracy of evaluation. Experimental results showcase the superior performance of Devign compared to baseline methods and state-of-the-art vulnerability prediction techniques. Across multiple datasets from popular open-source projects, Devign consistently outperforms other approaches. For instance, it achieves an average accuracy gain of 10.51%, with a minimum of 8.54% on the QEMU dataset [89]. Notably, in scenarios with imbalanced datasets, Devign demonstrates remarkable performance, outperforming static analyzers by 27.99% in terms of average F1 score. Moreover, when applied to the latest CVEs, Devign achieves an average accuracy of 74.11%, highlighting its potential for identifying zero-day vulnerabilities.

Despite its promising results, Devign faces certain limitations. One notable limitation is the computational complexity associated with processing large functions, necessitating the filtering of functions above a certain node size threshold. Additionally, Devign's performance may be impacted by the quality and completeness of code representations obtained from code analysis tools like Joern. Further research is needed to address these limitations and enhance the scalability and robustness of the model.

3.3.3 IVDetect

Li et al. proposed IVDetect [86] an interpretable vulnerability detection system that combines Artificial Intelligence (AI) for detecting vulnerabilities with an Intelligence Assistant (IA) for providing interpretations of vulnerable statements. It leverages context-aware representation learning of vulnerable code and interpretable machine learning (ML) to provide fine-grained interpretations in terms of vulnerable statements. Existing vulnerability detection (VD) approaches are categorized into program analysis (PA)-based and machine learning (ML)-based techniques. While PA-based methods focus on specific vulnerabilities, ML-based

approaches aim to learn vulnerability patterns implicitly. However, ML/DL-based VD approaches lack fine-grained details and struggle to provide interpretations of vulnerable statements. IVDetect bridges this gap by combining AI for coarse-grained detection with interpretable ML for fine-grained interpretations.

IVDetect contributes an interpretable VD system that enhances coarse-grained vulnerability detection by providing fine-grained interpretations. It introduces context-aware representation learning of vulnerable code, distinguishing vulnerable statements from surrounding contextual code. Additionally, it leverages interpretable ML models to provide interpretations in terms of vulnerable statements relevant to detected vulnerabilities. Empirical evaluation demonstrates significant performance improvements over existing DL-based approaches.

IVDetect outperforms existing DL-based approaches by 43%–84% and 105%–255% in top-10 nDCG and MAP ranking scores for vulnerability detection. It accurately identifies relevant vulnerable statements in 67% of cases with a top-5 ranked list, enhancing interpretation accuracy by 12.3%–400% compared to baseline models. While IVDetect offers significant advancements, it also faces several limitations. Firstly, it may struggle with unseen vulnerabilities, as it relies on training data to recognize patterns. Secondly, there's a risk of incorrectly identifying vulnerable statements due to complex data and control dependencies, potentially leading to false positives or negatives. Lastly, the system may miss vulnerable statements when multiple edges of data/control dependencies obscure the relevant information. Addressing these limitations remains a focus for future improvements.

3.3.4 LineVD

Hin et al. proposed LineVD [83], an approach for statement-level vulnerability detection in code snippets, leveraging Graph Neural Networks (GNNs) and interpretable ML methods. Through extensive evaluations, LineVD demonstrates superior performance compared to existing models like IVDetect, showcasing its

ability to accurately identify vulnerable statements within code snippets and enhance software security. However, LineVD faces challenges with certain types of longer functions and trade-offs between interpretability and efficiency in real-time application, highlighting areas for further research and improvement. The paper addresses the need for statement-level vulnerability detection in source code, a task not extensively explored despite the growing popularity of graph-based models for function-level vulnerability detection. LineVD introduces a novel deep learning framework that formulates statement-level vulnerability detection as a node classification task, leveraging graph neural networks and transformer-based models to encode raw source code tokens. By integrating control and data dependencies between statements, LineVD significantly improves prediction performance, achieving a remarkable 105% increase in F1-score over the current state-of-the-art.

LineVD makes significant contributions to statement-level vulnerability detection by introducing a novel approach that combines Graph Neural Networks (GNNs) with interpretable ML methods. Through thorough evaluations, LineVD showcases superior performance compared to existing models like IVDetect, across various evaluation metrics. Notably, LineVD's ability to accurately identify vulnerable statements within code snippets, its versatility across different embedding methods and GNN architectures, and its generalizability across diverse software projects highlight its potential to enhance software security by enabling developers to pinpoint potential vulnerabilities at a granular level.xperimental results showcase the superior performance of LineVD compared to the state-of-the-art interpretation-based SVD model. LineVD achieves a significant increase of 105% in F1-score, indicating its effectiveness in identifying vulnerable statements. Moreover, the paper investigates the impact of different code embedding methods and graph neural networks on LineVD's performance. Cross-project classification experiments further validate LineVD's effectiveness and generalization capabilities, highlighting its potential for real-world application.LineVD achieves significant

performance advantages over the state-of-the-art SVD model, IVDetect. Through a comprehensive comparison of LineVD with IVDetect using various evaluation measures it is found that LineVD demonstrates superiority across all ranked metrics, with statistical significance ($p< 0.01$). Notably, LineVD improves accuracy metrics, such as N5 by 0.205 and MAP@5 by 0.336, showcasing its ability to generate more precise vulnerability detections at the statement level. Furthermore, LineVD enhances the performance of MFR by 4.373 ranks, representing a remarkable increase in the efficiency of correctly locating the first vulnerable statement by 114.5%.

While LineVD exhibits promising results, it is not without limitations. One notable limitation is its potential struggle with certain types of longer functions, as evidenced by the skewed distribution of rankings towards the lower end. Although LineVD correctly ranks the majority (89%) of first-ranking scores between 1 to 5, it may encounter challenges with longer functions, impacting its overall performance in these cases. Additionally, LineVD's efficiency in generating statement-level predictions is highlighted as an advantage over IVDetect. However, the use of an explanation model significantly increases inference time, posing practical challenges for real-time application in certain scenarios. For instance, a single inference using GNNExplainer can take over a minute, while LineVD's single forward pass requires less than a second, showcasing the trade-offs between interpretability and efficiency in vulnerability detection models.

3.4 Summary

This chapter reviews two primary approaches for vulnerability detection: text-based and graph-based. Text-based methods, including VulDeePecker, VulDeePecker, TokenCNN, LineVul, and COSTA, utilize static program analysis and natural language processing to identify vulnerabilities in source code. However, they

often fail to capture the semantics of the code. VulDeePecker, for instance, uses BLSTM models, while VulDeePecker incorporates code attention and control dependence. TokenCNN uses lexical analysis and CNN, LineVul focuses on code repositories, and COSTA employs deep learning on textual descriptions.

Graph-based methods address the limitations of text-based approaches by representing source code semantics as graphs and applying Graph Neural Networks (GNN). Examples include Devign, which uses a convolutional module for graph-level classification, Deepwukong, which analyzes program dependency subgraphs, and IVDetect, which employs a Graph Convolutional Network. VulCNN combines both text and graph-based techniques but struggles with complex code contexts and semantics.

Graph-based methods are effective in identifying vulnerabilities but face challenges in scalability and computational requirements. Conversely, text-based methods often miss inter-dependencies between code lines, reducing accuracy. The chapter highlights the underutilization of vulnerability reports, which provide valuable contextual information. It suggests leveraging these reports to narrow the search space for detecting vulnerable functions and introduces a technique for statement-level vulnerability detection within this focused area.

Chapter 4

Combination of Text and Graph Based Approach in Function Level Vulnerability Detection

Function level vulnerability detection involves identifying security weaknesses within individual functions or modules of software systems [6]. It aims to reduce the time spent on fixing vulnerabilities and enhance the accuracy of automated software vulnerability detection by detecting the correct vulnerable functions. Existing approaches have either used text [64], [65] or graph [63], [86] based approaches between vulnerable code and potential fixes. This chapter explores the potential of combining both approaches. By integrating text and graph based approaches, the detection of vulnerabilities may be enhanced because it incorporates both syntactic and semantic information of source code, thereby elevating the detection of correct vulnerabilities. This chapter introduces a novel hybrid approach that integrate both text and graph based approach aiming to to achieve both accuracy and scalability when identifying vulnerabilities in source code with their combined impact. Comparative analyses with existing techniques relying solely on text or graph reveal the superiority of our combined approach. Results demon-

strate that these integrated approaches outperforms both text and graph based methods underscoring the effectiveness of our approach in detecting vulnerabilities in large-scale source code. Hence, developers can employ these findings to promptly apply effective patches on vulnerable functions.

4.1 Introduction

Software vulnerabilities pose an increasing risk to software systems making them susceptible to attacks and potential damage, thereby raising security concerns [6]. In 2023, the *Open Source Security and Risk Analysis* (OSSRA) [90] conducted a comprehensive study involving 1703 codebases with audit data. The findings indicated that 76% of the codes were open source. Moreover, 48% of the codebases exhibited high-risk vulnerabilities and 84% of these vulnerabilities were associated with open source security flaws. Consequently, to enhance the security of software it is crucial to employ advanced methods for detecting vulnerabilities on a large scale.

Recently, several approaches for vulnerability detection using Deep Learning (DL) have emerged falling into two categories: the *text-based* approach [65, 66] and the *graph-based* approach [63, 59]. Prior studies VulDeePecker [65], VulDeePecker [66], TokenCNN [64], LineVul [69] COSTA [68] focusing on text-based identification of source code vulnerabilities applied static program analysis or natural language processing. However, these approaches often fall short in disregarding the semantics of the source code. To address these limitations, program analysis is employed to represent source code semantics as a graph. Graph analysis methods such as Graph Neural Networks (GNN), are then applied to identify vulnerabilities. While these graph-based approaches excel at vulnerability identification but their scalability is challenging, especially when compared to text-based approaches. Text-based approaches, lacking the ability to capture inter-dependencies between

different lines of source code, result in lower accuracy. On the other hand, graph-based methods achieve high accuracy but struggle with scalability in complex scenarios with many nodes in a graph representing statements in the program's source code. The most recent vulnerability detection system, VulCNN [85] and ASVD [91], attempts to combine both text-based and graph-based approaches to gather syntactic and semantic data from source code. However, VulCNN's scalability has improved compared to eight state-of-the-art vulnerability detectors. Still, its detection performance is deemed unsatisfactory as it leverages Sent2Vec [92] embedding in source code which struggles in capturing intricate code context and misinterpret code semantics. VulCNN only considers three network centralities [93] (degree centrality, katz centrality, and closeness centrality) for calculating the importance of each line of source code. The impact of incorporating other centrality measures such as eigenvector centrality and betweenness centrality in vulnerability detection has not been explored yet. But these centralities might have a greater impact on vulnerability detection combining with DL based approaches which accounts for the overall effect of a node or statement in source code and detects vulnerability.

To address these issues mentioned above, this chapter proposes an enhanced automated **Vul**nerability detection method using Code**BERT** and **C**onvolutional **N**eural **N**etwork named VulBertCNN which uses CodeBERT [94] embedding model leveraging the advantages of large-scale pre-training which captures both syntactic and semantic information of source code. Our method comprises of three phases. First, using the source code as input, we create a Program Dependency Graph (PDG) containing data flow and control flow information. In the second phase, CodeBERT embedding is applied to each node of the generated PDG. In the third phase, we utilize different combinations of five centralities on each node, with each centrality corresponding to a channel, to create an image. Finally, a Convolutional Neural Network (CNN) [81] model is trained on the produced images to identify

functions as either vulnerable or non-vulnerable.

To ensure the model's effectiveness in large-scale vulnerability scanning we explored pre-trained CodeBERT embedding model with various centrality combinations and we applied the proposed model on two benchmark dataset SARD and Big-Vul [82]. Results reveal its outperformance over three state-of-the-art vulnerability detectors TokenCNN [64], VulCNN [85]and ASVD [91] with a significant improvement in the F1-score increasing it from 62.6% to 89% and detection accuracy from 78.6% to 95.7% which is about 21.7% and 26.3%.

4.2 Methodology of the Proposed System

This section considers finding the vulnerable functions from a large system. The goal is to achieve accurate vulnerable functions taking into account both syntactic and semantic information of the source code. This section proposes VulBertCNN (Vulnerability detection with CodeBERT based Convolutional Neural Network) which consists of three major phases: *Graph Generation*, *Feature Extraction* and *Vulnerability Detection* as shown in Figure 4.1.

4.2.1 Graph Generation

As shown in Figure 4.2, this phase initially normalize the source code of a function before performing static analysis to obtain the function's Program Dependency Graph (PDG). Since the goal of VulBertCNN is to concurrently detect vulnerabilities with accuracy and scalability, firstly static analysis is performed to translate the program semantics of source code into a graph representation. Since a function can potentially implement a specific task, hence this phase concentrate on finding vulnerabilities at a more fine grained level (i.e., function-level) due to the coarse granularity of file-level vulnerability detection. The normalization is performed in three steps. A sample function transformation at three normalization steps has

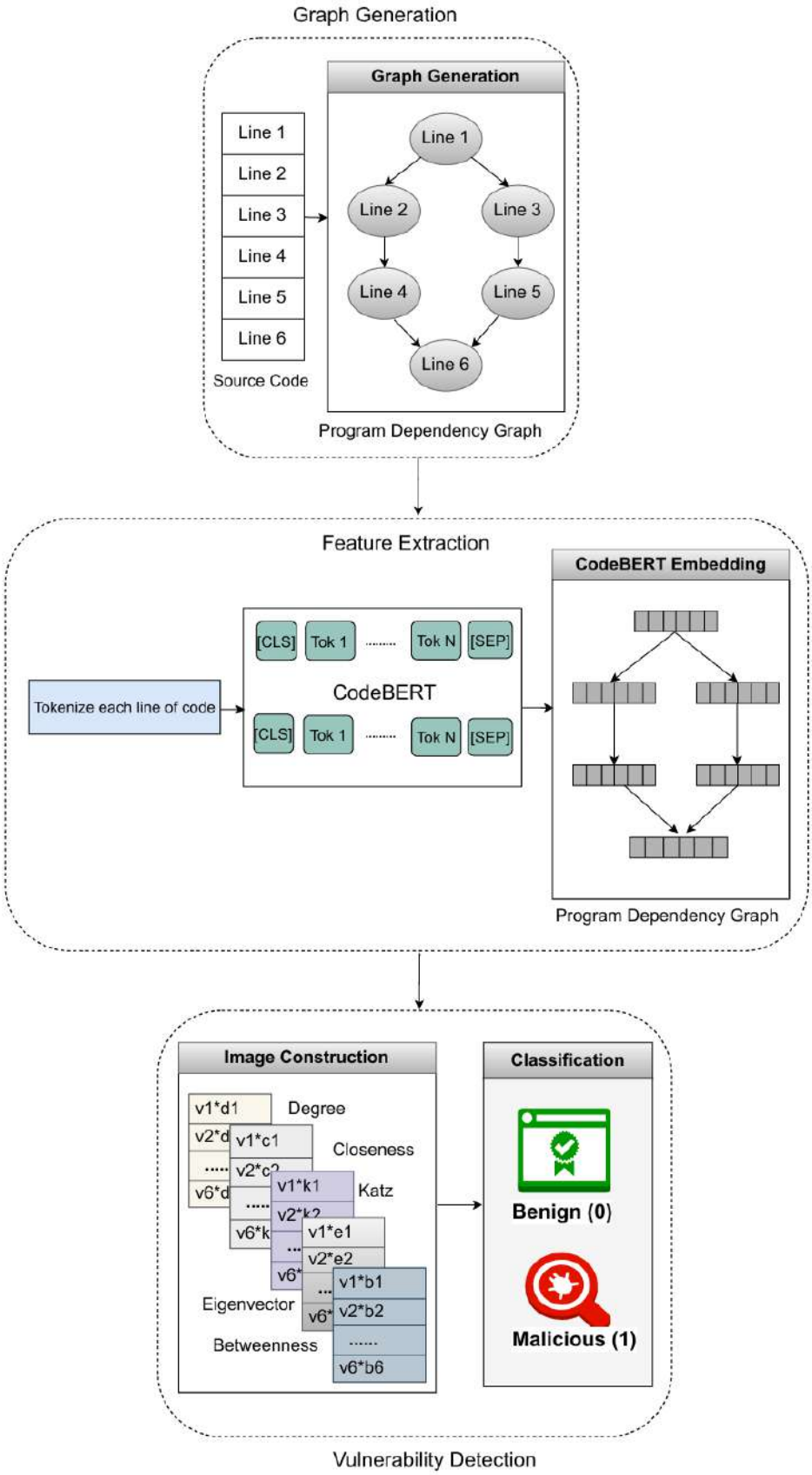


Figure 4.1: Overview of the Proposed Automated Vulnerability Detection System in Function Level

been demonstrated in Figure 4.2. The steps are -

- **Step 1**: Eliminates the comments from the source code because they have no effect on the semantics of the program. Comments in the source code are intended for human readability and have no impact on the actual semantics or execution of the program. Removing comments is essential because they do not contribute to the logic or behavior of the code. By eliminating comments, the analysis process focuses purely on the operational aspects of the code, ensuring that only relevant information is retained. This step helps streamline the code and reduces noise, allowing for more accurate static analysis and representation of the program's functionality.

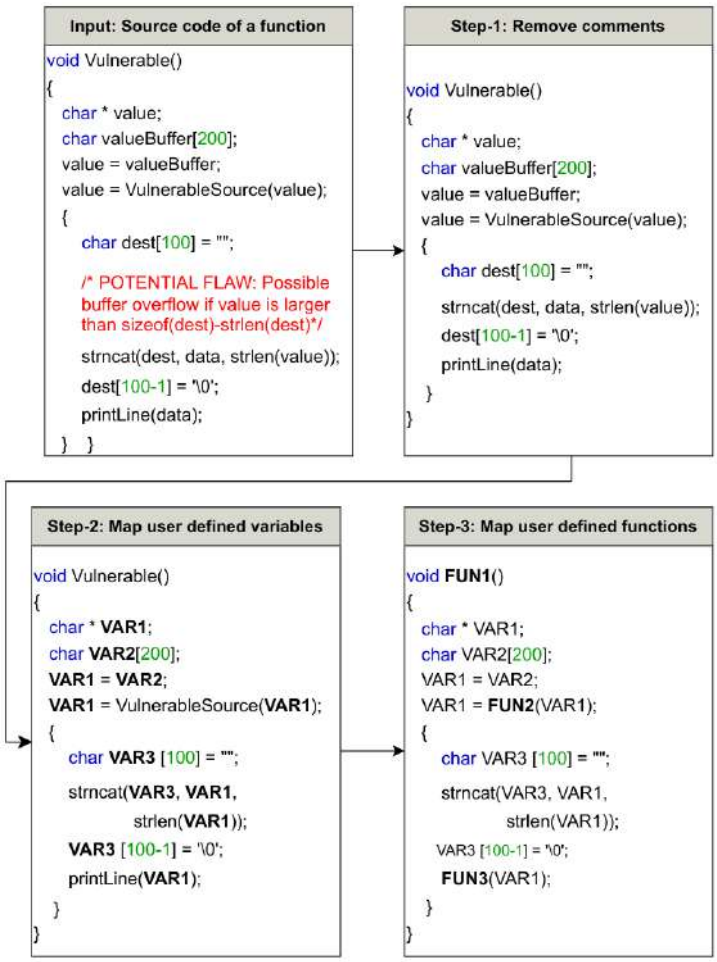


Figure 4.2: Steps of Source Code Normalization

Step 2: User-defined variables can vary greatly between different pieces of code, even when they perform similar functions. By mapping these variables to symbolic names (e.g., "value" to "VAR1"), the normalization process abstracts away specific identifiers, reducing the variability that can obscure underlying patterns in the code. This abstraction helps in identifying vulnerabilities by focusing on the structural and functional aspects of the code rather than the specifics of variable names. This step ensures that the analysis can generalize across different codebases, improving the scalability and accuracy of vulnerability detection.

Step 3: Similar to variables, user-defined function names can differ widely across various programs. Converting these function names to symbolic names (e.g., "Vulnerable()" to "FUN1()") removes specific identifiers, allowing the analysis to concentrate on the behavior and interaction of the functions within the code. This abstraction is crucial for detecting vulnerabilities at a function level, as it emphasizes the functional role and connections between code elements over the specific naming conventions used by different programmers. This step enhances the ability to identify common vulnerability patterns across different functions and codebases. After normalization the PDG of the function is then extracted using an open source code analysis tool for C/C++ named Joern [95]. Each line of code in the function represents a node in a PDG.

4.2.2 Feature Extraction

In this feature extraction step of our research, we employ CodeBERT [94], a pre-trained BERT model that integrates both Natural Language (NL) and Programming Language (PL) encodings creating a comprehensive model suitable for fine-tuning on source code tasks. Trained on an extensive dataset sourced from code repositories and programming documents, CodeBERT demonstrates enhanced effectiveness in software program training and source code analysis. Firstly, each node of program dependency graph which represents a statement of source code

in tokenized. Then each tokenized lines of code is given input to pre-trained codeBERT model and output contains contextual vector representation of each token. This phase has two steps : Tokenization and CodeBert Embedding.

- **Tokenization:** During the pre-training phase, input data is constructed by combining two segments using special tokens, [CLS], w1, w2, ..wn, [SEP], where [CLS] serves as a classification token. This input structure involves one segment representing natural language text and the other representing code from a specific programming language. The [CLS] token is a special token placed before the two segments. Following standard text processing in Transformer, natural language text is treated as a sequence of words and divided into WordPieces [96], while a code snippet is considered a sequence of tokens. The output of CodeBERT includes contextualized vector representations for each token, encompassing both natural language and code, as well as the representation of [CLS], which serves as a summarized representation.

- **CodeBert Embedding:** In the context of our research, a snippet of source code is extracted as a PDG from graph generation phase. Processing the PDG during this feature extraction phase includes obtaining a comprehensive code representation which is vital for subsequent model construction. We leverage CodeBert [94], designed to extract code features using a transformer-based method, specifically effective for source code-related tasks with self-supervised learning objectives. CodeBert takes a PDG of a single function of source code as the raw input and then processes and splits it into individual statements S_i. Each statement is tokenized using CodeBERT's pretrained BPE tokenizer. After collecting $S = S_1, S_2, ..., S_n$ the entire function is input into CodeBERT, enabling the acquisition of function-level and statement level code representations. This CodeBert embedding converts each sentence into a equivalent vector representation having dimension size

768. Finally an embedded function-level graph with statement level features is extracted from this feature extraction phase.

4.2.3 Image Construction

After completing CodeBert embedding, each node in the new embedded PDG is substituted with its corresponding embedded vector. At first Program Dependency Graph is generated by using joern tool from a function as shown in Figure **??**. Each node of a program dependency graph represents a statement and the program dependency graph is passed through pre-trained CodeBERT model and the embedded program dependency graph is found from which image is constructed. Each node in the PDG corresponds to a line of code in the vulnerability. Red lines and blue lines show the data flows and control flows between different lines of code in the function, respectively. To simply display the PDG of the vulnerability, we replace each line of code with a numbered circular node. Eight lines of code in the function correspond to eight circular nodes. In graph theory, an adjacency matrix is a square matrix used to represent a finite graph. Therefore, to represent the PDG of the vulnerability, we compute the corresponding adjacency matrix and describe it in Figure 4.3.

The elements of the matrix indicate whether pairs of nodes are adjacent or not in the graph. Since our PDG is a directed graph, an element represents the number of directed edges between two different nodes. For example, node 3 has two edges pointing to node 4, while node 4 has no edges pointing to node 3. Therefore, the corresponding values between node 3 and node 4 in the matrix are two and zero, respectively. After obtaining the adjacency matrix of the PDG, we find that when we sum a row in the matrix, the value obtained corresponds to the out-degree of the node. Moreover, when we sum a column, the value obtained is the in-degree of the node. For example, the sum of the row of node 4 is three (i.e., 1 + 1 + 1 = 3), which indicates that the out-degree of node 4 is three. To better illustrate

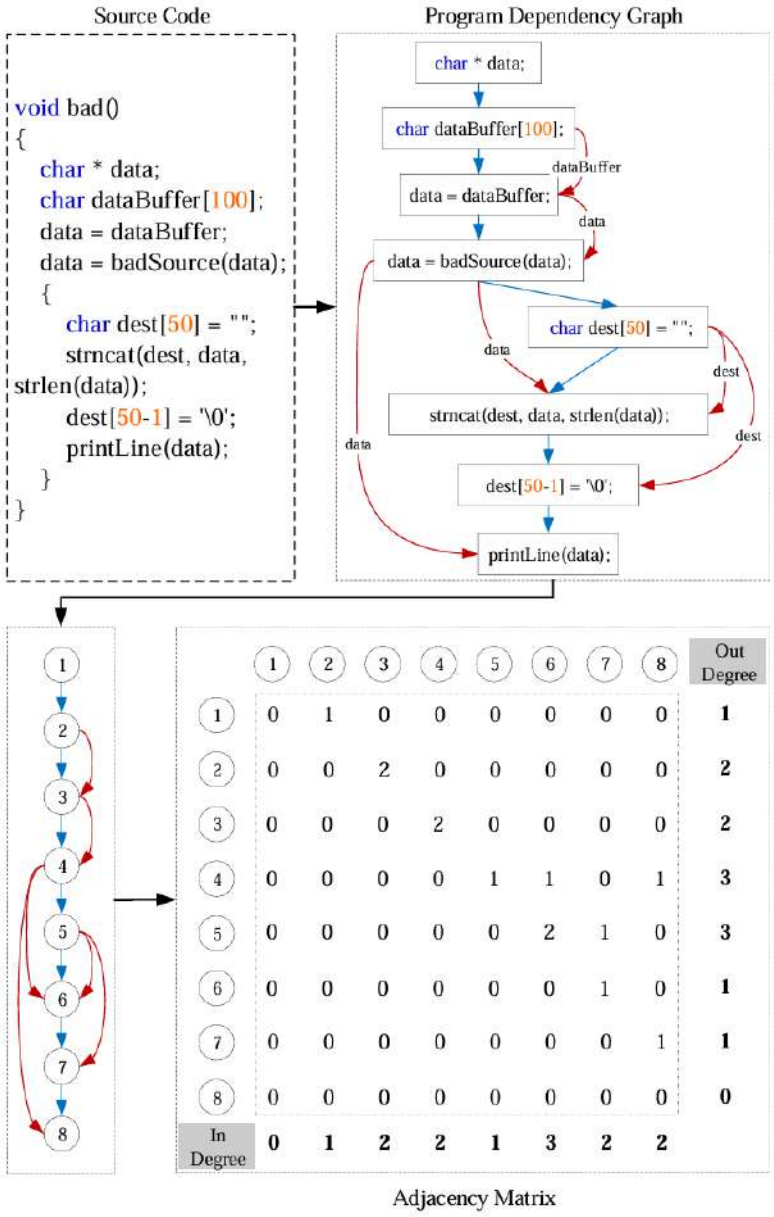


Figure 4.3: The Program Dependency Graph (PDG) and corresponding adjacency matrix of a vulnerable function

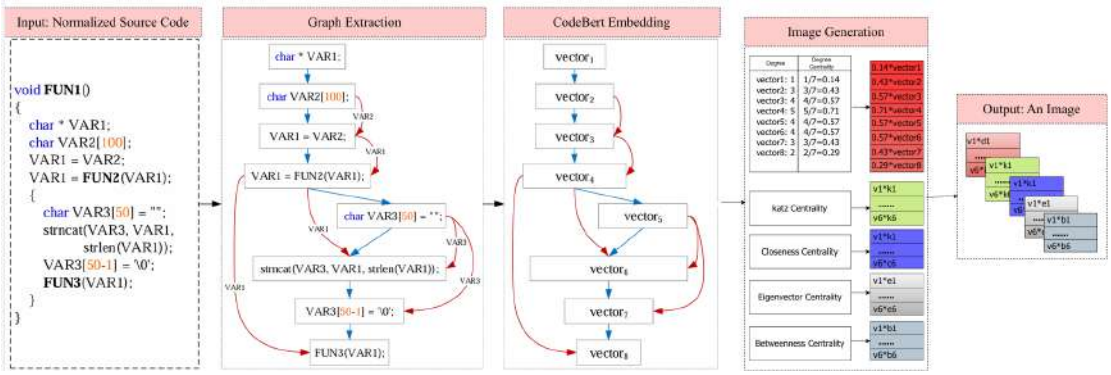


Figure 4.4: The Program Dependency Graph (PDG) and corresponding adjacency matrix of a vulnerable function

our insight, we present the in-degrees, out-degrees, and degrees of all lines of code of the vulnerability in 4.4. Through the results in 4.4, we find that the degrees of different lines of code are basically different. It is reasonable because there are different relationships. (i.e., control-flows and data-flows) between different lines of code, and the vulnerability is triggered based on these relationships. If we directly process these codes by treating them as text, the degrees of all lines of code are one, which may decrease the vulnerability detection accuracy. After graph extraction and sentence embedding, we can obtain a new PDG where each node is a vector representation. In this phase, we aim to efficiently convert the new PDG into an image while considering the contributions of different lines of code to program semantics. To complete our purpose, we regard the new PDG as a social network and apply social network centrality analysis to obtain the importance of all lines of code. Centrality [93] concepts were first developed in social network analysis [97, 98] whose original purpose is to measure the importance of a node in the network. Centrality analysis has been used in many different areas (e.g., biological network and transportation network), these successful applications have validated the effectiveness of network analysis. In fact, there have been proposed many different types of centrality to quantify the importance of a node in a network from different aspects, for example: Degree centrality, Katz Centrality, Closenness Centrality, Betweenness Centrality, Eigenvector Centrality.

These five centralities can compute the importance of all lines of code in a function from three different aspects. By this, we can achieve complete consideration of different lines of code's contribution to a function's program semantics. As shown in Figure 4.4, we first perform degree centrality analysis on all nodes in the new PDG to collect the degree centralities of all nodes (i.e., vectors). All vectors are then arranged one by one according to the number of lines of code after multiplying by the corresponding degree centrality. We call these arranged new vectors the "degree channel". Similarly, after applying katz, closeness, betweenness, eigenvector centrality analysis on the new PDG, we can obtain the other four "channels" which are "katz channel", "closeness channel", "Betweeness channel", "Eigenvector channel".In this study the five centrality metrics which are considered are discussed below.

1. **Degree Centrality:** Degree centrality for a node is determined by counting both the incoming edges (indegree) and outgoing edges (outdegree), essentially representing the number of links associated with the node. To obtain the standardized value, the highest degree is divided by $N-1$ as shown in Equation 4.1, where N represents the total number of nodes in the network.

$$x_{\mathrm{i}} = \frac{Degree(i)}{N-1} \tag{4.1}$$

2. **Katz Centrality:** It calculates the centrality value of a node by taking into account the centrality of its neighboring nodes. This is determined by summing the number of directly connected nodes and the number of indirectly connected nodes through these immediate neighbors. The Katz centrality of a node 'n' can be expressed as follows in equation 4.2:

$$x_n = \alpha \sum_j A_{nj} x_j + \beta \tag{4.2}$$

In the given expression, A represents the Adjacency Matrix, and α, β and λ stand for the Attenuation Factor, Initial Centrality Controller and Eigenvalues of the graph G respectively. These parameters are instrumental in assigning increased weight to nearby neighbors (via β), while simultaneously applying a penalty to distant links (utilizing α) as in equation 4.3. Notably, α must be smaller than the inverse of the largest eigenvalue of the adjacency matrix to ensure the proper computation of Katz centrality allowing for an accurate measurement by considering the influence of various factors.

$$\alpha < \frac{1}{\lambda_{max}} \tag{4.3}$$

3. **Closeness Centrality:** Closeness centrality measures how close a node is to every other node in the graph. It is computed by averaging the shortest path lengths between the nodes within the graph. A node with a smaller average distance signifies greater closeness to the center of the graph. The average distance is essentially the inverse of the node's proximity centrality among all $x - 1$ accessible nodes.

$$C(x) = \frac{x - 1}{\sum_{n=1}^{x-1} d(n, x)} \tag{4.4}$$

In this equation 4.4, where x represents the total number of nodes in the graph and $d(n, x)$ denotes the distance between nodes n and x.

4. **Eigenvector Centrality:** Eigenvector centrality assesses a node's overall impact based on the centrality of its neighboring nodes. In accordance with the given definition, the i-th element in the n vector corresponds to the eigenvector centrality of the i node.

$$An = \lambda n \tag{4.5}$$

Here, in equation 4.5 the eigenvalue λ is associated with the adjacency matrix A.

5. **Betweenness Centrality:** Betweenness centrality calculates the shortest paths for nodes within a graph, representing the overall percentage of total pairs of shortest routes that traverse through a specific node α a. Mathematically, it is expressed as:

$$C_B(a) = \sum_{s,t \in N} \frac{\sigma(s,t|a)}{\sigma(s,t)} \tag{4.6}$$

In this formula of equation 4.6, N signifies the set of nodes, $\sigma(s,t)$ denotes the count of shortest paths between nodes s and t, and $\sigma(s,t|a)$ represents the count of paths passing through node a other than nodes s and t.

Due to the typical composition of a RGB image with three channels: Red, Green, and Blue, prior studies utilize three different centrality measures: degree centrality, Katz centrality, and closeness centrality. These centrality measures provide insights into the importance of various lines of code within a function from distinct perspectives. By incorporating additional two centrality measures: Betweenness centrality and Eigenvector centrality we can achieve a more comprehensive assessment of each line of code's contribution to the overall program semantics because Eigenvector centrality assesses the importance of a node based on its connections to other highly central nodes, offering insight into critical elements within a PDG. Betweenness centrality identifies nodes that act as key intermediaries, revealing critical pathways for information flow or dependencies between different modules or functions in a program. Both measures provide valuable insights into the structural and functional importance of nodes within a network, aiding in understanding program semantics and dependencies in source code.

In summary, we compute the centrality values for all nodes in the new embedded PDG. Subsequently, we arrange the resulting vectors in accordance with the number of lines of code multiplied by the corresponding centrality measure. These arranged vectors represent the "Degree channel" "Katz channel", "Closeness channel" , "Betweennes channel" and "Eigenvector channel" Additionally, by applying betweenness centrality and eigenvector centrality analysis we obtain two more channels. Ultimately, the combination of these five channels is utilized to generate the final image representation from a source code function.

4.2.4 Vulnerability Classification

Deep learning algorithms have outperformed previous technologies in various domains, such as speech and image recognition. By utilizing effective hierarchical feature extraction methods and unsupervised or semi-supervised feature learning, deep learning presents the advantage of replacing manual feature acquisition. In the domain of image processing, Convolutional Neural Network (CNN) has gathered significant attention. This is attributed to its ability not only to eliminate the need for manual image preparation but also to enable users to extract features at a level comparable to human capabilities. Following the image generation phase, a function's source code is transformed into an image. To identify vulnerabilities, the initial step involves training a CNN model on an image. While CNN typically utilizes input images of equal size, it encounters variations in the number of lines of code required for each function. Hence, adjustments are necessary. To identify vulnerabilities, the initial step involves training a CNN model with images. While CNN typically utilizes equal-sized input images, the varying number of statements in each function within our input dataset necessitates an adjustment. To determine a threshold for fixed-size images, an analysis is conducted to assess the number of lines of code in each function. Figure 4.5 shows the Cumulative Distribution Function (CDF) of the total number of statements (lines of code) in

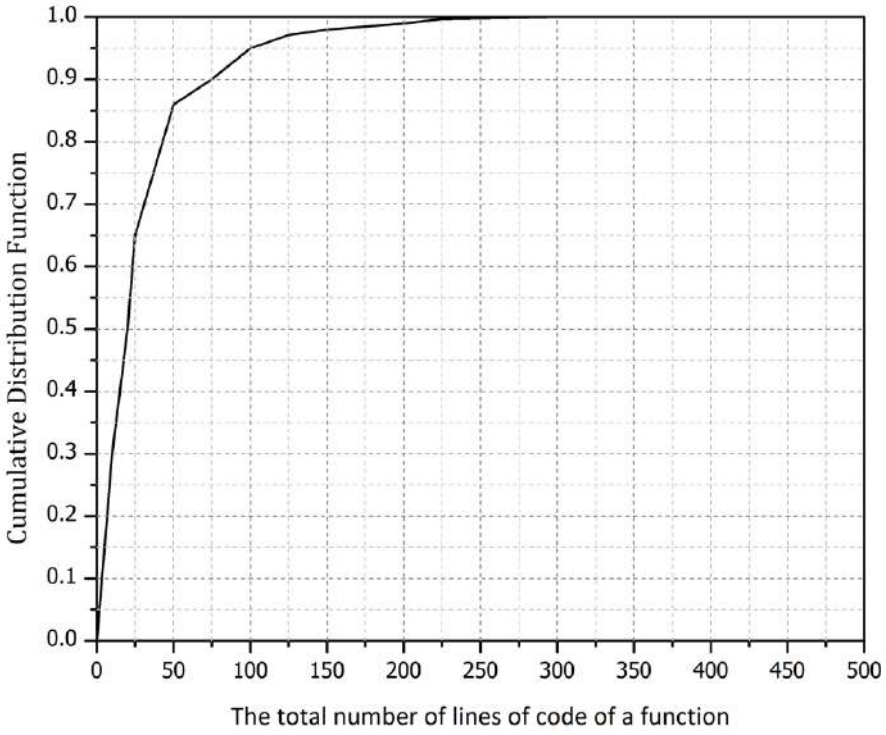


Figure 4.5: Cumulative Distribution Function of total number of lines of code of a function

functions from the input dataset is calculated. It is observed that over 99% of the functions contain fewer than 200 statements. After experimenting with various threshold values (ranging from 40 to 200 statements) for vulnerability detection, a decision is made to set the cutoff at the first 100 statements of a function. This decision is based on considerations of detection accuracy and related runtime overhead. For functions with fewer than 100 statements, zeros are padded at the end of the vectors. In functions with more than 100 statements, the vector's tail is discarded. The input images are typically of sizes $3 \times 100 \times 768$, $4 \times 100 \times 768$, or $5 \times 100 \times 768$, where 3, 4, 5 denote the number of channels, and 786 denotes the dimensions of the embedded vector.

In Figure 4.6, the input image shape is specified as $5 * 100 * 768$, where 5 represents five distinct channels: "degree" (D),"Katz" (K),"Closeness" (C),"Eigenvector" (E), and "Betweenness" (B). These channels are created to incorporate different aspects. Following the generation of fixed-size images, they are utilized for train-

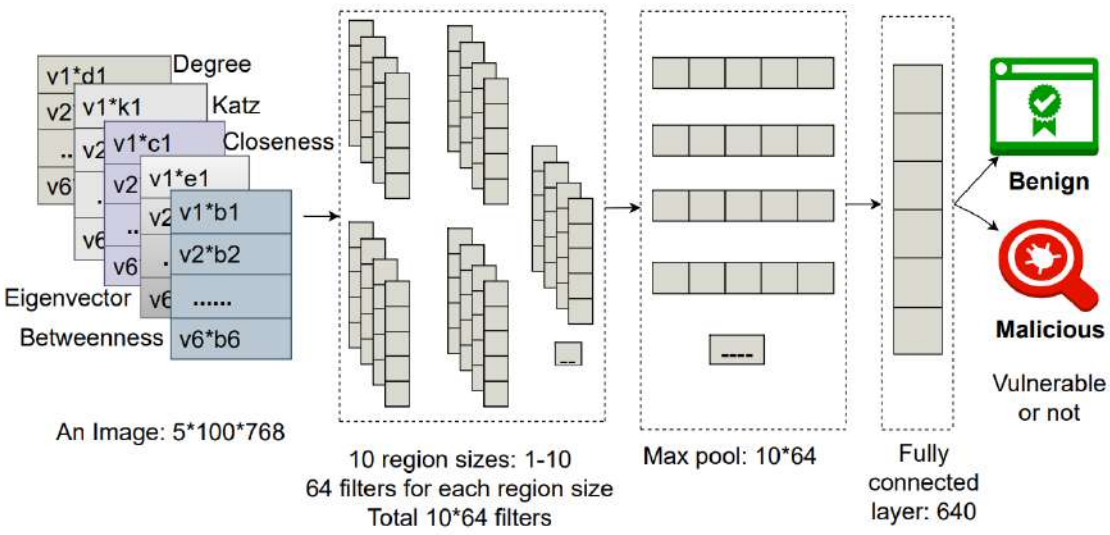


Figure 4.6: CNN Classification of VulBertCNN

ing a CNN model. The CNN model employs various convolution filters, each with a shape of $m \times 768$ as depicted in Figure 4.6. This design allows each filter to independently cover the entire capacity of the CodeBert embedding. The variable m denotes the size of the filter, indicating the number of consecutive sentences to be considered.

Table 4.1: Hyper-Parameter Settings In VulBertCNN

Hyper-Parameters	**Value**
Learning Rate	0.0001
Number of Epochs	100
Batch-Size	64
loss function	Cross Entropy Loss
Activation Function	ReLU
Optimizer	Adam

In our proposed method, we extract features from different parts of the image by employing various filter sizes ranging from 1 to 10, each with 64 feature maps. This process is followed by max pooling and the application of the Rectified Linear Unit (ReLU) [99] activation function across the entire model. The datasets are initially divided based on their original split. However, in cases where the

split information is unavailable, we adopt a default split of 80:10:10 for training, validation and testing sets respectively. The hyperparameters utilized in the CNN architecture are detailed in Table 4.1 where we took optimum epoch number 100 and it is shown by a loss curve where our model underfits at 50 and overfits from 150. Training and validation both loss curve are shown in Figure 4.7.

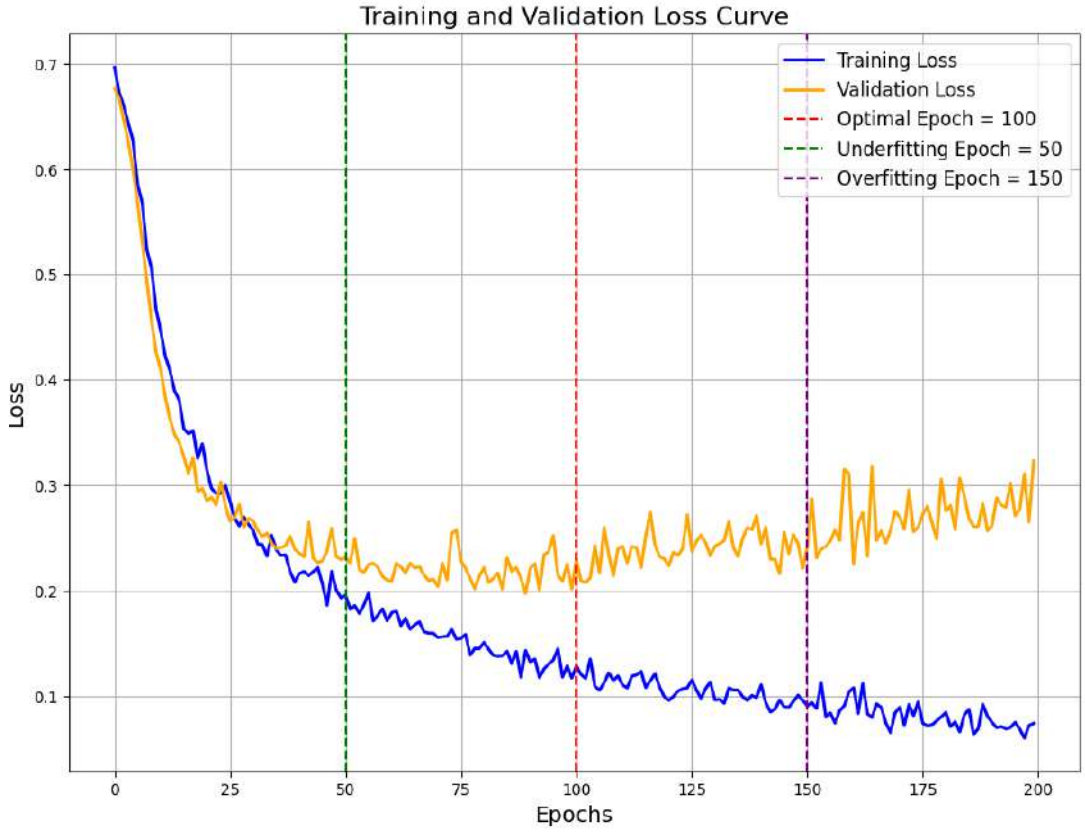


Figure 4.7: Loss Curve with respect to Number of Epochs

4.3 Experiment and Result Analysis

In this section, we perform experiments to compare the detection accuracy of VulBertCNN with state-of-the-art solutions TokenCNN [64], VulCNN [85] and ASVD [91]. Before exploring the effectiveness of VulBertCNN, we provide below the details on the implementation specifics.

4.3.1 Implementation

The proposed method described in the above section is implemented in Python (version 3.11.5). We conducted experiments using an ASUS TUF Gaming laptop featuring an Intel Core i7-8th generation CPU on a Windows server. The processor in the laptop has six cores, each with a maximum operating frequency of 2.5 GHz. The source code of VulBertCNN is made publicly available in GitHub [1]. The details of dataset used and results of VulBertCNN are discussed in following subsections.

4.3.2 Datasets

To evaluate the effectiveness of VulBertCNN, we use two benchmark vulnerability datasets: SARD [100] and Big-Vul [82] in alignment with state-of-the-art methods. The datasets were collected from two sources: National Institute of Standards and Technology (NIST) [101] and Common Vulnerability Exposure (CVE) [9] database. The details of the experimental datasets are presented below in Table 4.2.

Table 4.2: Details of Datasets

Dataset	**Total**	**Vul**	**Non-Vul**	**Vul** (%)
SARD	40,584	13,684	26,900	33.71
Big-Vul	188,770	10,670	178,100	5.65

SARD: The software assurance reference dataset (SARD) dataset encompasses a significant volume of production, synthetic, and academic security flaws (i.e., bad functions) and non-vulnerable function (i.e., good functions). This study concentrates on identifying vulnerabilities specifically in C/C++, thus we exclusively target functions written in C/C++ within SARD. The dataset from SARD

[1] `https://github.com/Rabaya-Sultana-Mim/Automated-Software-Vulnerability-Detection-in-Statement-Level-using-Vulnerability-Reports`

comprises 12,300 instances of vulnerable functions and 21,000 instances of non-vulnerable functions. Recognizing the potential lack of realism in synthetic programs within SARD, we supplement our data with another dataset derived from real-world software. For real-world vulnerabilities, we utilize the National Vulnerability Database (NVD) as our primary data source, resulting in 1,384 vulnerable functions from various open-source C/C++ projects. To complement this, we randomly select a subset of non-vulnerable functions from the dataset which contains non-vulnerable functions from diverse open-source projects, ensuring a balanced representation. The ultimate dataset comprises 13,684 functions with vulnerabilities and 26,900 functions without vulnerabilities.

Big-Vul: We utilized the benchmark dataset Big-Vul, created by Fan et al. [82]. This dataset contains reliable and comprehensive code vulnerabilities directly associated with the publicly accessible CVE database. Notably, the construction of this dataset involved a significant investment of manual resources to ensure its high quality. Additionally, it stands out for its substantial scale, being one of the most extensive vulnerability datasets available. The dataset is compiled from 348 open source GitHub projects spanning from 2002 to 2019, covering 91 distinct Common Weakness Enumeration (CWE) categories.

This comprehensive dataset includes approximately 188,800 C/C++ functions, with 5.9% identified as vulnerable, equivalent to 10,700 vulnerable functions. It offers detailed ground-truth information at the function level, specifying which functions within a codebase are susceptible to vulnerabilities.

4.3.3 Evaluation Metrics

To effectively evaluate model predictions, we established and defined ground truth values as follows: True Positive (TP) is the number of vulnerable samples correctly detected as vulnerable. True Negative (TN) is the number of non-vulnerable samples correctly identified as not vulnerable. False Positive (FP) is the number of

non-vulnerable samples incorrectly classified as vulnerable. False Negative (FN) is the number of vulnerable samples erroneously identified as not vulnerable. Hence, we employ four metrics for our experiments, namely:

Accuracy: This metric indicates the number of samples that are correctly classified into their respective classes (e.g., positive or negative labels for vulnerable or non-vulnerable function).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.7}$$

Precision: Precision is the ratio of correctly classified adversarial examples to the total number of adversarial examples.

$$Precision = \frac{TP}{TP + FP} \tag{4.8}$$

Recall: Recall refers to the ratio of incorrectly classified adversarial examples.

$$Recall = \frac{TP}{TP + FN} \tag{4.9}$$

F1 Score: F1 Score is a metric that combines precision and recall, providing a simple and convenient way to compare our results.

$$F1 - Score = 2 * \frac{Recall * Precision}{Recall + Precision} \tag{4.10}$$

4.3.4 Detection Performance Evaluation

To assess the effectiveness of the proposed technique, we conducted experiments with a total of 16 combinations of centralities on the benchmark SARD and BigVul dataset. As we observed in Figure 4.5 that more than 99% of the functions have thresholds below 200 lines of code, we initiated our evaluations by selecting 10 thresholds (40, 60, 80, 100, 120, 140, 160, 180, and 200 lines). The experimental

results of our proposed method with different combination of centralities using SARD dataset are presented in Figure 4.8. As TokenCNN, VulCNN and ASVD all are evaluated using this dataset that is why to better compare the detection performance of our VulBertCNN approach it is evaluated with the same dataset.

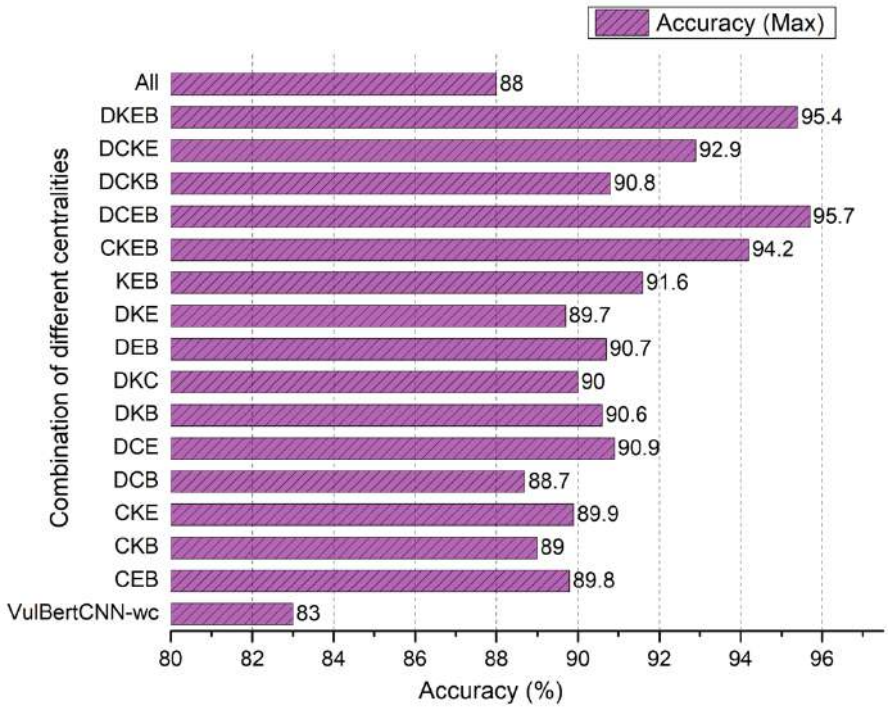
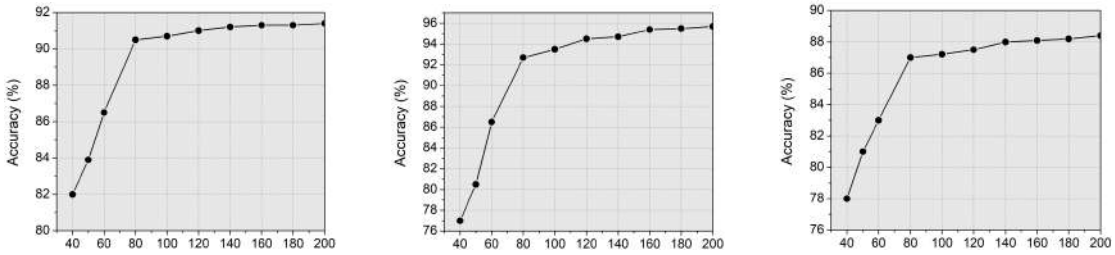


Figure 4.8: Comparison of the detection accuracy for different centralities using CodeBert embedding and CNN on SARD dataset

We conducted CodeBERT embedding and centrality analysis on the graph derived from the source code. Since the centralities align with the channels of an image, we conducted our experiment considering three, four, or five channels representing various combinations of the aforementioned five centralities. The experiment consisted of 100 epochs and the final outcomes of the maximum detection performance in each of the combination using three, four, five or without centralities are illustrated in Figure 4.9. We examined the following four specific cases to observe the influence of centrality on vulnerability detection.

Vulnerability Detection with CodeBert Embedding using 3 Centralities: We experimented with different combinations of centralities to enhance vulnera-



(a) Number of lines of code analyzed with 3 centralities

(b) Number of lines of code analyzed with 4 centralities

(c) Number of lines of code analyzed with 5 centralities

Figure 4.9: Maximum vulnerability detection performance in VulBertCNN with CodeBert embedding and different centralities

bility detection. Initially, we explored combinations of three centralities, selecting from a pool of five. The optimal performance, with a maximum accuracy of 90.9%, was achieved using the DCE (Degree-Closeness-Eigenvector) centralities, as illustrated in Figure 6(a). In comparison, the state-of-the-art VulCNN detector, employing the DKC (Degree-Katz-Closeness) combination, achieved a maximum accuracy of 83%.

Vulnerability Detection with CodeBert Embedding using 4 Centralities: An image constructed with four centralities corresponds to a four-channel image. The highest accuracy, reaching 95.7% was observed with the DCEB combination (Figure 6(b)). Additionally, DCEB and CKEB demonstrated substantial detection performance in comparison to other combinations achieving accuracies of 95.7% and 95.4%, respectively.

Vulnerability Detection with CodeBert Embedding using 5 Centralities: For the case of five centralities, representing an image with five channels, the maximum detection accuracy reached 88% (Figure 6(c)), which was not as significant as the performance achieved with four centralities.

Vulnerability Detection with CodeBert Embedding Without using Centrality: To assess the impact of centrality analysis on vulnerability detection, we conducted another experiment which is Vulnerability Detection with CodeBert Embedding using CNN without centrality (VulBertCNN-wc). This experiment

achieved only 83% accuracy. However, upon incorporating centrality analysis, the accuracy significantly improved to approximately 96%. This underscores the importance of considering centrality measures in code lines to enhance the accuracy of vulnerability identification in software.

In summary, the analysis highlights that the best accuracy (95.7%) was achieved with combinations of four centralities, specifically CKEB, DCEB, and DKEB in SARD dataset. Furthermore, VulBertCNN emphasizes the significant role of CodeBERT, eigenvector and betweenness centrality measures in effective vulnerability detection.

4.3.5 Baseline Methods Comparison

We evaluate VulBertCNN's performance in vulnerability detection with three state-of-the-art detection approaches based on image processing which are TokenCNN, VulCNN and ASVD.

Russell et al. [64] annotate the source code and transform it into the corresponding matrix. They utilize convolutional neural networks, integrated learning, and a random forest classifier for detecting vulnerabilities in the code. In TokenCNN no semantic relation is considered here only simple lexical analysis is done on source code which is then given input in CNN. Thats why it cannot accurately detect vulnerability from source code.

Wu et al. [85] developed VulCNN, a vulnerability detection method using CNN. They used a dataset with both vulnerable and non-vulnerable C/C++ functions, created program dependency graphs (PDGs), and applied a sentence embedding technique i.e., Sent2Vec [102] to each statement. Utilizing centrality techniques, they transformed functions into images and trained a CNN model. In a case study involving projects such as Libav, Xen and Seamonkey, VulCNN detected 73 previously unknown vulnerabilities. VulCNN achieves 83% accuracy but it only considers three centralities there are other centralities yet to be explored.

Mim et al. [91] developed ASVD with comparison to VulCNN method with sentence embedding and five centralities. The detection accuracy obtained in ASVD is 88% which outperforms the accuracy of VulCNN (83%) by 5% which we targeted to improve further by leveraging CodeBert embedding instead of Sent2Vec embedding approach.

To evaluate the efficiency of our embedding approach utilizing the CodeBERT model (as detailed in Section 3) we performed experiments to assess the performance of both the Sent2vec and CodeBERT embedding models. The results were then compared with those of three related studies, TokenCNN, VulCNN and ASVD. Tables 4.3 provides a summary of the evaluation results in terms of Accuracy, Precision, Recall and F1 score.

Table 4.3: Evaluation of VulBertCNN using CodeBert Embedding

Dataset	**Methods**	**Accuracy**	**Precision**	**Recall**	**F1 Score**
SARD	TokenCNN	78.6	55.7	71.6	61.9
	VulCNN	83.4	86.9	81.9	84.3
	ASVD	88.2	89.5	85.3	87.3
	VulBertCNN	**95.7**	**90.4**	**87.3**	**89.0**
Big-Vul	TokenCNN	61.4	50.6	79.7	62.6
	VulCNN	80.3	83.3	78.2	82.3
	ASVD	89.6	94.2	86.5	90.2
	VulBertCNN	**91.8**	89.1	**87.4**	88.2

Table 4.3 indicates that our embedding approach which is utilizing the CodeBERT model has demonstrated significantly superior evaluation outcomes when compared to existing state-of-the-art approaches. Our method achieved an accuracy of 95.7% and F1 score of 89% on SARD dataset and 91.8% accuracy on Big-Vul dataset showcasing a substantial enhancement over other existing methods. Moreover, both Precision (90.4%) and Recall (87.3%) metrics exhibited im-

provements in SARD dataset. While the F1 measure with CodeBERT (88.2%) on Big-Vul did not outperform ASVD, it displayed improvement in accuracy when contrasted with our experimental evaluation using Sent2Vec (90.2%).The lower precision and recall values observed for the proposed approach in the Bigvul dataset compared to the Sard dataset can be attributed to several key differences between synthetic and real-world data. Sard, being a synthetic dataset, is designed with controlled conditions that may facilitate more straightforward model training and evaluation. In contrast, Bigvul consists of real-world vulnerability data, presenting inherent complexities such as diverse data quality, variable class distributions, and nuanced vulnerabilities that may be more challenging for the model to accurately detect. These differences underscore the importance of adapting models and strategies to handle the complexities and nuances present in real-world datasets like Bigvul, which can impact performance metrics such as precision and recall. In these scenarios, the accuracy metric proves more suitable for evaluating vulnerability detection performance.

In summary, the result analysis reveals that our embedding method using CodeBERT outperforms the Sent2Vec embedding method used by VulCNN and ASVD. By combining CNN with CodeBERT embeddings and centrality analysis, we achieved a significant improvement in the F1-score increasing it from 62.6% to 89% and detection accuracy from 78.6% to 95.7% which is about 21.7% and 26.3% improvement compared to the state-of-the-art approaches.

4.4 Threats to Validity

In this section, we discussed below the potential threats which may affect the validity of this study.

External Validity: External validity presents challenges in generalizing study findings. Differences in programming languages (e.g., C/C++ vs. Java or Python)

and various code vulnerabilities are key considerations. Generalizing findings to a broader range of open-source and industrial systems requires careful handling due to the complexity of obtaining and analyzing diverse industrial systems. To overcome these challenges, future plans include collecting data from industrial systems across industries and countries to enhance dataset diversity.

Internal Validity: Internal validity concerns the accuracy of causal inferences in our vulnerability detection research. Variations in dataset characteristics and algorithmic parameters pose potential confounding issues. To enhance internal validity, we employ a rigorous experimental design, exercise precise variable control, and conduct sensitivity analyses. For instance, strategic dataset partitioning over time ensures a robust assessment of temporal dynamics, contributing to the internal validity of our vulnerability classification methodology.

Construct Validity: Construct validity focuses on the tool used for extracting Program Dependency Graphs (PDGs), namely Joern. While commonly used, Joern may have inherent flaws. Despite this, we chose Joern for PDG extraction and performed manual reviews to identify and address any issues. There's a potential threat from modifying baseline approaches, but we mitigate this risk by retrieving the original source code directly from GitHub repositories associated with analyzed techniques.

Criterion Validity: In vulnerability detection, metrics like precision, recall, and F-measure quantify the alignment of identified vulnerabilities with actual vulnerable functions. High value of criterion validity indicates our algorithm effectively predicts vulnerabilities in line with widely accepted standards.

4.5 Summary

This study proposes an automated Software **Vul**nerability Detection with Code**B**ert and **C**onvolutional **N**eural **N**etwork named VulBertCNN, aiming to overcome the

limitations of state-of-the-art individual text and graph-based approaches in vulnerability detection. Here a vulnerability detection approach is proposed which focuses on integrating Codebert embedding model with multiple centralities in image generation from PDGs to assess the overall impact of each line of code within a function, thereby determining its vulnerability status. The evaluation involves the generation of 16 centrality combinations derived from 5 centralities, revealing that the highest accuracy is attained with a combination of 4 centrality measures. This achieves an accuracy surpassing the previous state-of-the-art techniques from 78.6% to 95.7% and F1-score increasing it from 62.6% to 89%. It is observed that leveraging codebert embedding with CNN emerges effective role in vulnerability detection in function level but to more narrow down findings in granular level i.e., statement level analyzing only source code is not sufficient rather more contextual information is necessary along with source code and vulnerability report can provide those essential information. Because when a vulnerability is reported by users understanding vulnerability reports is crucial for identifying statement-level vulnerabilities in code particular where fixing should be done. These reports provide essential information like vulnerability description, vulnerability type, unexpected behaviour of system and in particular that helps to narrow down potential vulnerabilities within a function. Hence, the following chapter outlines a technique for identifying vulnerabilities at the statement level, utilizing a ranked list of statements by comparing source code with the details of vulnerability reports collected from National Vulnerability Database and strengthen our software's security.

Chapter 5

A Statement Level Vulnerability Detection Technique using Vulnerability Reports

This chapter introduces a technique named Vulnerable Functions and Statements Detector (VFSDetector), aimed at minimizing search space and then particular vulnerable statements within source code using vulnerability reports. It maximizes the useful information of each statement by looking at its connections with other statements. It also uses Information Retrieval (IR) techniques to compare vulnerability reports with the source code and find similarities. To understand why VFSDetector is important, the chapter first explains the limitations of current statement level vulnerability detection methods and gives an overview of VFSDetector. Then, it goes into detail about how VFSDetector works. The chapter also analyzes the results by answering two questions: how does the combination of vulnerability report and source code impact in statement level vulnerability detection and does considering connected statements improve vulnerability detection performance Finally, the chapter concludes by discussing its contributions and future works.

5.1 Introduction

As software vulnerabilities make software systems increasingly vulnerable to attack and damage, which raises security concerns [6, 103]. Developers need to spend a lot of time in identifying vulnerable function from large codespace when a new vulnerability is reported. Identifying vulnerable functions effectively is a perquisite of writing a patch for the reported vulnerability. This is essential for enhancing software security by addressing vulnerabilities to mitigate potential risks and threats more effectively at earliest time.

Existing studies have focused on detecting software vulnerabilities employing text-based [64], [104], [66] or graph-based [47], [105] approaches. These approaches either treat source code as plain text or apply graph analysis by representing the source code as graph. In practice, prior text-based studies treat source code as plain text and apply static program analysis or natural language processing. However, the performance of these approaches is not optimal for disregarding the source code semantics. On the other hand, graph based approaches conduct program analysis which represent the source code semantics as a graph, and then apply graph analysis methods such as Graph Neural Networks (GNN) [106] to identify vulnerabilities. Although these graph-based approaches are more efficient at identifying vulnerabilities taking into account the semantic relationship of various lines of source code but their scalability is substantially less than that of text-based approaches.

However, existing studies have underutilized vulnerability reports which can be a good source of contextual information to detect vulnerability in source code. In this context, the current study aims to verify the role of vulnerability reports in identifying vulnerable function. Vulnerability report can contain contextual information about a vulnerability which may be used to identify vulnerable functions. When a function is vulnerable against a scenario some keywords should be shared

between that function and the vulnerability report. These motivate the authors to study whether vulnerable functions can be identified by analysing the source code and vulnerability report.

For this purpose, our study introduces an information retrieval-based approach named Vulnerable Functions and State ments Detector (VFSDetector), aimed at identifying vulnerable functions and pinpointing vulnerable statements within source code using vulnerability reports. It takes all source code files of a system as input. First, it extracts all source code functions of that system. Then static analysis is performed to extract the contents of those functions. Several text pre-processing analysis such as tokenization, stopwords removal, multiword splitting, semantic meaning extraction and lemmatization are applied on these source code along with the vulnerability report to produce code and report corpora. In addition, programming language specific keywords is removed for generating code corpora. Finally, to rank the vulnerable functions and statements, similarity scores are measured between the code corpora of the functions and report corpora by a modified version of traditional Vector Space Model (tVSM) where larger methods get more weight while ranking.

In experiments, as an initial study ten Common Vulnerabilities and Exposures (CVE) reports are chosen randomly from six open source GitHub repositories. Based on the commit link available in reports we crawled the corresponding projects before the vulnerability was patched. The result analysis shows that VFDetector ranks the vulnerable functions at the first position in 40% cases, whereas it ranks the actual vulnerable function within top 5 in 90% cases and within top 7 in 100% cases. Then it is evaluated on 1200 vulnerability reports from 14 prominent open source projects. Results indicate that VFSDetector accurately ranks the genuine vulnerable statements in 1st position in 40% cases and upto 100% cases in Top 20 ranking. It also demonstrates its outperformance compared to the base line methods like LineVul, COSTA, IVDetect, LineVD across

all top-k values with maximum accuracy, precision and recall of 70.43%, 42.37% and 95.81% which underscores its effectiveness in accurately identifying vulnerable statements.

It is evident from the results that VFDetector performs promisingly in detecting vulnerable functions against a vulnerability report in a large scale software systems. It is also observed that in Top 5, 10, 15 and 20 ranking, the functions which ranks above the actual vulnerable function are the related functions of that vulnerability which acquires higher similarity. It guides a developer to patch those related functions too in order to mitigate that vulnerability from the system.

5.2 Methodology

This study proposes an approach VFSDetector which detects vulnerable functions and statements from huge volume of files of a large software system using vulnerability reports. The overall process of this approach consist of three distinct steps and those are Source Code Corpora Generation, Vulnerability Report Corpora Generation, Vulnerable Functions Ranking and Vulnerable Statements Ranking. Each of these steps encompasses a series of tasks as illustrated in Figure 5.1 and 5.2. At first, all files and their corresponding functions are extracted from a particular version of a software system. Then these source code is processed to create code corpora. Similarly vulnerability report is processed to produce report corpora. Finally, similarity between the report and code corpora is measured using tweaked Vector Space Model (tVSM) to rank the vulnerable source code functions and corresponding function's vulnerable statements.

5.2.1 Function Level Vulnerability Detection by Minimized Search Space using Vulnerability Reports

In this section, we implement the model described in Figure 5.1 by developing a methodology aimed at enhancing ranking accuracy through focusing on relevant functions of the source code for a vulnerability and thus the search space is minimized. The proposed vulnerability detection technique's methodology is detailed in the following subsections.

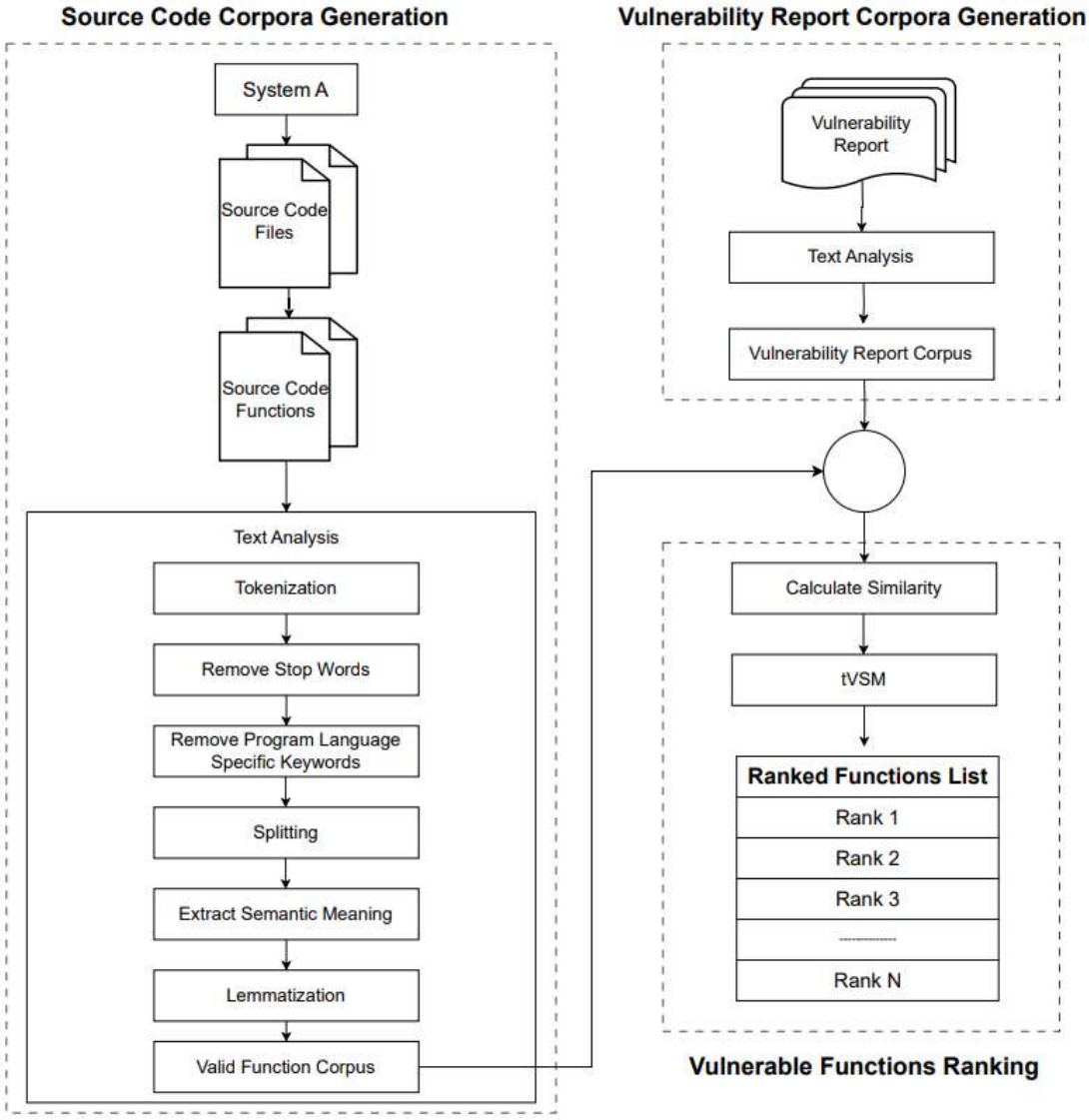


Figure 5.1: Overview of VFSDetector in Function Level

The overall process of improving vulnerability detection accuracy can be sum-

marized as follows. Extracting relevant information by disregarding irrelevant methods from a large solution space can significantly boost the accuracy of vulnerability detection. Initially, all files and their corresponding functions are extracted from a particular version of a software system. This is followed by static analysis to examine the contents of these extracted functions. The contents of the extracted functions are processed to create code corpora via static analysis. Since vulnerability reports only contain textual information related to a specific vulnerable scenario, static analysis is performed once more to process the vulnerability report. Finally, the generated corpora from the vulnerability report and the source code are matched, and based on this matching, the source code functions are ranked. The entire function level vulnerability detection process can be divided into three steps:

- Source Code Corpora Generation
- Vulnerability Report Corpora Generation
- Vulnerable Functions Ranking

Each step involves a series of tasks, as illustrated in Figure 5.1.

5.2.1.1 Source Code Corpora Generation

Source code corpora consist of source code terms used to assess similarity with vulnerability report corpora. Therefore, the precision of code corpora generation directly impacts the precision of matching, consequently enhancing the accuracy of vulnerability localization. In this step all the folders are extracted from a system with their corresponding C/C++ files. From each of these files all functions are extracted automatically in individual C files which ensures function level analysis. For Example: CVE-2014-2038 of Linux version 3.13.5 consist of 15,675 files which has total 229,682 functions.

This stage generates a vector of lexical tokens by doing lexical analysis on every source code file. There are unnecessary tokens in source code which do not contain any vulnerability related information. These tokens are discarded from source code such as programming language specific keywords (e.g., int, if, float, switch, case, struct), stop words (e.g., all, and, an, the). Many words in the source code may include multiple words. For example, the term "addRequest" consists of the keywords "add" and "Request". Mutiwords are separated using multi word identifier. Furthermore, statements are divided according to certain syntax-specific separators like ' , ', '=', '(', ')', '{', '}', '/', and so on. WordNet[1] is used to derive each word's semantic meanings because a term might have more than one synonym. In specific cases, developers and Quality Assurance (QA) personnel may employ different terminology, even though they are referring to the same scenario with equivalent meanings. For example, the term 'finalize' may have multiple synonyms such as 'conclude' or 'complete.' When describing a situation, if a developer uses 'finalize' but QA opts for 'conclude', it's challenging for the system to identify these variances without considering the semantic meanings of these words. Therefore, the extraction of semantic meaning is crucial in achieving accurate rankings for vulnerable functions.

The final stage of code corpora generation incorporates WordNet lemmatization, a technique that normalizes words to their base or dictionary form. WordNet lemmatization utilizes the comprehensive WordNet lexical database, organizing words into synonymous sets called synsets. This method identifies word lemmas based on the word's part of speech and context within WordNet, offering a more context-aware approach to lemmatization. As a result, it considers a word's meaning and contextual usage, allowing for precise reduction of words. For instance, it transforms "running" to "run" and "better" to "good" based on their meanings and parts of speech, unlike standard lemmatization that typically relies on suffix

[1] `https://wordnet.princeton.edu/`

removal.

5.2.1.2 Vulnerability Report Corpora Generation

A vulnerability report details the security weaknesses found in a software program. These reports are crucial for identifying and addressing potential security risks before they can be exploited by malicious attackers. Typically vulnerability reports are prepared by security analysts, penetration testers, cybersecurity researchers, incident response teams, and automated security tools to identify and mitigate security risks.

A software vulnerability report contains information like description about the vulnerability, severity rating, vulnerability identifier (CVE-ID), reference to additional sources which gives valuable insights about a software vulnerability issue. However, these reports can also include irrelevant terms such as stop words and words in various tenses (present, past, or future). To refine vulnerability reports, pre-processing is necessary. In the initial stage of vulnerability report corpora creation, stop words are eliminated. We apply WordNet Lemmatizer, similar to what's used for source code corpora generation, to generate refined report corpora containing only relevant terms.

5.2.1.3 Ranking Vulnerable Functions

In this step, relevant vulnerable functions are ranked based on the textual similarity between the query (report corpus) and each of the function in the code corpus. Vulnerable functions are ranked by applying tVSM. We employ tVSM, which modifies the Vector Space Model (VSM) by emphasising large-scale functions. In traditional VSM, the cosine similarity is used to measure the ranking score between the associated vector representations of a report corpus (r) and

function (f), according to Equation 5.1.

$$\text{Similarity}(r, f) = \cos(r, f) = \frac{\vec{V_r} \cdot \vec{V_f}}{|\vec{V_r}| \cdot |\vec{V_f}|} \tag{5.1}$$

Here, $\vec{V_r}$ and $\vec{V_f}$ are the term vectors for the vulnerability report *(r)* corpus and function *(f)* corpus respectively. Throughout the years, numerous adaptations of the $tf(t, d)$ function have been introduced with the aim of enhancing the VSM model's effectiveness. These encompass logarithmic, augmented, and Boolean modifications of the traditional VSM [107]. It has been noted that the logarithmic version can yield improved performance, as indicated by prior studies [108], [109], [110]. From that point of view, tVSM modified Equation 5.1 and uses the logarithm of term frequency (tf) and iff(inverse function frequency) to give more importance on rare terms in the functions. Inverse Function Frequency (IFF) measures the rarity of a function within a collection of programs, assigning higher importance to functions that appear in fewer programs. It is calculated as the logarithm of the total number of programs divided by the number of programs containing the specific function. Thus tf and iff are calculated using Equation 2 and 3 respectively.

$$tf(t, f) = 1 + log f_{tf} \tag{5.2}$$

$$iff = log(\frac{\#functions}{n_t}) \tag{5.3}$$

Here, f_{tf} represents the frequency of a term t appearing in a function f, $\#functions$ denotes the total count of functions within the search space, n_t signifies the overall number of functions that include the term t. Thus in equation 5.4 each term weight is calculated as follows:

$$\text{weight}_{t \in f} = (tf)_{tf} \times (iff)_t = (1 + log f_{tf}) \times log(\frac{\#functions}{n_t}) \tag{5.4}$$

The VSM score is calculated using equation 5.5.

$$\cos(r, f) = \sum_{t \in r \cap f} (1 + \log f_{tr}) \times (1 + \log f_{tf}) \times iff^2 \times \frac{1}{\sqrt{\sum (1 + \log f_{tr}) \times iff^2}} \times \frac{1}{\sqrt{\sum (1 + \log f_{tf}) \times iff^2}} \tag{5.5}$$

Traditional VSM tends to give preference to smaller functions when ranking them, which can be problematic for large functions because they may receive lower similarity scores. Past research [111, 112] has indicated that larger source code files are more likely to contain vulnerabilities. Therefore, in the context of vulnerability localization, it's crucial to prioritize larger functions in our ranking. To address this issue, we introduce a function denoted as 'x' (as shown in Equation 5.6) within the tVSM model, aiming to account for the function's length.

$$\text{x(terms)} = 1 - e^{-Normalize(\#terms)} \tag{5.6}$$

Equation 5.6 represents a logistic function, specifically an inverse logit function, designed to ensure that larger functions receive higher rankings. We employ Equation 6 to calculate the length value for each source code function based on the number of terms contained within the function. Here we apply the normalized value of '#terms' as the argument for the exponential function e^{-x}. The normalization process is defined in Equation 5.7. Let z denote a set of data, with z_{max} and z_{min} representing the maximum and minimum values of z term, respectively. The normalized value for z term is determined as:

$$\text{Normalize}(z) = \frac{z - z_{min}}{z - z_{max}} \tag{5.7}$$

Considering the above analysis, tVSM score is calculated by multiplying the weight of each function, denoted as x(terms), with the cosine similarity score represented by cos(r, f), as described in Equation 5.8:

$$tVSM(r, f) = \text{x(terms)} \times cos(r, f) \tag{5.8}$$

Once the tVSM score for each function has been computed, a list of vulnerable functions is arranged in descending order of scores. The function with the highest similarity score is positioned at the top of the ranked list.

5.2.2 Statement Level Vulnerability Detection using Vulnerability Reports

This section outlines the statement level vulnerability detection technique as illustrated in Figure 5.2 which is more granular methodology of function level. It involves two primary phases: minimizing the irrelevant search space while maximizing the relevant data domain, and ranking vulnerable statements within that domain. Section 5.2.1's methodology offers superior ranking of vulnerable functions which is serving as the basis for narrowing down the search space to a list of potential vulnerable functions. Static analysis is then applied to these suggested functions to pinpoint the actual vulnerable statements.

For each function, a graph is constructed where each statement is a node, and edges represent the relationship between caller and callee statements, forming what's called a Function Statement Dependency Graph (FSDG). To maximize valid information for each statement, a super node is created for each node by aggregating it with its predecessor nodes, resulting in a Predecessor Node Dependency Graph (PNDG).

To identify potential vulnerable statements, a similarity score is calculated for each statement using the Vector Space Model, comparing each node of the PNDG with the Vulnerability report. This Node or Statement similarity score (S_{score}) is further weighted by the score of the Function (F_{score}) containing the node. These steps are elaborated in the following sections.

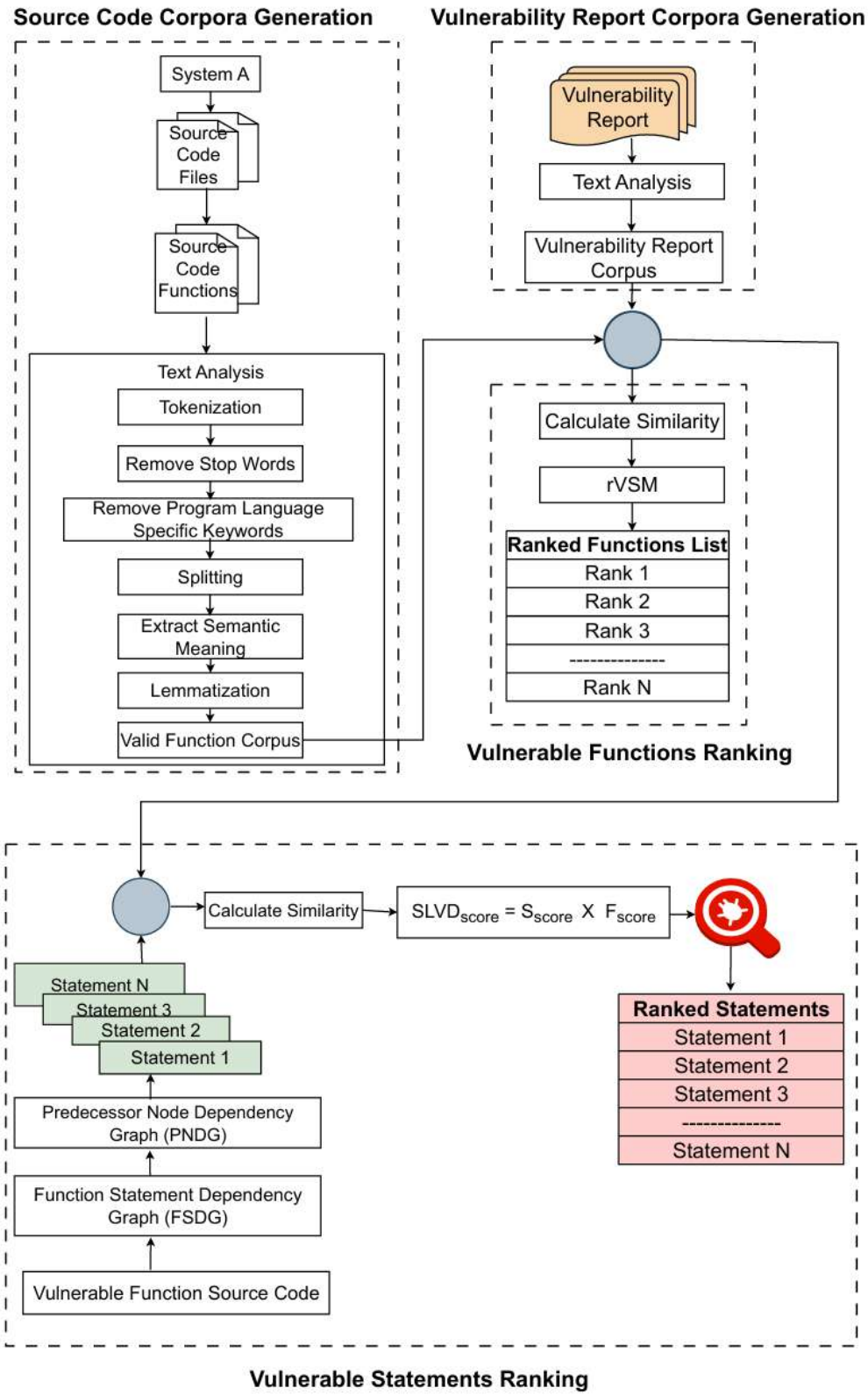


Figure 5.2: Overview of VFSDetector in Statement Level

5.2.2.1 Function Statement Dependency Graph (FSDG) Generation

The primary motivation for Vulnerable Function and Statement Detector (VFS-Detector) is to enhance the validity of each statement in the source code. Statements in source code are interconnected, and vulnerabilities can propagate from one statement to another. Establishing these relationships among statements can therefore increase the valid information of each statement. As detailed in Section 5.2.1, we have a list of vulnerable functions in the source code, which we use here to establish relationships among the statements within each function. To represent these relationships, a Function Statement Dependency Graph (FSDG) is constructed.

FSDG is essential because statements often have interdependencies (e.g., in the statements $p = 5; q = p + r;$, p and q are related, and p is a predecessor of q). Defects commonly propagate from one statement to the next (e.g., a null value in one variable causing a null pointer exception when used by another variable). Typically, vulnerability propagation is forward-directed, meaning a statement executed first can affect subsequent statements that use its result.

Consider a C source code class (listing 5.1), which is part of a banking software application calculating profit. It contains two functions: *determineProfit()* and *computeOperationalExpenses()*. Both function introduce local, global, and function call variables. For instance, determineProfit uses local variables sell and buy, and a global variable netGain. It also relies on the result of the computeOperationalExpenses function, making OperationalExpenses (at line 7) a method call variable. A FSDG for the determineProfit function is constructed as depicted

in Figure 5.3 is a directed graph where $G = (V, E)$. Here, V represents the set of vertices or statements in the function. E represents the directed edges between statements, indicating data flow. If an edge $(v1, v2)$ exists from node $v1$ to $v2$, $v1$ is the predecessor of $v2$. In source code, predecessors are defined by assignment statements, increments, decrements, and conditional statements (e.g., if $(x > 0)$).

Identifying these predecessors maximizes the valid information of each statement, thereby improving vulnerability detection accuracy.

Listing 5.1: Source Code for Generating FSDG

```
#include stdio.h
// Global variable for netGain
double netGain = 0.0;
// Function declarations
double computeOperationalExpenses();
void determineProfit(int sellingPrice, int buyingPrice);
// Function to determine profit
void determineProfit(int sellingPrice, int buyingPrice) {
    int sell = sellingPrice;
    int buy = buyingPrice;
    double operationalExpenses = computeOperationalExpenses();
    netGain += sell - buy - operationalExpenses;
    if (netGain > 0.0) { printf("Success, you made a profit\n"); }
    else { printf("Unfortunately, you incurred a loss\n"); }
}
// Function to compute operational expenses
double computeOperationalExpenses() {
    double electricityBill = 50;
    double waterBill = 10;
    double fuelCost = 15;
    double totalExpenses = electricityBill + waterBill + fuelCost;
    return totalExpenses;
}
```

Three cases are identified for finding the last modification of a variable: Lo-

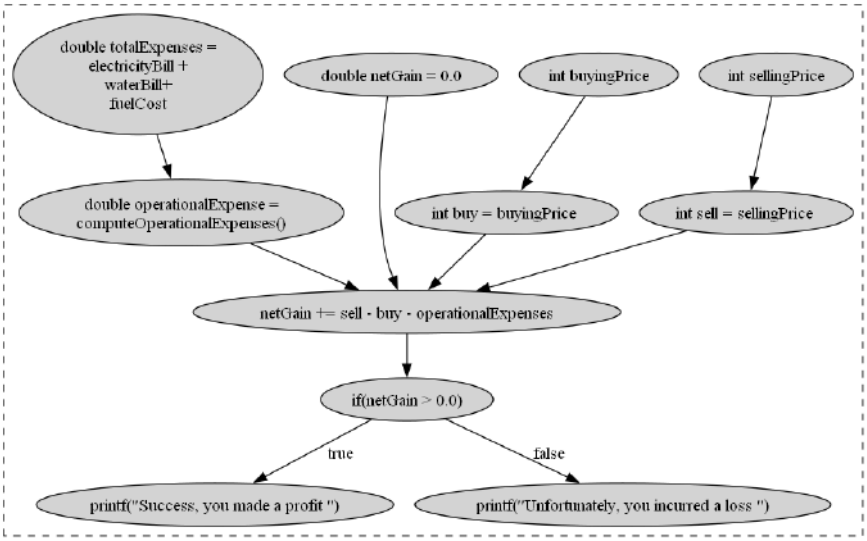


Figure 5.3: Function Statement Dependency Graph of *determineProfit* Function

cal Variable Usage (LVU), Global Variable Usage (GVU), and Function Return Variable Call (FRVC).

1. Local Variable Usage (LVU): The scope of predecessor variables is limited to the method. In determineProfit, int sellingPrice affects sell making int sellingPrice the predecessor of sell. Similarly, int buyingPrice is the predecessor of buy. At line 12, sell, buy, operationalExpenses, and netGain are predecessors of netGain. Since line 13 depends on the calculation in line 12, it is a predecessor of line 13.

2. Global Variable Usage (GVU): The predecessor variable is declared globally and can be used by multiple functions. For example, the global variable double netGain = 0.0 at line 3 affects line 12, where netGain is calculated.

3. Function Return Variable Call (FRVC): This involves a statement calling a function that returns a value. The last modification of the called function's return variable is the predecessor of the current node. For instance, line 7 calls the computeoperationalExpenses function, which returns totalExpenses (last modified at line 20). Hence, line 20 is the predecessor of line 11, as shown in Figure 5.3.

The final two nodes (Success, you made a profit and Unfortunately, you incurred a loss) in Figure 5.3 have no connections with other statements as they do not use any variables they are only conditional output, thus not fitting into the LVU, GVU, or MRVC cases.

5.2.2.2 Node Predecessor Dependency Graph (PNDG) Generation

In this phase, information is maximized by combining each statement with its predecessors. Initially, each node in the FSDG containing a source code statement is processed using several preprocessing techniques like function level. Stop words and programming language-specific keywords are removed as they do not provide vulnerability specific information and can increase ambiguity, thereby affecting the accuracy of vulnerability localization. Examples of stop words include “am” ,“is,”, “are” and programming keywords such as “int”, “HashMap”, “array”, and “public”. Multi-word identifications are also performed using separators like Camel case, underscores, and dots, which help increase valid information about a bug. FFor instance, the word ”userLoginDetails” can be split into “user”, “login”, and “details” using Camel case identification. Remaining words are transformed to lowercase for uniformity, and their semantic meanings are identified using WordNet to enhance the information domain. Only single-word synsets from WordNet are considered. Words are processed to identify the basic concept of their multiple forms, crucial for bug localization, as it relies on textual similarity between related words. For example, “connected” and “connection” share the same root “connect”. Stemming, specifically Porter stemming [50] is applied to reduce words to their root forms. These preprocessing steps result in a list of valid code corpus in each node.

Since statement-level vulnerability detection depends on the similarity between the vulnerability report and source code statements, increasing the valid information of a statement is essential. A super node is created by combining each node

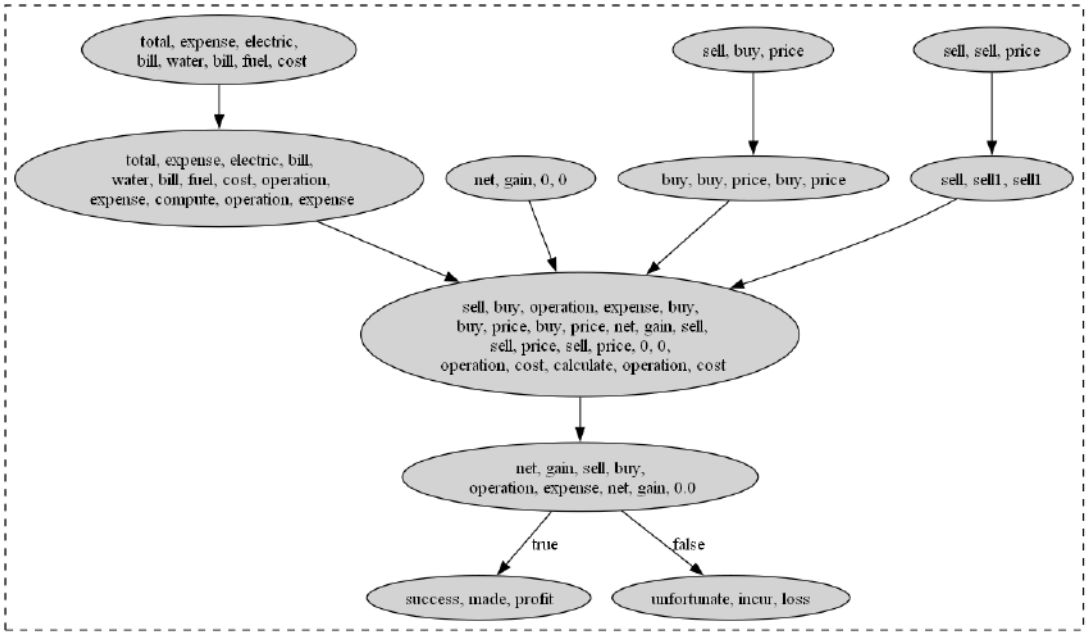


Figure 5.4: Predecessor Node Dependency Graph of FSDG in Figure 5.3

with its predecessor nodes' corpora, as predecessors can affect successors, resulting in an PNDG (illustrated in Figure 5.4). In this graph, each node from Figure 5.3 is combined with its predecessor nodes. For example, the node "int sell = sellingPrice" is combined with its predecessor "int sellingPrice." Similarly, the statement "netGain += sell - buy - operationalExpenses" has four predecessor nodes (netGain, sell, buy, and operationalExpenses), and the PNDG includes the corpora of these nodes.

Following the construction of the PNDG, the process computes the similarity between the vulnerability report and each node within the PNDG. This involves traversing the PNDG to identify shared terms between the vulnerability report and the corpora of each node. Through this traversal, the frequencies of shared terms are computed to determine the term frequency (tf), a crucial component in the Vector Space Model (VSM). Among the various tf formulations proposed for enhancing VSM performance, the logarithm variant is found to be the most effective than others [113],[114], [115]. Thus, term frequency of source code node which represents statements$tf(t, n)$ and term frequency of vulnerability report

$tf(t, v)$ are computed as per Equations 5.9 and 5.10:

$$tf(t, n) = \log(f_{tn} + 1) \tag{5.9}$$

$$tf(t, v) = \log(f_{tv} + 1) \tag{5.10}$$

Here, f_{tn} and f_{tv} represent the frequencies of a term in the PNDG node and the vulnerability report, respectively. These values are used to calculate how similar the vulnerability report is to each statement in the source code., a computation performed through VSM as depicted in Equation 5.11:

$$Ss(n, v) = VSM(n, v) = \cos(n, v) = \frac{\sum_{t \in (v \cap n)} (\log(f_{tn} + 1) \cdot \log(f_{tv} + 1))}{\sqrt{\sum_{t \in n} (\log(f_{tn} + 1))^2} \cdot \sqrt{\sum_{t \in v} (\log(f_{tv} + 1))^2}} \tag{5.11}$$

Here, $Ss(n, v)$ denotes the similarity score of each node for a given vulnerability, where f_{tv} and f_{tn} represent the term frequencies in the vulnerability report and each node, respectively. A higher score (Ss) suggests a greater similarity between the node and the vulnerability report, while a lower score implies lesser relevance. Subsequently, each statement's score is adjusted by it's corresponding function score, because statements typically contain fewer words compared to functions. This is illustrated in Equation 5.12:

$$SLVD_{score} = S_{score} \times F_{score} \tag{5.12}$$

Here, S_s and V_s denote the function score containing statement s and the statement similarity score respectively. $SLVD_{score}$ represents the Statement Level Vulnerability Detection ranking score of each statement weighted by the function score F_s. Finally, the statements are ranked in descending order based on this $SLVD_{score}$.

5.3 Experiment and Result Analysis

This section provides information on the practical implementation, dataset, the criteria used for evaluation and experimental result analysis of this study.

5.3.1 Dataset

We used the benchmark dataset[2] Big-Vul[82]. This dataset comprises reliable and comprehensive code vulnerabilities which are directly linked to the publicly accessible CVE database. Notably, the creation of this dataset involved a significant investment of manual resources to ensure its high quality. Furthermore, this dataset is noteworthy for its substantial scale, being one of the most extensive vulnerability datasets available. It is derived from a collection of 348 open-source Github projects, encompassing a time span from 2002 to 2019, and covers 91 distinct Common Weakness Enumeration (CWE) categories. This comprehensive dataset comprises approximately 188,800 C/C++ functions, with 5.9% of them identified as vulnerable (equivalent to 10,700 vulnerable functions). This dataset provides granular ground-truth information at the function level, specifying which functions within a codebase are susceptible to vulnerabilities.

5.3.2 Implementation

The proposed method is implemented in Python (version 3.11.5). The experiment was conducted on an Windows server equipped with an Intel(R) Core(TM) i5-10300H CPU processor @3.0GHz and having 64GB of RAM. The implementation involves various python libraries and NLTK (Natural Language Toolkit) libraries for text processing and feature extraction. It takes function files as input and provides ranking of suspicious vulnerable functions as output.

[2] https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset

5.3.3 Evaluation Metrics

To evaluate the performance of VFSDetector we used six metrics: Top N Rank, Mean Reciprocal Rank (MRR), and Mean Average Precision (MAP), Top-k Accuracy, Top-k Precision, and Top-k Recall. These metrics are commonly used to assess the effectiveness of retrieval and ranking systems. Generally, higher values indicate better performance. The details are as follows:

1. **Top N Rank**: This measures the number of vulnerabilities detected within the top N ranks ($N = 1, 5, 10, 20$ for this system). For example, if $N = 10$, it means the vulnerable statements are found within the top 10 suggestions. If one of the fixed files of a vulnerability is in the result set, it is considered detected.

2. **MRR**: Mean Reciprocal Rank is the inverse of the rank of the first correct result of a query. For instance, if a bug is found in the 5th position, the reciprocal rank is $\frac{1}{5}$. MRR ranges from 0 to 1, with higher values indicating better accuracy. It is calculated using the formula 5.13:

$$\text{MRR} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{rank_i} \tag{5.13}$$

Here, N is the number of queries and $rank_i$ is the rank of the i-th query.

3. **MAP**: Mean Average Precision measures how well the system locates all vulnerable files, not just the first one. MAP is the mean of the average precision values for a set of queries. The precision for the top N files is calculated by equation 5.14 and 5.15:

$$P(N) = \frac{\text{Vul(n)}}{N} \tag{5.14}$$

The average precision for a query is calculated by:

$$\text{AverageP} = \sum_{N=1}^{K} \frac{P(N) \times V(N)}{\text{Vul(f)}} \tag{5.15}$$

Here, N is the rank, Vul(f) is the number of vulnerable files, Vul(n) is the number

of vulnerable files detected in Top N, K is the total number of ranked files, $V(N)$ indicates if the N-th file contains a vulnerability, and $P(N)$ is the precision for the top N files. MAP is the mean of these average precision values. Higher MAP values indicate better vulnerability detection performance.

4. **Top-k Accuracy (A@K)**: The percentage of vulnerable functions where at least one vulnerable statement appears in the Top-K ranking of the corresponding functions. Let N be the total number of vulnerable functions and X be the number of functions where at least one vulnerable statement is ranked in the Top-K as shown in equation 5.16.

$$A@K = \frac{x}{n} \times 100\% \tag{5.16}$$

5. **Top-k Precision (P@K)**: The average proportion of actual vulnerable statements in the Top-K statements of the functions ranked by the model.

Let $\{\text{ActualVuls}_i\}$ and $\{\text{RankedVuls}_i\}@K$ be the set of vulnerable statements and the list of statements ranked in the Top-K of function i, respectively. Then, the P@K is calculated as follows in equation 5.17:

$$P@K = \frac{1}{n}\sum_{i=1}^{n}\frac{|\{\text{ActualVuls}_i\} \cap \{\text{RankedVuls}_i\}@K|}{|\{\text{RankedVuls}_i\}@K|} \tag{5.17}$$

6. **Top-k Recall (R@K)**: The average proportion of correctly predicted vulnerable statements in the Top-K statements among the total number of vulnerable statements in the functions.The R@K value is calculated using equation 5.18 :

$$R@K = \frac{1}{n}\sum_{i=1}^{n}\frac{|\{\text{ActualVuls}_i\} \cap \{\text{RankedVuls}_i\}@K|}{|\{\text{ActualVuls}_i\}|} \tag{5.18}$$

In practice, the vulnerability detection tool is effective only if it ranks the vulnerable statements of the function at the top of the list. Otherwise, developers might ignore the recommendation [69], [116]. Therefore, we consider relatively small values of K (i.e., $K \in \{1, 5, 10, 20\}$) for these metrics. Additionally, the

problem of identifying vulnerable statements can be treated as a binary classification problem. If the predicted score of a statement is greater than a threshold θ, the statement is predicted as vulnerable (label 1); otherwise, it is non-vulnerable (label 0). Thus, we use Accuracy, Precision, Recall, and F1-score, which are common metrics for evaluating classification models, to measure the performance of the approaches.

5.3.4 Result Analysis in Function Level

To conduct this research we used the extensive Big-Vul dataset which contains large scale vulnerability reports of C/C++ code from open source GitHub projects. Other C/C++ datasets can also be used. Based on the highest number of vulnerabilities reported, as an initial study, in our paper VFDetector [16] was evaluated using ten vulnerability reports from these six open-source projects which are chosen randomly from the dataset. Table 5.1 lists the analysed project name, CVE ID of report, and corresponding github source code link of function level vulnerability detection. We choosed top six well known projects from this dataset which are Chrome, Linux, Radare2, ImageMagick, Tcpdump and FFmpeg as shown in Table 5.1. As the selected projects are open-source in nature and are hosted on GitHub, serving as the primary platform for storing code and managing version control. It allows us to extract all essential commits for our analysis. Additional information about the repositories can be found in Table 5.2.

To measure the effectiveness of the proposed vulnerability detection method, we use the Top N Rank metric. This metric signifies the count of vulnerable functions ranked in the top N (where N can be 1, 5, or 7) in the obtained results. When assessing a reported vulnerability, if the top N query results include at least one function that corresponds to the location where the vulnerability needs to be addressed, we determine that the vulnerable function is detected successfully. Table 5.2 includes ten vulnerability reports from six open source projects with

their total number of commits, files, functions, actual vulnerable functions name and finally VFDetector ranking in Top N ranked functions in output. The results of Table 5.2 shows that among the ten CVE reports VFSDetector ranks the actual vulnerable function at the 1st position for four (40%) reports which are CVE ID #13000, #14470, #15033, #16359. For five reports (50%) with CVE ID #3916, #2094, #2038, #10190 and #11339 it ranks the vulnerable function in Top 5 rank. It indicates that nine (90%) reports are ranked in Top 5. For one report CVE-2013-6763 of Linux Kernel version 3.12.1 it ranks the vulnerable function in Top 7 rank i.e., in 7^{th} position out of total 273,898 functions. Upon manual inspection, we observed that the six functions preceding the vulnerable function exhibit a higher similarity score compared to the actual vulnerable function. The reason behind this can be the inter-connectedness of these six functions with the vulnerable function through function calls. It is also noticeable that projects with less number of functions ranks the vulnerable function in 1^{st} position and with large number of functions the ranking decreases slightly. The reason behind this is larger projects might contain more associated functions which are needed to be fixed in order to address a particular vulnerability.

In summary, the experimental results show that VFDetector can detect vulnerable functions from a huge volume of functions and can also suggest developers with the related functions having highest similarity scores which might need to be patched to address the reported vulnerability. Moreover, to the best of our

Table 5.1: List of Initially Analyzed Open Source Projects in Function Level

#	Projects	CVE ID	Source Code
1	Chrome	CVE-2011-3916	github.com/chromium/chromium/tree/f1a59e0513d63758588298e98500cac82ddccb67
2	Radare2	CVE-2017-16359	github.com/radareorg/radare2/tree/1f5050868eedabcbf2eda510a05c93577e1c2cd5
3	Linux	CVE-2013-6763	github.com/torvalds/linux/tree/f9ec2e6f7991e748e75e324ed05ca2a7ec360ebb
4	Linux	CVE-2013-2094	github.com/torvalds/linux/tree/41ef2d5678d83af030125550329b6ae8b74618fa
5	Linux	CVE-2014-2038	github.com/torvalds/linux/tree/a9ab5e840669b19aca2974e2c771a77df2876434
6	ImageMagick	CVE-2017-15033	github.com/ImageMagick/ImageMagick/tree/c29d15c70d0eda9d7ffe26a0ccc181f4f0a07ca5
7	Tcpdump	CVE-2017-13000	github.com/tcpdump-group/tcpdump/tree/a7e5f58f402e6919ec444a57946bade7dfd6b184
8	Tcpdump	CVE-2018-14470	github.com/tcpdump-group/tcpdump/tree/aa3e54f594385ce7e1e319b0c84999e51192578b
9	FFmpeg	CVE-2016-10190	github.com/FFmpeg/FFmpeg/tree/51020adcecf4004c1586a708d96acc6cbddd050a
10	FFmpeg	CVE-2019-11339	github.com/FFmpeg/FFmpeg/tree/3f086a2f665f9906e0f6197cddbfacc2f4b093a1

Table 5.2: Summary of Tested Projects in Function Level

CVE ID	#Commits	#Files	#Functions	Vulnerable Functions	VFDetector Ranking
CVE-2017-13000	4,466	180	638	`extract_header_length()`	1
CVE-2018-14470	4,548	185	640	`babel_print_V2()`	1
CVE-2017-15033	12,558	586	4,694	`ReadYUVImage()`	1
CVE-2017-16359	16,362	965	9,197	`store_versioninfo_gnu_verdef()`	1
CVE-2011-3916	93,104	4,929	15,042	`WebGLObject()`	2
CVE-2019-11339	93,322	2,572	16,724	`mpeg4_decode_studio_block()`	2
CVE-2016-10190	82,768	2,286	14,713	`http_buf_read()`	3
CVE-2014-2038	413,259	15,674	229,682	`nfs_can_extend_write()`	4
CVE-2013-2094	362,534	18,358	257,550	`perf_swevent_init()`	5
CVE-2013-6763	401,141	19,260	273,898	`uio_mmap_physical()`	7

knowledge we are the first to incorporate vulnerability reports in software vulnerability detection from the concept that vulnerability report's description contain conceptual information about a reported vulnerability. Based on the promising results in this initial evaluation, further analysis is made in statement level in the next subsection with more vulnerability reports from diverse projects to make the approach comparable and generalizable with other state-of-the-art statement level vulnerability detection approaches.

5.3.5 Result Analysis in Statement Level

To conduct the statement level analysis we used the same BIg-Vul dataset.This statement level vulnerability detection VFSDetector is mainly an extension of our previous paper in function level [16]. Apart from identifying vulnerable functions in [16], this part pinpoints vulnerable statements in source code where patches is to be done. Table 5.3 summarizes ten vulnerability reports detailing their files, functions,statements, Vulnerable Function (VF) Rank and Vulnerable Statement (VS) Rank in the TopN ranked functions. Notably, for 20% of reports, the VFSDetector ranks the actual vulnerable statement 1st, while 70% rank within the Top 5 and 100% reports rank within top 15. However, for two reports CVE #2094 and CVE #6763 the genuine vulnerable statement ranks 9th and 13th out of total 6,783,409 and 8,623,609 statements in the large system. This suggests that larger

projects may have lower rankings due to more associated statements which need fixes.

Table 5.3: Initially Tested Software Systems Tested for Statement Level Detection

Vulnerability Report	#Files	#Functions	#Statements	VF Rank	VS Rank
CVE-2017-13000	180	638	12,138	1	1
CVE-2018-14470	185	640	11,990	1	1
CVE-2017-15033	586	4,694	84,742	1	3
CVE-2017-16359	965	9,197	91,116	1	2
CVE-2011-3916	4,929	15,042	136,904	2	4
CVE-2019-11339	2,572	16,724	284,324	2	5
CVE-2016-10190	2,286	14,713	246,938	3	3
CVE-2014-2038	15,674	229,682	4,345,678	4	7
CVE-2013-2094	18,358	257,550	6,783,409	5	9
CVE-2013-6763	19,260	273,898	8,623,609	7	13

Experimental results indicate that VFSDetector consistently ranks vulnerable functions 1st in 40% of cases and accurately identifies vulnerable statements in 20% of cases. Moreover, it places the vulnerable function within the top five ranks in 90% of cases and the vulnerable statement in 70% of cases. Additionally, all analyzed reports have vulnerable statements ranked in the top 15. This suggests that VFSDetector shows promising results in identifying vulnerable functions and statements within extensive software systems.

After initial evaluation, the performance of the VFSDetector was evaluated vastly on 1200 vulnerability reports across 14 different projects as illustrated in Table 5.4, revealing notable strengths in its detection capabilities. The graph in Figure 5.5 illustrates the visual comparison of detection performance across various software projects. The performance is measured at different ranking levels: Top 1, Top 5, Top 10, and Top 20. The x-axis represents the different projects, while the y-axis shows the percentage of vulnerabilities correctly detected at each ranking level. The projects included in the test are Openssl, Wireshark, FFmpeg, Kuberos, Radare2, Curl, Php, Libarchive, LibRaw, Tcpdump, ImageMagick, Chrome, Android, and Linux. The results highlight the variation in detection accuracy across projects of different sizes and complexity. This section also provides

an interpretation of the results in large code bases with Top N Rank also including the Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) metrics, which are crucial for understanding the effectiveness of the VFSDetector.

Table 5.4: The performance of VFSDetector Tested on 1200 Vulnerabilities

Projects	**#Vul**	**Top 1**	**Top 5**	**Top 10**	**Top 20**	**MRR**	**MAP**
Openssl	70	22 (31.43%)	31 (44.29%)	43 (61.43%)	54 (77.14%)	37.7%	32.3%
Wireshark	70	23 (32.86%)	32 (45.71%)	51 (72.86%)	66 (94.29%)	43.18%	38.57%
FFmpeg	50	20 (40%)	35 (70%)	40 (80%)	45 (90%)	32.75%	30.23%
Kuberos	50	18 (36%)	25 (50%)	28 (56%)	32 (64%)	43.2%	38.6%
Radare2	50	15 (30%)	20 (40%)	23 (46%)	28 (56%)	40.2%	35.3%
Curl	70	18 (25.71%)	25 (35.71%)	30 (42.86%)	35 (50%)	35%	30.8%
Php	70	20 (28.57%)	30 (42.86%)	35 (50%)	40 (57.14%)	45.2%	40.1%
Libarchive	40	15 (37%)	25 (62.5%)	30 (75%)	35 (87.5%)	37.5%	32.8%
LibRaw	70	29 (41.43%)	35 (50%)	48 (68.57%)	55 (78.57%)	40.1%	35.3%
Tcpdump	100	40 (40%)	70 (70%)	87 (87%)	100 (100%)	66%	61%
ImageMagick	100	40 (40%)	62 (62%)	78 (78%)	88 (88%)	69.3%	51.3%
Chrome	200	38 (19%)	62 (31%)	98 (49%)	122 (61%)	40%	35%
Android	200	43 (21.5%)	56 (28%)	87 (43.5%)	100 (50%)	45.38%	41.88%
Linux	200	46 (23%)	59 (29.5%)	74 (37%)	90 (45%)	28%	26%

Exceptional Detection Performance: Projects like Tcpdump and ImageMagick demonstrate the VFSDetector's exceptional detection capabilities. Tcpdump accurately identifies 40% of vulnerabilities at the top 1 position and achieves a perfect 100% within the top 20. ImageMagick similarly shows strong results with 40% of vulnerabilities at the top 1 position and 88% within the top 20. These results underscore the VFSDetector's high accuracy and effectiveness in pinpointing vulnerabilities quickly and reliably.

Robust Performance Across Various Projects: In medium-sized projects such as Openssl, Wireshark, FFmpeg, LibRaw, Libarchive and Kuberos, the VFSDetector continues to exhibit robust performance. For example, Openssl has 31.43% of vulnerabilities at the top 1 and 77.14% within the top 20, while Wireshark achieves 32.86% at the top 1 and an impressive 94.29% within the top 20. LibRaw ranks 41.43% in Top 1 and 78.75% in Top 20. Libarchive achieves 37%

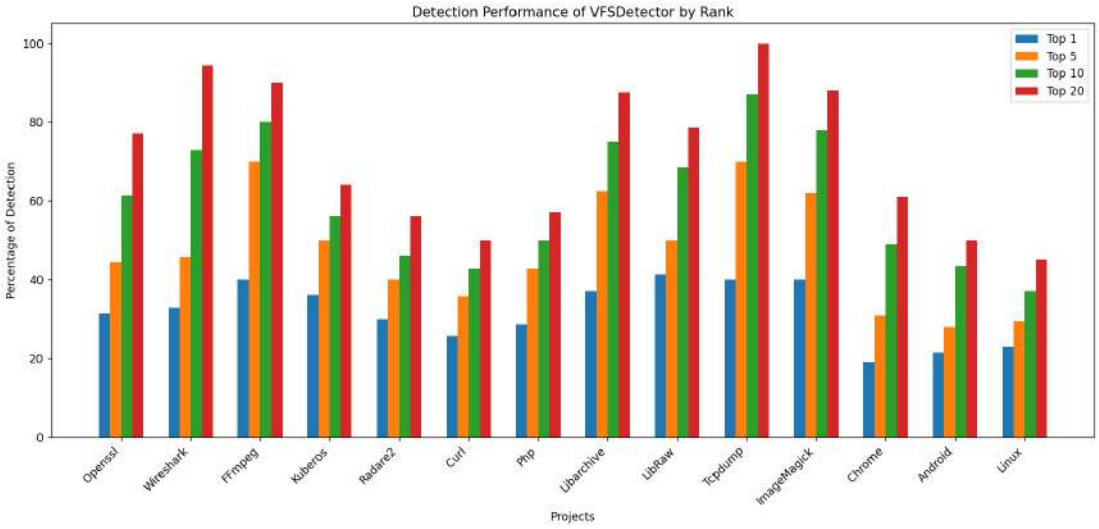


Figure 5.5: Detection Performance of VFSDetector Across Different Projects

at the top 1 and 87.5% within the top 20. FFmpeg shows 40% at the top 1 and 90% within the top 20, and Kuberos has 36% at the top 1 and 64% within the top 20. These figures highlight the VFSDetector's consistent accuracy and reliability across different project sizes and complexities.

Consistent High Accuracy: Other projects such as Php, Curl, Libarchive, and Radare2 also demonstrate the VFSDetector's consistent high accuracy. Php identifies 28.57% of vulnerabilities at the top 1 and 57.14% within the top 20, while Curl shows 25.71% at the top 1 and 50% within the top 20. Radare2 reaches 30% at the top 1 and 56% within the top 20. These results further illustrate the tool's precision and dependability in detecting vulnerabilities effectively.

Wide-Ranging Effectiveness: The VFSDetector's effectiveness extends to larger projects such as Android, Chrome, and Linux, showcasing its ability to handle increased complexity while still delivering valuable detection insights. For example: This is evident from specific cases such as for vulnerability report CVE-ID-2094 and CVE-ID-6763, where the genuine vulnerable statements rank 9th and 13th among millions of statements, illustrating the difficulty in large-scale systems. Android achieves 21.5% at the top 1 and 50% within the top 20, Chrome shows 19% at the top 1 and 61% within the top 20, and Linux, although with lower

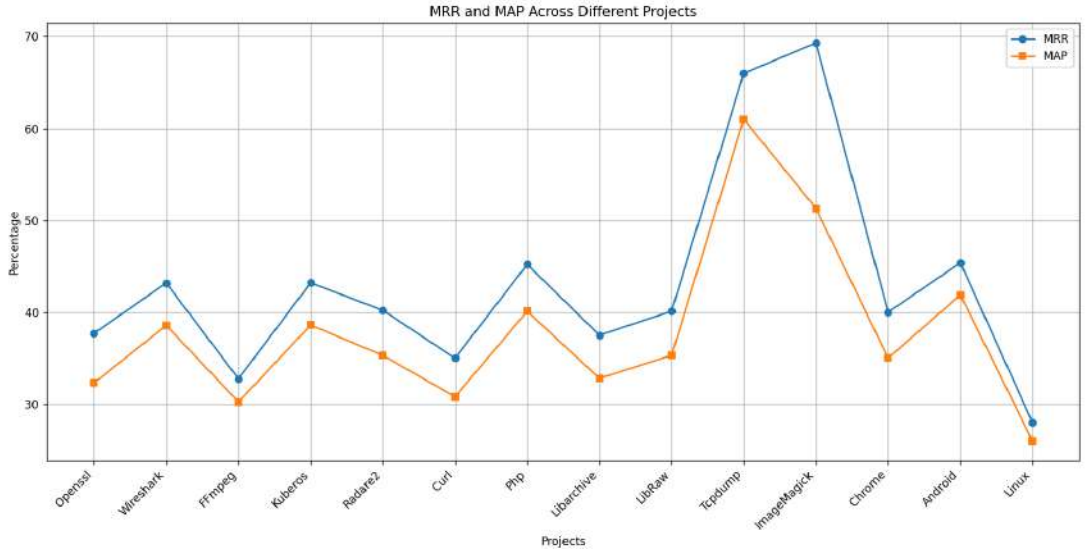


Figure 5.6: Comparison of Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) Across Various Software Projects

rankings, still highlights the tool's capability to manage extensive codebases.

The Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) metrics further substantiate the VFSDetector's effectiveness in localizing vulnerabilities in Figure 5.6. The MRR is a measure of how quickly the correct results are found, with higher values indicating that the actual vulnerable statements are ranked higher on average. For instance, ImageMagick has an MRR of 69.3%, and Tcpdump has an MRR of 66%, reflecting the high ranking of vulnerabilities in these projects. Wireshark and Php also exhibit strong MRR values of 43.18% and 45.2%, respectively, indicating that the correct vulnerable statements are frequently found early in the ranked list. The MAP measures the precision of the results, averaged over all vulnerabilities. Higher MAP scores indicate better overall accuracy in identifying vulnerable statements. Tcpdump and ImageMagick perform well with MAP scores of 61% and 51.3%, respectively. Wireshark and Php also show robust MAP values of 38.57% and 40.1%, respectively, underscoring the VFSDetector's precision in these projects.

In summary, the large MRR and MAP scores across various projects highlight the VFSDetector's effectiveness in accurately localizing vulnerable statements.

The tool's high detection rates and precision underscore its robustness and reliability, making it a valuable asset for vulnerability detection across a diverse range of projects. These results affirm the VFSDetector's capability in effectively pinpointing vulnerabilities at the statement level.

5.3.6 Baseline Methods Comparison

Table 5.5: Vulnerability Detection Performance Comparison with State-of-the-art Approaches

Metric	Approach	Top-1	Top-5	Top-10	Top-15	Top-20
Top-k Accuracy	LineVD	16.67%	31.17%	40.83%	47.81%	54.93%
	IVDetect	27.81%	44.46%	51.44%	56.94%	58.68%
	LineVul	27.28%	44.32%	52.11%	55.87%	58.28%
	COSTA	31.15%	36.84%	48.87%	53.42%	56.70%
	VFSDetector	**39.53%**	**48.74%**	**59.27%**	**65.66%**	**70.43%**
Top-k Precision	LineVD	16.67%	16.94%	17.87%	18.53%	18.99%
	IVDetect	27.81%	22.71%	21.62%	21.14%	20.14%
	LineVul	27.27%	22.22%	21.06%	20.55%	19.86%
	COSTA	38.15%	27.19%	23.40%	21.70%	20.58%
	VFSDetector	36.54%	**36.82%**	**31.21%**	**33.23%**	**42.37%**
Top-k Recall	LineVD	38.62%	54.81%	66.20%	74.85%	83.83%
	IVDetect	46.69%	65.80%	76.18%	83.90%	88.37%
	LineVul	46.62%	65.22%	76.37%	82.82%	87.81%
	COSTA	52.10%	70.02%	81.25%	86.01%	90.84%
	VFSDetector	**58.13%**	67.22%	**82.34%**	84.97%	**95.81%**
F1 Score	LineVD	23.29%	25.88%	28.14%	29.71%	30.97%
	IVDetect	34.86%	33.77%	33.68%	33.77%	32.80%
	LineVul	34.41%	33.15%	33.02%	32.93%	32.39%
	COSTA	44.05%	39.17%	36.34%	34.66%	33.56%
	VFSDetector	**44.87%**	**47.58%**	**45.26%**	**47.78%**	**58.76%**

The table 5.5 presents a comparative analysis of vulnerability detection performance between the VFSDetector and four baseline methods: LineVD, IVDetect, LineVul, and COSTA. The analysis of the results of VFSDetector outperforming other baseline methods are discussed below and visualized in Figure 5.7.

Accuracy: VFSDetector demonstrates superior accuracy compared to the base-

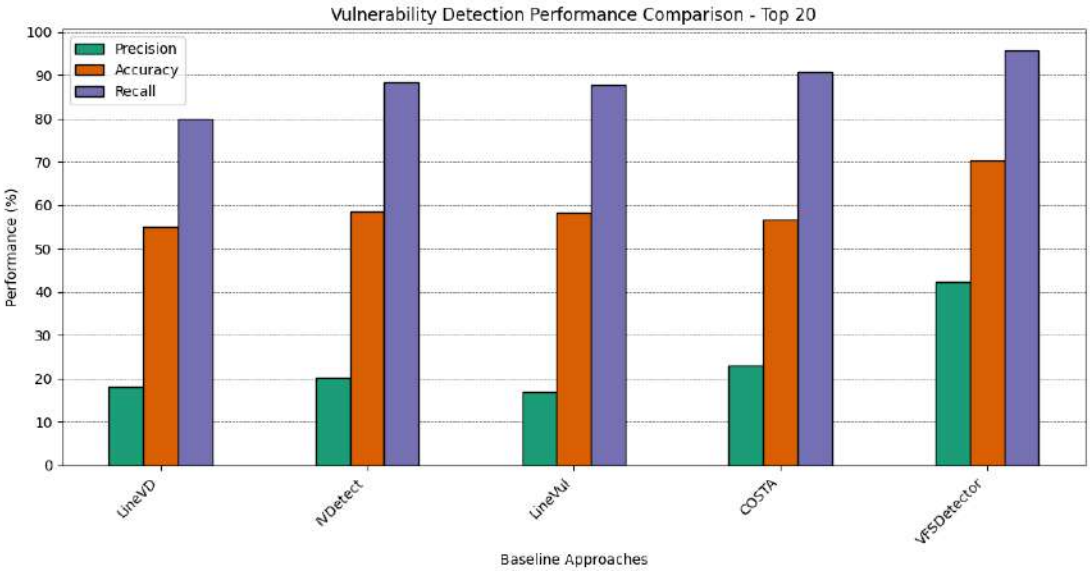


Figure 5.7: Performance Comparison of Top 20 Vulnerability Detection Metrics for Different Baseline Approaches

line methods across all top-k values. For instance, at Top-1 accuracy, VFSDetector achieves 39.53%, outperforming LineVD by 22.86%, IVDetect by 11.72%, LineVul by 12.25%, and COSTA by 8.38%. This trend continues across all top-k values, with VFSDetector consistently surpassing the baselines by significant margins. At Top-20 accuracy, VFSDetector achieves 70.43%, which is notably higher compared to the other methods. Precision: In terms of precision, VFSDetector maintains its superiority over the baseline methods. At Top-5 precision, VFSDetector achieves 36.82%, outperforming LineVD by 19.88%, IVDetect by 14.11%, LineVul by 14.76%, and COSTA by 8.24%. This pattern persists across higher top-k values, with VFSDetector consistently exhibiting higher precision percentages compared to the baselines. At Top-20 precision, VFSDetector achieves 42.37%, highlighting its effectiveness in minimizing false positives.

Recall: VFSDetector also excels in recall compared to the baseline methods. For example, at Top-1 recall, VFSDetector achieves 58.13%, surpassing LineVD by 19.51%, IVDetect by 11.44%, LineVul by 11.51%, and COSTA by 6.03%. This trend continues across in top-10 values, with VFSDetector consistently achieving higher recall percentages compared to the baselines. At Top-20 recall, VFSDe-

tector achieves 95.81%, indicating its capability to detect a larger proportion of vulnerabilities.

Overall, the VFSDetector demonstrates significant advancements in vulnerability detection, consistently outperforming baseline methods across all evaluation metrics. Its achievements include higher accuracy, precision, and recall rates, underscoring its effectiveness in accurately identifying vulnerable statements and contributing substantially to advancing the state-of-the-art in vulnerability detection techniques.

5.4 Threats to Validity

In this section, we discussed the potential threats which may affect the validity of this study.

Threats to external validity: The generalizability of the acquired results poses a threat to external validity. The dataset that we used in our research was gathered from open-source. Open-source projects may contain data that differs from those created by software companies with sound management practices. Seven Apache projects are examined in this study. More projects from other systems are needed to be evaluated for the generalisation. However, to overcome this threat large-scale diversified projects with long change history is to be chosen.

Threats to internal validity: One limitation of our approach is its reliance on sound programming practices when naming variables, methods, and classes. If a developer uses non-meaningful names, it could have an adverse impact on the effectiveness of vulnerability detection.ay not fully represent the characteristics of the whole program. Additionally, our model is evaluated with C/C++ functions and it may encounter challenges in detecting vulnerabilities in other programming languages.

Threats to construct validity: We used the WordNet database and lem-

matizer of NLTK library as essential components in text pre-processing to extract word semantics and reduce words to their base forms. Since these resources are well known for their usefulness in NLP, we relied on their accuracy. Moreover, vulnerability reports offer essential information that developers rely on to address and patch vulnerable functions. A bad vulnerability report delays the fixing process. It's worth noting that our approach is dependent on the quality of these reports. If a vulnerability report lacks sufficient information or contains misleading details, it might show that VFSDetector doesn't perform as well.

5.5 Summary

This chapter introduces a technique for detecting vulnerabilities at the statement level by establishing connections among statements. Initially, vulnerable functions are identified which serve as a relevant and minimized search space for ranking vulnerable statements. Subsequently, relationships among statements are defined for each function within this search space by proposing the Function Statement Dependency Graph (FSDG). As nodes in FSDG contain limited information, efforts are made to maximize information by incorporating predecessor nodes, resulting in the creation of the Predecessors Node Dependency Graph (PNDG). The similarity between each node in PNDG and vulnerability reports is then matched and similarity score is calculated. Finally, each statement is weighted based on the function similarity score generated earlier in function level leading to a more precise ranking of vulnerable statements. The next chapter concludes the entire thesis and discusses potential directions for future research.

Chapter 6

Conclusion

Automated statement level vulnerability detection has recently drawn the attention of researchers due to its potentiality of minimizing the debugging effort [117], [118], [119] and fixing vulnerabilities. Existing automated vulnerability detection approaches [69], [85], [120], [63] often overlook or insufficiently address vulnerabilities at the statement level, despite the prevalence of these types of vulnerabilities [86]. This occurs because detection at the granularity of statement level significantly increases the search space and the likelihood of generating incorrect classification of actual vulnerabilities [121]. Furthermore, these techniques use either text or graph based approach in function level analysis to guide the detection process. However, none of these approaches explore whether vulnerability detection can be further enhanced by integrating the advantages of both approaches. In this context, the current thesis examines the impact of combining text and graph based methods on detecting vulnerabilities from source code in function level. Next, it introduces an automated vulnerability detection approach that detects vulnerable functions and statements by comparing textual content from vulnerability report corpora with source code corpora at the statement level because existing studies have overlooked the potential of vulnerability reports to provide contextual insights for detecting software vulnerabilities. This chapter provides a summary of

the thesis regarding its contributions and achievements by answering the research questions. Finally, it concludes with potential directions for future research.

6.1 Impact of Combining Text and Graph Based Approach in Function Level Vulnerability Detection

The sub research question 1 of Section 1.2 is addressed in this section:
SQ1: What is the impact of combining text-based and graph-based methods on function-level vulnerability detection?
At first, this study introduces an innovative approach for automated software vulnerability detection by integrating text-based and graph-based analyses for improved performance in identifying vulnerabilities within source code. It leveraged the pre-trained CodeBERT [94] embedding model and Convolutional Neural Network (CNN) [81] for function-level code analysis, then integrates multiple centrality measures (degree, Katz, closeness, betweenness, eigenvector) to enhance the image representation of code, and achieved significant performance improvements over existing state-of-the-art methods. The proposed method employs a multi-step process. Functions are first embedded using CodeBERT, a transformer-based model optimized for code representation which is basically the textual analysis of source code. Program Dependency Graphs (PDGs) are then constructed to capture the structural dependencies between lines of code. Five centrality measures are computed to assess the importance of nodes within the PDG. These centrality measures are used to create multi-channel images representing the code, with each channel corresponding to a different centrality measure. This is the graphical analysis of source code. And finally combining text and graph analysis, Convolutional Neural Networks (CNNs) are trained on these images to classify code functions as

vulnerable or non-vulnerable.

Experiments were conducted using two benchmark datasets, SARD and Big-Vul, to evaluate the effectiveness of VulBertCNN. The model's performance was compared to three existing methods: TokenCNN, VulCNN, and ASVD. VulBertCNN achieved an accuracy of 95.7% and an F1 score of 89% on the SARD dataset, with precision and recall scores of 90.4% and 87.3%, respectively. On the Big-Vul dataset, VulBertCNN achieved an accuracy of 91.8% and an F1 score of 88.2%. These results demonstrate substantial improvements over the previous methods: VulBertCNN outperformed ASVD, which achieved 88% accuracy, and VulCNN, which achieved 83% accuracy, by leveraging CodeBERT embeddings instead of Sent2Vec embeddings. The accuracy improved from 78.6% to 95.7% on the SARD dataset and from 61.4% to 91.8% on the Big-Vul dataset. The F1 score increased from 62.6% to 89% on the SARD dataset.

The evaluation reveals that the integration of CodeBERT with CNNs and centrality analysis significantly enhances vulnerability detection accuracy. Although the F1 score on the Big-Vul dataset did not outperform ASVD, the accuracy improvement demonstrates the robustness of the proposed method. The findings suggest that combining code embeddings with contextual information from vulnerability reports can further enhance the detection of statement-level vulnerabilities, underscoring the importance of comprehensive contextual analysis for accurate vulnerability identification which highlights the potential of this approach for enhancing software security through precise and automated vulnerability detection.

6.2 Impact of using Vulnerability Reports in Function and Statement Level Vulnerability Detection

The sub research question 2 and 3 is addressed in this section:

SQ2: What is the impact of combining vulnerability reports with source code in minimizing the solution search space?

Based on the result of the function level vulnerability identification with both text and graph based approach, to further narrow down the detection in statement level, this study proposes a technique which minimizes the search space by discarding irrelevant portions and focusing on probable vulnerable functions through static analysis of the source code. The method involves processing the text of both source code and vulnerability reports using various text processing techniques, including stop words removal, programming language-specific keywords removal, multi-word identification, semantic meaning extraction, and stemming. These techniques generate two corpora: one for the source code and one for the vulnerability report. A slightly tweaked Vector Space Model (tVSM) is then applied to rank the source code functions based on their relevance to a specific vulnerability, adjusting for function size. The accuracy of this minimized search space is evaluated theoretically and experimentally. The technique introduces a Function Statement Dependency Graph (FSDG) to capture relationships among statements within functions, and a Predecessor Node Dependency Graph (PNDG) to combine statements with their predecessors for richer information. Each node in the PNDG undergoes text processing to create a bag of words, and the similarity between these nodes and the vulnerability report is measured using VSM, producing a ranked list of potentially vulnerable statements. This method enhances the precision of vulnerability detection by focusing on relevant code segments and

enriching statement information.

SQ3: What is the impact of combining vulnerability reports with source code in improving performance and developing an effective statement-level vulnerability detection technique?

For answering sub research question 3 the proposed VFSDetector was evaluated using the comprehensive Big-Vul dataset, which includes reliable code vulnerabilities linked to the CVE database and encompasses 348 open-source projects from 2002 to 2019, covering 91 Common Weakness Enumeration (CWE) categories. The dataset comprises around 188,600 C/C++ functions, with 5.6% identified as vulnerable. Six metrics were used to evaluate the VFSDetector: Top N Rank, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), Top-k Accuracy, Top-k Precision, and Top-k Recall. The performance was tested on 1200 vulnerability reports across 14 different projects, revealing strong detection capabilities. Graphical comparisons showed that the VFSDetector maintained high accuracy across projects of varying sizes and complexities, with notable performance in projects like Tcpdump and ImageMagick, which achieved 40% accuracy at the top 1 position and near-perfect accuracy within the top 20 ranks. Medium-sized projects such as Openssl, Wireshark, FFmpeg, and Kuberos also exhibited robust performance, further illustrating the tool's consistent accuracy and reliability.

The VFSDetector demonstrated exceptional detection performance, with projects like Tcpdump and ImageMagick achieving high accuracy in identifying vulnerabilities. Medium-sized projects also showed strong results, maintaining consistent performance across different top-k rankings. The Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) metrics substantiated the tool's effectiveness, with projects like ImageMagick and Tcpdump displaying high MRR values of 69.3% and 66%, respectively, indicating that vulnerable statements were ranked high. The MAP scores further supported the tool's precision, with Tcpdump and ImageMagick scoring 61% and 51.3%, respectively. Comparatively, VFSDetector

outperformed baseline methods such as LineVD, IVDetect, LineVul, and COSTA in terms of accuracy, precision, and recall across varoious top-k values, consistently achieving higher scores. This superiority underscores the VFSDetector's capability to detect vulnerabilities accurately and reliably, contributing significantly to advancing vulnerability detection techniques.

6.3 Future Directions Related to this Thesis

In this thesis two automated vulnerability detection approaches are proposed that detects vulnerabilities at the function level. One method combined text and graph based approaches to detect vulnerability in statement levl and another appproach used vulnerability reports to narrow down the detection in statement level. Future research could explore the following directions based on this work:

- **Experimenting alternative benchmark datasets:** Although automated software vulnerability detcetion experiments in statement level are time-consuming [122], the proposed approaches are evaluated using both small and large projects from Big-Vul [82] and SARD [100] benchmark datasets to ensure generalizability. In future, these approaches can be further explored using other real world industrial projects or datasets to gain insights about different vulnerability types and their severity level.

- **Identifying common fix patterns for statement level vulnerabilities:** This thesis investigates the impact of combining text and graph based approach and vulnerability reports to narrow the search space and accurately distinguish correct vulnerable statements or functions from large code bases. Future work will focus on identifying common fix patterns for statement level vulnerability, such as method or variable name replacements. These patterns can then be incorporated into the proposed methods to see if they further limit the search space and help eliminate incorrect vulnerability detection.

- **Assigning weights by similarity score to the centrality values in Image Construction phase:**

  In Chapter 4, during the image construction phase of our methodology, we calculate centrality values for all nodes within the embedded Program Dependency Graph (PDG). We then generate resulting vectors by multiplying the vector values of each line of code by the corresponding centrality measure, denoted as $V1 \times d1$ (where $V1$ represents the vector for line number 1 and $d1$ the degree centrality value). An improvement can be made by assigning weights to each line of code based on its similarity score, resulting in $V1 \times d1 \times S1$ (Vector $\times$ Centrality $\times$ Similarity). This enhanced weighting system could be implemented using a machine learning model, such as a Convolutional Neural Network (CNN), to dynamically assign different weights based on the significance of each line of code. This would lead to a more accurate construction of images from source code. However, developing such a model requires a substantial and diverse dataset of vulnerabilities along with corresponding calculations for each function and statement, which is beyond the scope of this thesis. This will be addressed as a separate work in future research.

- **Incorporating the proposed approaches with other automated software vulnerability techniques:** The goal of this thesis is to explore vulnerability detection techniques that handle vulnerabilities at statement level. In future, this approach can be integrated with other existing vulnerability detection approaches that work on statement level [86], [69] or function level [83]. Modifying these techniques to incorporate the proposed approaches could complement and enhance their effectiveness.

6.4 Future Directions Related to the Domain

Looking ahead in the realm of vulnerability detection, the introduction of statement-level vulnerability detection via vulnerability reports within this thesis hints at potential enhancements in the quality of vulnerability reports themselves. This suggests that as this method gains traction, vulnerability reports may become more detailed and informative, facilitating more effective vulnerability resolution processes in software development.

Given that statement-level vulnerability detection is a relatively nascent field, it is plausible that further advancements will occur, leading to the development of even more precise techniques for pinpointing vulnerability at this level. As researchers delve deeper into this area, they may uncover novel approaches and methodologies that improve upon existing techniques, thereby refining the accuracy and efficiency of vulnerability detection processes.

Furthermore, the superior performance demonstrated by the proposed technique, particularly in the context of open-source projects, highlights its potential applicability beyond specific domains. This suggests that the method could be seamlessly integrated into projects across various industries, offering benefits in terms of bug detection and resolution regardless of the sector. Therefore, as the field progresses, we anticipate broader adoption of statement-level vulnerability detection techniques, contributing to more robust and reliable software development practices across diverse project landscapes.

Appendix A

During pre-processing of source code or vulnerability reports the unnecessary tokens in source code which do not contain any vulnerability related information. These tokens are discarded from source code such as stop words or programming language specific keywords. The list of stop words in C/C++ language are given below in Appendix A.1 and A.2.

A.1 Stop Words

a, as, able, about, above, according, accordingly, across, actually, after, afterwards, again, against, ain't, all, allow, allows, almost, alone, along, already, also, although, always, am, among, amongst, an, and, another, any, anybody, anyhow, anyone, anything, anyway, anyways, anywhere, apart, appear, appreciate, appropriate, are, aren't, around, as, aside, ask, asking, associated, at, available, away, awfully, b, be, became, because, become, becomes, becoming, been, before, beforehand, behind, being, believe, below, beside, besides, best, better, between, beyond, both, brief, but, by, c, c'mon, c's, came, can, can't, cannot, can't, cause, causes, certain, certainly, changes, clearly, co, com, come, comes, concerning, consequently, consider, considering, contain, containing, contains, corresponding, could, couldn't, course, currently, d, definitely, described, despite, did, didn't, different, do, does, doesn't, doing, don't, done, down, downwards, during, e, each, edu, eg, eight, either, else, elsewhere, enough, entirely, especially, et, etc, even,

ever, every, everybody, everyone, everything, everywhere, ex, exactly, example, except, f, far, few, fifth, first, five, ve, followed, following, follows, for, former, formerly, forth, four, from, further, furthermore, g, get, gets, getting, given, gives, h, had, hadn't, happens, hardly, has, hasn't, have, haven't, having, he, he's, hello, help, hence, her, here, here's, hereafter, hereby, hereupon, hers, herself, hi, him, himself, his, hither, hopefully, how, howbeit, however, i, i'd, i'll, i'm, i've, ie, if, ignored, immediate, in, inasmuch, inc, indeed, indicate, indicated, indicates, inner, insofar, instead, into, inward, is, isn't, it, it'd, it'll, it's, its, itself, just, k, keep, keeps, kept, know, knows, known, l, last, lately, later, latter, latterly, least, less, lest, let, let's, like, liked, likely, little, look, looking, looks, ltd, m, mainly, many, may, maybe, me, mean, meanwhile, merely, might, more, moreover, most, mostly, much, must, my, myself, n, name, namely, nd, near, nearly, necessary, need, needs, neither, never, nevertheless, new, next, nine, no, nobody, none, noone, nor, normally, not, nothing, novel, now, nowhere, o, obviously, of, oh, ok, okay, old, on, once, one, ones, only, onto, or, other, others, otherwise, ought, our, ours, ourselves, out, outside, over, overall, own, p, particular, particularly, per, perhaps, placed, please, plus, possible, presumably, probably, provides, q, que, quite, qv, rather, rd, re, really, reasonably, regarding, regardless, regards, relatively, respectively, right, s, said, same, saw, say, saying, says, second, secondly, see, seeing, seem, seemed, seeming, seems, seen, self, selves, sensible, sent, serious, seriously, seven, several, shall, she, she'd, she'll, she's, should, shouldn't, since, six, so, some, somebody, somehow, someone, something, sometime, sometimes, somewhat, somewhere, soon, sorry, specified, specify, specifying, still, sub, such, sup, sure, t, t's, take, taken, tell, tends, th, than, thank, thanks, thanx, that, that's, that'll, the, their, theirs, them, themselves, then, thence, there, there's, thereafter, thereby, therefore, therein, there'll, there's, thereupon, these, they, they'd, they'll, they're, they've, think, third, this, thorough, thoroughly, those, though, three, through, throughout, thru, thus, to, together, too, took, toward, towards, tried, tries, truly,

try, trying, twice, two, u, un, under, unfortunately, unless, unlikely, until, unto, up, upon, us, use, used, useful, uses, using, usually, uucp, v, value, various, very, via, viz, vs, w, want, wants, was, wasn't, way, we, we'd, we'll, we're, we've, welcome, well, went, were, weren't, what, what's, whatever, when, whence, whenever, where, where's, whereafter, whereas, whereby, wherein, whereupon, wherever, whether, which, while, whither, who, who's, whoever, whole, whom, whose, why, will, willing, wish, with, within, without, won't, wonder, would, wouldn't, x, y, yes, yet, you, you'd, you'll, you're, you've, your, yours, yourself, yourselves, z, zero, quot.

A.2 Programme Language Specific Keywords

auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, inline, int, long, register, restrict, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while, _Bool, _Complex, _Imaginary,alignas, alignof, and, and_eq, asm, bitand, bitor, bool, catch, char16_t, char32_t, class, compl, const_cast, constexpr, decltype, delete, dynamic_cast, explicit, export, false, friend, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, reinterpret_cast, static_assert, static_cast, template, this, thread_local, throw, true, try, typeid, typename, using, virtual, wchar_t, xor, xor_eq, override, final,constexpr, decltype, nullptr, static_assert, thread_local.

Bibliography

[1] CVE website. Available at: https://www.cve.org/CVERecord?id=CVE-2022-3456 [Accessed: 23 May 2024].

[2] Navlani, A. (2018) Python LSI/LSA (Latent Semantic Indexing/analysis), DataCamp. Available at: https://www.datacamp.com/tutorial/discovering-hidden-topics-python [Accessed: 23 May 2024].

[3] Otten, N.V. (2023) Vector space model made simple with examples tutorial in Python, Spot Intelligence. Available at: https://spotintelligence.com/2023/09/07/vector-space-model/ [Accessed: 23 May 2024].

[4] D. Buenano-Fernandez, M. Gonzalez, D. Gil, and S. Luján-Mora, "Text mining of open-ended questions in self-assessment of university teachers: An lda topic modeling approach," *Ieee Access*, vol. 8, pp. 35318–35330, 2020.

[5] A. F. Hidayatullah, S. K. Aditya, S. T. Gardini, *et al.*, "Topic modeling of weather and climate condition on twitter using latent dirichlet allocation (lda)," in *IOP Conference Series: Materials Science and Engineering*, vol. 482, p. 012033, IOP Publishing, 2019.

[6] H. Alves, B. Fonseca, and N. Antunes, "Software metrics and security vulnerabilities: dataset and exploratory study," in *2016 12th European Dependable Computing Conference (EDCC)*, pp. 37–44, IEEE, 2016.

[7] J. Han, D. Gao, and R. H. Deng, "On the effectiveness of software diversity: A systematic study on real-world vulnerabilities," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 127–146, Springer, 2009.

[8] "National vulnerability database (NVD)" (2023) NIST. Available at: https://www.nist.gov/programs-projects/national-vulnerability-database-nvd (Accessed: 09 May 2024).

[9] "Security vulnerabilities, CVES", published in 2023 "(2023). Available at: https://www.cvedetails.com/vulnerability-list/year-2023/vulnerabilities.html (Accessed: 09 May 2024).

[10] D. L. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*, pp. 279–287, IEEE, 1994.

[11] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt, "Automated analysis of requirement specifications," in *Proceedings of the 19th international conference on Software engineering*, pp. 161–171, 1997.

[12] O. Lahav and D. Mioduser, "Multisensory virtual environment for supporting blind persons' acquisition of spatial cognitive mapping, orientation, and mobility skills," in *Proc. 4th Intl. Conf. Disability, Virtual Reality & Assoc. Tech*, pp. 213–220, 2002.

[13] J. Zhang, S. Liu, X. Wang, T. Li, and Y. Liu, "Learning to locate and describe vulnerabilities," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 332–344, IEEE, 2023.

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

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[16] R. S. Mim, T. Ahammed, and K. Sakib, "Identifying vulnerable functions from source code using vulnerability reports," in *IWESQ/QuASoQ@APSEC*, 2023.

[17] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *Journal of systems and software*, vol. 9, no. 3, pp. 191–195, 1989.

[18] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, "Localizing vulnerabilities statistically from one exploit," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pp. 537–549, 2021.

[19] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 533–544, 2019.

[20] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd annual conference on computer security applications*, pp. 201–213, 2016.

[21] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *Acm Sigplan Notices*, vol. 41, no. 1, pp. 372–382, 2006.

[22] P. Anand, "Overview of root causes of software vulnerabilities-technical and user-side perspectives," in *2016 International Conference on Software Security and Assurance (ICSSA)*, pp. 70–74, IEEE, 2016.

[23] F. Piessens, "A taxonomy of causes of software vulnerabilities in internet software," in *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*, pp. 47–52, Citeseer, 2002.

[24] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM computing surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.

[25] G. Schryen, "Security of open source and closed source software: An empirical comparison of published vulnerabilities," *AMCIS 2009 Proceedings*, p. 387, 2009.

[26] A. Ahadi, R. Lister, S. Lal, and A. Hellas, "Learning programming, syntax errors and institution-specific factors," in *Proceedings of the 20th Australasian computing education conference*, pp. 90–96, 2018.

[27] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 311–322, IEEE, 2018.

[28] M. Zeng, Y. Wu, Z. Ye, Y. Xiong, X. Zhang, and L. Zhang, "Fault localization via efficient probabilistic modeling of program semantics," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 958–969, 2022.

[29] W. Li and N. Li, "A formal semantics for program debugging," *Science China Information Sciences*, vol. 55, pp. 133–148, 2012.

[30] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 international workshop on Mining software repositories*, pp. 1–5, 2005.

[31] M. Pinzger, K. Grafenhain, P. Knab, and H. C. Gall, "A tool for visual understanding of source code dependencies," in *2008 16th IEEE International conference on program comprehension*, pp. 254–259, IEEE, 2008.

[32] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *ACM Sigplan Notices*, vol. 19, no. 5, pp. 177–184, 1984.

[33] L. Sampaio and A. Garcia, "Exploring context-sensitive data flow analysis for early vulnerability detection," *Journal of Systems and Software*, vol. 113, pp. 337–361, 2016.

[34] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2017.

[35] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 652–661, IEEE, 2013.

[36] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 41–50, IEEE, 2019.

[37] 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*, pp. 590–604, IEEE, 2014.

[38] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *2009 IEEE international conference on software maintenance*, pp. 351–360, IEEE, 2009.

[39] D. Binkley and D. Lawrie, "Information retrieval applications in software maintenance and evolution," *Encyclopedia of software engineering*, pp. 454–463, 2010.

[40] M. Vanamala, X. Yuan, and K. Roy, "Topic modeling and classification of common vulnerabilities and exposures database," in *2020 International Conference on Artificial Intelligence, Big Data, Computing and Data Communication Systems (icABCD)*, pp. 1–5, IEEE, 2020.

[41] A. Mahmoud and G. Bradshaw, "Semantic topic models for source code analysis," *Empirical Software Engineering*, vol. 22, pp. 1965–2000, 2017.

[42] M. Edmund, J. Clarke, O. Grumberg, and D. A. Peled, "Model checking," *MIT Press*, p. 314, 1999.

[43] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977.

[44] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Model Checking Software: 10th International SPIN Workshop Portland, OR, USA, May 9–10, 2003 Proceedings 10*, pp. 235–239, Springer, 2003.

[45] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 544–555, 2022.

[46] T. Ball and S. K. Rajamani, "The slam toolkit," in *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, vol. 2102, pp. 260–264, 2000.

[47] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[48] V. Mulwad, W. Li, A. Joshi, T. Finin, and K. Viswanathan, "Extracting information about security vulnerabilities from web text," in *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, vol. 3, pp. 257–260, IEEE, 2011.

[49] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus, "On the relationship between the vocabulary of bug reports and source code," in *2013 IEEE International Conference on Software Maintenance*, pp. 452–455, IEEE, 2013.

[50] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[51] E. Hill, S. Rao, and A. Kak, "On the use of stemming for concern location and bug localization in java," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 184–193, IEEE, 2012.

[52] C. D. Paice, "An evaluation method for stemming algorithms," in *SIGIR'94: Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval, organised by Dublin City University*, pp. 42–50, Springer, 1994.

[53] L. Wang, Q. Zhang, and P. Zhao, "Automated detection of code vulnerabilities based on program analysis and model checking," in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 165–173, IEEE, 2008.

[54] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[55] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*, pp. 797–812, IEEE, 2015.

[56] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Transactions on dependable and secure computing*, vol. 12, no. 6, pp. 688–707, 2014.

[57] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th annual computer security applications conference*, pp. 359–368, 2012.

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

[59] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[60] 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 *IJCAI*, pp. 4665–4671, 2019.

[61] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

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

[63] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[64] 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)*, pp. 757–762, IEEE, 2018.

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

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

[67] Y. Chen, "Convolutional neural network for sentence classification," Master's thesis, University of Waterloo, 2015.

[68] T.-T. Nguyen and H. D. Vo, "Context-based statement-level vulnerability localization," *Information and Software Technology*, vol. 169, p. 107406, 2024.

[69] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 608–620, 2022.

[70] "Flawfinder Home Page," dwheeler.com. Available: https://www.dwheeler.com/flawfinder/. [Accessed: May 14, 2024].

[71] "Rough Audit Tool for Security," Accessed: Jun. 5, 2014. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security.

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

[73] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd annual conference on computer security applications*, pp. 201–213, 2016.

[74] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[75] 2023. "National Vulnerability Database," [Online]. Available: https://nvd.nist.gov. [Accessed: 28-Dec-2023].

[76] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3156–3164, 2015.

[77] "Debian – The Universal Operating System," Debian.org, 2019. Available: https://www.debian.org/.[Accessed: May 14, 2024].

[78] "GitHub: Where the world builds software," GitHub, 2022. Available: https://github.com/. [Accessed: May 14, 2024].

[79] "Cppcheck - A tool for static C/C++ code analysis," cppcheck.sourceforge.net. Available: http://cppcheck.sourceforge.net/.[Accessed: May 14, 2024].

[80] "Common weakness enumeration (CWE) ". Available at: https://cwe.mitre.org/ (Accessed: 14 May 2024).

[81] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 international conference on engineering and technology (ICET)*, pp. 1–6, Ieee, 2017.

[82] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 508–512, 2020.

[83] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th international conference on mining software repositories*, pp. 596–607, 2022.

[84] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray, "Velvet: a novel ensemble learning approach to automatically locate vulnerable statements," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 959–970, IEEE, 2022.

[85] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: An image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2365–2376, 2022.

[86] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 292–303, 2021.

[87] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, pp. 618–626, 2017.

[88] A. Chattopadhay, A. Sarkar, P. Howlader, and V. N. Balasubramanian, "Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks," in *2018 IEEE winter conference on applications of computer vision (WACV)*, pp. 839–847, IEEE, 2018.

[89] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[90] "Open Source Security and analysis report," Jun. 2023. [Online]. Available: https://www.synopsys.com/software-integrity/resources.html.

[91] R. S. Mim, A. Khatun, T. Ahammed, and K. Sakib, "Impact of centrality on automated vulnerability detection using convolutional neural network," in *2023 International Conference on Information and Communication Technology for Sustainable Development (ICICT4SD)*, pp. 331–335, IEEE, 2023.

[92] M. N. Moghadasi and Y. Zhuang, "Sent2vec: A new sentence embedding representation with sentimental semantic," in *2020 IEEE International Conference on Big Data (Big Data)*, pp. 4672–4680, IEEE, 2020.

[93] L. C. Freeman *et al.*, "Centrality in social networks: Conceptual clarification," *Social network: critical concepts in sociology. Londres: Routledge*, vol. 1, pp. 238–263, 2002.

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

[95] 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*, pp. 590–604, IEEE, 2014.

[96] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[97] R. Guimera, S. Mossa, A. Turtschi, and L. N. Amaral, "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles," *Proceedings of the National Academy of Sciences*, vol. 102, no. 22, pp. 7794–7799, 2005.

[98] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, no. 6833, pp. 41–42, 2001.

[99] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 8609–8613, IEEE, 2013.

[100] "sard." "Software Assurance Reference Dataset," [Online]. Available: https://samate.nist.gov/SRD/index.php. [Accessed: 18-Jun-2023].

[101] 2023. "National Institute of Standards and Technology. [Online] https://samate.nist.gov/SRD/index.php [Accessed : 28-Dec-2023].

[102] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 528–540, 2018.

[103] J. Han, D. Gao, and R. H. Deng, "On the effectiveness of software diversity: A systematic study on real-world vulnerabilities," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 127–146, Springer, 2009.

[104] 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 *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[105] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[106] 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*, pp. 590–604, IEEE, 2014.

[107] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*, vol. 39. Cambridge University Press Cambridge, 2008.

[108] W. B. Croft, D. Metzler, and T. Strohman, *Search engines: Information retrieval in practice*, vol. 520. Addison-Wesley Reading, 2010.

[109] S. Rahman and K. Sakib, "An appropriate method ranking approach for localizing bugs using minimized search space.," in *ENASE*, pp. 303–309, 2016.

[110] S. Rahman, M. M. Rahman, and K. Sakib, "A statement level bug localization technique using statement dependency graph.," in *ENASE*, pp. 171–178, 2017.

[111] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[112] H. Zhang, "An investigation of the relationships between lines of code and defects," in *2009 IEEE international conference on software maintenance*, pp. 274–283, IEEE, 2009.

[113] W. B. Croft, D. Metzler, and T. Strohman, *Search engines: Information retrieval in practice*, vol. 520. Addison-Wesley Reading, 2010.

[114] S. T. Dumais, "Improving the retrieval of information from external sources," *Behavior research methods, instruments, & computers*, vol. 23, no. 2, pp. 229–236, 1991.

[115] E. Garcia, "The classic vector space model description, advantages and limitations of the classic vector space model," *online] http://www. miislita. com/term-vector/term-vector-3. html*, 2006.

[116] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proceedings of the 2011 international symposium on software testing and analysis*, pp. 199–209, 2011.

[117] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering (TSE)*, 2017.

[118] X.-B. D. Le, "Towards efficient and effective automatic program repair," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pp. 876–879, ACM, 2016.

[119] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 12–23, 2018.

[120] Z. Xu, T. Kremenek, and J. Zhang, "A memory model for static analysis of c programs," in *Leveraging Applications of Formal Methods, Verification, and Validation: 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I 4*, pp. 535–548, Springer, 2010.

[121] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2469–2485, 2019.

[122] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: a survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.