

Towards Automatic Vulnerability Management in Open-Source Software

by

Xu Yang

A Thesis submitted to
Faculty of Graduate Studies of
University of Manitoba
in partial fulfillment of the requirements
of the degree of

Ph.D. in Computer Science

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada
June 2025

© Copyright 2025 by Xu Yang

Thesis advisor
Shaowei Wang

Author
Xu Yang

Towards Automatic Vulnerability Management in Open-Source Software

Abstract

Open-source software (OSS) is the foundation of the digital world, yet its transparent development model exposes systems to security vulnerabilities. Effective OSS vulnerability management requires two fundamental capabilities: vulnerability detection (VD), which identifies vulnerable code in repositories before exploitation, and vulnerability-fix detection (VFD), which monitors upstream commits to identify security patches. While VD prevents vulnerable code from reaching production, VFD enables downstream organizations to proactively apply critical patches before public disclosure.

However, data challenges limit current approaches. This thesis addresses four critical data-centric challenges through targeted studies. First challenge is extreme class imbalance. Vulnerability code comprising less than 5% repositories, data imbalance causing detection models to lose up to 73% effectiveness. We systematically evaluated sampling strategies on vulnerability detection models, demonstrating that random oversampling on raw code improves both VD model’s performance and interpretability.

Second, long-tail vulnerability distributions leave rare but critical vulnerability

types underrepresented in training data, yet these edge cases often matter most for safety-critical systems. We developed MoEVD, a mixture-of-experts architecture that routes input to specialized experts, achieving 12.8% higher F1 while improving detection of underrepresented categories.

Third, security-critical code changes are often subtle, only a small part of code changes are security related. Existing work often overlook such distinctive patterns. We created VFDelta using fine-grained delta embeddings to capture nuanced changes, improving F1 by 77.4%.

Fourth, vulnerability fixes hide within tangled commits combining security patches with unrelated changes, while crucial context resides in scattered artifacts. We introduced LLM4VFD, which uses large language models to integrate code changes with development artifacts and vulnerability history, achieving 68-145% F1 improvements while providing explanations.

This thesis provides novel techniques and practical guidance for improving automated vulnerability and vulnerability-fix detection in OSS. The presented approaches address four critical limitations in existing tools through data-centric solutions, offering both enhanced detection accuracy and explanations to bridges the gap between academic prototypes and real-world deployment.

Preface

0.1 Contribution of Authors

This thesis is the original work of the author, Xu Yang. The research presented herein was conducted under the supervision of Dr. Shaowei Wang in the Software Management, Maintenance, and Reliability (Mamba) Lab. Chapters 3, 4, 5, and 6 are based on collaborative research that has been published, accepted for publication, or submitted for review in peer-reviewed venues. The specific contributions of all authors for each of these works are detailed below.

Chapter 3: Does Data Sampling Improve Deep Learning-based Vulnerability Detection? Yeas! and Nays!

This chapter is based on a paper co-authored with Shaowei Wang, Yi Li, and Shaohua Wang, published in the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), doi: 10.1109/ICSE48619.2023.00192. As the primary author, my responsibilities included the conceptualization of the study, designing the experimental methodology, implementing the data sampling techniques, conducting all experiments, analyzing the results, and writing the initial manuscript. Yi Li provided helps with experiment results for baseline methods. My thesis supervisor,

Shaowei Wang, provided overall guidance and supervision for the project, contributing significantly to refining the research questions, interpreting the results, and revising the manuscript for clarity and impact. Shaohua Wang contributed through valuable discussions, provided critical feedback on the experimental design, and assisted in the review and revision of the final manuscript.

Chapter 4: One-for-All Does Not Work! Enhancing Vulnerability Detection by Mixture-of-Experts (MoE)

This chapter is based on a paper co-authored with Shaowei Wang, Jiayuan Zhou, and Wenhan Zhu, accepted for publication in The 2025 ACM International Conference on the Foundations of Software Engineering (FSE), doi:10.1145/3715736. As the first author, my role involved designing the MoEVD framework, implementing the architecture, conducting the evaluation, analyzing the findings, and authoring the manuscript. Shaowei Wang provided supervision, expert guidance on the methodology, and contributed to the critical review and revision of the paper. Jiayuan Zhou and Wenhan Zhu participated in technical discussions, offered feedback that helped refine the framework, and contributed to the revision of the manuscript.

Chapter 5: VFDelta: A Framework for Detecting Silent Vulnerability Fixes by Enhancing Code Change Learning

This chapter is based on a paper co-authored with Shaowei Wang, Jiayuan Zhou, and Xing Hu, submitted to the ACM Transactions on Software Engineering and Methodology (TOSEM) journal. As the first author, I designed and implemented the VFDelta framework, including the novel Code Delta Representation, and was responsible for all experiments, data analysis, and the preparation of the manuscript.

Shaowei Wang provided supervision and high-level guidance, contributed to the core ideas of the framework, and extensively reviewed and edited the manuscript. Jiayuan Zhou and Xing Hu contributed through insightful discussions, providing feedback on the methodology and experimental results, and assisted in revising the manuscript.

Chapter 6: Code Change Intention, Development Artifact, and History Vulnerability: Putting Them Together for Vulnerability Fix Detection by LLM

This chapter is based on a paper co-authored with Wenhan Zhu, Michael Pacheco, Jiayuan Zhou, Shaowei Wang, Xing Hu, and Kiu Liu, accepted for publication in The 2025 ACM International Conference on the Foundations of Software Engineering (FSE), doi:10.1145/3715738. I was the primary researcher for this project, and my contributions included designing the LLM4VFD framework, collecting and processing the data, implementing the methodology, conducting the experiments and user study, analyzing the results, and writing the manuscript. Wenhan Zhu, Michael Pacheco, and Jiayuan Zhou contributed to the development of the methodology and assisted with data collection and analysis, providing valuable feedback during the manuscript preparation. As my supervisor, Shaowei Wang provided critical guidance throughout the project, helped shape the research direction, and contributed to the review and revision of the manuscript. Xing Hu and Kiu Liu provided important feedback through technical discussions and contributed to the review of the final paper.

0.2 Acknowledgments

I would like to express my deepest gratitude to my supervisor, Dr. Shaowei Wang. His mentorship, support, and insightful guidance have been the cornerstone

of my PhD journey. Dr. Wang's expertise and encouragement not only shaped the direction of this research but also fostered my growth as a researcher.

I'am also grateful for the opportunity to have collaborate with such a talented group of researchers. The contributions, insights, and critical feedback from all my co-authors were invaluable at every stage of this work. This thesis would not have been possible without their shared expertise and dedication.

Contents

Abstract	ii
0.1 Contribution of Authors	iv
0.2 Acknowledgments	vi
Preface	vii
Table of Contents	xii
List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 OSS Vulnerability Management	2
1.2 Challenges	5
1.3 Thesis Objectives	7
1.4 Thesis Outline	12
2 Literature Review	14
2.1 Machine learning-based Vulnerability detection	14
2.2 Class-Imbalance Mitigation via Data Sampling	15
2.3 Mixture-of-Experts Frameworks in Software-Engineering Task	16
2.4 Approaches to Vulnerability-Fix Detection	16
2.5 Large Language Models for Software Engineering Task	17
3 Does Data Sampling Improve Deep Learning-based Vulnerability Detection? Yeas! and Nays!	19
3.1 Introduction	19
3.2 Background	22
3.2.1 Overview of deep learning-based vulnerability detection	22
3.2.2 Existing DL-based vulnerability detection approaches	23
3.2.3 Data sampling	24
3.3 Experimental design	27
3.3.1 Research questions	27
3.3.2 Datasets	28

3.3.3	Studied DLVD & Data sampling approaches	29
3.3.4	Evaluation metrics	30
3.3.5	Approach of RQs	31
	Approach of RQ1	32
	Approach of RQ2	32
3.3.6	Implementation details	34
3.4	Results	35
3.4.1	RQ1: Does data sampling improve the effectiveness of existing DLVD approaches?	35
3.4.2	RQ2: Does data sampling improve the ability of DLVD for learning the vulnerable patterns?	40
3.5	Discussion	43
3.5.1	Why does ROS_R perform the best among all studied approaches?	43
3.5.2	Why does random undersampling boost recall?	44
3.5.3	Implications of our findings	46
3.5.4	Threats to validity	48
3.6	Conclusion	49
4	One-for-All Does Not Work! Enhancing Vulnerability Detection by Mixture-of-Experts (MoE)	51
4.1	Introduction	51
4.2	Background and Motivation	55
4.2.1	Pre-trained Model for Code Representation Learning	55
4.2.2	Common Weakness Enumeration	56
4.2.3	Mixture-of-Experts (MoE)	57
	Mixture-of-Experts Framework Design	57
	Vulnerability Detection with Mixture-of-Experts	58
4.3	Methodology	59
4.3.1	Splitting Input Space Based on CWE Types	60
4.3.2	Training Phase	62
	Expert Training	62
	Router Training	63
	Combiner	65
4.3.3	Inference Phase	66
4.4	Experiment Design	67
4.4.1	Research Questions	67
4.4.2	DLVD Baselines	68
4.4.3	Experiment Dataset	69
4.4.4	Evaluation Metrics	70
4.4.5	Approach for Research Questions	70
	Approach of RQ1	70
	Approach of RQ2	70

	Approach of RQ3	71
	Approach of RQ4	72
4.4.6	Implementation Details	73
4.5	Results	74
4.5.1	RQ1: MoEVD vs. DLVD Baselines	74
4.5.2	RQ2: Effectiveness of MoE	75
4.5.3	RQ3: Effectiveness of MoEVD across CWE Types	78
4.5.4	RQ4: Impact of the Number of Selected Experts	80
4.6	Discussion	81
4.6.1	Expert Training Task	81
4.6.2	Overhead of MoE Compared With One-for-All Model	82
4.6.3	Potential Future Direction	82
4.6.4	Threats to Validity	83
	Internal Validity	83
	External Validity	84
4.7	Conclusion	84
5	VFDelta: A Framework for Detecting Silent Vulnerability Fixes by Enhancing Code Change Learning	85
5.1	Introduction	85
5.2	Background	90
5.2.1	Silent commit-level vulnerability fix identification	90
5.2.2	Code-Change Representation Learning with Pre-trained Models	93
5.3	Methodology	93
5.3.1	File-level vulnerability-fixing classification model training	95
	Context-reserved Code Change Construction	95
	Code Delta Representation	95
	Model training	96
5.3.2	Commit-level Inference	97
5.3.3	Why element-wise subtraction?	97
5.4	Experimental Setting	98
5.4.1	Research Questions	99
5.4.2	Dataset preparation	99
	VFM2021 dataset	99
	Enhanced temporal VFM2023 dataset	101
5.4.3	Evaluation metrics	102
5.4.4	Code embedding models	103
5.4.5	Implementation details	103
5.4.6	Approach for RQs	103
	Approach of RQ1	103
	Approach of RQ2	104
	Approach of RQ3	107

	Approach of RQ4	107
5.5	Results	108
5.5.1	Results of RQ1	108
5.5.2	Results of RQ2	111
5.5.3	Results of RQ3	116
5.5.4	Results of RQ4	117
5.6	Discussion	119
5.6.1	Scalability and computational cost	119
5.6.2	Precision and Recall trade-off in practice	120
5.6.3	Real World implementation of VFDelta	120
5.6.4	Potential applications of VFDelta on other tasks	121
5.6.5	Ethical implications	121
5.6.6	Threats to Validity	122
5.7	Conclusion	123
6	Code Change Intention, Development Artifact, and History Vulnerability: Putting Them Together for Vulnerability Fix Detection by LLM	124
6.1	Introduction	124
6.2	Background	128
6.2.1	Vulnerability Fix Detection	128
6.2.2	Large Language Models	129
6.3	Methodology	129
6.3.1	Code Change Intention (CCI)	131
6.3.2	Development Artifact (DA)	132
6.3.3	Historical Vulnerability (HV)	135
6.3.4	Comprehensive Analysis and Vulnerability Fix Detection (CAVFD)	136
6.4	Experimental Settings	137
6.4.1	Research Questions	137
6.4.2	Data Collection	137
	Date Range Selection	137
	Vulnerability and Non-vulnerability Fix Commit Selection . .	139
6.4.3	IR/PR Data Collection	140
6.4.4	Historical Vulnerability Data Collection	141
6.4.5	Implementation Detail	141
	LLMs Selection	141
	Embedding Model and RAG Implementation	142
6.4.6	Evaluation Metrics	142
6.4.7	RQ Approaches	143
	<i>RQ1: How effective is LLM4VFD compared with SOTA techniques?</i>	143
	<i>RQ2: How effective is each component in LLM4VFD?</i>	144

	<i>RQ3: Can the analysis generated by LLM4VFD help security experts in identifying vulnerability fixes?</i>	144
	<i>RQ4: Bad Case Analysis: In which scenarios does LLM4VFD fail?</i>	146
6.5	Results	147
6.5.1	RQ1 - Effectiveness	147
	PLM-based Approaches vs. LLM4VFD	147
	Vanilla LLMs (0-shot CoT) vs. LLM4VFD	148
6.5.2	RQ2 - Ablation analysis.	150
6.5.3	RQ3 - User Study	151
6.5.4	RQ4 - Failure Analysis	153
6.6	Discussion	155
6.6.1	Compromising between Precision and Recall	155
6.6.2	Potential Future Direction	156
6.6.3	Threats to Validity	156
	Internal Validity	156
	External Validity	157
6.7	Conclusion	157
7	Conclusion and Future Work	159
7.1	Conclusion	159
7.2	Future Work	161
	Bibliography	192

List of Figures

3.1	Overview of our approach. The deep learning-based vulnerability detection framework is presented in the green rectangle. We mark steps where the proper data sampling approaches could be performed in red round-rectangles.	22
3.2	The explanation of LIME on a vulnerable function, which is predicted wrongly as non-vulnerable before ROS (left) and predicted correctly after ROS_R (right).	42
3.3	The distribution of data points in the experiment instance <i>LineVul</i> + <i>BigVul</i> after applying ROS_R and ROS_L.	44
3.4	The distribution of data points in the Reveal dataset without (NoSampling) and without applying RUS_R.	45
4.1	The performance of one-for-all Model (LineVul) vs. expert model trained on each specific CWE type across CWE types.	55
4.2	Hierarchy structure of CWE types. The arrows present the parent-child relationship between two CWE types. For instance, CWE-664 is the parent of CWE-221.	55
4.3	An Example Design of Mixture-of-Experts (MoE).	57
4.4	The pipeline of MoEVD.	60
4.5	The performance matrix of each expert model on each CWE type in terms of F1-score. Darker colors indicate better performance.	77
4.6	Recall of CodeBERT, MoEVD _{CodeBERT} , and CausalVul _{CodeBERT} on different CWE types. The CWE types are displayed in descending order of frequency, from left to right.	79
4.7	F1-score of MoEVD _{CodeBERT} and MoEVD _{UniXcoder} with different number of selected experts (K).	79
5.1	A sample file-level code changes for fixing CVE-2018-11776 [1] vulnerability.	92
5.2	The workflow of VFDelta.	94
5.3	Comparison of different code change representation learning approaches.	105

5.4	Comparison of training loss between VFDelta, EmbedSubtract_Single and EmbedConcat_Duo.	113
5.5	The barplot denotes the effectiveness of different approaches on the groups of commits with different change sizes in terms of F1. The black line denotes the proportion of each group of commits. For instance, 46% indicates 46% of the commits have a code change size between 0 and 20.	114
5.6	Vulnerability fix for CVE-2018-1304 with minor changes that EmbedSubtract_Duo (VFDelta) successfully detected while other variant failed.	115
5.7	A Silent Vulnerability Fix for CVE-2022-46363 [2] detected by VFDelta.	119
6.1	The framework of LLM4VFD.	131
6.2	The prompt template of Code Change Intention.	133
6.3	The prompt template of Development Artifact.	134
6.4	The prompt template of Comprehensive Analysis and Vulnerability Fix Detection.	138

List of Tables

3.1	Comparison of various vulnerability detection approaches.	25
3.2	Overview of the studied datasets.	29
3.3	Parameter setting for studied DLVDs.	34
3.4	Average rank of each data sampling approach in terms of different evaluation metrics. For each metric, we highlight the records that have a smaller rank than NoSampling with green, and darker color indicates better.	35
3.5	The results of each data sampling approach in terms of different evaluation metrics, including NoSampling. To indicate which sampling approach is more effective than NoSampling in each metric, the records that have a larger value than the NoSampling are highlighted in green, and darker color indicates better. The best records for each experimental instance in terms of F1 are boldened.	36
3.6	The hit ratio of various data sampling approaches in the experimental instances <i>LineVul</i> + <i>BigVul</i> and <i>Reveal</i> + <i>BigVul</i>	40
3.7	Number of neighbors before and after RUS-R in the experimental instances related to the Reveal model.	46
4.1	Overview of the studied dataset.	69
4.2	The performance of different VD approaches on BigVul dataset in terms of studied metrics.	75
4.3	The performance of variants with different ensemble methods and routers.	76
4.4	The performance of MoEVD when the router can select experts correctly (Correctly selected) and wrongly (Wrongly selected).	78
4.5	Performance of CodeBERT, MoEVD _{CodeBERT} , and CausalVul _{CodeBERT} on Head and Tail groups.	78
5.1	Statistics of the VFM2021 dataset (cross-project setting) for both Java and Python.	100
5.2	Statistics of the VFM2023 dataset (cross-project setting) for both Java and Python.	102

5.3	Comparison of VFDelta with baselines on the Java split of the VFM_2021 dataset. We highlight the cells with the best performance in bold. . .	108
5.4	Comparison of VFDelta with baselines on the Python split of the VFM_2021 dataset. We highlight the cells with the best performance in bold.	109
5.5	Comparison of VFDelta and studied variants on VFM2021.	111
5.6	Performance of VFDelta _{Star} under different context windows on VFM_2021.	116
5.7	Comparison of VFDelta and baselines on VFM_2023 dataset.	118
6.1	The performance of vulnerability fix detection approaches.	147
6.2	The ablation results. The cells with the lowest performance are marked in bold, indicating the largest contribution from that component. . .	150
6.3	Bad Case Analysis on Qwen2 Models	153

Chapter 1

Introduction

Open-source software (OSS) has become the foundational infrastructure of the modern digital world, powering everything from mobile applications and cloud services to safety-critical industrial control systems. A report by Synopsys¹ reveals the extent of this dependency: 96% of codebases contain OSS code, with each software application incorporating an average of 526 OSS components.

This widespread adoption of OSS, however, presents a fundamental security challenge. The collaborative nature of OSS development results in an overwhelming volume of code changes. Major repositories process over one hundred million commits annually, with each new feature, dependency update, or code refactoring potentially introducing exploitable vulnerabilities. This sheer volume of continuous code changes far exceeds human capacity for manual security review, creating an urgent need for automated vulnerability detection tools. The scale of undetected vulnerabilities is staggering: public datasets such as the National Vulnerability Database (NVD) [3]

¹<https://www.synopsys.com/blogs/software-security/open-source-trends-ossra-report.html>

record tens of thousands of new OSS vulnerabilities annually, representing only those that have been discovered and reported.

The real-world impact of these vulnerabilities can be catastrophic. In December 2021, the Log4Shell vulnerability (CVE-2021-44228) in Apache Log4j [4] demonstrated this severity by enabling unauthenticated remote code execution in millions of Java applications. This single vulnerability forced governments and enterprises worldwide into emergency patch-and-scan cycles, with estimated costs reaching hundreds of millions of dollars. Such incidents underscore the critical consequences of delayed vulnerability remediation, which can expose organizations to data breaches, service disruptions, and regulatory penalties [5; 6]. The combination of OSS's ubiquitous presence, the continuous emergence of vulnerabilities, and the severe consequences of exploitation has created an urgent need for automated OSS vulnerability management.

1.1 OSS Vulnerability Management

OSS Vulnerability Management is a comprehensive pipeline that spans from the initial discovery of vulnerable code to the complete protection of all downstream consumers. This management process involves two distinct types of projects: source OSS projects, where vulnerabilities originate within the codebase itself, and victim projects, which incorporate source OSS projects as external dependencies. Understanding this distinction is crucial as vulnerabilities propagate through the software supply chain from source to victim projects [7]. The OSS Vulnerability Management pipeline consists of five key phases that work together to secure the OSS ecosystem.

Vulnerability detection identifies potentially insecure code in repositories through static analysis, fuzzing, and deep learning models [8; 9; 10]. Vulnerability analysis then confirms exploitability, performs root cause analysis to understand how the defect was introduced, assigns severity scores, and determines disclosure timelines. Following analysis, vulnerability repair is developed, tested, and the patch is distributed to remediate confirmed flaws. Critically, to ensure the safety of projects that rely on external projects, vulnerability-fix detection operates in parallel to monitor upstream the repositories (i.e., source projects) for security-relevant commits, particularly those silent fixes that lack public advisories—this early detection enables downstream maintainers to proactively cherry-pick or backport critical security patch before vulnerabilities are publicly disclosed. Finally, vulnerability exploitability analysis evaluates the real-world impact on specific deployments by mapping advisories to software bills of materials. This thesis focuses on the first and fourth stages of this pipeline: **vulnerability detection (VD)** and **vulnerability-fix detection (VFD)**. These stages are fundamental because vulnerability detection prevents insecure code from entering production systems, while vulnerability-fix detection ensures security patches are rapidly identified and propagated throughout the dependency chain. Together, they form the early warning system that dictates the effectiveness of the entire vulnerability management process.

Vulnerability detection (VD) is the task of classifying whether a function contains vulnerable code. An accurate VD tool can flag vulnerable code before it is merged into the OSS software and can also detect existing vulnerable code by scanning existing repositories, therefore reducing the total volume of exposed vulnerabilities

and cutting remediation costs.

Vulnerability-fix detection (VFD) is the task of classifying whether a commit contains vulnerability fixing. Unlike other stages in the vulnerability management pipeline that primarily serve OSS maintainers, VFD specifically addresses a critical security gap faced by downstream organizations consuming OSS dependencies. Following coordinated vulnerability disclosure in OSS contexts [11; 12], a vulnerability is suggested to be silently fixed without providing CVE identifiers, security advisories, or even mentions of security implications in commit messages. This creates a dangerous window of exposure where upstream maintainers have already patched critical vulnerabilities, but downstream users remain unaware and their projects remain vulnerable because they rely on public disclosure channels that may lag weeks or months behind the actual fix. For enterprise security teams managing hundreds of OSS dependencies, this silent fix problem represents a significant blind spot [13]. Without VFD capabilities, organizations remain exposed to patched vulnerabilities until formal advisories appear. A delay that sophisticated attackers exploit through patch-diffing of public commits. An accurate VFD tool transforms this reactive security posture into proactive defense by continuously monitoring upstream repositories and alerting teams to probable fixes before public disclosure. This early sensing capability can reduce the window of exposure from months to days, enabling organizations to assess and deploy critical patches based on actual code changes rather than waiting for public disclosure.

These two front-line stages are fundamental to OSS vulnerability management, enhancing the effectiveness of all subsequent steps, such as root-cause analysis, ex-

exploitability analysis, and vulnerability repair.

1.2 Challenges

Although vulnerability detection and vulnerability fix detection are crucial to OSS vulnerability management, each task presents unique technical challenges.

Challenge 1 – Severe class imbalance. In a typical OSS code repository fewer than 5% of functions are vulnerable [14], and in commit histories fewer than 1% of commits carry a security fix [15]. This significant imbalance of data makes these detection tasks like searching for a needle in a haystack. And such imbalance will lead to high false-positive rates for VD and VFD tools, overwhelming users with false alerts and undermining their trust in these tools [16].

Challenge 2 – Long-tailed vulnerability types. Rare CWE categories receive little training data. When a vendor integrates OSS into a safety-critical product, the categories that matter most are often precisely those “edge-case” vulnerabilities (e.g., CWE-284 improper access control) that today’s VD tool miss. This often results in existing tools being effective only on a specific but common set of types, which limits the tool’s true usability in real-world scenarios where rare vulnerabilities are critical.

Challenge 3 – Subtle security code changes. Many vulnerability fixes consist of minimal code modifications that are easily overlooked by automated detection tools [17]. A single-line boundary check, an added null pointer validation, or a modified comparison operator can be the difference between a critical security vulnerability and safe code. These subtle changes often lack distinctive syntactic patterns that traditional detection approaches rely upon. For example, the Heartbleed vulnerability

fix [18] involved changing few lines of code to add proper bounds checking, yet this minimal change addressed one of the most severe vulnerabilities in internet history. Current VFD tools struggle to distinguish such security related modifications from routine code maintenance [17; 13; 19], as both may involve similarly small syntactic changes but with vastly different security implications.

Challenge 4 – Tangled commits and lack of explanations. Modern software development practices often result in commits that combine multiple concerns: feature additions, refactoring, dependency updates, and security fixes may all appear within a single commit containing hundreds or thousands of changed lines [13]. Existing VFD tools fail to identify vulnerability fixes buried within such tangled commits because their detection mechanisms become overwhelmed by the noise of unrelated changes [17]. A critical security patch comprising just a few lines can be completely missed when surrounded by extensive refactoring or feature development code. This detection failure is compounded by another limitation: even when VFD tools do identify potential vulnerability fixes in simpler commits, they provide only opaque binary predictions without explanation. Security analysts receive no insight into which specific code changes are security-relevant or why they warrant attention. For downstream security teams managing dozens of dependencies, this dual challenge means that many critical fixes in tangled commits go undetected entirely, while those fixes that are detected in cleaner commits still require extensive manual analysis to understand their security implications and relevance to specific deployment contexts. The combination of poor detection accuracy on complex commits and absent explanations for any detections severely limits the practical utility of current VFD tools in

real-world software ecosystems where tangled commits are the norm rather than the exception.

1.3 Thesis Objectives

This thesis aims to improve the effectiveness and efficiency of vulnerability detection (VD) and vulnerability fix detection (VFD) models. The specific objectives are:

- Objective 1: Enhance the robustness and effectiveness of vulnerability detection.
 - Quantify and investigate data sampling’s effectiveness in mitigating the impact of class imbalance on deep learning vulnerability detection models.
 - Design scalable model architectures that maintain high accuracy and efficiency across long-tailed CWE categories.
- Objective 2: Improve the effectiveness and usability of vulnerability-fix detection models.
 - Design fine-grained code change representations that can identify subtle security-critical code change within commits.
 - Leverage large language models to detect vulnerability fixes within tangled commits and generate actionable explanations by integrating code changes with development artifacts and historical vulnerability knowledge.

Research Study 1 – Quantify and investigate data sampling’s effectiveness in mitigating the impact of class imbalance on deep learning vulner-

ability detection models (Chapter 3)

Motivation: Vulnerability detection datasets exhibit extreme imbalance: vulnerable functions are typically outnumbered by benign functions at ratios exceeding 1:16, and prior work reported that state-of-the-art deep learning-based vulnerability detection (DLVD) models lose up to 73% of F1 due to data imbalance. Data sampling is effective in alleviating data imbalance for machine learning models and has been demonstrated in various software engineering tasks, yet its impact on DLVD models had never been quantified. Therefore, a systematic and extensive study is necessary to assess the impact of data sampling strategies on addressing Challenge 1 of DLVD models.

Findings: Across 96 groups of controlled experiments (4 sampling algorithms $\times$ 2 sampling strategies $\times$ 4 DLVD models $\times$ 3 benchmark datasets) we obtained four decisive results. First, Random oversampling on raw code (ROS_R) is consistently the most effective technique, raising F1 over the no-sampling baseline in every setting. Second, sampling on raw data outperforms sampling in latent space; projecting into embeddings before balancing discards useful structural information. Third, One-Sided Selection (OSS) fails to alleviate imbalance and can even degrade performance when the underlying detector already struggles, while Random undersampling (RUS_R) yields the largest recall gains but at the cost of precision, confirming its value when maximizing true positives. Fourth, we find that ROS_R not only boosts metrics but improves explanation fidelity: interpretable AI analyses (LIME, GNNExplainer) show that models trained with ROS_R more often base their decisions on the truly vulnerable statements.

Research Study 2 –Design scalable model architectures that maintain high accuracy and efficiency across long-tailed CWE categories (Chapter 4)

Motivation: Existing state-of-the-art DLVD tools adopt a "one-for-all" paradigm: a single network is expected to recognize all Common Weakness Enumeration (CWE) patterns. This approach fails to capture the distinct patterns of different vulnerability types, resulting in suboptimal performance, particularly for long-tailed vulnerabilities that are often underrepresented in training datasets yet critical for safety-critical deployments. The Mixture-of-Experts (MoE) design offers an architectural alternative: routing each input to specialized experts that focus on coherent sub-spaces might mitigate this problem. Prior to this work, however, MoE had never been instantiated for vulnerability detection. Therefore, this motivated us to conduct a study to discover the effectiveness of MoE design on DLVD.

Findings: We proposed MoEVD, which splits the detection problem by top-level CWE type and couples a lightweight router with twelve CWE-specialized experts. On the BigVul benchmark it delivers four major results. First, MoEVD attains an F1 of 0.44—12.8% higher than the strongest of six competitive baselines—and boosts AUC, precision, and recall, improving overall effectiveness. Second, MoEVD raises recall by 9% to 77.8% across almost every CWE, and F1 on long-tailed categories climbs by at least 7.3%. MoEVD boosts DLVD's robustness on long-tailed CWE types.

Research Study 3 – Design fine-grained code change representations that can identify subtle security-critical code change within commits (Chapter 5) **Motivation:** Silent vulnerability fixes often involve subtle code changes, such

as inserting a boundary check on a single line, adding a null pointer validation, or adjusting a comparison operator that current detection models overlook. Current approaches concatenate removed and added code, either as raw text or latent vectors, and their training involves multi-stage pipelines in which encoders and commit-level classifiers are trained separately. Such designs (i) blur the interaction between before and after code, (ii) miss subtle security-critical changes, and (iii) demand costly retraining whenever new data arrive. A lightweight representation that captures fine-grained code change semantics and supports joint end-to-end training was needed to detect these subtle but critical security modifications.

Findings: We proposed VFDelta, which embeds code before and after changes using independent models with surrounding code as context. By performing element-wise subtraction on these embeddings, VFDelta can capture fine-grained changes. Our experiments demonstrate the following findings. First, VFDelta improves overall effectiveness. On the VFM-2021 dataset, VFDelta improves F1 from 0.186 to 0.330 for Java (+77.4%) and from 0.085 to 0.194 for Python (+128.2%). Second, we enhanced the vulnerability fix detection dataset to follow temporal split (VFM-2023), and VFDelta still improves F1 by 64.5% over the best baseline, confirming VFDelta’s robustness in real-world scenarios. Third, we find that our Delta design enables the model to capture nuanced code changes, thereby reducing false positives and raising precision from 0.17 to 0.28 without harming recall.

Research Study 4 – Leverage large language models to detect vulnerability fixes within tangled commits and generate actionable explanations by integrating code changes with development artifacts and historical vul-

nerability knowledge (Chapter 6)

Motivation: State-of-the-art vulnerability fix detection models fail when facing tangled commits that combine security fixes with feature additions, refactoring, and dependency updates. These models analyze only the code diff while neglecting valuable context from issue reports, pull-request discussions, and accumulated security history. Furthermore, they provide opaque binary predictions without explanation, forcing downstream security teams to manually analyze hundreds of changed lines to understand security implications. This limitation is particularly problematic for organizations managing dozens of dependencies, where each flagged commit requires significant manual effort to verify its relevance. Large language models (LLMs) now exhibit strong code-reasoning ability and can process multiple information sources simultaneously, but their potential for vulnerability fix detection had not been explored. LLM4VFD was conceived to fill this gap by coupling chain-of-thought prompting with in-context learning and retrieval-augmented generation. The approach weaves together (i) a Code-Change Intention summary, (ii) Development Artifact context, and (iii) similar Historical Vulnerabilities, then leverages LLMs to deliver both accurate detection in tangled commits and actionable explanations that help security teams understand precisely which changes are security-critical and why they matter for their specific deployment context.

Findings: Using six open-weights models (7B–236B parameters) and the new post-2023 BigVulFixes corpus (1,689 fixes; 26,468 non-fixes), the study reports three key results. First, LLM4VFD outperforms the best-performing PLM-based approach VulCurator by 68.1%–145.4% in terms of F1-score when using different LLMs as

the base model. Second, LLM4VFD demonstrates performance gains ranging from 12.7%–105.6% over its vanilla variant, with smaller models generally benefiting more compared to their larger counterparts. Third, our user study involving security experts shows that in 80% of cases, LLM4VFD’s analysis helps security experts understand the code change and improve efficiency in identifying whether the change is a vulnerability fix commit.

Thesis Statement

Vulnerability detection in open-source software is inherently a data problem rather than a model problem: the extreme imbalance, heterogeneous patterns, subtle code change, and contextual complexity of vulnerability data fundamentally limit traditional approaches. This thesis proves that only through systematic data-centric solutions that address each of these characteristics can automated tools achieve the detection accuracy and interpretability necessary for practical deployment in software security workflows.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 contains a literature review for this thesis. Chapter 3 quantifies the impact of class imbalance on deep learning vulnerability detection and demonstrates how data sampling can mitigate the problem. Chapter 4 introduces MoEVD, a Mixture-of-Experts architecture for vulnerability detection that combines a CWE router with specialist CWE-type vulnerability experts. Chapter 5 presents VFDelta, a lightweight framework for iden-

tifying silent vulnerability-fix commits. Chapter 6 proposes LLM4VFD, the first large language model approach to vulnerability fix detection. Finally, Chapter 7 concludes the thesis and discusses directions for future research.

Chapter 2

Literature Review

2.1 Machine learning-based Vulnerability detection

Early machine-learning approaches to vulnerability detection relied on *human-defined features*. Typical pipelines extracted surface cues such as the “frequency of occurrence of specific terms” via text mining and then trained traditional classifiers to predict vulnerable code segments [20; 21]. With the rapid development of deep learning, researchers shifted to representation learning, allowing the models themselves to discover salient patterns. Token-based methods treat source code as a sequence of tokens and embed it with text encoders such as GloVe or CodeBERT; representatives include VulDeePecker and SySeVR, which focus on code “slices” around security-sensitive operations [22; 23], and LineVul, which feeds the entire function into CodeBERT [24]. A complementary line of work constructs graphs that make syntactic and semantic dependencies explicit. Design and Reveal, for instance, build code property graphs to capture program semantics [25; 26], while IVDetect and LineVD anchor

on vulnerable statements within program-dependency graphs [27; 28]. Several works fine-tune or further pre-train general-purpose encoders for security tasks. Vulberta pre-trains a RoBERTa model on 2 TB of C/C++ and achieves state-of-the-art F1 on both binary and multi-class benchmarks [29]. VulD-Transformer slices functions around API calls and feeds the slices into a vanilla Transformer [30], yielding up to 7% F1 improvement over Devign on the Big-Vul dataset.

2.2 Class-Imbalance Mitigation via Data Sampling

Severe class imbalance—fewer than five percent of functions are vulnerable and less than one percent of commits repair vulnerabilities—has motivated extensive experimentation with data-sampling techniques across defect prediction, bug classification, software-quality prediction, and change prediction [31; 32; 33; 34]. Most studies conclude that sampling *helps*. Yedida and Menzies, for example, showed that random oversampling (e.g. “fuzz sampling”) can significantly improves deep defect predictors on majority-class datasets [35], a finding echoed by our own DLVD experiments. Zheng et al. compared SMOTE, ADASYN, and ROSE when classifying security bug reports and observed that ROSE combined with a random-forest classifier performs best [36]. Kamei et al. reported that over- and undersampling have comparable effectiveness for fault-proneness prediction [37], whereas our results favour oversampling.

2.3 Mixture-of-Experts Frameworks in Software-Engineering Task

Mixture-of-Experts (MoE) is a classic ensemble learning architecture that originated in the early 1990s, where multiple specialized models (“experts”) are combined via a learned gating network which routes each input to the most appropriate expert. The foundational work of Jacobs et al. [38] first introduced this divide-and-conquer approach, and Jordan and Jacobs [39] extended it into a hierarchical probabilistic MoE framework, demonstrating that training separate expert networks under a gating mechanism could improve accuracy on complex tasks.

Mixture-of-Experts (MoE) design is still relatively new in the software-engineering domain and so far has been applied chiefly to defect prediction [40; 41]. ME-SFP combines decision-tree and multilayer-perceptron experts behind a Gaussian-mixture gating network [40]. Shankar and Santosh distinguished a *mixture of implicit experts* (MloE), which partitions the input space on-the-fly according to an error function, from a *mixture of explicit experts* (MEoE), which assigns experts after an upfront clustering step [41].

2.4 Approaches to Vulnerability-Fix Detection

General commit-level vulnerability-fix detection. Commit-level vulnerability-fix detection began with Zhou and Sharma’s word2vec classifier that “leveraged the latent information represented in commit messages” [42]. Later systems enriched this textual signal with additional artefacts or with learned code-change embeddings. Sa-

betta and Bezzi combined commit messages and code diffs via handcrafted features and an SVM [43], whereas HERMES fused commit messages with linked issue discussions [44]. VulFixMiner, CoLeFunDa, and MIDAS all embed code changes using variants of CodeBERT: VulFixMiner fine-tunes the PLM on extracted diff, CoLeFunDa adds contrastive pre-training to learn function-change encoders, and MIDAS concatenates embeddings from commit-, file-, hunk-, and line-level views [13; 19; 17]. Graph-based alternatives such as VF-Finder’s before/after AST representation [45] and SPAIN’s binary-level approach [46] further diversify the landscape.

Silent vulnerability-fix detection. Coordinated-disclosure guidelines recommend maintainers to “fix silently” so that attackers cannot exploit the window between patch release and advisory publication. As a result, commit messages may omit any hint of the security relevance, forcing detectors to rely almost exclusively on code-level signals. Recent work tackles this challenge by representing the change itself more directly. VulFixMiner and CoLeFunDa fine-tune PLMs on *pairs* of removed and added code [13; 19], MIDAS [17] captures complementary granularities, and VF-Finder builds parallel AST graphs.

2.5 Large Language Models for Software Engineering Task

Large language models (LLM) have recently entered security research. Chan et al., Thapa et al., and Du et al. studied LLMs on source-level vulnerability detection [47; 48; 49], while Cheng et al. traced vulnerable versions across OSS releases [50].

Beyond security, Shen et al. benchmarked LLMs on code understanding [51], Achiam et al. explored code generation [52], and Wang et al. enhanced summarization with CodeT5+ [53]. Recent works have investigate and tried to improve LLM’s ability to fix Github Issues

Prompting strategies such as chain-of-thought (CoT) reasoning [54] and in-context learning (ICL) [55] “consistently boost performance,” yet have rarely been coupled with rich development-artifact context for vulnerability-fix identification.

Chapter 3

Does Data Sampling Improve Deep Learning-based Vulnerability Detection? Yeas! and Nays!

3.1 Introduction

Deep learning-based Vulnerability Detection(DLVD) approaches have attracted more attention from research community since they can learn vulnerability patterns from prior vulnerable code automatically [20; 56; 14; 57; 58]. Especially, recent progress in Deep Learning (DL) has sparked interest in using DL to detect software vulnerabilities automatically. In fact, recent studies have demonstrated very promising results achieving high accuracy in detecting vulnerabilities [14; 57; 58].

The prominent and practical issue for vulnerability detection is data imbalance. The ratio of vulnerable and non-vulnerable cases in real-world projects is extremely

unbalanced. Vulnerable cases are far fewer than non-vulnerable ones.

Prior study [14] observed that the performance of state-of-the-art (SOTA) DL-based vulnerability detection (DLVD) approaches drops precipitously in real world unbalanced data. On average, a 73% drop of F1 across all the models is observed. Such a significant performance drop can disable the practical usage of any DLVD approaches. Various approaches were developed to deal with the problem, such as data sampling, cost sensitive learning, and ensemble methods. Data sampling is a widely-used technique and proved to help solve the data imbalance issues in Software Engineering tasks, such as defect prediction [31; 59], software quality prediction [33], and software change prediction [34]. Despite the prevalent existence of imbalance issue in DLVD, no research that systemically and extensively studies data sampling for data imbalance problem in DLVD has been done. Thus, it is in dire need to understand data sampling for data imbalance problem in DLVD.

In this work, we aim to assess the impact of data sampling on the effectiveness of existing SOTA DLVD approaches and their ability of learning vulnerable patterns. For this goal, we conducted an extensive study on four data sampling approaches (random under/oversampling, SMOTE [60], and OSS [61]), and their impact on four SOTA DLVD approaches: Devign [62], Reveal [14], IVDetect [58], and LineVul [24], using three benchmark datasets. Note that some advanced data sampling approaches could only be applied to the data points that have been projected into latent space (feature space) since they require computation in the latent space, such as the popular one SMOTE [60]. While simple ones, such as random under/oversampling, can be applied both on raw data (without any projection) and on latent space. On one

hand, latent space sampling is cheaper than sampling on raw data, since it saves the resource and time for preprocessing data (e.g., data cleanup and feature extraction) and training models. On the other hand, projection into latent space causes information loss. It is unknown which strategy is better. Therefore, we also investigate those two sampling strategies - sampling on raw data (i.e., raw code without any preprocessing and feature extraction) and sampling on latent space (i.e., representation vectors of code). We formulate our research as the following research questions:

- **RQ1: Does data sampling improve the effectiveness of existing DLVD approaches?**

We applied various data sampling approaches to the state-of-the-art DLVD approaches and compared them with the ones without sampling.

- **RQ2: Does data sampling improve the ability of DLVD for learning the vulnerable patterns?**

We aim to understand if a trained DL model with sampling approaches can make more prediction decision reasoning over real vulnerable statements in functions than the same DL model without sampling approaches. We used interpretable AI techniques (LIME [63] and GNNExplainer [64]) to interpret the predictions made by DLVD approaches and examine whether their decisions are made based on the real vulnerable patterns (i.e., vulnerable statements).

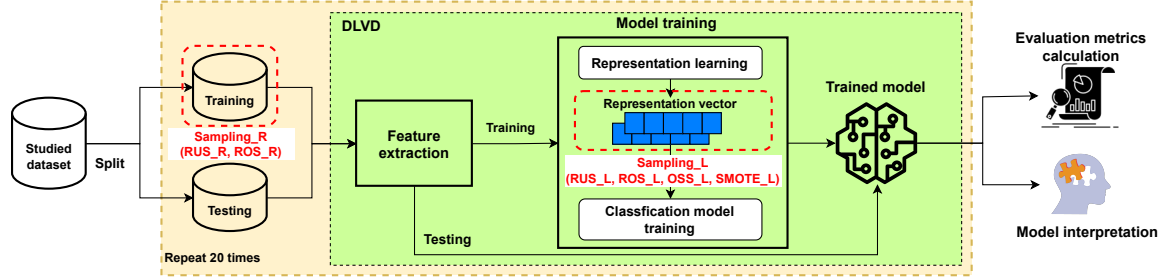


Figure 3.1: Overview of our approach. The deep learning-based vulnerability detection framework is presented in the green rectangle. We mark steps where the proper data sampling approaches could be performed in red round-rectangles.

3.2 Background

In this section, we introduce the background related to deep learning-based vulnerability detection and data sampling.

3.2.1 Overview of deep learning-based vulnerability detection

In general, DLVD approaches consist of three phases: feature extraction, model training, and model deployment. Figure 3.1 presents the general framework of DLVD approaches in a green rectangle. First, in the feature extraction phase, various features from code units that could well capture the semantic and syntactic properties of code to discriminate vulnerable code from non-vulnerable code are extracted. Second, the extracted features are transformed into a real-valued vector, which is a compact representation of code units (i.e., *representation learning*). What features are extracted and how a representation is learned depend on the techniques being selected. Once the representation learning is done, a binary classification model is selected to

perform vulnerability detection. We discuss more details on specific representation learning in Section 3.2.2. In the deployment phase, the same features extracted in the training phase are extracted and the same representation learning techniques are applied to the code unites of target software systems to generate the representation vector.

3.2.2 Existing DL-based vulnerability detection approaches

Generally, DLVD approaches differ in two aspects: the types of features extracted from a code unit (i.e., feature extraction) and the way of transforming the extracted features into a representation vector (i.e., representation learning). Based on this, DLVD approaches could be categorized into two families: token-based or graph-based approaches.

Token-based: In the token-based approach, code is considered as a sequence of tokens and is represented as a vector via text embedding techniques (e.g., Word2Vec [65], GloVe [66]). For instance, LineVul [24] leverages CodeBERT [67] to embed whole sequence of tokens in a function for vulnerability detection. CodeBERT is a transformer-based model pre-trained on a large corpus of source code in various languages and can use the attention mechanism to learn the relationship between tokens. Instead of considering the whole code, approaches such as VulDeePecker [56] and SySeVR [23], extract slices from points of interest in code (e.g., API calls, array indexing, pointer usage, etc.) and use them for vulnerability detection since they assume that different lines of code are not equivalently important for vulnerability detection.

Graph-based: Besides considering the semantic information represented in the se-

quential tokens, graph-based approaches also consider code as graphs and incorporate the information in different syntactic and semantic dependencies using graph neural network (GNN) [68] when generating the representation vectors. Different types of syntactic/semantic graphs, (e.g., abstract syntax tree (AST), program dependency graph (PDG), code property graph (CPG)) can be used. For example, Devign [62] and Reveal [14] leverage code property graph (CPG) [69] to build their graph-based vulnerability detection model. They both use the gated graph neural network (GGNN) to represent the code. IVDetect [58] and LineVD [57] consider the vulnerable statements and capture their surrounding contexts via program dependency graph. IVDetect uses Feature-Attention GCN model(FA-GCN) for graph embedding and LineVD use graph attention network (GAT) model for graph embedding. Differently, IVDetect performs function-level detection while LineVD performs line-level.

In both graph-based and text-based models, once the representation for code units is learned, the vector will be passed into a classifier for final learning. Various classifiers have been investigated in prior studies. For instance, IVDetect [58] and Reveal [14] use Multilayer perceptron (MLP) as the classification model. Table 3.1 summarizes the framework of the above-mentioned DLVD approaches.

3.2.3 Data sampling

In general, data sampling approaches could be categorized into two families: oversampling and undersampling.

Undersampling vs. oversampling Undersampling refers to deleting data points

Table 3.1: Comparison of various vulnerability detection approaches.

Approach	Extracted feature	Represent learning	Classifier	Detection level
LineVul [24]	tokens	CodeBERT	MLP	Function/line
VulDeePecker [56]	tokens + code slices	word2vec	BLSTM	Function
SySeVR [70]	tokens + code slices	word2vec	BGRU	Function
Devign [62]	tokens + CPG	word2vec + GGNN	MLP	Function
Reveal [14]	tokens + CPG	word2vec + GGNN	MLP	Function
IVDetect [58]	tokens + AST+PDG	GloVe + FA-GCN	MLP	Function
LineVD [57]	tokens + PDG	CodeBERT + GAT	MLP	Line

from the majority class of the dataset so that the remaining majority class is approximately equal in number to the minority class [71]. In the context of vulnerability detection, undersampling deletes a certain number of non-vulnerable cases from the training data so that the ratio of vulnerable/non-vulnerable cases approximately equals 1. For instance, random undersampling (RUS) randomly removes cases from the majority class, with or without replacement. Tomek Links locates all cross-class nearest neighbor pairs and remove the corresponding cases in the majority class that

are closest to the minority class [72]. One-sided selection (OSS) combines Tomek Links and the Condensed Nearest Neighbor (CNN) Rule [61]. However, undersampling is very likely to discard useful or important samples [71] and cause information loss.

Oversampling increases the number of instances in the minority class to balance the dataset [71]. For instance, random oversampling (ROS) randomly selects cases from the minority class, with replacement, and adding them to the training dataset. ROS has been proven to be robust [73]. One limitation of ROS is that it does not provide any additional information to the model. Another popular oversampling technique called Synthetic Minority Oversampling Technique (SMOTE) [71] was developed to address this issue by augmenting data for the minority class. It first selects a minority class instance a at random and finds its nearest majority class neighbor b . The synthetic instance is then created by choosing one point on the line connecting a and b in the feature space.

Raw data sampling vs. latent space sampling Some advanced data sampling approaches could only be applied to the data points that have been projected into latent space (or feature space) since they require computation (e.g., calculating distance between points) in the latent space, such as SMOTE. While some simple ones, such as RUS and ROS, can be done on the raw data (raw code units in our case) without any projection into latent space. For simplicity, we refer to the sampling performed on the raw data without any further process as **sampling_R** and the sampling performed on data in the latent space as **sampling_L**. Based on the definition, random oversampling and random undersampling could be performed both on raw

data and latent space, while SMOTE and OSS could only be on latent space. In the context of DLVD, we can perform sampling_R on the raw training data and perform sampling_L on representation vectors after representation learning as indicated in Figure 3.1. On one hand, sampling_L is cheaper than sampling_R, since sampling_L saves the resource and time to preprocess (e.g., data cleanup, feature extraction) and train the backbone model on the extra instances that are added in the minority class. On other hand, projection into latent space cause information loss. We compare them in our RQs. For simplicity, we refer to the same sampling approach applied to raw data and latent space as two approaches in the rest of this research. For instance, we refer to random oversampling on latent space as ROS_L and on raw data as ROS_R.

3.3 Experimental design

In this section, we present our research questions (RQs), our studied dataset, DLVD approaches, data sampling approaches, and our analysis approach for RQs.

3.3.1 Research questions

We aim to answer the following research questions:

- **RQ1:** Does data sampling improve the effectiveness of existing DLVD approaches?
- **RQ2:** Does data sampling improve the ability of DLVD for learning the vulnerable patterns?

Data sampling has been shown its effectiveness in alleviating data imbalance issues in various software engineering tasks, such as defect prediction [31; 59] and quality prediction [33]. Little research was on the impact of data sampling on the performance of SOTA DLVD approaches. **In RQ1**, we investigated whether data sampling could improve the effectiveness of existing SOTA DLVD approaches. Through RQ1, we could provide practitioners insights on the selection of proper data sampling approaches based on their context. Prior research has shown that even if a DL model is able to correctly predict an instance, the prediction is not always based on the real pattern [14; 63]. For instance, in the vulnerability detection context, suppose a DLVD approach correctly predicts a function as vulnerable. However, the statements that the model reasons to make the decision are not the real vulnerable statements in the function. In other words, the model does not really learn the vulnerable patterns for discriminating from vulnerable code to non-vulnerable code. Therefore, **in RQ2**, we aim to understand whether data sampling could help a DLVD approach improve its ability to learn real vulnerable patterns.

3.3.2 Datasets

To answer our research questions, we conducted our study on three popular vulnerability datasets including BigVul [74], Reveal [14], and Devign [62]. BigVul [74] covers the CWEs from 2002 to 2019 that are extracted from over 300 different open source C/C++ projects and contains the trustworthy source code vulnerabilities spanning 91 different vulnerability types. It contains +10K vulnerable methods and +160K non-vulnerable methods. The Reveal dataset [14] contains +12K methods with 9.16%

Table 3.2: Overview of the studied datasets.

Dataset	#Vulnerabilities	#Non-Vulnerabilities	Ratio
Reveal [14]	1,664	10,547	1:9.9
Devign [62]	10,067	12,294	1:1.2
BigVul [74]	10,547	168,752	1:16.3

of the vulnerable ones. The Devign dataset [62] has +22K methods collected from projects FFMpeg and Qemu, in which 45.0% of the methods are vulnerable. The Devign dataset is balanced compared with another two datasets. Those datasets are widely used to evaluate various DLVD approaches in prior studies [62; 14; 58; 57; 24]. Note that only the BigVul dataset has vulnerability-fixing information.

3.3.3 Studied DLVD & Data sampling approaches

DLVD. We chose DLVD approaches based on two criteria. First, DLVD should be representative of the two families of DLVD approaches, token-based and graph-based approaches as discussed in Section 3.2.2. Second, the approaches should be recently developed and leverage deep learning techniques. The goal behind the criterion is to foster the generalizability and applicability of our results to SOTA approaches. We selected Devign [62], Reveal [14], and IVDetect [58] as the representative for graph-based approaches. For the token-based approach, we selected LineVul [24] as the representative since it is SOTA approach leveraging CodeBERT [67], and has been proven to be more effective than other approaches (e.g., Devign, Reveal, and IVDetect). Note that LineVul can be used to perform both function-level and line-

level vulnerability detection. In this study, we used LineVul to perform function-level detection for consistency.

Data sampling. We selected data sampling approaches based on two criteria. First, the selected approach should cover two families of sampling approaches in Section 3.2.3, i.e., oversampling and undersampling. Second, the selected approaches can be applied on both raw data and latent space. Therefore, we selected random undersampling (RUS) and random oversampling (ROS) as our studied approaches. In addition, we also wish to include advanced approaches besides ROS and RUS. We selected one-side selection (OSS) as the representative for undersampling, and Synthetic Minority Oversampling Technique (SMOTE) as the representative for oversampling. Note that OSS and SMOTE could only be applied on latent space as discussed in Section 3.2.3.

3.3.4 Evaluation metrics

We consider two families of evaluation metrics to assess the effectiveness of DLVD approaches. First, considering the vulnerability detection as a binary classification task, we consider four popular evaluation metrics [75]- recall (short for **R**), precision (short for **P**), F1, and Area Under The Curve (AUC) following prior studies [14; 62; 56; 76; 77]. Second, considering the cost of reviewing potential vulnerable code from a returned list, we used popular evaluation metric for ranking algorithm [78; 58] - Precision@k (short for **P@k**), which is calculated as $\frac{\text{true positive}}{k}$ and k is the size of returned list. We selected k to be 10, 20, 50, and 100.

3.3.5 Approach of RQs

We first introduce certain settings in our study. We consider the assessment of data sampling approaches of one DLVD approach on one dataset as one *experimental instance*. For simplicity, we use the format $\{DLVD\} + \{Dataset\}$ to denote one specific instance in the rest of the sections. Therefore, We have 12 experimental instances, i.e., 4 DLVD * 3 datasets.

For each experimental instance, we have the following three settings of data sampling to apply and compare their impact on the DLVD approach:

NoSampling: We train the DLVD approach on the original training data without applying any data sampling approach and evaluate the DLVD approach on the testing data.

Sampling_R: We apply data sampling approaches (i.e., RUS and ROS) on the raw training data (shown in Figure 3.1 in red round-rectangle) and train the DLVD approach on the sampled training data. We then evaluate the DLVD approach on testing data. We refer to RUS and ROS on raw data as RUS_R and ROS_R, respectively.

Sampling_L: This setting is similar to **Sampling_R**. The only difference is that instead of applying data sampling approaches on raw training data, we apply data sampling (i.e., RUS, ROS, OSS, and SMOTE) on representation vectors of training data (latent space). We refer to RUS, ROS, OSS, and SMOTE on latent space as RUS_L, ROS_L, OSS_L, and SMOTE_L, respectively.

Data splitting for generating training and testing data involves a random process, to alleviate the bias caused by the randomness, we repeated the process 20 times following prior studies [79; 80; 81] and calculated the mean value across the 20 splits

for each evaluation metric as our results. We split the training and testing to 80%/20% following prior studies [14; 15; 82].

Approach of RQ1

In RQ1, our goal is to examine if data sampling could improve the effectiveness of DLVD and if so, which data sampling approaches perform the best. For this purpose, we rank data sampling approaches in each experimental instance based on our studied evaluation metrics (see Section 3.3.4 for details). For example, suppose we apply the studied sampling approaches on the Reveal dataset for training the IVDetect model (i.e., *IVDetect + Reveal*). In this experimental instance, ROS_R performs the best (ranks 1st) in terms of F1, we note that ROS_R ranks 1st in the instance *IVDetect + Reveal* in terms of F1. To get an overall rank for a data sampling approach across the studied experimental instances, we compute an average rank. For instance, suppose ROS_R ranks 1st in 6 instances and ranks 2nd in another 6 instances, then its average rank is 1.5. Therefore, we can compare data sampling approaches based on their rank in terms of different evaluation metrics. A smaller rank indicates more effective. We also investigated which sampling approach performs the best by comparing the number of experimental instances won by each sampling approach.

Approach of RQ2

To understand if the trained model makes correct prediction decision reasoning over real vulnerable statements in a function, we selected to use interpretable AI techniques. The three graph-based models we studied all use GNNs to capture

graph-related properties. Hence, we selected the Reveal model as the representative for graph-based approaches and selected LineVul as the representative for text-based approaches. We selected the BigVul dataset since only this dataset provides us the line information for vulnerable codes, i.e, specific statements that make the code vulnerable. Therefore, we conducted the analysis on two experimental instances *LineVul + BigVul* and *Reveal + BigVul*.

We used Local Interpretable Model-Agnostic Explanations (LIME)[63], which is a widely-used model-agnostic explainable algorithm for explaining deep learning models, to interpret LineVul. LIME returns tokens in a sequence that make important contribution to the prediction. For our case, we compared the results of LIME with the real vulnerable statements in the functions. More specifically, we designed the following experiment for LineVul: we selected the true positive (TP) cases (the vulnerable functions that are predicted correctly) predicted by LineVul in the testing set of BigVul, and analyzed those cases using LIME. In the result of LIME, for each true positive case, we selected the top k most important tokens that drive the model to make the decision and see whether those tokens are inside the vulnerable statements. If at least one of the top k is inside the vulnerable statements, we consider it as a **hit**, and we compare the ratio of **hit** over all TP cases between NoSampling and all sampling approaches. We considered 1, 3, 5, and 10 for k .

We used GNNExplainer [64] to interpret Reveal following prior study [58]. Similarly, we interpreted the true positive (TP) cases predicted by the Reveal model. GNNExplainer provides the importance of each edge in the graph. Reveal uses the code property graph (CPG), and each line of code is a node in the graph, we applied

Table 3.3: Parameter setting for studied DLVDs.

Parameter	IVDetect	Devign	Reveal	LineVul
Epochs	50	50	50	4
Learning_rate	1e-4	1e-4	1e-4	5e-5
GNN_Layer	4	6	6	NaN
graph embedding size	100	200	200	NaN
feature embedding size	100	100	100	768
total parameters	575K	924K	402K	123M

GNNExplainer on the CPG. GNNExplainer does not directly tell us which nodes are important, therefore we consider the nodes connected by the important edges returned by GNNExplainer as the important nodes in the graph. Similar to the experiment of Lime, we selected the top k most important edges and their connected nodes that drive the model to predict a function to be vulnerable and see whether those nodes (lines of codes) are inside the vulnerable statements of the function.

3.3.6 Implementation details

We ran our experiments on Compute Canada Narval HPC[83] with Nvidia A100 GPUs and a Linux server with four Nvidia RTX 3090 GPUs, AMD Ryzen 48-Core CPU with 256 GB Ram. We used the implementations of studied DLVD approaches published in their GitHub repositories. For LIME and GNNExplainer, we used the implementation in Github-Lime and dgl-GNNExplainer [84].

We fine-tuned the DLVD approaches using the recommended parameters in the original papers. The parameter settings are presented in Table 3.3. Noted that we used a learning rate of 5e-5 in the LineVul model because it is a pre-trained model and our training on the model can be considered as fine-tuning the Codebert for a downstream task – DLVD. Thus the learning rate is lower than other models. Our experiments involve 12 experimental instances, and for each experimental instance we have 7 data sampling approaches (NoSampling plus various sampling approaches) to examine. Therefore, we have 84 (7*12) combinations. For each data sampling approach, we ran 20 times for getting reliable results. We totally ran $12 \times 7 \times 20 = 1,680$ experiments, which costs more than 10,200 GPU hours.

3.4 Results

3.4.1 RQ1: Does data sampling improve the effectiveness of existing DLVD approaches?

Table 3.4: Average rank of each data sampling approach in terms of different evaluation metrics. For each metric, we highlight the records that have a smaller rank than NoSampling with green, and darker color indicates better.

Sampling	AUC	P	R	F1	P@10	P@20	P@50	P@100
NoSampling	3.9	3.3	6.8	6.6	4.0	3.8	4.2	4.3
RUS.R	3.0	4.6	2.3	3.3	4.5	4.7	4.4	4.2
ROS.R	3.4	2.9	4.3	2.8	2.4	2.5	2.5	2.5
SMOTE.L	5.3	5.0	3.3	3.3	4.3	3.8	3.8	3.9
OSS.L	4.3	3.3	4.8	4.5	4.8	4.7	4.2	4.8
RUS.L	3.8	5.1	2.5	3.8	4.2	4.8	4.8	4.5
ROS.L	4.3	3.9	4.0	3.8	3.8	3.8	4.0	3.9

Table 3.5: The results of each data sampling approach in terms of different evaluation metrics, including NoSampling. To indicate which sampling approach is more effective than NoSampling in each metric, the records that have a larger value than the NoSampling are highlighted in green, and darker color indicates better. The best records for each experimental instance in terms of F1 are boldened.

	DLVD	IVDetect								Devign							
	Sampling	AUC	P	R	F1	P@10	P@20	P@50	P@100	AUC	P	R	F1	P@10	P@20	P@50	P@100
Devign	NoSampling	0.573	0.534	0.494	0.512	0.7	0.65	0.64	0.67	0.569	0.504	0.479	0.49	0.788	0.763	0.678	0.648
	RUS_R	0.596	0.524	0.523	0.547	0.6	0.6	0.62	0.65	0.564	0.501	0.532	0.514	0.62	0.59	0.648	0.608
	ROS_R	0.598	0.528	0.544	0.535	0.7	0.75	0.66	0.68	0.567	0.505	0.488	0.495	0.7	0.7	0.678	0.64
	SMOTE_L	0.555	0.483	0.858	0.61	0.667	0.631	0.626	0.618	0.583	0.511	0.557	0.527	0.74	0.73	0.68	0.686
	OSS_L	0.608	0.535	0.549	0.542	0.629	0.621	0.614	0.611	0.584	0.51	0.569	0.535	0.64	0.67	0.684	0.67
	RUS_L	0.608	0.537	0.538	0.537	0.676	0.645	0.615	0.62	0.584	0.512	0.534	0.517	0.74	0.66	0.652	0.656
	ROS_L	0.607	0.537	0.533	0.535	0.695	0.645	0.626	0.619	0.584	0.509	0.566	0.531	0.68	0.7	0.668	0.68
Reveal	NoSampling	0.402	0.008	1E-04	3E-04	0	0	0	0	0.711	0.274	0.188	0.188	0.838	0.775	0.625	0.541
	RUS_R	0.718	0.196	0.688	0.305	0.4	0.3	0.3	0.31	0.701	0.178	0.656	0.278	0.4	0.3	0.325	0.33
	ROS_R	0.71	0.286	0.492	0.46	0.6	0.45	0.44	0.43	0.697	0.213	0.3	0.248	0.867	0.839	0.813	0.786
	SMOTE_L	0.624	0.142	0.593	0.228	0.265	0.303	0.242	0.22	0.695	0.248	0.397	0.304	0.914	0.886	0.814	0.75
	OSS_L	0.38	0	0	0	0	0	0	0	0.706	0.273	0.287	0.279	0.943	0.907	0.84	0.741
	RUS_L	0.619	0.134	0.661	0.22	0.26	0.305	0.245	0.213	0.708	0.25	0.459	0.322	0.886	0.836	0.789	0.734
	ROS_L	0.622	0.139	0.626	0.226	0.275	0.303	0.237	0.22	0.697	0.25	0.414	0.31	0.943	0.921	0.834	0.733
BigVul	NoSampling	0.461	0	0	0	0	0	0	0	0.701	0.326	0.201	0.247	0.68	0.72	0.668	0.586
	RUS_R	0.679	0.111	0.628	0.188	0.2	0.2	0.14	0.12	0.687	0.1	0.682	0.174	0.2	0.18	0.156	0.146
	ROS_R	0.77	0.244	0.622	0.347	0.2	0.1	0.14	0.15	0.703	0.295	0.253	0.271	0.48	0.42	0.472	0.488
	SMOTE_L	0.568	0.08	0.507	0.137	0.06	0.088	0.08	0.097	0.717	0.259	0.284	0.269	0.614	0.65	0.631	0.557
	OSS_L	0.447	0.1	5E-05	1E-04	0.005	0.003	0.001	5E-04	0.712	0.316	0.221	0.253	0.714	0.686	0.611	0.577
	RUS_L	0.561	0.075	0.584	0.131	0.065	0.088	0.086	0.094	0.711	0.221	0.354	0.268	0.714	0.714	0.646	0.571
	ROS_L	0.566	0.078	0.541	0.136	0.07	0.088	0.084	0.097	0.723	0.232	0.333	0.27	0.7	0.714	0.654	0.6
Devign	DLVD	Reveal								LineVul							
	Sampling	AUC	P	R	F1	P@10	P@20	P@50	P@100	AUC	P	R	F1	P@10	P@20	P@50	P@100
	NoSampling	0.555	0.512	0.42	0.46	0.525	0.5	0.438	0.444	0.71	0.616	0.537	0.573	0.942	0.942	0.934	0.918
	RUS_R	0.553	0.499	0.545	0.519	0.47	0.47	0.446	0.46	0.704	0.595	0.59	0.592	0.989	0.981	0.973	0.967
	ROS_R	0.553	0.508	0.492	0.497	0.49	0.445	0.416	0.418	0.705	0.606	0.557	0.58	0.968	0.971	0.973	0.965
	SMOTE_L	0.54	0.495	0.511	0.502	0.42	0.458	0.474	0.467	0.661	0.563	0.618	0.586	0.95	0.95	0.937	0.919
	OSS_L	0.54	0.494	0.547	0.519	0.44	0.45	0.468	0.458	0.663	0.564	0.587	0.573	0.95	0.945	0.937	0.912
Reveal	NoSampling	0.583	0.282	0.119	0.159	0.625	0.556	0.56	0.484	0.775	0.4	0.249	0.299	0.73	0.72	0.683	0.625
	RUS_R	0.674	0.178	0.594	0.273	0.425	0.45	0.415	0.395	0.841	0.74	0.284	0.41	0.512	0.538	0.508	0.491
	ROS_R	0.636	0.23	0.414	0.292	0.61	0.61	0.588	0.576	0.811	0.443	0.506	0.472	0.965	0.963	0.948	0.907
	SMOTE_L	0.621	0.192	0.311	0.235	0.415	0.343	0.237	0.215	0.716	0.301	0.51	0.355	0.735	0.74	0.687	0.625
	OSS_L	0.636	0.245	0.211	0.219	0.34	0.318	0.255	0.244	0.72	0.421	0.248	0.295	0.72	0.742	0.687	0.626
	RUS_L	0.676	0.187	0.557	0.277	0.31	0.325	0.34	0.3	0.724	0.263	0.599	0.354	0.755	0.715	0.679	0.646
	ROS_L	0.63	0.197	0.311	0.24	0.29	0.265	0.239	0.203	0.724	0.374	0.352	0.357	0.745	0.728	0.701	0.647
BigVul	NoSampling	0.679	0.387	0.109	0.168	0.72	0.7	0.648	0.578	0.554	0.07	0.002	0.005	0.07	0.07	0.063	0.046
	RUS_R	0.684	0.107	0.617	0.182	0.28	0.23	0.248	0.242	0.825	0.164	0.722	0.266	0.59	0.618	0.627	0.592
	ROS_R	0.661	0.232	0.304	0.261	0.457	0.5	0.477	0.444	0.745	0.455	0.367	0.406	0.947	0.958	0.943	0.904
	SMOTE_L	0.617	0.19	0.276	0.224	0.34	0.375	0.394	0.398	0.532	0.062	0.587	0.112	0.175	0.145	0.133	0.127
	OSS_L	0.632	0.248	0.224	0.235	0.41	0.43	0.437	0.44	0.544	0.064	0.003	0.005	0.075	0.068	0.056	0.043
	RUS_L	0.664	0.114	0.564	0.189	0.29	0.29	0.273	0.279	0.542	0.07	0.559	0.11	0.175	0.15	0.14	0.142
	ROS_L	0.62	0.193	0.285	0.23	0.335	0.323	0.338	0.353	0.539	0.075	0.501	0.121	0.18	0.163	0.142	0.147

Table 3.5 presents the detailed results of various data sampling approaches in the 12 experimental instances in terms of various evaluation metrics. We observe that almost all data sampling techniques improve the effectiveness of all experimental instances in terms of F1, which is a major evaluation metric for the binary classification task. We observe the same results in Table 3.4 which presents the average rank of each studied data sampling approach including the NoSampling across the 12 experimental instances, all studied data sampling approaches have a smaller rank than NoSampling in terms of F1. Similar results are observed when looking at P@k on imbalanced datasets (Reveal and BigVul). For instance, in terms of P@10, at least one data sampling approach outperforms NoSampling in all experimental instances except the instances related to the Reveal model.

Finding1: In general, data sampling improves the effectiveness of all studied models, especially on imbalanced datasets.

In Table 3.4, only ROS_R outperforms NoSampling in terms of all evaluation metrics. In Table 3.5, we observe that ROS_R wins (ranks 1st) 7 out of the 12 instances in terms of F1. If only considering the imbalanced datasets, ROS_R wins 7 out of 8 instances in terms of F1, except *Devign + Reveal*. ROS_R improves the NoSampling in terms of F1 across all experimental instances. Although ROS_R is not the best in *Devign + Reveal*, it still improves NoSampling by 31.9% from 0.188 to 0.248. When considering the improvement of ROS_R over NoSampling in the 8 experimental instances on imbalanced datasets, ROS improves the NoSampling at least by 31.9%. In terms of P@10, ROS_R outperforms NoSampling in 5 out of the 8 instances on imbalanced datasets. Surprisingly, ROS_R performs much better than

SMOTE_L in all evaluation metrics, which is an advanced sampling approach and is reported at least comparable to random oversampling reported in prior studies [85; 86; 87].

Finding2: Surprisingly, the simple approach ROS_R outperforms all other studied sampling approaches (including the advanced one SMOTE_L) and it significantly improves the effectiveness of DLVD approaches in terms of all studied metrics.

When comparing oversampling with undersampling, we observe that ROS_R outperforms RUS_R in all metrics except AUC and R in Table 3.4. Similarly, ROS_L outperforms RUS_L in all P@k and R. Such a finding is expected since undersampling deleting data points from the dataset, which causes information loss while oversampling keeps all the original data. Interestingly, when looking at the improvement in recall, we observe that RUS_R achieves the best performance in all experimental instances on imbalanced datasets (Reveal and BigVul), except for *LineVul + Reveal*, in which RUS_L performs the best. Although in some instances, RUS_R is not the best, RUS_R still improves the recall across all the experimental instances at least with an improvement of 10% over NoSampling. Our finding is compatible with prior studies [88] that undersampling achieves the largest improvement of recall. We investigate the reason behind this in Section 3.5.2.

Finding3: In general, oversampling outperforms undersampling, while random undersampling performs the best in improving recall on imbalanced datasets.

When comparing sampling_L with sampling_R, we observe that ROS_R outperforms ROS_L in terms of all metrics in Table 3.4. For RUS, RUS_R has a similar rank as RUS_L in terms of all P@k, however, RUS_R outperforms RUS_L in terms of AUC,

P, R, and F1. Therefore, in general, sampling_R outperforms sampling_L, typically for ROS. We speculate the possible reasons in two folds. First, projecting the code into latent space leads to information loss and brings bias compared with the original code. Sampling on the latent space magnifies such bias. Second, since the sampling is applied after the representation learning, in other words, data sampling does not provide help in the representation learning process.

Finding4: In general, sampling_R outperforms sampling_L.

Table 1 shows that for model Reveal and Devign which perform relatively well on the original datasets (e.g., Devign, Reveal, and BigVul), OSS helps improve their performance (e.g., in terms of F1). While OSS does not help DLVD approaches in handling data imbalance for the model IVDetect and LineVu which perform poorly on imbalanced datasets (i.e., Reveal and BigVul) in the NoSampling scenario. For instance, the IVDetect and LineVul perform poorly on imbalanced datasets (Reveal and BigVul), i.e., values of all metrics nearly equal to 0. The reason is that those two models do not have tolerance for the imbalance in training data and they predict all data points as non-vulnerable. Applying OSS_L does not help improve the effectiveness of IVDetect and LineVul. Why can OSS not balance data well? OSS relies on Tomek Links [72] and Condensed Nearest Neighbor (CNN) [89] to decide which non-vulnerable data to delete. However, such deleted non-vulnerable data points have to meet certain criteria. OSS will stop deleting non-vulnerable data if no more data points are found to meet the criteria even the data still remains imbalanced after running OSS. Therefore, OSS does not guarantee to reduce the size of the majority class to the minority class. We investigate the datasets Reveal and BigVul after ap-

plying OSS_L and find that the datasets are still extremely imbalanced. The ratio of vulnerable and non-vulnerable is 1:8.7 for the Reveal dataset, and 1:16 for the BigVul dataset.

Finding5: Surprisingly, OSS does not help alleviate the data imbalance problem in DLVD if the DLVD approaches (i.e., Reveal and BigVul) perform poorly on imbalanced data originally.

3.4.2 RQ2: Does data sampling improve the ability of DLVD for learning the vulnerable patterns?

Table 3.6: The hit ratio of various data sampling approaches in the experimental instances *LineVul + BigVul* and *Reveal + BigVul*.

	<i>LineVul + BigVul</i>				<i>Reveal + BigVul</i>			
	Top1	Top3	Top5	Top10	Top1	Top3	Top5	Top10
NoSampling	0.39	0.55	0.66	0.66	0.55	0.76	0.81	0.90
ROS_R	0.41	0.62	0.72	0.82	0.67	0.83	0.89	0.92
RUS_R	0.32	0.45	0.56	0.71	0.69	0.85	0.91	0.95
ROS_L	0.22	0.39	0.52	0.68	0.65	0.84	0.90	0.95
RUS_L	0.15	0.37	0.50	0.66	0.65	0.86	0.91	0.95
SMOTEL	0.13	0.35	0.47	0.63	0.67	0.84	0.90	0.95
OSS_L	0.32	0.53	0.61	0.72	0.69	0.84	0.90	0.95
ROS_R_2X	0.47	0.67	0.76	0.85	0.67	0.83	0.89	0.93
ROS_R_4X	0.45	0.66	0.75	0.85	0.64	0.86	0.91	0.94

The hit ratio of various data sampling approaches in *LineVul + BigVul* and *Reveal + BigVul* are shown in Table3.6. In *Reveal + BigVul*, all studied data sampling approaches improve hit ratio compared with NoSampling. In *LineVul + BigVul*, only ROS_R improves the hit ratio. A possible explanation is that LineVul does not learn well on BigVul on imbalanced data, and only ROS_R helps handle data

imbalance (see the effectiveness of LineVul on BigVul in Table 3.5). Nevertheless, ROS_R improves the NoSampling on both two instances. For instance, after ROS_R, the hit ratio in *LineVul+BigVul* increases from 0.39, 0.55, 0.66, and 0.66 to 0.41 0.62, 0.72, and 0.82 when k equals to 1, 3, 5, and 10 with improvements of 5.6%, 12.7%, 9.1%, and 24.2%, respectively. Similarly, ROS_R improves the hit ratio in *Reveal + BigVul* by 21.8%, 8.6%, 9.4%, and 2.4% when k equals to 1, 3, 5, and 10, respectively. Such results suggest that ROS_R helps LineVul and Reveal learn better about the characteristics of vulnerable patterns. For example, Figure 3.2 presents the LIME result of a vulnerable function before and after ROS_R. The function has a Use-after-free vulnerability at lines marked with green rectangle [90]. Before ROS_R, LineVul predicts it as non-vulnerable according to the statement highlighted in blue, while after ROS_R, LineVul is able to predict correctly and really based on the vulnerable lines highlighted in orange. The observation probably indicates that **deduplicated data points in the raw training dataset are not necessarily noisy to the model, they could provide a positive influence on the model for learning the real patterns for detecting vulnerabilities when added properly**. To validate this assumption, we increase the ratio of vulnerable and non-vulnerable cases ($Ratio_{Vul}$) from 1:1 to 2:1, and 4:1 using ROS_R and compare their hit ratio. The results are presented in Table 3.6 for different $Ratio_{Vul}$, i.e., ROS_R for 1:1, ROS_R_2X for 2:1 and ROS_R_4X for 4:1. We can see that the hit ratio for top 1 increases from 0.41 to 0.47 when $Ratio_{Vul}$ increases from 1 to 2, and slightly drops when the $Ratio_{Vul}$ reaches 4 (0.45) for LineVul. Meanwhile the effectiveness of LineVul and Reveal increases after increasing the repetition as well, i.e., F1 values for LineVul are

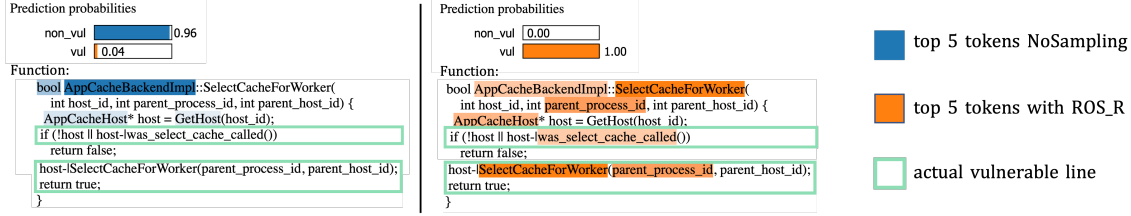


Figure 3.2: The explanation of LIME on a vulnerable function, which is predicted wrongly as non-vulnerable before ROS (left) and predicted correctly after ROS_R (right).

0.406, 0.42, and 0.42 after applying ROS_R, ROS_R.2X, and ROS_R.4X, respectively.

We observe a similar phenomenon for the Reveal.

We further perform additional analysis to understand the impact of ROS_R with different multiplications (i.e., 2x and 4x) of vulnerable cases on model LineVul for dataset BigVul (LineVul+BigVul) in terms of effectiveness. The results show that the F1 scores are 0.406 (ROS_R) to 0.420 (ROS_R.2x), and 0.418 (ROS_R.4x), respectively. We observe the same trend for AUC. The Top1 hit ratios arrive the best value at 2x and slightly drop afterward. Too many multiplications of vulnerable cases could bias the classification and decrease the effectiveness of the model. The results suggest that ROS_R.2x works best for the LineVul+BigVul instance. The result indicates that increasing the size of repeated vulnerable cases properly could both improve the effectiveness and its ability of learning vulnerable patterns.

Finding6: ROS_R improves the ability of DLVD approaches for learning real vulnerable patterns.

Although our results show that ROS_R can improve the ability of DLDV for capturing vulnerable patterns and predicting based on them. It still has potential for future improvement even after applying data sampling. If we look at the top1

results, the hit ratios are 0.39 and 0.55 for *LineVul* + *BigVul* and *Reveal* + *BigVul*, respectively. Even after applying RUS_R, hit ratios are improved to 0.42 and 0.67. There are still a significant number of cases in which the decision is not really made based on the vulnerable statements. 58% and 33% of the vulnerable functions that are predicted as vulnerable are not based on their real vulnerable statements for LineVul and Reveal.

Finding7: In a significant portion of cases (at least 33%), DVLD approach cannot reason their prediction over real vulnerable statements. There is still room for improving the ability of learning real vulnerable patterns for DLVD approaches.

3.5 Discussion

3.5.1 Why does ROS_R perform the best among all studied approaches?

In RQ1, we observe that ROS_R performs the best among all studied approaches in all studied metrics across experimental instances, typically for imbalanced datasets. Why is ROS_R the king? To some extent, the results in RQ2 provide insights, i.e., ROS_R improves the ability of DLVD approaches for learning real vulnerable patterns. We speculate ROS_R performs the best due to its natural compared with other approaches. First, compared with undersampling (i.e., RUS_R), instead of deleting non-vulnerable code from the training data which causes information loss, ROS_R, as an oversampling approach does not have this issue. Second, compared with other sampling approaches on latent space, ROS_R performs sampling on raw data. We speculate that sampling_R separates vulnerable cases from non-vulnerable cases bet-

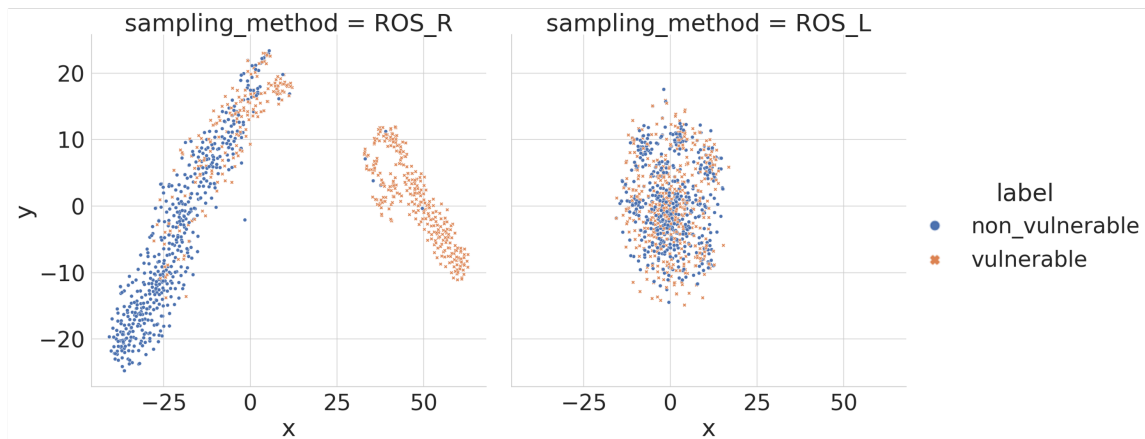


Figure 3.3: The distribution of data points in the experiment instance *LineVul* + *BigVul* after applying ROS_R and ROS_L.

ter than sampling_L. For example, to illustrate the effect of ROS_L and ROS_R on the data distribution in latent space, we used the t-distributed stochastic neighbor embedding (t-SNE) [91] to visualize the distribution of the data points when using ROS_L and using ROS_R in *LineVul* + *BigVul*, in which the F1 value for ROS_R and ROS_L are 0.406 and 0.121, respectively. Figure 3.3 presents the distribution of data points after applying ROS_R and ROS_L. Vulnerable and non-vulnerable data is better separated when applying ROS_R than applying ROS_L.

3.5.2 Why does random undersampling boost recall?

Recall measures how well a DLVD approach could identify all vulnerable code. Prior studies report that there is a significant portion of similar code in code corpus [92; 93; 94]. Therefore, we assume there probably exists a significant number of similar non-vulnerable cases surrounding vulnerable cases and introduce significant noise for models to identify various vulnerable code. Random undersampling reduces

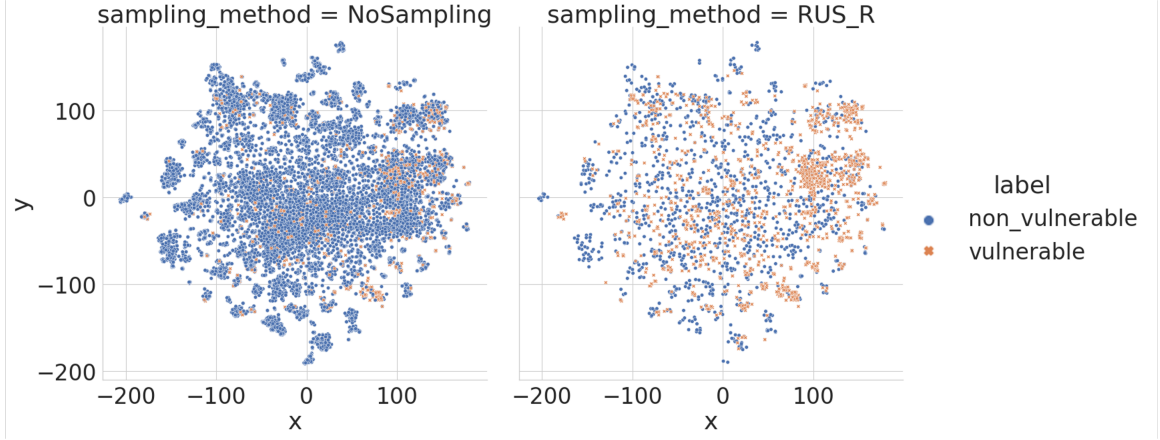


Figure 3.4: The distribution of data points in the Reveal dataset without (NoSampling) and without applying RUS_R.

such noise, therefore boosts recall, although it reduces precision due to information loss.

To validate our assumption, we conduct a case study on the experimental instances for the Reveal model. We applied t-SNE on the representation vectors generated by the Reveal model and visualize the distribution of points before and after RUS_R which performs the best. To quantitatively validate our assumption, we also calculated the mean and medium number of neighbors for each vulnerable point in the generated t-SNE plot. We consider all non-vulnerable points having a Euclidean distance less than 4 to a vulnerable point as its neighbor, given the map size is 400*400. We discuss the threats to validity caused by the threshold selection (i.e., 4) in Section 3.5.4.

Figure 3.4 presents data points distribution for *Reveal + Reveal* (due to the space limitation we do not show the plots for another two datasets). We observe that without applying RUS_R (NoSampling), non-vulnerable points cover the majority of non-vulnerable points, and vulnerable and non-vulnerable are almost mixed together.

Table 3.7: Number of neighbors before and after RUS_R in the experimental instances related to the Reveal model.

Dataset	Devign		Reveal		BigVul	
Sampling	mean	median	mean	median	mean	median
NoSampling	6.68	6	38.11	38	41.43	43
RUS_R	5.43	5	3.60	3	2.60	2

After applying RUS_R, better separation between the vulnerable and non-vulnerable points is observed. Table 3.7 presents the median/mean number of neighbors for each vulnerable point before and after RUS_R. On all studied datasets, the mean/median number is reduced significantly, but the ratio of reduction varies in different datasets. The mean is reduced from 6.68 to 5.43 on balanced data (Devign), while on the imbalanced datasets, the mean is reduced from 38.11 to 3.6 and 41.43 to 2.6 on the Reveal and BigVul datasets with a reduction of more than 10 times. We observe a correlation between the improvement of recall and the size of the reduction. The recall achieves an improvement of 464% and an improvement of 397% on BigVul and Reveal datasets, respectively, while only obtains an improvement of 12.4% on Devign. Such results explain to some extent why the random undersampling can reduce noise for vulnerable code and thus boost recall.

3.5.3 Implications of our findings

We recommend future practitioners to use oversampling over undersampling, `sampling_R` over `sampling_L`, and typically to use `ROS_R` to handle data imbalance issue in DLVD. In RQ1, we observe that generally

oversampling outperforms undersampling, sampling_R outperforms sampling_L. Typically, ROS_R performs the best among all studied sampling approaches. Moreover, we observe that ROS_R can improve the ability of learning real vulnerable patterns in source code for DLVD approaches (both text-based and graph-based models). Therefore, ROS_R is recommended for handling data imbalance issue in DLVD.

We do not recommend practitioners to use OSS_L for imbalanced data in DLVD. In RQ1, we observe that among all studied sampling approaches, OSS_L is the only one that does not help improve the effectiveness of DLVD on imbalanced datasets at all due to its mechanism that cannot guarantee to really balance the target dataset, typically when the ratio between vulnerable and non-vulnerable code is large. Therefore, we do not suggest practitioners to use OSS_L on imbalanced data for training DLVD.

We recommend future practitioners to use RUS_R if they wish to improve the recall. In RQ1, we observe that undersampling approaches, typically RUS_R, can boost the recall for DLVD approaches. In Section 3.5.2, our analysis shows that RUS_R reduces the number of surrounding similar non-vulnerable points for vulnerable points dramatically to reduce noise. In practice, if practitioners aim to improve the recall of their approaches, they probably could consider using RUS_R.

We encourage future research to develop new data augmentation techniques to improve the ability of DLVD approaches for learning real vulnerable patterns. In RQ2, although our results show that ROS_R can improve the ability of DLVD for learning vulnerable patterns. However, there are still a significant number of cases (58% and 33% of the vulnerable functions in *LineVul* + *BigVul* and

Reveal + *BigVul*), in which the decision is not made based on the vulnerable statements. It still has potential for future improvement even after applying data sampling. Our findings shed light for addressing/alleviating this problem. For instance, data augmentation probably is a valuable direction to investigate, as our results show that the simple repetition strategy (e.g., ROS_R) is helpful in RQ2.

3.5.4 Threats to validity

Internal Validity One threat relates to hyperparameter settings when training the studied DLVD approaches. As hyperparameter tuning is extremely expensive for our studied approaches that consist of millions of parameters. We used the recommended parameters from previous studies. In addition, time-wise evaluation scenarios (use the earlier data for training and later data for testing) are not considered in this paper, since the dataset is at the function level, not the commit level. Note that the results obtained for the studied approaches (IVDetect, Devign, Reveal, and LineVul) are not exactly the same as the results reported in the original papers. One possible explanation is that we ran the experiment 20 times and take an average, while they report the best one. To migrate the threat, we reused the implementations published by the authors and followed the experimental setup of DLVD approaches specified in their papers. Nevertheless, our goal is not to compare the effectiveness of those DLVD approaches, instead we focus on studying the impact of data sampling approaches on the DLVD approaches. In Section 3.5.2, we selected 4 as the threshold to calculate the neighbourhood of vulnerable cases, which may introduce bias to the results. To migrate the threat, we performed the same experiments using different thresholds and

our finding still hold. Similarly, in RQ2, we selected to examine the k tokens returned by LIME and GNNExplainer, different k may lead to different results. To migrate this threat, we examined different values 1, 3, 5, 10 for k and find the finding is hold for different k . In addition, recent study LIME can be too random and may generate inconsistent results of important tokens [95] and may introduce bias to our study. We encourage future studies to try more advanced techniques for the explanation. We published our replication package to improve the transparency of our work.

External validity Threats to the external validity relate to the generalizability of our findings. Our findings might not be generalized to other datasets, DLVD approaches, data sampling approaches. We encourage future research on more approaches and datasets.

3.6 Conclusion

In this study, we conducted the first systematic and extensive study to assess the impact of data sampling on data imbalance issue in SOTA DLVD approaches. Generally, oversampling outperforms undersampling, and sampling on raw data outperforms sampling on latent space, typically random oversampling on raw data performs the best among all studied ones (including advanced one SMOTE and OSS). Surprisingly, OSS does not help alleviate the data imbalance issue in DLVD at all. While if recall is pursued, random undersampling is the best choice. Random oversampling on raw data also improves the ability of DLVD approaches for learning real vulnerable patterns. However, for a significant portion of cases (at least 33% in our datasets), DLVD approaches cannot reason their prediction based on real vulnerable statements. We

provide actionable suggestions and a roadmap to practitioners and researchers, e.g., random oversampling is recommended to handle data imbalance issue in DLVD while OSS is not recommended.

Chapter 4

One-for-All Does Not Work!

Enhancing Vulnerability Detection by Mixture-of-Experts (MoE)

4.1 Introduction

Vulnerability detection is a critical task in software engineering, aiming to identify security weaknesses that could be exploited by malicious entities. Deep learning-based vulnerability detection (DLVD) techniques have shown impressive results in academic settings [20; 22; 26; 28; 27; 96; 97].

However, a recent study [98] reveals a significant challenge that prevents deep learning models from being adopted in industry settings. The challenge is the **one-for-all design limitation**. Existing DLVD techniques follow the “one-for-all” design, where a single model is trained to predict vulnerabilities across all types of

code [24; 22; 23; 96; 26; 25; 27; 28]. While this design simplifies the development and deployment process, it does not align with the diverse and specific needs of real-world applications. In practice, large organizations often encounter various types of vulnerabilities, each with unique characteristics and implications [98]. A one-for-all model may fail to adequately address the nuances of different vulnerability types and lack the capability of detecting certain types of vulnerabilities.

One significant issue stemming from the one-for-all design is the uneven and long-tailed distribution of different Common Weakness Enumeration (CWE) [99] types in real-world codebases. Certain types of vulnerabilities, such as those listed in the CWE Top 25 [100], are more prevalent and thus receive more attention in both research and industry. However, numerous other vulnerability types are less common and often underrepresented in training datasets. This imbalance leads to a situation where DLVD techniques are highly effective at detecting common vulnerabilities but struggle with rare ones. In practical terms, this means that while a model might excel in identifying frequent issues, its performance on less common, yet potentially critical vulnerabilities is subpar. This uneven performance can undermine the overall effectiveness of DLVD techniques, leaving some vulnerabilities undetected and increasing the risk of security breaches. For instance, Figure 4.1 presents the performance of LineVul [24], a SOTA approach that is a single model trained to detect all types of vulnerabilities, on different CWE types compared with experts using the same model trained on each specific CWE type. As we can see, LineVul performs poorly on less frequent CWE types with F1-score ranging from 0.016 to 0.268. A similar trend can be observed on other one-for-all models as well.

The Mixture-of-Experts (MoE) framework [38; 101] is a promising solution proposed by previous research to resolve the limitations of the one-for-all design.

The MoE framework addresses this limitation by first splitting the input space into sub-spaces, and then leveraging multiple specialized models (i.e., experts), trained to handle each sub-space. Instead of relying on a single model to handle the entire input space, MoE includes a router that dynamically assigns the most appropriate expert for handling a given input.

To address the challenges mentioned above, we propose MoEVD, which leverages the Mixture-of-Experts (MoE) framework for vulnerability detection. Two key challenges need to be resolved to effectively leverage MoE in this context. First, how to split the input space effectively, so that each expert could focus on one type or a set of related vulnerabilities. Second, how to design a router that can effectively assign the input to the appropriate experts for handling the vulnerability detection task. To address the first challenge, we split the input code according to Common Weakness Enumeration (CWE) types [99], a widely used classification system for software vulnerabilities. This splitting mechanism allows each expert to focus on learning patterns and detecting vulnerabilities specific to a particular type of CWE or a subset of CWE types. More specifically, to construct the expert for each CWE type, we fine-tune a pre-trained model (e.g., CodeBERT [67]) to predict if a piece of code is vulnerable. To address the second challenge, we train a router to dynamically select the appropriate experts for a given input code by formulating this task as a CWE type multi-class classification task. Essentially, MoEVD decomposes the task of binary vulnerability detection into a CWE type classification task and a CWE-specific

vulnerability detection task so that specific experts can be trained to handle a specific set of vulnerabilities instead of handling all types of vulnerabilities within one model.

We performed an extensive evaluation on a widely used benchmark dataset BigVul [102]. Our results show that MoEVD achieves an F1-score of 0.44, significantly outperforming all studied state-of-the-art (SOTA) baselines by at least 12.8%. MoEVD outperforms the best SOTA baseline across almost all CWE types in terms of F1-score and recall. For instance, MoEVD improves the recall of the best SOTA baseline by a range from 9% to 77.8%. More importantly, MoEVD does not sacrifice performance in long-tailed CWE types. On the contrary, benefiting from the MoE design that enables the model to learn various vulnerability patterns individually, MoEVD improves the performance on vulnerabilities of long-tailed CWE types. For instance, MoEVD mitigates long-tailed issues by improving the F1-score of SOTA baselines on the long-tailed CWEs by at least 7.3%.

Our contributions are summarized below:

- We are the first to adapt the Mixture-of-Experts (MoE) framework to enhance vulnerability detection and our extensive evaluation demonstrates its effectiveness, showing improvements across all CWE types.
- MoEVD enhances the detection of rare CWE types of vulnerabilities that are overlooked by existing one-for-all models and makes it more suitable for real-world applications.
- To facilitate future research, we have made all the datasets, results, and code used in this study openly available in our replication package [103].

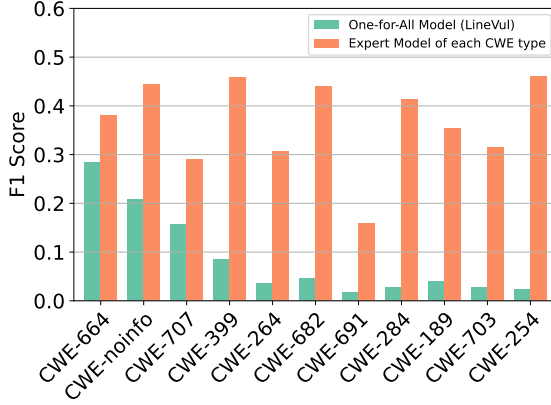


Figure 4.1: The performance of one-for-all Model (LineVul) vs. expert model trained on each specific CWE type across CWE types.

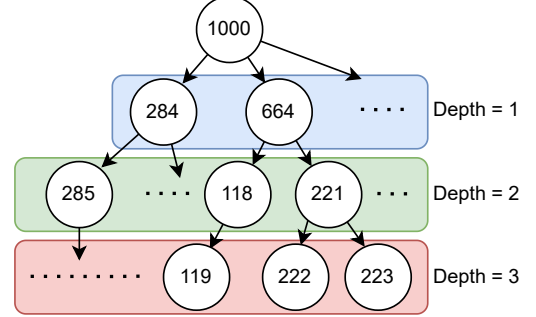


Figure 4.2: Hierarchy structure of CWE types. The arrows present the parent-child relationship between two CWE types. For instance, CWE-664 is the parent of CWE-221.

4.2 Background and Motivation

In this section, we introduce the background of our study, focusing on the pre-trained models for code representation Learning, CWE hierarchy, and Mixture-of-Experts (MoE) framework.

4.2.1 Pre-trained Model for Code Representation Learning

Deep learning-based vulnerability detection techniques rely on code representation learning to capture the syntactic and semantic properties of the code [104; 25; 27]. These techniques transform code snippets into vectors, which can be effectively processed by classification models. Recently, several pre-trained models tailored for programming languages have emerged and demonstrated promising performance in code-related tasks, such as CodeBERT [67], GraphCodeBERT [105], and UniXcoder [106]. Those models are based on transformer architecture, which has proven effective

in code-related tasks. For instance, CodeBERT is built upon the foundations of BERT [107] through further pre-training on natural language (NL) - programming language (PL) pairs. Its design and training methods enable CodeBERT to understand not only the relationship between natural language and code but also the semantics of the source code. These pre-trained models have demonstrated SOTA performance in vulnerability detection [24; 108]. In this study, we use pre-trained models as the foundation for our experts and router by fine-tuning them to adapt to downstream tasks. Specifically, we add a binary classification head to a pre-trained model for vulnerability detection and a multi-class classification head for CWE type multi-class classification.

4.2.2 Common Weakness Enumeration

Common Weakness Enumeration(CWE) is a category system for weaknesses and vulnerabilities. CWE follows a hierarchical structure, as illustrated in Figure 4.2, where each CWE type has ancestors and related successors [99]. For instance, CWE-664 (Improper Control of a Resource Through its Lifetime) has two children, CWE-118 (Incorrect Access of Indexable Resource ('Range Error')) and CWE-221 (Information Loss or Omission). Typically, the children CWE types under a parent CWE type are related and share similar vulnerability patterns. A parent CWE is a more general category encompassing its child CWE types. For vulnerabilities where the details are unknown or unspecified, the NVD assigns a placeholder value, "CWE-noinfo," when it is unable to categorize a CVE under a specific CWE entry.

4.2.3 Mixture-of-Experts (MoE)

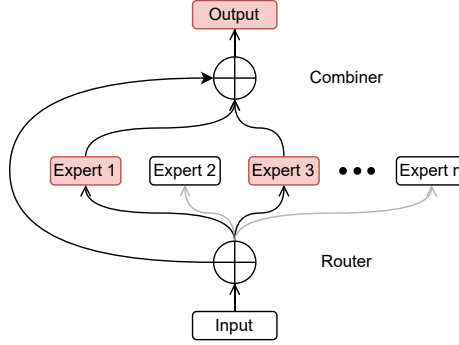


Figure 4.3: An Example Design of Mixture-of-Experts (MoE).

Mixture-of-Experts Framework Design

Mixture-of-Experts (MoE) is a combinatory approach that leverages individual learning techniques as experts specializing in distinct sub-spaces of the input space [101]. MoE, proposed by Jacobs et al. in 1991 [38], uses the idea of dividing the input space into sub-spaces, trains experts within each sub-space, and combines the knowledge of experts using a router, also called gating function. The core idea behind MoE is to select suitable experts for a particular input using a router. Typically, MoE contains the following three components:

- **Experts:** These are individual models or neural networks trained to specialize in different parts of the input space or in different tasks. Experts can be of various types, such as linear models, decision trees, or deep neural networks.
- **Router (Gating function):** The router is a model that determines which expert or combination of experts should handle a particular input. It is typically implemented as a neural network and can be trained alongside the experts.

Usually, not all experts are employed for inference; the router selects the most appropriate subset of experts for each input.

- **Combiner:** The combiner takes the outputs of the experts, weighted by the probabilities or weights assigned by the router, and produces the final output.

Let $\theta = \{\theta_g, \theta_e\}$ represent the set of parameters of MoE, where θ_g represent the parameters for the router, and θ_e denotes the parameters of experts. Given an input vector X and an output vector Y , the final probability of P can be formulated as:

$$P(Y|X, \theta) = \text{Aggregation}_{i=1}^N(P(Y, i|X, \theta)) \quad (4.1)$$

$$= \text{Aggregation}_{i=1}^N(g(i|X, \theta_g)P(Y|i, X, \theta_e^i)) \quad (4.2)$$

, where N donotes the number of experts, $g(i|X, \theta_g)$ denotes the probability of selecting the i -th expert to handle input data X , $P(Y|i, X, \theta_e^i)$ is the probability of Y generated by the i -th expert given X , and θ_e^i is the parameters for i -th expert. The function *Aggregation* aggregates results from each expert. The training objective for the MoE framework is to train both θ_g and θ_e or only θ_g .

Figure 4.3 presents a typical design of MoE. The input is first fed to the router and the router decides the most appropriate experts to handle the input. In this case, Expert 1 and Expert 3 are selected. The results returned by those two experts are aggregated in the combiner and output as the final results.

Vulnerability Detection with Mixture-of-Experts

The motivation for our approach stems from the significant limitations of the prevalent “one-for-all” design in existing deep learning-based vulnerability detection

techniques. Those techniques train a single model to handle all types of vulnerabilities, which often fails to capture the unique patterns and characteristics of different vulnerability types. This is particularly problematic for rare vulnerabilities that are underrepresented in training datasets, leading to low performance and increased security risk.

In contrast, the MoE framework allows for specialization, where each expert focuses on detecting vulnerabilities specific to a particular CWE type. This inherent partitioning aligns well with the MoE approach, where each expert can be dedicated to a specific CWE type. By leveraging the MoE framework, our goal is to improve the overall accuracy and efficiency of vulnerability detection, making it more suitable for real-world applications where diverse and specific vulnerabilities need to be addressed effectively.

4.3 Methodology

In this section, we present the design and implementation of MoEVD. Figure 4.4 illustrates the overall pipeline of MoEVD, which leverages the Mixture-of-Experts (MoE) framework to identify and detect vulnerabilities with diverse patterns. During the training phase, we focus on training two key components within the MoE framework: the experts and the router. Each expert is specialized and trained to handle a specific CWE type (or combination) of vulnerabilities. Meanwhile, the router is trained using a CWE type classification task, enabling it to recognize the characteristics of specific CWE types and direct the input code to the appropriate experts. At inference time, the router assigns the input code to the most appropriate experts

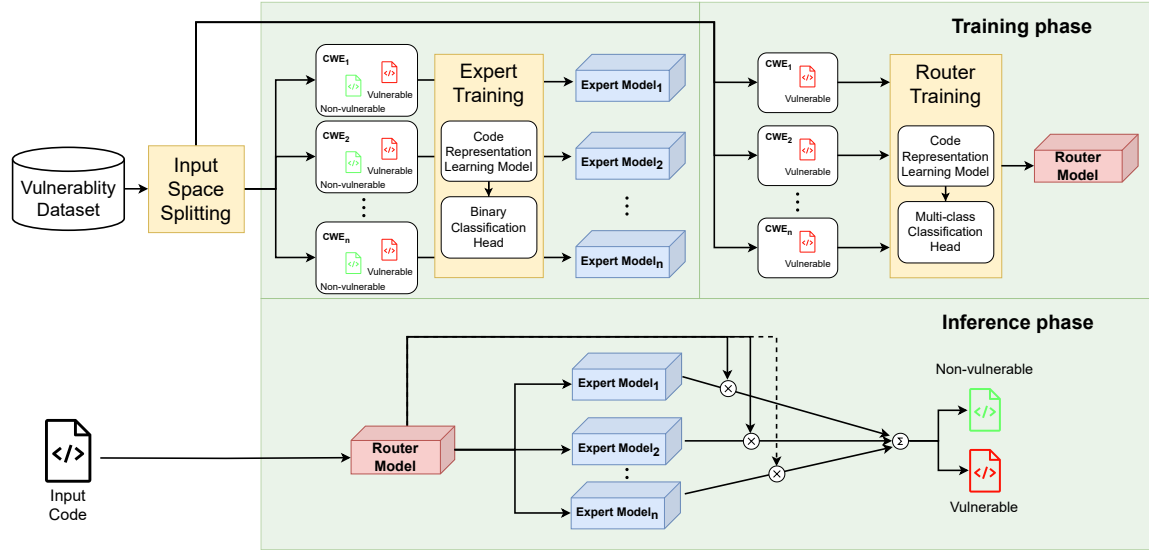


Figure 4.4: The pipeline of MoEVD.

based on its characteristics. The selected experts then process the input code to predict vulnerabilities, and their outputs are aggregated to produce the final prediction. Essentially, MoEVD decomposes the task of binary vulnerability detection task into a CWE type classification task and a CWE-specific vulnerability detection task so that specific experts can be trained to handle a specific set of vulnerabilities instead of handling all types of vulnerabilities within one model.

4.3.1 Splitting Input Space Based on CWE Types

The Mixture-of-Experts (MoE) framework provides a promising solution to the limitations of one-in-all models. To effectively leverage the MoE framework, it is crucial to split the input space into sub-spaces, allowing each expert to handle a specific sub-space. This division enables the specialization of experts, where each expert is trained to focus on and master the nuances of its designated sub-space,

leading to improved performance in their respective areas.

A common classification system to classify vulnerabilities is Common Weakness Enumeration (CWE) types, each representing a distinct category of software weaknesses, ranging from buffer overflows and injection flaws to improper authentication and information leakage [99]. These vulnerabilities of different CWE types exhibit different and often complex patterns that require specialized knowledge to detect effectively [109; 110].

To leverage the MoE framework, we split the input space of vulnerabilities into sub-spaces based on their CWE types. However, it is infeasible to train an expert for each specific CWE type directly for several reasons. First, there are too many CWE types (e.g., the BigVul dataset [102] contains 88 CWE types). Training an expert for each individual CWE type would make the number of experts excessively large and incur significant overhead in expert training. Second, the distribution of different CWE types of vulnerabilities follows a long-tailed pattern, meaning some CWE vulnerability types are rare and have only a few vulnerable instances [111]. It is challenging to train a robust expert on such rare CWE types.

To address the aforementioned challenges, we construct a CWE tree to present the hierarchical relationships and select the top-level CWE types as the basis for input space splitting. As introduced in Section 6.2, CWE follows a hierarchical structure that naturally clusters CWEs with similar vulnerability patterns, and experts trained on the parent CWE types are capable of handling their child CWE types as well.

In this study, we begin by selecting the top-level CWE types (e.g., depth=1) as the basis for input space splitting. However, we found that several CWE types

(e.g., CWE-388 and CWE-320) do not have sufficient instances. For example, in the BigVul vulnerability detection dataset, these types have less than 100 instances. For these cases, we aggregate all such top-level CWE types into a single category, named CWE-agg. This aggregation ensures that each CWE type has a sufficient number of instances for effective training while reducing the overall model complexity. We ended up with 12 CWE types. In other words, we need to train one expert for each of those CWE types with 12 experts in total.

4.3.2 Training Phase

As introduced in Section 6.2, an MoE framework typically has three components: experts, router, and combiner. To reduce the training complexity, we train the experts and the router separately. Below, we elaborate on the process of training experts and the router, and aggregating the output of experts into a final prediction.

Expert Training

Vulnerabilities of the same CWE type share similar characteristics. Therefore, for each CWE type, we train a specific expert model. For a given CWE type CWE_i , our goal is to train an expert model (Expert Model_{*i*}) to predict the probability of an input code c being vulnerable vul , i.e., $\hat{y} = P(vul|CWE_i, c, \theta_e^i)$, where θ_e^i are the parameters for Expert Model_{*i*}. Assume we have a training dataset $\mathcal{D}_{CWE_i} = \{(c_j, y_j)\}_{j=1}^{N_{CWE_i}}$ for the CWE type CWE_i , where c_j is an input code instance and y_j is the corresponding ground truth label (i.e., vulnerable or non-vulnerable). The overall loss for the expert model of CWE type CWE_i over the entire dataset can be defined as the cross-entropy

loss:

$$\mathcal{L}_{CWE_i}(\mathcal{D}_{CWE_i}, \theta_e^i) = -\frac{1}{N_{CWE_i}} \sum_{j=1}^{N_{CWE_i}} y_j \log(\hat{y}_j) \quad (4.3)$$

To construct the training data $\mathcal{D}_{CWE_i}$, we consider the vulnerable code of the specific CWE type (i.e., CWE_i) as positive labels, while treating all other code (including those labeled as other CWE types and all non-vulnerable code) as negative. This data construction strategy allows each expert to specialize in detecting vulnerabilities related to its specific CWE type, providing a more focused and accurate detection model for each type. This strategy is a key aspect of MoEVD, distinguishing it from other vulnerability detection models that classify code simply as vulnerable or non-vulnerable without considering the specific CWE type (see more discussion in Section 4.6.1).

We construct experts using pre-trained models (e.g., CodeBERT) as the backbone and fine-tune an individual expert for each CWE type by following previous studies [24; 96; 104]. We initialize each expert model with the pre-trained weights and add a binary classification head. Then we optimize the model parameters θ_e by minimizing the overall loss function $\mathcal{L}_{CWE_i}(\mathcal{D}_{CWE_i}, \theta_e^i)$.

Router Training

The goal of this step is to train a router that can distinguish the code characteristics of different CWE types, so that the appropriate experts can be selected during routing. We formulate the task of router training as a CWE type multi-class classification task. Given an input code c , the task is to classify its CWE type,

$CWE_i \in \{CWE_1, CWE_2, \dots, CWE_i, \dots\}^N$, where N is the number of CWE types. The training objective is to learn a model to predict the probability of the code c being categorized as a specific CWE type CWE_i , $g_i(CWE_i|c, \theta_g)$. To make the router proficient in differentiating between the features of different CWE types and recognizing the nuances of various vulnerabilities, we do not include non-vulnerable code in the training dataset. Let $\mathcal{D}_R = \{(c_j, t_j)\}_{j=1}^{N_R}$ be the training dataset for the router, where c_j is an input code instance and t_j is the corresponding CWE type label, N_R is the size of the training data for the router. The probability of code c_j being classified as the i th class CWE_i by the router model g is given by:

$$\hat{t}_{ji} = g(CWE_i|c_j, \theta_g)$$

Note that the CWE types suffer from long-tail issues [111], meaning CWE types are highly imbalanced. To mitigate this issue, we use Focal Loss [112], defined as:

$$\mathcal{L}_{\text{focal}}(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t) \quad (4.4)$$

where p_t is the predicted probability for the true class. α_t is a balancing factor for class t which is used to balance the importance of different classes. α_t can be set higher for minority classes to ensure they contribute more to the loss, helping the model pay more attention to underrepresented classes. γ is a focusing parameter to down-weight easy examples and focus on hard examples.

We adopt Focal Loss to our multi-class classification task and the overall focal loss for the router model over the entire dataset $\mathcal{D}_R$ can be defined as:

$$\mathcal{L}_R(\mathcal{D}_R, \theta_g) = -\frac{1}{N_R} \sum_{j=1}^{N_R} \sum_{i=1}^N \alpha_i (1 - \hat{t}_{ji})^\gamma t_{ji} \log(\hat{t}_{ji}) \quad (4.5)$$

where N is the number of CWE types, α_i is the balancing factor for class CWE_i , γ is the focusing parameter, t_{ji} is the ground truth label for code c_j for class CWE_i , and $\hat{t}_{ji}$ is the predicted probability for code c_j for class CWE_i .

Similar to expert training, we construct a router by fine-tuning a pre-trained model (e.g., CodeBERT) by adding a multi-class classification head. We select the pre-trained model because it has demonstrated proficiency in capturing the patterns from source code [104; 25; 27], which is essential for our router to assign input code to the appropriate experts based on their CWE patterns. We initialize the router model with the pre-trained weights and then optimize the parameters θ_g by minimizing the overall loss function.

Combiner

The combiner aggregates the output of the selected experts using a weighted summing mechanism:

$$P(vul|c) = \sum_{i=1}^K Softmax(g(CWE_i|c, \theta_g)) P(vul|CWE_i, c, \theta_e^i) \quad (4.6)$$

K is the number of experts that are selected to handle the code c . In this study, we choose the top K experts most suitable for handling c based on the probabilities $g_i(c, \theta_g)$ produced by the router. Instead of using all experts with different weights, using only the top K experts reduces the inference cost and reduces the impact/bias caused by irrelevant experts. Note that we use the probability of each expert produced

by the router output as the weight to control the contribution from each selected expert.

The *Softmax()* function ensures that the probabilities for the selected top K experts sum up to 1. For example, setting K to 2 and the probabilities for the top 2 experts are 0.45 and 0.15. After Softmax normalization, their probabilities are 0.57 and 0.43, respectively.

We investigate the impact of different K values in MoEVD in Section 5.5.4. Our experiments suggest by selecting $K = 2$, MoEVD provides the best trade-off between performance and inference time agreeing with existing practices on the selection of K [113].

4.3.3 Inference Phase

During the inference phase, the router processes both vulnerable and non-vulnerable input code. The router first assigns the input code to one or more experts based on the CWE classification result. Then the selected experts further predict whether the input code belongs to a specific vulnerability or not. The aggregation mechanism combines the outputs from the experts to make a final prediction on whether the code is vulnerable or non-vulnerable. By leveraging the specialized knowledge of each expert and the ability of the router to classify CWE types, MoEVD is able to outperform existing tools in vulnerability detection.

Note that our router is trained exclusively on vulnerable code, it may lack specific knowledge about non-vulnerable code. One question raised here is which expert will be assigned for the non-vulnerable code. According to the design of our router, non-

vulnerable code will be assigned to the experts that handle vulnerable code with the most similar characteristics to that of the input non-vulnerable code, so that it can be further identified by the corresponding experts. However, even though the non-vulnerable code is assigned to other experts, we do not need to worry about this since we train all our experts including the data of the entire same set of non-vulnerable code. In theory, all experts should have consistent capability in identifying non-vulnerable code.

4.4 Experiment Design

In this section, we outline our research questions (RQs), describe the datasets used, define the evaluation metrics, present our analysis approach for each RQ, and provide implementation details.

4.4.1 Research Questions

We evaluate MoEVD across various dimensions to address the following research questions:

- *RQ1: How effective is MoEVD compared with SOTA approaches in vulnerability detection?*
- *RQ2: How effective is the MoE framework within MoEVD?*
- *RQ3: How effective is MoEVD in detecting each CWE type and handling long-tailed distributions compared to SOTA approaches?*

- *RQ4: What is the impact of varying the number of experts on the performance of MoEVD?*

In **RQ1**, we aim to assess the overall performance of MoEVD compared to current SOTA approaches. **RQ2** focuses on evaluating the effectiveness of the MoE framework in routing inputs to the most appropriate experts. **RQ3** examines MoEVD’s effectiveness in detecting various CWE types and its ability to handle long-tailed distributions compared to SOTA approaches. **RQ4** investigates the impact of selecting different numbers of experts on the performance of MoEVD.

4.4.2 DLVD Baselines

In our study, we select two representative groups of DLVD baselines, graph-based models and transformer-based models:

- **Graph-based models:** We first select three of the most commonly used models, SySeVR [23], Devign [25] and Reveal [26]. SySeVR leverages the abstract syntax tree (AST) graph generated from code. The rest two models leverage graph neural network (GNN) [114] to learn code feature representations and have shown promising results. We also employ LIVABLE [97], which is so far the state-of-the-art approach that combines graph-based model with text-based model.
- **Transformer-based models:** We select two base models CodeBERT [67] and UniXcoder [106], which have been shown to be effective in vulnerability detection [67; 106; 24]. Additionally, we select the recent SOTA approach CausalVul [96], which introduced calculus-based causal learning. For

a fair comparison, we implement CausalVul on these two models, resulting in CausalVul_{CodeBERT} and CausalVul_{UniXcoder}.

4.4.3 Experiment Dataset

We select our experiment vulnerability detection dataset based on two criteria: 1) Diversity. They must be collected from a wide range of code repositories to maintain various types of vulnerability. 2) Annotations. They must contain CWE type or CVE-id annotation for training MoEVD. Therefore, we select the BigVul dataset [102] as our experiment dataset, which is widely used in prior studies [104; 27; 24; 97]. The BigVul dataset contains CVEs from 2002 to 2019, extracted from over 300 different open-source C/C++ projects, encompassing 88 different CWE vulnerability types. It contains 10,547 vulnerable functions and 168,752 non-vulnerable functions. Following the same experiment setting of previous studies [96; 104], we use the same 80%/10%/10% split as train/val/test data for the BigVul dataset. We also use the same exact dataset partition used in CausalVul for a fair comparison. After aggregating CWE types as described in Section 4.3.1, we obtain 12 CWE types.

Table 4.1: Overview of the studied dataset.

Dataset	Granularity	#Project	#Vuln	#Non-Vuln	#CWE
BigVul [102]	function	348	10,547	168,752	88

4.4.4 Evaluation Metrics

We use F1-score, precision, and recall as our evaluation metrics, which are widely used in previous studies [25; 27; 26; 96; 104]. The F1-score provides a balance between precision and recall, offering a single metric that considers both false positives and false negatives. Precision measures the accuracy of positive predictions, while recall measures the ability to identify all relevant instances. Unlike previous research, we do not use accuracy as an evaluation metric due to the highly imbalanced nature of vulnerability detection datasets, which can result in misleading accuracy values dominated by the majority class [115].

4.4.5 Approach for Research Questions

Approach of RQ1

To demonstrate the effectiveness of MoEVD, we compare it with graph-based models (Devign [25], Reveal [26], LIVABLE [97]) and transformer-based models (CodeBert [67], UniXcoder [106], and CausalVul [96]). Depending on different base transformer models (CodeBERT and UniXcoder), there are two variants of CausalVul, namely CausalVul_{CodeBERT} and CausalVul_{UniXcoder}. Similarly, we also implement MoEVD using two different base transformer models, resulting in two variants, namely MoEVD_{CodeBERT} and MoEVD_{UniXcoder}.

Approach of RQ2

In RQ2, we evaluate the effectiveness of the MoE framework of MoEVD. To do that, we keep every trained expert untouched and compare MoEVD with two different

variants: ensemble and random as described below.

- **Ensemble** Ensemble techniques train a machine learning (ML) model to fuse multiple models. We compare our MoE framework with ensemble techniques, replacing the MoE framework with ensemble techniques while keeping all other components unchanged. Specifically, we use the prediction probabilities of the positive class (vulnerable) from each expert model as input features for training an ensemble model. We select Random Forest (RF)[116], XGBoost[117], CatBoost [118], and LightGBM [119] as baselines due to their widespread use and competitive performance in classification tasks [120; 121; 122]. For simplicity, we call those variants with different ensemble methods as Ensemble_{RF} , $\text{Ensemble}_{XGBoost}$, $\text{Ensemble}_{CatBoost}$, $\text{Ensemble}_{LightGBM}$.
- **Random Router:** To test the effectiveness of our router, we also compare MoEVD with a variant in which the router selects experts *randomly*. In this variant, we replace our router with a random expert selector and keep all other components unchanged. We define this variant as MoEVD_{random} .

We evaluate the performance of the above two settings and compare them with MoEVD. In this RQ, we choose the best-performing implementation of MoEVD, $\text{MoEVD}_{CodeBERT}$ to compare with.

Approach of RQ3

To answer RQ3, we compare the performance between the selected best-performing baseline ($\text{CausalVul}_{CodeBERT}$) and $\text{MoEVD}_{CodeBERT}$. We first analyze

the performance (recall) of selected models in detecting vulnerabilities across each CWE type. BigVul dataset has 88 CWE types. Note that the model performance may not be stable enough on some CWE types which have an extremely small amount of vulnerabilities. In order to reduce the potential performance bias, we group the CWE types that have fewer than 10 vulnerable codes together as $\text{CWE-N} \leq 10$. Note that for RQ3, CWE types are not aggregated by the CWE hierarchy; therefore, they are not the same as the CWE types used for training the expert models.

To evaluate the effectiveness of MoEVD in the long-tailed scenario, we first construct the long-tailed dataset. We split the BigVul dataset into two groups (the head group and the tail group) by following the settings from previous research [111]. More specifically, we categorize vulnerabilities into two groups: the head group, comprising the most frequent CWEs that constitute at least 50% of BigVul, and the tail group, consisting of the remaining vulnerabilities. In the BigVul dataset, 4 CWE types (CWE-119, CWE-noinfo, CWE-20, and CWE-399) belong to the head group (56.2% of vulnerabilities in total), while the other 84 CWE types belong to the tail group (43.8% of vulnerabilities in total). Finally, we compare the performance of $\text{CausalVul}_{\text{CodeBERT}}$ and $\text{MoEVD}_{\text{CodeBERT}}$ between the two groups, respectively.

Approach of RQ4

In MoE, the number of experts K is an important parameter. In this RQ, we aim to investigate the impact of the value K on MoEVD’s performance. Our goal is to find the value of K that achieves the optimal balance between performance and computational cost.

4.4.6 Implementation Details

We downloaded the official pre-trained models and tokenizers for all evaluated transformer models from HuggingFace [123]. We implemented and conducted all experiments using PyTorch [124] and Autogluon [125]. To train the expert and router models, we used a batch size of 32, a learning rate of 1e-5, the AdamW [126] as the optimizer, and trained the model for 10 epochs. For Focal Loss, we use the default setting [127], i.e., setting the focusing parameter γ to 1 and the balancing factor α_t as the inverse of its percentage of number of samples for each CWE type. To optimize computational efficiency, we used the bfloat16 (brain floating point) mixed precision. All experiments were conducted on a Linux server equipped with four Nvidia RTX 3090 GPUs, 24 CPU cores, and 128GB of memory. Each expert’s training time is approximately 1 hour, and the router model’s training time is approximately 10 minutes. During inference, when we set the number of experts to $K = 2$, MoEVD can scan more than 100 code snippets per second. For training the ensemble model in 4.4.5, We used a state-of-the-art AutoML tool Autogluon [128] for training the ensemble model. Autogluon trains ML models with hyperparameter tuning, bagging, stacking and other techniques to optimize the performance, we select the four highest-performing ensemble models as our baselines.

We reproduce all DLVD baselines by using the replication packages provided in their works [25; 26; 67; 106; 97; 96]. We also adopt the same hyperparameter settings in their works.

4.5 Results

4.5.1 RQ1: MoEVD vs. DLVD Baselines

MoEVD outperforms the best baseline by 12.8%. Table 4.2 presents the performance comparison between our approach and studied baselines. We observe that MoEVD outperforms all SOTA baselines, including graph-based and transformer-based models in all evaluated metrics (F1-score, precision, and recall). Specifically, MoEVD_{CodeBERT} achieves the best F1-score of 0.44, improving baselines from 12.8% to 69.2%. When looking at recall, MoEVD_{CodeBERT} also achieves the best performance (0.47), while in terms of precision, MoEVD_{UniXcoder} achieves the best performance (0.47).

MoEVD is able to deliver the best performance for all base models. For both CodeBERT and UniXcoder variants of MoEVD, MoEVD outperforms the base model and the corresponding CausalVul variant. MoEVD_{CodeBERT}, which is the best-performing variant based on CodeBERT, achieves the highest F1-score of 0.44, achieving a 15.8% improvement over CodeBERT with an F1-score of 0.38. MoEVD also consistently outperforms other approaches in terms of recall, which indicates MoEVD is effective in recognizing more vulnerabilities, highlighting the effectiveness of MoEVD over existing one-for-all DLVD approaches.

Table 4.2: The performance of different VD approaches on BigVul dataset in terms of studied metrics.

Model Type	Approaches	F1-score	Precision	Recall
Graph-based	SySeVR	0.19	0.31	0.14
	Devign	0.27	0.30	0.25
	Reveal	0.26	0.23	0.30
	LIVABLE	0.39	0.40	0.37
Transformer-based	CodeBERT	0.38	0.45	0.33
	CausalVul _{CodeBERT}	0.39	0.43	0.36
	MoEVD _{CodeBERT}	0.44	0.42	0.46
	UniXCoder	0.38	0.46	0.32
	CausalVul _{UniXcoder}	0.39	0.35	0.45
	MoEVD _{UniXcoder}	0.43	0.47	0.40

MoEVD significantly outperforms all SOTA baselines. MoEVD’s implementation with the CodeBERT model (MoEVD_{CodeBERT}) achieves the best F1-score of 0.44 with 12.8% improvement over the best SOTA baseline. Additionally, MoEVD’s implementation with the UniXcoder model (MoEVD_{UniXcoder}) achieves the highest precision of 0.47 and a notable F1-score of 0.43.

4.5.2 RQ2: Effectiveness of MoE

MoEVD outperforms variants with ensemble methods by at least **12.7% F1-score**. Table 4.3 presents the comparison between MoEVD and its variants with ensemble methods. MoEVD significantly outperforms those variants in

Table 4.3: The performance of variants with different ensemble methods and routers.

Variant	F1-score	Precision	Recall
Ensemble _{LightBGM}	0.38	0.54	0.29
Ensemble _{CatBoost}	0.39	0.50	0.31
Ensemble _{XGBoost}	0.39	0.51	0.31
Ensemble _{RF}	0.38	0.49	0.30
MoEVD _{random}	0.08	0.07	0.09
MoEVD	0.44	0.42	0.46

terms of F1-score and recall. Specifically, compared to the best-performing variant, Ensemble_{XGBoost}, MoEVD achieves a 12.7% improvement in F1-score. These results suggest that the MoE framework is superior to other ensemble methods. A potential reason is that combining the results of all experts introduces additional noise, leading to less accurate final results.

To understand the impact of the router, we compare the performance of MoEVD and MoEVD_{random}. From Table 4.3, we observe that the performance of MoEVD_{random} is extremely low (a F1-score of 0.08). Such a result indicates that selecting appropriate experts is crucial for our approach. We further measure the router’s accuracy in selecting the correct expert for vulnerable input code. If the router can select the correct expert within the top K assigned experts, we consider it correct, otherwise, we consider it incorrect. The ground truth of an expert is determined by its CWE types after aggregation. If a vulnerable code of a CWE type A is assigned to an expert whose training data consists of CWE A, we consider the vulnerable code

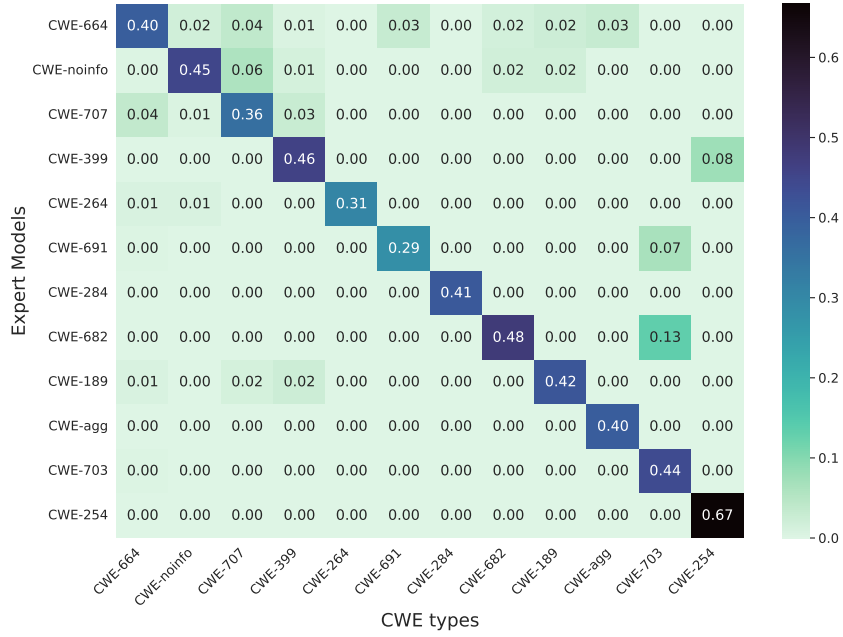


Figure 4.5: The performance matrix of each expert model on each CWE type in terms of F1-score. Darker colors indicate better performance.

to be assigned correctly. As shown in Table 4.4, in 63.8% of the cases, the router can select the correct experts. In other words, there is still room for improvement (see more details in Section 4.6.3).

The router is an essential component of the MoE framework, correctly assigning the vulnerability to the correct CWE expert is important. To further understand how the router’s performance affect the overall VD performance of MoEVD, we compare the performance between the cases where experts are correctly selected by the router and experts are wrongly selected. As the results shown in Table 4.4, if the experts can be selected correctly, the performance is significantly better (by 133.7%) than those who failed.

The subpar performance when the expert is wrongly selected is expected due to

Table 4.4: The performance of MoEVD when the router can select experts correctly (Correctly selected) and wrongly (Wrongly selected).

Expert Selection	% vuln code	F1-score	Precision	Recall
Correctly Selected	63.8%	0.44	0.36	0.56
Wrongly Selected	36.2%	0.18	0.13	0.28

Table 4.5: Performance of CodeBERT, MoEVD_{CodeBERT}, and CausalVul_{CodeBERT} on Head and Tail groups.

Model	Head			Tail		
	F1	Prec.	Rec.	F1	Prec.	Rec.
CodeBERT	0.34	0.30	0.38	0.27	0.24	0.32
CausalVul _{CodeBERT}	0.34	0.30	0.39	0.26	0.23	0.31
MoEVD _{CodeBERT}	0.37	0.29	0.50	0.29	0.23	0.42

the design of MoE. Each expert is trained on a specific CWE type and cannot identify the pattern of other CWE types. This can be observed in Figure 4.5, where we show the performance of each expert across different CWE types. As expected, the experts only perform well on their own specific CWE type.

Mixture-of-Experts (MoE) design outperforms all ensemble models. The router plays a crucial role in MoEVD design, and significantly impacts overall MoEVD performance.

4.5.3 RQ3: Effectiveness of MoEVD across CWE Types

MoEVD outperforms baselines in most CWE types in terms of all studied metrics, typically recall and F1-score. Figure 4.6 presents the recall achieved

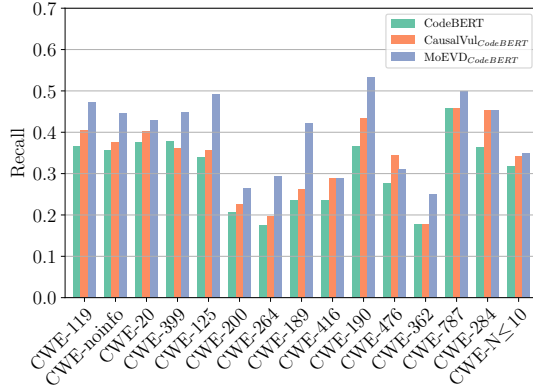


Figure 4.6: Recall of CodeBERT, MoEVD_{CodeBERT}, and CausalVul_{CodeBERT} on different CWE types. The CWE types are displayed in descending order of frequency, from left to right.

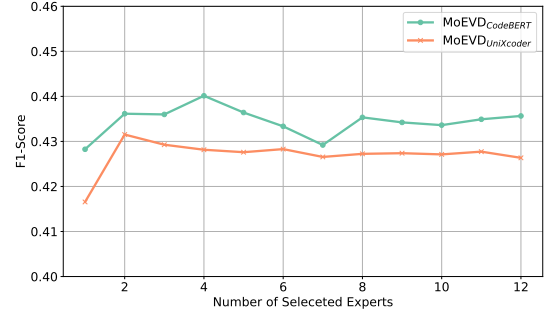


Figure 4.7: F1-score of MoEVD_{CodeBERT} and MoEVD_{UniXcoder} with different number of selected experts (K).

by CodeBERT, CausalVul_{CodeBERT}, and MoEVD_{CodeBERT} on all CWE types. We observe that in most of the CWE types (14 out of 15), MoEVD_{CodeBERT} outperforms CodeBERT and CausalVul_{CodeBERT} with a range of improvement from 9% (CWE-787) to 77.78% (CWE-189) in terms of recall. In terms of F1-score, we observe a similar trend that in 11 out of 15 CWE types, MoEVD_{CodeBERT} outperforms CausalVul_{CodeBERT}, and 9 out of 15 in terms of precision. Due to the page limit, we put the complete results of F1-score and precision in our replication package [103].

MoEVD_{CodeBERT} outperforms SOTA baselines CausalVul_{CodeBERT} and CodeBERT in both the head and the tail groups in terms of F1-score and recall, and achieves similar precision compared to SOTA baselines. Table 5.6 presents the results of studied approaches on the head and the tail groups. We observe an improvement of MoEVD_{CodeBERT} over the SOTA baselines in both the head and the tail groups. In terms of F1-score, MoEVD_{CodeBERT} improves

CodeBERT by 9.8% and CausalVul_{CodeBERT} by 7.3% in the head group. In the tail group, MoEVD_{CodeBERT} improves CodeBERT by 7.3% and CausalVul_{CodeBERT} by 11.4%. When comparing recall, MoEVD_{CodeBERT} achieves a significant improvement of 31.2%, compared with baselines while maintaining a similar precision. The results indicate that, unlike baselines, when improving the binary classification performance, MoEVD does not sacrifice the performance in long-tailed CWE types. On the contrary, benefiting from the MoE design that enables the model to learn various vulnerability patterns individually, MoEVD improves the performance on the tail group.

MoEVD_{CodeBERT} outperforms SOTA baselines on most of the CWE types in terms of F1-score and recall. For instance, MoEVD_{CodeBERT} improves the recall of SOTA baseline by a range from 9% to 77.8%. MoEVD_{CodeBERT} also mitigates long-tailed issue by improving F1-score of SOTA baselines on the tail group at least by 7.3%.

4.5.4 RQ4: Impact of the Number of Selected Experts

More experts do not yield significant performance improvement. Figure 4.7 presents the F1-score of MoEVD_{CodeBERT} and MoEVD_{UniXcoder} with a different number of selected experts (K). We observe that no significant advantages are achieved by increasing the number of experts. The worst performance for both models occurs at $K = 1$. MoEVD_{CodeBERT} achieve its optimal performance when $K = 4$, and MoEVD_{UniXcoder} achieves its optimal performance at $K = 2$. When K is greater than 2, the increase of the number of selected experts does not yield significant benefits for both models. To balance the overhead and the performance, we set K to 2 in this study.

One possible reason for the lack of significant improvement with more experts is that each expert is only effective in its own CWE type and cannot identify the vulnerability for other types as indicated in Figure 4.5. Assigning additional experts will reduce the contribution of the most effective experts therefore adding more noise to the combiner.

We do not observe significant advantages in increasing the number of experts. Employing a single expert yields sub-optimal results. Through our experiments, we find $K = 2$ achieves an optimal balance between inference time and performance.

4.6 Discussion

In this section, we discuss our special design in terms of expert training, the overhead, and the potential future research direction.

4.6.1 Expert Training Task

Different from 1) classifying vulnerability types for a given vulnerability and 2) identifying whether a given input is a vulnerability or not, our experts learn vulnerability knowledge by identifying whether the given input belongs to a specific type of vulnerability or not. As described in Section 4.3, when training an expert for a specific CWE type, we only consider the vulnerable code of that type as positive, and all other code (including other CWE type vulnerable code and non-vulnerable code) as negative. By doing so, an expert is more focusing on learning the pattern of a specific type of vulnerability, improving the performance on distinguishing the target type of vulnerability.

To validate our approach, we also trained a variant of MoEVD_{CodeBERT} using only non-vulnerable data as the negative label, excluding code from other CWE types. The F1-score dropped from 0.44 to 0.41. This result indicates that using data from other CWE categories as negative helps the experts learn to distinguish between different types of vulnerabilities more effectively.

4.6.2 Overhead of MoE Compared With One-for-All Model

The training cost of MoEVD depends on the number of experts to train. For example, in the BigVul dataset, we have 12 CWE types and consequently we have 12 experts to train. Assume the total tunable parameters of a one-for-all model with the same base model (e.g. CodeBERT) is N . Then, MoEVD have a total of $12N + R$ tunable parameters, where R is the tunable parameters of the router network. During inference, the computation cost depends on the number of experts K as discussed in RQ4. Assume the inference cost of a one-for-all model is x , and we set $K = 2$, then the total computation cost will be $2x + r$ where r is the inference cost of the router. In our case, we use the same base model for experts and the router, and we set K to 2. The inference cost is $3x$. The time cost under our experiment setting can be found in section 4.4.6.

4.6.3 Potential Future Direction

Our research is the first study in vulnerability detection that employs the MoE framework. The design of MoEVD is straightforward, utilizing the vanilla MoE framework proposed by Jacobs et al. [38], which already demonstrates a significant improve-

ment over the one-for-all design. In RQ2 (table 4.4), we investigate the performance of the router, and we notice that only 63.8% of the vulnerable code was correctly directed to the appropriate experts. In the ideal setting, if all the vulnerable code can be correctly assigned to the appropriate experts, the F1-score for MoEVD would be improved from 0.44 to 0.51. Thus, there remains a large margin to improve with the MoE framework and we believe that leveraging MoE in vulnerability detection is key to making DLVD methods more practical in real-world scenarios. One possible future direction is to improve the accuracy of the router to select the appropriate experts. Another possible direction is to leverage a more advanced MoE framework [113; 129; 130]. We encourage future research to investigate those two directions.

4.6.4 Threats to Validity

Internal Validity

MoEVD’s performance may be influenced by the distribution of CWE types within the training data, particularly for unseen CWEs. To mitigate this potential bias, we trained MoEVD on the BigVul dataset, which is one of the most comprehensive datasets, containing over 10,000 vulnerability instances across 88 distinct CWE types. For vulnerabilities belonging to unseen CWE types, MoEVD can be easily extended by training and adding specific experts for new vulnerabilities and re-training the router.

External Validity

Threats to external validity relate to the generalizability of our findings. In this study, we evaluate MoEVD using CodeBERT and UniXCoder as base models due to their widespread use in code-related tasks. While our findings might not be generalized to other models. However, As a framework, MoEVD is designed to be flexible and can integrate any model as an expert, future research is encouraged to explore the performance of MoEVD with a broader range of models.

4.7 Conclusion

In this work, we present MoEVD, the first approach in vulnerability detection using the Mixture-of-Experts (MoE) framework. MoEVD mitigates the limitations of traditional “one-for-all” design by leveraging specialized experts for different CWE types. MoEVD achieved the highest F1-score of 0.44 on the BigVul vulnerability detection dataset, surpassing the best existing approach by 12.8%.

In handling long-tailed distributions, MoEVD maintains strong performance across both common and rare CWE types. Benefiting from its ability to learn various vulnerability patterns individually, MoEVD improved the performance on the tail group by 31.2% in terms of recall while maintaining similar precision. As the first application of the MoE framework in vulnerability detection, our work suggests the possibility of leveraging specialized models and indicates potential for more advanced MoE architectures to further enhance detection capabilities.

Chapter 5

VFDelta: A Framework for Detecting Silent Vulnerability Fixes by Enhancing Code Change Learning

5.1 Introduction

Open-source software (OSS) is widely used by developers around the world. Developers must effectively manage OSS vulnerabilities, e.g., timely detection and remediation. Failure to address these vulnerabilities promptly exposes users to significant security risks, potentially leading to severe consequences [5; 6]. To enhance OSS vulnerability management, the Coordinated Vulnerability Disclosure (CVD) process is widely adopted [131; 132; 133]. This process dictates that vulnerabilities should be

fixed before being publicly disclosed, enabling developers to initiate the remediation process promptly upon disclosure. Additionally, to mitigate the risk of disseminating vulnerability-related information, CVD recommends fixing vulnerabilities silently, such as refraining from including vulnerability details in commit messages. However, due to various factors like limited resources or lengthy fix-to-integration release cycles, the timing of public disclosure often deviates from the fixed date, resulting in a time gap and opening the door for attack [134; 13; 135]. Due to the transparent nature of OSS, attackers can exploit this time gap to uncover fixes and derive the corresponding vulnerabilities before public disclosure. For instance, an industry report shows that unpatched vulnerabilities are directly responsible for up to 60% of all data breaches [6]. Therefore, developers need to detect silent fixes to start the remediation process as soon as possible.

Therefore, there is an emergence for the research community and industry to invest efforts in developing techniques to identify silent vulnerability fixes. Current approaches typically train the commit-level prediction model in two phases [13; 17; 19]. First, they train embedding models independently on different levels of granularity by fine-tuning pre-trained models (e.g., CodeBERT [67]). In the second stage, they feed the joint representations produced by multiple embedding models into a new ensemble classification model, which needs to be trained as well, and predict whether the commit is to fix a vulnerability.

Existing approaches exhibit several limitations.

❶ Inefficient code change representation. They typically represent changes as the concatenation of added code and removed code as a single sequence [13; 17; 19],

or as the concatenation of the resultant embeddings of added code and removed code produced by embedding models [17]. They assume the code changes represented in the concatenated sequence or the concatenated embedding could be captured well in the new ensemble classification model. However, pre-trained language models are not trained in such a concatenated input style. Those representation approaches lack the direct induction to highlight the changes or interaction between the removed/added code, which hinders existing approaches from effectively capturing nuanced information from code changes. **❷ Missing context.** They only consider added/removed code, but do not consider the surrounding code as the context [13; 17; 19], which has been demonstrated to help capture the semantics of the code in various software engineering tasks [136; 137]. **❸ Heavy training process.** They tend to be heavy-weight, necessitating the training of multiple models individually in multiple stages to handle various granularity of information (e.g., lines, hunks, files, or entire commits) [13; 17; 19]. The fine-tuning of the embedding models is separated from the final commit-level classification model’s training, which under-optimizes the entire model’s effectiveness.

To address the limitations mentioned above, we present *VFDelta*, a lightweight yet effective framework to identify commit-level vulnerability fixes by enhancing the code change representation learning and model training process. To address the Limitations **❶** and **❷** and improve the code change representation learning, we propose *Code Delta Representation*. The design of Code Delta Representation is inspired by the principles behind word2vec [138], which captures the semantic relationship between two words by analyzing their latent space. Similarly, to directly capture the

difference between two pieces of code in *latent* space, Code Delta Representation first uses two independent code embedding models to embed the code before and after the change (including the surrounding code as the context), respectively. Then the code change representation is represented as the result of the *element-wise subtraction* between the two embeddings.

The resultant delta embedding is then fed into a fully connected layer (as our classification head) for file-level prediction. More specifically, in the training phase, we train a lightweight model to predict file-level vulnerability fixes (i.e., predicting whether a modified file in a commit is vulnerability-fixing or not). During the inference phase, given a commit, we aggregate the probability of all the files of the commit by averaging them as the final commit-level prediction result. VFDelta avoids ensembling multiple embedding models into the final commit-level classification model. Benefiting from the lightweight of our model architecture, the entire model (both the code embedding models and file-level classification network), could be optimized together at once during the training phase (to address Limitation ③). A key architectural choice in VFDelta is to embed the *before* and *after* code with two independent model. Keeping the parameters separate lets each model specialise: the "before" model focuses on capturing patterns that commonly occur in *vulnerable* code, while the "after" model focuses on the characteristics of *repaired* code. Subtracting the two latent vectors therefore magnifies nuance vulnerability fix (e.g., > to >= in a boundary check) that would be almost invisible to a single shared model. Our ablation study demonstrates the effectiveness of our design: replacing the dual-model setup with a single shared model leads to a performance drop in both F1-score and CostEffort.

We evaluate our framework on the VFM2021 dataset [13], which consists of 1,147 Java CVE patches from 146 OSS projects collected until Feb. 2021 and 747 Python CVE files (522 commits) from 84 OSS projects. We compare it with state-of-the-art (SOTA) baselines VulFixMiner [13] and MiDas [17]. For Java, VFDelta outperforms the baselines on all metrics and achieves an F1 of 0.33, a CostEffort@5 of 0.632, and a CostEffort@20 of 0.76, with an improvement of 77.4%, 8.4%, and 5.1% over the best baseline, respectively. For Python, VFDelta achieve an F1 of 0.194, CostEffort@5 of 0.605, and CostEffort@20 of 0.856, outperforming the strongest baseline by 128.2%, 28.4%, and 6.3%, respectively. A higher CostEffort@L indicates that VFDelta can find more vulnerability fixes with less human inspection effort. Our ablation analysis shows that our code change representation learning plays a crucial role in VFDelta, and is typically effective in capturing small code changes. Furthermore, previous studies on identifying vulnerability fixes overlook evaluating their approach in the real-world setting (i.e., splitting the data in chronological order) [13; 17; 19], which is unrealistic [139; 140]. To fill this gap, we tested the effectiveness of VFDelta in real-world scenario. We extended the VFM dataset to December 2023 and performed temporal prediction (use data before 2021 for training and data of 2021-2023 for evaluation). VFDelta achieves a large margin of improvement over baselines, i.e., improving at least 64.5% and 93.9%. in terms of F1 and CostEffort@5, respectively.

In summary, this paper makes the following contributions:

- We propose a lightweight yet effective framework to identify vulnerability fixes by enhancing code change representation learning and the training process.
- We conducted extensive experiments and show that our framework significantly

outperforms all state-of-the-art baselines in identifying commit-level silent vulnerability fixes in real-world scenario.

- We release our enhanced temporal VFM2023 dataset to facilitate future research [141].

5.2 Background

5.2.1 Silent commit-level vulnerability fix identification

The primary objective of the task is to predict whether a commit is intended for vulnerability-fixing. The input is a commit and the output is a label either vulnerability-fixing (VF) or non-vulnerability-fixing (NVF). Commit comprises several types of information, which can be broadly categorized into three groups: commit metadata (including details such as the committer, commit date, repository, etc), commit message, and commit code changes. A commit's code change consists of alterations across multiple files, with each file's modification potentially including multiple changes at the hunk level. Models can, therefore, be designed to operate at various granularities, e.g., commit-level and file-level.

The typical framework for training a model to identify the commit-level vulnerability fixes could be decomposed into the following phases: 1) feature extraction, 2) code change representation learning, and 3) classification training. In the feature extraction, features such as code changes and commit messages will be extracted. During code change representation learning, the extracted features will be represented into vectors (i.e., embedding). How features are extracted and how a representation is

learned depends on the techniques and model design. We will introduce more details on the code change representation learning in Section 5.2.2. Once the representation learning is done, a binary classification model is trained to distinguish vulnerability fixes. Many previous works have developed mechanisms (e.g. neural network or ensemble model) to aggregate features from multiple fine-grained levels, culminating in a final prediction at the commit level. This aggregation ensures that the classifier makes a commit-level prediction based on a comprehensive view of the diff, facilitating accurate vulnerability-fixing commit identification.

Early work focused on using commit messages to identify vulnerability fixes [42; 44]. However, Coordinated Vulnerability Disclosure (CVD) recommends fixing vulnerabilities silently, such as refraining from including vulnerability details in commit messages, which makes it challenging to leverage the message for this task. Therefore, in this study, we focus on only using code changes to identify silent commit-level vulnerability fixes as recent work [13; 17].

In this paper, we make the following definitions and use them in the following sections.

- **diff:** The removed/added code in a file of a commit. Note that the diff we define here is distinct from the output by the Git diff algorithm, which contains three lines of surrounding code.
- **context:** The surrounding k lines of code before and after the added/removed code.
- **code_before:** The removed code plus the context before the modification.

- **code_after:** The added code plus the context after the modification.

For instance, Figure 5.1 illustrates these definitions using an example of file-level changes aimed at fixing a CVE vulnerability. The added lines are highlighted in green, while the removed lines are highlighted in red, collectively representing the diff. Additionally, the surrounding lines are considered as the context. Specifically, the removed code highlighted in red, along with its context, constitutes the `code_before`, whereas the added code highlighted in green, along with its context, constitutes the `code_after`. Our study focuses on leveraging these code changes at the file level to identify commit-level vulnerability fixes.

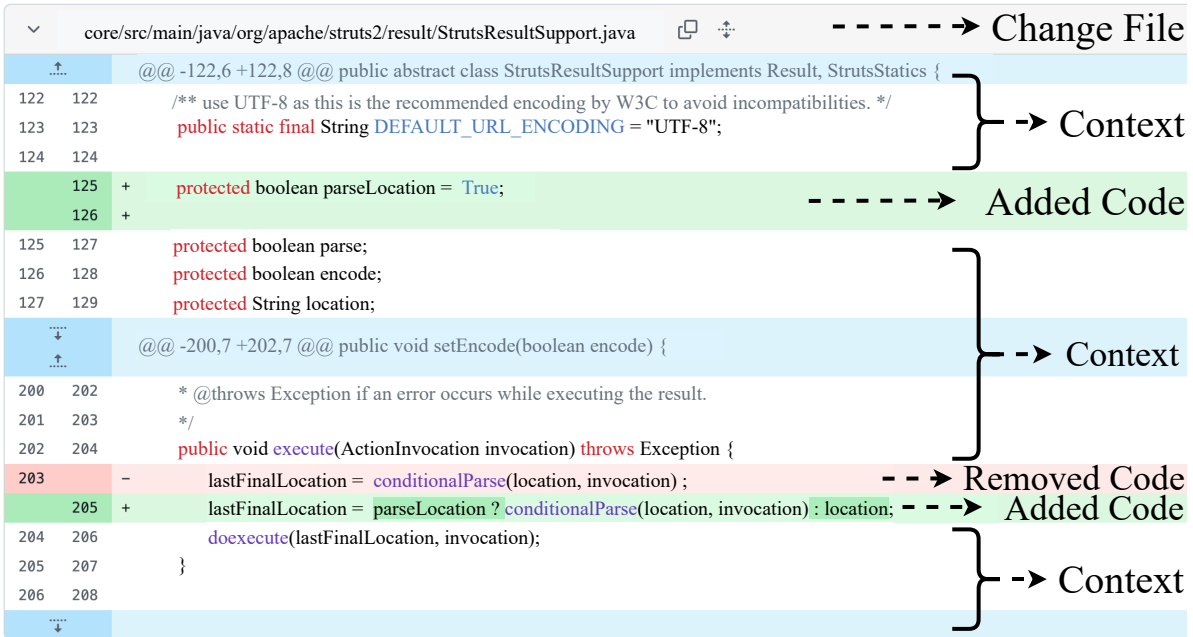


Figure 5.1: A sample file-level code changes for fixing CVE-2018-11776 [1] vulnerability.

5.2.2 Code-Change Representation Learning with Pre-trained Models

Pre-trained language models for code, most notably CODEBERT[67], extend the BERT architecture[107] with training on NL–PL pairs. Their ability to capture both natural-language and programming-language semantics has led to state-of-the-art results in clone detection, code search, program repair, vulnerability detection, and—relevant here—vulnerability fix identification [142; 143; 144; 145; 104].

In prior work on vulnerability-fix detection, pre-trained models are typically applied in one of two ways:

1. Embed the diff (added *and* removed lines concatenated) directly [13; 17]; or
2. Embed the removed and added code separately and then concatenate the resulting vectors [17].

Both strategies require a subsequent classifier to reason over the resulting embeddings, and they do not explicitly highlight the semantic *difference* between `code_before` and `code_after`.

5.3 Methodology

In this section, we introduce the methodology of VFDelta. Figure 5.2 shows the overall framework, which consists of two phases: the file-level vulnerability-fixing classification model training and the commit-level inference. In the training phase, we first extract file-level code changes from each commit and construct context-reserved

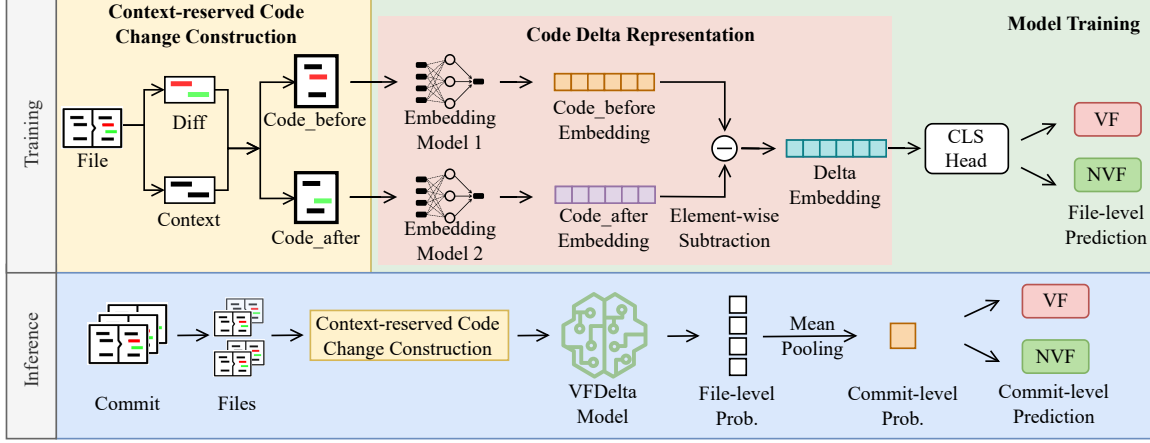


Figure 5.2: The workflow of VFDelta.

code change information, i.e., the *code_before* and *code_after*. Our model, VFDelta, is then trained to predict file-level vulnerability fixes based on the *code_before* and *code_after*. We introduce a lightweight yet effective approach (i.e., *Code Delta Representation*) for generating file-level code change representation, directly reflecting the difference between the original code and modified code from their latent space. The resultant representation is then fed into the final classification head for training. Since we train the embedding and the classifier simultaneously, we can train the entire model in one phase to optimize model performance. During the inference phase, given a commit, we first perform file-level prediction to obtain the probability of each file, then aggregate the probability across all the files to produce the commit-level prediction result. Below, we elaborate on the details of each phase and component. We also explain the motivation of Code Delta Representation in Section 5.3.3.

5.3.1 File-level vulnerability-fixing classification model training

Context-reserved Code Change Construction

In the data construction phase, we decompose a commit into file level and then extract `code_before` and `code_after` from each file by following previous studies [19; 13]. We consider all files in a VF commit as VF and files in an NVF commit as NVF. Specifically, for each file, we first split the diff into two segments, i.e., code removed and code added. We then extract the surrounding code as context. Context is important for learning the difference between two pieces of code, and has been demonstrated by previous studies [136; 137]. We extract k lines of code before and after the diff. In this study, we empirically set k to 3 as it achieves the best performance. We conducted a sensitivity analysis to evaluate the impact of different context sizes in Section 5.5.2. Note that we select to use the surrounding lines of code as the context since we aim to keep our framework lightweight (e.g., avoid heavy program analysis). Nevertheless, more advanced context extraction approaches, such as program slicing, could be adapted to our framework.

Code Delta Representation

Code Delta Representation takes `code_before` and `code_after` as the input and returns the representation of their difference. Firstly, we use two individual embedding models to embed the input code, one for `code_before` and one for `code_after`. The reason we select to use two independent embedding models is that we aim to amplify and capture the nuance between `code_before` and `code_after` since we increase

the tune-able parameters compared to using only one identical model for embedding (evidenced by results in Section 5.5.2). The reason we select to use two independent embedding models is that each model can specialize in capturing the characteristics of `code_before` and `code_after` without interference from the other’s learning process. Secondly, to capture the essence sometimes nuances of the modifications between `code_before` and `code_after`, we compute the delta between them through element-wise subtraction of their embeddings as shown in Figure 5.2. This design is inspired by word2vec [138], which captures the relationship between two words using element-wise subtraction between the two embeddings. For instance, word2vec captures the relationship between gender in monarchical using subtraction, e.g. $\text{vec}(\text{King}) - \text{vec}(\text{Man}) \simeq \text{vec}(\text{Queen}) - \text{vec}(\text{Women}) = \Delta$, where Δ captures the semantic relationship between the gender and royalty. Similarly, the code changes could be represented as $\Delta = \text{embedding}_1(\text{code_before}) - \text{embedding}_2(\text{code_after})$, in which Δ captures the semantic relationship of code changes. In addition, it is worth noting that our framework is flexible to any code embedding model. This flexibility ensures that our model can leverage the most advanced model available to better learn and understand code changes’ nuances (as shown in Section 5.5.1).

Model training

Upon establishing the representation of code changes Δ , we pass it through a feed-forward neural network leading to a classification head designed to distinguish between VF and NVF files. The entire network is trained on a dataset where each file-level change is labeled as either VF or NVF, thus learning to identify potential

vulnerability fixes based on the characteristics of the code modifications. Note that different from previous approaches[13; 19; 17], in which the model has to be trained in multiple phases (e.g., fine-tune the embedding models and train the final commit-level classification model), we fine-tune embedding models and final classification model in one phase.

5.3.2 Commit-level Inference

Once trained, the model’s inference process is geared towards predicting whether an unseen commit is VF or not. This inference is not merely a file-level analysis but extends to a comprehensive commit-level assessment. To achieve this, we aggregate the predictions for all files within a commit by calculating their mean prediction probability as $\text{Prediction}(\text{Commit}) = \frac{1}{N} \sum_{i=1}^N \text{Prediction}(\text{File}_i)$, as N denotes the number of files in a commit. We consider a commit with a probability larger than 0.5 as VF, otherwise NVF. Although, in theory, we could add and train an extra layer or model to aggregate the file-level prediction probability and learn the weights during the training phase, we decided to use a simple mean pooling since we found it is effective for predicting commit-level vulnerability fix.

5.3.3 Why element-wise subtraction?

Apart from the above-mentioned intuitive example, we would like to elaborate on why we select element-wise subtraction, as previous work [146; 147; 148] have also used element-wise subtraction as one part of their code change representation, although they only used one identical embedding model to embed removed/added

separately, which is different from our architecture. However, none of them explain why. We argue that element-wise subtraction is a special case of concatenation with inductive bias [149], and this induction helps with code change representation learning. Consider two vector representations e_1 and e_2 , corresponding to the embedding of `code_after` and `code_before`, $e_1 - e_2$ represents the element-wise subtraction of the two vectors, the concatenation of two vectors side by side can be represented as $[e_1, e_2]^\top$. Specifically, if we apply the transformation $W[e_1, e_2]^\top$, with W being the associated weight matrix $[w_1, w_2]$ in the model. The operation of element-wise subtraction $e_1 - e_2$ can be obtained via the transformation $W_0(e_1 - e_2) = [w_0, -w_0][e_1, e_2]^\top$. Here, W_0 represents the weights associated with the element-wise subtraction and is a special case of $[w_1, w_2]$. This perspective shows element-wise subtraction as a special form of concatenation facilitated by a subsequent linear transformation. We use element-wise subtraction by explicitly focusing on differences between two vectors, and it introduces a strong induction that aids the learning process. Models leveraging this heuristic can more effectively learn to discern patterns of code changes and converge faster (see our analysis in Section 5.5.2).

5.4 Experimental Setting

In this section, we present research questions (RQs), datasets, evaluation metrics, our analysis approach for RQs, and implementation details.

5.4.1 Research Questions

We evaluate VFDelta in different aspects to answer the following research questions.

- *RQ1: How effective is VFDelta compared with SOTA approaches?*
- *RQ2: How effective is our code change representation learning?*
- *RQ3: To what extent does the context contribute to the effectiveness of VFDelta?*
- *RQ4: What is the effectiveness of VFDelta in real-world scenario?*

In RQ1, we aim to evaluate the performance of VFDelta by comparing it to current SOTA approaches. We propose a simple yet effective approach to learning the representation of code changes. Therefore, in RQ2, we aim to understand the contribution of the representation learning approach by comparing it to various variants. In RQ3, we aim to investigate the impact of context on VFDelta. Lastly, in RQ4, we aim to conduct a rigorous evaluation of VFDelta in a challenging real-world scenario, i.e., using the historical data for training and future data for evaluation.

5.4.2 Dataset preparation

VFM2021 dataset

To evaluate the effectiveness of VFDelta in identifying silent vulnerability fixes, we used the dataset containing Java and Python data proposed by recent work VulFixMiner [13], short as VFM2021. VFM2021 dataset includes fixes for vulnerabilities disclosed by February 07, 2021. It contains 1,474 vulnerability-fixing commits from

Table 5.1: Statistics of the VFM2021 dataset (cross-project setting) for both Java and Python.

VFM2021 Dataset — Java									
	Training Set			Validation Set			Testing Set		
	V.F.	N.V.F.Proj.		V.F.	N.V.F.Proj.		V.F.	N.V.F.Proj.	
File	1 889	72 635	117	163	7 564	115	596	207 979	29
Commit	796	30 120	117	135	6 304	115	242	87 457	29

VFM2021 Dataset — Python									
	Training Set			Validation Set			Testing Set		
	V.F.	N.V.F.Proj.		V.F.	N.V.F.Proj.		V.F.	N.V.F.Proj.	
File	747	27 737	84	83	3 082	83	254	72 752	22
Commit	522	20 362	84	80	2 949	83	195	55 638	22

150 Java repositories and 797 vulnerability-fixing commits from 107 Python repositories. A key feature of the VFM2021 dataset is its project-level separation, ensuring no repository appears in both the training/validation/testing sets. Furthermore, to mitigate the challenge of label imbalance, the non-vulnerability fixing (NVF) commits in the training and validation sets were down-sampled. We evaluate RQ1 - RQ3 on VFM2021.

The original VFM2021 dataset itself only contains added code and removed code (i.e., diff), while our approach requires the surrounding code as the context to achieve better performance. Therefore, we updated the VFM2021 to include more surrounding context for each piece of code change. To do so, we cloned the repositories and

gathered all commit information via Pydriller¹, which is a framework that helps developers for mining software repositories. Any commits untraceable by Pydriller or repositories that are no longer accessible on GitHub were omitted. Table 5.1 shows the statistics of the VFM2021 dataset.

Enhanced temporal VFM2023 dataset

To answer RQ4, we need to construct a dataset that is split chronologically. To do so, we extend the original VFM2021 Java dataset to December 2023. First, we re-divided the entire VFM2021 Java dataset according to time. Then, we split it into a training set with the first 90% VF commits and a validation set with the last 10% VF commits. To form the test set, we looked for Vulnerability Fixes (VFs) after 2021. First, we collected all commits between 2021 to 2023 from the VFM2021's Java Open Source Software (OSS) repositories. We then cross-referenced these commits with records in the National Vulnerability Database (NVD) and Snyk Vulnerability Database from 2021 to 2023 to identify VF commits by following previous study [13]. From this process, we found 527 VFs in 52 repositories. To ensure dataset quality and proper evaluation of the model on adequately represented OSS repositories, we removed repositories with less than 10 VFs in either the training or testing set. Ultimately, the refined dataset comprised 9 repositories with 285 VFs and 31,374 Non-Vulnerable Fixes (NVFs). The detailed statistics of this dataset are presented in Table 5.2.

¹<https://pydriller.readthedocs.io/en/latest/intro.html>

Table 5.2: Statistics of the VFM2023 dataset (cross-project setting) for both Java and Python.

VFM2023 Dataset (Temporal Setting)									
	Training Set			Validation Set			Testing Set		
	(2012/06 - 2019/12)			(2019/12 - 2021/01)			(2021/02 - 2023/12)		
	V.F.	N.V.F	Proj.	V.F.	N.V.F	Proj.	V.F.	N.V.F	Proj.
File	2407	93908	145	241	8119	127	892	87138	9
Commit	980	37783	145	109	3164	127	285	31374	9

5.4.3 Evaluation metrics

We evaluate the effectiveness of VFDelta from two perspectives: classification effectiveness and effort-awareness. We use F1-Score(short as F1), precision, and recall as the classification effectiveness metric. Those metrics are widely used in classification tasks [104; 150]. Second, to measure the effort-awareness of our approach, we select CostEffort@L by following previous studies [17; 13; 19]. Given the fact that vulnerability fixes are rare among the code commits (i.e., the extremely imbalanced dataset scenario), the goal of VFDelta is to help users reduce the efforts in identifying vulnerability fixes from an enormous scope of code commits. CostEffort@L is the proportion of inspected vulnerability fixes among all the actual vulnerability fixes when $L\%$ lines of source code in all commits are inspected and is computed as $\frac{n}{N}$, where n is the correctly identified vulnerability fixes, N denotes for the total vulnerability fixes, and L accounts for $L\%$ of total LOC of commits.

5.4.4 Code embedding models

To embed code, we select two widely used pre-trained models CodeBERT [67] and StarEncoder [151] that are trained on source code and designed for source code-related tasks. We select CodeBERT since it is also used in VulFixMiner [13] and MiDas [17] as the model for code embedding so that we can compare with those baselines fairly. We select StarEncoder as it is a more recent and advanced model. However, both models have the same model parameters - 125M. We intend to investigate whether using a more advanced model could improve the effectiveness of VFDelta.

5.4.5 Implementation details

We begin by downloading the official pre-trained model checkpoint for all evaluated models from HuggingFace. Subsequently, the PyTorch and Transformers packages are employed to perform all the experiments. We use 5e-05 as the training rate and AdamW as the optimizer, and we train the model for 10 epochs with a batch size of 128. We use bfloat16 (brain floating point) mixed precision and all experiments are done in Python 3.10. Experiments were conducted on a Linux server equipped with four Nvidia RTX 3090 GPUs, a 24-core CPU, and 128 GB of RAM.

5.4.6 Approach for RQs

Approach of RQ1

To demonstrate the effectiveness of VFDelta, we considered three recent SOTA baselines: VulFixMiner [13], CoLeFunDa [19] and MiDas [17] on the VFM2021 datasets

in terms of all evaluation metrics. We obtain these models from the original authors and use them according to guidance from the authors and or replication packages. Due to its design, CoLeFunda’s implementation is limited to the Java language, therefore, we only compare CoLeFunda with VFDelta on Java data. As we discussed in Section 5.4.4, we use two code embedding models, thereby we refer to the version of implementation that employs CodeBERT as to $VFDelta_{Bert}$ and the version that uses StarEncoder as to $VFDelta_{Star}$.

Approach of RQ2

In this RQ, we aim to evaluate the contribution of Code Delta Representation to VFDelta. More specifically, we first aim to elucidate the contributions of 1) using two independent embedding models for embedding `code_before` and `code_after`, and 2) adapting element-wise subtraction for embedded representation. Hence, we compare our approach VFDelta with the following two variants with different code change representation learning designs to understand the contribution of the two designs. We elaborate on the details of each variant below.

EmbedSubtract_Single In this variant, we retain the element-wise subtraction of embeddings but process both `code_before` and `code_after` through a single embedding model. A similar design was used by CCRRep [148] to capture code change representation. This experiment aims to quantify the impact of using separate embedding models for each piece of code, allowing us to understand if the nuances and differences between `code_before` and `code_after` can be captured and represented more effectively using two independent embedding models compared with one.

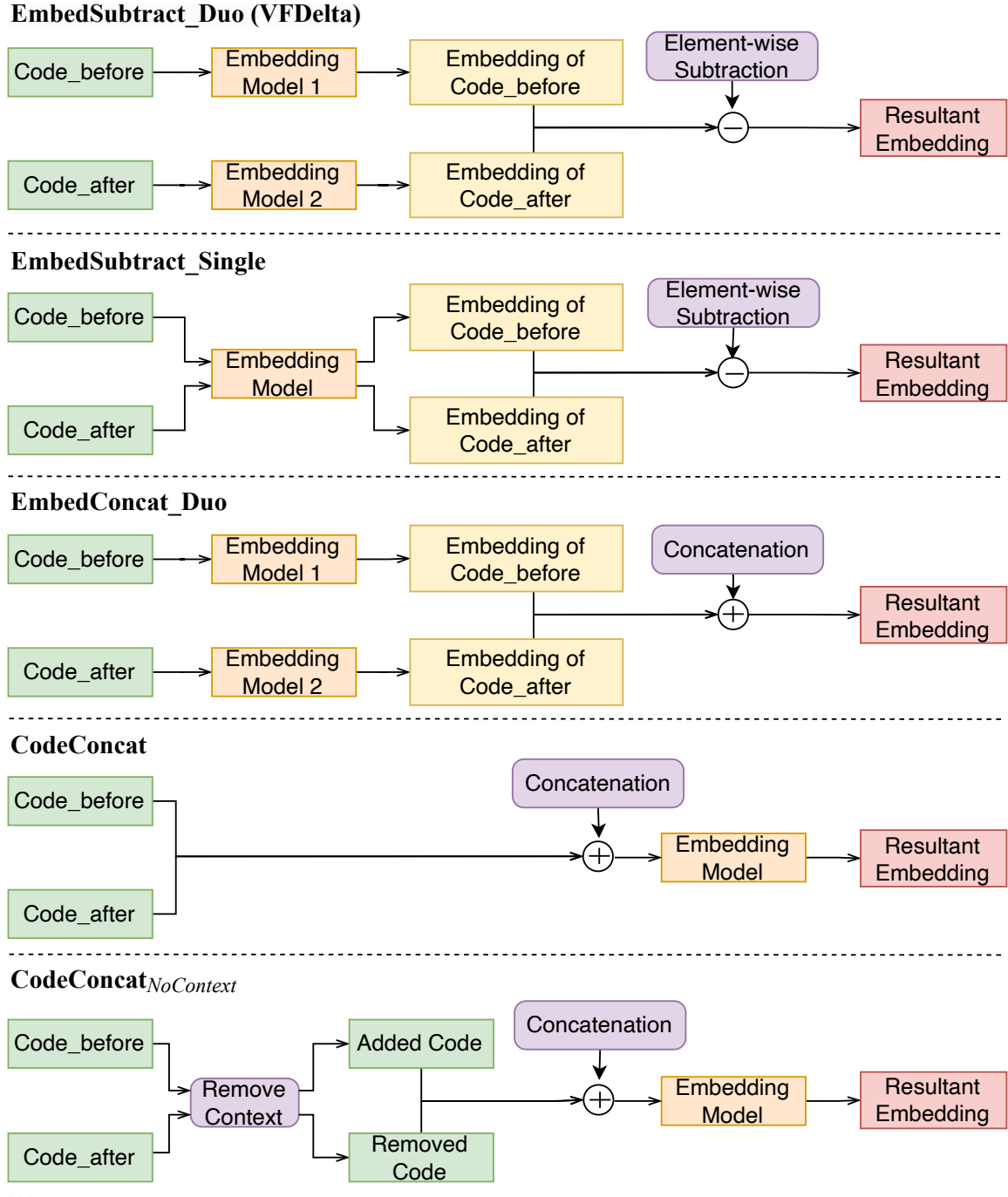


Figure 5.3: Comparison of different code change representation learning approaches.

EmbedConcat_Duo In this variant, we replace the element-wise subtraction operation with concatenation, in which we concatenate the outputs of the two embedding models. This variant assesses the importance of element-wise subtraction in capturing the relational dynamics between `code_before` and `code_after`, providing insights into how different embedding combination methods affect model performance.

Second, we are interested in investigating the impact of performing operations (i.e., concatenation and subtraction) on the raw code and embedding (before and after the embedding).

CodeConcat In this variant, raw `code_before` and `code_after` are concatenated with a separate token (i.e., [`code_before`, `sep`, `code_after`]) before being fed into the embedding model.

CodeConcat_{NoContext} In this variant, due to the subtraction on the raw code, the context of `code_before` and `code_after` are discarded. Therefore, only the changed code (i.e., [`removed code`, `sep`, `added code`]) will be retained. This design was adopted by previous studies MiDas [17] and Vulfixminer [13]. This variant allows us to compare with the code change representation of previous approaches directly.

RawGitDiff In this variant, we do not conduct any pre-processing to the code diff generated by the Git Diff algorithm, the input is represented as a sequence [3 lines of context before, added code, removed code, 3 lines of context after]. We use it directly as input to the embedding model. This design simulates the most simple approaches to represent code change.

Figure 5.3 provides the comparison of our approach with the studied variants. Note that since VFDelta employs two individual embedding models and subtraction,

therefore we name it EmbedSubtract_Duo in the Figure.

In addition, to better understand the capacity of VFDelta, particularly in capturing small code changes in a commit. We measure the effectiveness of each studied representation approach on commits with different sizes of code changes (i.e., number of added/removed lines of code). We group the commits into the following buckets (1,20], (20, 40], (40,60], (60, 80], (80, 100], and (100,∞).

In this RQ, we focus on CodeBERT as the base model on the VFM2021 Java dataset, and keep the same setting as RQ1.

Approach of RQ3

In this RQ, we aim to investigate the impact of the window size of context on the effectiveness of VFDelta. We set context size to 0, 1, 3, 5, 7, 9. We re-train our models with different context sizes and evaluate their performance accordingly. To accommodate as much context as possible within the token limit of the pre-trained model, we evaluate VFDelta with StarEncoder, which supports up to 1024 tokens input. Same as RQ2, we use the VFM2021 Java adataset, and We keep all other experimental settings the same as RQ1.

Approach of RQ4

In the previous RQs, we assessed the performance of VFDelta utilizing the VFM2021 dataset. This dataset is organized based on open-source software (OSS) projects (cross-project setting), which may not adequately simulate real-world scenarios where the emergence of vulnerabilities (VFs) is inherently unpredictable. To address this limitation and more accurately reflect real-world application scenarios, this RQ aims

to examine the efficacy of VFDelta in navigating the complexities of temporal data. Specifically, this RQ uses the VFM2023 temporal dataset, which was crafted to represent a real-world challenge where the model is trained on historical data to forecast future vulnerability fixes (VF). We compare VFDelta with the baselines (i.e., VulFixMiner and MiDas) to provide a comprehensive understanding of their performance in predicting emergent vulnerabilities within the evolving landscape of OSS projects. Similar to RQ1, we compare both the VFDelta_{Bert} and VFDelta_{Star} with other baseline models.

5.5 Results

5.5.1 Results of RQ1

Table 5.3: Comparison of VFDelta with baselines on the **Java** split of the VFM_2021 dataset. We highlight the cells with the best performance in bold.

Method	F1	Precision	Recall	CostEffort@5	CostEffort@20
VulFixMiner	0.186	0.113	0.540	0.583	0.619
MiDas	0.120	0.072	0.240	0.484	0.723
CoLeFunDa	0.062	0.033	0.533	0.590	0.721
VFDelta _{Bert}	0.316	0.262	0.360	0.616	0.744
VFDelta _{Star}	0.330	0.285	0.392	0.632	0.760

VFDelta_{Bert} improves over the state-of-the-art (SOTA) baselines by at least 69.9%, 5.7%, and 2.9% in terms of F1, CostEffort@5, and CostEf-

Table 5.4: Comparison of VFDelta with baselines on the **Python** split of the VFM_2021 dataset. We highlight the cells with the best performance in bold.

Method	F1	Precision	Recall	CostEffort@5	CostEffort@20
VulFixMiner	0.085	0.085	0.077	0.318	0.564
MiDas	0.085	0.088	0.082	0.471	0.805
VFDelta _{Bert}	0.144	0.129	0.164	0.533	0.821
VFDelta _{Star}	0.194	0.212	0.179	0.605	0.856

fort@20 on the Java split. On the Python split, VFDelta_{Bert} achieves improvements of at least 69.4%, 13.2%, and 1.98% in F1, CostEffort@5, and CostEffort@20, respectively. Table 6.1 provides the detailed performance comparison on Java split, while Table 5.4 reports the Python results. First, for a fair comparison, in the Java data split, we compare VFDelta_{Bert} with MiDas, CoLeFunda, and VulFixMiner, which all use codeBert as their embedding model, VFDelta_{Bert} achieves an F1 of 0.316, and outperforms VulFixMiner (0.186), CoLeFunDa (0.062) and MiDas (0.120) with an improvement of 69.9% , 409.7% and 163.3%, respectively. This better F1 achieved by VFDelta_{Bert} is attributed to the better precision, although we have a worse recall compared with VulFixMiner, which suggests our approach can reduce false positives significantly. When examining cost-efficiency metrics, VFDelta_{Bert} improves CostEffort@5 to 0.616 compared to VulFixMiner’s 0.583, CoLeFunDa’s 0.590, and MiDas’s 0.484, achieving an improvement of 5.7%, 4.4%, and 26.5%, respectively. Similarly, VFDelta_{Bert} achieves an improvement of 20.2%, 3.2%, and 2.9% over VulFixMiner, CoLeFunDa and MiDas, respectively, in terms of CostEffort@20.

In the Python data split, the same trend appears. F1, Precision and recall both surge. With CodeBERT, precision climbs from 0.088 (MiDas) and 0.085 (VulFixMiner) to 0.129, while recall rises from 0.082 (MiDas) to 0.164 and F1 jump from 0.085 to 0.144, a 69.4% improvement. CostEffort@5 also increases by 13.2% over MiDas (and by 67.6% over VulFixMiner), and CostEffort@20 edges up to 0.821, the best overall.

Note that VFDelta achieves much better precision while sacrificing some recall in the Java split, typically compared with VulFixMiner. Higher precision indicates the reduction of false positives. Note that in real-world scenarios, reducing false positives is important as developers may lose confidence in using vulnerability detection tools for auditing if the tools produce many false positives [152; 153].

VFDelta_{Star} achieves the best performance among all baselines including VFDelta_{Bert}, on both languages, benefiting from its more advanced embedding model. For Java data split, VFDelta_{Star}, which uses the StarEncoder as the embedding model achieves better performance than VFDelta_{Bert}. VFDelta_{Star} improve VFDelta_{Bert}'s F1, CostEffort@5, and CostEffort@20 from 0.316, 0.616, and 0.744 to 0.33, 0.632, and 0.76, typically achieve a 4.4% improvement in terms of F1. If we compare VFDelta_{Star} with MiDas, CoLeFunDa and VulFixMiner, VFDelta can improve the baselines at least by 77.4%, 7.1%, and 5.1% in terms of F1, CostEffort@5, and CostEffort@20. For Python the jump is even larger: F1 rises from 0.085 to 0.194, a 128% improvement—while CostEffort@5 climbs from 0.471 to 0.605 (↑ 28.4%) and CostEffort@20 from 0.805 to 0.856 (↑ 6.3%) when comparing with the best baseline MiDas.

VFDelta_{Star} demonstrates potential benefits from utilizing more advanced embedding models. StarEncoder [151] and CodeBERT [67] share the same BERT-style encoder architecture with a similar model size. However, StarEncoder supports a longer input window (1,024 tokens vs. 512), allowing it to better capture long code contexts. More importantly, it is trained on a much larger and more diverse dataset (The Stack [154]), covering 300+ programming languages. This extensive training enables StarEncoder to generalize better and outperform CodeBERT across most code-related benchmarks. The results indicate that while foundational models like CodeBert provide a solid base, exploring and incorporating cutting-edge models can lead to further notable performance improvements.

VFDelta improves SOTA baselines significantly in both Java and Python. For instance, VFDelta_{Star} achieves an F1 of 0.33 and 0.194 on the respective splits, with corresponding CostEffort@5 scores of 0.632 and 0.605, with an improvement of at least 77.4 % (Java) and 128 % (Python) in F1 over the strongest baselines.

5.5.2 Results of RQ2

Table 5.5: Comparison of VFDelta and studied variants on VFM2021.

Variant	F1	Precision	Recall	CostEffort@5	CostEffort@20
EmbedConcat_Duo	0.226 (28.5% ↓)	0.155(45.0% ↓)	0.410(13.9% ↑)	0.591(4.1% ↓)	0.682(8.3% ↓)
EmbedSubtract_Single	0.219 (30.7% ↓)	0.149(47.2% ↓)	0.409(13.6% ↑)	0.562(8.8% ↓)	0.661(11.2% ↓)
CodeConcat	0.221 (30.1% ↓)	0.149(47.2% ↓)	0.430 (19.4% ↑)	0.566(8.1% ↓)	0.690(7.3% ↓)
CodeConcat _{NoContext}	0.236 (25.3% ↓)	0.166(41.1% ↓)	0.413(14.7% ↑)	0.566(8.1% ↓)	0.678(8.9% ↓)
RawGitDiff	0.216 (31.6% ↓)	0.144(48.9% ↓)	0.430 (19.4% ↑)	0.570(7.5% ↓)	0.711(4.4% ↓)
EmbedSubtract_Duo (VFDelta)	0.316	0.282	0.360	0.616	0.744

Using independent embedding models for `code_before` and `code_after` and element-wise subtraction to capture the difference between two embeddings as a combination plays a critical role in our approach. Table 6.2 summarizes the results of each variant. Compared to VFDelta, EmbedConcat_Duo, which concatenates embedding outputs instead of element-wise subtraction, has a 28.5% decrease in F1 score, 4.1% drop in CostEffort@5, and 8.3% drop in CostEffort@20. This result demonstrates using the subtraction operation to capture the differences between two embeddings is more effective than concatenation. As discussed in Section 6.3, subtraction is the special case of concatenation, using element-wise subtraction in the model provides a direct induction to the model to capture the difference in such a way and facilitate the learning process. This is evidenced in Figure 5.4, where we can observe that the training loss of VFDelta drops faster than EmbedConcat_Duo and EmbedSubtract_Single. EmbedSubtract_Single processes both pieces of code through the identical embedding model but retains element-wise subtraction. This variant results in a 30.7% decrease in F1, 8.8% drop in CostEffort@5, and 11.2% in CostEffort@20 compared to VFDelta, which demonstrates the importance of using two independent embedding models for code representations. More importantly, as we can see, only using one strategy does not improve the performance much compared with other variants. Two individual embedding models and element-wise subtraction need to be used as a combination to optimize their effectiveness.

When comparing the two families of variants that apply operations (e.g., subtraction and concatenation) before and after embedding models, we find that they have similar performance in all metrics. CodeConcat, CodeConcat_{NoContext}, and RawGitD-

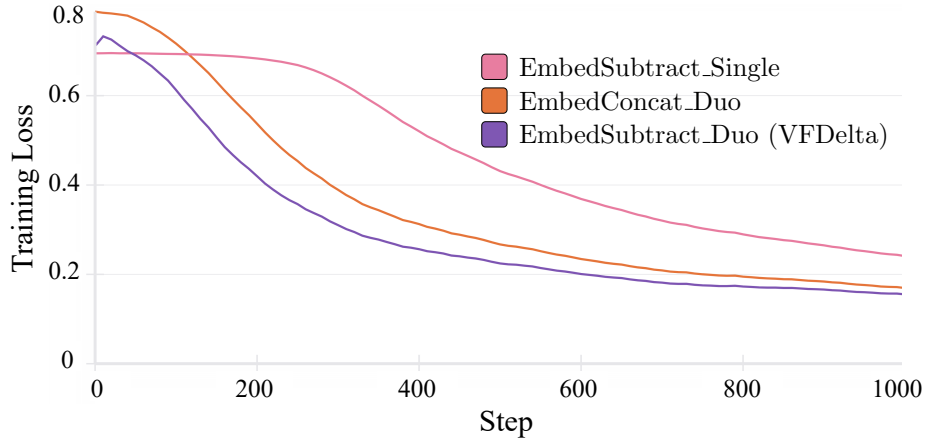


Figure 5.4: Comparison of training loss between VFDelta, EmbedSubtract_Single and EmbedConcat_Duo.

iff share similar performance as EmbedConcat_Duo and EmbedSubtract_Single, which indicates that capturing differences in raw code and embeddings have similar effectiveness for the VF identification without a strong induction. Also, the performance of CodeConcat and CodeConcat_{NoContext} shows that providing a context within the raw code level is not beneficial.

Our approach of learning code change representation is more effective in capturing small changes than other studied variants and achieves better performance in identifying VF with small changes.

Figure 5.5 presents the performance (F1) of the studied variants in the groups of commits with different sizes of code changes, and the data distribution of each group. As observed, VFDelta outperforms other approaches when the code change is small (i.e., LOC is 0 -60). It is worth noting that the majority of the commits (72%) fall into this range, and our approach performs better for the majority of commits. This analysis provides evidence that our design is more effective in capturing small

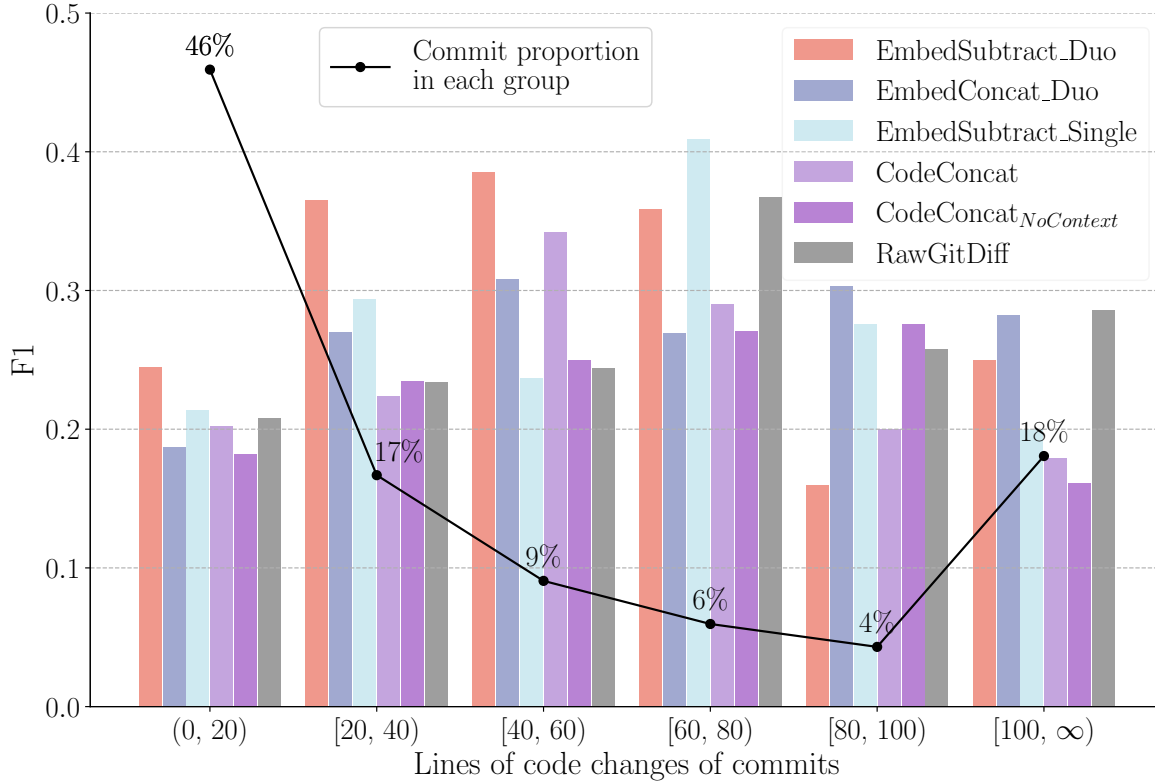


Figure 5.5: The barplot denotes the effectiveness of different approaches on the groups of commits with different change sizes in terms of F1. The black line denotes the proportion of each group of commits. For instance, 46% indicates 46% of the commits have a code change size between 0 and 20.

code changes and leads to better performance in identifying VFs with small changes, typically when the changed LOC is less than 60.

Compared to other variants, our design improves precision by a large margin and reduces false positives. As we can observe, VFDelta achieves the best F1, precision, CostEffort@5, and CostEffort@20, which indicates that VFDelta introduces fewer false positives. One explanation is that using two independent embedding models enables our approach to better capture the nuances between VF and NVF. This is evidenced by the best performance of EmbedConcat_Duo among the variants (excluding VFDelta_{Bert}) typically for the small changes as shown in Fig-

ure 5.5, which is the only one using two embedding models to represent `code_before` and `code_after`.

java/org/apache/catalina/real/RealmBase.java		
...	@@ -546,9 +546,9 @@	public void backgroundProcess() {
546	546	
547	547	// Check each defined security constraint
548	548	String uri = request.getRequestPathMB().toString();
549	-	// Bug47080 - in rare cases this may be null
549	+	// Bug47080 - in rare cases this may be null or ""
550	550	// Mapper treats as '' do the same to prevent NPE
551	-	if (uri == null) {
551	+	if (uri == null) uri.length() == 0) {
552	552	uri = "";
553	553	}
554	554	
...	@@ -580,7 +580,8 @@	public void backgroundProcess() {
580	580	}
581	581	
582	582	for(int k=0; k < patterns.length; k++) {
583	-	if (uri.equals(patterns[k])) {
583	+	// Exact match including special case for the context root.
584	+	if (uri.equals(patterns[k]) patterns[k].length() == 0 && uri.equals("")) {
584	585	found = true;
585	586	if (collection[j].findMethod(method)) {
586	587	if (results == null) {
...		

Figure 5.6: Vulnerability fix for CVE-2018-1304 with minor changes that EmbedSubtract_Duo (VFDelta) successfully detected while other variant failed.

Figure 5.6 presents an example commit where the vulnerability was fixed from Apache tomcat (CVE-2018-1304), as we can observe, only three conditional checks are changed, yet the change removes an off-by-one path that could leave the application unsecured when `uri` is the empty string. Our embedding design can detect this change while other embedding variant designs fail in capturing small code changes.

To understand why our designs outperform other variants, we compare the distance by using different embeddings. With EmbedSubtract_Duo, the Euclidean dis-

tance between the embedding of `code_before` and `code_after` is 16.61, and the cosine similarity is -0.02 . For comparison, the `EmbedConcat_Duo` sees the fragments as much closer (distance 6.58, cosine 0.82), and the `EmbedSubtract_Single` variant falls in between (distance 9.63, cosine 0.53). VFDelta pushes the two embeddings almost orthogonal ($\cos \approx 0$) and maximises Euclidean distance. In contrast, other variants view the fragments as highly similar ($\cos \geq 0.5$). This example illustrates the intuition behind our design and echoes the quantitative difference observed in Table 6.2.

Code Delta Representation plays a critical role in our approach and is typically effective in capturing small code changes.

5.5.3 Results of RQ3

Table 5.6: Performance of VFDelta_{Star} under different context windows on VFM_2021.

Window size	F1	Precision	Recall	CostEffort@5	CostEffort@20
0	0.280	0.214	0.405	0.603	0.727
1	0.295	0.254	0.351	0.541	0.752
3	0.330	0.285	0.392	0.632	0.760
5	0.293	0.243	0.368	0.558	0.744
7	0.268	0.210	0.368	0.579	0.748
9	0.264	0.206	0.368	0.545	0.744

VFDelta achieves the optimal performance with a 3-line context win-

dow. Table 5.6 illustrates the performance of our approach using varying numbers of surrounding code lines as context. We note that the effectiveness of VFDelta, indicated by F1 score, precision, recall, and cost-effort metrics, fluctuates with different window sizes. Remarkably, the model achieves its peak performance with a 3-line context window. Subsequently, the performance of VFDelta diminishes as the number of lines increases. This trend suggests that inadequate context may hinder accurate vulnerability detection, whereas excessive context might introduce noise and divert the model’s attention from pertinent changes. This observation underscores the critical role of context in refining the model’s capability to identify vulnerability fixes precisely.

It is worth noting that even without context, VFDelta outperforms the baselines (i.e., MiDas and VulFixMiner) in terms of F1, CostEffort@5, and CostEffort@20, which indicates the superiority of our framework.

Too little or much context hinders the performance. VFDelta achieves the optimal performance with a 3-line context window.

5.5.4 Results of RQ4

VFDelta significantly outperforms the SOTA baselines VulFixMiner and MiDas across all metrics in this temporal dataset setting.. For instance, VFDelta_{Star} achieves an F1 of 0.199, demonstrating a notably higher prediction performance than MiDas (0.121), CoLeFunDa (0.080) and VulFixMiner (0.053), with an improvement of 64.5%, 148.8% and 275.5%, respectively. For CostEffort@5 and CostEffort@20, we observe a similar trend. VFDelta_{Star} improves MiDas which outperforms VulFixMiner and CoLeFunDa, from 0.228 to 0.442 for CostEffort@5 and

Table 5.7: Comparison of VFDelta and baselines on VFM_2023 dataset.

Method	F1	Precision	Recall	CostEffort@5	CostEffort@20
VulFixMiner	0.053	0.058	0.049	0.133	0.228
MiDas	0.121	0.103	0.147	0.228	0.589
CoLeFunDa	0.080	0.052	0.168	0.218	0.575
VFDelta _{Bert}	0.158	0.162	0.154	0.351	0.540
VFDelta _{Star}	0.199	0.165	0.256	0.442	0.618

0.589 to 0.613 for CostEffort@20, with an improvement of 93.9% and 4.08%. Effort-awareness metrics are particularly important in the real-world scenario under the constraints of limited resources or inspection capacity. Our observations demonstrate the efficiency of VFDelta in real-world scenarios.

For instance, VFDelta successfully detects a silent vulnerability fix commit while other baseline failed to identify. The commit was a silent vulnerability fix for CVE-2022-46363, as shown in Figure 5.7. The vulnerability was initially reported to Apache by Qihoo 360 adlab. Then, it was silently fixed without any indication of the vulnerability fix in the commit message on Nov 24, 2022, following the CVD procedure before it was revealed and published on NVD on Dec 13, 2022.

VFDelta significantly outperforms the SOTA baselines VulFixMiner, CoLeFunDa and MiDas across all metrics in this temporal dataset setting. For instance, VFDelta_{Star} improves the baselines at least by 64.5% and 93.9%, in terms of F1 and CostEffort@5, respectively.

[Before Silent Fix] Report to Apache by: Qihoo 360 adlab (finder)
[Nov24, 2023] Code Change Snippet: <pre> if ((dispatcherServletPath != null dispatcherServletName != null) — && (redirectList != null && matchPath(redirectList, request) + && (redirectList != null && matchPath(redirectQueryCheck, redirectList, request) redirectList == null)) { ... boolean staticResourcesMatch = staticResourcesList != null — && matchPath(staticResourcesList, request); + && matchPath(false, staticResourcesList, request); boolean staticWelcomeFileMatch = staticWelcomeFile != null ... — private boolean matchPath(List<Pattern> values, HttpServletRequest request) { private static boolean matchPath(boolean checkRedirect, List<Pattern> + values, HttpServletRequest request) { ... — if (redirectQueryCheck) { + if (checkRedirect) { String queryString = request.getQueryString(); </pre>
[Nov24, 2023] Commit Message: Don't use redirect-query-check with the static resource list (#1034)
[Dec 13, 2023] Report to NVD and Disclosed: CVE-2022-46353

Figure 5.7: A Silent Vulnerability Fix for CVE-2022-46363 [2] detected by VFDelta.

5.6 Discussion

5.6.1 Scalability and computational cost

VFDelta is intentionally lightweight, both for training and inference. Using of BF16 mixed-precision, training on the VFM2021 Java training set (≈ 30 k commits) can be done within 4 GPU-hours on a single NVIDIA RTX 3090 (24 GB). At inference, the model classifies a commit in ≈ 30 ms, i.e., roughly 33, 000 commits per hour,

allowing one GPU server to monitor hundreds of active OSS repositories in real time. Because the framework treats the embedding model as a plug-in component, any newer or larger pre-trained model can be used when additional capacity is available, while the overall training and inference costs remain within practical bounds.

5.6.2 Precision and Recall trade-off in practice

Throughout the experiment, we adopted the conventional 0.5 decision threshold to keep comparisons with earlier work fair. With this cut-off, VFDelta trades a modest recall decrease for a substantial precision increase, resulting in leading F1 scores. In practice, the vulnerability-fix detection tool is the first stage in a larger human-in-the-loop pipeline: security experts still need to examine the commits flagged by the tool. Their daily review budget is small relative to the volume of OSS activity, making it impossible to inspect all the flagged commits. Therefore, tools with low precision carry a high opportunity cost and erode trust in the tool [155; 16]. Compared to the existing baselines, VFDelta achieves much higher precision, mitigating this problem.

5.6.3 Real World implementation of VFDelta

In RQ4, we evaluated VFDelta on the VFM2023 temporal dataset. Although VFDelta outperformed existing approaches, we observed that all methods, including ours, exhibited low precision. This reflects the inherent difficulty of the task in both industry and academia due to the highly imbalanced nature of the data. While manual code review is still necessary to verify the suggestions made by our tool, the development of such tools remains crucial in reducing the overall workload for

security experts, who cannot realistically review every commit across all repositories. Considering more practical metrics like CostEffort, which reflect real-world usage scenarios, VFDelta demonstrates significant value. For instance, by reviewing only 5% of the retrieved commits, security experts were able to identify over 60% of the vulnerability fixes. Feedback from our industry partners further supports the tool’s utility, as they found the positive rate to be acceptable, requiring only a modest amount of additional manual effort to review the results.

5.6.4 Potential applications of VFDelta on other tasks

While our task is to identify silent vulnerability fixes, our framework is flexible. It can be used for other code change-related work, such as just-in-time defect prediction [156; 146; 148] and commit message generation [146; 148], which rely on capturing the code change information as the fundamental component. For instance, for the commit message generation, our code change representation could be used as the encoder and then connected to the decoder to generate the message. We encourage future research to investigate our code change representation to other tasks.

5.6.5 Ethical implications

Cybersecurity is always an arms race and often a two-edged sword. Different from solutions used by malicious parties, our solution will be open and is created with the intent to protect. Our solution is the same as many other solutions (e.g., fuzzing and automatic exploit generation), which can be used by both malicious parties and OSS users. However, such research is still valuable as it empowers OSS users to defend

against potential attacks.

5.6.6 Threats to Validity

Internal Validity In this study, we use two categories of metrics to evaluate the effectiveness of VFDelta and baselines. Although there might be other metrics that could be used, our selected metrics are commonly used in evaluation in previous studies [13; 17] for the same task. VFDelta has a parameter which is the size of the context window. We selected the window size empirically and investigated their impact in RQ3. In this study, we evaluate the impact of the size of context on one model with a max token limit of 1024. Although our experiment results suggest that using a context window of 3 lines provided the best result, and we observe a decline trend after 3 lines. However, more advanced models that support more tokens might perform better with a larger context window, and we suggest future research to investigate models with larger window sizes. Another threat is that we consider all files in a VF commit as VF when training file-level prediction model, this may not always be the case and introduce bias. To mitigate this, we used a widely used dataset VFM, and demonstrated the superiority of VFDelta over SOTA baselines. Also, VFDelta assumes that a vulnerability fix is within single commit and therefore cannot detect cross-commit fixes.

External Validity Threats to external validity relate to the generalizability of our findings. In this study, we evaluated our proposed framework on one task and demonstrated its superiority in identifying vulnerability fixes over baselines. Future research is encouraged to evaluate our approach to more code change-related tasks.

5.7 Conclusion

In this paper, we propose a simple yet effective framework to identify commit-level vulnerability fixes by enhancing code change representation learning to improve the capacity of our model for capturing small code changes, and simplifying the training process to enable us to train the embedding models and final classification model jointly. Our evaluation shows that VFDelta outperforms SOTA baselines, On the Java split, VFDelta achieves an F1 of 0.33, a CostEffort@5 of 0.632, and a CostEffort@20 of 0.76, with an improvement of 77.4%, 7.1%, and 5.1% over the best baselines, respectively. On the Python split, VFDelta achieve an F1 of 0.194, CostEffort@5 of 0.605, and CostEffort@20 of 0.856, surpassing the best baseline by 128.2%, 28.4%, and 6.3%. Our ablation analysis shows that our Code Delta Representation plays a critical role in VFDelta, and is typically effective in capturing small code changes. Evaluation of the temporal dataset demonstrates the superiority of VFDelta over baselines with a large margin improvement in real-world situation.

Chapter 6

Code Change Intention, Development Artifact, and History Vulnerability: Putting Them Together for Vulnerability Fix Detection by LLM

6.1 Introduction

Software development heavily relies on the use of open-source software (OSS). However, failing to detect and mitigate vulnerabilities in OSS can lead to catastrophic consequences [157]. OSS organizations generally follow the Coordinated Vulnerability Disclosure (CVD) [158] model to disclose vulnerabilities. In this model, details of

vulnerabilities are publicly shared once the developers feel they have had sufficient time for remediation of the security risk. This often causes a delay between when a commit that fixes a vulnerability (*i.e.*, *vulnerability fix commit*) is integrated into the codebase and when the vulnerability and or its fix are publicly announced. The time between fix and disclosure can provide malicious users a window of opportunity to find details of vulnerabilities and exploit them in software dependent on the OSS. Although CVD recommends applying fixes silently to avoid leaking sensitive information about the vulnerability, the risk of exploitation remains. Hence, OSS users have a crucial incentive to monitor integrated OSS and discover vulnerability fixes to begin the remediation process as fast as possible (*i.e.*, *vulnerability fix detection*). Vulnerability fix detection is a complex task and poses a significant challenge for general OSS organizations without an automated approach. Furthermore, manually monitoring all commits across integrated OSS is highly time-consuming and costly. To solve this, automated approaches to identify vulnerability fixes have been proposed in prior studies, which typically train a deep learning model using commit-level information. For instance, *VulFixMiner* [15] trains a pre-trained language model (PLM) using code change information from commits. In their next work [19], they proposed an approach to detect vulnerability fixes at a finer granularity by using function-level code change information. However, those approaches only leverage code change information and neglect additional software development artifacts related to commits, making it challenging to identify the intent of code changes without sufficient context, typically for nuanced code changes (*e.g.*, adding condition check). Vulnerability fix is a complex task that is often associated with issue reports [159], but such information

is not adequately utilized in existing methods [15; 19]. The only work that leverages development artifacts (*i.e.*, issue report) is Vulcurator [45], however, it used a voting mechanism without leveraging the semantic association between artifacts. Previous approaches also fail to identify vulnerability fixes within a tangled commit — those with a mixture of code modifications for various purposes (*e.g.*, refactoring) [160]. The limitations of existing approaches mentioned above cause them to miss many true vulnerability fix commits (evidenced by very low recalls (0.06 to 0.26) as shown in Section 6.5.1). More importantly, identifying vulnerability fixes requires specialized security expertise and project-specific domain knowledge [25; 161], prior approaches only provide a prediction, without an explanation behind this decision, which hinders the use of such approaches in practice.

Recently, Large Language Models (LLMs) have demonstrated promising results in code-related tasks, such as code understanding [52; 162] and vulnerability understanding [163; 164]. Intuitively, vulnerability fix detection is a task that requires the abilities of natural language understanding and code comprehension, typically in vulnerability understanding. Hence, the strong natural language and code understanding capabilities of LLMs fit the requirements of this task well.

Therefore, to tackle the problem and overcome existing limitations (*i.e.*, neglecting information in development artifacts, tangled commits, and lack of sufficient explanation) of previous works, we propose *LLM4VFD*, a novel framework that enhances vulnerability fix detection by leveraging multiple sources of information distilled by LLMs, consisting of four components. The Code Change Intention (CCI) component analyzes a commit to extract its summary, purpose, and implications, using Chain-

of-Thought (CoT) reasoning. The Development Artifact (DA) component utilizes information from related development artifacts, such as issue reports (IRs) and pull requests (PRs), to provide additional context and enhance understanding of the commit. The Historical Vulnerability (HV) component retrieves similar vulnerability fix commits from historical vulnerability data to further enhance the context of the commit. Last, the Comprehensive Analysis and Vulnerability Fix Detection (CAVFD)

component combines the information distilled from the previous components to enable In-Context Learning. The synthesized information is used to generate a final prediction, while also providing a detailed explanation in analysis to help security experts understand the rationale behind the decision.

We evaluated LLM4VFD on a newly created multi-language vulnerability fix dataset which we call the BigVulFix dataset. This dataset consists of 1,689 vulnerability fix commits after 2023 collected from the National Vulnerability Database (NVD) [3]. LLM4VFD is evaluated on 6 LLMs spanning 3 LLM families in varying parameter sizes (*i.e.*, 7B–236B). LLM4VFD outperforms PLM-based approach consistently in terms of MCC, F1-score, and recall. For instance, LLM4VFD outperforms the best-performed PLM-based approach VulCurator by 68.1%–145.4% in terms of F1-score when using different LLMs as the base model. Our framework demonstrates performance gains ranging from 12.7%–105.6% over its vanilla variant, with smaller models generally benefiting more compared to their larger counterparts. In addition, the conducted ablation analysis shows that all three components are crucial in contributing to the final performance of LLM4VFD. Through a user study involving security experts, we find in 80.0% of the cases LLM4VFD’s analysis helps security ex-

perts to understand the code change and improve the efficiency in identifying whether the change is a vulnerability fix commit. We also conducted a failure case analysis to understand the limitations of LLM4VFD, and provide insights for future work.

In summary, this paper makes the following contributions:

- To the best of our knowledge, we are the first study to investigate vulnerability fix detection using LLM.
- We propose a novel framework, LLM4VFD, that enhances vulnerability fix detection by leveraging multiple sources of information based on LLMs.
- We conducted extensive evaluation on LLM4VFD, including an ablation study to evaluate the contribution of each component in LLM4VFD and a user study to evaluate the usefulness of LLM4VFD’s analysis result for security experts.
- We release BigVulFixes, a new dataset containing 1,689 vulnerability fix commits from 7 major programming languages after 2023 to avoid data leakage with LLM’s pre-training data.
- We perform a bad case analysis on the limitations of our approach to promote future work on vulnerability fix detection.

6.2 Background

6.2.1 Vulnerability Fix Detection

A typical *vulnerability fix detection* task takes a commit as input and predicts whether the commit fixes a vulnerability. The task is motivated by the coordinated

vulnerability disclosure (CVD) model used in OSS, where a silent time window often separates the integration of a fix from the public disclosure of the vulnerability. Monitoring that silent window is essential: without an automated approach, manually inspecting every incoming commit is prohibitively costly for most organisations. A successful detector therefore needs to reason about *why* a change was made, handle tangled commits that mix security and non-security edits, and provide evidence that security engineers can trust and act upon.

6.2.2 Large Language Models

Large Language Models (LLMs), popularised by the Generative Pre-trained Transformer (GPT) family, have recently shown strong performance in code understanding, generation, and vulnerability-related tasks. Their abilities can be steered by *prompt engineering*—crafting input text that guides the model—and by *in-context learning* (ICL), which includes task demonstrations directly in the prompt. Techniques such as Chain-of-Thought (CoT) prompting further enhance the model’s reasoning by breaking a problem into explicit intermediate steps. Because LLMs are bounded by a finite context window and a fixed pre-training cut-off date, prompt engineering also condenses relevant information and injects up-to-date knowledge that would otherwise be missing.

6.3 Methodology

To address the challenges discussed in Section Section 6.1, we present LLM4VFD, which leverages information that is distilled from multiple sources by leveraging LLMs

to enhance the vulnerability fix detection. Figure 6.1 provides an overview of our framework. LLM4VFD consists of four components: Code Change Intention (CCI), Development Artifact (DA), Historical Vulnerability (HV), and Comprehensive Analysis and Vulnerability Fix Detection (CAVFD).

More specifically, to address Challenge #1, the CCI component leverages LLMs with Chain-of-Thought (CoT) techniques to construct a summary of the commit that focuses on three crucial aspects (*i.e.*, code change summary, purpose of the change, and implications of the change). To address Challenge #2, the DA component leverages the information extracted from related development artifacts (*i.e.*, issue reports (IR) or pull requests (PR) in our case) to provide additional information to enrich the context for a commit. To address Challenge #3, the HV component builds a vector database and retrieves similar vulnerability fix commits from historical vulnerability data. The information from these components is then combined to enhance the analysis in the CAVFD component, where a comprehensive prompt template is employed to guide the LLM through all the relevant aspects of the commit. This structured prompt not only predicts whether the given commit is a vulnerability fix, but more importantly, it generates an in-depth explanation in analysis to assist security experts in understanding the rationale behind the prediction.

As highlighted in Figure 6.1, we provide a running example to illustrate the procedure of LLM4VFD. The outputs are taken from the commit from Challenge #1, which by leveraging our approach, can be correctly identified as a VF commit.

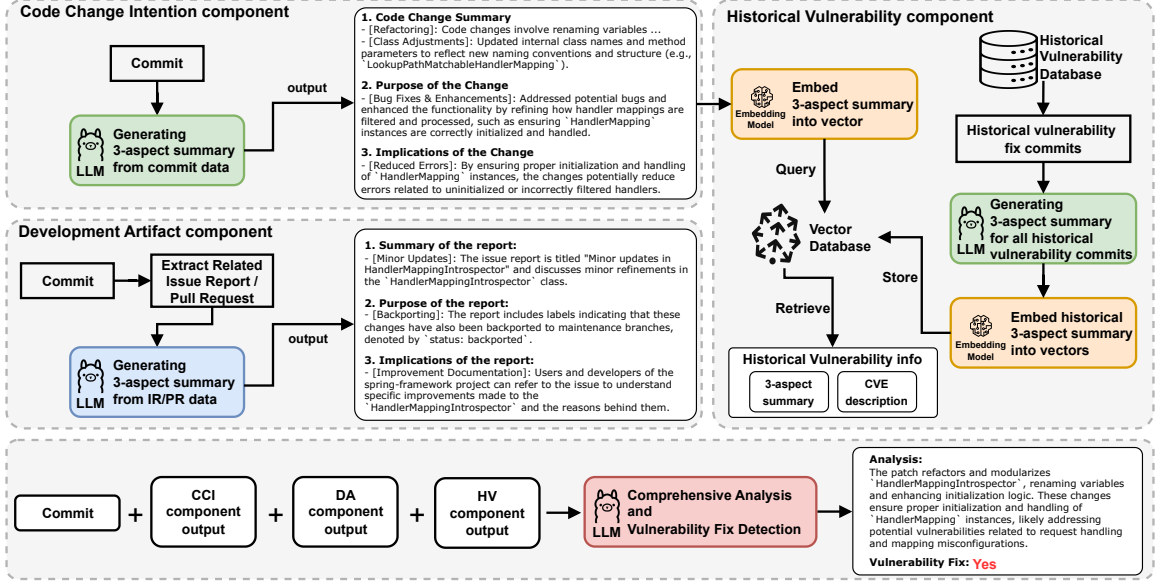


Figure 6.1: The framework of LLM4VFD.

6.3.1 Code Change Intention (CCI)

The Code Change Intention component takes raw commit data (*i.e.*, the code diff and commit message) as input, and outputs a structured summary that abstracts the intention behind the commit. However, obtaining this information is not trivial, as the commit data does not explicitly provide this level of structured reasoning. To generate such information, we leverage and enhance the reasoning ability of LLMs using CoT techniques. Specifically, we design a structured prompt that breaks down the reasoning process into steps, mimicking how a security expert would analyze the commit. The prompt template, as shown in Figure 6.2, guides the LLM to analyze the commit code changes and distill relevant information into three key aspects (referred to as *3-aspect summary*): the summary, the purpose, and the implications as below.

1. **Code Change Summary.** In the first step of the prompt, we instruct the LLM to focus on identifying the primary action of the code change. The goal

here is to abstract the core modification and summarize different kinds of code changes.

2. **Purpose of the Change.** The second step of the prompt instructs the LLM to reason through why the change was made. This step categorizes the commit into broader categories such as refactoring, feature enhancement, or fixing a vulnerability.
3. **Implications of the Change.** In the final step, the prompt asks the LLM to consider the broader consequences of the change, including its potential impact. This step ensures that any security-related modifications are properly captured and that the potential risks or fixes introduced by the commit are fully understood.

In addition, to ease the extraction process for further components, we provide concrete instructions with a demonstration example to ensure LLM outputs the 3-aspect summary in the format we expect. Note that one commit, such as tangled commits, probably has multiple points of summary, purposes, and implications. We instruct LLM to generate multiple points (i.e., Key Point/Optional Key Points) for each aspect if it is applicable.

6.3.2 Development Artifact (DA)

In the DA component, we aim to integrate the information of external development artifacts such as issue reports (IRs) and pull requests (PRs) to enrich the analysis of a given commit. However, these artifacts can be lengthy and include irrelevant details, such as issue report templates and CI/CD notifications. To address this, and inspired

Prompt Template: Code Change Intention

System Prompt: You are a helpful software developer assistant specializing in software development life-cycle to help other developers understand the characteristics of software patches.

User Prompt: You are given the following software patch: **{Commit}**

Think step by step and provide an analysis describing the following characteristics.

1. Code Change Summary
2. Purpose of the Change
3. Implications of the Change

Provide the analysis in bullet point format for each characteristic. Each bullet point should start with a key point and then briefly describe a main idea or fact from the text. Ensure each point is concise and captures the essence of the main idea it's summarizing. Here is an example of the desired format:

1. Code Change Summary
 - [Key Point]: <description>
 - [Optional Key Point]: <description>
2. Purpose of the Change
 - [Key Point]: <description>
 - [Optional Key Point]: <description>
3. Implications of the Change
 - [Key Point]: <description>
 - [Optional Key Point]: <description>

Figure 6.2: The prompt template of Code Change Intention.

by the CCI component in Section 6.3.1, we design a structured prompt that guides the LLMs through the process of analyzing and summarizing the related IRs and PRs to generate a 3-aspect summary which is similar to that for a commit. More specifically,

the DA component generates a concise summary that abstracts the intention behind these artifacts by focusing on three core aspects: the summary, the purpose of the changes, and the potential implications. The prompt template is shown in Figure 6.3.

Prompt Template: IR/PR Summary

System Prompt: You are a helpful software developer assistant specializing in software development lifecycle to help other developers understand characteristics of software components such as patches, issue reports, pull requests, etc.

User Prompt: You are given the following Github issue report title and body information in JSON format which is related to a commit: `{ Commit }`

Think step by step and provide an analysis describing the following characteristics.

1. Summary of the report
2. Purpose of the report
3. Implications of the report

Provide the analysis in bullet point format for each characteristic. Each bullet point should start with a key point and then briefly describe a main idea or fact from the text. Ensure each point is concise and captures the essence of the main idea it's summarizing. Include 1-3 key points. Here is an example of the desired format:

1. Summary of the report:
 - [Key Point]: <description>; - [Optional Key Point]: <description>
2. Purpose of the report:
 - [Key Point]: <description>; - [Optional Key Point]: <description>
3. Implications of the report:
 - [Key Point]: <description>; - [Optional Key Point]: <description>

Figure 6.3: The prompt template of Development Artifact.

6.3.3 Historical Vulnerability (HV)

In this component, we aim to leverage information from historical vulnerability fix commits. More specifically, given a commit, HV aims to retrieve similar vulnerability fix commits to enrich the given commit. As discussed in Section 6.1, the vulnerability fix commit can be multi-purpose and contain code addressing other issues. Therefore, we decide to retrieve similar vulnerability fix commits based on their intention. To do this, we need to construct a database of historical vulnerability fixes. First, we collect historical vulnerability fix commits from existing vulnerability databases such as NVD [3]. We generate the 3-aspect summary for all the collected vulnerability fix commits using the CCI component. We then embed and vectorize the generated three-aspect summary for each vulnerability fix commit using a sentence embedding model [165], and store them in a vector database. Alongside the generated summaries, we also store their corresponding vulnerability metadata, including the CVE ID and CVE description.

When processing a new commit, we begin by using the CCI component to obtain a 3-aspect summary for the commit. We search for the nearest instance from historical vulnerability database, based on the similarity of their 3-aspect summary. With the nearest instance, we gather the 3-aspect summary and CVE description from its metadata as the output of the HV component.

6.3.4 Comprehensive Analysis and Vulnerability Fix Detection (CAVFD)

In the final step, we combine multiple sources of output collected from the three components (*i.e.*, CCI, DA, and HV), together with the commit's code change and commit message, into a comprehensive prompt for the LLM. This step ensures that the model has access to all relevant characteristics of the commit, not only in terms of the code itself, but also the surrounding context and historical vulnerability fix commits. The output is a prediction with a structured analysis and reasoning on how the decision is made.

The prompt template is shown in Figure 6.4. The prompt is designed to follow a multi-dimensional approach [166] to ensure a thorough, structured analysis for a given commit. We do this by guiding the LLM to analyze and integrate information from the components using the CoT and ICL techniques before making its decision. We first provide the raw commit data (code diff and commit message) as the patch content, following by the output from the three components. Next, we design the prompt to include a two-step task for the LLM: Comparison and Analysis. In the Comparison phase, we prompt the LLM to evaluate the current patch against the retrieved historical fixes, to avoid potential bias and ensure an evidence-based analysis. This is because we cannot ensure that the retrieved historical vulnerabilities are actually relevant to the current commit. In the Analysis step, we ask the LLM to synthesize information from the three components to determine whether the patch is a vulnerability fix, and to provide justification. Finally, we instruct the LLM to generate the response in JSON format including its detailed analysis and finally its

decision. The analysis process is designed to output an analysis that can assist the user of LLM4VFD in the manual screening process to help them better understand the decision-making process of the LLM. We also conduct a user study in Section 6.5.3 to evaluate the usefulness of the resulting generated analyses.

6.4 Experimental Settings

In this section, we present research questions (RQs), datasets, evaluation metrics, our analysis approach for RQs, and implementation details.

6.4.1 Research Questions

We evaluate LLM4VFD in different aspects to answer the following research questions.

- *RQ1: How effective is LLM4VFD compared with SOTA techniques?*
- *RQ2: How effective is each component in LLM4VFD?*
- *RQ3: Can the analysis generated by LLM4VFD help security experts in identifying vulnerability fixes?*
- *RQ4: Bad Case Analysis: In which scenarios does LLM4VFD fail?*

6.4.2 Data Collection

Date Range Selection

LLMs are trained on extensive data, which results in a knowledge cutoff date reflecting the most recent information they possess. For our task, feeding LLMs with

Prompt Template: Comprehensive Analysis and Vulnerability Fix Detection

System Prompt: You are a helpful software developer assistant specializing in vulnerability detection to help other developers understand characteristics of software patches and discover potential vulnerabilities.

User Prompt: You are given the following details for analysis:

1. Patch Content: *{Commit}*
2. Related Issue Report / Pull Request Summary: *{DA component output}*
3. Three Aspect Analysis of the Patch: *{CCI component output}*
4. Similar Historical Vulnerability Fix Information: *{HV component output - CVE description}*
5. Three Aspect Analysis of the Historical Vulnerability Fix: *{HV component output - 3-aspect summary}*

Task:

1. Comparison:
 - Carefully compare the current patch with the historical vulnerability fix to avoid bias.
 - Ensure that you consider the similarities and differences highlighted in the three aspect analyses.
2. Analysis:
 - Use the information from the Related Issue Report / Pull Request Summary to understand the context and motivation behind the patch.
 - Determine whether the current patch is intended to fix a vulnerability. You must provide evidence if you think its a vulnerability fix.

Your output should follow below syntax:

```
{ "analysis": "<Detailed analysis of whether the patch is to fix a vulnerability>", "vulnerability_fix": "<yes or no>" }
```

Figure 6.4: The prompt template of Comprehensive Analysis and Vulnerability Fix Detection.

historical vulnerabilities predating their knowledge cutoff can lead to data leakage, as the model might already be aware of these vulnerabilities. As a mitigation, we restrict our analysis to vulnerabilities after 2023 to ensure that the data used is post-knowledge cutoff and reduces the risk of data leakage.

Vulnerability and Non-vulnerability Fix Commit Selection

We begin the data collection process by collecting historical CVE data from NVD [3]. NVD contains a vast amount of vulnerability data covering numerous open-source software (OSS) that cover a large spectrum of development activities and purposes. We only include CVEs that possess GitHub commit URLs in their references, which indicate a vulnerability fix. To obtain this info, we use GitHub's REST API endpoint to directly access repository and the language key in the response JSON [167]. To avoid the long tail effect from the diversity of vulnerabilities, we limit ourselves to vulnerabilities from 7 programming languages, namely Java, C, C++, Rust, JavaScript, Python, and Go.

In real-world open-source software (OSS) development, VF commits are exceptionally rare, comprising only a tiny fraction of total commits. The ratio of VF to normal commits can be as low as 1 in 1,000. For example, in the OSS project FFmpeg, we collected 114,210 total commits, of which only 124 were VF commits (0.1%). This extreme class imbalance makes our task significantly more challenging than typical binary classification tasks, which often assume a more balanced distribution (close to a 1:1 ratio). To address this imbalance and reduce the number of NVF commits evaluated, we followed the sampling strategy from the Big-Vul dataset [102] and selected

a ratio of 1:16 between VF and NVF commits. We randomly sample NVF commits from the same OSS where we collected vulnerability fix commits.

Our final evaluation dataset BigVulFixes consists of 1,689 VF and 26,468 NVF commits, reflecting this sampled ratio. Additionally, to handle extreme outliers and ensure compatibility with the maximum token length of the language models we use, we excluded patches longer than the 99th percentile of patch token lengths (approximately 30,000 tokens).

6.4.3 IR/PR Data Collection

For all commits gathered from the previous steps, we collect related IR/PR URLs and their information using two approaches. The first approach parses the commit message to find any references to IR/PRs. GitHub uses an autolink feature via specific formatting syntax, allowing developers to reference IR/PR information directly in commit messages [167]. We design a regular expression to find such references to IR/PRs based on the defined autolink formatting syntax. The second approach accesses the “List pull requests associated with a commit” GitHub REST API endpoint. This endpoint provides the pull request the corresponding commit is included in. This method is particularly useful for cases where developers do not reference the related pull request in the commit message, causing the first approach to fail. Finally, we use the GitHub REST API to retrieve the information from each of the collected IR/PR URLs. We collected a total of 17,791 IR/PR data for the BigVulFixes dataset.

6.4.4 Historical Vulnerability Data Collection

In the previous section, we describe how we collect data after 2023 for evaluation, therefore, to avoid data leakage, the data range we select for the history vulnerability dataset will be all vulnerable data before 2023. Similar to the previous section, we start by collecting all history CVE information on NVD [3] before 2023 and collected 22,745 vulnerabilities from vulnerability advisory NVD. For each historical vulnerability, we collect its associate vulnerability fix commit and CVE description.

6.4.5 Implementation Detail

LLMs Selection

LLM4VFD is a framework that can be implemented with any LLM and embedding models. In this study, we selected SOTA LLMs based on their performance on both code-related benchmarks (*e.g.*, HumanEval [168], MBPP EvalPlus [169]) and general benchmarks (*e.g.*, MMLU [170], IFEval [171]). We focused on three LLM families: Llama [172], Qwen [173], and Deepseek [174]. For the Llama family, we chose Llama3.1-70B and Llama3.1-8B. From the Qwen family, we selected Qwen2-70B and Qwen2-7B. For the Deepseek family, we included Deepseek-Coder-V2 (236B) and its smaller version, Deepseek-Coder-V2-Lite (16B). We use the "instruct" fine-tuning version of all LLMs.

We deployed Llama, Qwen and Deepseek-Coder-V2-Lite using vLLM [175] on Ascend 910 NPUs. For Deepseek-Coder-V2, we utilized the official API provided by DeepSeek. The total computational cost of the experiment was approximately 2.5 billion tokens. The inference speed is approximately 1 item/second for ≈ 70 B LLMs

and 10 items/second for ≈ 7 B LLMs. While more powerful LLMs like Llama3.1-405B or GPT-4o are available, they are either closed-source or too computationally intensive to deploy for our study. Therefore, we selected the models mentioned above.

Embedding Model and RAG Implementation

For the embedding model used in our Retrieval-Augmented Generation (RAG) to embed the 3-aspect summary, we chose gte-Qwen2-7B-instruct [165]. This model was the state-of-the-art sentence embedding model on the Massive Text Embedding Benchmark [176] as of June 16, 2024. We use ChromaDB [177] to implement the vector database to use as part of HV and RAG. When querying the HV database, We filter the HV query results using a condition to retrieve historical vulnerabilities only with the same programming language of the current commit. We use the default Euclidean Distance function from ChromaDB to calculate and retrieve the most similar historical vulnerability.

6.4.6 Evaluation Metrics

We use precision, recall, F1-score and Matthews correlation coefficient (MCC) [178] as our evaluation metrics, which are widely used in previous studies [25; 27; 26; 96; 104]. Unlike previous research, we avoid using accuracy as an evaluation metric due to the highly imbalanced nature of vulnerability fix detection datasets, where the majority class can dominate and lead to misleading results [115].

6.4.7 RQ Approaches

RQ1: How effective is LLM4VFD compared with SOTA techniques?

In RQ1, we compare LLM4VFD with existing SOTAs, including three PLM-based techniques and three selected LLMs with different sizes under the CoT 0-shot setting.

First, we select three SOTA PLM-based techniques, including VulFixMiner [15], CoLeFunda [19] and VulCurator [179]. We select VulFixMiner and CoLeFunda since they are the most commonly used baseline for vulnerability fix detection [180; 181]. VulCurator extends the approach proposed in VulFixMiner to include commit related development artifacts (*i.e.*, IR/PRs, commit message). Due to its design, CoLeFunda’s implementation is limited to the Java language, therefore, we only compare CoLeFunda with LLM4VFD on Java data. We obtain these SOTA models as provided from the original authors, and use them according to guidance from the authors and or replication packages. Note that we do not reuse the VulFixMiner dataset which was used to evaluate both VulFixMiner and CoLeFunda since this dataset poses a threat of data leakage since it contains data before 2023, which predates the cutoff dates of all LLMs included in our study. Therefore, we evaluate all models using our newly collected dataset BigVulFixes.

Secondly, we also compare the LLMs under their vanilla setting without LLM4VFD, defined as the 0-shot CoT setting. For a fair comparison, we maintain the structure of the vulnerability fix detection prompt template as shown in Figure 6.4, but omit all information from the Code Change Intention (CCI), Development Artifact (DA), and History Vulnerability (HV) components. In other words, the LLM is only given the patch content and is tasked with determining whether the commit is a VF, without

any additional context from LLM4VFD. Additionally, the task instruction is simplified to: “Determine whether the current patch is intended to fix a vulnerability. You must provide evidence if you think it’s a vulnerability fix”.

RQ2: How effective is each component in LLM4VFD?

In our framework, we integrate three key components: Code Change Intention (CCI), Development Artifacts (DA), and Historical Vulnerability (HV). To understand the contribution of each component, we conduct an ablation study to evaluate their individual impact on the overall performance of LLM4VFD. For this study, we selected two models, Qwen2-72B and Qwen-7B, from the Qwen family. These models were chosen because both the larger and smaller versions demonstrated strong performance in our initial experiments (see section 6.5.1 for details). In the ablation study, we systematically removed each component one at a time and compared the performance of the models with and without the removed component.

RQ3: Can the analysis generated by LLM4VFD help security experts in identifying vulnerability fixes?

Having a model detect and label a commit as a VF is generally not the end of the story. Security experts will often perform screening on the instance to confirm the prediction was correct. This is an essential step before beginning remediation, such as applying the patch into downstream software dependencies. Therefore, in RQ3 we conduct a user study to investigate whether analysis generated by LLM4VFD can help developers identify vulnerability fixes more effectively. We elaborate on the methodology of our user study below.

Participants: We invited 10 security experts from industry and academia with 3–5 years of software security experience for the user study from industry, with 6 participants from industry and 4 from academia. We also ensure that these security experts experience of screening vulnerability fixes before.

Task: We randomly selected 40 VF and their analysis generated by LLM4VFD based on Qwen2-72B.

Procedure: Each participant is tasked to answer a sequence of yes or no questions with the option of adding additional comments given a VF and the generated analysis from LLM4VFD.

The user study was conducted via questionnaires. We present the analysis results generated by LLM4VFD and ask reviewers to respond with "yes" or "no" to the following questions, while also providing a comment section for additional feedback:

1. Does the analysis help understand the intent behind the code changes?
2. Does the analysis accurately characterize the vulnerability?
3. Does the analysis provide a better understanding of the root cause of the vulnerability?
4. Does the analysis provided by the large model help improve the efficiency of identifying vulnerabilities fix?
5. Are you satisfied with the quality of the generated content (*e.g.*, it contains redundant information, inaccurate information, not thorough enough, hallucination, historical vulnerabilities mentioned are irrelevant to the vulnerability, etc.)

By collecting the answers to these questions, we aim to determine if the analysis information provided by LLM4VFD can help participants effectively understand the commit and verify vulnerabilities fixes.

RQ4: Bad Case Analysis: In which scenarios does LLM4VFD fail?

To better understand the limitations of LLM4VFD, we conduct a manual analysis on failed predictions, focusing on two types of misclassifications: (1) False Positives (FP): Cases where LLM4VFD incorrectly classifies a commit as a VF. This helps us examine scenarios where LLM4VFD misinterprets the intent or context of the commit. And (2) False Negatives (FN): Cases where LLM4VFD fails to identify a commit as a VF. This analysis highlights situations where LLM4VFD overlooks important indicators of a VF.

The first three authors each conducted a manual inspection on 20 FP and 20 FN for each of Qwen-72B and Qwen-7B following RQ2. A total of 240 cases has been reviewed. We aim to identify the specific reasons for the misclassifications. By pinpointing the root causes of the errors, we seek to understand the current limitations of LLM4VFD and identify room for future improvement.

Table 6.1: The performance of vulnerability fix detection approaches.

Foundation Model	Parameter Size	Approach	Precision	Recall	F1-score	MCC
CodeBERT	125M	VulFixMiner	0.17	0.26	0.20	0.14
	125M	CoLeFunDa *	0.50	0.06	0.11	0.15
	125M	VulCurator	0.77	0.13	0.22	0.30
Deepseek-Coder-V2	236B	Vanilla	0.33	0.84	0.47	0.48
		LLM4VFD	0.40	0.78	0.53	0.53
	16B	Vanilla	0.23	0.44	0.30	0.26
		LLM4VFD	0.28	0.65	0.39	0.37
Llama3.1	70B	Vanilla	0.41	0.53	0.47	0.43
		LLM4VFD	0.49	0.61	0.54	0.52
	8B	Vanilla	0.10	0.86	0.18	0.18
		LLM4VFD	0.30	0.50	0.37	0.34
Qwen2	72B	Vanilla	0.32	0.72	0.44	0.43
		LLM4VFD	0.38	0.77	0.51	0.50
	7B	Vanilla	0.24	0.48	0.32	0.29
		LLM4VFD	0.52	0.48	0.50	0.47

* Due to the tool limitation, CoLeFunDa is evaluated on only Java vulnerabilities.

6.5 Results

6.5.1 RQ1 - Effectiveness

PLM-based Approaches vs. LLM4VFD

LLM4VFD outperforms PLM-based approach consistently in terms of MCC, F1-score, recall. For instance, LLM4VFD outperforms the best-performed PLM-based approach VulCurator by 68.1% - 145.4% across

LLMs in terms of F1-score. As shown in Table 6.1, LLM4VFD significantly outperforms all three PLM-based approaches, VulFixMiner, CoLeFunda, and VulCurator, in terms of F1-score, MCC, and Recall. LLM4VFD with various LLMs achieves an F1-score ranging from 0.37 to 0.54, which is significantly higher than PLM-based approaches with an F1-score of 0.14 to 0.22. For instance, the best performed LLM4VFD (with Llama3.1-70B) achieves an F1-score of 0.54, exhibiting a 145.5% improvement in F1-score compared to the best PLM-based approach VulCurator. Although VulCurator and ColeFunDa achieve good precision 0.50 and 0.77 respectively, they suffer from very low Recall of 0.06 and 0.13 and miss many true vulnerability fixes, which hinders their practical application.

Vanilla LLMs (0-shot CoT) vs. LLM4VFD

LLM4VFD outperforms all vanilla LLMs in terms of F1-score and MCC. Compared to the vanilla LLM, we observe that LLM4VFD improves performance across all evaluation metrics, except Recall on two LLMs (*i.e.*, DeepseekCoder-V2 and Llama3.1-8B-Instruct). In terms of F1-score, for each LLM LLM4VFD achieves a range of 12.7% to 105.6% improvements over its vanilla setting. We observe a similar pattern of improvement for MCC. In particular, LLM4VFD with Deepseekcoder-V2 achieves a Precision of 0.40 and a Recall of 0.78, resulting in an F1-score of 0.53, representing a 13% improvement in Precision and a 12.8% increase in F1-score compared to its vanilla setting. Similarly, Llama3.1-70B improves its Precision from 0.41 to 0.49 (a 19.5% increase) and Recall from 0.53 to 0.61 (a 15.1% increase), resulting in an F1-score of 0.54 (a 14.9% increase).

Larger models typically outperform smaller models, while smaller models benefit more from our framework. For instance, F1-score for smaller models achieve an improvement of 64.0% on average after applying LLM4VFD compared with the vanilla setting, while large models achieve an average improvement of 14.4%. When comparing smaller models with their larger counterparts within the same family, we observed that the larger models consistently outperform the smaller ones, both with and without the LLM4VFD framework. This is likely because vulnerability fix detection is a complex task that benefits from the extra parameters providing more code understanding capabilities. In addition, the explanations for LLMs in the vanilla setting show significant variation, with F1-scores ranging widely from 0.18 to 0.50, especially for smaller models. For example, within the Llama family, the vanilla Llama3.1-70B-Instruct model achieves an F1-score of 0.47, whereas the smaller Llama3.1-8B-Instruct model only achieves an F1-score of 0.18. However, after applying our LLM4VFD framework, we find that the improvement in performance is more pronounced for smaller models than for larger ones. On average, the F1-scores see a relative improvement of 64.0%, whereas larger models saw a more modest average increase of 14.4%. These results indicate that LLM4VFD significantly enhances the performance of smaller models, making them more competitive with larger models and narrowing the performance gap.

Table 6.2: The ablation results. The cells with the lowest performance are marked in bold, indicating the largest contribution from that component.

Ablation Setting	Qwen2-72B				Qwen2-7B			
	Precision	Recall	F1	MCC	Precision	Recall	F1	MCC
LLM4VFD	0.38	0.77	0.51	0.50	0.52	0.48	0.50	0.47
w/o CCI component	0.33	0.77	0.46	0.45	0.40	0.54	0.46	0.42
w/o DA component	0.36	0.75	0.48	0.48	0.49	0.47	0.48	0.45
w/o HV component	0.36	0.74	0.49	0.48	0.39	0.62	0.48	0.45

LLM4VFD outperforms PLM-based approach consistently in terms of MCC, F1-score, and recall. For instance, LLM4VFD outperforms the best-performed PLM-based approach VulCurator by 68.1%–145.4% across LLMs in terms of F1-score. Our framework demonstrates performance gains ranging from 12.7% to 105.6% over its vanilla variant, with smaller models generally benefiting more compared to their larger counterparts.

6.5.2 RQ2 - Ablation analysis.

Overall, all three components in LLM4VFD make positive contributions to the overall performance. Among them, Code Change Intention have a larger impact than Development Artifact and Historical Vulnerability. Table 6.2 shows the results of our ablation analysis. The removal of CCI component leads to a significant decrease in performance, particularly in terms of Precision and F1-score. For Qwen2-72B, the Precision drops from 0.38 to 0.33, representing a 13.1% reduction. In Qwen2-7B, the decline is similar, with the Precision decreasing from 0.52 to 0.40, an 15.4% reduction. These results emphasize the important role of CCI

in improving the model’s ability to capture vulnerability fix commits more accurately by reducing false positives, and the improvement is especially notable in smaller models, where CCI helps balance Precision and Recall effectively. DA component has a more similar effect on Qwen2-7B and Qwen2-72B. In Qwen2-72B, the F1-score improves significantly from 0.48 to 0.51, marking an 6.3% increase, indicating that the additional contextual information provided by DA substantially boosts both precision and recall. Removing the HV component or DA components results in slight reductions in F1-score and MCC for both models. However, we find that Qwen2-7B sees a more significant reduction in precision without the HV components, dropping from 0.52 to 0.39, a 25% decrease.

The ablation study highlights the importance of each component in LLM4VFD, particularly CCI, which demonstrates significant improvements in precision and F1-score, especially for smaller models like Qwen2-7B. HV further enhances precision by reducing false positives, albeit with a more moderate effect. Collectively, these components provide a comprehensive framework that significantly enhances the accuracy and reliability of vulnerability fix detection, particularly by augmenting smaller models with crucial contextual and historical information.

All components in LLM4VFD make positive contributions to the performance. The impact from Code Change Intention is larger than Development Artifact and Historical Vulnerability.

6.5.3 RQ3 - User Study

The analysis generated by LLM4VFD helps security experts to understand the intent of code changes and the vulnerabilities, which improves

the efficiency of identifying vulnerability fixes. Our result shows that in 95.0% (38 out of 40) of the cases, LLM4VFD’s analysis can help with understanding the intent of commits (Question 1). A case like CVE-2024-29199 [182] has 47 changed files with 517 lines of addition and 226 lines of deletion. It is too long for participants to understand, however LLM4VFD’s analysis helped participants to understand the change. In addition, participants think that LLM4VFD accurately describes the characteristics of the vulnerabilities in 90% of the commits (Question 2) and the root causes in 75.0% (30 out of 40) of the cases (Question 3). With LLM4VFD, participants can identify VF commit more efficiently. This is evidenced by the fact that in 80.0% (32 out of 40) of the commits, LLM4VFD improved their efficiency in identifying vulnerability fixes (Question 4). We investigate the 8 cases where participants did not think the analysis generated by LLM4VFD helped them to identify vulnerability fixes more efficiently, 7 of them are due to the vulnerability fixes are easy to identify even without the help of LLM4VFD. In one case of CVE-2024-28103 [183], the participant did not understand the vulnerability even with the help of LLM4VFD.

When investigating the feedback we collected related to the quality of the analysis generated by LLM4VFD (Question 5), we find that in 75% of the cases, participants are satisfied with the overall quality of the analysis. There were some cases (*e.g.*, CVE-2023-48014 [184] and CVE-2023-37061 [185]) where participants noted that the analysis contained redundant information, or lacked depth analysis for more complex vulnerabilities (*e.g.*, CVE-2023-48657 [186]). This suggests a need for further refinement in handling edge cases and ensuring that the historical context provided is directly relevant and concise. Nevertheless, ours is the first work on this direction

Table 6.3: Bad Case Analysis on Qwen2 Models

Type	Reason	72B	7B
FP	Potential unreported vulnerability fix	1	7
	Non-vulnerability security fix misclassified as vulnerability fix	45	36
	Non-functional Change	7	8
	Fail to realize the change is not related to security	4	5
	Mislead by retrieved similar vulnerability	2	3
	Others	1	1
FN	Vulnerability fix misclassified as non-vulnerability security fix	28	33
	Unable to identify security related code change	21	18
	Mislead by retrieved similar vulnerability	5	6
	Unable to pinpoint vulnerability related code change from long context	1	0
	Others	5	3

and more future research is encouraged.

Overall, the user study shows that the analysis generated by LLM4VFD helps security experts to understand the intent of code changes and the vulnerabilities, which improves the efficiency of identifying vulnerability fixes.

6.5.4 RQ4 - Failure Analysis

Table 6.3 summarizes the results of our failure analysis. To better understand the misclassification issues encountered by LLM4VFD, it is crucial to clarify the relationship between security fixes and vulnerability fixes. A *vulnerability fix* is a subset of *security fixes*. Security fixes encompass any code changes that enhance the overall security of the software, such as improvements to authentication mechanisms, encryp-

tion implementations, or adherence to security best practices, whereas vulnerability fixes aim to address security flaw, glitch, or weakness found in software code that could be exploited by an attacker [3]. Therefore, in our dataset, a security fix commit that does not aim to fix a vulnerability is considered as non-vulnerability fix (i.e., *non-vulnerability security fix*), although they are related to security fix.

The most mis-classification made by LLM4VFD due to its struggling to distinguish between vulnerability fix and a non-vulnerability security fix. For instance, in false positive (FP) cases, 75.0% of FP in Qwen2-72B and 60.0% in Qwen2-7B occurred due to non-vulnerability security fixes are misclassified as vulnerability fixes. Vice versa, it led to 46.7% and 55.0% of the FN cases where vulnerability fixes are misclassified as non-vulnerability fixes in Qwen2-72B and Qwen2-7B. The result indicates that even if LLM4VFD can identify security-related commits, it sometimes fails to interpret their severity. Specifically, in the analysis of an FN case, the LLM explains with: “Although changes are related to security, I am uncertain if the changes fix a vulnerability”. One possible explanation is that the LLM is unable to differentiate between security related change and vulnerability fix when generating the intention for commits. Although, LLM4VFD struggles to distinguish between vulnerability fix and non-vulnerability security fix. Both vulnerability fix and non-vulnerability security fix are security fixes, and identifying them is beneficial.

Another frequent failure in FN cases is the inability to identify security related code change (32.5%). Another notable failure case is related to its use of historical vulnerability data through the HV component. The retrieval of irrelevant history vulnerability causes misclassification in both FP and FN cases. Specifically, in 4.2%

of the FP cases, and 9.2% of the FN cases, the HV component retrieved vulnerabilities that seemed relevant but did not match the functional or contextual details of the current commit, and such information caused confusion to LLM, leading LLM4VFD to make incorrect assumptions on the commits. While our HV component improves the overall performance (as shown in our ablation study in Section 6.5.2), it is not perfect and can sometimes retrieve irrelevant historical vulnerabilities. Future studies can focus on this aspect by developing better retrieval techniques and improving the underlying vulnerability database.

We also find in a few FP cases where the commit is potentially fixing an unreported vulnerability. For example, in one of the checked cases, the commit is a merge of changes from a GHSA [187] pull request. GHSA is an independent security advisor that maintains its own vulnerability dataset and may have different assessments compared to NVD.

6.6 Discussion

6.6.1 Compromising between Precision and Recall

PLM approaches typically output a probability to indicate the likelihood of a commit being made for a vulnerability fix. A threshold between 0 and 1 is provided to make final binary decision. By adjusting the threshold value, we could favor precision or recall based on the needs. However, this technique is no longer applicable when using LLM since LLM’s strength is in generating natural language outputs. During our prompt engineering phase, we noticed that by tweaking the word choice in the

system and user prompt, we can instruct the LLM to favor precision or recall in the final prediction. However, this adjustment is more subtle and hard to quantify. We encourage future work to further explore this aspect and design LLM-based systems that allow some flexibility in favoring precision or recall.

6.6.2 Potential Future Direction

Our research is the first study to explore vulnerability fix detection using LLMs. However, insights from our user study and failure case analysis indicate several areas for future exploration. First, incorporating more relevant project-specific information, such as security policies and historical commit patterns, could help LLMs better distinguish vulnerability fixes among security-related commits. Second, the prompt templates guiding the LLM could be refined to more effectively leverage context and produce more insightful analysis results for security experts. Finally, the current RAG design is basic, and incorporating advanced techniques — such as dynamic retrieval strategies or reranking mechanisms — could improve the precision and relevance of retrieved historical vulnerabilities.

6.6.3 Threats to Validity

Internal Validity

Due to the extremely imbalanced nature of vulnerability fixes and non-vulnerability fixes, when our approach is applied to monitor real-world projects there may be a large amount of false negatives. Based on feedback from our industry partners, the false positive rate was considered acceptable, as it requires only a small

amount of additional human effort to review the results. Previous studies suggest that LLM settings, such as temperature, have an impact on outputs [188; 189]. In this study, we use the default settings for the studied LLMs for all RQs.

External Validity

Threats to external validity relate to the generalizability of our approach. Since LLM4VFD is a framework, and its implementation relies on LLMs. The effectiveness of LLM4VFD also depends on the performance and capabilities of the LLMs. To mitigate this threat, we evaluated LLM4VFD using several well-known and state-of-the-art LLMs, including both large models (70B–236B parameters) and smaller models (7B–16B parameters), and our results show that with all models, LLM4VFD outperforms SOTA baselines. Future research is encouraged to investigate the performance with more LLMs using our framework.

6.7 Conclusion

In this paper, we propose LLM4VFD, a novel framework that leverages Large Language Models (LLMs) enhanced with Chain-of-Thought reasoning and In-Context Learning to improve the accuracy of vulnerability fix detection by integrating information from multiple sources (e.g., related development artifacts and historical vulnerabilities). More importantly, on top of the prediction, LLM4VFD also provides a detailed explanation in analysis to help security experts understand the rationale behind the decision. Experimental results demonstrate that LLM4VFD outperforms PLM-based approach consistently in terms of MCC, F1-score, and recall. For in-

stance, LLM4VFD outperforms the best-performed PLM-based approach VulCura-
tor by 68.1% - 145.4% in terms of F1-score when using different LLMs as the base
model. Furthermore, we conducted a user study involving security experts to assess
the effectiveness of the analysis generated by LLM4VFD in aiding vulnerability fix
identification. The feedback from the participants demonstrates that the analysis
provided by LLM4VFD improves the efficiency of identifying vulnerability fixes.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Open-source software (OSS) is integral to modern software engineering, underpinning critical infrastructure and numerous software applications across diverse domains. Despite the immense benefits of OSS, such as rapid development and collaborative innovation, they inherently come with substantial security risks. Vulnerabilities constantly emerge and must be managed promptly to prevent exploitation, maintain system stability, and safeguard user trust. As OSS scales dramatically, the sheer volume of contributions, patches, and repositories makes manual vulnerability management impractical and error-prone. This drives a critical need for efficient, accurate, and interpretable automatic vulnerability management tools that can keep pace with today's software ecosystems, where robust automation can significantly reduce the window between vulnerability emergence and mitigation, fostering greater software security.

To advance automatic vulnerability management, this thesis has undertaken four empirical studies aimed at enhancing two essential OSS vulnerability management tasks: (1) vulnerability detection (VD) and (2) vulnerability fix detection (VFD). Our research systematically addressed critical challenges associated with these tasks, including data imbalance in training vulnerability detection models, the limitations of existing single-model vulnerability detection frameworks for diverse vulnerability types, tangled commit characterization, and a lack of interpretability in fix prediction models. Specifically, we have: (1) conducted a systematic quantification and evaluation of data sampling strategies on deep learning-based vulnerability detection models, (2) developed *MoEVD* that employs a Mixture-of-Experts (MoE) architecture to overcome drawbacks associated with existing “one-for-all” frameworks for VD, (3) introduced *VFDelta*, a framework for efficiently detecting vulnerability fixes hidden within subtle, tangled commits, by enhancing code change semantic representation, and (4) proposed *LLM4VFD*, the first Large Language Model (LLM) based approach that leverages diverse software-related artifacts and historical vulnerability knowledge for explainable detection of vulnerability fix commits.

Our first in Chapter 3 study addressed severe class imbalance in vulnerability detection datasets through systematic evaluation of data sampling strategies. We found that random oversampling on raw code consistently improves both detection performance and model interpretability across multiple DLVD approaches.

Chapter 4 presented *MoEVD*, a Mixture-of-Experts architecture that overcomes the limitations of “one-for-all” vulnerability detection frameworks. By routing inputs to CWE-specialized experts, *MoEVD* achieved 12.8% higher F1 than state-of-the-art

methods while significantly improving detection of long-tailed vulnerability types.

Chapter 5 introduced VFDelta, which detects silent vulnerability fixes through explicit modeling of code change semantics using delta embeddings. VFDelta demonstrated substantial improvements in detecting subtle security-critical modifications, with F1 improvements exceeding 77% for Java and 128% for Python. Chapter 6 proposed LLM4VFD, the first LLM-based approach for vulnerability fix detection that integrates code changes with development artifacts and historical vulnerability knowledge. Beyond achieving 68-145% better F1 scores than existing methods, LLM4VFD provides actionable explanations that substantially improve security experts' assessment efficiency.

7.2 Future Work

This thesis systematically explored several critical dimensions necessary for making automated OSS vulnerability management practical and reliable. Despite significant advancements, our findings point toward several promising research directions.

Data Sampling for Enhanced Vulnerability Detection In Chapter 3, we demonstrated the effectiveness of simple oversampling approaches, particularly random oversampling on raw code. However, this technique largely focuses on the syntactic textual representation, while code structures—such as call graphs or code property graphs—remain inadequately addressed. Consequently, future research can extend our current data sampling strategies to leverage more sophisticated representations of source code, such as code property graphs, call graphs, or even abstract syntax

trees. Such code-specific attributes capture intricate structural relationships and dependencies, potentially further mitigating data imbalance impacts and improving the robustness of deep-learning vulnerability detection models.

Advancing the Mixture-of-Experts Framework for Vulnerability Detection

Our investigation in Chapter 4 pioneered the first application of the Mixture-of-Experts (MoE) architecture in vulnerability detection, validating the considerable benefits that MoE affords over traditional “one-for-all” approaches. Nonetheless, our analysis revealed notable room for improvement: for instance, although powerful, the vanilla MoE routing mechanism correctly directed only 63.8% of vulnerable code instances to their specialized experts. In an ideal scenario, perfect router accuracy could elevate the F1-score from 0.44 to 0.51. Thus, enhancing router accuracy is a highly promising research direction. Future work might explore advanced routing techniques, such as dynamic routing or context-aware routers that better interpret subtle code patterns and vulnerability types. Moreover, integrating more sophisticated MoE frameworks (e.g., Shazeer et al.[113], GShard[129], Switch Transformers [130]) can further optimize computational costs, scalability, and effectiveness, ultimately enhancing MoE-based vulnerability detection models for practical and large-scale usage.

VFDelta for General Code Change Applications Our VFDelta framework, described in Chapter 5, revealed strong effectiveness when detecting vulnerabilities hidden within tangled commits. While our experiments focused primarily on silent vulnerability fix detection, the underlying delta embedding approach to mod-

eling subtle semantic differences strategically aligns with other code-change related tasks. Therefore, future investigations could leverage VFDelta’s generalizable architecture in broader applications within software engineering that rely on semantic representation of code changes. For instance, tasks such as Just-In-Time defect prediction [156; 146; 148] could significantly benefit from VFDelta’s ability to characterize nuanced code modifications accurately. Similarly, commit message generation [146; 148], which inherently depends on the comprehensive understanding and summarization of commit-level changes, could employ our delta embedding representation as an impactful encoder module. Further research in these areas can validate VFDelta’s flexibility and broad utility beyond the specific scope of silent vulnerability fix detection.

Improving LLM4VFD with Advanced Retrieval and Context Techniques

As the first exploration utilizing Large Language Models (LLMs) for vulnerability fix detection, Chapter 6 demonstrated the significant potential of applying this technique in vulnerability fix detection. Yet, user feedback and failure case analyses identified several valuable directions for further enhancement. First, our current LLM4VFD system primarily focuses on leveraging commit messages, code diffs, historical vulnerability fixes, and related development artifacts. However, incorporating additional project-specific information, such as specific security policies, documentation describing common security practices within projects, or historical commit patterns, could further improve the effectiveness and accuracy of identifying and distinguishing intended vulnerability fixes. Second, refining the prompt templates accompanying the LLM could optimize their utility and effectiveness, enabling deeper context utiliza-

tion and generating more accurate and insightful explanations for security experts in real-world usage. Lastly, the adopted Retrieval-Augmented Generation (RAG) mechanisms currently employed by LLM4VFD remain relatively basic. Future research could explore and integrate advanced retrieval methodologies—such as dynamic or adaptive retrieval strategies, retrieval re-ranking mechanisms, or even query refinement techniques—to enhance the precision, relevance, and quality of the retrieved historical vulnerabilities, thus further improving prediction accuracy and analysis quality.

Extending to Automated Vulnerability Repair While this thesis focused on vulnerability detection and vulnerability-fix detection, our findings lay important groundwork for addressing another critical task in the OSS vulnerability management: vulnerability repair. Our evaluation of LLM4VFD demonstrated that large language models possess strong code reasoning capabilities and comprehensive understanding of vulnerability patterns, development contexts, and security implications. These capabilities make LLM a strong tool for automated vulnerability repair, where the model must not only understand the vulnerability but also synthesize appropriate patches that address the security flaw without breaking existing functionality. Future work could leverage the contextual understanding developed in LLM4VFD to guide patch generation and validation. By extending our current frameworks to vulnerability repair, we could complete the automation of critical early stages in the vulnerability management pipeline, enabling end-to-end automated response to emerging security threats in OSS ecosystems and reducing the time between vulnerability discovery and remediation.

Bibliography

- [1] Apache Software Foundation, “CVE-2018-11776,” <https://nvd.nist.gov/vuln/detail/CVE-2018-11776>, 2018.
- [2] —, “CVE-2022-46363,” <https://nvd.nist.gov/vuln/detail/CVE-2022-46363>, 2022.
- [3] NIST, “NVD - Home,” 2024, accessed 2024-09-10. [Online]. Available: <https://nvd.nist.gov/>
- [4] Apache Software Foundation, “CVE-2021-44228,” <https://nvd.nist.gov/vuln/detail/CVE-2021-44228>, 2021.
- [5] “Equifax to Pay at Least \$650 Million in Largest-Ever Data Breach Settlement,” <https://www.nytimes.com/2019/07/22/business/equifax-settlement.html>, accessed: 2024-02-05.
- [6] “Bad cyber hygiene: 60 percent of breaches tied to unpatched vulnerabilities,” <https://www.automox.com/blog/bad-cyber-hygiene-breaches-tied-to-unpatched-vulnerabilities>, accessed: 2024-02-05.

- [7] M. Alfadel, D. E. Costa, and E. Shihab, “Empirical analysis of security vulnerabilities in python packages,” *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.
- [8] S. Li, M. Kang, J. Hou, and Y. Cao, “Detecting node.js prototype pollution vulnerabilities via object lookup analysis,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 268–279.
- [9] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, J. F. Santos, and N. Santos, “Study of javascript static analysis tools for vulnerability detection in node.js packages,” *IEEE Transactions on Reliability*, vol. 72, no. 4, pp. 1324–1339, 2023.
- [10] Z. Y. Ding and C. Le Goues, “An empirical study of oss-fuzz bugs,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 131–142.
- [11] ISO/IEC JTC 1/SC 27, *Security techniques – Vulnerability disclosure*, Std. ISO/IEC 29147:2018, 2018. [Online]. Available: <https://www.iso.org/standard/72311.html>
- [12] A. D. Householder, G. Wassermann, A. Manion, and C. King, “The cert guide to coordinated vulnerability disclosure,” *Software Engineering Institute, Pittsburgh, PA*, 2017.
- [13] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, “Find-

- ing a needle in a haystack: Automated mining of silent vulnerability fixes,” in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.
- [14] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet,” *IEEE Transactions on Software Engineering*, 2021.
- [15] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, “Finding a needle in a haystack: Automated mining of silent vulnerability fixes,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021.
- [16] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at google,” *Communications of the ACM*, 2018.
- [17] T. G. Nguyen, T. Le-Cong, H. J. Kang, R. Widyasari, C. Yang, Z. Zhao, B. Xu, J. Zhou, X. Xia, A. E. Hassan *et al.*, “Multi-granularity detector for vulnerability fixes,” *IEEE Transactions on Software Engineering (TSE)*, 2023.
- [18] OpenSSL Software Foundation, “CVE-2014-0160,” [://nvd.nist.gov/vuln/detail/cve-2014-0160](https://nvd.nist.gov/vuln/detail/cve-2014-0160), 2014.
- [19] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, and A. E. Hassan, “Cofunda: Explainable silent vulnerability fix identification,” in *Proceedings of the*

- 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.
- [20] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, “Predicting vulnerable software components via text mining,” *IEEE Transactions on Software Engineering*, 2014.
- [21] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, “Graph-based mining of multiple object usage patterns,” in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE ’09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: <https://doi.org/10.1145/1595696.1595767>
- [22] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeep-ecker: A deep learning-based system for vulnerability detection,” in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.
- [23] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “Sysevr: A framework for using deep learning to detect software vulnerabilities,” *IEEE Transactions on Dependable and Secure Computing*, 2021.
- [24] M. Fu and C. Tantithamthavorn, “Linevul: A transformer-based line-level vulnerability prediction,” 2022.
- [25] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability

- identification by learning comprehensive program semantics via graph neural networks,” in *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS)*, 2019.
- [26] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?” *IEEE Transactions on Software Engineering (TSE)*, 2021.
- [27] 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*, 2021.
- [28] 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*, 2022.
- [29] H. Hanif and S. Maffei, “Vulberta: Simplified source code pre-training for vulnerability detection,” in *2022 International joint conference on neural networks (IJCNN)*. IEEE, 2022.
- [30] X. Zhang, F. Zhang, B. Zhao, B. Zhou, and B. Xiao, “Vuld-transformer: source code vulnerability detection via transformer,” in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, 2023, pp. 185–193.
- [31] L. Pelayo and S. Dick, “Evaluating stratification alternatives to improve software defect prediction,” *IEEE transactions on reliability*, 2012.

-
- [32] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 2015.
 - [33] C. Seiffert, T. M. Khoshgoftaar, and J. Van Hulse, "Improving software-quality predictions with data sampling and boosting," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 2009.
 - [34] R. Malhotra and M. Khanna, "An empirical study for software change prediction using imbalanced data," *Empirical Software Engineering*, 2017.
 - [35] R. Yedida and T. Menzies, "On the value of oversampling for deep learning in software defect prediction," *IEEE Transactions on Software Engineering*, 2021.
 - [36] W. Zheng, Y. Xun, X. Wu, Z. Deng, X. Chen, and Y. Sui, "A comparative study of class rebalancing methods for security bug report classification," *IEEE Transactions on Reliability*, 2021.
 - [37] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *First international symposium on empirical software engineering and measurement (ESEM 2007)*. IEEE, 2007.
 - [38] R. Jacobs, M. Jordan, S. Nowlan, and G. Hinton, "adaptive mixtures of local experts," *neural computation*, vol. 3," 1991.
 - [39] M. I. Jordan and R. A. Jacobs, "Hierarchical mixtures of experts and the em algorithm," *Neural computation*, 1994.

-
- [40] A. Omer, S. S. Rathore, and S. Kumar, “Me-sfp: A mixture-of-experts-based approach for software fault prediction,” *IEEE Transactions on Reliability*, 2023.
- [41] A. Shankar Mishra and S. Singh Rathore, “Implicit and explicit mixture of experts models for software defect prediction,” *Software Quality Journal*, 2023.
- [42] Y. Zhou and A. Sharma, “Automated identification of security issues from commit messages and bug reports,” in *Proceedings of the 11th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017.
- [43] A. Sabetta and M. Bezzi, “A practical approach to the automatic classification of security-relevant commits,” in *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018.
- [44] G. Nguyen-Truong, H. J. Kang, D. Lo, A. Sharma, A. E. Santosa, A. Sharma, and M. Y. Ang, “Hermes: Using commit-issue linking to detect vulnerability-fixing commits,” in *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022.
- [45] S. Nguyen, T. T. Vu, and H. D. Vo, “Vffinder: A graph-based approach for automated silent vulnerability-fix identification,” in *2023 15th International Conference on Knowledge and Systems Engineering (KSE)*, 2023.
- [46] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, “Spain: security patch analysis for binaries towards understanding the pain and pills,” in *Proceed-*

- ings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2017.
- [47] A. Chan, A. Kharkar, R. Z. Moghaddam, Y. Mohylevskyy, A. Helyar, E. Kamal, M. Elkamhawy, and N. Sundaresan, “Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning?” *arXiv preprint arXiv:2306.01754*, 2023.
- [48] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, “Transformer-based language models for software vulnerability detection,” in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022.
- [49] X. Du, G. Zheng, K. Wang, J. Feng, W. Deng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou, “Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag,” *arXiv preprint arXiv:2406.11147*, 2024.
- [50] Y. Cheng, L. K. Shar, T. Zhang, S. Yang, C. Dong, D. Lo, S. Lv, Z. Shi, and L. Sun, “Llm-enhanced static analysis for precise identification of vulnerable oss versions,” *arXiv preprint arXiv:2408.07321*, 2024.
- [51] D. Shen, X. Chen, C. Wang, K. Sen, and D. Song, “Benchmarking language models for code syntax understanding,” in *Findings of the Association for Computational Linguistics: EMNLP 2022*, 2022.
- [52] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman,

- D. Almeida, J. Altschmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [53] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, “Codet5+: Open code large language models for code understanding and generation,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.
- [54] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, 2022.
- [55] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, “A survey on in-context learning,” *arXiv preprint arXiv:2301.00234*, 2022.
- [56] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeep-ecker: A deep learning-based system for vulnerability detection,” in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.
- [57] D. Hin, A. Kan, H. Chen, and M. A. Babar, “Linevd: Statement-level vulnerability detection using graph neural networks,” in *IEEE/ACM 19th International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, 2022.
- [58] 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*, 2021.
- [59] H. Xu, R. Duan, S. Yang, and L. Guo, “An empirical study on data sampling for just-in-time defect prediction,” in *International Conference on Artificial Intelligence and Security*. Springer, 2021.
- [60] K. W. Bowyer, N. V. Chawla, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: synthetic minority over-sampling technique,” *CoRR*, 2011. [Online]. Available: <http://arxiv.org/abs/1106.1813>
- [61] M. Kubat, S. Matwin *et al.*, “Addressing the curse of imbalanced training sets: one-sided selection,” in *Icml*. Citeseer, 1997.
- [62] 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*, 2019.
- [63] M. T. Ribeiro, S. Singh, and C. Guestrin, ““ why should i trust you?” explaining the predictions of any classifier,” in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016.
- [64] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, “Gnnexplainer: Generating explanations for graph neural networks,” *Advances in neural information processing systems*, 2019.
- [65] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *Proceedings of Workshop at ICLR*, 2013.

- [66] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014.
- [67] 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.
- [68] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, “A comprehensive survey on graph neural networks,” *IEEE transactions on neural networks and learning systems*, 2020.
- [69] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014.
- [70] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, “Automated vulnerability detection in source code using deep representation learning,” in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018.
- [71] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “Smote: synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, 2002.
- [72] I. Tomek, “Two Modifications of CNN,” *IEEE Transactions on Systems, Man, and Cybernetics*, 1976.

-
- [73] C. X. Ling and C. Li, “Data mining for direct marketing: Problems and solutions.” in *Kdd*, 1998.
- [74] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “Ac/c++ code vulnerability dataset with code changes and cve summaries,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.
- [75] D. M. Powers, “Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation,” *arXiv preprint arXiv:2010.16061*, 2020.
- [76] G. K. Rajbahadur, S. Wang, G. Ansaldi, Y. Kamei, and A. E. Hassan, “The impact of feature importance methods on the interpretation of defect classifiers,” *IEEE Transactions on Software Engineering*, 2021.
- [77] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, “Impact of discretization noise of the dependent variable on machine learning classifiers in software engineering,” *IEEE Transactions on Software Engineering*, 2019.
- [78] T.-D. B. Le, D. Lo, and M. Li, “Constrained feature selection for localizing faults,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015.
- [79] A. Okutan and O. T. Yıldız, “Software defect prediction using bayesian networks,” *Empirical Software Engineering*, 2014.
- [80] N. Chen, S. C. Hoi, and X. Xiao, “Software process evaluation: A machine

- learning approach,” in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011.
- [81] W. Fu, V. Nair, and T. Menzies, “Why is differential evolution better than grid search for tuning defect predictors?” *arXiv preprint arXiv:1609.02613*, 2016.
- [82] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye, “Interpreting deep learning-based vulnerability detector predictions based on heuristic searching,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.
- [83] <https://docs.alliancecan.ca/wiki/Narval/en/>.
- [84] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, “Deep graph library: A graph-centric, highly-performant package for graph neural networks,” *arXiv preprint arXiv:1909.01315*, 2019.
- [85] K. E. Bennin, J. Keung, A. Monden, P. Phannachitta, and S. Mensah, “The significant effects of data sampling approaches on software defect prioritization and classification,” in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017.
- [86] L. Gong, S. Jiang, and L. Jiang, “Tackling class imbalance problem in software defect prediction through cluster-based over-sampling with filtering,” *IEEE Access*, 2019.
- [87] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, “Mahakil: Diversity based oversampling approach to alleviate the class imbalance

- issue in software defect prediction,” *IEEE Transactions on Software Engineering*, 2017.
- [88] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, “The impact of class rebalancing techniques on the performance and interpretation of defect prediction models,” *IEEE Transactions on Software Engineering*, 2018.
- [89] P. Hart, “The condensed nearest neighbor rule (corresp.),” *IEEE transactions on information theory*, 1968.
- [90] <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6766>.
- [91] L. Van der Maaten and G. Hinton, “Visualizing data using t-sne.” *Journal of machine learning research*, 2008.
- [92] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.
- [93] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset,” *Empirical Software Engineering*, 2017.
- [94] M. Gharehyazie, B. Ray, and V. Filkov, “Some from here, some from there: Cross-project code reuse in github,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017.
- [95] C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam, “Pyexplainer: Explaining the predictions of just-in-time defect mod-

- els,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021.
- [96] M. M. Rahman, I. Ceka, C. Mao, S. Chakraborty, B. Ray, and W. Le, “Towards causal deep learning for vulnerability detection,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024.
- [97] X.-C. Wen, C. Gao, F. Luo, H. Wang, G. Li, and Q. Liao, “Livable: exploring long-tailed classification of software vulnerability types,” *IEEE Transactions on Software Engineering*, 2024.
- [98] S. Wan, J. Saxe, C. Gomes, S. Chennabasappa, A. Rath, K. Sun, and X. Wang, “Bridging the gap: A study of ai-based vulnerability management between industry and academia,” *arXiv preprint arXiv:2405.02435*, 2024.
- [99] S. Christey, J. Kenderdine, J. Mazella, and B. Miles, “Common weakness enumeration,” *Mitre Corporation*, 2013.
- [100] MITRE, “Cwe top 25 most dangerous software weaknesses,” 2024. [Online]. Available: <https://cwe.mitre.org/top25/>
- [101] S. E. Yuksel, J. N. Wilson, and P. D. Gader, “Twenty years of mixture of experts,” *IEEE transactions on neural networks and learning systems*, 2012.
- [102] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “Ac/c++ code vulnerability dataset with code changes and cve summaries,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.

-
- [103] *MoEVD replication package*. Zenodo, 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.11661787>
- [104] X. Yang, S. Wang, Y. Li, and S. Wang, “Does data sampling improve deep learning-based vulnerability detection? yeas! and nays!” in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2023.
- [105] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, “Graphcodebert: Pre-training code representations with data flow,” *arXiv preprint arXiv:2009.08366*, 2020.
- [106] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “Unixcoder: Unified cross-modal pre-training for code representation,” *arXiv preprint arXiv:2203.03850*, 2022.
- [107] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [108] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, “Regvd: Revisiting graph neural networks for vulnerability detection,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022.
- [109] M. Aota, H. Kanehara, M. Kubo, N. Murata, B. Sun, and T. Takahashi, “Automation of vulnerability classification from its description using machine learn-

- ing,” in *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2020.
- [110] M. Fu, V. Nguyen, C. K. Tantithamthavorn, T. Le, and D. Phung, “Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types,” *IEEE Transactions on Software Engineering*, 2023.
- [111] X. Zhou, K. Kim, B. Xu, J. Liu, D. Han, and D. Lo, “The devil is in the tails: How long-tailed code distributions impact large language models,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023.
- [112] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal loss for dense object detection,” in *Proceedings of the IEEE international conference on computer vision*, 2017.
- [113] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer,” *arXiv preprint arXiv:1701.06538*, 2017.
- [114] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE transactions on neural networks*, 2008.
- [115] H. He and E. A. Garcia, “Learning from imbalanced data,” *IEEE Transactions on knowledge and data engineering*, 2009.
- [116] G. Biau and E. Scornet, “A random forest guided tour,” *Test*, 2016.

-
- [117] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016.
- [118] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, “Catboost: unbiased boosting with categorical features,” *Advances in neural information processing systems*, 2018.
- [119] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, “Lightgbm: A highly efficient gradient boosting decision tree,” *Advances in neural information processing systems*, 2017.
- [120] Y. Zhang, W. Dou, J. Zhu, L. Xu, Z. Zhou, J. Wei, D. Ye, and B. Yang, “Learning to detect table clones in spreadsheets,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.
- [121] R. Szczepanek, “Daily streamflow forecasting in mountainous catchment using xgboost, lightgbm and catboost,” *Hydrology*, 2022.
- [122] C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz, “A comparative analysis of gradient boosting algorithms,” *Artificial Intelligence Review*, 2021.
- [123] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, “Huggingface’s transformers: State-of-the-art natural language processing,” *arXiv preprint arXiv:1910.03771*, 2019.
- [124] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen,

- Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, 2019.
- [125] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, “Autogluon-tabular: Robust and accurate automl for structured data,” *arXiv preprint arXiv:2003.06505*, 2020.
- [126] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” *arXiv preprint arXiv:1711.05101*, 2017.
- [127] focal loss, “Focal loss documentation,” https://focal-loss.readthedocs.io/en/latest/generated/focal_loss.binary_focal_loss.html, 2024, accessed: 2024-06-08.
- [128] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, “Autogluon-tabular: Robust and accurate automl for structured data,” *arXiv preprint arXiv:2003.06505*, 2020.
- [129] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, “Gshard: Scaling giant models with conditional computation and automatic sharding,” *arXiv preprint arXiv:2006.16668*, 2020.
- [130] W. Fedus, B. Zoph, and N. Shazeer, “Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity,” *Journal of Machine Learning Research*, 2022.
- [131] “Microsoft’s approach to coordinated vulnerability disclosure,” <https://www.microsoft.com/en-us/msrc/cvd>, accessed: 2024-02-05.

-
- [132] “Working with repository security advisories,” <https://docs.github.com/en/code-security/security-advisories/working-with-repository-security-advisories>, accessed: 2024-02-05.
- [133] T. Walshe and A. C. Simpson, “Coordinated vulnerability disclosure programme effectiveness: Issues and recommendations,” *Computers & Security*, 2022.
- [134] F. Li and V. Paxson, “A large-scale empirical study of security patches,” in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.
- [135] N. Imtiaz, A. Khanom, and L. Williams, “Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages,” *IEEE Transactions on Software Engineering (TSE)*, 2022.
- [136] B. Lin, S. Wang, M. Wen, and X. Mao, “Context-aware code change embedding for better patch correctness assessment,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022.
- [137] Y. Li, S. Wang, and T. N. Nguyen, “Dlfix: Context-based code transformation learning for automated program repair,” in *Proceedings of the 42nd ACM/IEEE international conference on software engineering (ICSE)*, 2020.
- [138] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” *Advances in neural information processing systems*, 2013.

-
- [139] A. A. Bangash, H. Sahar, A. Hindle, and K. Ali, “On the time-based conclusion stability of cross-project defect prediction models,” *Empirical Software Engineering*, 2020.
- [140] H. Jahanshahi, D. Jothimani, A. Başar, and M. Cevik, “Does chronology matter in jit defect prediction? a partial replication study,” in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, 2019.
- [141] Anonymous, “Vfdelta vulnerability fix dataset,” 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.10901854>
- [142] X. Zhou, D. Han, and D. Lo, “Assessing generalizability of codebert,” in *Proceedings of the 37th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021.
- [143] C. Tao, Q. Zhan, X. Hu, and X. Xia, “C4: Contrastive cross-language code clone detection,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2022.
- [144] Y. Shi, Y. Yin, Z. Wang, D. Lo, T. Zhang, X. Xia, Y. Zhao, and B. Xu, “How to better utilize code graphs in semantic code search?” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.
- [145] E. Mashhadi and H. Hemmati, “Applying codebert for automated program

- repair of java simple bugs,” in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 2021.
- [146] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, “Cc2vec: Distributed representations of code changes,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020.
- [147] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, “Evaluating representation learning of code changes for predicting patch correctness in program repair,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.
- [148] Z. Liu, Z. Tang, X. Xia, and X. Yang, “Ccrep: Learning code change representations via pre-trained code model and query back,” in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2023.
- [149] L. Mou, R. Men, G. Li, Y. Xu, L. Zhang, R. Yan, and Z. Jin, “Natural language inference by tree-based convolution and heuristic matching,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, K. Erk and N. A. Smith, Eds., 2016.
- [150] J. Zhou, S. Wang, C.-P. Bezemer, Y. Zou, and A. E. Hassan, “Studying the association between bountysource bounties and the issue-addressing likelihood of github issue reports,” *IEEE Transactions on Software Engineering (TSE)*, 2020.

- [151] R. Li, L. B. allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. LI, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, J. Lamy-Poirier, J. Monteiro, N. Gontier, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. T. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, U. Bhattacharyya, W. Yu, S. Luccioni, P. Villegas, F. Zhdanov, T. Lee, N. Timor, J. Ding, C. S. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. V. Werra, and H. de Vries, “Starcoder: may the source be with you!” *Transactions on Machine Learning Research (TMLR)*, 2023.
- [152] F. Cheirdari and G. Karabatis, “Analyzing false positive source code vulnerabilities using static analysis tools,” in *Proceedings of the 4th IEEE International Conference on Big Data (Big Data)*. IEEE, 2018.
- [153] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities,” in *Proceedings of the 26th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2006.
- [154] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, “The stack: 3 tb of permissively licensed source code,” *Preprint*, 2022.
- [155] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software

- developers use static analysis tools to find bugs?” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.
- [156] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, “Deepjit: an end-to-end deep learning framework for just-in-time defect prediction,” in *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 2019.
- [157] T. N. Y. Times, “Equifax to Pay at Least \$650 Million in Largest-Ever Data Breach Settlement,” 2019, accessed 2024-08-19. [Online]. Available: <https://www.nytimes.com/2019/07/22/business/equifax-settlement.html>
- [158] OpenSSF, “Guide to coordinated vulnerability disclosure for open source software projects,” 2024, accessed 2024-07-26. [Online]. Available: <https://github.com/ossf/oss-vulnerability-guide>
- [159] S. Pan, J. Zhou, F. R. Cogo, X. Xia, L. Bao, X. Hu, S. Li, and A. E. Hassan, “Automated unearthing of dangerous issue reports,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.
- [160] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, “Helping developers help themselves: Automatic decomposition of code review changesets,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.
- [161] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, and Y. Xiang, “Cd-

- vuld: Cross-domain vulnerability discovery based on deep domain adaptation,” *IEEE Transactions on Dependable and Secure Computing*, 2020.
- [162] J. Leinonen, P. Denny, S. MacNeil, S. Sarsa, S. Bernstein, J. Kim, A. Tran, and A. Hellas, “Comparing code explanations created by students and large language models,” in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 2023.
- [163] R. Fang, R. Bindu, A. Gupta, Q. Zhan, and D. Kang, “Teams of llm agents can exploit zero-day vulnerabilities,” *arXiv preprint arXiv:2406.01637*, 2024.
- [164] N. T. Islam, J. Khoury, A. Seong, G. D. L. T. Parra, E. Bou-Harb, and P. Najafirad, “Llm-powered code vulnerability repair with reinforcement learning and semantic reward,” *arXiv preprint arXiv:2401.03374*, 2024.
- [165] Z. Li, X. Zhang, Y. Zhang, D. Long, P. Xie, and M. Zhang, “Towards general text embeddings with multi-stage contrastive learning,” *arXiv preprint arXiv:2308.03281*, 2023.
- [166] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, “Unleashing the potential of prompt engineering in large language models: a comprehensive review,” *arXiv preprint arXiv:2310.14735*, 2023.
- [167] G. Docs, “GitHub REST API documentation,” 2022, accessed 2024-08-19. [Online]. Available: <https://docs.github.com/en/rest?apiVersion=2022-11-28>
- [168] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards,

- Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [169] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *Advances in Neural Information Processing Systems*, 2024.
- [170] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, “Measuring massive multitask language understanding,” *arXiv preprint arXiv:2009.03300*, 2020.
- [171] J. Zhou, T. Lu, S. Mishra, S. Brahma, S. Basu, Y. Luan, D. Zhou, and L. Hou, “Instruction-following evaluation for large language models,” *arXiv preprint arXiv:2311.07911*, 2023.
- [172] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, “The llama 3 herd of models,” *arXiv preprint arXiv:2407.21783*, 2024.
- [173] A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou, C. Li, C. Li, D. Liu, F. Huang *et al.*, “Qwen2 technical report,” *arXiv preprint arXiv:2407.10671*, 2024.
- [174] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, “Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence,” *arXiv preprint arXiv:2406.11931*, 2024.
- [175] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez,

- H. Zhang, and I. Stoica, “Efficient memory management for large language model serving with pagedattention,” in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- [176] N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, “Mteb: Massive text embedding benchmark,” *arXiv preprint arXiv:2210.07316*, 2022.
- [177] Chroma, “Chroma is the open-source ai application database. batteries included.” 2024, accessed 2024-09-12. [Online]. Available: <https://www.trychroma.com/>
- [178] B. W. Matthews, “Comparison of the predicted and observed secondary structure of t4 phage lysozyme,” *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.
- [179] T. G. Nguyen, T. Le-Cong, H. J. Kang, X.-B. D. Le, and D. Lo, “Vulcurator: a vulnerability-fixing commit detector,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.
- [180] S. Pan, L. Bao, X. Xia, D. Lo, and S. Li, “Fine-grained commit-level vulnerability type prediction by cwe tree structure,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.
- [181] T. G. Nguyen, T. Le-Cong, H. J. Kang, R. Widyasari, C. Yang, Z. Zhao, B. Xu, J. Zhou, X. Xia, A. E. Hassan *et al.*, “Multi-granularity detector for vulnerability fixes,” *IEEE Transactions on Software Engineering (TSE)*, 2023.

-
- [182] NVD, “Nvd - cve-2024-29199,” 2024, accessed 2024-08-25. [Online]. Available: <https://nvd.nist.gov/vuln/detail/CVE-2024-29199>
- [183] —, “Nvd - cve-2024-28103,” 2024, accessed 2024-08-25. [Online]. Available: <https://nvd.nist.gov/vuln/detail/CVE-2024-28103>
- [184] —, “Nvd - cve-2023-48014,” 2023, accessed 2024-08-25. [Online]. Available: <https://nvd.nist.gov/vuln/detail/CVE-2024-48014>
- [185] —, “Nvd - cve-2023-37061,” 2023, accessed 2024-08-25. [Online]. Available: <https://nvd.nist.gov/vuln/detail/CVE-2024-37061>
- [186] —, “Nvd - cve-2023-48657,” 2023, accessed 2024-08-25. [Online]. Available: <https://nvd.nist.gov/vuln/detail/CVE-2024-48657>
- [187] GitHub, “GitHub Advisory Database,” 2024, accessed 2024-08-19. [Online]. Available: <https://github.com/advisories>
- [188] M. Renze and E. Guven, “The effect of sampling temperature on problem solving in large language models,” *arXiv preprint arXiv:2402.05201*, 2024.
- [189] M. Peeperkorn, T. Kouwenhoven, D. Brown, and A. Jordanous, “Is temperature the creativity parameter of large language models?” *arXiv preprint arXiv:2405.00492*, 2024.