



# MONASH University

## **Toward More Effective Deep Learning-based Automated Software Vulnerability Prediction, Classification, and Repair**

Yeh Fu (aka. Michael Fu)  
Ph.D in Information Technology

A thesis submitted for the degree of Doctor of Philosophy at Monash University  
in 2025

Faculty of Information Technology

## Copyright notice

©Yeh Fu (aka. Michael Fu) (2025).

1. Under the Copyright Act 1968, this thesis must be used only under the normal conditions of scholarly fair dealing. In particular no results or conclusions should be extracted from it, nor should it be copied or closely paraphrased in whole or in part without the written consent of the author. Proper written acknowledgement should be made for any assistance obtained from this thesis.
2. I certify that I have made all reasonable efforts to secure copyright permissions for third-party content included in this thesis and have not knowingly added copyright content to my work without the owner's permission.

For publications included in Chapters 5, and 9 that have been published at IEEE venues:

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Monash University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to [http://www.ieee.org/publications\\_standards/publications/rights/rights\\_link.html](http://www.ieee.org/publications_standards/publications/rights/rights_link.html) to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

For publications included in Chapters 3, 6, 7, and 10 that have been published at ACM venues:

In reference to ACM copyrighted material which is used with permission in this thesis, the ACM does not endorse any of Monash University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing ACM copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to <https://authors.acm.org/author-resources/author-rights> to learn how to obtain a License from RightsLink. The definitive version of these works is available online at the ACM Digital Library (<https://dl.acm.org/>).

For publications included in Chapters 8 that have been published at Springer venues:

In reference to Springer copyrighted material which is used with permission in this thesis, Springer does not endorse any of Monash University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing Springer copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to <https://www.springer.com/gp/rights-permissions/obtaining-permissions/882> to learn how to obtain the necessary permissions. The final authenticated version of these works is available online at the SpringerLink platform (<https://link.springer.com/>).

## Abstract

Software vulnerabilities are prevalent in modern systems, often leading to severe consequences such as system failures or data breaches if left unresolved. To mitigate these risks within the DevSecOps (Development, Security, and Operations) framework, it is crucial to detect vulnerable code, identify the specific type of vulnerability, and propose timely repairs. Recently, deep learning (DL)-based approaches have been introduced to automate these tasks, including software vulnerability detection (SVD), classification (SVC), and neural machine translation (NMT)-based automated vulnerability repair (AVR). However, these approaches face significant challenges, such as the coarse-grained predictions of SVD, unresolved data imbalance in SVC, and inaccuracies in AVR, highlighting the need for further research and improvement.

In this thesis, we aim to support security analysts and developers by providing accurate automated predictions for detecting vulnerabilities, identifying their types, and proposing corresponding repairs. We first introduce LineVul, a transformer-based line-level SVD approach. Evaluated on a large-scale dataset of over 188k C/C++ functions, LineVul achieves 160%-379% higher F1-measure at the function level and 12%-25% higher Top-10 Accuracy at the line level compared to baselines, demonstrating its potential to significantly reduce analysts' workload. We also propose OptiMatch, a novel technique using vector quantization (VQ) and optimal transport (OT) to accurately locate function- and line-level vulnerabilities by learning hidden patterns.

For SVC, we introduce VulExplainer, a hierarchical distillation approach designed to mitigate data imbalance in SVC while explaining detected vulnerabilities. Evaluated on a dataset of 8,636 real-world C/C++ vulnerabilities, it outperforms all baselines by 5%-29%, proving effective with transformer models like CodeBERT, GraphCodeBERT, and CodeGPT. We also propose a multi-objective optimization (MOO) approach to further improve multi-task learning accuracy in SVC.

We then present VulRepair, a T5-based AVR approach, and VQM, inspired by the Vision Transformer (ViT), to focus on vulnerable code during repair generation. Our experiments on two datasets with over 5k real-world C/C++ vulnerability fixes show that VQM achieves a Percentage of Perfect Prediction of 46%, outperforming baselines by 3%-32%, and successfully repairing well-known vulnerabilities like Use After Free and OS Command Injection.

We seamlessly integrated our innovations – LineVul, VulExplainer, and VQM approaches – into a practical DL-based software security tool for C/C++ languages named AIBugHunter. AIBugHunter is available free of charge within the widely-used integrated development environment (IDE), VSCode. When provided with a C or C++ file, AIBugHunter performs three essential functions: (1) detecting vulnerabilities at both function and line levels, (2) providing vulnerability type explanations for the detected vulnerabilities, and (3) suggesting repair patches to address the detected vulnerabilities.

This thesis advances DL-based methods for SVD, SVC, and AVR within the software development phase of DevSecOps. To broaden future research, we conduct a systematic literature review (SLR) that examines AI-driven security approaches across the DevSecOps lifecycle. We analyzed 99 papers from top software engineering and security venues, identifying 12 key security tasks, evaluating AI-driven methods, and reviewing 64 benchmarks. We uncovered 15 challenges and proposed future research directions. This SLR informs the current work and lays the foundation for expanding research beyond this thesis.

## **Declaration**

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: .....

Print Name: Yeh Fu.

Date: .....

---

## Publications during enrolment

- **Michael Fu** and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In Proceedings of the 19th International Conference on Mining Software Repositories (MSR). 608–620.
- **Michael Fu**, Trung Le, Van Nguyen, Chakkrit Tantithamthavorn, and Dinh Phung. 2024. Optimal Transport for Line-Level Vulnerability Detection Under Review at IEEE Transactions on Dependable and Secure Computing (TDSC).
- **Michael Fu**, Van Nguyen, Chakkrit Kla Tantithamthavorn, Trung Le, and Dinh Phung. 2023. Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types. IEEE Transactions on Software Engineering (TSE) 49, 10, 4550–4565.
- **Michael Fu**, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2024. AIBugHunter: A practical tool for predicting, classifying, and repairing software vulnerabilities. Empirical Software Engineering (EMSE) 29, 4.
- **Michael Fu**, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-based automated software vulnerability repair. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE). 935–947.
- **Michael Fu**, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. ACM Transactions on Software Engineering and Methodology (TOSEM) 33, 1–29.
- **Michael Fu**, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for vulnerability detection, classification, and repair: How far are we? In Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC). 632–636. IEEE.
- **Michael Fu**, Jirat Pasuksmit, and Chakkrit Tantithamthavorn. 2025. AI for DevSecOps: A Landscape and Future Opportunities. Accepted at ACM Transactions on Software Engineering and Methodology (TOSEM).

## Thesis including published works declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

This thesis includes 3 original papers published in peer reviewed journals and 3 original papers published in peer reviewed conferences. The core theme of the thesis is *Deep Learning (DL)-based Software Vulnerability Detection, Classification, and Repair*. The ideas, development, and writing up of all the papers in the thesis were the principal responsibility of myself, the student, working within the *Doctor of Philosophy (0190)* under the supervision of *Dr. Chakkrit (Kla) Tantithamthavorn*.

The inclusion of co-authors reflects the fact that the work came from active collaboration between researchers and acknowledges input into team-based research.

In the case of Chapter 3, 5, 6, 7, 8, 9, and 10 my contribution to the work involved the following:

Thesis Chapter	Publication Title	Status published, in press, accepted or returned for revision, submitted	Nature and % of student contribution	Co-author name(s) Nature and % of Co-author's contribution*	Co-author(s), Monash student Y/N*
3	<i>Linevul: A transformer-based line-level vulnerability prediction</i>	<i>Published in MSR 2022 [1]</i>	50%. Concept and writing first draft and conducting experiments	1) Chakkrit (Kla) Tantithamthavorn, input into manuscript 50%	No
5	<i>Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types</i>	<i>Published in TSE 2023 [2]</i>	50%. Concept and writing first draft and conducting experiments	1) Van Nguyen, input into manuscript 15% 2) Chakkrit (Kla) Tantithamthavorn, input into manuscript 15% 3) Trung Le, input into manuscript 10% 4) Dinh Phung, input into manuscript 10%	1) No 2) No 3) No 4) No

6	<i>VulRepair: a T5-based automated software vulnerability repair</i>	<i>Published in FSE 2022 [3]</i>	50%. Concept and writing first draft and conducting experiments	1) Chakkrit (Kla) Tantithamthavorn, input into manuscript 20% 2) Trung Le, input into manuscript 10% 3) Van Nguyen, input into manuscript 10% 4) Dinh Phung, input into manuscript 10%	1) No 2) No 3) No 4) No
7	<i>Vision transformer inspired automated vulnerability repair</i>	<i>Published in TOSEM 2024 [4]</i>	50%. Concept and writing first draft and conducting experiments	1) Van Nguyen, input into manuscript 15% 2) Chakkrit (Kla) Tantithamthavorn, input into manuscript 15% 3) Dinh Phung, input into manuscript 10% 4) Trung Le, input into manuscript 10%	1) No 2) No 3) No 4) No
8	<i>AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities</i>	<i>Published in EMSE 2023 [5]</i>	50%. Concept and writing first draft and conducting experiments and tool implementation	1) Chakkrit (Kla) Tantithamthavorn, input into manuscript 15% 2) Trung Le, input into manuscript 10% 3) Yuki Kume, tool implementation 10% 4) Van Nguyen, input into manuscript 5% 5) Dinh Phung, input into manuscript 5% 6) John Grundy, input into manuscript 5%	1) No 2) No 3) Yes 4) No 5) No 6) No
9	<i>Chatgpt for vulnerability detection, classification, and repair: How far are we?</i>	<i>Published in APSEC 2023 [6]</i>	50%. Concept and writing first draft and conducting experiments	1) Chakkrit (Kla) Tantithamthavorn, input into manuscript 40% 2) Van Nguyen, input into manuscript 5% 3) Trung Le, input into manuscript 5%	1) No 2) No 3) No
10	<i>AI for DevSecOps: A Landscape and Future Opportunities</i>	<i>Accepted in TOSEM [7]</i>	50%. Conducting systematic literature review and writing first draft	1) Jirat Pasuksmit, input into manuscript 25% 2) Chakkrit (Kla) Tantithamthavorn, input into manuscript 25%	1) No 2) No

I have not renumbered sections of submitted or published papers in order to generate a consistent presentation within the thesis.

---

**Student name: Yeh Fu (aka. Michael Fu)**

**Student signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

I hereby certify that the above declaration correctly reflects the nature and extent of the student's and co-authors' contributions to this work. In instances where I am not the responsible author I have consulted with the responsible author to agree on the respective contributions of the authors.

**Main Supervisor name: Chakkrit (Kla) Tantithamthavorn**

**Main Supervisor signature:** \_\_\_\_\_ **Date:** \_\_\_\_\_

---

## Acknowledgements

First and foremost, I would like to extend my deepest gratitude to my supervisor, Dr. Chakkrit (Kla) Tantithamthavorn. Over the past three years, I have greatly enjoyed working with Kla, an exceptional mentor throughout my PhD journey. He has taught me invaluable skills, from writing papers and conducting experiments to handling author rebuttals and paper revisions. Kla has continually inspired me with interesting research ideas and has fostered a rich environment of knowledge sharing. He also guided me in preparing for job applications, sharing information on job opportunities, and providing detailed feedback on my teaching and research statements, CV, and cover letter. Kla has gone above and beyond what is expected of a PhD supervisor, and I am deeply grateful for his unwavering support and encouragement throughout this journey. Thank you, Kla.

I would also like to extend my heartfelt thanks to my co-supervisor, Dr. Trung Le. I was fortunate to have Trung join as my co-supervisor in 2022, as I was eager to learn about deep learning, and Trung brought a wealth of knowledge and numerous innovative ideas to the table. As a newcomer to AI and deep learning, I benefited immensely from Trung's guidance and expertise. I vividly remember our first submission to an AI conference. Trung spent significant time with me, meticulously editing the paper to ensure it was well-structured, with rigorous formulas and notations. Through this experience, I learned so much about becoming an AI researcher. Most importantly, Trung's innovative thinking has always inspired me. One of my favorite parts of our collaboration was our weekly meetings, where we brainstormed and discussed new ideas for deep learning. These sessions were incredibly enriching and motivating, and I always looked forward to them. Trung has truly been a source of inspiration, and I have gained invaluable problem-solving skills in deep learning from him. Thank you, Trung.

I would like to express my gratitude to Dr. Van Nguyen for guiding me in writing AI papers and providing valuable advice whenever needed. I am also thankful to Professor Rashina Hoda for serving as the chair during my PhD milestone presentation, and to Professor Aldeida Aleti and Dr. Xiaoning Du for their roles as panel members. Their constructive feedback and suggestions have been invaluable to my PhD journey and in improving the quality of this thesis. Additionally, I would like to thank Julie Holden for teaching me the fundamental skills of conducting research and paper writing, as well as the FIT-GR staff for all their academic support. I am also grateful to my fellow research team members, Chanathip Pornprasit, Wannita Takerngsaksiri, and Yue Liu, who provided useful guidance

---

and took the time to discuss with me, helping me navigate my PhD journey. I would also like to thank Yuki Kume, a Bachelor of Computer Science (Honors) student at Monash, who played a crucial role in the success of our AIBugHunter project with his significant contributions and support.

I would like to extend a special thank you to Professor John Grundy, who supported me greatly during the revision of one of my papers. I learned a lot from his perspective on how to effectively address reviewers' questions and get straight to the point. John was also incredibly helpful with my job applications, providing constructive feedback on my application materials and serving as a reference. I am also deeply grateful to Professor Dinh Phung for inviting me to present in his research lab, allowing me to learn a great deal from other researchers on his team. Dinh has been very supportive of my future career, and I appreciate all his encouragement and guidance.

Next, I would like to thank the open-source contributors of Huggingface Transformers and PyTorch. Without their dedication to maintaining and implementing these tools, the work in this thesis would not have been possible. I am also grateful to Stanford University for making their lectures publicly available, especially CS231N (Convolutional Neural Networks for Visual Recognition) and CS224W (Machine Learning with Graphs), which helped me build a solid foundation in deep learning during my PhD studies. Additionally, I would like to thank DeepLearning.AI and Andrew Ng for providing free material that allowed me to learn state-of-the-art deep learning techniques.

Last but not least, I want to dedicate this paragraph to my family, friends, and mentor. To my partner, V, who came to Australia with me in 2020, your unwavering support throughout my Master's and PhD journey has been invaluable. I truly appreciate your time, patience, and companionship as we navigated this experience together. We have been through so many ups and downs that I could not possibly list them all, but the biggest thank you is undoubtedly for you, V. To my mother, Winnie, who has been both a wonderful mom and a great friend, your constant support has been crucial for my success in Australia. You have always offered your opinions and experiences at the right moments, helping us keep moving forward on our journey here. To my grandmother, Yue-Lian, who raised me and taught me countless life lessons through her actions and her role as a caring grandmother, I want to express my deepest gratitude to you. Though you are no longer with us, I hope you are at peace in the afterlife. To my brother, Jet, my cool and intelligent brother whom I always care about, thank you for being

---

there for me and for supporting me. I appreciate all our discussions and your willingness to share your thoughts. To my basketball coaches, Coach Hong and Coach Hong, who are brothers and have both been instrumental in my basketball and life journey, thank you for being my first inspiration in reading and critical thinking. I deeply appreciate the countless conversations we have had over the phone, where we shared ideas and learned together. Finally, a very special thank you to my mentor, Charlie Munger, whose wisdom and insight continue to inspire me. Munger's philosophy of striving each day to be a little wiser than you were when you woke up, as well as learning from others' experiences and becoming a reliable and moral individual, has profoundly inspired me. Munger once compared himself to Old Valiant-for-Truth in *The Pilgrim's Progress*, saying: "My sword I leave to him who can wear it." I hope that one day I will be worthy of wielding such a sword.

## Acknowledging the use of generative artificial intelligence

I used ChatGPT (gpt-3.5-turbo/gpt-4o-mini/gpt-4o) [8] to help polish the English and rephrase some of the paragraphs in this thesis to improve fluency and grammar. The tool was used to provide suggestions for rephrasing and grammatical improvements across multiple drafts. The output from this tool was carefully reviewed and modified to align with the intended meaning and context of the content.

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>29</b>
1.1	Problem Statement . . . . .	29
1.2	Overview of Research . . . . .	31
1.2.1	Thesis Goal . . . . .	31
1.2.2	Software Vulnerability Detection . . . . .	31
1.2.3	Software Vulnerability Classification . . . . .	32
1.2.4	Automated Vulnerability Repair . . . . .	32
1.2.5	A DL-based Software Security Tool . . . . .	33
1.2.6	Contributions to Knowledge . . . . .	33
1.2.7	Thesis Overview . . . . .	37
<b>2</b>	<b>Background &amp; Literature Review – Challenges in DL-Based Software Vulnerability Detection, Classification, and Repair</b>	<b>44</b>
2.1	DL-based Software Vulnerability Detection . . . . .	44
2.2	DL-based Software Vulnerability Classification . . . . .	45
2.3	DL-based Automated Vulnerability Repair . . . . .	46
2.4	Static Application Security Testing (SAST) Tools in IDEs . . . . .	47
2.5	ChatGPT for Software Vulnerability Predictions . . . . .	47
2.6	AI for DevSecOps: A Systematic Literature Review for Future Research Directions Beyond This Thesis . . . . .	48
<b>3</b>	<b>Locating Vulnerabilities – A Novel Transformer-Based Approach for Pinpointing Line-Level Software Vulnerabilities</b>	<b>49</b>
3.1	Introduction . . . . .	50
3.2	Background . . . . .	52
3.2.1	IVDetect: A state-of-the-art fine-grained vulnerability prediction & Limitations . . . . .	53
3.3	LineVul: A Line-Level Vulnerability Prediction Approach . . . . .	56
3.3.1	Function-level Vulnerability Prediction . . . . .	57
3.3.2	Line-level Vulnerability Localization . . . . .	59
3.4	Experimental Design . . . . .	60
3.4.1	Research Questions . . . . .	60
3.4.2	Studied Dataset . . . . .	61
3.4.3	Experimental Setup . . . . .	61
3.5	Experimental Results . . . . .	62
3.6	Discussion . . . . .	69
3.6.1	Why LineVul performs so well for vulnerability predictions? . . . . .	69

---

3.6.2	How accurate is our LineVul for predicting the Top-25 Most Dangerous CWEs?	71
3.7	Related Work	72
3.7.1	DL-based Vulnerability Prediction	72
3.7.2	Line-Level Vulnerability Prediction	73
3.7.3	Explainable AI for SE	73
3.8	Summary	74
<b>4</b>	<b>Learning Vulnerability Patterns – Matching Line-Level Vulnerabilities with Optimal Transport (OT) and Vector Quantization (VQ)</b>	<b>76</b>
4.1	Introduction	77
4.2	Background & Related Work	80
4.2.1	Optimal Transport	82
4.2.2	Vector Quantization	83
4.2.3	Related Work	83
4.2.3.1	Deep Learning-based Vulnerability Detection	83
4.2.3.2	Large Language Models for Vulnerability Detection	84
4.3	Approach	85
4.3.1	Problem Statement	85
4.3.2	Line Embedding Using RNN	86
4.3.3	Training of Warm-Up Phase	87
4.3.4	Quantizing Vulnerability Vectors: Optimal Transport and Subsequent Main Training Phase	89
4.3.4.1	Collect Vulnerability Vectors from Vulnerable Functions	89
4.3.4.2	Learn to Transport Vulnerability Vectors to Vulnerability Centroids	89
4.3.4.3	Main Training Phase	90
4.3.5	Vulnerability Detection Through Explicit Vulnerability Patterns Matching	90
4.4	Experimental Design and Results	91
4.4.1	Research Questions	91
4.4.2	Baseline Approaches	92
4.4.3	Experimental Datasets	93
4.4.4	Parameter Settings and Model Training	93
4.5	Discussion	98
4.5.1	Does our RNN line embedding method perform better on long sequences than the token embedding?	98

---

4.5.2	Does our optimal transport process effectively aggregate vulnerability vectors into representative vulnerability centroids? . . . . .	99
4.5.3	Can we use clustering algorithms to automatically identify an ideal number of centroids for our OptiMatch approach? . . . . .	101
4.6	Threats to Validity . . . . .	102
4.7	Summary . . . . .	103
<b>5</b>	<b>A Novel Knowledge Distillation Framework for Explaining Detected Vulnerabilities</b> . . . . .	<b>105</b>
5.1	Introduction . . . . .	106
5.2	Background & Problem Statement . . . . .	109
5.2.1	Background . . . . .	109
5.2.1.1	CWE Abstract Types . . . . .	111
5.2.1.2	Knowledge Distillation . . . . .	111
5.2.2	Challenge & Motivation . . . . .	116
5.2.2.1	Preliminary Analysis . . . . .	116
5.2.3	Problem Statement . . . . .	117
5.3	Our proposed framework . . . . .	118
5.3.1	Grouping Source Codes into the Groups with the Same CWE Abstract Types . . . . .	119
5.3.2	Training Multiple TextCNN Teachers . . . . .	120
5.3.3	Hierarchical Transformer-based Distillation . . . . .	120
5.4	Experimental Design and Results . . . . .	122
5.4.1	Research Questions . . . . .	122
5.4.2	Baseline approaches . . . . .	123
5.4.3	Experimental dataset . . . . .	125
5.4.4	Parameter Setting . . . . .	125
5.4.5	Experimental Results . . . . .	126
5.5	Discussion . . . . .	131
5.5.1	What is the effect of using a language model pre-trained on code for our CWE-ID classification task? . . . . .	132
5.5.2	What is the performance of our TextCNN teacher model? . . . . .	132
5.5.3	What are the effects of our hierarchical knowledge distillation method on a transformer student model? . . . . .	133
5.6	Related work . . . . .	134
5.7	Threats to Validity . . . . .	136
5.8	Summary . . . . .	137

---

<b>6 Repair Vulnerabilities with Language Models (LMs)</b>	<b>138</b>
6.1 Introduction . . . . .	139
6.2 Background & Problem Motivation . . . . .	141
6.3 VulRepair: A T5-based Vulnerability Repair Approach . . . . .	144
6.3.1 Code Representation . . . . .	144
6.3.2 VulRepair Model Architecture . . . . .	146
6.3.3 Vulnerability Repair Generation . . . . .	148
6.4 Experimental Design . . . . .	148
6.4.1 Research Questions . . . . .	148
6.4.2 Studied Dataset . . . . .	149
6.4.3 Experimental Setup . . . . .	150
6.5 Experimental Results . . . . .	151
6.6 Discussion . . . . .	158
6.6.1 What Types of CWEs that Our VulRepair Can Correctly Repair? . . . . .	158
6.6.2 How Do the Function Lengths and Repair Lengths Impact the Accuracy of Our VulRepair? . . . . .	159
6.6.3 How Does the Complexity of the Input Functions Impact the Accuracy of Our VulRepair? . . . . .	161
6.6.4 How Well Can Our VulRepair Handle the OOV Problem of Vulnerability Repairs? . . . . .	162
6.7 Related Work . . . . .	163
6.8 Threats to Validity . . . . .	164
6.9 Summary . . . . .	165
<b>7 When Vulnerability Repair Meets Computer Vision – A Vision Transformer-Inspired Approach for Guiding Transformer Models to Focus on Vulnerable Code Areas and Generate Accurate Repair Patches</b>	<b>166</b>
7.1 Introduction . . . . .	167
7.2 Our Proposed Approach . . . . .	170
7.2.1 Usage Scenario . . . . .	170
7.2.2 Problem statement . . . . .	172
7.2.3 Vulnerability Repair Via Vulnerability Query and Mask . . . . .	173
7.2.3.1 Vulnerability Repair Encoder . . . . .	175
7.2.3.2 Vulnerability Repair Decoder . . . . .	175
7.2.3.3 Learning and applying vulnerability mask . . . . .	177
<b>Learning vulnerability mask</b> . . . . .	177
<b>Applying vulnerability mask to our model.</b> . . . . .	178

---

7.3	Experimental Design . . . . .	179
7.3.1	Research Questions . . . . .	179
7.3.2	Baseline approaches . . . . .	180
7.3.3	Experimental Dataset . . . . .	181
7.3.4	Parameter Setting . . . . .	182
7.3.5	Model Training . . . . .	182
7.4	Experimental results . . . . .	182
7.5	Discussion . . . . .	187
7.5.1	Can our VQM repair the common dangerous vulnerability types (i.e., CWE-IDs)? . . . . .	188
7.5.2	How does our VQM perform across different vulnerability types that have different data frequencies in our experimental dataset? . . . . .	188
7.5.3	Does our proposed vulnerability mask help highlight vulnerable code areas during vulnerability repair? . . . . .	192
7.5.4	Can our proposed vulnerability mask be applied to decoder-only transformer architectures? . . . . .	193
7.6	A User Study of AI-generated Vulnerability Repairs . . . . .	193
7.6.1	Survey Design . . . . .	194
7.6.2	Survey Results . . . . .	196
7.7	Related work . . . . .	203
7.8	Threats to Validity . . . . .	204
7.9	Summary . . . . .	205
8	<b>AIBugHunter – A Practical Vulnerability Analysis Tool for Locating, Explaining, Estimating, and Repairing Software Vulnerability</b>	207
8.1	Introduction . . . . .	208
8.2	AIBugHunter: Our Approach . . . . .	210
8.2.1	AIBugHunter security tool . . . . .	210
8.2.2	Example Usage . . . . .	212
8.2.3	AIBugHunter Implementation . . . . .	213
8.2.3.1	Front-End Implementation . . . . .	213
8.2.3.2	Back-End Implementation . . . . .	214
8.3	Learning to predict vulnerability type and severity . . . . .	215
8.3.1	Multi-Objective CWE Classification . . . . .	216
8.3.1.1	Sequence Representation . . . . .	217
8.3.1.2	Two Non-Shared Classification Heads . . . . .	217
8.3.1.3	Multi-Objective Optimization . . . . .	218

---

8.3.2	CVSS Severity Score Estimation . . . . .	219
8.4	A Quantitative Evaluation of AIBugHunter . . . . .	220
8.4.1	Research Questions . . . . .	220
8.4.2	Studied Dataset . . . . .	221
8.4.3	Experimental Setup . . . . .	221
8.4.4	Experimental Results . . . . .	223
8.5	Qualitative Evaluations of AIBugHunter . . . . .	232
8.5.1	A Qualitative Survey Study . . . . .	232
8.5.1.1	Survey Design . . . . .	232
8.5.1.2	Survey Results . . . . .	233
8.5.2	A Preliminary User Study . . . . .	237
8.5.2.1	User Study Design . . . . .	238
8.5.2.2	User Study Results . . . . .	239
8.5.3	The implications of AIBugHunter to researchers and practitioners . . . . .	239
8.6	Threats to Validity . . . . .	240
8.6.1	Construct Validity . . . . .	240
8.6.2	Internal Validity . . . . .	241
8.6.3	External Validity . . . . .	241
8.7	Related Work . . . . .	241
8.7.1	ML-Based Vulnerability Type Classification . . . . .	241
8.7.2	Multi-Task Learning for Software Vulnerability Prediction . . . . .	242
8.8	Summary . . . . .	243
<b>9</b>	<b>ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?</b> . . . . .	<b>244</b>
9.1	Introduction . . . . .	245
9.2	Related Work . . . . .	246
9.3	Problem Statement & Prompt Design . . . . .	247
9.3.1	Prompt ChatGPT for Software Vulnerability Prediction . . . . .	247
9.3.2	Prompt ChatGPT for Software Vulnerability Classification . . . . .	247
9.3.3	Prompt ChatGPT for Vulnerability Severity Estimation . . . . .	249
9.3.4	Prompt ChatGPT for Automated Vulnerability Repair . . . . .	250
9.4	Experimental Design and Results . . . . .	250
9.4.1	Experimental Datasets . . . . .	250
9.4.2	Parameter Settings and Execution Environment . . . . .	252
9.4.3	Experimental Results . . . . .	252
9.5	Summary . . . . .	256

---

<b>10 Paving the Road Ahead: AI-driven Security in DevSecOps – Current Landscape, Challenges, and Future Research Opportunities</b>	<b>257</b>
10.1 Introduction . . . . .	258
10.2 Background and Related Work . . . . .	260
10.2.1 DevOps . . . . .	260
10.2.2 Other Reviews in DevSecOps . . . . .	264
10.3 Approach . . . . .	265
10.3.1 Research Questions . . . . .	265
10.3.2 Literature Search Strategy . . . . .	266
10.3.3 Literature Selection: Inclusion-Exclusion Criteria and Quality Assessment Criteria . . . . .	268
10.3.4 Snowballing Search . . . . .	270
10.3.5 Data Extraction and Analysis . . . . .	270
10.3.6 Data Synthesis and Mapping . . . . .	272
10.4 RQ1: AI Methods Overview For DevSecOps . . . . .	273
10.4.1 Plan . . . . .	274
10.4.1.1 Threat Modeling . . . . .	274
10.4.1.2 Software Impact Analysis . . . . .	274
10.4.2 Development . . . . .	275
10.4.2.1 Software Vulnerability Detection . . . . .	277
10.4.2.2 Software Vulnerability Classification . . . . .	282
10.4.2.3 Automated Vulnerability Repair . . . . .	284
10.4.2.4 Security Tools in IDEs. . . . .	289
10.4.3 Code Commit . . . . .	291
10.4.3.1 Dependency Management . . . . .	291
10.4.3.2 CI/CD Secure Pipelines . . . . .	293
10.4.4 Build, Test, and Deployment . . . . .	296
10.4.4.1 Configuration Validation . . . . .	297
10.4.4.2 Infrastructure Scanning . . . . .	299
10.4.5 Operation and Monitoring . . . . .	300
10.4.5.1 Log Analysis and Anomaly Detection . . . . .	301
10.4.5.2 Cyber-Physical Systems . . . . .	305
10.5 RQ2: Challenges and Research Opportunities in AI-Driven DevSecOps . . . . .	306
10.5.1 Common Challenges . . . . .	307
10.5.2 Plan . . . . .	313
10.5.2.1 Threat Modeling & Impact Analysis . . . . .	313
10.5.3 Development . . . . .	314

---

10.5.3.1 Software Vulnerability Detection . . . . .	314
10.5.3.2 Software Vulnerability Classification . . . . .	316
10.5.3.3 Automated Vulnerability Repair . . . . .	317
10.5.4 Code Commit . . . . .	319
10.5.4.1 Dependency Management . . . . .	319
10.5.4.2 CI/CD Secure Pipelines . . . . .	320
10.5.5 Build, Test, and Deployment . . . . .	321
10.5.5.1 Configuration Validation . . . . .	321
10.5.5.2 Infrastructure Scanning . . . . .	322
10.5.6 Operation and Monitoring . . . . .	323
10.5.6.1 Log Analysis and Anomaly Detection . . . . .	323
10.5.6.2 Cyber-Physical Systems . . . . .	324
10.6 Threats to Validity . . . . .	325
10.7 Summary . . . . .	327
<b>11 Conclusion</b>	<b>327</b>

---

## List of Tables

2	The contribution of each component of LineVul for function-level vulnerability predictions. . . . .	70
3	(Discussion) The Accuracy of our LineVul for the Top-25 Most Dangerous CWEs ( <a href="https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html">https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html</a> ). . . . .	71
4	(RQ1 Results) We compare our OptiMatch approach against 12 other baseline methods and present results in percentage. . . . .	95
5	(RQ2 Results) We compare our proposed method to other variants to investigate the impact of the individual components. The metrics are reported as percentages. . . . .	96
6	(Discussion) The line-level performance comparison between the token embedding and our RNN line embedding. . . . .	99
7	(Discussion) The comparison between our approach with randomly initialized vulnerability centroids and our approach with OPTICS clustering algorithm to determine the number of vulnerability centroids. . . . .	101
8	The training schemes of teacher and student models in our VulExplainer approach. . . . .	125
9	(RQ1 results) The multi-class accuracy of our proposed method and each baseline approach. We present CWE-ID classification results for each group of CWE abstract types and the overall result. Measure using multi-class accuracy shown in percentage. The weighted F1 is also presented in percentage, which considers the class imbalance. A description of each CWE abstract type can be found on the official CWE website [152]. . . . .	126
10	(RQ2 results) The experimental results when comparing our proposed approach with other loss-based methods for the data imbalance problem. We measure the accuracy of CWE-ID classification using multi-class accuracy shown in percentage. The weighted F1 is also presented in percentage, which considers the class imbalance. (FL - Focal Loss, LA - Logit Adjustment). . . . .	128
11	(RQ3 Results) The experimental results of the ablation study to investigate the three key steps in our VulExplainer method. We conduct the ablation study for all three transformer models, i.e., GCB - GraphCodeBERT, CB - CodeBERT, and GPT - CodeGPT. Relative improvement is presented for comparison. . . . .	129

---

12	(Discussion) The comparison between TextCNN Teacher, CodeBERT <sub>VulExplainer</sub> , and CodeBERT. We measure the accuracy of CWE-ID classification using multi-class accuracy shown in percentage. . . . .	132
13	(Discussion) Performance analysis of CodeBERT <sub>VulExplainer</sub> and CodeBERT on three different testing subsets. Measure using multi-class accuracy shown in percentage. CNNT - CNNTeacher, CB - CodeBERT, <i>VulExp</i> - VulExplainer . . . . .	133
14	Descriptive statistics of the studied dataset. . . . .	149
15	(Discussion) The % Perfect Predictions of our VulRepair for the Top-10 Most Dangerous CWEs. . . . .	159
16	(Discussion) The % Perfect Predictions of our VulRepair according to the function length and the repair length. . . . .	162
17	Training scheme of our VQM approach. Note. #: Scheme for training the mask prediction model; *: Scheme for training the repair model. . . . .	182
18	(Main results) The comparison between our VQM approach and other baselines. Accuracy is presented in percentage. Beam=k shows the measure of %PP. We conducted the experiments five times with different random seeds and reported the mean performance plus minus standard deviation. . . . .	183
19	(Ablation results) The comparison between our proposed method and four other variants. Accuracy is presented in percentage. . . . .	185
20	Caption . . . . .	186
21	The %PP of our VQM approach across the top 25 most dangerous CWE-IDs in 2022. The %PP is shown based on the beam search results where Beam=5. . . . .	187
22	(Discussion Results) The effectiveness of applying our vulnerability masks to a decoder-only transformer architecture. . . . .	193
23	Descriptive statistics of our studied datasets that describes the distribution of the severity score, and the distributions of cardinalities of CWE-ID and CWE-Type. . . . .	222
24	(RQ1 Discussion) The Accuracy of our approach for the Top-25 Most Dangerous CWEs ( <a href="https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html">https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html</a> ). . . . .	225
25	(RQ2) The Multiclass Accuracy of each CWE type for all of the approaches evaluated in RQ2. . . . .	227

---

26	(RQ2 Discussion) The analysis of how different function lengths affect the multi-class accuracy of our approach for CWE-ID and CWE-Type prediction tasks. Note. Function lengths counted by number of tokens in a tokenized function. . . . .	227
27	Comparison with other related reviews focusing on DevSecOps. #P - total number of reviewed papers, SLR - Is the study a systematic literature review?, A - Does the study focus on the machine learning and deep learning approaches for DevSecOps?, B - Does the study encompass all steps in DevSecOps? . . . . .	264
28	The overview of the selected software engineering and security conferences and journals. . . . .	266
29	The inclusion-exclusion criteria and quality assessment criteria. . . . .	269
30	An overview of AI-driven approaches for security-related tasks in the development step. . . . .	276
31	Benchmarks used in evaluating AI-driven software vulnerability detection. . . . .	276
32	Benchmarks used in evaluating AI-driven vulnerability type classification. . . . .	282
33	Benchmarks used in evaluating AI-driven program and vulnerability repairs. . . . .	285
34	An overview of AI-driven approaches for security-related tasks in the code commit step. . . . .	291
35	Benchmarks used in evaluating AI-driven just-in-time (JIT) software defect prediction. . . . .	293
36	An overview of AI-driven approaches for security-related tasks in build, test, and deployment steps. . . . .	297
37	Benchmarks used in evaluating AI-driven configuration validation. . . . .	297
38	Benchmarks used in evaluating AI-driven infrastructure security scanning. . . . .	299
39	An overview of AI-driven approaches for security-related tasks in the operation and monitoring step. . . . .	300
40	Benchmarks used in evaluating AI-driven log analysis and anomaly detection. . . . .	301
41	Benchmarks used in evaluating AI-driven cyber-physical system (CPS) security approaches. . . . .	304
42	(RQ2) The overview of the 15 challenges and future research opportunities derived from previous studies. . . . .	307

---

## List of Figures

1	An overview of the thesis structure.	38
2	A motivating example of our LineVul vs IVDetect.	54
3	An overview architecture of our LineVul.	56
4	(RQ1) The experimental results of our LineVul and the seven baseline comparisons for function-level vulnerability prediction. (↗) Higher F1, Precision, Recall = Better.	63
5	(RQ2) The Top-10 Accuracy and IFA of our self-attention approach and five other methods. (↗) Higher Top-10 Accuracy = Better, (↘) Lower IFA = Better.	67
6	(RQ3) The Effort@20%Recall and Recall@1%LOC of our self-attention approach and five other methods. (↘) Lower Effort@20%Recall = Better, (↗) Higher Recall@1%LOC = Better.	69
7	In both functions, the CWE-787 (Out-of-Bound Write) vulnerability is triggered by an inappropriate data type assignment. In the <i>unPremulSkImageToPremul</i> [9] function above, the <i>size_t</i> type should be changed to the <i>unsigned</i> type. In the <i>isValidSize</i> [10] function below, the <i>int_32t</i> should be changed to <i>uint_32t</i> to prevent potential buffer overflow. Despite sharing the same vulnerability type and pattern, the vulnerable lines in each function and their context are different in their written form, variable names, and positions.	78
8	Caption	81
9	Caption	88
10	Caption	88
11	Caption	100
12	A real-world vulnerability example of CWE-787 [163]. The upper part shows the vulnerability prediction generated by line-level SVP models while the lower part presents the same prediction with an extended explanation provided by the SVC approaches to illustrate the detected vulnerability.	113
13	Statistics that measure the imbalance of grouped and ungrouped data distributions.	114

---

14	An example of prompting Google BARD to identify CWE-ID based on the input of a C++ function. BARD mistakenly describes CWE-754 (Improper Check for Unusual or Exceptional Conditions) as CWE-20 (Improper Input Validation). Furthermore, the actual vulnerability associated with the input function is CWE-787 as described in Figure 12. . . . .	115
15	The overview architecture of our VulExplainer during knowledge distillation. The left part describes the inference process of TextCNN teachers. The CWE-IDs are grouped hierarchically based on the CWE abstract types $g_i$ . A tied TextCNN backbone is connected with multiple classification heads, where each head predicts CWE-IDs belonging to their own CWE abstract type. The right part illustrates the training process of the student model. A distillation token [dis] and a [cls] token are added to the input $F_i$ to learn from the knowledge of teachers and ground-truth labels respectively. The representation of $F_i$ forwards through a 12-layer GraphCodeBERT. Finally, the student relies on a KL loss to learn the representation of [dis] by distilling knowledge from predictions of the teacher models, and a CE loss to learn the representation of [cls] token from ground-truth labels. . . . .	119
16	An overview architecture of our VulRepair. . . . .	144
17	(RQ1) The experimental results of our VulRepair and the two baseline comparisons for vulnerability repairs. (↗) Higher % Perfect Predictions = Better. . . . .	152
18	(RQ2) The experimental results of the ablation study with six different models. (↗) Higher % Perfect Predictions = Better. . . . .	153
19	(RQ3) The experimental results of various approaches with different tokenization techniques for vulnerability repairs. (↗) Higher %Perfect Predictions = Better. . . . .	155
20	(RQ4) The ablation study result of VulRepair. (↗) Higher %Perfect Predictions = Better. . . . .	157
21	(Discussion) The %Perfect Predictions (y-axis) of our VulRepair according to each type of CWE (x-axis, sorted by % perfect predictions and sorted by the majority of CWEs in the dataset). Detailed statistics can be found in Appendix. . . . .	160
22	(Discussion) The accuracy of our VulRepair for various ranges of the Cyclomatic Complexity of the input vulnerable functions in the testing set. (↗) Higher % Perfect Predictions = Better. . . . .	161

---

23	Caption . . . . .	170
24	Caption . . . . .	171
25	Caption . . . . .	172
26	An overview architecture of our VQM approach. Input tokens $x_i = [t_1, \dots, t_n]$ and vulnerability masks $M(x_i)$ are input to encoders that output the embeddings of input tokens $H_{enc}^L$ , where $M(x_i)$ helps to emphasize the vulnerable embeddings. In decoders, each vulnerability query $VQ_i$ is initialized from the previous repair token $r_{i-1}$ , which is forwarded through multiple decoder layers followed by a linear layer to generate a repair token $r_i$ . In each decoder, a cross-attention with $M(x_i)$ to emphasize vulnerable embeddings is leveraged to cross-match $VQ_i$ and $H_{enc}^L$ and generate repairs corresponding to the vulnerable tokens. . . . .	174
27	Caption . . . . .	179
28	The performance analysis of our VQM accross different CWE-IDs. The bar chart represents the %PP while the blue line is the training frequency and the red line is the testing frequency across all vulnerability types. Note that the ticks of the Y axis on the left are for the %PP metric while those on the right are for the data frequency of each CWE-ID. . . . .	188
29	The comparison of the cross-attention weights of our VQM approach and the variant, VQM w/o VM. The red bars indicate the cross-attention weights for vulnerable code areas while the green bars indicate the cross-attention weights for benign code areas. . . . .	190
30	The visualization of the cross-attention weights between embeddings of generated repair patches (i.e., Vulnerability Queries - axis X) and embeddings of vulnerable functions (axis Y). The ground-truth vulnerable code areas are highlighted in red boxes. It can be seen that the cross-attention map is more highly activated for vulnerable code areas when applying vulnerability masks (VMs). In other words, our VMs help better distinguish the vulnerable and benign code areas. . . . .	191
31	Caption . . . . .	194
32	Caption . . . . .	196
33	Caption . . . . .	197
34	Caption . . . . .	198
35	The user interface of our AIBugHunter. . . . .	211
36	The back-end implementation of our AIBugHunter. . . . .	213

---

37	An overview architecture of our approach. . . . .	215
38	An overview architecture of multi-objective CWE classification. . . . .	216
39	Two concrete examples of high and low CVSS severity scores. . . . .	219
40	(RQ1 and RQ2) The Multiclass Accuracy of our approach and four other baselines. ( $\nearrow$ ) Higher Multiclass Accuracy = Better. . . . .	223
41	(RQ1 Discussion) Our method's Multiclass Accuracy of CWE-ID classification for each CWE-ID in the testing set. The accuracy is shown in percentage. . . . .	226
42	The Multiclass Accuracy of our approach, our approach w/o MOO, and single-task CodeBERT. ( $\nearrow$ ) Higher Multiclass Accuracy = Better.	227
43	(RQ3) The Mean Squared Error (MSE) and Mean Absolute Error (MAE) of our approach and three other baselines. ( $\searrow$ ) Lower MSE, MAE = Better. . . . .	230
44	(RQ3 Discussion) The left part is the multi-class accuracy of the CVSS score for each approach evaluated in RQ3. The right part is the confusion matrix of our approach. Note that each class of CVSS is directly mapped from the CVSS score as shown at the bottom of the confusion matrix table. . . . .	230
45	(Q1-Q4) A summary of the survey questions and the results obtained from 21 participants. . . . .	234
46	The demographics of our survey participants in terms of their profession and professional experience. . . . .	234
47	The experimental results of our user study with six participants. Wherein the first task was to locate the vulnerability, the second task was to explain the vulnerability type, the third task was to estimate the vulnerability severity, and the fourth task was to suggest repairs. The time was measured in minutes and the satisfaction ranged from 1 (highly dissatisfied) to 5 (highly satisfied). . . . .	237
48	An example prompt for function and line-level vulnerability prediction. . . . .	248
49	An example prompt for CWE-ID classification. . . . .	249
50	An example prompt for severity estimation. . . . .	249
51	An example prompt for automated vulnerability repair. . . . .	251
52	(RQ1) The experimental results of function-level and line-level vulnerability prediction. ( $\nearrow$ ) For all metrics, higher = better. . . . .	253
53	(RQ2) The experimental results of vulnerability type (i.e., CWE-ID) classification. ( $\nearrow$ ) Higher Multiclass Accuracy = better. . . . .	254
54	(RQ3) The experimental results of vulnerability severity estimation. ( $\searrow$ ) Lower MSE, MAE = better. . . . .	255

---

55	(RQ4) The experimental results of automated vulnerability repair. (↗) Higher %PP, BLEU, METEOR = better.	256
56	Caption	261
57	Caption	268
58	Caption	271

---

# 1 Introduction

Software vulnerabilities are security flaws, glitches, or weaknesses found in software code [11] that could be exploited or triggered by attackers. Those unresolved vulnerable programs associated with critical software systems may result in extreme security or privacy risks. For instance, the recent data breaches of Optus [12] (a telecommunications company) and Medibank [13] (a private health insurer) in 2022 have put millions of customers' privacy in danger. Thus, software security is an essential aspect that needs to be considered during the development of software systems to prevent irreversible crises afterward.

Recently, researchers have proposed various Deep Learning (DL)-based software vulnerability detection (SVD) approaches that can predict whether a software program (e.g., a file or a function) is vulnerable by learning the patterns of source code [14–18]. In addition, software vulnerability classification (SVC) approaches have also been proposed to identify the vulnerability types of detected vulnerabilities [19–21]. SVCs can provide security analysts with more explanation, in-depth analysis, and mitigation strategies. Last but not least, the rise of automated vulnerability repair (AVR) using DL models opens a new possibility to suggest repairs for vulnerable programs automatically [22, 23], which may save more effort for security analysts during their patching process.

However, existing DL-based SVD, SVC, and AVR methods still have limitations that can be addressed to further improve their performance and applicability to be adopted in practice, which we will detail in Section 1.2. In what follows, we first give our problem statement and explain the purposes of the SVD, SVC, and AVR approaches.

## 1.1 Problem Statement

Cybersecurity Ventures expects global cybercrime costs to reach USD 10.5 trillion by 2025, up from USD 3 trillion in 2015 [24]. As cyberattacks become the main contributing factors to a revenue loss of some businesses [25], ensuring the safety of software systems becomes a critical challenge for private and public sectors.

To combat security issues during the implementation of software, program analysis (PA)-based tools [26–29] have been introduced to analyze source code statically based on predefined vulnerability patterns. Moreover, PA-based tools such

---

as Checkmarx [29] and CPP Check [28] have been integrated into different Integrated Development Environments (IDEs), hence can achieve real-time static application security testing (SAST) and support developers to identify potential vulnerabilities during development. However, recent studies have shown that deep learning (DL)-based software vulnerability detection (SVD) exhibits better performance than PA-based tools on both function-level [30] and fine-grained [31] vulnerability detection. In this research, we provide a complete automated DL-based pipeline including the following three aspects to support SAST:

**Software Vulnerability Detection (SVD)** - DL-based vulnerability detection approaches learn vulnerability patterns from historical data via supervised learning. Similar to PA-based tools, SVDs rely on DL models to analyze source code statically without the need for code compilation. Thus, SVDs can also be integrated into IDEs as a SAST tool for software developers and security analysts. In particular, various SVDs have been proposed that can predict vulnerabilities on either file or function levels [14–18]. Nevertheless, a file or a function can still consist of multiple lines of code, where end users may need to manually inspect to locate the exact vulnerabilities. In our research, we mainly develop SVD methods that can predict fine-grained vulnerabilities (e.g., line-level vulnerability) to effectively reduce the effort of manual analysis.

**Software Vulnerability Classification (SVC)** - As SVD approaches can only locate vulnerabilities but fail to explain the detected vulnerabilities, SVC approaches were proposed to identify the vulnerability type of detected vulnerability and provide extended descriptions such as the likelihood of exploitation and mitigation strategies. Previous studies focused on classifying vulnerability types based on the vulnerability descriptions [20, 21], which require vulnerability descriptions to be available and limit their usage scenario. In our research, we focus on classifying vulnerability types based on source code input instead of vulnerability descriptions. Thus, our SVC approach can be deployed with SVD approaches as part of the SAST tool to identify the vulnerability types and provide explanations for the vulnerability detected by SVDs.

**Automated Vulnerability Repair (AVR)** - As mentioned above, researchers have proposed various DL-based approaches to help under-resourced security analysts predict vulnerability [14–18], localize the location of vulnerabilities [32], and classify the vulnerability types [19–21] to obtain the potential impact of the vulnerability and the corresponding mitigation strategy based on its type. However, security analysts still have to spend a huge amount of effort manually fixing or re-

---

pairing vulnerabilities after the detection[33, 34]. Thus, researchers proposed to leverage neural machine translation (NMT)-based methods which consist of encoders to encode vulnerable functions and decoders to generate corresponding repairs [22, 23, 35, 36]. In our research, we define the AVR problem as an NMT problem and explore large pre-trained language models. Moreover, we focus on developing an AVR approach that can be deployed as part of the SAST tool to accurately suggest repair patches to security analysts in IDEs.

Last but not least, PA-based tools such as CPP Check [28] and Checkmarx [29] have been integrated into the software development life cycle. However, DL-based automated vulnerability prediction approaches are still not available for security analysts and developers. To bridge this gap, we integrate our research works from SVD, SVC, and AVR tasks and extend them to a DL-based software security tool that can predict, localize, classify, and suggest repairs for software vulnerabilities.

## 1.2 Overview of Research

In this section, we present the overview of our research.

### 1.2.1 Thesis Goal

The overarching goal of this research is to develop deep learning-based methodologies that enhance the accuracy and effectiveness of automated software vulnerability detection, classification, and repair, thereby providing security analysts with a security analysis tool in IDE for identifying, understanding, and addressing vulnerabilities early during software development. To achieve this goal, we explore prior studies to understand the problems in each vulnerability prediction task and raise the research questions to address the problems.

### 1.2.2 Software Vulnerability Detection

DL-based software vulnerability detection (SVD) approaches have been proposed to predict vulnerable code at a file or function-level [14–18]. However, these DL-based approaches can only generate coarse-grained predictions where a vulnerable file or function may consist of multiple lines of code that need to be manually inspected by security analysts. In contrast, PA-based tools can elaborate on the fine-grained details of code and pinpoint specific lines of code that might be involved in the detected vulnerability. To address the need for fine-grained vulnerability detection and enhance the level of DL-based vulnerability

---

detection, we raise the following research question:

**RQ1:** How can intelligent approaches effectively pinpoint the vulnerable lines of code involved in detected vulnerabilities?

### 1.2.3 Software Vulnerability Classification

DL-based software vulnerability classification (SVC) approaches have been proposed to identify vulnerability types (i.e., CWE-ID [37]) [19, 20]. In the real world, some vulnerability types can be common while others might be rare to find. Thus, the SVC task is a long-tailed learning task where DL models need to combat highly imbalanced data during training to achieve promising performance on both common and rare vulnerability types. Previous studies tried mitigating this challenge using data augmentation [20] or ignoring rare CWE-IDs [19]. However, the performance did not improve and the imbalance problem remains unresolved. To address the data imbalance issue and provide more accurate SVC methods to security analysts, we raise the following research question:

**RQ2:** Can we help security analysts identify vulnerability types more accurately to provide further explanations of the detected vulnerabilities?

### 1.2.4 Automated Vulnerability Repair

The aforementioned DL-based vulnerability prediction approaches only help security analysts detect, localize, and identify the types of vulnerabilities. However, security analysts still have to spend much effort manually fixing or repairing vulnerabilities [33, 34, 38]. Thus, researchers proposed DL-based automated vulnerability repair (AVR) approaches that treat vulnerability as a neural machine translation (NMT) problem and generate the repair patches for vulnerable programs automatically [22, 23]. Nevertheless, existing AVR approaches have limitations such as using word-level tokenization and non-pretrained transformer models, which lead to inaccurate predictions. Moreover, current transformer-based AVR lacks a mechanism to guide the encoders and decoders to focus more on vulnerable code areas while generating corresponding repair patches. To address those limitations and improve the performance of AVR models, we raise the following research question:

**RQ3:** Can we help security analysts suggest repair patches for vulnerable code more accurately?

---

### 1.2.5 A DL-based Software Security Tool

PA-based tools such as CPP Check [28] and Checkmarx [29] have been integrated into the software development life cycle and identify security issues in the early stage of development. Even if DL-based methods could locate, classify, and repair vulnerability and outperform PA-based tools, they have not been integrated into modern IDEs such as VSCode [39], hence not available for security analysts and developers. To bridge the gap between DL-based vulnerability approaches and developers, we raise the following research question:

**RQ4:** Can we help security analysts detect and localize vulnerabilities, identify vulnerability types, and suggest corresponding repair patches during the early stage of software development?

### 1.2.6 Contributions to Knowledge

The nine contributions of this thesis are presented below:

**Contribution 1:** We present LineVul, a transformer-based line-level vulnerability detection approach to address several limitations of the state-of-the-art IVDetect [32] approach. Through an empirical evaluation of a large-scale real-world dataset with 188k+ C/C++ functions, we show that LineVul achieves (1) 160%-379% higher F1-measure for function-level predictions; (2) 12%-25% higher Top-10 Accuracy for line-level predictions; and (3) 29%-53% less Effort@20%Recall than the baseline approaches, highlighting the significant advancement of LineVul towards more accurate and more cost-effective line-level vulnerability detection. The training code and pre-trained models are available at <https://github.com/awsm-research/LineVul>. This work has been published in the following paper:

Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In Proceedings of the 19th International Conference on Mining Software Repositories (MSR). 608–620.

The contributions of this paper build an important step towards line-level software vulnerability detection, making DL-based vulnerability detection more accurate and capable of precisely locating vulnerabilities at the line level.

**Contribution 2:** We propose using vector quantization (VQ) and optimal transport (OT) to capitalize on the observed cluster characteristics of vulnerable line

---

embeddings and further enhance model performance. We present a vulnerability-matching technique to leverage the learned vulnerability clusters and identify vulnerable functions and lines during inference. We evaluated our approach against 12 competitive baseline approaches on two large-scale vulnerability datasets of diverse vulnerabilities parsed from real-world projects. Our approach achieves the highest F1 score for function and line-level VD. The training code and pre-trained models are available at <https://github.com/awsm-research/optimatch>. This work is under review at IEEE Transactions on Dependable and Secure Computing (TDSC) and the pre-print is available in arXiv:

Michael Fu, Trung Le, Van Nguyen, Chakkrit Tantithamthavorn, and Dinh Phung. 2023. Learning to quantize vulnerability patterns and match to locate statement-level vulnerabilities. arXiv preprint arXiv:2306.06109.

The contributions of this paper build an important step towards using vector quantization (VQ) and optimal transport (OT) for more accurately learning vulnerability patterns and detecting software vulnerabilities at both the function and line levels.

**Contribution 3:** We introduce a transformer-based hierarchical distillation for software vulnerability classification to address the highly imbalanced types of software vulnerabilities. Through an extensive evaluation using the real-world 8,636 vulnerabilities, our approach outperforms all of the baselines by 5%–29%. The results also demonstrate that our approach can be applied to transformer-based architectures such as CodeBERT, GraphCodeBERT, and CodeGPT. Moreover, our method maintains compatibility with any transformer-based model without requiring any architectural modifications but only adds a special distillation token to the input. The training code and pre-trained models are available at <https://github.com/awsm-research/VulExplainer>. This work has been published in the following paper:

Michael Fu, Van Nguyen, Chakkrit Kla Tantithamthavorn, Trung Le, and Dinh Phung. 2023. VulExplainer: A transformer-based hierarchical distillation for explaining vulnerability types. IEEE Transactions on Software Engineering (TSE) 49, 10, 4550–4565.

The contributions of this paper build an important step towards providing more accurate vulnerability explanations and mitigating data imbalance issues, which address both fundamental and practical challenges in explaining software vulnerabilities.

---

**Contribution 4:** We present a novel multi-objective optimization (MOO)-based vulnerability classification approach and a transformer-based estimation approach to explain vulnerability types and estimate severity accurately. Our empirical experiments on a large dataset consisting of 188K+ C/C++ functions confirm that our proposed approaches are more accurate than other state-of-the-art baseline methods for vulnerability type classification and estimation. The training code and pre-trained models are available at <https://github.com/awsm-research/AIBugHunter>. This work has been published in the following paper:

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2024. AIBugHunter: A practical tool for predicting, classifying, and repairing software vulnerabilities. Empirical Software Engineering (EMSE) 29, 4.

The contributions of this paper build an important step towards improving multi-task learning for more accurate vulnerability type classification and severity estimation.

**Contribution 5:** We present VulRepair, a T5-based automated software vulnerability repair approach that leverages the pre-training and BPE components to address various technical limitations of prior work. Through an extensive experiment with over 8,482 vulnerability fixes from 1,754 real-world software projects, we find that our VulRepair achieves a Perfect Prediction of 44%, which is 13%-21% more accurate than competitive baseline approaches. The training code and pre-trained models are available at <https://github.com/awsm-research/VulRepair>. This work has been published in the following paper:

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-based automated software vulnerability repair. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE). 935–947.

The contributions of this paper represent an important step towards more accurate NMT-based automated vulnerability repair by developing a refined method using a language model and thoroughly examining the effects of pre-training and subword tokenization.

**Contribution 6:** We present a novel vulnerability repair framework inspired by the Vision transformer-based approaches for object detection in the computer

---

vision domain. Through an extensive evaluation using the real-world 5,417 vulnerabilities, our approach outperforms all of the automated vulnerability repair baseline methods by 2.68% to 32.33%. The training code and pre-trained models are available at <https://github.com/awsml-research/VQM>. This work has been published in the following paper:

Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. ACM Transactions on Software Engineering and Methodology (TOSEM) 33, 1–29.

The contributions of this paper build an important step towards enhancing transformer encoder-decoder models for automated vulnerability repair via our proposed vulnerability masking mechanism.

**Contribution 7:** We propose AIBugHunter, a novel deep learning-based software vulnerability analysis tool for C/C++ languages that is integrated into the Visual Studio Code (VS Code) IDE. AIBugHunter helps software developers achieve real-time vulnerability detection, explanation, and repairs during programming. We conduct qualitative evaluations including a survey study and a user study to obtain software practitioners' perceptions of our AIBugHunter tool and assess the impact that AIBugHunter may have on developers' productivity in security aspects. Our survey study shows that our AIBugHunter is perceived as useful where 90% of the participants consider adopting our AIBugHunter during their software development. Last but not least, our user study shows that our AIBugHunter can enhance developers' productivity in combating cybersecurity issues during software development. AIBugHunter is now publicly available in the VS Code marketplace at <https://marketplace.visualstudio.com/items?itemName=AIBugHunter.aibughunter>. This work has been published in the following paper:

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2024. AIBugHunter: A practical tool for predicting, classifying, and repairing software vulnerabilities. Empirical Software Engineering (EMSE) 29, 4.

The contributions of this paper build an important step towards advancing real-time software vulnerability detection, explanation, and repair in C/C++ using deep learning through the integration of AIBugHunter into the Visual Studio Code IDE.

---

**Contribution 8:** We undertake a comprehensive study by instructing ChatGPT for four prevalent vulnerability tasks: function and line-level vulnerability detection, vulnerability classification, severity estimation, and vulnerability repair. We compare ChatGPT with state-of-the-art language models designed for software vulnerability purposes. Through an empirical assessment employing extensive real-world datasets featuring over 190,000 C/C++ functions, we found that ChatGPT achieves limited performance, trailing behind other language models in vulnerability contexts by a significant margin. The studied dataset, experimental prompts for ChatGPT, and experimental results are available at <https://github.com/awsm-research/ChatGPT4Vul>. This work has been published in the following paper:

Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for vulnerability detection, classification, and repair: How far are we? In Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC). 632–636. IEEE.

The contributions of this paper build an important step towards understanding the generalizability of ChatGPT on various software vulnerability predictions.

**Contribution 9:** We analyze 99 research papers spanning from 2017 to 2023. Specifically, we address two key research questions (RQs). In RQ1, we identify 12 security tasks associated with the DevOps process and review existing AI-driven security approaches along with the benchmark used for evaluation. In RQ2, we discover 15 challenges encountered by existing AI-driven security approaches and derive future research opportunities. This work is under review at ACM Transactions on Software Engineering and Methodology (TOSEM) and the pre-print is available in arXiv:

Michael Fu, Jirat Pasuksmit, and Chakkrit Tantithamthavorn. 2024. AI for DevSecOps: A landscape and future opportunities. arXiv preprint arXiv:2404.04839.

The contributions of this paper build an important step towards understanding the current research landscape on AI-driven security approaches for DevSecOps, challenges encountered, and future research opportunities.

### 1.2.7 Thesis Overview

In this subsection, we outline an overview of this thesis summarized in Figure 1.



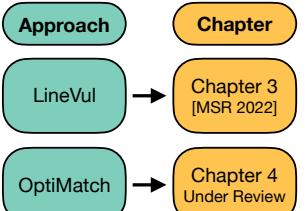
**Thesis Goal:** To develop deep learning-based methodologies that enhance the accuracy and effectiveness of automated software vulnerability detection, classification, and repair, thereby providing security analysts with a security analysis tool in IDE for identifying, understanding, and addressing vulnerabilities early during software development.

**RQ1:** Can we help security analysts pinpoint the lines of code involved in the detected vulnerabilities?

**Objective**

To develop more accurate software vulnerability detection to pinpoint line-level vulnerabilities for security analysts.

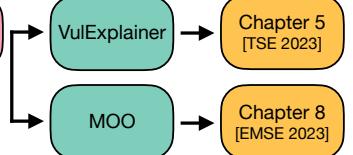
**Approach**



Contribution 1  
Contribution 2

**RQ2:** Can we help security analysts identify vulnerability types more accurately to provide further explanations of the detected vulnerabilities?

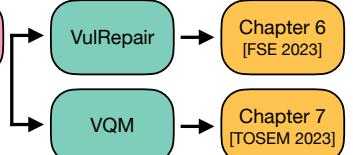
To develop more accurate software vulnerability classification to explain detected vulnerabilities for security analysts.



Contribution 3  
Contribution 4

**RQ3:** Can we help security analysts suggest repair patches for vulnerable code more accurately?

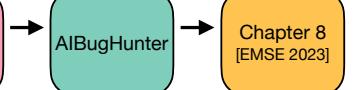
To develop more accurate automated vulnerability repair to suggest repair patches for security analysts.



Contribution 5  
Contribution 6

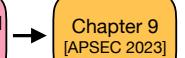
**RQ4:** Can we help security analysts detect and localize vulnerabilities, identify vulnerability types, and suggest corresponding repair patches during the early stage of software development?

To develop an IDE-integrated security analysis tool that localizes vulnerabilities, identifies their types, and suggests repair patches for software developers.



Contribution 7

To evaluate ChatGPT's performance for four vulnerability tasks: function and line-level vulnerability detection, classification, severity estimation, and vulnerability repair.



Contribution 8

To analyze AI-driven security approaches across the DevSecOps lifecycle, identify their limitations, and outline future research opportunities.



Contribution 9

Figure 1: An overview of the thesis structure.

---

## **[Chapter 2] Background & Literature Review – Challenges in DL-Based Software Vulnerability Detection, Classification, and Repair.**

In Chapter 2, we undertake a literature review of prior works related to software vulnerability detection (SVD), software vulnerability classification (SVC), and automated vulnerability repairs (AVR). Through this review, we examine existing works, identifying key research gaps and limitations that warrant further investigation. By highlighting these gaps, we provide a clear motivation for the proposals and research presented in subsequent chapters. The insights gained from this literature review serve as the foundational basis for the contributions made in Chapters 3 through 9, where we address the identified challenges with novel approaches and methodologies.

## **[Chapter 3] Locating Vulnerabilities – A Novel Transformer-based Approach for Pinpointing Line-Level Software Vulnerabilities.**

In Chapter 3, we present LineVul, a transformer-based line-level vulnerability detection approach designed to address three significant limitations of the state-of-the-art IVDetect method [32]. First, rather than relying on RNN-based models to generate code representations, LineVul utilizes a BERT architecture with self-attention layers, which are more effective at capturing long-term dependencies within lengthy sequences through dot-product operations. Second, instead of depending on project-specific training data, LineVul leverages the pre-trained CodeBERT [40] language model (LM) to generate robust vector representations of source code, enhancing the model’s generalizability. Third, rather than employing GNNExplainer to identify sub-graphs that contribute to predictions, LineVul uses the self-attention mechanism to pinpoint vulnerable lines at a finer granularity, offering a more precise vulnerability detection. Our experiments on a large-scale real-world dataset with 188k+ C/C++ functions show that LineVul achieves (1) 160%-379% higher F1-measure for function-level predictions; (2) 12%-25% higher Top-10 Accuracy for line-level predictions; and (3) 29%-53% less Effort@20%Recall than the baseline approaches. Our additional analysis also shows that our LineVul is also very accurate (75%-100%) for predicting vulnerable functions affected by the top 25 most dangerous CWEs, highlighting the potential impact of our LineVul in real-world usage scenarios.

## **[Chapter 4] Learning Vulnerability Patterns – Matching Line-Level Vulnerabilities with Optimal Transport (OT) and Vector Quantization (VQ).**

---

In Chapter 4, we present OptiMatch which uses vector quantization (VQ) and optimal transport (OT) to capitalize on the observed cluster characteristics of vulnerable line embeddings and further enhance model performance. We also introduce a vulnerability-matching technique to leverage the learned vulnerability clusters and identify vulnerable functions and lines during inference. Specifically, for each vulnerable function in the training data, we obtain embeddings of vulnerable lines and aggregate them into a flat vulnerability vector, resulting in 6,361 vulnerability vectors. Nevertheless, the large number of vulnerability vectors poses a computing burden to our vulnerability-matching process. By leveraging the identified cluster characteristics, we employ the VQ principle in conjunction with OT to redistribute the vulnerability vectors, aiming for a more condensed representation. This aggregation procedure facilitates the grouping of vulnerability vectors into cohesive patterns within the embedding space, resulting in the reduction of the original 6,361 vectors to a set of 150 common patterns. Inspired by static analysis tools that match pre-defined patterns to capture vulnerabilities, we match our learned patterns to pinpoint vulnerabilities during inference with DL models. We evaluated our approach against 12 competitive baselines on two large-scale vulnerability datasets, encompassing diverse vulnerabilities from real-world open-source projects. Our approach demonstrated its effectiveness in helping security analysts locate vulnerabilities, achieving the highest F1 score for both function and line-level VD.

## **[Chapter 5] A Novel Knowledge Distillation Framework for Explaining Detected Vulnerabilities.**

In Chapter 5, we present VulExplainer – an approach to explain the type of vulnerabilities. We represent VulExplainer as a vulnerability classification task. However, vulnerabilities have diverse characteristics (i.e., CWE-IDs) and the number of labeled samples in each CWE-ID is highly imbalanced (known as a highly imbalanced multi-class classification problem), which often leads to inaccurate predictions. Thus, we introduce a transformer-based hierarchical distillation for software vulnerability classification to address the highly imbalanced types of software vulnerabilities. Specifically, we split a complex label distribution into sub-distributions based on CWE abstract types (i.e., categorizations that group similar CWE-IDs). Thus, similar CWE-IDs can be grouped and each group will have a more balanced label distribution. We learn TextCNN teachers on each of the simplified distributions respectively, however, they only perform well in their group. Thus, we build a transformer student model to generalize the per-

---

formance of TextCNN teachers through our hierarchical knowledge distillation framework. Through an extensive evaluation using the real-world 8,636 vulnerabilities, our approach outperforms all of the baselines by 5%–29%. The results also demonstrate that our approach can be applied to transformer-based architectures such as CodeBERT, GraphCodeBERT, and CodeGPT. Moreover, our method maintains compatibility with any transformer-based model without requiring any architectural modifications but only adds a special distillation token to the input. These results highlight our significant contributions to the fundamental and practical problem of explaining software vulnerability.

## **[Chapter 6] Repair Vulnerabilities with Language Models (LMs).**

In Chapter 6, we present VulRepair – a T5-based automated software vulnerability repair approach that leverages the pre-training and BPE components to address various technical limitations of prior work. Our VulRepair employs a pre-training CodeT5 component from a large codebase (i.e., CodeSearchNet+C/C# with 8.35 million functions from 8 different Programming Languages) to generate more meaningful vector representation, employs BPE tokenization to handle Out-Of-Vocabulary (OOV) issues, and uses a T5 architecture that considers the relative position information in the self-attention mechanism. Through an extensive experiment with over 8,482 vulnerability fixes from 1,754 real-world software projects, we find that our VulRepair achieves a Perfect Prediction of 44%, which is 13%-21% more accurate than competitive baseline approaches. These results lead us to conclude that our VulRepair is considerably more accurate than the two baseline approaches, highlighting the substantial advancement of NMT-based Automated Vulnerability Repairs. Our additional investigation also shows that our VulRepair can accurately repair as many as 745 out of 1,706 real-world well-known vulnerabilities (e.g., Use After Free, Improper Input Validation, OS Command Injection), demonstrating the practicality and significance of our VulRepair for generating vulnerability repairs, helping under-resourced security analysts on fixing vulnerabilities.

## **[Chapter 7] When Vulnerability Repair Meets Computer Vision – A Vision Transformer-Inspired Approach for Guiding Transformer Models to Focus on Vulnerable Code Areas and Generate Accurate Repair Patches.**

In Chapter 7, we present VQM – Vulnerability Repair Through Vulnerability Query and Mask. While vulnerable programs may only consist of a few vulnerable code areas that need repair, we found that existing automated vulnerability repair

---

(AVR) approaches lack a mechanism guiding their model to pay more attention to vulnerable code areas during repair generation. Thus, we propose a novel vulnerability repair framework inspired by the Vision Transformer (ViT)-based approaches for object detection in the computer vision domain. Similar to the object queries used to locate objects in object detection in computer vision, we introduce and leverage vulnerability queries (VQs) to locate vulnerable code areas and then suggest their repairs. In particular, we leverage the cross-attention mechanism to achieve the cross-match between VQs and their corresponding vulnerable code areas. To strengthen our cross-match and generate more accurate vulnerability repairs, we propose to learn a novel vulnerability mask (VM) and integrate it into decoders' cross-attention, which makes our VQs pay more attention to vulnerable code areas during repair generation. In addition, we incorporate our VM into encoders' self-attention to learn embeddings that emphasize the vulnerable areas of a program. Through an extensive evaluation using the real-world 5,417 vulnerabilities, our approach outperforms all of the AVR baseline methods by 2.68% to 32.33%. Additionally, our analysis of the cross-attention map of our approach confirms the design rationale of our VM and its effectiveness. Finally, our survey study with 71 software practitioners highlights the usefulness of AI-generated vulnerability repairs for repairing security vulnerabilities.

## **[Chapter 8] AIBugHunter – A Practical Vulnerability Analysis Tool for Locating, Explaining, Estimating, and Repairing Software Vulnerability.**

In Chapter 8, we present AIBugHunter – a deep learning-powered real-time security analysis tool for C/C++ languages that detects and locates vulnerabilities, explains their types, estimates severity, and recommends repair patches, all within the Visual Studio Code (VS Code) IDE. We integrate our previous works, LineVul presented in Chapter 3 and VulRepair presented in Chapter 6, to achieve vulnerability localization and repairs. In this article, we present a novel multi-objective optimization (MOO)-based vulnerability classification approach and a transformer-based estimation approach to help AIBugHunter accurately identify vulnerability types and estimate severity. Our empirical experiments on a large dataset consisting of 188K+ C/C++ functions confirm that our proposed approaches are more accurate than other state-of-the-art baseline methods for vulnerability classification and estimation. Furthermore, we conduct qualitative evaluations including a survey study and a user study to obtain software practitioners' perceptions of our AIBugHunter tool and assess the impact that AIBugHunter may have on developers' productivity in security aspects. Our survey study shows that

---

our AIBugHunter is perceived as useful where 90% of the participants consider adopting our AIBugHunter during their software development. Last but not least, our user study shows that our AIBugHunter can enhance developers' productivity in combating cybersecurity issues during software development. AIBugHunter is now publicly available in the VS Code marketplace.

## **[Chapter 9] ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?**

In Chapter 9, we undertake a comprehensive study by instructing a large language model (LLM), ChatGPT (i.e., gpt-3.5-turbo and gpt-4), for four prevalent vulnerability tasks: function and line-level vulnerability detection, vulnerability classification, severity estimation, and vulnerability repair. We compare ChatGPT with state-of-the-art language models designed for software vulnerability purposes. Through an empirical assessment using extensive real-world datasets featuring over 190,000 C/C++ functions, we found that ChatGPT achieves limited performance, trailing behind other language models in vulnerability contexts by a significant margin. The experimental outcomes highlight the challenging nature of vulnerability prediction tasks, requiring domain-specific expertise. Despite ChatGPT's substantial model scale, exceeding that of source code-pre-trained language models (e.g., CodeBERT) by a factor of 14,000, the process of fine-tuning remains imperative for ChatGPT to generalize for vulnerability prediction tasks.

## **[Chapter 10] Paving the Road Ahead: AI-driven Security in DevSecOps – Current Landscape, Challenges, and Future Research Opportunities.**

In Chapter 10, we present a systematic literature review (SLR) to pave my future research directions by analyzing the current landscape of AI-driven security methods across the DevSecOps lifecycle, identifying existing challenges, and uncovering opportunities for advancing the integration of AI in securing every stage of DevSecOps. We analyze 99 research papers spanning from 2017 to 2023. Specifically, we address two key research questions (RQs). In RQ1, we identify 12 security tasks associated with the DevOps process and review existing AI-driven security methods along with the benchmark used for evaluation. In RQ2, we discover 15 challenges encountered by existing AI-driven security methods and derive future research opportunities. Drawing insights from our findings, we discussed cutting-edge AI-driven security methods, highlighted challenges in existing research, and proposed avenues for future opportunities.

---

## 2 Background & Literature Review – Challenges in DL-Based Software Vulnerability Detection, Classification, and Repair

In this chapter, we describe previous studies related to deep learning (DL)-based (1) software vulnerability detection, (2) software vulnerability classification, and (3) automated vulnerability repair. We then discuss existing software security tools that can be used early in the software development life cycle (e.g., a tool in developers' IDE) to inspect security issues. Based on the discussion of these technological studies, we define the scopes and novelties of this thesis.

### 2.1 DL-based Software Vulnerability Detection

Software vulnerability detection (SVD) is a task to predict vulnerable programs based on code-related metrics or the source code itself. For instance, machine learning (ML)-based vulnerability detection approaches leverage software metrics (e.g., code complexity, number of engineers, etc.) to predict vulnerable code [41, 42]. However, these approaches require a manual feature engineering process that is time-consuming. Moreover, some features may not be available during the early stage of software development, which hinders those approaches from being integrated into integrated development environments (IDEs).

On the other hand, deep learning (DL)-based approaches have been proposed to detect vulnerabilities using source code without any manual feature collection process. These methods embed code information (e.g., textual code, graph properties of code) into vector spaces and learn vulnerability patterns from historical data via supervised learning. For instance, Li et al. [16] proposed an RNN-based VulDeePecker model that is learned from different types of graph properties of source code (e.g., Data Dependency Graph). On the other hand, Zhou et al. [18] leveraged a graph neural network (GNN) to learn four types of graph properties of source code, i.e., Abstract Syntax Tree, Control Flow Graph, Data Flow Graph, and syntactic features. Chakraborty et al. [14] proposed Reveal, which is a gated graph neural network (GGNN) that learns the graph properties of source code.

However, these approaches detect at file or function levels, where a file or a function may consist of multiple lines of code, which may still require much inspection effort from security analysts. Thus, Chapter 3 and Chapter 4 will focus on the line-level vulnerability detection problem, aiming to provide security analysts with a more cost-effective and finer-grained prediction—which remains

---

largely unexplored.

## 2.2 DL-based Software Vulnerability Classification

Common Weakness Enumeration (CWE) is a community-developed list of software and hardware weaknesses and vulnerabilities [37]. CWE provides identification (i.e., CWE-ID) for each known vulnerability, where each CWE-ID consists of an extended description to explain the specific vulnerability type, related vulnerabilities, and potential mitigations.

Prior works have proposed DL-based software vulnerability classification (SVC) methods to classify CWE-IDs for the detected vulnerabilities based on the CVE vulnerability description (i.e., Common Vulnerabilities and Exposures) of programs [20, 21]. These methods are only applicable to source code labeled with CVE information. However, the CVE information is not available until released by the MITRE corporation [43]. More importantly, prior works also encountered a serious data imbalance problem, that some of the buffer overflow-related vulnerability types are common while other types rarely occur. In particular, Das et al. [20] leveraged data augmentation [44] in their experiments but it did not further improve the performance of their transformer model; Wang et al. [19] only focused on the top 10 frequency CWE-IDs in their experiment to mitigate the data imbalance, which hinders the model to identify rare vulnerability types.

To address these challenges, we focus on developing a DL-based approach that can identify vulnerability types based on the vulnerable source code without using any CVE description. Thus, the method also applies to source code without CVE description to help identify vulnerability types early before the corresponding CVE entry is released. In particular, Chapter 5 will mitigate the data imbalance issue of SVC based on the hierarchical nature of CWE-IDs, hence learning a more accurate model to provide explanations of detected vulnerabilities to security analysts.

Another mainstream of SVC is to leverage multi-task learning that could improve the model by learning with other related tasks (e.g., CVSS [45] severity score estimation). However, the existing methods rely on loss summation to update the model for different tasks, which does not guarantee an optimal solution for all of the learning tasks [46–48]. To address this limitation and provide security analysts with more accurate vulnerability classification and severity estimation, Chapter 8 will explore a multi-objective optimization (MOO) approach that finds

---

the optimal solution for updating the model in each learning task.

### 2.3 DL-based Automated Vulnerability Repair

Neural machine translation (NMT)-based methods have been proposed [22, 23, 35, 36, 49] to repair software vulnerability automatically using encoder-decoder deep learning (DL) models. These approaches treat automated vulnerability repair (AVR) as a sequence-to-sequence problem where the encoders encode the input programs and the decoders generate corresponding repairs. However, these approaches have two limitations. First, RNN-based models have a limited ability to generate long repair sequences. Second, the length of repair sequences can sometimes be long, hence difficult for NMT models to generate correct repairs.

Thus, Chen, Kommrusch, and Monperrus [23] (accepted at TSE 2022) propose VRepair that leverages (1) a vanilla transformer architecture to improve from RNN-based AVR and (2) context matching to shorten the repair length. Similarly, et al. Chi et al. [35] proposed SeqTrans which relies on the same architecture as VRepair. Both VRepair and SeqTrans leverage a word-level tokenizer, however, Chen, Kommrusch, and Monperrus [23] leverage the copy mechanism while Chi et al. [35] leverage code normalization to mitigate the out-of-vocabulary (OOV) problem. However, the word-level tokenizer limits the model’s ability to generate newly introduced tokens that did not appear in the training data. Moreover, neither VRepair nor SeqTrans were pre-trained on large code bases, limiting the model’s ability to learn better code representation for accurate vulnerability repairs.

To address the limitations of current state-of-the-art AVR like VRepair, Chapter 6 will explore using a pre-trained transformer language model (LM) for source code with subword tokenization (i.e., byte pair encoding [50]) to better handle the OOV problem and obtain improved code representation for more accurate vulnerability repair. Additionally, we identified another limitation of existing transformer-based approaches [23, 35]: vulnerable programs often contain only a few code areas that require repair, yet current AVR methods lack a mechanism to guide the model’s attention to these critical areas during repair generation, potentially hindering the model’s performance. To address this, Chapter 7 will develop an enhanced approach that directs the model to focus more on vulnerable code regions, improving the accuracy and effectiveness of the repair process.

---

## 2.4 Static Application Security Testing (SAST) Tools in IDEs

As introduced in the previous sections, this thesis focuses on advancing deep learning (DL)-based methods to detect, classify, and repair vulnerable source code, thus automating key aspects of the software security analysis process during software development. In particular, Section 2.1 explores DL-based methods for early vulnerability detection, Section 2.2 delves into approaches for identifying the types of detected vulnerabilities to provide detailed explanations, and Section 2.3 examines techniques for suggesting repairs of identified vulnerabilities.

In the context of the DevSecOps (Development, Security, and Operations), Program Analysis (PA)-based static application security testing (SAST) tools [26–29] have already been integrated into software development workflows to facilitate security testing during the development phase. These PA-based tools support the DevSecOps approach by enabling continuous security monitoring and testing during the early stage of the software development lifecycle. However, despite their effectiveness, current DL-based approaches have yet to be integrated into any integrated development environments (IDEs) to assist in detecting security issues during software development.

To bridge this gap, Chapter 8 will propose a real-time software security tool that integrates the DL-based methods developed in this thesis for detecting, classifying, and repairing vulnerable source code during software development. This tool will be developed as an extension for Visual Studio Code [39], providing developers with the ability to identify and address potential security issues early in the development lifecycle. By shifting security considerations from the testing phase to the development phase, this tool will help realize the DevSecOps paradigm, promoting a 'shift-left' approach where security is embedded directly into the software development process.

## 2.5 ChatGPT for Software Vulnerability Predictions

Large language models, such as ChatGPT, represented by versions like gpt-3.5-turbo and gpt-4, have showcased remarkable capabilities in various source code-related tasks. These tasks encompass simulating system behavior based on provided requirements, formulating API specifications, and identifying implicit assumptions embedded within code, as noted in a study by White et al. [51]. Notably, these accomplishments are achieved without any need for parameter updates.

---

Exploiting ChatGPT’s substantial capacity, which stands at 175 billion parameters for gpt-3.5-turbo [52] and a staggering 1.7 trillion parameters for gpt-4 [53], offers a multitude of opportunities for its application in tasks related to software vulnerabilities. Notably, prior works have mainly focused on either using ChatGPT for predicting vulnerabilities [54, 55] or employing it for program repair [56–58]. However, there exists a gap in the literature, as no comprehensive study has evaluated ChatGPT’s performance across the entire spectrum of the vulnerability workflow, which encompasses four vital aspects: (1) predicting software vulnerabilities, (2) classifying software vulnerabilities, (3) estimating their severity, and (4) automating vulnerability repairs.

To address this critical gap, we will undertake an empirical evaluation of ChatGPT’s performance in these four key aspects of vulnerability prediction, as detailed in Chapter 9.

## 2.6 AI for DevSecOps: A Systematic Literature Review for Future Research Directions Beyond This Thesis

In the previous sections, we have explored prior works, defining the scope and novelties of this thesis. The core focus of this research has been on developing more effective deep learning (DL)-based approaches for software vulnerability detection, classification, and repair. Additionally, we have integrated these approaches into a security analysis tool within integrated development environments (IDEs), aimed at supporting security analysts and software developers in enhancing the secure software development phase within DevSecOps.

While this thesis primarily addresses the software development phase of DevSecOps, there are other phases—such as planning, testing, deployment, and monitoring—that also require AI-driven security solutions. To broaden the scope of research beyond this thesis, future work will explore AI-driven security solutions for these other phases.

To this end, Chapter 10 will present a systematic literature review (SLR) that examines the existing landscape of AI-driven security research relevant to DevSecOps, focusing on publications from high-impact venues. This review will identify the current challenges in the field and derive future research directions to pave the way for further advancements beyond the scope of this thesis.

---

### **3 Locating Vulnerabilities – A Novel Transformer-Based Approach for Pinpointing Line-Level Software Vulnerabilities**

© 2022 Association for Computing Machinery. This is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive version was published in Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022), pp. 608–620, published 17 October 2022, <https://doi.org/10.1145/3524842.3528452>.

---

### 3.1 Introduction

Software vulnerabilities are weaknesses in an information system, security procedures, internal controls, or implementations that could be exploited or triggered by a threat source [59]. Those unresolved weaknesses associated with software systems may result in extreme security or privacy risks. For instance, in 2021, a criminal group leveraged the ProxyLogon flaw [60] to access highly confidential data of PC-maker Acer and issued an opening ransom demand of \$50 million USD [61]. The vulnerability was on Microsoft Exchange Server, which allowed an attacker to bypass the authentication and impersonate. Therefore, an unauthenticated attacker could execute arbitrary commands on the server. Cybersecurity Ventures expects global cybercrime costs to reach \$10.5 trillion USD by 2025, up from \$3 trillion USD in 2015 [24]. As cyberattacks become the main contributing factors to a revenue loss of some businesses [25], ensuring the safety of software systems becomes one of the critical challenges of private and public sectors.

To mitigate this challenge, program analysis (PA) tools [26–29] have been introduced to analyze source code using predefined vulnerability patterns. For instance, Gupta et al. [62] found that there are static analysis approaches that were used to detect SQL injection and cross-site scripting vulnerabilities. Both PA-based and ML/DL-based approaches fall short of the capability to detect fine-grained vulnerabilities.” – PA tools, highlight the specific lines in code that contain the detected vulnerabilities.

On the other hand, Machine Learning (ML) / Deep Learning (DL) approaches have been proposed. Specifically, these ML/DL-based approaches [14–18] first generate a representation of source code in order to learn vulnerability patterns. Finally, such approaches will learn the relationship between the representation of source code and the ground truth (i.e., whether a given piece of code is vulnerable). Despite the advantages of dynamically learning the vulnerability patterns without manual predefined vulnerability patterns, previous ML/DL-based approaches still focus on coarse-grained vulnerability prediction where models only point out vulnerabilities at the file level or the function level—which is still coarse-grained.

Recently, Li et al. [32] proposed an IVDetect approach to address the need for fine-grained vulnerability prediction. IVDetect leverages an FA-GCN (i.e., Feature-attention Graph Convolution Network) approach to predict function-level vulnerabilities and a GNNExplainer to locate the fine-grained location of vulner-

---

abilities. Li et al. [32] found that the IV Detect approach achieves an F-measure of 0.35, which outperforms the state-of-the-art approaches. However, IV Detect has the following three limitations:

- **First, the training process of IV Detect is limited to a project-specific dataset.** Due to the limited amount of training data used by IV Detect, the language models may not be able to capture the most accurate relationship between tokens and their surrounding tokens. Thus, the vector representation of source code by IV Detect is still suboptimal.
- **Second, the RNN-based architecture of the IV Detect approach is still not effective in capturing the meaningful long-term dependencies and semantics of source code.** IV Detect relies on RNN-based models to generate vector representations to be used by its graph model during the prediction step. However, RNN-based models often have difficulties in learning the long sequence of source code. Such limitations could make the generated vector representations less meaningful, resulting in inaccurate predictions.
- **Third, the sub-graph interpretation of IV Detect is still coarse-grained.** IV Detect leveraged a GNNExplainer to identify which sub-graph contributed the most to the predictions. Although such sub-graph interpretation can help developers narrow down to locate vulnerable lines, such sub-graphs still contain many lines of code. Thus, security analysts still need to manually locate which lines of these sub-graphs are actually vulnerable.

In this work, we propose LineVul, a Transformer-based fine-grained vulnerability prediction approach to address the three important limitations of IV Detect. First, instead of using RNN-based models to generate a representation of code, we leverage a BERT architecture [63] with self-attention layers that are capable of capturing long-term dependencies within a long sequence using dot-product operations. Second, instead of using project-specific training data, we leverage a CodeBERT pre-trained language model to generate a vector representation of source code. Third, instead of using a GNNExplainer to identify sub-graphs that contribute to the predictions, we leverage the attention mechanism of the BERT architecture to locate vulnerable lines, which is finer-grained than the IV Detect approach. Finally, we conduct an experiment to compare our LineVul approach with seven baseline approaches (i.e., IV Detect [32], Reveal [14], SySeVR [15], Devign [18], Russell et al. [17], VulDeePecker [16], and BoW+RF), and evaluated on both coarse-grained (i.e., function-level) and fine-grained (i.e., line-level) vul-

---

nerability prediction scenarios. Through an extensive evaluation of our approach on 188k+ C functions including 91 different types of CWEs (Common Weakness Enumeration), we answer the following three research questions:

**(RQ1) How accurate is our LineVul for function-level vulnerability predictions?**

**Results.** Our LineVul achieves an F-measure of 0.91, which is 160%-379% better than the state-of-the-art approaches with a median improvement of 250%. Similarly, our LineVul achieves a Precision of 0.97 and a Recall of 0.86, which outperform the baseline approaches by 322% and 19%, respectively.

**(RQ2) How accurate is our LineVul for line-level vulnerability localization?**

**Results.** Our LineVul achieves a Top-10 Accuracy of 0.65, which is 12%-25% more accurate than the other baseline approaches. In addition, LineVul achieves the lowest median IFA of 1, while the baseline approaches achieve a median IFA of 3-4.

**(RQ3) What is the cost-effectiveness of our LineVul for line-level vulnerability localization?**

**Results.** Our LineVul achieves the lowest Effort@20%Recall of 0.75, which is 29%-53% less than other baseline approaches. In addition, LineVul achieves the highest Recall@1%LOC of 0.24, which is 26%-85% higher than other baseline approaches.

### 3.2 Background

A software vulnerability is a defect or a weakness in software implementation due to the way the software is designed or the way the software is coded. In computer security, such software vulnerabilities could lead to an inner system crash or allow an attacker to gain control of a system by crossing privileged boundaries within a computer system. Thus, two types of automated vulnerability prediction approaches, Program Analysis (PA)-based and ML/DL-based, have been proposed to early predict software vulnerabilities.

PA-based techniques use pre-defined patterns to detect software vulnerabilities. Hence, PA-based methods only focus on specific types of vulnerabilities. For instance, the FlawFinder [26] supports common weakness enumeration (CWE),

---

which takes the C/C++ program as input and generates a list of vulnerabilities sorted by risk level. RATS [27] is another static program analysis tool, which can detect the vulnerabilities such as buffer overflows and TOCTOU (Time of Check, Time of Use) race conditions. Cppcheck [28] is a static analysis tool for C/C++ code using unique code analysis and focuses on detecting undefined behavior and dangerous coding constructs. Checkmarx [29] provides automatic scans of uncompiled source code that can be integrated into DevOps, enabling developers to identify security vulnerabilities during development. However, such pre-defined patterns for PA-based techniques need to be manually crafted by security experts, which is time-consuming.

ML/DL-based approaches leverage Machine Learning (ML) and Deep Learning (DL) techniques to automatically learn vulnerability patterns to detect software vulnerabilities. Specifically, code programs are transformed into vector representations for the model to learn the implicit patterns of vulnerabilities from prior vulnerable programs. Recently, several DL models have been applied to vulnerability prediction tasks. For instance, VulDeePecker [16] leverages symbolic representations on program slices, Devign [18] uses graph embedding on code property graphs (i.e., AST, CFG, DFG), SySeVR [15] relies on semantic information induced by data dependency, Reveal [14] adopted graph embedding with triplet loss function for representation learning, Russell et al. [17] leverages CNNs and RNNs to extract representation. Despite these DL models being able to generate better representation, they still focus on coarse-grained vulnerability prediction that provides vulnerability prediction at the file level or the function level.

### 3.2.1 IVDetect: A state-of-the-art fine-grained vulnerability prediction & Limitations

Both PA-based and ML/DL-based approaches fall short of the capability to detect fine-grained vulnerabilities. Therefore, developers would still need to inspect many lines of code to look for and fix the vulnerabilities in their code. Recently, Li et al. [32] proposed IVDetect—a fine-grained graph-based vulnerability prediction approach, which consists of three steps:

**Step 1: Code Representation Learning.** IVDetect leverages the GloVe word embedding (Global Vectors for Word Representation) to capture semantic similarity among tokens and a GRU model to summarize the sequence of vectors into one feature vector. IVDetect generates four feature vectors from the given code statement: 1) a sequence of sub-tokens to capture lexical information, 2) variable names and types to be used as node information in graph model, 3) data depen-

// Line-level Vulnerability Predictions by LineVul			
	LineVul	IVDetect	Ground truth
third_party/WebKit/Source/core/frame/ImageBitmap.cpp https://github.com/chromium/chromium/commit/d59a4441697f6253e7dc3f7ae5caad6e5fd2c778			
224 static sk_sp<SkImage> unPremulSkImageToPremul (SkImage* input) {	0.8	0	0
225 SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),	0.6	0	0
226 kN32_SkColorType, kPremul_SkAlphaType);	0.5	0	0
227 RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);	0.8	0	0
228 if (!dstPixels)	0.3	1	0
229 return nullptr;	0.3	0	0
230 return newSkImageFromRaster(	0.5	1	0
231 info, std::move(dstPixels),	0.6	1	0
232 static_cast<size_t>(input->width()) * info.bytesPerPixel());	1	1	1
233 }	0	0	0

Figure 2: A motivating example of our LineVul vs IVDetect.

dency context, and 4) control dependency context. In addition, a Tree-LSTM is used to generate the representation for AST trees. After obtaining all of the five feature vectors (i.e.,  $F_1, \dots, F_5$ ), IVDetect uses a Bi-GRU and an attention layer to learn the weight vector  $W_i$  for each feature vector  $F_i$ . Finally, each feature vector is multiplied by the computed weight vector:  $F'_i = W_i F_i$ .

**Step 2: Vulnerability Prediction with FA-GCN.** Given an input method  $m$ , IVDetect first processes  $m$  into a program dependency graph (PDG) consisting of many statements, for each statement, the five feature vectors will be generated through the process discussed in **Step 1**. FA-GCN performs sliding a small window along all the nodes (statements) of the PDG and leverages a joint layer to link all generated feature vectors into a feature matrix  $\mathcal{F}_m$  where each row corresponds to a small window in PDG. Then, the symmetric normalized Laplacian matrix [64]  $\mathcal{L}_m$  is calculated and combined with feature matrix  $\mathcal{F}_m$  through the convolution to generate the representation matrix  $\mathcal{M}_m$  for the method  $m$ . Finally, FA-GCN uses a spatial pyramid pooling layer for normalization purposes followed by a fully connected layer to transform matrix  $\mathcal{M}_m$  into vector  $V_m$  and performs classification using two hidden layers and a softmax function to produce a prediction score for  $m$ . The scores are used as vulnerability scores to rank the functions.

---

**Step 3: Fine-grained Vulnerability Prediction by GNNExplainer.** IVDetect leverages GNNExplainer [65] with a masking technique to explain which sub-graphs contribute the most to the vulnerability predictions from the FA-GCN model. The goal of GNNExplainer is to find out a sub-graph  $\mathcal{G}_m$  from the whole PDG  $G_m$  of the method  $m$  that minimizes the difference in the prediction scores between using the entire graph  $G_m$  and using the minimal graph  $\mathcal{G}_m$ . To do so, GNNExplainer learns the set of edge-mask  $EM$  that derives the minimal graph  $\mathcal{G}_m$ . As an  $EM$  is applied, GNNExplainer checks if the FA-GCN model produces the same result (i.e., predicted as vulnerable). If yes, the edge in the edge mask is not important and not included in  $\mathcal{G}_m$ . Otherwise, the edge is important and included in  $\mathcal{G}_m$ . IVDetect then utilizes the best sub-graph  $\mathcal{G}_m$  learned from GNNExplainer as an interpretation for the given function to detect fine-grained vulnerabilities. However, there exist the following limitations:

**Limitation ①: The training process of IVDetect is limited to the project-specific dataset.** The quality of vector representation heavily relies on the language models of code being used. For the IVDetect approach, Li et al. [32] leveraged a GloVe (see Step 1 of IVDetect), which is an unsupervised learning algorithm for obtaining vector representations of words. However, their Glove language models are trained on the project-specific dataset without pre-training on the large code base, which may not be able to generate the most meaningful code representation. Thus, the suboptimal vector representation of source code by IVDetect may lead to inaccurate predictions.

**Limitation ②: The RNN-based architecture of the IVDetect approach is still not effective in capturing the meaningful long-term dependencies and semantics of source code.** IVDetect relies on RNN-based architectures to generate code representation in Step 1, which will encounter problems when processing a long sequence. RNN-based models process a sequence token by token, where the models consider a context vector and a hidden vector of the last token when processing each token. The hidden vector is used to capture short-term dependencies between tokens while the context vector is used for long-term dependencies. However, the context vector has problems capturing adequate long-term dependencies given a long sequence (e.g., a sequence of 500 tokens) due to its limited memory. Thus, this limitation could make the generated feature representations less meaningful, which further negatively impacts the accuracy of the vulnerability prediction models.

**Limitation ③: The sub-graph interpretation of IVDetect is still coarse-grained.**

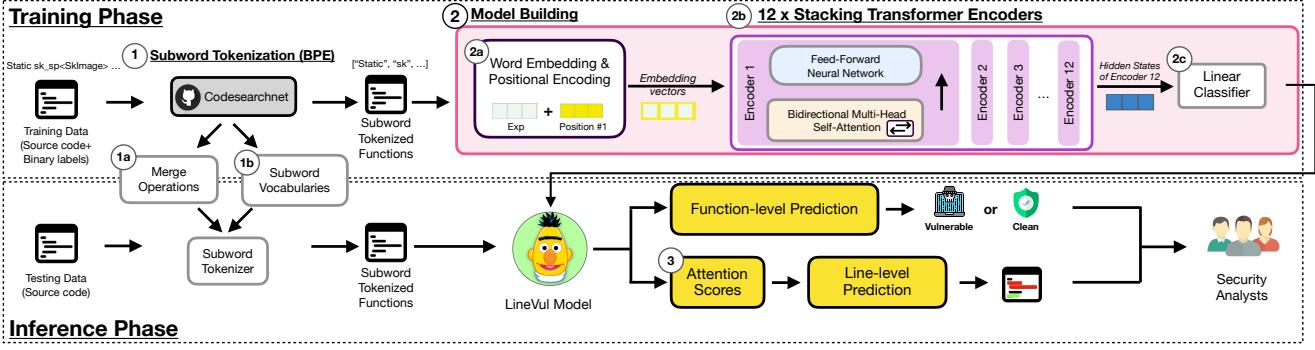


Figure 3: An overview architecture of our LineVul.

Third, in Step 3, IV Detect leverages GNNExplainer to generate PDG sub-graph interpretations as fine-grained vulnerability predictions. Such sub-graph interpretations could consist of multiple lines of code, and they are not fine enough to effectively reduce the manual code inspection effort. For instance, in Figure 2, the vulnerable function *unPremulSkImageToPremul* contains a vulnerable line (i.e., the ninth line) in which the variable type was wrongly defined and further caused a vulnerable type of CWE-787 [66]. IV Detect generated a PDG sub-graph using IV Detect, which pointed out the fifth, seventh, eighth, and ninth lines as vulnerable patterns. On the other hand, our LineVul generated a line-level interpretation, which directly pinpoints the actual vulnerable line.

### 3.3 LineVul: A Line-Level Vulnerability Prediction Approach

In this section, we present the design rationale and the architecture of our LineVul approach.

**Design Rationale.** To address the three key limitations of IV Detect, we propose the architecture of our LineVul, a Transformer-based line-level vulnerability prediction approach. First, instead of using RNN-based models to generate the representation of code, we leverage BERT architecture [63] with self-attention layers that are capable of capturing long-term dependencies within a long sequence using dot-product operations. Second, instead of using project-specific training data, we leverage a CodeBERT pre-trained language model to generate a vector representation of source code. The pre-trained CodeBERT language model was pre-trained on 20GB of code corpus (i.e., CodeSearchNet) using a Robustly Optimized BERT pre-training approach [67]. Therefore, our approach can capture more lexical and logical semantics for the given code input and generate a more meaningful vector representation. Third, instead of using a GNNExplainer to identify sub-graphs that contribute to the predictions, we leverage the

---

attention mechanism of the BERT architecture to locate vulnerable lines, which is finer-grained than the IVDetect approach.

We design our LineVul approach as a two-step approach: to predict vulnerable functions and to locate vulnerable lines. Figure 3 presents an overview architecture of our LineVul approach.

### 3.3.1 Function-level Vulnerability Prediction

The function-level vulnerability prediction consists of 2 main steps:

**① BPE Subword Tokenization.** In Step ①, we leverage the Byte Pair Encoding (BPE) approach [50] to build our tokenizer with two main steps. ①a generating merge operations to determine how a word should be split, and ①b applying merge operations based on the subword vocabularies. Specifically, BPE will split all words into sequences of characters and identify the most frequent symbol pair (e.g., the pair of two consecutive characters) that should be merged into a new symbol. BPE is an algorithm that will split rare words into meaningful subwords and preserve the common words (i.e., will not split the common words into smaller subwords) at the same time. For instance in Figure 2, the function name, *unPremulSkImageToPremul*, will be split into a list of subwords, i.e., ["un", "Prem", "ul", "Sk", "Image", "To", "Prem", "ul"]. The common word "Image" was preserved and other rare words were split. The use of BPE subword tokenization will help reduce the vocabulary size when tokenizing various function names because it will split rare function names into multiple subcomponents instead of adding the full function name into the dictionary directly. In this work, we apply the BPE approach on the CodeSearchNet [68] corpus to produce a subword tokenizer that is suitable for a pre-trained language model of source code corpus.

**② LineVul Model Building.** In Step ②, we build a LineVul model based on the BERT architecture and leverage the initial weights pre-trained by Feng et al. [40]. In Step ②a, LineVul performs a word & positional encoding for the subword-tokenized function to generate an embedding vector of each word and its position in the function. Then, in Step ②b, the vector is fed into the BERT architecture, which is a stack of 12 Transformer encoder blocks. Each encoder consists of a multi-head self-attention layer and a fully connected feed-forward neural network. Finally, in Step ②c, the output vector is fed into a single linear layer in order to perform binary classification for the given function. We describe each step below.

---

**(2a) Word & Positional Encoding.** Source code consists of multiple tokens where the meaning of each token heavily relies on the context (i.e., surrounding tokens) and the position of each token in a function. Therefore, it is important to capture the code context and its position within the function, especially, for function-level vulnerability predictions. The purpose of this step is to generate encoding vectors that capture the semantic meaning of code tokens and their positions in the input sequence. To do so, for each subword-tokenized token, we generate two vectors: (1) a word encoding vector to represent the meaningful relationship between a given code token and the other code tokens and (2) the positional encoding vector to represent the position of a given token in the input sequence. The token encoding vectors are generated according to the word embedding matrix  $W_{te}^{|V| \times d}$  where  $|V|$  is the vocabulary size and  $d$  is an embedding size. The positional encoding vectors are generated according to the positional embedding matrix  $W_{pe}^{c \times d}$  where  $c$  is the context size and  $d$  is the embedding size. Finally, both the word encoding vector and the positional encoding vector are concatenated to produce input vectors of the Transformer encoder blocks.

**(2b) A Stack of 12 Transformer Encoders with Bidirectional Self-attention.** In this Step, the encoding vectors are fed into a stack of 12 encoder-only Transformer blocks (i.e., the BERT architecture [63]). Each encoder block consists of two components, i.e., a bidirectional multi-head self-attention [63] layer and a fully-connected feed-forward neural network. Below, we briefly describe the multi-head self-attention and the feed-forward neural network.

The multi-head self-attention layer is used to compute an attention weight of each code token, that produces an attention vector. The use of bidirectional self-attention allows every token to attend context to its left and right. The generated attention weights are used to indicate which code statements the Transformer model should pay attention to. Generally, the self-attention mechanism is used to obtain global dependencies where the attention weights represent how each code token in the sequence is influenced by all the other words in the sequence, allowing our LineVul approach to capture dependencies between every code token to generate more meaningful representation.

The self-attention mechanism [69] employs a concept of information retrieval, that computes the relevant scores of each code token using the dot product operation where each token interacts with other tokens once. The self-attention mechanism relies on three main components, Query ( $Q$ ), Key ( $K$ ), and Value ( $V$ ). The Query

---

is a representation of the current code token used to score against all the other tokens based on their keys stored in the Key vectors. The attention scores of each token are obtained by taking the dot product between Query vectors and Key vectors. The attention scores are then normalized to probabilities using the Softmax function to get the attention weights. Finally, the Value vectors can be updated by taking the dot product between the Value vectors and the attention weight vectors. The self-attention used in our LineVul is a scaled dot-product self-attention, in which the attention scores are divided by  $\sqrt{d_k}$ . The self-attention mechanism we adopted can be summarized by the following equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

To capture richer semantic meanings of the input sequence, we used a multi-head mechanism to realize self-attention, which allows the model to jointly attend to information from different code representation subspaces at different positions. For  $d$ -dimension  $Q$ ,  $K$ , and  $V$ , we split those vectors into  $h$  heads where each head has  $\frac{d}{h}$ -dimension. After all of the self-attention operation, each head will then be concatenated back again to feed into a fully connected feed-forward neural network including two linear transformations with a ReLU activation in between. The multi-head mechanism can be summarized by the following equation:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O,$$

where  $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$  and  $W^O$  is used to linearly project to the expected dimension after concatenation.

**2c Single Linear Layer.** Multiple linear transformations with non-linear activation functions (i.e., ReLU) are built in the stacking Transformer encoder blocks, therefore, the representation output by the final encoder is meaningful. We only need a single linear layer to map the code representation into the binary label, i.e.,  $\hat{y} = W^T X + b$ .

### 3.3.2 Line-level Vulnerability Localization

Given a function predicted as vulnerable by LineVul, we perform a line-level vulnerability localization by leveraging the self-attention mechanism inside the Transformer architecture to locate the vulnerable lines. The intuition is that to-

---

kens that are most contributed to the predictions are likely to be vulnerable tokens.

For each subword token in the function, in Step ③, we summarize the self-attention scores from each of the 12 Transformer encoder blocks. After obtaining the attention subword-token scores, we then integrate those scores into line scores. We split a whole function into many lists of tokens (each list of tokens represents a line) by the Newline control character. Finally, for each list of token scores, we summarize it into one attention line score and rank line scores in descending order.

## 3.4 Experimental Design

In this section, we present the motivation for our three research questions, our studied dataset, and our experimental setup.

### 3.4.1 Research Questions

To evaluate our LineVul approach, we formulate the following three research questions.

**(RQ1) How accurate is our LineVul for function-level vulnerability predictions?** Recently, Li et al.[32] proposed IVDetect, a state-of-the-art fine-grained vulnerability predictions approach. However, as mentioned in Section 3.2.1, IVDetect has three key limitations, leading to inaccurate and coarse-grained predictions. Therefore, we propose our LineVul approach to address these challenges. Thus, we investigate if the accuracy of our LineVul outperforms the state-of-the-art function-level vulnerability prediction approaches.

**(RQ2) How accurate is our LineVul for line-level vulnerability localization?** Line-level vulnerability prediction is needed to help developers identify the fine-grained locations of vulnerable lines, instead of wasting their time inspecting non-vulnerable lines. Although IVDetect can identify the sub-graphs of a vulnerable function, such sub-graphs still consist of many lines of code that security analysts need to inspect, which is still coarse-grained. Thus, we investigate the accuracy of our LineVul for line-level vulnerability predictions.

**(RQ3) What is the cost-effectiveness of our LineVul for line-level vulnerability localization?** One of the primary objectives of vulnerability prediction is to help security analysts locate vulnerable lines in a cost-effective manner by un-

---

covering the maximum number of vulnerabilities with the least amount of effort. Thus, the amount of effort required to inspect source code becomes a critical concern when security analysts decide to deploy advanced approaches in practice. Thus, we investigate the cost-effectiveness of our LineVul for line-level vulnerability predictions.

### 3.4.2 Studied Dataset

We use the benchmark dataset provided by Fan et al. [70] for the following reasons. The first is to establish a fair comparison with the IVDetect approach. Seconds is to evaluate our line-level vulnerability approach since Fan et al. [70]’s dataset is the only vulnerability dataset that provides line-level ground truths (i.e., which lines in a function are vulnerable). On the other hand, other existing vulnerability datasets (i.e., Devign [18], Reveal [14]) only provide the ground-truths at the function level, but not the line level—which are not suitable for our scope. Fan et al.’s dataset is one of the largest vulnerability datasets that includes line-level ground-truths. The dataset is collected from 348 open-source Github projects, which includes 91 different CWEs from 2002 to 2019, 188,636 C/C++ functions with a ratio of vulnerability functions of 5.7% (i.e., 10,900 vulnerable functions), and 5,060,449 LOC with a ratio of vulnerable lines of 0.88% (i.e., 44,603 vulnerable lines). Among the 10,900 vulnerable functions, the ratio of vulnerable lines varies from 2.5% (1<sup>st</sup> quantile) - 20% (3<sup>rd</sup> quantile) with a median of 7%.

### 3.4.3 Experimental Setup

**Data Splitting.** Similar to Li et al. [32], we use the same data splitting approach, i.e., the whole dataset is split into 80% of training data, 10% of validation data, and 10% of testing data.

**Function-level Model Implementation.** To implement our LineVul approach for the function-level vulnerability prediction, we mainly use two Python libraries, i.e., Transformers [71] and Pytorch [72]. The Transformers library provides API access to the transformer-based model architectures and the pre-trained weight, while the PyTorch library supports the computation during the training process (e.g., back-propagation and parameter optimization). We download the CodeBERT tokenizer and CodeBERT model pre-trained by Feng et al. [40]. We use our training dataset to fine-tune the pre-trained model to get suitable weights for our vulnerability prediction task. The model was fine-tuned on an NVIDIA RTX

---

3090 graphic card and the training time was around 7 hours and 10 minutes. As shown in Equation 1, the Cross-Entropy Loss was used to update the model and optimize between the function-level predicted values and the ground-truth labels, where  $x$  is the number of classes,  $p$  is the ground truth probability distribution (one-hot), and  $q$  is the predicted probability distribution. To retrieve the best-fine-tuned weight, we used the validation set to monitor the training process by epoch, and the best model was selected based on the optimal F1 score against the validation set (not the testing set).

$$H(p, q) = - \sum_x p(x) \log_q(x) \quad (1)$$

**Line-level Model Implementation.** To implement our LineVul approach for the line-level vulnerability prediction, we use the self-attention matrix returned by the fine-tuned model during inference. We summarize the dot product attention score for each token, hence, every token has one attention score. We integrate tokens into lines and summarize the score of each token to get the line scores. The line scores are then used to prioritize the lines where a higher line score represents that the line is likely to be a vulnerable line.

**Hyper-Parameter Settings for Fine-tuning.** For the model architecture of our LineVul approach, we use the default setting of CodeBERT, i.e., 12 Transformer Encoder blocks, 768 hidden sizes, and 12 attention heads. We follow the same fine-tuning strategy provided by Feng et al. [40]. During training, the learning rate is set to 2e-5 with a linear schedule where the learning rate decays linearly throughout the training process. We use backpropagation with AdamW optimizer [73] which is widely adopted to fine-tune Transformer-based models to update the model and minimize the loss function.

### 3.5 Experimental Results

In this section, we present the experiment results with respect to our three research questions.

#### (RQ1) How accurate is our LineVul for function-level vulnerability predictions?

**Approach.** To answer this RQ, we focus on the function-level vulnerability predictions and compare our LineVul with the other seven baseline models described as follows:

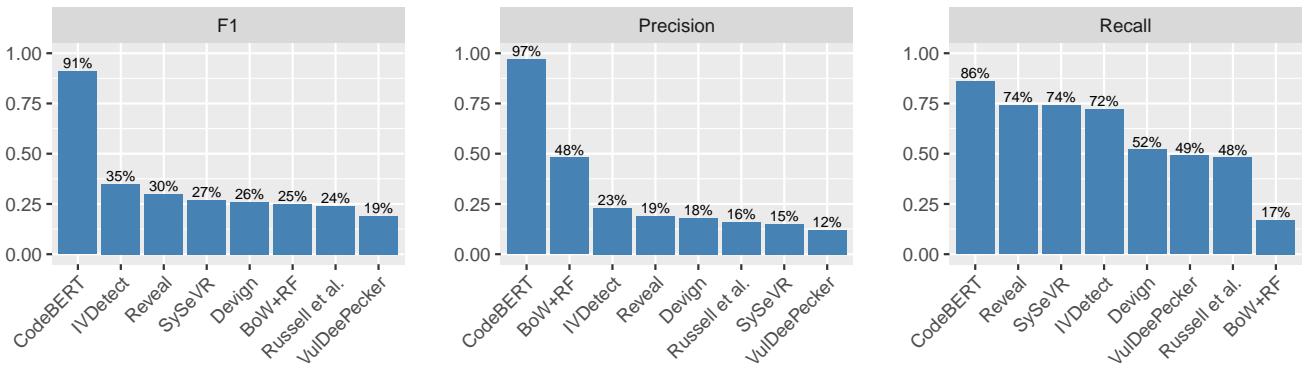


Figure 4: (RQ1) The experimental results of our LineVul and the seven baseline comparisons for function-level vulnerability prediction. (↗) Higher F1, Precision, Recall = Better.

1. IVDetect [32] leverages Feature-Attention Graph Convolutional Network (GCN) for vulnerability predictions using five types of feature representation (i.e., sequence of sub-tokens, AST sub-tree, variable names and types, data dependency context, and control dependency context) from the source code;
2. ReVeal [14] leverages a Gated Graph Neural Network (GGNN) to learn the graph properties of source code;
3. Devign [18] leverages a Gated Graph Neural Network (GGNN) to automatically learn the graph properties of source code (i.e., AST, CFG, DFG, and code sequences);
4. SySeVR [15] uses code statements, program dependencies, and program slicing as features for several RNN-based models (LR, MLP, DBN, CNN, LSTM, etc.) for classification;
5. VulDeePecker [16] leverages a Bidirectional LSTM network for statement-level vulnerability predictions;
6. Russell et al. [17] leverages an RNN-based model for vulnerability predictions.
7. BoW+RF (LineDP/JITLine) uses a bag of words as features together with a Random Forest model for software defect predictions [74, 75].

Similar to Li et al. [32], we evaluate our LineVul with three binary classification measures i.e., Precision, Recall, and F1-score. Precision measures the proportion of the functions that are correctly predicted as vulnerable and the number

---

of functions predicted as vulnerable by the model, which is computed as  $\frac{TP}{TP+FP}$ . Recall measures the proportion of the functions that are correctly predicted as vulnerable and the number of actual vulnerable functions, which is computed as  $\frac{TP}{(TP+FN)}$ . F-measure is a harmonic mean of precision and recall, which is computed as  $\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$ . We use the probability threshold of 0.5 as a cut-off value to indicate the prediction label. The values of these measures range from 0 to 1 where 1 indicates the highest accuracy.

**Results.** Figure 4 presents the experimental results of our LineVul and the seven baseline approaches according to our three evaluation measures (i.e., F1, Precision, Recall).

**Our LineVul achieves an F-measure of 0.91, which is 160%-379% better than the state-of-the-art approaches with a median improvement of 250%.** In terms of F-measure, Figure 4 shows that LineVul achieves the highest F-measure of 0.91, while the state-of-the-art approaches achieve an F-measure of 0.19-0.35. This finding shows that LineVul substantially improves the state-of-the-art by 160%-379% with a median improvement of 250%. In terms of Precision, Figure 4 shows that LineVul achieves the highest Precision of 0.97, while the state-of-the-art approaches achieve a Precision of 0.12-0.48. This finding shows that LineVul substantially improves the state-of-the-art by 102%-708% with a median improvement of 439%. In terms of Recall, Figure 4 shows that LineVul achieves the highest Recall of 0.86, while the state-of-the-art approaches achieve a Recall of 0.17-0.86. This finding shows that LineVul substantially improves the state-of-the-art by 16%-406% with a median improvement of 65%.

In other words, our results demonstrate that **the use of semantic and syntactic features with a Transformer architecture outperforms existing work that uses graph properties of source code**. Our finding is different from the findings of many recent studies which found that the use of graph properties (e.g., Data Dependency Graph, Abstract Syntax Tree, Control Flow Graph, and Data Flow Graph) often outperforms the use of syntactic and semantic features for vulnerability predictions [14, 18, 64]. This is because the RNN-based approaches (e.g., RNN, LSTM, GRU) used as a baseline in prior studies (1) are trained on a project-specific dataset; and (2) suffer from capturing the long-term dependencies of source code, as discussed in Section 2.1 (cf. Limitations ①, ②). Different from prior studies, our results confirm that our LineVul approach is more accurate than the state-of-the-art approaches, highlighting the substantial benefits of

---

the use of the CodeBERT pre-trained language model trained on million GitHub repositories and the use of Transformer architecture to capture the long-term dependencies of source code, leading to significant improvement of the vulnerability prediction approach at the function level.

**(RQ2) How accurate is our LineVul for line-level vulnerability localization?**

**Approach.** To answer this RQ, we focus on evaluating the accuracy of the line-level vulnerability localization. Thus, we start from the vulnerable functions that are correctly predicted by our approach. Then, we perform the Step ③ (see Section 3.2) to identify which lines are likely to be vulnerable for a given function predicted as vulnerable. Thus, each line in the given function will have its own score (i.e., line-level score). Since other approaches in RQ1 are not designed for line-level localization, we do not compare our approach with them. Instead, we compare our LineVul approach with 5 other model-agnostic techniques that are commonly used for deep learning models as follows:

1. Layer Integrated Gradient (LIG) [76] is an axiomatic path-attribution method that attributes an importance score to each input feature by approximating the integral of gradients of the model’s output with respect to the inputs along the path (straight line) from given baselines to inputs.;
2. Saliency [77] takes a first-order Taylor expansion of the network at the input, and the gradients are simply the coefficients of each feature in the linear representation of the model. The absolute value of these coefficients can be taken to represent feature importance.;
3. DeepLift [78, 79] compares the activation of each neuron to its reference activation and assigns contribution scores according to the difference;
4. DeepLiftSHAP [80] extends the DeepLift algorithm and approximates SHAP values using the DeepLift approach;
5. GradientSHAP [80] approximates SHAP values by computing the expectations of gradients by randomly sampling from the distribution of baselines;
6. CppCheck [28] is a static code analysis tool for the C and C++ programming languages.

To evaluate our LineVul approach for line-level vulnerability localization, we use the following two measures described below:

- 
- 1) Top-10 Accuracy measures the percentage of vulnerable functions where at least one actual vulnerable line appears in the top-10 ranking. The intuition is that security analysts may ignore line-level recommendations if they do not appear in the top-10 ranking, similar to any recommendation systems [81]. Thus, top-10 accuracy will help security analysts better understand the accuracy of the line-level vulnerability localization approaches.
  - 2) Initial False Alarm (IFA) measures the number of incorrectly predicted lines (i.e., non-vulnerable lines incorrectly predicted as vulnerable or false alarms) that security analysts need to inspect until finding the first actual vulnerable line for a given function. IFA is calculated as the total number of false alarms that security analysts have to inspect until finding the first actual vulnerable line. A low IFA value indicates that security analysts only spend less amount of effort in inspecting false alarms.

**Results.** Figure 5 presents the experimental results of our LineVul and the five baseline approaches according to our two evaluation measures (i.e., Top-10 Accuracy and IFA).

**Our LineVul achieves a Top-10 Accuracy of 0.65, which is 12%-25% more accurate than the other baseline approaches.** In terms of Top-10 Accuracy, Figure 5 shows that LineVul achieves the highest Top-10 Accuracy of 0.65 while the state-of-the-art approaches achieve a Top-10 Accuracy of 0.52-0.58. In terms of IFA, Figure 5 shows that LineVul achieves the lowest median IFA of 1 while the baseline approaches achieve a median IFA of 3-4. This finding shows that LineVul substantially improves the baseline approaches by 67%-75% with a median improvement of 67%. These results confirm that our LineVul approach is more accurate than the baseline approaches for line-level vulnerability localization.

In other words, our results demonstrate that **the attention mechanism outperforms other model-agnostic techniques**. In the line-level defect prediction literature, prior studies [74, 75] often leverage a LIME model-agnostic technique [82] to explain the predictions of DL/ML-based defect prediction models. However, such model-agnostic techniques are considered an extrinsic model-agnostic technique (i.e., a model-agnostic is applied to explain to a black-box DL/ML model after the model is trained), not an intrinsic model-agnostic technique (i.e., a DL/ML model that is interpretable by itself so extrinsic model-agnostic techniques are not needed to apply afterward). Although it is widely known that deep learning is complex and hardly interpretable, this work is among the

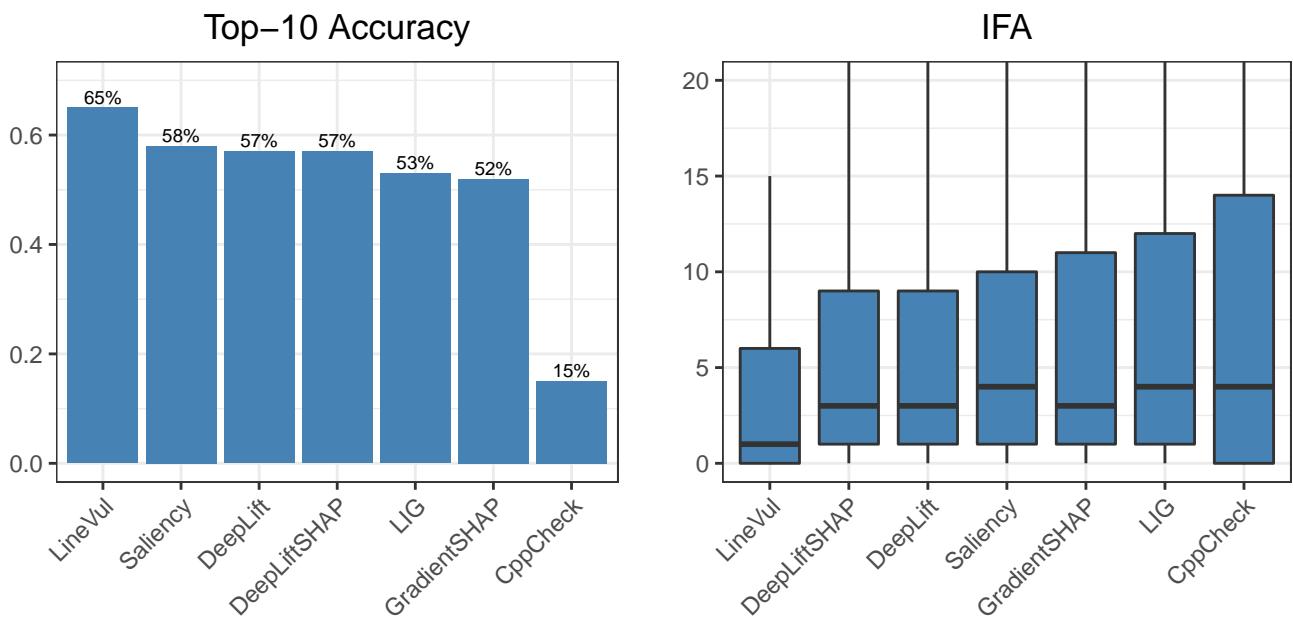


Figure 5: (RQ2) The Top-10 Accuracy and IFA of our self-attention approach and five other methods. ( $\nearrow$ ) Higher Top-10 Accuracy = Better, ( $\searrow$ ) Lower IFA = Better.

first attempts to leverage the attention mechanism to explain the prediction of Transformer-based models, highlighting the substantial benefits of the attention mechanism for line-level vulnerability localization.

#### (RQ3) What is the cost-effectiveness of our LineVul for line-level vulnerability localization?

**Approach.** To answer this RQ, we focus on evaluating the cost-effectiveness of our LineVul approach for line-level vulnerability localization. In the real-world scenario, the most cost-effective line-level vulnerability prediction approaches should help security analysts find the highest number of actual vulnerable lines with the least amount of effort. Thus, let's assume that the 18,864 functions (i.e., 504,886 LOC) in the testing dataset are functions that security analysts have to inspect. To measure the cost-effectiveness of our approach, we first obtain the predictions from our LineVul approach. Then, we sort the predicted functions according to the predicted probability. Among the functions predicted as vulnerable, we sort the lines according to the line score obtained from Step ③ of our approach in descending order. Thus, lines with the highest score (i.e., likely to be vulnerable) will be ranked at the top. Then, we evaluate the cost-effectiveness using the following measures:

- 1) Effort@20%Recall measures the amount of effort (measured as LOC) that

---

security analysts have to spend to find out the actual 20% vulnerable lines. It is computed as the total LOC used to locate 20% of the actual vulnerable lines divided by the total LOC in the testing set. A low value of Effort@20%Recall indicates that the security analysts may spend a smaller amount of effort to find the 20% actual vulnerable lines.

- 2) Recall@1%LOC measures the proportion of actual vulnerable lines that can be found (i.e., correctly predicted) given a fixed amount of effort (i.e., the top 1% of LOC of the given testing dataset). Recall@1%LOC is computed as the total number of correctly located vulnerable lines within the top 1% lines in the testing set divided by a total number of actual vulnerable lines in the testing set. A high value of Recall@1%LOC indicates that an approach can rank many actual vulnerable lines at the top.

**Results.** Figure 6 presents the experimental results of our LineVul and the five baselines according to our two evaluation measures (i.e., Effort@20%Recall and Recall@1%LOC).

**Our LineVul achieves an Effort@20%Recall of 0.75, which is 29%-53% less than other baseline approaches.** In terms of Effort@20%Recall, Figure 6 shows that LineVul achieves the lowest Effort@20%Recall of 0.75% while the baseline approaches achieve an Effort@20%Recall of 1.06%-1.60%. It means that security analysts may spend the effort to inspect 0.75% of the total LOC in the testing dataset (i.e.,  $0.75\% \times 504,886 = 3,786$  LOC) to find 20% of the actual vulnerable lines (i.e., 20%Recall). Thus, this finding indicates that our approach may help security analysts spend less amount of effort to find the same amount of actual vulnerable lines.

In terms of Recall@1%LOC, Figure 6 shows that LineVul achieves the highest Recall@1%LOC of 0.24, while the baselines achieve a Recall@1%LOC of 0.13-0.19, indicating that LineVul substantially improves the baselines by 26%-85% with a median improvement of 85%. This finding implies that our approach may help security analysts to find more actual vulnerable lines than other baseline approaches given the same amount of effort that they have to inspect.

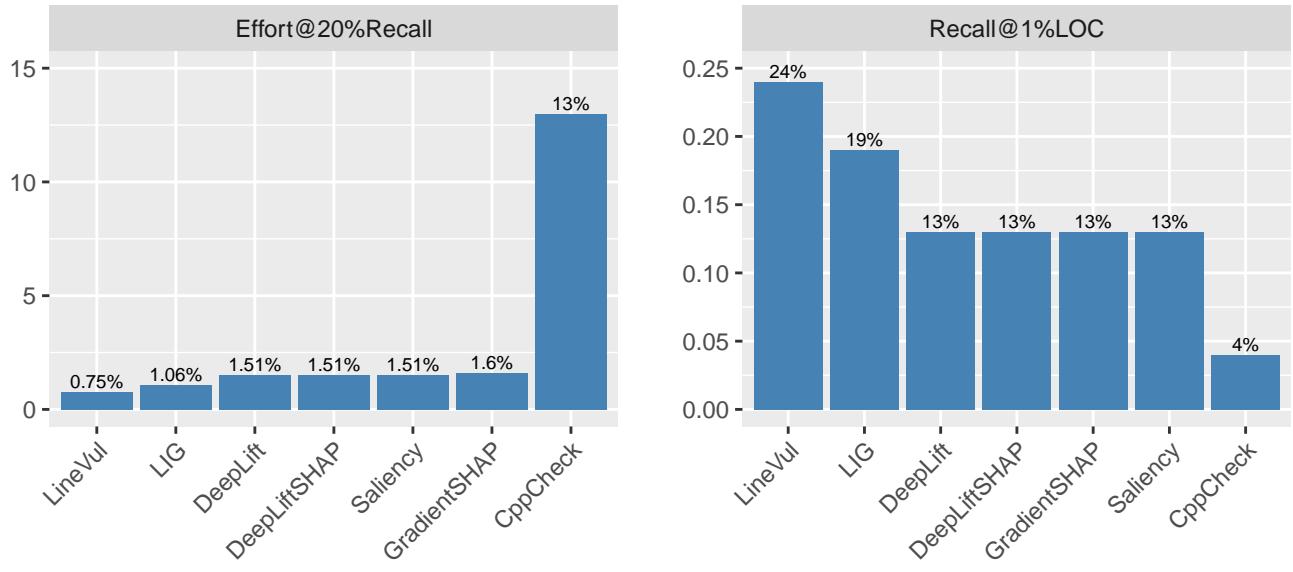


Figure 6: (RQ3) The Effort@20%Recall and Recall@1%LOC of our self-attention approach and five other methods. (↙) Lower Effort@20%Recall = Better, (↗) Higher Recall@1%LOC = Better.

## 3.6 Discussion

### 3.6.1 Why LineVul performs so well for vulnerability predictions?

We conduct an ablation study on function-level vulnerability prediction to quantify the contributions of the components of our LineVul approach. Generally, our LineVul approach consists of 3 components: BPE+Pretraining<sub>Code</sub>+BERT. To understand the contribution of each component, we alter each of the components as follows (highlighted as underlined):

- Word-Level+Pretraining<sub>Code</sub>+BERT: Remove the BPE subword tokenization but use a word-level tokenization instead.
- BPE+No Pretraining+BERT: Remove the pretraining, but use non-pretrained weights to initialize BERT instead.
- Word-Level+No Pretraining+BERT: Remove the BPE and the pre-training on source code components, but use Word-Level tokenization and non-pretrained weights to initialize BERT.

We find that **the BPE component of our LineVul is the most important**. Within our LineVul, the BPE component contributes to 53.8% of the F-measure. When

---

Table 2: The contribution of each component of LineVul for function-level vulnerability predictions.

Model	F1	Precision	Recall
LineVul (BPE+Pretraining <sub>Code</sub> +BERT)	<b>0.91</b>	<b>0.97</b>	<b>0.86</b>
BPE+No Pretraining+BERT	0.80	0.86	0.75
Word-Level+Pretraining <sub>Code</sub> +BERT	0.42	0.55	0.34
Word-Level+No Pretraining+BERT	0.39	0.43	0.36
IVDetect	0.35	0.23	0.72

comparing (BPE+Pre+BERT and Word-level+Pre+BERT) where the BPE component is changed to a word-level tokenizer, we observe a performance decrease from 0.91 to 0.42, accounting for 53.8%. This finding indicates that BPE subword-level tokenization is more beneficial for source code preprocessing than word-level tokenization. We suspect that source code often contains uncommon keywords (e.g., variable names, identifiers) than the natural languages (e.g., English). Thus, language models of code may not be able to generate the most meaningful vector representation for such uncommon keywords in the source code when using word-level tokenization. Instead, when using BPE subword-level tokenization, such uncommon words (e.g., `[‘unPremulSkImageToPremul’]`) are broken into common subwords (e.g., `[‘un’, ‘Prem’, ‘ul’, ‘Sk’, ‘Image’, ‘To’, ‘Prem’, ‘ul’]`). Karampatsis et al. [83] conducts a study of how different modeling choices affect the model performance of language models of source code and finds the advancement of leveraging the BPE model for language modeling of source code. Similar to their results, we find that the use of BPE will not only reduce the size of the unique vocabulary but also help the language models of code to better understand the relationship between a given subword and its surrounding subwords to generate more meaningful vector representation, achieving higher accuracy.

Within our LineVul, the pre-training component contributes to 12.1% of the F-measure. When comparing between (BPE+Pre+BERT and BPE+No Pre+BERT) where the pre-training component is eliminated, we observe a performance decrease from 0.91 to 0.80, accounting for 12.1%. This is because the pre-training language model of code has learned the relationship of tokens from millions of GitHub repositories, therefore, generating more meaningful vector representation than the approach without pre-training. This finding confirms that the use of a pre-training language model of code (i.e., CodeBERT) to generate vector representation outperforms an approach that is only trained on project-specific

---

Table 3: (Discussion) The Accuracy of our LineVul for the Top-25 Most Dangerous CWEs ([https://cwe.mitre.org/top25/archive/2021/2021\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html)).

Rank	CWE Type	Name	TPR	Proportion
1	CWE-787	Out-of-bounds Write	75%	18/24
4	CWE-20	Improper Input Validation	86%	98/114
8	CWE-22	Path Traversal	100%	4/4
12	CWE-190	Integer Overflow	90%	27/30
17	CWE-119	Improper Restriction	88%	173/197
20	CWE-200	Exposure of Sensitive Info	85%	45/53
25	CWE-77	Improper Neutralization	100%	2/2
		TOTAL	87%	367/424

datasets.

Within our LineVul, The BPE component combined with the pre-training component contributed to 57.1% of the F-measure. When comparing “BPE+Pre+BERT” and “Word-level+No Pre+BERT” where both BPE and Pre-training are altered, we observe a performance decrease from 0.91 to 0.39, accounting for 57.1%.

Last but not least, we find that **our LineVul that leverages both BPE and pre-training provides the best F-measure among all variants**, which is 160% better than IV Detect, highlighting the significant advancement of LineVul for line-level vulnerability predictions.

### 3.6.2 How accurate is our LineVul for predicting the Top-25 Most Dangerous CWEs?

**Our LineVul can correctly predict 87% of the vulnerable functions affected by the Top-25 most dangerous CWEs.** CWE (Common Weakness Enumeration) is a list of vulnerability weaknesses in software that can lead to security issues with its severity of risk, providing guidance to organizations and security analysts to best secure their software systems. To better understand the sig-

---

nificance of our LineVul on the practical usage scenarios, we perform a further investigation to better understand our accuracy for the Top-25 most dangerous CWEs. The top 25 most dangerous CWEs are the most common and impactful issues experienced over the previous two calendar years. Such weaknesses are dangerous because they are often easy to find, and exploit, and can allow adversaries to completely take over a system, steal data, or prevent an application from working. Since not all of the top 25 most dangerous CWEs are included in the studied datasets, Table 3 presents the results for the ones that are included in the dataset. We compute the accuracy as the True Positive Rate (TPR) to focus on the vulnerable functions that are correctly predicted by our LineVul. We find that LineVul achieves an accuracy of 75% (CWE-787 Out-of-bounds Write) to 100% (CWE-22 Path Traversal, CWE-77 Improper Neutralization), depending on the CWE types in the dataset.

### 3.7 Related Work

#### 3.7.1 DL-based Vulnerability Prediction

Traditionally, ML-based vulnerability prediction approaches are proposed by using software metrics as features (e.g., code complexity) [41, 42]. The use of software metrics is also widely used in the defect prediction literature [84–92]. However, the collection of such software metrics is manual and time-consuming. Thus, multiple DL-based approaches have been proposed to automatically learn the vulnerability patterns from historical data [14–18, 32, 93, 94].

Therefore, an RNN-based architecture (i.e., LSTM) is used to automatically learn the semantic and syntactic features of source code [17, 95]. For example, Russell et al. [17] proposed an RNN-based architecture to automatically extract features of source code for vulnerability prediction. Dam et al. [95] proposed an LSTM-based architecture to automatically learn the semantic and syntactic features of source code. However, the RNN-based approaches often assume that the source code is a sequence of tokens without considering the graph structure of the source code (e.g., Abstract Syntax Trees), leading to inaccurate predictions.

Therefore, Li et al. [16] proposed VulDeePecker which is an RNN-based model that is learned from different types of graph properties of source code (e.g., Data Dependency Graph). However, the VulDeePecker approach still learns the graph properties in a sequential fashion, without leveraging the graph neural network.

---

Therefore, a Graph Neural Network has been recently used to learn the graph properties of source code for vulnerability predictions. For example, Zhou et al. [18] leveraged a Graph Neural Network to learn four types of graph properties of source code, i.e., Abstract Syntax Tree, Control Flow Graph, Data Flow Graph, and syntactic features. Chakraborty et al. [14] proposed Reveal, which is a Gated Graph Neural Network (GGNN) that learns the graph properties of source code.

While these studies focus on the vulnerability predictions at the file/function level, our LineVul focuses on the line-level vulnerability prediction problem—which still remains largely unexplored.

### 3.7.2 Line-Level Vulnerability Prediction

Although various vulnerability prediction approaches are proposed, they mainly focus on the granularity of file, function, and method levels—which is still coarse-grained. Thus, Li et al. [15, 93] proposed VulDeeLocator, which is based on a program slicing technique to narrow down the scope of vulnerability localization. In addition, Li et al. [32] also proposed IV Detect, which leverages a Graph Neural Network (GNN) for function-level predictions and a GNNExplainer to identify which sub-graph contributes the most to the predictions. Yet, security analysts still need to manually locate which lines in the sub-graph are actually vulnerable.

Similarly, line-level defect prediction has recently received high attention from the research community [74, 75, 94]. For example, Pornprasit and Tantithamthavorn [74] and Wattanakriengkrai et al. [75] proposed a machine learning-based approach with LIME model-agnostic technique (BoW+RF+LIME) to predict which lines are likely to be defective in the future. However, such approaches only learn the frequency of the appearance of code tokens in a file (i.e., Bag-of-Word), without considering the lexical and semantics of the source code (i.e., the sequence of code tokens).

To the best of our knowledge, this paper is among the first to leverage the attention mechanism inside the BERT architecture for line-level vulnerability predictions.

### 3.7.3 Explainable AI for SE

The explainability of AI models in SE becomes one of the research grand challenges [96] (see <http://xai4se.github.io>) since practitioners often do not trust the predictions [97], hindering the adoption of AI-powered software development

---

tools in practices. Recently, Explainable AI has been actively investigated in the domain of defect prediction [97, 98]. For example, recent works have shown some successful case studies to make defect prediction models more practical [74, 75], explainable [99, 100], and actionable [101, 102]. However, these studies only focus on explaining the traditional machine learning models, not the complex black-box deep learning models.

Recently, researchers have started to explore the explainability of AI models in various SE tasks (i.e., leveraging the attention weights to provide meaningful ‘explanations’ for predictions). For example, Fu and Tantithamthavorn [103] proposed a GPT-2 based Agile story point estimation, by leveraging the Integrated Gradient attention to interpret the GPT-2 model and understand what words in a JIRA issue report contributed to the estimation of Agile story points. Similarly, Pornprasit and Tantithamthavorn [94] proposed a Hierarchical Attention Network (HAN) architecture for line-level defect prediction, by leveraging the attention mechanism of the HAN architecture to understand what code tokens in a source code contributed to the prediction of defective files. However, Jain et al. [104] argued that the learned attention weights are frequently uncorrelated with gradient-based measures of feature importance, while Wiegreffe et al. [105] argued that the accuracy/reliability of such attention weights could provide meaningful explanations, depending on the definition and the rigor of experimental design.

To the best of our knowledge, this paper is among the first to leverage the attention mechanism of BERT architecture for line-level vulnerability predictions. This concept is directly aligned with findings from the AI discipline by Wiegreffe et al. [105]. However, this concept is still novel for line-level vulnerability predictions, since prior works [94, 103] only focused on explaining Agile story point estimations and defect predictions—*not CodeBERT-based line-level vulnerability predictions*. Our results in RQ2 and RQ3 confirm that the use of the self-attention mechanism outperforms other model-agnostic techniques for line-level vulnerability predictions.

### 3.8 Summary

In this chapter, we propose LineVul, a Transformer-based vulnerability prediction approach to predict software vulnerabilities on both function and line levels. Through an empirical evaluation of a large-scale real-world dataset with 188k+ C/C++ functions, we show that LineVul achieves (1) 160%-379% higher

---

F1-measure for function-level predictions; (2) 12%-25% higher Top-10 Accuracy for line-level predictions; and (3) 29%-53% less Effort@20%Recall than the state-of-the-art approaches. Our results confirm that LineVul is more accurate and finer-grained than existing vulnerability prediction approaches. Thus, we expect that our LineVul may help security analysts find vulnerable lines cost-effectively.

---

## **4 Learning Vulnerability Patterns – Matching Line-Level Vulnerabilities with Optimal Transport (OT) and Vector Quantization (VQ)**

\* **The work in this chapter is under review** by the journal of IEEE Transactions on Dependable and Secure Computing (TDSC).

---

## 4.1 Introduction

The number of software vulnerabilities has been escalating rapidly in recent years. In particular, the National Vulnerability Database (NVD) [106] reported 26,447 software vulnerabilities in 2023, soaring 40% from 18,938 in 2019. The extensive use of open-source libraries, in particular, may contribute to this rise in vulnerabilities. For instance, the Apache Struts vulnerabilities [107] indicate that this poses a tangible threat to organizations. The root cause of these vulnerabilities is often insecure coding practices, making the source code exploitable by attackers who can use them to infiltrate software systems and cause considerable financial and social harm.

To mitigate security threats, security experts leverage static analysis tools that check the code against a set of known patterns of insecure or vulnerable code, such as buffer overflow vulnerabilities and other common security flaws. Deep learning (DL)-based vulnerability detection (VD) methods have demonstrated higher accuracy compared to static analysis tools [30]. Moreover, recent studies have proposed line-level VDs, that can pinpoint vulnerable lines to minimize the manual analysis burden on security analysts [1, 108, 109].

*In this work, we consider a vulnerability scope of a function as the collection of all vulnerable lines in that function.* As illustrated in Figure 7, each function consists of one vulnerable line with similar vulnerability scopes. This suggests that even if two functions contain the same CWE-787 out-of-bound write vulnerability (the top-1 dangerous CWE-ID in 2023 [110]), the specific vulnerable lines can be written in different ways and located in different parts of the code. Thus, identifying vulnerabilities at the line level is challenging for DL models. On the other hand, we observed that the embeddings of vulnerable lines form clusters in the feature space, where similar embeddings are close to each other as presented in the upper part of Figure 11. Harnessing this cluster information could empower DL models to more effectively capture the latent patterns within the feature space, which could significantly enhance the accuracy of VD approaches for both function and line-level predictions. Nevertheless, our analysis reveals that state-of-the-art VD approaches have not effectively leveraged the information presented in vulnerability scopes.

To address this gap, we propose a novel DL framework that uses vector quantization (VQ) [111] with optimal transport (OT) [112] to aggregate similar vulnerability scopes in the training data, generating a “vulnerability codebook” con-

## CWE-787 Example | Language: C

```
1 static sk_sp<SkImage> unPremulSkImageToPremul(SkImage* input) {  
2     SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),  
3                                         kN32_SkColorType, kPremul_SkAlphaType);  
4     RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);  
5     if (!dstPixels)  
6         return nullptr;  
7     return newSkImageFromRaster(  
8         info, std::move(dstPixels),  
9         static_cast<size_t>(input->width()) * info.bytesPerPixel()); // Vulnerable  
10 }
```

## CWE-787 Example | Language: C

```
1 GPMF_ERR IsValidSize(GPMF_stream *ms, uint32_t size)  
2 {  
3     if (ms)  
4     {  
5         int32_t nestsize = (int32_t)ms->nest_size[ms->nest_level]; // Vulnerable  
6         if (nestsize == 0 && ms->nest_level == 0)  
7             nestsize = ms->buffer_size_longs;  
8         if (size + 2 <= nestsize) return GPMF_OK;  
9     }  
10    return GPMF_ERROR_BAD_STRUCTURE;  
11 }
```

Figure 7: In both functions, the CWE-787 (Out-of-Bound Write) vulnerability is triggered by an inappropriate data type assignment. In the *unPremulSkImageToPremul* [9] function above, the *size\_t* type should be changed to the *unsigned* type. In the *IsValidSize* [10] function below, the *int\_32t* should be changed to *uint\_32t* to prevent potential buffer overflow. Despite sharing the same vulnerability type and pattern, the vulnerable lines in each function and their context are different in their written form, variable names, and positions.

---

sisting of compressed vulnerability patterns that encapsulate similar vulnerability scopes. Additionally, we introduce a vulnerability-matching technique to leverage the learned patterns during inference. It is worth noting that in our approach, the labels of vulnerable lines are only required during the training phase to construct our codebook. Importantly, our method does not depend on vulnerable line labels during the subsequent validation and testing phases.

In particular, we collect 6,361 vulnerability scopes from our training data and embed each scope into a flat vulnerability vector. However, the numerous vulnerability vectors will require intensive computation during our vulnerability-matching inference. Thus, we leverage the clustering characteristic of vulnerability vectors and apply the principle of VQ with OT to transfer the mass distribution of vulnerability vectors into a more compact distribution of vulnerability codebook involving vulnerability patterns representing similar vulnerability vectors. We effectively quantize 6,361 vectors into 150 patterns and ensure that similar vulnerability vectors are mapped to the same pattern in the embedding space. Ultimately, the codebook encapsulates important vulnerable line information from the training data.

Inspired by the pattern-matching concept used in program analysis tools[26, 29], we further propose a vulnerability matching to leverage critical vulnerable line information learned from our VQ and OT processes. During inference, our model matches the input program against all patterns in the learned vulnerability codebook. By examining all the vulnerability patterns in the codebook, the matching process enables a thorough search for potential vulnerabilities. We name this model OptiMatch, a function and line-level VD approach via vector quantization (VQ), optimal transport (OT), and vulnerability matching. To the best of our knowledge, we are the first to exploit vulnerable line information presented in training data using VQ and OT along with a vulnerability matching process, allowing us to further improve the overall capability of DL-based VD.

We extensively evaluated our OptiMatch using two datasets: the Big-Vul dataset [70] and the D2A dataset [113]. These datasets contain line-level vulnerability labels and are widely used by prior VD works [1, 32, 108, 109, 114]. Specifically, the Big-Vul dataset comprises over 188K C/C++ functions with diverse real-world vulnerabilities extracted from multiple large-scale open-source software projects spanning from 2002 to 2019. Through this evaluation, we aim to address the following two research questions:

- 
- What is the accuracy of our OptiMatch for predicting function-level and line-level vulnerabilities?

**Results.** Our OptiMatch approach achieves the highest F1 scores for both function and line-level vulnerability predictions on both experimental datasets. Particularly noteworthy is the line-level F1 score of 82% achieved by OptiMatch on the Big-Vul dataset, surpassing the performance of the best baseline approach by 10%.

- What are the contributions of each component in our OptiMatch approach?

**Results.** The ablation study confirms the effectiveness of our RNN-based line embedding over mean or max pooling methods proposed in Sentence-BERT. Furthermore, our vulnerability matching approach, employing optimal transport (OT) and vector quantization (VQ), significantly improves the F1 score from 37% to 82%. Additionally, the study validates our decision regarding the number of patterns used for the OT process. These results validate the technical proposal within our OptiMatch approach.

**Novelty & Contributions.** To the best of our knowledge, the contributions of this work are as follows:

- OptiMatch, an innovative DL-based vulnerability-matching framework utilizing the optimal transport (OT) theory and vector quantization (VQ) to locate line-level vulnerabilities.
- A novel line embedding approach using recurrent neural networks (RNNs) to represent code lines efficiently.
- A thorough evaluation of our method compared to other DL-based vulnerability prediction methods on two real-world vulnerability datasets.
- A comprehensive ablation study along with an extended discussion to investigate each component in our OptiMatch approach.

## 4.2 Background & Related Work

We observed that vulnerability scopes tend to form clusters in the feature space, indicating recognizable hidden patterns, as depicted in the upper part of Figure 11. Thus, we anticipate that incorporating this clustering characteristic will significantly enhance the performance of vulnerability detection (VD) models.

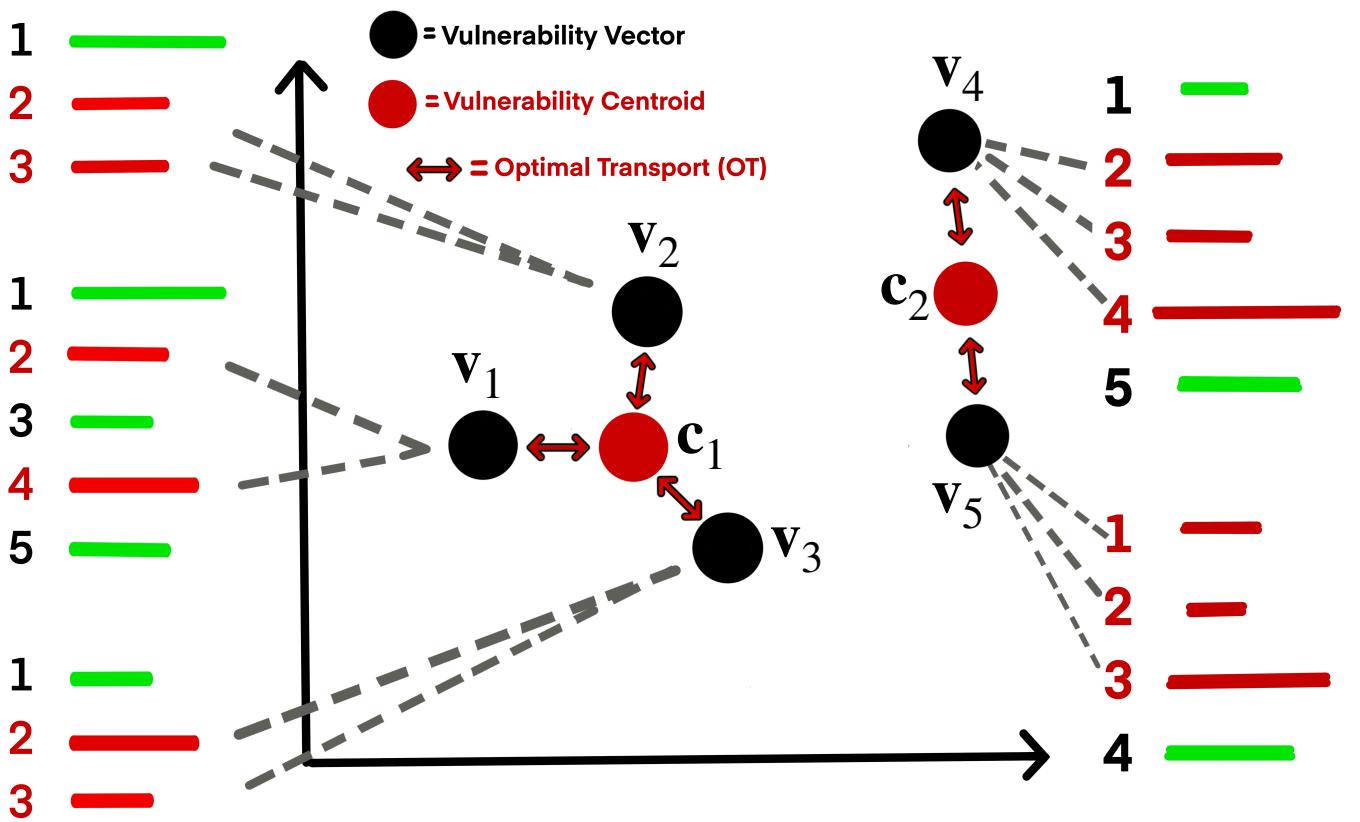


Figure 8: The green lines are benign code lines while the red lines are vulnerable ones. We extract vulnerable lines as a vulnerability vector denoted as  $\mathbf{v}$ . We show that similar  $\mathbf{v}$  sharing the same vulnerability pattern will stay close to each other in the feature space. Thus, they can be further grouped into vulnerability centroids denoted as  $\mathbf{c}$  using optimal transport (OT).

---

To this end, we capture a set of distinct vulnerability scopes in the training set. In total, we curate 6,361 diverse vulnerability scopes, transforming each into a vulnerability vector  $\mathbf{v}$  to compose our vulnerability collection  $\mathcal{V}_v$ . However, such a huge collection will introduce a computational burden during inference. To address this, we follow the principle of vector quantization (VQ) [111] and employ optimal transport (OT) [112] to quantize similar  $\mathbf{v}$  instances into a vulnerability centroid  $\mathbf{c}$  as depicted in Figure 8. This approach aligns with our observation that vulnerability scopes tend to cluster together, which effectively reduces the collection size. The outcome is a vulnerability codebook  $\mathcal{C}$  comprising 150 vulnerability centroids  $\mathbf{c}$  representing common hidden patterns. To leverage the potential of this codebook, we propose a vulnerability-matching methodology. We match a testing function with each centroid in our learned codebook to predict vulnerabilities at the function and line levels. The matching concept is inspired by static analysis tools that match human-defined patterns to identify vulnerabilities, while we match patterns learned in the feature space to detect vulnerabilities using deep learning models.

Below, we introduce background knowledge of optimal transport (OT) and vector quantization (VQ), and the motivations for using OT and VQ, followed by a discussion of related works.

#### 4.2.1 Optimal Transport

Optimal transport theory, also known as transportation theory or the theory of mass transportation, is a mathematical framework that deals with the optimal ways to transport objects from one place to another. Originally developed by Gaspard Monge [115], optimal transport theory has found applications in various fields, including economics, physics, computer science, and machine learning [116].

In the context of machine learning, optimal transport theory is often used to measure the similarity between probability distributions. The Wasserstein distance is a metric derived from optimal transport theory that quantifies the minimum amount of “work” required to transform one distribution into another. It measures how much “mass” needs to be moved from each point in one distribution to its corresponding point in the other distribution, with the total amount of work being minimized.

**Motivation.** The Wasserstein distance [117] has several important properties

---

that make it useful in our application of clustering vulnerability vectors into centroids. Firstly, it takes into account the underlying structure of the distributions, allowing us to capture both global and local differences between vulnerability vectors. Global differences refer to broad variations or distinctions that exist across the entire set of vulnerability vectors while local differences pertain to more specific variations that occur within subsets or individual instances of vulnerability vectors. By considering both global and local differences, we can gain a comprehensive understanding of the structure and characteristics of our collected vulnerability vectors. This enables us to identify common patterns, leading to more effective clustering. By minimizing the Wasserstein distance between vulnerability vectors and centroids during the clustering process, we can ensure that the centroids become representative patterns, facilitating the identification of common vulnerabilities and enhancing the overall effectiveness of our vulnerability detection model.

#### 4.2.2 Vector Quantization

Vector quantization is a technique used to reduce the dimensionality of data by representing a large set of vectors with a smaller set of prototype vectors, often referred to as centroids. Each vector in the original set is assigned to the nearest prototype vector, effectively quantizing the original data into a compressed representation.

**Motivation.** While the optimal transport learns to transfer vulnerability vectors to centroids, we still need to determine how we assign each vulnerability vector to its corresponding centroid during training. Inspired by the VQ-VAE approach in the computer vision domain [111], we leverage the VQ principle to address the centroid assignment for each vulnerability vector. In particular, each vulnerability vector will be assigned to its representative centroid based on the Euclidean distance, allowing us to effectively group similar vulnerability vectors during training.

#### 4.2.3 Related Work

##### 4.2.3.1 Deep Learning-based Vulnerability Detection

Prior works proposed various deep learning-based vulnerability detections (VDs) such as convolutional neural networks (CNNs) [17], recurrent neural networks (RNNs) [16, 118, 119], graph neural networks (GNNs) [14, 18, 32, 108, 109, 120], and pre-trained transformers [1, 40, 109, 121]. RNN-based methods [15–17] have been shown more accurate than program analysis tools such as Check-

---

marx [29] and RATS [27] to predict function-level vulnerabilities.

However, RNNs face difficulty in capturing long-term dependencies in long sequences as the model’s sequential nature may result in the loss of earlier sequence information. Furthermore, function-level predictions lack the required granularity to accurately locate vulnerable lines. Thus, prior works proposed transformer-based methods that predict line-level vulnerabilities and capture long-term dependencies [1, 109].

Ding et al.[109] propose an ensemble approach that uses a transformer model to capture global features and a GNN to capture local features while Fu et al.[1] leverage a pre-trained transformer model and interpret its attention scores as line-level predictions. In addition, Nguyen et al.[122] leverages bidirectional RNNs with information theory to detect line-level vulnerabilities.

On the other hand, Zhou et al.[18] embed the abstract syntax tree (AST), control flow graph (CFG), and data flow graph (DFG) for a code function and learn the graph representations for function-level predictions. Nguyen et al.[120] proposed constructing a code graph as a flat sequence for function-level predictions. Hin et al.[108] constructed program dependency graphs (PDGs) for functions and predicted line-level vulnerabilities.

While previous studies focus on varying model architectures such as RNNs, GNNs, and transformers, this work emphasizes a distinct avenue. We noticed that vulnerability scope embeddings tend to cluster within the feature space. Despite the potential enhancement this characteristic could offer to VD models, prior works have overlooked this critical aspect. In response to this gap, we employ the principle of vector quantization (VQ) with optimal transport (OT) to exploit the clustering characteristics and hidden patterns inherent in vulnerability scope embeddings. Thus, this work represents a pioneering step in learning and matching vulnerability patterns using OT and VQ.

#### 4.2.3.2 Large Language Models for Vulnerability Detection

Recent works have explored the use of large language models (LLMs) for vulnerability detection (VD) [6, 123]. Fu et al.[6] evaluated the performance of zero-shot ChatGPT for tasks such as vulnerability detection, classification, and repair. Steenhoek et al.[123] fine-tuned LLMs such as UniXcoder [124] for VD. It is worth noting that LLMs like ChatGPT [125], BARD [126], and CodeX [127] have demon-

---

strated their capabilities in source code-related tasks, such as code generation. However, these LLMs were not originally designed for software security applications. Additionally, the architectures of ChatGPT, BARD, and CodeX are not fully open-sourced. While our proposed framework could theoretically be applied to these LLMs, the lack of access to modify the model architecture makes it challenging for us to implement our framework effectively for these advanced LLMs.

Our study distinguishes itself from prior works that primarily concentrate on model architecture or the use of large language models (LLMs). Instead, this work aims to bridge the gap between the clustering characteristics of vulnerability scope embeddings observed in the training data and vulnerability detection (VD) models. Since large language models (LLMs), typically consisting of billions of parameters, are not the primary focus of this work, we deliberately selected base-size language models with 125M-225M parameters. This decision was made to enhance the accessibility of our approach and research, as well as to facilitate future reproduction. By opting for base-size language models, we aim to reduce resource consumption and lower the barrier of computational requirements.

## 4.3 Approach

### 4.3.1 Problem Statement

Let us consider a dataset of  $N$  functions in the form of source code. The data set includes both vulnerable and benign functions, where the function-level and line-level ground truths have been labeled by security experts. We denote a function as a set of code lines,  $X_i = [\mathbf{x}_1, \dots, \mathbf{x}_n]$ , where  $n$  is the max number of lines we consider in a function. Let a sample of data be  $\{(X_i, y_i, \mathbf{z}_i) : X_i \in \mathcal{X}, y_i \in \mathcal{Y}, \mathbf{z}_i \in \mathcal{Z}, i \in \{1, 2, \dots, N\}\}$ , where  $\mathcal{X}$  denotes a set of code functions,  $\mathcal{Y} = \{0, 1\}$  with 1 represents vulnerable function and 0 otherwise, and  $\mathcal{Z} = \{0, 1\}^n$  denotes a set of binary vectors with 1 represents vulnerable code line and 0 otherwise. Our objective is to identify the vulnerability on both *function and line levels*. It should be emphasized that our experimental datasets label the information of vulnerable code lines,  $\mathcal{Z}$ . However, it is crucial to note that we only rely on this information during supervised training, while *such information is not required by our method during the validation and testing phases*.

We formulate function-level vulnerability detection (VD) as a binary classification and line-level VD as a multi-label classification problem. Given  $X$ , we use an RNN denoted as  $RNN_{line}$  to obtain  $\mathbf{S} \in \mathbb{R}^{n \times d}$ , the d-dimensional line embeddings

---

for  $X$ . Let us denote  $X^{vul}$  as a set of all vulnerable lines in a vulnerable function. We extract  $X^{vul}$  from a vulnerable function and embed those vulnerable lines as  $\mathbf{P} \in \mathbb{R}^{q \times d}$ , where  $q$  is the maximum number of vulnerable lines. We use the same  $RNN_{line}$  to embed  $X^{vul}$ .

Note that for a benign function with no vulnerable lines, we use a special learnable embedding denoted as  $\mathbf{P}_{benign} \in \mathbb{R}^{q \times d}$ . We transform  $\mathbf{P}$  into a flat vulnerability vector (denoted as  $\mathbf{v} \in \mathbb{R}^d$ ) using an RNN denoted as  $RNN_{vul}$ . Additionally, we transform  $\mathbf{P}_{benign}$  into a flat vector denoted as  $\mathbf{v}_b$  for a benign function. We collect all  $\mathbf{v}$  to form a vulnerability collection  $\mathcal{V}_v = [\mathbf{v}_1, \dots, \mathbf{v}_a]$  where  $a$  is the total number of vulnerable functions in our training set. We obtain around 6,361 vulnerability vectors from our training set. Handling such an extensive collection size ( $\mathcal{V}_v$ ) will require significant computing resources during our vulnerability matching inference, as each testing sample needs to undergo 6,361 matches. Thus, we condense  $\mathcal{V}_v$  using optimal transport (OT) to cluster vulnerability centroids, denoted as  $\mathbf{c}$ . Each  $\mathbf{c}$  represents a set of similar  $\mathbf{v}$ . Subsequently, we construct our vulnerability codebook  $\mathcal{C} = [\mathbf{c}_1, \dots, \mathbf{c}_k]$ , with  $k$  representing the number of vulnerability centroids.

Let us denote a stack of transformer encoders as  $\mathcal{F}$ . In the warm-up training, we input  $\mathbf{H} = \mathbf{S} \oplus \mathbf{v} \in \mathbb{R}^{(n+1) \times d}$  into  $\mathcal{F}$ . We then make function and line-level predictions based on the output of  $\mathcal{F}$ .

#### 4.3.2 Line Embedding Using RNN

Prior research has used subword tokenization techniques to segment code functions [1, 40, 108, 121]. In these studies, the approach involved concatenating individual code lines within a function and then transforming them into a sequence of subword tokens using the byte pair encoding (BPE) algorithm [50]. However, many base-size models, such as CodeBERT-base, face a limitation in processing only up to 512 token embeddings. Consequently, this limitation has the potential to result in information loss, particularly for longer functions that exceed the 512-token threshold.

To address this limitation, our embedding transforms each code line into a sequence of subword tokens, without concatenating the lines. Each code line is represented as a set of token embeddings. Subsequently, we utilize a shared RNN to summarize these token embeddings within each line, thus creating a line embedding and representing each code line as a vector.

---

Given  $X = [\mathbf{x}_1, \dots, \mathbf{x}_n]$ , we use BPE to tokenize  $\mathbf{x}$  to a list of tokens,  $[\mathbf{t}_1, \dots, \mathbf{t}_r]$ , where  $r$  is the number of tokens we consider in a code line. We embed each  $\mathbf{t}$  using a token embedding layer  $E \in \mathbb{R}^{v \times d}$  where  $v$  is the vocab size. This results in a  $d$ -dimensional token embedding, denoted as  $\mathbf{S}_i \in \mathbb{R}^{n \times r \times d}$ . We then input token embeddings into an RNN and get line embeddings  $\mathbf{S}$ , which can be summarized as  $RNN_{line}(E(\mathbf{x}_1), \dots, E(\mathbf{x}_n))$  where  $E(\mathbf{x}_i) \in \mathbb{R}^{r \times d}$ . We use the identical line embedding method to embed  $X^{vul}$  as  $\mathbf{P}$ .

It is worth noting that our line embedding methods can process up to 3,100 tokens per input function which is six times more than 512 tokens that can be accepted by common source code language models such as CodeBERT-base [40]. Table 4 shows that our line embedding method results in more than 30% enhancements in the performance of the CodeBERT and CodeGPT models for line-level predictions.

Our approach consists of a warm-up phase (Figure 9) and a main training phase (Figure 10). During the warm-up, we gather vulnerability vectors  $\mathbf{v}$  to create a vulnerability collection  $\mathcal{V}_v$  and fine-tune model parameters, enhancing the representation of line embeddings for input functions and vulnerability scopes. In the main training phase, we execute our core proposal—quantizing  $\mathbf{v}$  into their respective vulnerability centroid  $\mathbf{c}$ , aligning with the conceptual framework illustrated in Figure 8. Now, we delve into the technical details of these two training phases.

#### 4.3.3 Training of Warm-Up Phase

For a vulnerable function, we concatenate line embeddings  $\mathbf{S}$  with a vulnerability vector  $\mathbf{v}$  as  $\mathbf{H}$  and input it to  $\mathcal{F}$ . For a benign function, we input  $\mathbf{H} = \mathbf{S} \oplus \mathbf{v}_b$  to  $\mathcal{F}$ . This input consists of  $n$  code line embeddings with an additional vulnerability vector. We select the  $n$  code line embeddings as  $\mathbf{H}[1 : n]$ . We predict line-level vulnerabilities with  $\mathbf{H}[1 : n]W^I$ , where  $W^I \in \mathbb{R}^{d \times 1}$ . We apply an RNN denoted as  $RNN_{func}$  to summarize line embeddings into a vector to predict function-level vulnerabilities with  $RNN_{func}(\mathbf{H}[1 : n])W^J$ , where  $W^J \in \mathbb{R}^{d \times 2}$ . We minimize the training objective as follows:

$$\frac{1}{N} \sum_{i=1}^N \left[ \mathcal{L}_f(RNN(\mathcal{F}(\mathbf{S}, \mathbf{v})), y) + \mathcal{L}_s(\mathcal{F}(\mathbf{S}, \mathbf{v}), \mathbf{z}) \right] \quad (2)$$

where  $\mathcal{L}_f$  and  $\mathcal{L}_s$  are cross-entropy over function and line labels respectively.

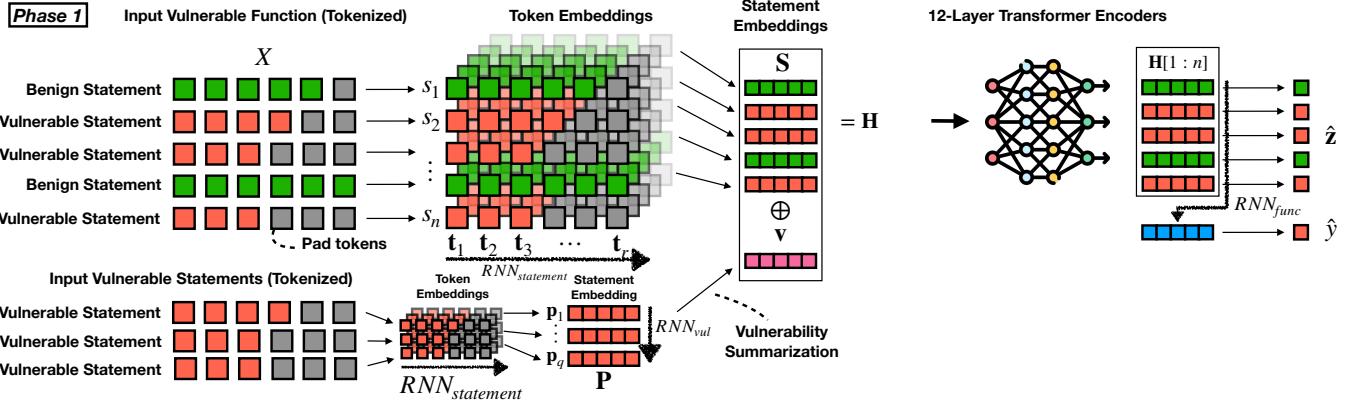


Figure 9: The overview of the warm-up phase in our approach. We tokenize each line in a vulnerable function (i.e.,  $X$ ) followed by an embedding layer to map each token into a vector. We use  $RNN_{line}$  to summarize the token embeddings and get the line embedding ( $\mathbf{S}$ ,  $\mathbf{P}$ ). For benign functions,  $\mathbf{P}$  is replaced by a special learnable embedding,  $\mathbf{P}_{benign}$ . We use  $RNN_{vul}$  to summarize the embeddings of vulnerable lines  $\mathbf{P}$  in a vector  $\mathbf{v}$  that represents the vulnerability scope. We concatenate  $\mathbf{S}$  and  $\mathbf{v}$  as the input to transformer encoders to consider vulnerability scopes that arise in the function and align with our vulnerability matching process. We select the line embeddings output from the last encoder, i.e.,  $\mathbf{H}[1 : n]$ . Each line embedding vector is mapped to a probability as line-level predictions, the function-level prediction is obtained by summarising  $\mathbf{H}[1 : n]$  to a vector using an  $RNN_{func}$  and mapping it to a probability.

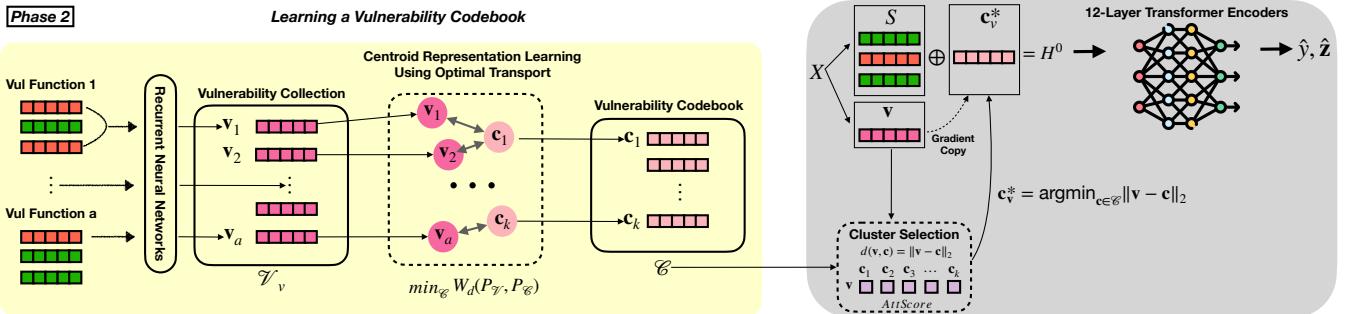


Figure 10: The overview of the main training phase in our approach. We introduce how to learn our vulnerability codebook on the left. We first collect a set of vulnerable line embeddings from our training data. We then use  $RNN_{vul}$  to pool a set of line embeddings from each vulnerable function, forming a vulnerability vector  $\mathbf{v}$ . The set of these scopes forms our vulnerability collection  $\mathcal{V}_v = \{\mathbf{v}_1, \dots, \mathbf{v}_a\}$ . Next, we learn vulnerability centroids  $\mathbf{c}$  using the Wasserstein distance metric to create a more compact vulnerability codebook  $\mathcal{C} = \{\mathbf{c}_1, \dots, \mathbf{c}_k\}$ , where each centroid represents a group of  $\mathbf{v}$ . During training, we minimize the Wasserstein distance between each  $\mathbf{v}$  and its corresponding centroid  $\mathbf{c}_v^*$ . As shown on the right, we input  $\mathbf{H} = \mathbf{S} \oplus \mathbf{c}_v^*$  to  $\mathcal{F}$ . To overcome the non-differentiability of the argmin operation in the networks, we copy the gradients from  $\mathbf{v}$  to  $\mathbf{c}_v^*$  to learn  $RNN_{line}$  and  $RNN_{vul}$ .

---

#### 4.3.4 Quantizing Vulnerability Vectors: Optimal Transport and Subsequent Main Training Phase

In this section, we introduce how we construct vulnerability collection  $\mathcal{V}_v$ . Then we describe how we transform  $\mathcal{V}_v$  into a vulnerability codebook  $\mathcal{C}$  to reduce the collection size and facilitate more efficient vulnerability matching. Finally, we introduce how we use  $\mathcal{C}$  in our main training phase.

##### 4.3.4.1 Collect Vulnerability Vectors from Vulnerable Functions

We extract vulnerable lines  $X^{vul}$  in the training set and use  $RNN_{line}$  to embed  $X^{vul}$  as  $\mathbf{P}$ . We summarize each  $\mathbf{P}$  into a vulnerability vector  $\mathbf{v}$  using  $RNN_{vul}$ . We collect a total of 6,361  $\mathbf{v}$  from all of vulnerable  $X$  in our training set to form our vulnerability collection  $\mathcal{V}_v \in \mathbb{R}^{6,361 \times h}$ . We reduce the d-dimensional  $\mathbf{v}$  to h-dimensional to facilitate the upcoming clustering. The large collection size of  $\mathcal{V}_v$  will demand substantial computing resources during inference because we need to match each function with 6,361 vulnerability vectors. Therefore, we use optimal transport (OT) to quantize similar vulnerability vectors that share the same vulnerability patterns into vulnerability centroids. In what follows, we outline the process of OT that effectively reduces 6,361 vulnerability vectors to 150 centroids.

##### 4.3.4.2 Learn to Transport Vulnerability Vectors to Vulnerability Centroids

As depicted in the left part of Figure 10, our goal is to quantize  $\mathcal{V}_v$  to a vulnerability codebook,  $\mathcal{C} = [\mathbf{c}_1, \dots, \mathbf{c}_k]$ . This codebook consists of a more compact set of vulnerability patterns. To ensure that each  $\mathbf{c}$  can represent a group of similar  $\mathbf{v}$ , we leverage the optimal transport (OT) theory to transfer sets of  $\mathbf{v}$  to their corresponding  $\mathbf{c}$ .

We randomly initialize the embedding space of our vulnerability codebook as  $\mathcal{C} = [\mathbf{c}_1, \dots, \mathbf{c}_k]$  with the  $k$  number of clusters. We minimize the Wasserstein distance [117] using the Sinkhorn approximation [128] between  $\mathcal{V}_v$  and  $\mathcal{C}$ . Consequently,  $\mathbf{v}$  and their respective  $\mathbf{c}$  will converge toward each other as shown in Figure 8. Our codebook will ultimately comprise vulnerability centroids acting as representative patterns that symbolize different sets of vulnerability vectors. This allows us to aggregate similar vulnerability scopes into patterns based on the Euclidean distance. We summarize the process as follows:

$$\min_{\mathcal{C}} W_d := W_d(\mathbb{P}_{\mathcal{V}_v}, \mathbb{P}_{\mathcal{C}}) \quad (3)$$

---

where  $d(\mathbf{v}, \mathbf{c}) = \|\mathbf{v} - \mathbf{c}\|_2$  represents Euclidean distance,  $W_d$  is the Wasserstein distance [117],  $\mathbb{P}_{\mathcal{V}_v} = \frac{1}{a} \sum_{i=1}^a \delta_{\mathbf{v}_i}$ ,  $\mathbb{P}_{\mathcal{C}} = \frac{1}{a} \sum_{j=1}^a \delta_{\mathbf{c}_j}$ , and  $\delta_{\mathbf{c}_j}$  represents the Dirac delta distribution. According to the clustering view of optimal transport [129, 130], when minimizing  $\min_{\mathcal{C}} W_d(\mathbb{P}_{\mathcal{V}_v}, \mathbb{P}_{\mathcal{C}})$ ,  $\mathcal{C}$  will become the centroids of the clusters formed by  $\mathcal{V}_v$ . This clustering approach ensures that similar vulnerability scopes sharing the same vulnerability pattern are grouped, leading to a quantized vulnerability codebook involving common vulnerability patterns. Next, we introduce how we utilize our vulnerability codebook in the main training phase.

#### 4.3.4.3 Main Training Phase

The right part of Figure 10 summarizes our main training phase. We load the model parameters from the warm-up phase. We obtain the line embeddings  $\mathbf{S}$  and a vulnerability vector  $\mathbf{v}$  for the input function  $X$  as introduced in Section 4.3.2. We employ a cluster selection process inspired by VQ-VAE [111], utilizing the Euclidean distance to map  $\mathbf{v}$  to its most similar centroid denoted as  $\mathbf{c}_v^*$  selected from our codebook  $\mathcal{C}$ :

$$\mathbf{c}_v^* = \operatorname{argmin}_{\mathbf{c} \in \mathcal{C}} \|\mathbf{v} - \mathbf{c}\|_2 \quad (4)$$

Note that for a benign function, we directly assign  $\mathbf{c}_v^*$  as  $\mathbf{v}_b$ . We then concatenate  $\mathbf{S}$  with  $\mathbf{c}_v^*$  and input  $\mathbf{H} = \mathbf{S} \oplus \mathbf{c}_v^*$  to  $\mathcal{F}$ . The subsequent forward passes are the same as our warm-up phase. We minimize the loss function as follows:

$$\frac{1}{N} \sum_{i=1}^N \left[ W_d + \mathcal{L}_f \left( RNN(\mathcal{F}(\mathbf{S}, \mathbf{c}_v^*)), y \right) + \mathcal{L}_s \left( \mathcal{F}(\mathbf{S}, \mathbf{c}_v^*), \mathbf{z} \right) \right] \quad (5)$$

Note that no real gradient is defined for  $\mathbf{v}$  once we map it to a  $\mathbf{c}_v^*$  via the argmin operation in Equation 4. To let the RNNs that embed and summarize vulnerable lines be trainable via backpropagation, we follow the idea in VQ-VAE [111] which was shown effective for vector quantization. We copy gradients from  $\mathbf{v}$  to  $\mathbf{c}_v^*$ . Below, we present how to use our learned codebook for vulnerability matching during inference.

#### 4.3.5 Vulnerability Detection Through Explicit Vulnerability Patterns Matching

Similar to static analysis tools that identify vulnerabilities by matching predefined patterns, we match our pre-learned vulnerability codebook to detect vulnerabilities using deep learning models.

For each testing sample  $X$ , we obtain d-dimensional line embeddings  $\mathbf{S}$ . We input  $\mathbf{H} = \mathbf{S} \oplus \mathbf{c}$  to  $\mathcal{F}$ . We select the  $n$  code line embeddings as  $\mathbf{H}[1 : n]$ . We

---

obtain function and line-level predictions based on the output of  $\mathcal{F}$ . We apply  $RNN_{func}$  to summarize line embeddings into a vector to predict function-level vulnerabilities with  $RNN_{func}(\mathbf{H}[1 : n])W^J$ , where  $W^J \in \mathbb{R}^{d \times 2}$ . We predict line-level vulnerabilities with  $\mathbf{H}[1 : n]W^I$ , where  $W^I \in \mathbb{R}^{d \times 1}$ . This process is iterated for  $k = 150$  times to match all of the vulnerability centroids  $\mathbf{c}$  in our codebook  $\mathcal{C}$ .

To aggregate the  $k$  number of predictions produced by our matching process, apply max and mean pooling. We use max pooling to select the most prominent matching result and then use argmax to obtain the function-level prediction  $\hat{y}$ . If  $X$  is predicted as a benign function, we directly output a zero vector as the line-level prediction. Otherwise, we use mean pooling to aggregate the line-level predictions into a flat probability vector. We apply a probability threshold of 0.5 to transform this probability vector into a binary vector as the final line-level prediction  $\hat{z}$ .

## 4.4 Experimental Design and Results

### 4.4.1 Research Questions

One of the key goals of this work is to evaluate our OptiMatch thoroughly by comparing it with other baseline approaches that focus on the function-level and line-level vulnerability detection tasks. We also formulate an ablation study to support the design decision of our OptiMatch approach. Below, we present the motivation for our two research questions:

**What is the accuracy of our OptiMatch for predicting function-level and line-level vulnerabilities?** Recent studies have leveraged pre-trained language models with transformer architectures to perform vulnerability detection effectively [1, 109, 123]. We found that different vulnerable functions may form similar vulnerability patterns, as demonstrated in Figure 7. This discovery suggests valuable information that could enhance the performance of deep learning (DL) models for locating line-level vulnerabilities. However, the use of vulnerability patterns in DL models remains unexplored. To address this gap, we propose OptiMatch, an innovative framework that harnesses optimal transport (OT) and vector quantization (VQ) techniques to learn and match vulnerability patterns. This approach could facilitate the precise identification of vulnerabilities at both function and line levels. Consequently, we formulate this RQ to evaluate the accuracy of our proposed framework in function and line-level vulnerability detection.

---

## What are the contributions of each component in our OptiMatch approach?

Our OptiMatch framework incorporates two crucial components: (i) code line embeddings using RNNs, and (ii) optimal transport (OT) and vector quantization (VQ). However, the specific contributions of these components remain unexplored. We thus formulate this research question as an ablation study to assess the components in our OptiMatch framework and empirically validate our design decisions.

### 4.4.2 Baseline Approaches

We compare our OptiMatch approach with language models pre-trained on code data, state-of-the-art (SOTA) transformer-based, GNN-based, RNN-based, and CNN-based vulnerability detection (VD) approaches proposed by prior studies. For all of the pre-trained transformer baselines, we use the standard size model (e.g., CodeT5-base, CodeBERT-base, etc.) to ensure a fair comparison where each transformer baseline has a similar number of parameters of our approach. We reproduce 12 baselines in total, where each baseline is fine-tuned on our studied dataset based on the code and hyperparameters provided by the original authors.

**Language Models (LMs) for code:** We include five LMs pre-trained for code-related tasks: **CodeT5P-220m** [131], **CodeT5-base** [132], **CodeBERT-base** [40], **CodeGPT** [133], and **GraphCodeBERT-base** [121].

**Transformer-based VD:** We include 2 transformer-based baselines as follows:

- **LineVul** [1] is designed to perform function-level prediction by leveraging a pre-trained transformer model. Although it can also provide line-level predictions by interpreting and ranking the attention scores of the transformer, this approach is not suitable for the line-level classification setting. To ensure a fair comparison, we only evaluate our approach against LineVul on the function level.
- **VELVET** [109] is an ensemble method that leverages a vanilla transformer with GNNs.

**GNN-based VD:** We include 3 graph-based baselines as follows (ReGVD and Devign only predict function-level vulnerabilities):

- 
- LineVD [108] leverages pre-trained CodeBERT embeddings with GNNs to detect line-level vulnerabilities.
  - ReGVD [120] represents a code function as a sequential graph and uses GNNs to detect function-level vulnerabilities.
  - Devign [18] leverages code property graph (CPG) [134] with GNNs to detect function-level vulnerabilities.

**RNN-based and CNN-based VD:** We include an RNN-based and a CNN-based baseline as follows:

- **ICVH** [122] leverages Bi-RNN with information theory to detect line-level vulnerabilities. ICVH was initially trained in the unsupervised setting for line-level vulnerability prediction, but we found that it was not effective in our context. Therefore, we adopted the original ICVH architecture and added a cross-entropy loss to train ICVH in the supervised setting to achieve a fair comparison.
- **TextCNN** [135] uses convolutional layers for sentence classification tasks.

#### 4.4.3 Experimental Datasets

It is important to highlight that common vulnerability datasets such as Devign [18] and DiverseVul [136] are not included in this study due to the absence of line-level vulnerability labels. To identify vulnerabilities on function and line levels, we select the Big-Vul dataset [70] and the D2A dataset [113] as they are two of the largest vulnerability data sets with line-level vulnerability labels and has been used to assess line-level vulnerability detection methods [1, 108]. Big-Vul was collected from 348 Github projects and consists of 188k C/C++ functions with 3,754 code vulnerabilities spanning 91 vulnerability types. The data distribution of Big-Vul resembles real-world scenarios, where the proportion of vulnerable to benign functions is 1:20. In contrast, D2A comprises around 6.5k samples, with an approximate 1:1 ratio of vulnerabilities, all of which were extracted from real-world projects.

#### 4.4.4 Parameter Settings and Model Training

We split the data into 80% for training, 10% for validation, and 10% for testing. For both our approach and baselines, we consider  $n = 155$  lines in each function

---

and  $r = 20$  tokens in each line as the descriptive statistics of the whole data set suggest that 95% of source code functions have less than 155 lines and 95% of lines have less than 20 tokens. We use CodeT5-base [132] to initialize our transformer encoders. We provide all the details of the hyperparameter settings in our replication repository. In both training phases, we train our model through specific epochs and select the model that demonstrates the highest F1 score for line-level prediction in the validation set. The experiments were conducted on a Linux machine with an AMD Ryzen 9 5950X processor, 64 GB of RAM, and an NVIDIA RTX 3090 GPU.

**What is the accuracy of our OptiMatch for predicting function-level and line-level vulnerabilities?**

**Approach.** We conduct experiments on the Big-Vul [70] and D2A [113] datasets described in Section 4.4.3 and compare our OptiMatch methods with 12 other baselines described in Section 4.4.2. For both function and line-level vulnerability prediction, we report: Precision (Pre) =  $\frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$ ; Recall (Re) =  $\frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$ ; and  $F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$ . These measures enable us to assess the models' performance on both positive and negative classes, regardless of the class imbalance between vulnerable and benign functions. It is important to note that the line-level metrics are computed on the line level instead of the function level to determine if each line is correctly predicted. Furthermore, we conduct an extra trial for four baselines: CodeT5P-220m, CodeT5-base, CodeBERT-base, and CodeGPT. We use our RNN line embedding method instead of their token embedding to explore the potential enhancement of their performance. Note that our line embedding is not compatible with GraphCodeBERT's data flow construction.

**Result.** Table 4 presents the experimental results of our OptiMatch approach and 12 other baseline approaches according to the function and line-level precision, recall, and F1 score. **Our approach achieves an line-level F1 score of 82% which reveals a 32% improvement over the best baseline approach, VELVET [109].** Our approach achieves the highest performance across all metrics in the Big-Vul dataset [70]. Furthermore, it also secures the top F1 score for both function and line-level prediction in the D2A dataset [113]. **Notably, our RNN line embedding method significantly enhances the line-level F1 score of CodeT5 (29% → 72%), CodeT5P(29% → 71%), CodeBERT(30% → 63%), and CodeGPT(12% → 44%) on both datasets in line-level vulnerability prediction.** This suggests that our RNN line embedding approach is better suited

Table 4: (RQ1 Results) We compare our OptiMatch approach against 12 other baseline methods and present results in percentage.

Methods	Function Level			Line Level		
	Pre	Re	F1	Pre	Re	F1
Dataset	<i>Big-Vul Dataset</i>					
OptiMatch (Ours)	<b>97.66</b>	<b>89.83</b>	<b>93.58</b>	<b>86.80</b>	<b>77.96</b>	<b>82.14</b>
CodeT5P+Our Embedding	96.19	87.56	91.67	67.59	74.80	71.00
CodeT5P	94.04	83.01	88.18	18.79	68.07	29.45
CodeT5+Our Embedding	96.63	89.11	92.72	78.65	66.29	71.94
CodeT5	94.90	84.57	89.44	18.15	68.60	28.70
CodeBERT+Our Embedding	92.15	82.89	87.28	59.39	67.84	63.33
CodeBERT	93.90	77.27	84.78	19.29	63.54	29.60
CodeGPT+Our Embedding	91.25	84.81	87.91	32.54	67.34	43.88
CodeGPT	56.18	19.02	28.42	44.40	9.70	11.60
GraphCodeBERT	50.11	27.03	35.12	10.59	26.34	15.08
LineVul	89.25	78.47	83.15	-	-	-
VELVET	93.37	80.86	86.67	38.19	73.50	50.26
LineVD	-	-	-	27.10	53.30	36.00
ReGVD	77.92	50.24	61.09	-	-	-
Devign	72.29	50.24	59.28	-	-	-
ICVH	77.44	33.25	46.53	21.31	43.17	28.54
TextCNN	62.31	25.12	35.81	21.03	28.91	24.34
Dataset	<i>D2A Dataset</i>					
OptiMatch (Ours)	<b>54.98</b>	<b>73.89</b>	<b>63.04</b>	<b>23.46</b>	31.53	<b>26.90</b>
CodeT5P+Our Embedding	49.24	51.27	50.23	7.78	38.22	12.93
CodeT5P	58.18	61.15	59.63	1.74	56.37	3.38
CodeT5+Our Embedding	52.38	49.04	50.66	18.79	26.75	22.08
CodeT5	53.68	62.74	57.86	1.42	<b>57.01</b>	2.89
CodeBERT+Our Embedding	60.07	55.10	57.48	3.89	45.54	7.18
CodeBERT	67.12	31.21	42.61	2.70	30.89	4.96
CodeGPT+Our Embedding	62.13	33.44	43.48	3.32	31.21	6.01
CodeGPT	64.37	17.83	27.93	2.40	11.46	3.96
GraphCodeBERT	66.36	45.22	53.79	1.66	53.15	3.21
LineVul	<b>67.86</b>	30.25	41.85	-	-	-
VELVET	51.06	46.84	48.86	3.31	38.71	6.10
LineVD	-	-	-	3.24	37.07	5.95
ReGVD	59.87	58.92	59.39	-	-	-
Devign	50.95	68.47	58.42	-	-	-
ICVH	56.99	50.64	53.63	2.32	50.64	4.44
TextCNN	57.29	52.55	54.82	2.00	50.96	3.84

Table 5: (RQ2 Results) We compare our proposed method to other variants to investigate the impact of the individual components. The metrics are reported as percentages.

Methods	Function Level			Line Level		
	Pre	Re	F1	Pre	Re	F1
<b>Dataset</b>	<i>Big-Vul Dataset</i>					
OptiMatch (Ours)	97.66	89.83	93.58	86.80	77.96	<b>82.14</b>
w/o RNN emb (mean pooling)	98.49	86.00	91.83	<b>90.40</b>	67.89	77.54
w/o RNN emb (max pooling)	96.53	89.95	93.13	79.70	76.40	78.02
w/o codebook & matching	45.91	86.60	60.00	28.77	51.57	36.94
wt 50 centroids	23.95	<b>98.21</b>	38.51	16.92	<b>86.13</b>	28.28
wt 100 centroids	98.13	87.92	92.74	88.14	74.98	81.03
wt 150 centroids (ours)	97.66	89.83	93.58	86.80	77.96	<b>82.14</b>
wt 200 centroids	96.69	90.91	<b>93.71</b>	83.44	80.02	81.69
wt 400 centroids	<b>99.05</b>	62.32	76.51	81.91	70.47	75.76

for representing code functions. Furthermore, line embeddings learn contextual information on the line level, which may capture the relationships and dependencies between lines more accurately than token embeddings. This makes line embeddings more effective than token embeddings for tasks that require a deeper understanding of the code structure such as detecting line-level vulnerabilities.

Our experimental results confirm the effectiveness of our proposed deep learning framework for learning and matching vulnerability patterns to predict function and line-level vulnerabilities. These findings also validate our intuition that the line embeddings learned by our proposed method can capture contextual information more effectively than token embeddings, leading to more accurate identification of lines associated with vulnerabilities.

#### What are the contributions of each component in our OptiMatch approach?

**Approach.** Our OptiMatch approach consists of two key components: (i) code line embedding using RNN, introduced in Section 4.3.2, and (ii) optimal transport (OT) and vector quantization (VQ), introduced in Section 4.3.4. We conduct an ablation study to assess the contribution of these proposed components. Firstly, we examine the effect of our RNN line embedding approach by comparing it with commonly used mean and max pooling line embedding methods proposed in Sentence-BERT [137]. Secondly, we investigate the impact of our main components, OT and VQ, on learning and matching vulnerability patterns. We compare our approach with an identical variant that does not utilize OT and VQ. Thirdly,

---

our OT process requires us to define  $k$  (i.e., the number of vulnerability centroids) to initialize the centroid representations before aggregating similar vulnerability vectors into these centroids. Thus, we analyze the effect of  $k$  on the performance of our approach. We compare our approach ( $k = 150$ ) with other variants where  $k$  takes different values, specifically  $k = [50, 100, 200, 400]$ .

**Result.** The results of our ablation study are presented in Table 5. **Our RNN line embedding approach further improves the performance of mean and max pooling by 0.45%-1.75% on the function-level F1 score and 4.12%-4.6% on the line-level F1 score.** The max pooling would lead to information loss since it considers the maximum token embedding for each line, discarding all other token embeddings in the sequence. While the mean pooling considers all token embeddings, it treats all the token embeddings equally regardless of their importance or relevance to the line they belong where the prominent token features could be disregarded.

In contrast, our RNN line embedding approach offers several advantages. Firstly, it learns token features at each time step, allowing for a more nuanced understanding of the code line. Additionally, unlike mean and max pooling, our RNN-based approach retains more information from the entire sequence, thereby capturing a richer representation of the code line. Lastly, by considering the sequential nature of the input tokens, our RNN model can better capture contextual dependencies and relationships within the line. The results confirm the effectiveness of our RNN line embedding method, indicating that it is more effective in summarizing token embeddings.

**Our main components, OT and VQ for learning and matching vulnerability patterns, significantly improve the variant, “w/o codebook & matching”, by 33.58% on the function-level F1 score and 45.2% on the line-level F1 score.** This underscores the importance of these components in achieving high performance levels. The results suggest that OT and VQ play crucial roles in learning our proposed vulnerability codebook, which is responsible for retaining and leveraging the vulnerability patterns information present in vulnerable functions. This information is then utilized to identify vulnerable lines effectively during the vulnerability-matching inference. The results confirm our design decision of leveraging OT and VQ to effectively aggregate vulnerability vectors into patterns and match those patterns during inference.

The lower section of Table 5 illustrates the impact of the number of vulnerability

---

centroids ( $k$ ) on our approach. The results demonstrate that our approach attains favorable line-level F1 scores for  $k \in [100, 150, 200]$ . Thus, in our OptiMatch approach, we empirically set  $k = 150$  as it produces the optimal line-level F1 score. Notably,  $k$  represents a crucial factor, where a small value of  $k$  (e.g., 50) may result in unsatisfactory performance due to the grouping of too many vulnerability vectors together, resulting in an inadequate representation of each pattern. Conversely, a large value of  $k$  (e.g., 400) leads to a substantial embedding space of our codebook, making it challenging to update during the backward process. The results confirm the effectiveness of selecting  $k = 150$  as the optimal value for the number of vulnerability centroids ( $k$ ) in our approach.

## 4.5 Discussion

In Section 4.4, we empirically evaluated the performance of our OptiMatch and conducted an ablation study to support our design rationale. However, several important questions remain unanswered by our RQ1 and RQ2. Specifically, it is unclear whether our RNN line embedding truly improves the model performance on long sequences as expected, whether our optimal transport process effectively aggregates vulnerability vectors into centroids in the vector space as expected, and whether a clustering algorithm can be used to automatically identify an ideal number of centroids rather than empirically determining it for our OptiMatch. Thus, in this section, we perform an extended analysis to address the three questions. Our analysis is conducted on the Big-Vul dataset, which consists of sufficient 188k samples with diverse vulnerability types (i.e., CWE-IDs) and line-level vulnerability labels.

### 4.5.1 Does our RNN line embedding method perform better on long sequences than the token embedding?

In Section 4.3.2, we introduced our RNN-based line embedding approach. Specifically, our embedding approach can process up to 3,100 tokens for input, significantly improving over the 512-token threshold of common base-size pre-trained language models. Our experimental results in Table 4 further confirm that our embedding approach substantially enhances the performance of four language models: CodeT5 [132], CodeT5P [131], CodeBERT [40], and CodeGPT [133]. However, it remains unclear whether our embedding approach can improve the performance of these language models in identifying vulnerable lines within long functions comprising more than 512 tokens. To investigate this, we compare the line-level performance of the four language models using token embedding

Table 6: (Discussion) The line-level performance comparison between the token embedding and our RNN line embedding.

Methods	Line Level					
	<= Tokens			>512 Tokens		
	Pre	Re	F1	Pre	Re	F1
Dataset	<i>Big-Vul Dataset</i>					
CodeT5P+Our Embedding	<b>69.62</b>	70.75	<b>70.18</b>	<b>66.03</b>	<b>78.33</b>	<b>71.66</b>
CodeT5P	24.42	<b>83.63</b>	37.80	14.36	54.52	22.73
CodeT5+Our Embedding	<b>79.72</b>	69.17	<b>74.07</b>	<b>77.67</b>	<b>63.77</b>	<b>70.04</b>
CodeT5	22.82	<b>83.38</b>	35.83	14.33	55.72	22.79
CodeBERT+Our Embedding	<b>64.26</b>	70.89	<b>67.41</b>	<b>55.41</b>	<b>65.18</b>	<b>59.90</b>
CodeBERT	24.53	<b>78.71</b>	37.41	14.94	50.32	23.04
CodeGPT+Our Embedding	<b>54.45</b>	<b>62.24</b>	<b>58.09</b>	<b>24.96</b>	<b>71.78</b>	<b>37.03</b>
CodeGPT	16.21	18.14	17.12	8.25	2.36	3.66

to those using our line embedding and present the performance based on sequence length.

As shown in Table 6, our line embedding helps the four LMs achieve the best overall performance for both short and long functions. In particular, our approach substantially enhances the line-level F1 scores, increasing from 23% to 72% for CodeT5P, from 23% to 70% for CodeT5, from 23% to 60% for CodeBERT, and from 4% to 37% for CodeGPT. The results validate the effectiveness of our RNN line embedding approach in accurately identifying vulnerable lines within long functions.

#### 4.5.2 Does our optimal transport process effectively aggregate vulnerability vectors into representative vulnerability centroids?

We collected over six thousand vulnerability vectors from our training dataset. However, matching all these patterns during the inference phase would be impractical and computationally intensive. To address this, in Section 4.3.4.2, we introduced the use of optimal transport (OT) theory. This approach aggregates similar vulnerability vectors, those closely positioned in the vector space according to Euclidean distance, into representative vulnerability centroids. The goal is for the vulnerability vectors and centroids to converge (as shown in Figure 8), minimizing the Wasserstein distance [117] by following equation 3. Although RQ1 and RQ2 demonstrated the performance improvements of our approach, it is essential to determine whether our OT process effectively aggregates vulnerability

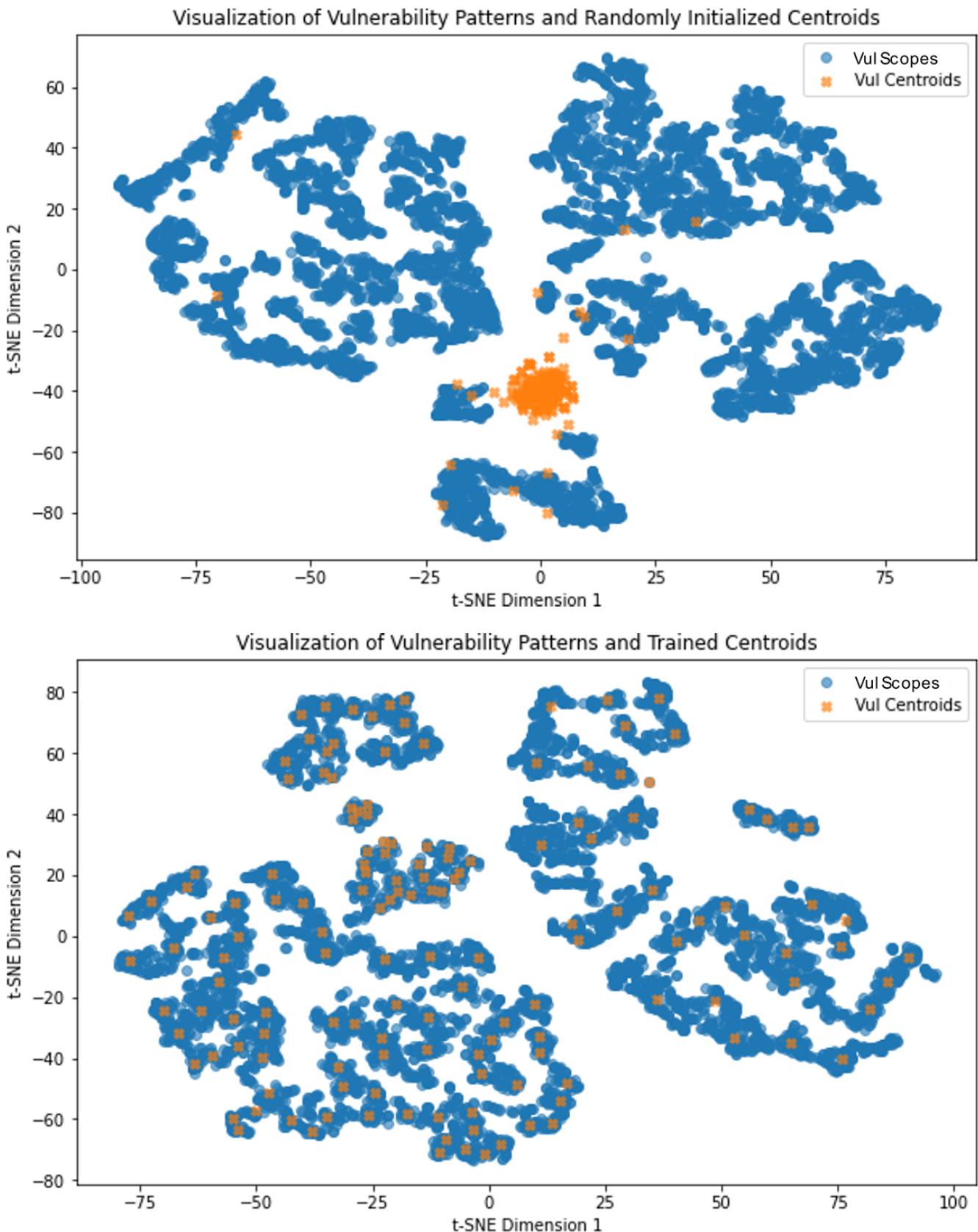


Figure 11: t-SNE visualization of vulnerability vectors and centroids. The plot above shows vulnerability vectors and randomly initialized centroids. It can be seen that after our optimal transport process of vulnerability centroids, the randomly initialized centroids become more representative, where each centroid is associated with a set of vulnerability vectors sharing the same vulnerability patterns.

---

Table 7: (Discussion) The comparison between our approach with randomly initialized vulnerability centroids and our approach with OPTICS clustering algorithm to determine the number of vulnerability centroids.

Methods	Function Level			Line Level		
	Pre	Re	F1	Pre	Re	F1
Dataset	<i>Big-Vul Dataset</i>					
Optimatch	97.66	89.83	93.58	86.80	77.96	82.14
Optimatch + OPTICS Clustering	95.80	89.95	92.78	82.13	76.47	79.20

vectors into representative centroids. To this end, we employ t-SNE visualization to assess whether each centroid represents a set of vulnerability vectors.

Figure 11 depicts the t-SNE visualization of our vulnerability vectors (depicted as blue dots) and vulnerability centroids (depicted as orange dots) both before and after our primary training phase outlined in Section 4.3.4.3. The upper section illustrates the vulnerability vectors and randomly initialized centroids before the optimal transport (OT) process while the lower section showcases the learned vulnerability vectors and their associated centroids following the OT process. This visualization validates that our proposed OT process consolidates closely related vulnerability vectors into representative centroids within the vector space.

#### 4.5.3 Can we use clustering algorithms to automatically identify an ideal number of centroids for our OptiMatch approach?

As introduced in Section 4.3.4.2, we need to empirically determine the number of centroids and randomly initialize them before proceeding with our optimal transport (OT) process. While our ablation study demonstrated that our 150 number of centroids is optimal, it could be computationally intensive for future works to empirically determine the number of centroids if using our approach on a large dataset. Thus, it is important to assess whether we can use clustering algorithms to automatically identify an ideal number of centroids for our OptiMatch approach.

Given that our task involves line-level vulnerability detection and the representation of vulnerability vectors, we sought a clustering algorithm that could adapt to varying cluster shapes and sizes without the need for pre-defined centroids. OPTICS (Ordering Points To Identify the Clustering Structure) [138] emerged as a compelling choice due to its density-based approach, which inherently accommodates clusters of different densities and shapes. Its ability to handle noise and outliers while remaining scalable to large datasets aligns well with our objec-

---

tives. Moreover, OPTICS obviates the need for pre-determining the number of clusters, offering flexibility in clustering our vulnerability vectors effectively. Thus, we evaluate our OptiMatch approach with the use of OPTICS to determine the optimal number of clusters and initialize centroids, replacing the 150 randomly initialized centroids.

As shown in Table 7, our approach achieves an F1 score of 94% and 82% at the function and line level predictions, respectively. Notably, OptiMatch+OPTICS Clustering also achieves comparable results, with F1 scores of 93% and 79% at the function and line levels, respectively. While the performance of OptiMatch+OPTICS Clustering is slightly lower than that of OptiMatch with randomly initialized centroids, it offers the advantage of automatically determining the ideal number of clusters without the need for empirical tuning. This can save computational resources, particularly for large-scale datasets. The OPTICS clustering algorithm identified a total of 115 clusters, consistent with the findings of our ablation study suggesting that 100-200 centroids are optimal for our studied dataset. These results validate the effectiveness of our OptiMatch approach when combined with the OPTICS clustering algorithm, which automatically identifies vulnerability centroids for the subsequent OT process.

## 4.6 Threats to Validity

**Threats to the construct validity** relate to the data quality and dataset selection. Our OptiMatch approach, similar to other data-driven tasks, relies on data quality, and the presence of noisy labels can impact the model’s performance. Croft et al.[139] conducted a systematic study to assess data quality, evaluating state-of-the-art vulnerability datasets such as Big-Vul [70], D2A [113], and Devign [18]. They analyzed the datasets manually to assess the accuracy of vulnerability labels. Among the three datasets, Devign achieved the highest label correctness. However, Devign only provides function-level labels, which are not compatible with our study’s focus on line-level vulnerability detection. It is worth noting that other common vulnerability datasets, such as Reveal [14] and DiverseVul [136], also only consist of function-level labels, which are not suitable for our study.

The Big-Vul dataset is one of the largest vulnerability datasets consisting of line-level vulnerability labels. It has been utilized in recent studies focusing on line-level vulnerability prediction [1, 32, 108]. The line-level labels in Big-Vul were derived from parsing code changes before and after addressing a vulnerability. However, the presence of noisy labels within the dataset may introduce bias

---

and compromise the generalizability of our OptiMatch approach. To mitigate this threat, we leveraged an additional experimental dataset, the D2A dataset [113], which also provides line-level labels. By including the D2A dataset, which has been used in previous line-level vulnerability detection studies [109], we enhance the robustness of our approach against potential biases introduced by noisy labels in the Big-Vul dataset.

**Threats to the internal validity** relate to hyperparameter settings in our OptiMatch approach. In particular, determining the number of vulnerability centroids,  $k$ , before the optimal transport (OT) process is crucial for our OptiMatch approach. We empirically initialize 150 centroids for the OT process, as shown in Table 5. However, this parameter  $k$  significantly impacts the performance of our approach, as discussed in our ablation study (RQ2). Empirically determining  $k$  could be impractical and computationally intensive for future studies with large-scale datasets. To address this threat, we explore using a clustering algorithm to automatically determine an optimal  $k$  in our extended discussion in Section 4.5.3. The results validate that our OptiMatch approach, combined with the OPTICS clustering algorithm [138], can determine  $k$  while maintaining comparable performance, outperforming all other baselines in Table 4.

**Threats to the external validity** relate to the generalizability of our OptiMatch approach. We conduct our experiment using Big-Vul [70] and D2A [113] datasets consisting of a large amount of C/C++ functions parsed from real-world software projects. However, our OptiMatch method is not necessarily to generalize to other datasets. To mitigate this threat, we open-source our experimental dataset and model training script in our public replication package available at <https://github.com/awsm-research/optimatch>. Nevertheless, other vulnerability datasets can be explored in future work.

## 4.7 Summary

In this chapter, we introduce OptiMatch, an approach for function- and line-level vulnerability detection (VD) that leverages vector quantization (VQ), optimal transport (OT), and a novel vulnerability-matching method. Our approach capitalizes on the vulnerability patterns present in vulnerable programs, which are typically overlooked in deep learning-based VD. Specifically, we collect vulnerability patterns from the training data and learn a more compact vulnerability codebook from the pattern collection using optimal transport (OT) and vector quantization. During inference, the codebook matches all learned patterns and detects poten-

---

tial vulnerabilities within a program. The evaluation results demonstrate that our method surpasses twelve other competitive baseline methods, while our ablation study confirms the soundness of our approach. Thus, we expect that our OptiMatch may help security analysts accurately pinpoint vulnerable lines in vulnerable programs.

---

## **5 A Novel Knowledge Distillation Framework for Explaining Detected Vulnerabilities**

© 2023 IEEE. This is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive version was published in IEEE Transactions on Software Engineering, **49**(10), pp. 4550–4565, published 01 October 2023, <https://doi.org/10.1109/TSE.2023.3305244>.

---

## 5.1 Introduction

As the number of discovered software vulnerabilities hit an all-time high of 20k in 2021 reported by National Vulnerability Database (NVD) [106], large software companies are spending more and more funds mitigating the security threats by granting bug bounties [140–142]. Software vulnerabilities are system weaknesses and glitches that can be further exploited by attackers to steal sensitive data or spread ransomware. Back in 2000, NVD was created by the U.S. government to analyze and track new vulnerabilities to mitigate software security breaches. Another community-developed Common Weakness Enumeration (CWE) [37] list consists of multiple CWE-IDs representing various categories of vulnerability, where some CWE-IDs are easier to exploit than others, hence requiring higher priority to be resolved. For instance, the widespread Log4j flaw inside an open-source Java library provided by the Apache Software Foundation was found at the end of 2021. Such flaws include different CWE-IDs such as CWE-20 (i.e., improper input validation) and CWE-89 (i.e., improper neutralization of special elements used in an SQL command) with a high likelihood of exploitation [143]. Thus, it is important to recognize the type of vulnerability for a vulnerable program that enables security engineers to prioritize accordingly to focus on the more severe ones.

Various Deep Learning-based software vulnerability prediction (SVP) methods have been proposed that can even detect the vulnerabilities down to line-level [1, 108]. Nevertheless, those models can not identify what type of vulnerability is detected. The vulnerability type (i.e., CWE-ID) further explains the detected vulnerable code and helps security engineers understand and categorize the detected vulnerability to propose repairs or mitigation. Thus, software vulnerability classification (SVC) is an important task that supports SVP models by providing more explanation of detected vulnerable code, which could assist end users in comprehending the detected vulnerabilities. Recently, several automated SVC approaches have been proposed to identify the CWE-IDs given a vulnerable program or a vulnerability description using Machine Learning/Deep Learning models. In particular, transformer-based models were leveraged to achieve superior performance through the self-attention mechanism [20, 144]. However, due to the complexity and the nature of the process to collect and label software vulnerabilities wherein some popular vulnerabilities are highly reported while other unpopular ones are rarely reported, the distribution of different software vulnerabilities is highly imbalanced with some highly and rarely occurring CWE-IDs in real-world datasets. For instance, CWE-119 is a common buffer overflow vulner-

---

ability that has 2,127 samples in our experimental dataset, while CWE-94 is a more specific vulnerability about Code Injection that only has 11 samples. Such an imbalanced nature of CWE-IDs leads to a long-tailed label distribution that hinders the learning process of deep learning and transformer-based models, where models could learn too well on the specific CWE-IDs while performing poorly on other CWE-IDs.

Learning from a long-tailed label distribution has been widely studied in computer vision [145–148], notably Focal Loss [145] and Logit Adjustment [148] methods. Although those methods have been demonstrated to couple well with CNNs and vision data, their direct application to transformer-based SVC does not perform satisfactorily. As shown in Table 10, focal loss and logit adjustment do not improve transformer-based SVC in most cases. Additionally, some recent works have proposed to group data by label frequencies and use a balanced group softmax [149] or distillation [150] to learn a better model inspired by knowledge distillation [151] that enables transferring the knowledge from one or more teacher models to a student model. Again, although these approaches work to some extent for vision data and CNNs, they cannot improve transformer-based long-tailed SVC as shown in Table 9 (see the results for BAGS and LFME). We conjecture that grouping by label frequencies helps to mitigate the imbalance in each group. This operation in return creates groups of less similar CWE-IDs, hence making it harder to train a good teacher model for each group.

The goal of this work is to explain the type of vulnerabilities for detected vulnerable functions by classifying CWE-IDs. Thus, we need to address the aforementioned long-tailed label distribution that occurs in the SVC problem. To this end, we propose a hierarchical distillation approach based on the characteristics of vulnerabilities. Specifically, the CWE community has developed hierarchical CWE abstract types [152] to organize complex and diverse CWE-IDs by grouping similar CWE-IDs based on their characteristics. In practice, such categorization is more readable and understandable for security analysts. Moreover, each CWE abstract type becomes a more balanced distribution consisting of similar CWE-IDs as shown in Figure 13, which enables us to learn a better model. Based on this observation, we propose a novel hierarchical distillation approach that is based on the hierarchical grouping of CWE-IDs to overcome the highly imbalanced problem. Particularly, we split a long-tailed label distribution  $Y$  into multiple distributions where each distribution corresponds to a specific CWE abstract type (i.e.,  $Y_{Base}$ ,  $Y_{Category}$ ,  $Y_{Class}$ ,  $Y_{Variant}$ , or  $Y_{Deprecated}$ ) as depicted in Figure 15. Our

---

grouping strategy leads to multiple more balanced label distributions that consist of CWE-IDs with similar characteristics in each group, hence they are simpler for a DL model to learn from. Therefore, for each group corresponding to a CWE abstract type, we train a TextCNN teacher [153] to predict the CWE-IDs in this CWE abstract type. Additionally, to save up the computation and enable the training of the teachers simultaneously, we tie the backbone of the teachers, hence the teachers are only different in the classification heads for predicting the CWE-IDs belonging to their CWE abstract type. Finally, we invoke a transformer-based student to distill from multiple teachers, allowing it to generalize to the entire label distribution. Note that the idea of distilling a transformer from a different CNN teacher has been realized in the DeLT approach [154] for vision data. However, in our approach, we hierarchically distill from multiple TextCNN teachers based on the hierarchy of source code data.

Through an extensive evaluation of our VulExplainer using the Big-Vul dataset [70] consisting of 3,754 vulnerabilities from 348 large-scale open-source software projects spanning from 2002 to 2019, we address the following three research questions:

- **(RQ1) What is the accuracy of our VulExplainer for classifying software vulnerabilities (i.e., CWE-IDs)?**

**Results.** Our VulExplainer method achieves an accuracy of 65%-66% when applying to different transformer-based models such as GraphCodeBERT [155], CodeBERT [40], and CodeGPT [156], which is 5%-29% more accurate than other baseline approaches.

- **(RQ2) Does VulExplainer approach outperform loss-based methods for imbalanced data?**

**Results.** Our approach outperforms the two loss-based methods, achieving the best performance for GraphCodeBERT, CodeBERT, and CodeGPT models.

- **(RQ3) What is the contribution of the components of our VulExplainer?**

**Results.** The ablation study reveals that our hierarchical grouping strategy achieves better performance than the grouping strategy that only focuses on label frequency. Furthermore, our TextCNN teacher models achieve advanced performance while being more efficient (requiring fewer parameters) than transformer-based teachers. Last but not least, the soft distillation (our method) that distills soft knowledge (probability distributions) is better

---

than hard distillation that distills the hard predictions (one-hot predictions) of teacher models.

**Novelty & Contributions.** To the best of our knowledge, the contributions of this work are as follows:

- VulExplainer, a hierarchical software vulnerability distillation approach including two phases aiming to address the imbalanced data issue of SVC: (i) a novel data division approach to split a label distribution into multiple more balanced sub-distributions consisting of more similar CWE-IDs based on the hierarchical nature of CWE-IDs; (ii) a distillation approach based on the self-attention mechanism of transformer models to hierarchically distill knowledge from multiple TextCNN teachers based on the hierarchy of source code data.
- An extensive evaluation by comparing our VulExplainer with seven competitive baseline approaches mentioned in Section 5.4.2.
- An empirical evaluation by comparing our VulExplainer with two advanced loss-based methods (i.e., focal loss and logit adjustment) proposed for the imbalanced data issue.
- A complete ablation study to investigate each step of our VulExplainer approach.

## 5.2 Background & Problem Statement

### 5.2.1 Background

Common Weakness Enumeration (CWE) is a community-developed list of software and hardware weaknesses and vulnerabilities. CWE provides a hierarchical categorization of software vulnerabilities where each CWE-ID determines a vulnerability type and each CWE abstract type determines a group of similar vulnerability types. Such a hierarchical categorization serves as a common language of software vulnerabilities, that helps security analysts identify and understand security flaws existing in software.

Recently, line-level software vulnerability prediction (SVP) methods [1, 75, 94, 108, 157] are proposed to detect vulnerable lines in source code that may save security analysts' efforts to locate vulnerabilities among a large number of codes.

---

For instance, in Figure 12, line-level SVP methods can detect the 9th line as a vulnerable line. However, SVP models can not provide further information such as CWE-IDs to explain the detected vulnerabilities. Therefore, such a lack of explainability concerns could hinder the adoption of SVP methods and make security analysts not fully understand the detected vulnerabilities, leading to more time spent on the security inspection [96, 97, 99–102, 158–161].

Therefore, software vulnerability classification (SVC) methods are proposed to classify vulnerable code into different CWE-IDs and explain the detected vulnerability [19] as shown at the bottom of Figure 12. With the explanation provided by SVC methods, security analysts can understand more about vulnerability prediction by SVP models and efficiently suggest corresponding repair or mitigation strategies.

However, existing SVC methods still encounter an unresolved data imbalance issue. For instance, Das et al. [20] leveraged data augmentation [44] in their experiments but it did not further improve the performance of their transformer model. Wang et al. [19] also experienced the data imbalance issue and only focused on the top 10 frequency CWE-IDs in their experiment to mitigate the data imbalance, which hinders the model from identifying rare vulnerability types. Our experimental dataset is also imbalanced; the samples per class range from 2127 to 10. Some approaches such as logit adjustment [148] and focal loss [145] were proposed in the vision domain to help DL models combat imbalanced label distribution for image classification tasks. Nevertheless, those methods from the vision domain have limited effect on improving transformer models for the SVC task as shown in Table 10.

To combat the imbalanced label distribution, we propose to simplify one complex/imbalanced data distribution into multiple simple/balanced data distributions based on the hierarchical characteristics of software vulnerability data (i.e., CWE abstract types) mentioned early this section. We then leverage a teacher-student knowledge distillation method to benefit from the divided balance distributions as detailed in Section 5.3. The studied data set used in this work was crawled from the CVE database by Fan et al. [70] where the information of CWE-IDs and CWE abstract types have been labeled by human experts. Below, we introduce CWE abstract types, followed by an overview of the teacher-student knowledge distillation method.

---

### 5.2.1.1 CWE Abstract Types

CWE abstract types can be assessed through five dimensions that help characterize weaknesses within the Common Weakness Enumeration (CWE) system. These dimensions include behavior, which refers to observable actions or patterns associated with weakness. The property focuses on specific attributes or qualities related to weaknesses. The technology identifies weaknesses that are specific to certain technologies or platforms. Language pertains to weaknesses that are specific to programming languages. Finally, resource relates to weaknesses that impact system resources.

In this work, we consider five common CWE abstract types, namely Class, Base, Category, Variant, and Deprecated, each providing valuable insights into different aspects of weaknesses within the Common Weakness Enumeration (CWE) system. The class represents weaknesses described in a highly abstract manner, devoid of specific language or technology references. These weaknesses are typically characterized by 1 or 2 dimensions, including behavior, property, and resource. Base weaknesses, on the other hand, are described in an abstract fashion, but with sufficient details to infer specific detection and prevention methods. They offer a level of specificity between Class and Variant weaknesses. Base-level weaknesses encompass 2 or 3 dimensions, incorporating aspects such as behavior, property, technology, language, and resources. Category serves as a structural element that aids users in identifying weaknesses that share common characteristics, facilitating efficient grouping and classification. Deprecate encompasses all deprecated CWE-IDs. Lastly, Variant weaknesses are linked to specific types of products, often associated with particular languages or technologies. These weaknesses offer a higher level of specificity compared to Base weaknesses and are described in terms of 3 to 5 dimensions, encompassing behavior, property, technology, language, and resource. We refer interested readers to the official CWE documentation [152] for concrete examples of these abstract types.

### 5.2.1.2 Knowledge Distillation

By grouping similar CWE-IDs based on CWE abstract types, similar CWE-IDs are grouped and the data becomes more balanced. Consequently, it becomes possible to train a collection of more precise CWE-ID classification models, with each model dedicated to a specific CWE abstract type. However, their scope is limited to identifying CWE-IDs within their respective abstract types for which they

---

were trained. Consequently, to extend the performance of these models to cover all CWE-IDs across various abstract types, we employ knowledge distillation to construct a student model.

Knowledge distillation is a procedure wherein knowledge is transferred from a single model or a set of models, often referred to as teacher models, to a single model known as the student model. The student model leverages the collective knowledge from the specialized models, enabling it to generalize and provide an accurate classification for CWE-IDs across different abstract types. Knowledge distillation can be seen as a type of model compression technique, originally introduced by Bucilua [162].

In particular, we leverage response-based knowledge that focuses on the final output layer of the teacher model. The underlying assumption is that the student model will acquire the capability to emulate the predictions made by the teacher model. To achieve this, we employ a distillation loss function that measures the disparity between the logits of the student and teacher models. By minimizing this loss during training, the student model gradually improves its ability to make predictions that align with those of the teacher model. We illustrate more technical details in Section 5.3.3.

Subsequently, we present an additional challenge encountered in the CWE-ID classification task, supported by a preliminary analysis. We then proceed to outline our proposed approach for mitigating this challenge.

Software Vulnerability Prediction	
1	static sk_sp<SkImage> unPremulSkImageToPremul (SkImage* input) {
2	SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
3	kN32_SkColorType, kPremul_SkAlphaType);
4	RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);
5	if (!dstPixels)
6	return nullptr;
7	return newSkImageFromRaster(
8	info, std::move(dstPixels),
9	static_cast<size_t>(input->width()) * info.bytesPerPixel()); <b>Detected Vulnerable Line</b>
10	}
Software Vulnerability Prediction with Vulnerability Type Explanation	
1	static sk_sp<SkImage> unPremulSkImageToPremul (SkImage* input) {
2	SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
3	kN32_SkColorType, kPremul_SkAlphaType);
4	RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);
5	if (!dstPixels)
6	return nullptr;
7	return newSkImageFromRaster(
8	info, std::move(dstPixels),
9	static_cast<size_t>(input->width()) * info.bytesPerPixel()); <b>Detected Vulnerable Line</b>
10	}
<b>CWE-787 (Out-of-bound Write)</b>	
Typically, this can result in corruption of data, a crash, or code execution. The software may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent write operation then produces undefined or unexpected results.	
<b>Sample Vulnerability Description</b>	
Bad casting in bitmap manipulation in Blink in Google Chrome prior to 55.0.2883.75 for Mac, Windows and Linux, and 55.0.2883.84 for Android allowed a remote attacker to potentially exploit heap corruption via a crafted HTML page.	

Figure 12: A real-world vulnerability example of CWE-787 [163]. The upper part shows the vulnerability prediction generated by line-level SVP models while the lower part presents the same prediction with an extended explanation provided by the SVC approaches to illustrate the detected vulnerability.

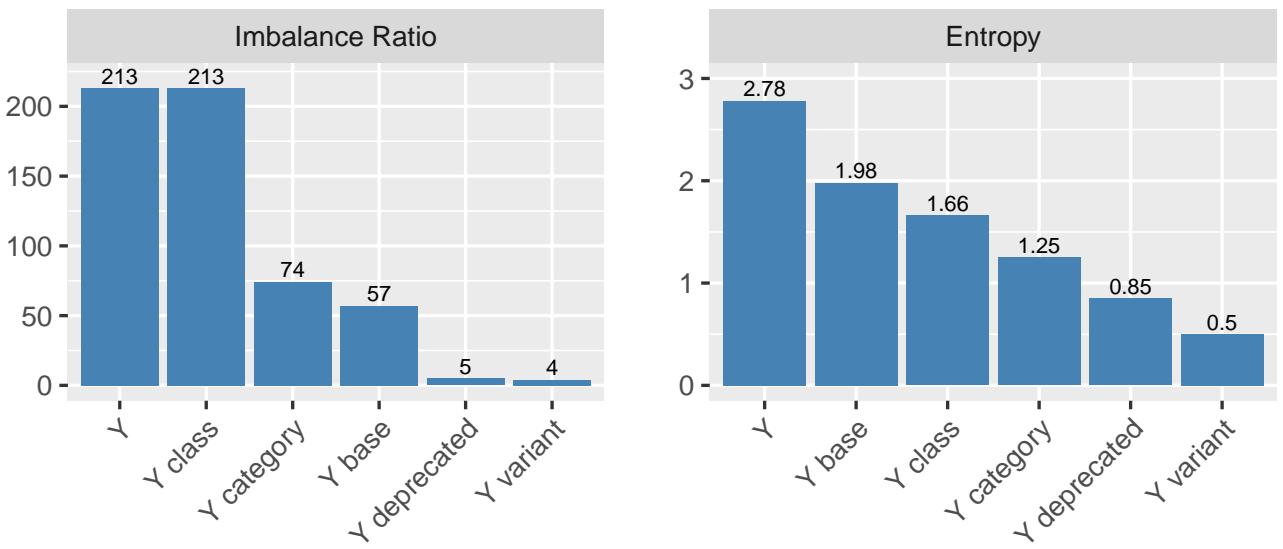


Figure 13: Statistics that measure the imbalance of grouped and ungrouped data distributions.



please classify CWE-ID for the following function:

```
static sk_sp<SkImage> premulSkImageToUnPremul(SkImage* input) {
    SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
                                         kN32_SkColorType, kUnpremul_SkAlphaType);
    RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);
    if (!dstPixels)
        return nullptr;
    return newSkImageFromRaster(
        info, std::move(dstPixels),
        static_cast<size_t>(input->width()) * info.bytesPerPixel());
}
```

[View other drafts](#)



The CWE-ID for the function `premulSkImageToUnPremul()` is **CWE-754**, which is **Improper Input Validation**. This is because the function does not properly validate the `input` parameter. This could lead to an attacker being able to pass in a malicious image that could then be used to exploit a variety of security vulnerabilities.

The function `premulSkImageToUnPremul()` first creates a new `SkImageInfo` object with the `kUnpremul_SkAlphaType` alpha type. The function then calls the `copySkImageData()` function to copy the data from the `input` image into a new `Uint8Array` object. If the `copySkImageData()` function fails, the function returns `nullptr`.

The problem with this function is that it does not properly validate the `input` parameter. This means that an attacker could pass in a malicious image that could then be used to exploit a variety of security vulnerabilities. For example, an attacker could pass in an image that contains malicious code. This malicious code could then be executed when the `premulSkImageToUnPremul()` function is called.

In order to fix the CWE-754 vulnerability in the `premulSkImageToUnPremul()` function, the following changes need to be made:

- The function should validate the `input` parameter to ensure that it is a valid image.
- The function should not call the `copySkImageData()` function if the `input` parameter is not a valid image.

Once these changes have been made, the `premulSkImageToUnPremul()` function will be protected from the CWE-754 vulnerability.



Figure 14: An example of prompting Google BARD to identify CWE-ID based on the input of a C++ function. BARD mistakenly describes CWE-754 (Improper Check for Unusual or Exceptional Conditions) as CWE-20 (Improper Input Validation). Furthermore, the actual vulnerability associated with the `input` function is CWE-787 as described in Figure 12.

---

### 5.2.2 Challenge & Motivation

Prior studies proposed methods for automatically classifying CWE-IDs based on textual vulnerability descriptions [164–166]. These approaches typically involve models learning to recognize CWE-IDs based on keywords acquired during training. However, our objective is to assist software developers in identifying CWE-IDs at the early stages of software development, where vulnerability descriptions may not be available. Consequently, we aim to develop models that can make predictions solely based on the source code input, which poses a more challenging task. This is because the same keywords, such as variable or function names, can be associated with different CWE-IDs depending on the context.

Large language models (LLMs) such as ChatGPT [125] and BARD [126] have demonstrated their ability to conduct code analysis and generate boilerplate code. However, we have reservations regarding the effectiveness of large language models (LLMs) such as ChatGPT and BARD in correctly identifying CWE-IDs given source code input, as this task is domain-specific and related to software security analysis. To validate our suspicions, we conducted a preliminary analysis to explore the capabilities of LLMs in this particular context. The primary objective of this analysis was to investigate whether LLMs can accurately and effectively identify CWE-IDs based on source code input.

#### 5.2.2.1 Preliminary Analysis

For our investigation, we randomly selected 10 samples from our testing dataset, which encompassed the top-10 dangerous CWE-IDs in 2022 [167], including CWE-787, CWE-79, CWE-20, CWE-125, and CWE-416. In our analysis, we utilized ChatGPT and BARD as our examples. To ensure that both ChatGPT and BARD comprehend our task, we initially prompted them to specify the context such as (1) the format of input prompts, (2) the desired generated output, and (3) the programming language under consideration. Additionally, we provided a list of all the CWE-IDs of interest to confine the scope of the analysis.

The findings of our preliminary analysis indicate that both ChatGPT and BARD demonstrate an inability to accurately identify any of the CWE-IDs present in our testing samples. Specifically, ChatGPT's response suggests that additional context is required to specify the CWE-ID for the 9 of the given input code and generate 1 incorrect prediction of the remaining one. On the other hand, BARD provides CWE-ID predictions for each input function, but none of these predic-

---

tions is correct. As depicted in Figure 14, the BARD model erroneously predicted that the function is linked to a CWE-754 Improper Input Validation. However, the CWE-754 is an Improper Check for Unusual or Exceptional Conditions according to the official CWE website [37], and the Improper Input Validation claimed by BARD is assigned as CWE-20. Moreover, the function is actually associated with a CWE-787 Out-of-Bound Write caused by the incorrect variable type assignment (i.e., `size_t`) at the 9th line, as illustrated in Figure 12. The vulnerability could be repaired by changing `size_t` to `unsigned`. These results emphasize the difficulty of the task of identifying vulnerability types based solely on the source code input. It is noteworthy that even though LLMs like ChatGPT and BARD have been trained on extensive datasets comprising hundreds of gigabytes, their performance in this context remains unsatisfactory.

To address this challenge, we utilize language models specifically designed for code, such as CodeBERT, GraphCodeBERT, and CodeGPT. These models have undergone extensive pre-training on millions of source code samples. To the best of our knowledge, we are the first to formally conceptualize the problem of CWE-ID classification using language models for code and rigorously evaluate their performance. While these language models have proven effective in various source code-related tasks like defect detection and program repair, there is no prior evidence demonstrating their suitability for our specific CWE-ID classification task. In order to surpass the direct application of these models, we propose a novel approach that leverages the hierarchical nature of CWE-IDs and incorporates knowledge distillation techniques to address the existing challenge of long-tailed label distribution.

In what follows, we describe how we formulate the problem based on the hierarchical nature of software vulnerability data to mitigate the data imbalance issue.

### 5.2.3 Problem Statement

Assuming we have a source code data set consisting of vulnerable source code functions and the corresponding ground-truth labels representing the vulnerability types (i.e., CWE-ID) of the corresponding vulnerable function. We denote the data set as  $D = \{(F_1, g_1, y_1), \dots, (F_N, g_N, y_N)\}$ , where  $F_i$  is a source code representation,  $g_i$  is its CWE abstract type, and  $y_i$  is its CWE-ID. Each vulnerable function can be considered as a sequence of code statements or a sequence of code tokens. In this work, we consider a vulnerable function  $F_i$  as a sequence of code tokens and denote it as  $F_i = [t_1, \dots, t_n]$  where each function consists of

---

$n$  number of code tokens split by BPE algorithm [50]. Each code token will be embedded into a vector as detailed in Section 5.3.

Moreover, our source code data has a hierarchical organization in which vulnerability labels (i.e., CWE-IDs) and group labels (i.e., CWE abstract types) are completed by software security experts based on the user’s reports. CWE abstract types are higher-level categorizations of CWE-IDs that simplify the categorization of CWE-IDs and define the different abstraction levels that apply to each CWE-ID. Thus, we can group CWE-IDs with similar characteristics based on CWE abstract types. There are some typical characteristics of this dataset. First, the number of classes is large, e.g., we have 44 different CWE-IDs in our experimental dataset. Second, the CWE-ID labels are hierarchically grouped based on the CWE abstract types. Moreover, the CWE-IDs in the same group (i.e., CWE abstract type) are more similar and have the same nature of vulnerabilities. Third, it is a long-tailed dataset for which due to the nature of vulnerabilities, some are more convenient to collect (i.e., frequent CWE-IDs) while some are much harder to collect (i.e., rare CWE-IDs). Fortunately, for the CWE-IDs in the same group, because they share a common nature, their frequencies are more balanced as shown in Figure 13.

We aim to take advantage of the hierarchical nature of vulnerabilities to propose a transformer-based hierarchical distillation framework, mitigate the long-tailed distribution issue efficiently, and benefit from the ability to expose *dark knowledge* from knowledge distillation as detailed in the following section.

### 5.3 Our proposed framework

We now present our main contribution of a novel framework that can effectively assist a transformer model to benefit from our data grouping method and learn better vulnerability classification. As we group the imbalanced label distribution into multiple label distributions based on the CWE abstract types, each sub-distribution consists of similar vulnerabilities and becomes relatively more balanced than the original distribution. Thus, we can learn a DL model more easily on each distribution that achieves promising performance. However, each model only performs well for CWE-IDs in one CWE abstract type. Therefore, we leverage a teacher-student learning framework where a model from each abstract type is treated as a teacher model whose ability will be generalized to a transformer-based student model through a knowledge distillation process. The student model can learn from each teacher, hence performing well for CWE-IDs

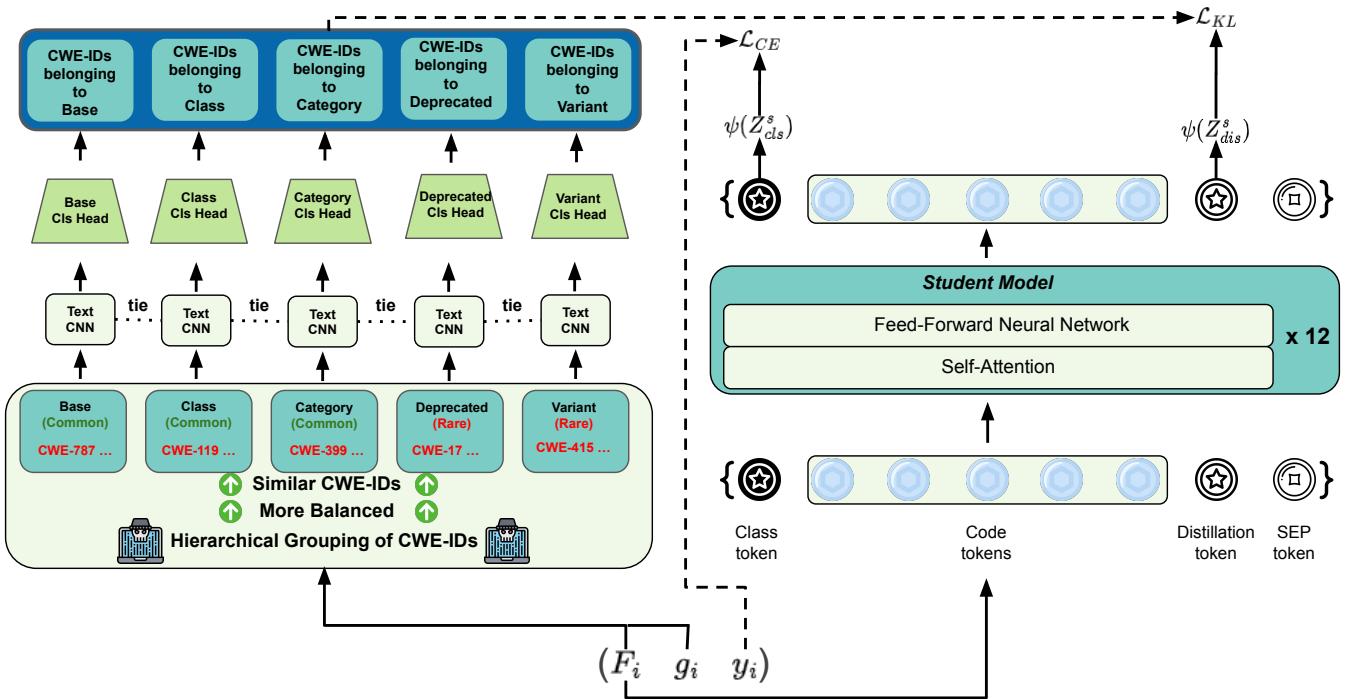


Figure 15: The overview architecture of our VulExplainer during knowledge distillation. The left part describes the inference process of TextCNN teachers. The CWE-IDs are grouped hierarchically based on the CWE abstract types  $g_i$ . A tied TextCNN backbone is connected with multiple classification heads, where each head predicts CWE-IDs belonging to their own CWE abstract type. The right part illustrates the training process of the student model. A distillation token **[dis]** and a **[cls]** token are added to the input  $F_i$  to learn from the knowledge of teachers and ground-truth labels respectively. The representation of  $F_i$  forwards through a 12-layer GraphCodeBERT. Finally, the student relies on a KL loss to learn the representation of **[dis]** by distilling knowledge from predictions of the teacher models, and a CE loss to learn the representation of **[cls]** token from ground-truth labels.

under any abstract type. In general, our framework consists of three sub-steps: (i) grouping source codes into groups with the same CWE abstract types to produce many balance distributions consisting of similar vulnerability types (CWE-IDs), (ii) training multiple TextCNN teachers, each of which aims to predict the CWE-IDs under one specific CWE abstract type, and (iii) hierarchically distill a transformer-based student from multiple diverse teachers trained in the previous step. We term our approach as VulExplainer, a transformer-based hierarchical distillation to explain vulnerabilities by classifying their CWE-IDs, which is overall summarized in Figure 15.

### 5.3.1 Grouping Source Codes into the Groups with the Same CWE Abstract Types

We first split a CWE-ID label distribution  $Y$  into multiple sub-distributions based on CWE abstract type to group similar CWE-IDs. Specifically, given a label dis-

---

tribution  $Y$  consisting of 44 different CWE-IDs, we first split them into 5 groups based on CWE abstract types (i.e.,  $Y_{Base}$ ,  $Y_{Category}$ ,  $Y_{Class}$ ,  $Y_{Variant}$ , and  $Y_{Deprecated}$ ) where each of the sub-distribution consists of multiple CWE-IDs belong to the same CWE abstract type. For instance,  $Y_{base}$  is a distribution that consists of all CWE-IDs in our dataset that belong to the base type. In Figure 13, we provide statistics of the imbalance measure of each grouped label distribution mentioned above and the ungrouped label distribution  $Y$ .

### 5.3.2 Training Multiple TextCNN Teachers

We learn many TextCNN teacher models, each of which predicts CWE-IDs in the same CWE abstract type. By grouping by the CWE abstract types, we achieve the groups consisting of many similar and more balancing CWE-IDs, hence allowing us to train more accurate and better teachers.

Additionally, to encourage training multiple teachers simultaneously and save up the computation overhead, we share the backbone of the TextCNN teachers. On top of this backbone, we build up the classification heads for predicting the CWE-IDs belonging to the same CWE abstract types. For instance, for our dataset, we have 5 classification heads corresponding to  $Y_{Base}$ ,  $Y_{Category}$ ,  $Y_{Class}$ ,  $Y_{Variant}$ , and  $Y_{Deprecated}$ , each of which aims to predict the CWE-IDs in the corresponding CWE abstract type.

So far, we can train good teachers, but they only perform well in the local distribution of an abstract type. In what follows, we present how to employ hierarchical distillation to distill knowledge from multiple teachers for achieving a transformer-based approach that can generalize to predict well entire label distributions.

### 5.3.3 Hierarchical Transformer-based Distillation

Our hierarchical distillation framework is adaptable to any transformer-based source code vulnerability classifier (SVC) with minimal modifications. For clarity, we detail its application using GraphCodeBERT [155].

We leverage GraphCodeBERT which considers the Data Flow Graph (DFG) of source codes. We use the Treesitter<sup>1</sup> package to construct a DFG for each vulnerable function and the GraphCodeBERT implementation [155] to integrate DFG information into a sequence of tokens along with a graph-guided attention

---

<sup>1</sup><https://github.com/tree-sitter/tree-sitter>

---

mask. We refer interested readers to GraphCodeBERT paper [155] for detailed operations of the graph-guided attention mask.

In particular, given a raw input function  $F$ , we tokenize  $F$  into a set of subword tokens and embed each token into  $t_i \in \mathbb{R}^{d=768}$  to obtain a representation as  $t_{1:n} = t_1 \oplus \dots \oplus t_n$  (i.e.,  $\oplus$  is the concatenation operator) using the pre-trained BPE tokenizer and the embedding layer of GraphCodeBERT [155]. We truncate and do padding to let  $n = 512$  tokens.

Two special tokens, **[cls]** and **[sep]**, are added during tokenization where the classification embedding (i.e., **[cls]**) is used to learn the representation of input functions, which will be used by a classification head to classify the CWE-ID. In addition, a **[dis]** token is added before the **[sep]** token to distill knowledge from the teachers. Such distillation embedding (i.e., **[dis]**) allows GraphCodeBERT to learn from the output of the TextCNN teachers, as in a regular distillation, while remaining complementary to the class embedding.

We denote  $H^0$  as the hidden vector output by GraphCodeBERT's embedding layer. The embedding vectors  $H^0$  go through 12 layers of BERT encoder with bidirectional self-attention to learn the representation of source code:  $H^n = E^n(H^{n-1})$ ,  $n \in \{1, \dots, 12\}$ . As proposed by Vaswani et al. [69], each encoder  $E^n$  consists of a multi-head self-attention operation followed by 2 layers of feed-forward neural networks.  $E^n$  takes the  $H^{n-1}$  as input to the self-attention operation to generate self-attention hidden vectors  $A^n$  where LN is a layer normalization and  $Attn$  is the multi-head self-attention mechanism [69]:

$$A^n = LN(Attn(H^{n-1})) + H^{n-1} \quad (6)$$

$A^n$  then goes through 2 layers of feed-forward layers to result in  $H^n$ , the final hidden vector generated by  $E^n$ :

$$H^n = LN(FFN(A^n) + A^n) \quad (7)$$

At the last hidden layer, we possess the token embeddings  $H^{12}$  consisting of the class token embedding  $H_{cls}$  and the distillation token embedding  $H_{dis}$ . We then feed them to two linear layers to work out the class token logits  $Z_{cls}^s$  and the distill token logits  $Z_{dis}^s$ . Similar to Touvron et al. [154], we consider both soft-label and hard-label distillations.

**Soft-label distillation** [151, 168] minimizes the Kullback-Leibler divergence between the softmax of the teacher and the student models. The output of softmax

---

activation is mapped into log space to prevent the underflow issue when computing the KL loss. Let  $Z^t$  be the logits of the teacher model for a given source code  $F$  with the ground-truth label  $y$ . We denote  $\lambda \in [0, 1]$  the tunable coefficient balancing the Kullback–Leibler divergence loss ( $\mathcal{L}_{KL}$ ) and the cross-entropy ( $\mathcal{L}_{CE}$ ) on ground truth labels  $y$  and  $\psi$  the softmax function. The soft distillation objective is as follows:

$$\mathcal{L}_{soft} = (1 - \lambda)\mathcal{L}_{CE}(\psi(Z_{cls}^s), y) + \lambda\mathcal{L}_{KL}(\psi(Z_{dis}^s), \psi(Z^t)) \quad (8)$$

**Hard-label distillation** [154] leverages the one-hot hard decision of the teacher  $y_t$  for a given source code as a true label. The hard-label distillation objective is as follows:

$$\mathcal{L}_{hard} = (1 - \lambda)\mathcal{L}_{CE}(\psi(Z_{cls}^s), y) + \lambda\mathcal{L}_{CE}(\psi(Z_{dis}^s), y_t) \quad (9)$$

**Inference with dual representation.** Given a source code, our VulExplainer relies on representations of both **[cls]** and **[dis]** tokens (i.e.,  $Z_{cls}$  and  $Z_{dis}$ ) to make the final prediction. To this end, we introduce a tunable hyperparameter  $\eta \in [0, 1]$  to tradeoff between the  $H_{cls}$  and  $H_{dis}$  as described in Equation 10 where  $\psi$  is a softmax function.

$$\begin{aligned} \hat{p} &= \eta\psi(Z_{cls}) + (1 - \eta)\psi(Z_{dis}) \\ \hat{y} &= argmax_k \hat{p}_k \end{aligned} \quad (10)$$

## 5.4 Experimental Design and Results

### 5.4.1 Research Questions

The key goal of this work is to evaluate our VulExplainer thoroughly by comparing it with other baseline approaches that focus on the source code classification task and data imbalance issue. We also formulate an ablation study to support the design decision of our VulExplainer approach. Below, we present the motivation for the following three research questions.

**(RQ1) What is the accuracy of our VulExplainer for classifying software vulnerabilities (i.e., CWE-IDs)?** Recently, transformer models have been used to achieve promising performance for SVC approaches [20, 144]. However, as pointed out in Section 5.2.1, those approaches have faced the data imbalance issue of the SVC task and no valid method has been proposed to solve this issue

---

for transformer models. Thus, we formulate this RQ to investigate the accuracy of our VulExplainer which aims to mitigate the data imbalance and further improve the performance of transformer models. We compare our method with seven baselines as described in Section 5.4.2

**(RQ2) Does VulExplainer approach outperform loss-based methods for imbalanced data?** Previous studies of CWE-ID classification tasks have shown that the dataset is unbalanced, with some CWE-IDs occurring significantly more frequently than others [20, 21]. Such a problem can be determined as a long-tailed learning problem which is well-known in the image classification domain where the model has trouble learning to recognize those rare images in the dataset. In particular, the imbalance ratio can be computed as  $N_{max}/N_{min}$  where  $N$  represents the number of samples in each class [169]. Our experiment dataset has an imbalance ratio of 213 where the samples per class range from 2127 to 10, which can be considered an imbalanced dataset compared with previous long-tailed learning studies [169, 170]. Thus, it is important to compare our proposed approach with other methods that help the model learn better about the imbalanced label distribution.

**(RQ3) What is the contribution of the components of our VulExplainer?** In general, our VulExplainer consists of three key steps, (i) split data into multiple subsets based on the CWE abstract types, (ii) train a TextCNN teacher model with multiple classification heads, and (iii) distill via soft distillation to build the final student model. However, little is known about the contributions of each step in our VulExplainer. Thus, we formulate this RQ to conduct an ablation study on the three key steps of our VulExplainer.

#### 5.4.2 Baseline approaches

We compare our method with large pre-trained Transformer-based models for source code, i.e., CodeBERT [40], GraphCodeBERT [155], and CodeGPT [156]. We also include Devign [18] and ReGVD [120], GNN-based models that were designed for software vulnerability detection tasks and achieved competitive results. Furthermore, we include BAGS [149] and LFME [150] that mitigate the imbalance of label distribution by splitting the data into subsets based on label frequencies. The baseline approaches are described as follows:

- **CodeBERT:** The pre-trained model for programming languages proposed by Feng et al. [40]. CodeBERT relies on the same architecture as the BERT

---

model consisting of 12 identical Transformer encoders with bidirectional self-attention. CodeBERT is pre-trained on bimodal data including both programming language and natural language to learn representations for source code and documentation. Specifically, it is pre-trained in 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go) using masked language modelling [63] and replaced token detection [171] objectives.

- **GraphCodeBERT**: The pre-trained code representation with data flow using BERT architecture proposed by Guo et al. [155]. This work is an extended version of CodeBERT and proposes to embed graph structure (i.e., data flow graph) with a sequence of source code tokens. To represent the relation between source code tokens and nodes of the data flow, GraphCodeBERT relies on graph-guided masked attention to define the interaction between code tokens and nodes.
- **CodeGPT**: The GPT-2 architecture pre-trained on programming languages corpus proposed by Lu et al. [156]. CodeGPT has the same model architecture and training objective as GPT-2 [172]. CodeGPT is one of the baseline approaches in the CodeXGLUE benchmark dataset for code understanding and generation [156].
- **Devign**: The GNN-based approach for vulnerability detection proposed by Zhou et al. [18]. This work builds a multi-edged graph from a source code function, then leverages Gated GNNs [173] to update node representations, and finally utilizes a 1-D CNN-based pooling (“Conv”) to make predictions. Note that the authors of Devign [18] do not release the official implementation of Devign. Thus, we reuse the available re-implementation provided by [120] with the same training protocols as the original Devign.
- **ReGVD**: GNN-based method with residual connections among GCN [64] layers for vulnerability detection is proposed in [120]. ReGVD views each source code function as a flat sequence of tokens to build a graph, wherein node features are initialized by only the token embedding layer of a pre-trained programming language (PL) model. ReGVD then leverages GCN layers with pooling layers to return a graph embedding for the source code function, which is utilized to predict final targets.
- **LFME**: Xiang et al. [150] proposed to learn from multiple expert (LFME) models to overcome an imbalanced image dataset. LFME first split the imbalance label distribution into groups where each group is more balanced than the original distribution. It then learned one expert model on each balanced

---

Table 8: The training schemes of teacher and student models in our VulExplainer approach.

Models	Optimizer	Scheduler	LR	Grad Clip	Batch	Seq Len	Epoch	$\lambda$	$\eta$
Teacher	AdamW	Linear	5e-3	1.0	128	512	50	-	-
Student	AdamW	Linear	2e-5	1.0	16	512	50	0.7	0.9

distribution and distilled from all experts to build a final student model. Note that the original LFME framework was designed for the image domain, we followed the original LFME proposal but used a TextCNN to implement the LFME approach. We split the imbalance label distribution into 3 balanced groups with a cardinality threshold set to 100,500 to fit our experimental dataset. Given that our problem domain is source-code related, we use the pre-trained embeddings of the CodeBERT model to map a code sequence into vector space before input to the TextCNN model.

- **BAGS:** A balanced training strategy based on group softmax for object detection, Li et al. [149] first split the imbalance dataset into more balanced groups and proposed to leverage a shared CNN model to extract the representation of images and trained multiple classification heads where each head was trained on a specific group of data. Similar to the implementation of LFME, we use TextCNN to implement the BAGS framework to adapt to our domain. We follow the same split as LFME to split an imbalance label distribution into balanced groups and use the pre-trained CodeBERT embeddings to build the BAGS approach adapted for the source code domain.

#### 5.4.3 Experimental dataset

We use the Big-Vul dataset [70] in our experiments, which is widely used to evaluate DL models for vulnerability detection [1, 32, 108]. Big-Vul is created by crawling from 348 open-source Github projects: the public Common Vulnerabilities and Exposures (CVE) database and CVE-related source code repositories. Big-Vul consists of both vulnerable and non-vulnerable C/C++ functions with 3,754 code vulnerabilities and a total number of 188k functions. To satisfy the vulnerability classification setting, we drop the non-vulnerable functions and obtain 8,636 vulnerable functions with 44 different kinds of CWE-IDs.

#### 5.4.4 Parameter Setting

We split the data into 80% for training, 10% for validation, and 10% for testing. For hyperparameters of baseline approaches, we follow the best setting as specified by the original authors. For our TextCNN teacher model, we use 3 hidden layers,

Table 9: (RQ1 results) The multi-class accuracy of our proposed method and each baseline approach. We present CWE-ID classification results for each group of CWE abstract types and the overall result. Measure using multi-class accuracy shown in percentage. The weighted F1 is also presented in percentage, which considers the class imbalance. A description of each CWE abstract type can be found on the official CWE website [152].

Method	Group By CWE Abstract Types					Overall Acc	Weighted F1
	$Y_{class}$	$Y_{base}$	$Y_{category}$	$Y_{variant}$	$Y_{deprecated}$		
Subsets							
Devign	58.11	44.05	45.10	31.71	38.46	51.16	48.71
ReGVD	60.42	55.95	56.21	36.59	57.69	57.52	56.45
CodeBERT	68.00	58.93	60.78	39.02	57.69	63.19	43.07
CodeGPT	65.26	60.12	64.05	51.22	53.85	63.08	62.30
GraphCodeBERT	63.16	63.69	64.05	41.46	<b>61.54</b>	62.27	62.74
BAGS	63.58	54.17	54.90	51.22	42.31	58.91	57.32
LFME	65.47	58.33	61.44	39.02	50.00	61.57	60.15
GraphCodeBERT <sub>Soft-VulExplainer</sub> (ours)	66.53	<b>64.29</b>	62.75	<b>56.10</b>	57.69	64.58	<b>63.91</b>
CodeBERT <sub>Soft-VulExplainer</sub> (ours)	68.00	<b>64.29</b>	<b>67.97</b>	48.78	<b>61.54</b>	<b>66.09</b>	62.93
CodeGPT <sub>Soft-VulExplainer</sub> (ours)	<b>68.63</b>	62.50	60.78	<b>56.10</b>	57.69	65.05	63.77

the window size of  $W = [3, 4, 5]$  respectively, 100 channels, and a dropout rate of 0.1. For our student model, we use the default model architecture for the GraphCodeBERT model which consists of 12 Transformer encoders with a dropout rate set to 0.1 and a hidden dimension of 768. The training scheme of our teacher and student models is reported in Table 8. We train each model through specific epochs as reported and select the best model based on the highest accuracy on the validation set. We run our experiments on a server with an AMD Ryzen 9 5950X with 16C/32T, 64 GB of RAM, and an NVIDIA RTX3090 GPU with 24GB of RAM.

#### 5.4.5 Experimental Results

**(RQ1) (RQ1) What is the accuracy of our VulExplainer for classifying software vulnerabilities (i.e., CWE-IDs)?**

**Approach.** We conduct experiments on Big-Vul dataset described in Section 5.4.3 and compare our methods with baselines described in Section 5.4.2. In addition, we apply our method on top of the CodeGPT [156] and CodeBERT [40] models given that our method can be used for any transformer-based models.

**Result.** Table 9 presents the experimental results of our VulExplainer methods (applied to GraphCodeBERT, CodeBERT, and CodeGPT) and seven other baseline approaches according to the multi-class accuracy evaluation metric. We provide the classification accuracy of CWE-ID for each abstract type, namely  $Y_{class}$ ,  $Y_{base}$ ,  $Y_{category}$ ,  $Y_{variant}$ , and  $Y_{deprecated}$ , as well as the accuracy for the entire testing dataset (referred to as "Overall").

---

**Our VulExplainer method achieves an accuracy of 65%-66% when applying to different transformer-based models, which is 5%-29% more accurate than other baseline approaches.** Our method outperforms all of the baselines and improves the performance of transformer-based models. In particular, our hierarchical soft distillation approach further improves the performance of GraphCodeBERT ( $62\% \rightarrow 65\%$ ), CodeBERT ( $63\% \rightarrow 66\%$ ), and CodeGPT ( $63\% \rightarrow 65\%$ ).

Furthermore, Figure 13 presents the imbalance measure for ungrouped label distribution (i.e.,  $Y$ ) and each grouped label distribution (i.e.,  $Y_{class}$ ,  $Y_{base}$ ,  $Y_{category}$ ,  $Y_{variant}$ ,  $Y_{deprecated}$ ). Our grouping strategy can reduce both the imbalance ratio (computed as  $N_{max}/N_{min}$  and  $N$  represents the number of samples in each class [169]) and the entropy of the original label distribution  $Y$ . Thus, the grouped label distributions become more balanced and contain less uncertainty, which makes it simpler for a DL model to learn the classification of labels.

Furthermore, our approach demonstrates the highest weighted F1 score of 64% as presented in Table 9. The weighted F1 score addresses class imbalance by assigning greater weight to classes (i.e., CWE-IDs) with larger sample sizes. This approach prevents the evaluation metric from being biased towards the majority class, ensuring a fair evaluation of the model’s performance across all CWE-IDs.

Our experimental results confirm that our grouping strategy can mitigate the imbalance of data while grouping similar vulnerability types, hence accurate teachers can be learned on each distribution. The experimental results confirm the effectiveness of our distillation method in building a generalized transformer student through soft distillation.

#### (RQ2) (RQ2) Does VulExplainer approach outperform loss-based methods for imbalanced data?

**Approach.** Recently, Menon et al. [148] proposed a softmax with a logit translation method which is inspired by the classic logit adjustment based on label frequencies [174–176]. On the other hand, focal loss [145] is a well-known extension of the cross-entropy loss function, commonly applied to overcome imbalance label distribution. It down-weights frequent classes and focuses training on rare classes. We compare our VulExplainer with both logit adjustment (LA) and focal loss (FL) approaches using the dataset described in Section 5.4.3. We set the hyperparameter  $\tau = 1$  for LA and hyperparameter  $\alpha = 0.25$ ,  $\gamma = 2$  for FL as

Table 10: (RQ2 results) The experimental results when comparing our proposed approach with other loss-based methods for the data imbalance problem. We measure the accuracy of CWE-ID classification using multi-class accuracy shown in percentage. The weighted F1 is also presented in percentage, which considers the class imbalance. (FL - Focal Loss, LA - Logit Adjustment).

Methods	Group By CWE Abstract Types						
Subsets	$Y_{class}$	$Y_{base}$	$Y_{category}$	$Y_{variant}$	$Y_{deprecated}$	Overall	Weighted F1
GraphCodeBERT	63.16	63.69	<b>64.05</b>	41.46	61.54	62.27	62.74
GraphCodeBERT <sub>FL</sub>	64.21	62.50	58.17	53.66	57.69	62.04	61.61
GraphCodeBERT <sub>LA</sub>	63.58	<b>64.29</b>	60.78	43.90	<b>65.38</b>	62.27	62.74
GraphCodeBERT <sub>VulExplainer</sub> (ours)	<b>66.53</b>	<b>64.29</b>	62.75	<b>56.10</b>	57.69	<b>64.58</b>	<b>63.91</b>
CodeBERT	68.00	58.93	60.78	39.02	57.69	63.19	43.07
CodeBERT <sub>FL</sub>	64.63	58.93	66.67	41.46	57.69	62.62	44.25
CodeBERT <sub>LA</sub>	<b>68.84</b>	<b>66.67</b>	60.13	<b>53.66</b>	<b>65.38</b>	<b>66.09</b>	51.64
CodeBERT <sub>VulExplainer</sub> (ours)	68.00	64.29	<b>67.97</b>	48.78	61.54	<b>66.09</b>	<b>62.93</b>
CodeGPT	65.26	60.12	64.05	51.22	53.85	63.08	62.30
CodeGPT <sub>FL</sub>	63.16	60.71	<b>64.71</b>	51.22	46.15	61.81	61.06
CodeGPT <sub>LA</sub>	62.32	59.52	59.48	<b>58.54</b>	<b>57.69</b>	61.00	61.47
CodeGPT <sub>VulExplainer</sub> (ours)	<b>68.63</b>	<b>62.50</b>	60.78	56.10	<b>57.69</b>	<b>65.05</b>	<b>63.77</b>

those values yielded the best results reported by the original authors [145, 148].

**Result.** Table 10 presents the experimental results of our VulExplainer approach and two other loss-based approaches according to the multiclass accuracy evaluation metric. Similar to RQ1, We provide the classification accuracy of CWE-ID for each abstract type and the accuracy for the entire testing dataset.

**Our approach achieves the best performance for all of the transformer-based models (i.e., GraphCodeBERT, CodeBERT, and CodeGPT).** In terms of GraphCodeBERT and CodeGPT, both FL and LA approaches do not further improve the original accuracy of GraphCodeBERT’s 62% and CodeGPT’s 63%. In contrast, our VulExplainer improves the performance of GraphCodeBERT (62% → 65%) and CodeGPT (63% → 65%). In terms of CodeBERT, both our approach and LA improve the performance of CodeBERT from 63% to 66% while FL does not improve the performance.

The focal loss reduces the loss contribution of frequent samples and the logit adjustment encourages a large relative margin between logits of rare versus dominant labels. Such approaches may benefit the rare labels, but the performance of the frequent labels may not benefit as much as the rare ones. On the other hand, our method encourages TextCNN teachers to focus on different subsets of data and transfer knowledge to the student model via distillation without adjusting loss weights for rare samples that may not further improve the performance

Table 11: (RQ3 Results) The experimental results of the ablation study to investigate the three key steps in our VulExplainer method. We conduct the ablation study for all three transformer models, i.e., GCB - GraphCodeBERT, CB - CodeBERT, and GPT - CodeGPT. Relative improvement is presented for comparison.

<b>Compare Grouping Methods</b>	<b>Models</b>	<b>Accuracy</b>	<b>Improvement</b>
<b>CWE Grouping (ours)</b>	GCB	<b>64.58</b>	+4%
Label Freq Grouping	GCB	62.27	-
<b>CWE Grouping (ours)</b>	CB	<b>66.09</b>	+5%
Label Freq Grouping	CB	62.73	-
<b>CWE Grouping (ours)</b>	GPT	<b>65.05</b>	+4%
Label Freq Grouping	GPT	62.27	-
<b>Compare Teacher Models</b>	<b>Models</b>	<b>Accuracy</b>	<b>Improvement</b>
<b>TextCNN Teacher (ours)</b>	GCB	<b>64.58</b>	+0.4%
Transformer Teacher	GCB	64.35	-
<b>TextCNN Teacher (ours)</b>	CB	<b>66.09</b>	+6%
Transformer Teacher	CB	62.27	-
<b>TextCNN Teacher (ours)</b>	GPT	<b>65.16</b>	+0.2%
Transformer Teacher	GPT	65.05	-
<b>Compare Distil Methods</b>	<b>Models</b>	<b>Accuracy</b>	<b>Improvement</b>
<b>Soft Distillation (ours)</b>	GCB	<b>64.58</b>	+4%
Hard Distillation	GCB	62.27	-
<b>Soft Distillation (ours)</b>	CB	<b>66.09</b>	+5%
Hard Distillation	CB	62.85	-
<b>Soft Distillation (ours)</b>	GPT	<b>65.05</b>	+0.7%
Hard Distillation	GPT	64.58	-

of those frequent classes.

Last but not least, as presented in Table 10, our approach demonstrates the highest weighted F1 score which further improves the performance of GraphCodeBERT ( $63\% \rightarrow 64\%$ ), CodeBERT ( $43\% \rightarrow 63\%$ ), and CodeGPT ( $62\% \rightarrow 64\%$ ). As discussed in RQ1, the weighted F1 score effectively handles class imbalance by assigning higher weights to CWE-IDs with larger sample sizes. This strategy ensures an unbiased evaluation metric that evaluates the model’s performance across all CWE-IDs fairly. In summary, these results confirm that our approach can achieve promising performance for both rare and common CWE-IDs, which is better than advanced long-tailed learning methods such as focal loss and logit adjustment.

### (RQ3) (RQ3) What is the contribution of the components of our VulExplainer?

**Approach.** Our VulExplainer consists of three steps as mentioned in Section 5.4.1. We conduct an ablation study for each step by comparing our VulEx-

---

plainer with other variants. First, we study the effect of our grouping strategy on the hierarchical nature of CWE-IDs. We compare our data splitting method of grouping by vulnerability types (i.e., CWE abstract types) with grouping by label frequency used by previous approaches that focus on label frequencies to balance the label distribution [149, 150]. Second, we study our choice of teacher models. It is feasible to use transformer-based models as teachers, hence we compare a variant that uses transformer-based teachers to study the effect of having an identical architecture for teachers and the student. Third, we study our choice of distillation methods. We compare the soft distillation 8 with the hard distillation 9 during the training of the student model. We conduct our ablation study for all of the transformer-based models, i.e., GraphCodeBERT, CodeBERT, and CodeGPT. Note that each method including our proposed VulExplainer and variants follows the same distillation framework to ensure fair comparisons.

**Result.** Table 11 presents the experimental results of the ablation study on each step of our VulExplainer approach.

**In terms of grouping methods, our hierarchical grouping strategy based on vulnerability types (our method) achieves the best performance for all models of interest.** The LFME grouping strategy focuses only on label frequencies and some irrelevant CWE-IDs may appear in the same group. In contrast, our hierarchical grouping mitigates the data imbalance while grouping similar CWE-IDs. The result confirms that our grouping strategy is more effective than the strategy focusing on label frequencies for the software vulnerability classification task.

**In terms of teacher models, the results show that distilling knowledge from TextCNN teachers (our method) outperforms distilling from transformer-based teachers for all models of interest.** It has been shown in the previous work from the image domain that using different architectures for teacher and student models yields better distillation [154]. Our experimental results reveal similar results that using different architectures for teacher and student models achieves better accuracy. More importantly, training TextCNN teachers is efficient in terms of time and parameters required where transformer-based models require around 125M parameters while TextCNN teachers only require 40M parameters.

By offering improved computational efficiency and reduced storage demands with fewer model parameters, our approach could enable software vulnerability

---

prediction systems to be implemented on a broader range of platforms, including devices with limited resources. This practicality and accessibility ensure that the benefits of our approach can be extended to various security analysis applications (e.g., Fu et al. [5] proposed, AIBugHunter, a security analysis tool in Visual Studio Code that can predict, classify, and repair software vulnerabilities powered by deep learning models.), making it a valuable solution for real-world vulnerability assessment tasks.

**In terms of distillation methods, the results show that soft distillation (our method) yields better results than hard distillation for all models of interest.** Previous work from the image domain [154] has shown that hard distillation achieves more advanced results than soft distillation for the image classification task. However, in the context of the SVC task, our findings indicate that soft distillation is superior to hard distillation. This discrepancy highlights the importance of considering the unique characteristics and requirements of different tasks when selecting an appropriate distillation method.

Soft distillation outperformed hard distillation in our SVC task for several reasons. Soft distillation preserves the soft probabilities or logits produced by the teacher models, which contain more nuanced information about the relative confidences of different class labels. This allows the student model to learn from the rich and continuous knowledge provided by the teacher models. In addition, soft distillation provides a more forgiving learning signal compared to hard distillation. Hard distillation relies solely on the discrete and often less reliable hard labels produced by the teacher models. In contrast, soft distillation allows the student model to learn from the teacher’s uncertainty and provides a smoother learning signal, making it more resilient to noisy or incorrect labels.

## 5.5 Discussion

In our previous experiment section, we empirically evaluated the performance of our VulExplainer and conducted an ablation study to support our design rationale. However, the question of how our VulExplainer method improves the performance of a transformer model remains unresolved. In this section, we perform an extended analysis of our method to resolve the question. We use the case of CodeBERT to perform our analysis because our VulExplainer approach improves the CodeBERT the most (i.e., 63% → 66%) when comparing with GraphCodeBERT and CodeGPT.

Table 12: (Discussion) The comparison between TextCNN Teacher, CodeBERT<sub>VulExplainer</sub>, and CodeBERT. We measure the accuracy of CWE-ID classification using multi-class accuracy shown in percentage.

Methods	Group By CWE Abstract Types						
	Subsets	$Y_{class}$	$Y_{base}$	$Y_{category}$	$Y_{variant}$	$Y_{deprecated}$	
TextCNN Teacher		<b>72.21</b>	<b>77.38</b>	<b>83.66</b>	<b>95.12</b>	<b>88.46</b>	<b>76.85</b>
CodeBERT <sub>VulExplainer</sub> (ours)		68	64.29	67.97	48.78	61.54	66.09
CodeBERT		68	58.93	60.78	39.02	57.69	63.19
CodeBERT w/o pre-training		60.63	47.02	55.56	34.15	34.62	54.98

### 5.5.1 What is the effect of using a language model pre-trained on code for our CWE-ID classification task?

Our preliminary analysis, presented in Section 5.2.2.1, demonstrates the difficulty of performing CWE-ID classification solely based on the source code input. Even advanced language models such as ChatGPT and BARD were unable to accurately identify CWE-IDs for vulnerable code functions. In order to tackle this challenge, we leverage language models that have been pre-trained on code (e.g., CodeBERT) and examine their performance in addressing our research questions. However, the extent to which pre-training improves performance remains unknown. To investigate this, we trained a model with the same architecture as CodeBERT but with randomly initialized weights (i.e., no pre-training), excluding any pre-training on the extensive CodeSearchNet dataset [68] comprising over 2 million code samples.

As depicted in Table 12, the accuracy of CodeBERT w/o pre-training is measured at 55%. This finding confirms the effectiveness of pre-training on the extensive code corpus conducted by Feng et al. [40], as it introduces an 8% improvement, raising the accuracy to 63%. Additionally, our distillation methods contribute an additional 3% improvement, resulting in a state-of-the-art performance of 66%. It is important to acknowledge that the pre-training step is resource-intensive and demands a large volume of data, with the model being trained on millions of samples using substantial computing resources. In contrast, our approach only requires approximately 7,000 training samples.

### 5.5.2 What is the performance of our TextCNN teacher model?

As shown in the first row of Table 12, our CNNTeacher achieves the best overall performance of 77% on the whole testing set, which is 11% and 14% better than the student model (CodeBERT<sub>VulExplainer</sub>) and CodeBERT respectively. Further-

---

Table 13: (Discussion) Performance analysis of CodeBERT<sub>VulExplainer</sub> and CodeBERT on three different testing subsets. Measure using multi-class accuracy shown in percentage. CNNT - CNNTeacher, CB - CodeBERT, *VulExp* - VulExplainer

Testing Subset	Methods	Accuracy
CNNT correctly predicted	CB <sub>VulExp</sub> (ours)	<b>79.52 (528/664)</b>
	CB	71.23 (473/664)
CB correctly predicted	CB <sub>VulExp</sub> (ours)	91.85 (462/503)
CB wrongly predicted	CB <sub>VulExp</sub> (ours)	30.19 (109/361)

more, each teacher also achieves the best performance on their assigned CWE abstract type. These results confirm the effectiveness of our approach to group the data based on CWE abstract types and train good teacher models.

However, as mentioned in Section 5.3, those teacher models only classify well for CWE-IDs under their own CWE abstract type. Thus, we leverage a hierarchical knowledge distillation process to transfer the prediction ability of teachers to a student model (e.g., CodeBERT) that generalizes to all CWE-IDs. In the following section, we analyze the effects of our hierarchical knowledge distillation method on the CodeBERT model.

### 5.5.3 What are the effects of our hierarchical knowledge distillation method on a transformer student model?

We perform our analysis using three testing subsets. The first subset consists of testing samples that were correctly predicted by the TextCNN teachers. Our goal is to analyze whether our VulExplainer, which is distilled from the TextCNN teachers, can achieve higher accuracy compared to the CodeBERT model, which solely relies on the ground-truth labels for learning. This analysis will provide insights into what extent our knowledge distillation method can improve the extraction from the teacher models. The second subset consists of testing samples that were correctly predicted by the CodeBERT model. Our objective is to investigate the extent to which correct knowledge can be preserved when constructing our VulExplainer using the CodeBERT architecture as a foundation. The third subset comprises testing samples that were incorrectly predicted by the CodeBERT model. Our goal is to examine the extent to which correct knowledge was acquired through our hierarchical distillation process, which corrects the erroneous predictions made by our base model, CodeBERT.

Results are presented in Table 13. In the first subset, correctly predicted by the teacher models, our student model (i.e., CodeBERT<sub>VulExplainer</sub>) outperforms Code-

---

BERT by 8% which has 55 more correct predictions. This observation indicates that the CodeBERT model acquires accurate knowledge throughout our distillation process, thereby aligning with the underlying assumption of knowledge distillation, that the student model will learn to emulate the predictions made by the teacher models.

Within the second subset, where the predictions made by the CodeBERT model were correct, our student model achieves an accuracy of 92%. Out of the total 503 samples in this subset, a notable 462 samples were accurately predicted by our method. This outcome highlights the remarkable retention of correct predictions, as our hierarchical distillation process successfully preserves 92% of the correct predictions made by our base model, CodeBERT. This demonstrates the effectiveness of our approach in maintaining the integrity and reliability of the initial model’s performance.

In the third subset, where the predictions made by the CodeBERT model were wrong, our student model achieves an accuracy of 30%. Out of the total 361 wrongly predicted samples in this subset, our approach successfully rectifies 109 of these erroneous predictions. This result emphasizes the effectiveness of our distillation approach in addressing the incorrect predictions made by the original CodeBERT model. It illustrates how our approach benefits from more accurate teacher models to further enhance the overall performance of our approach.

## 5.6 Related work

**Vulnerability classification** is a task to classify vulnerability labels given source code input. Traditionally, machine learning (ML)-based methods have been proposed to automatically classify vulnerability types based on textual vulnerability descriptions (e.g., the description shown at the bottom of Figure 12) [5, 164–166]. However, these methods often employ traditional data preprocessing approaches like a bag of words (BoW) and TF-IDF, which may not adequately capture the representativeness of text data compared to word embedding techniques used in deep learning. Consequently, the performance of the resulting classification models can be hindered. It is important to acknowledge that during the early stages of software development, such textual descriptions may not be readily available. Security analysts heavily rely on the source code itself to classify and identify vulnerabilities during this critical phase. In this work, our primary objective is to propose an end-to-end method that can serve as a vulnerability classification approach, empowering security analysts to accurately identify vulnerability

---

types. Therefore, we exclusively utilize the source code as the primary feature for making predictions, recognizing its significance in practical scenarios.

Recently, multiple deep learning (DL)-based approaches have been proposed to learn more comprehensive word embeddings for textual data and achieve promising performance. For instance, RNN-based models are proposed to learn the representation of source code sequentially [16, 17, 95, 118]. GNN-based models are proposed to learn from the graph properties (e.g., AST, CFG, and DFG) constructed using static code analysis [14, 18]. Recently, a graph construction based only on code tokens in source code is proposed without using an analyzer, which can also be learned from GNN models [120]. Transformer-based pre-trained language models are commonly adopted to learn the representation through self-attention for both binary [1, 177] and multi-class [20, 144] vulnerability classification. While most proposed techniques focus on binary vulnerability classification, we explore multi-class vulnerability classification that aims to classify the vulnerability type (i.e., CWE-ID) of vulnerable functions.

On the other hand, previous research has explored the utilization of multi-task learning techniques to enhance the performance of vulnerability classification. The underlying premise is that incorporating related tasks, such as predicting the CVSS severity score, can potentially improve the overall performance of the vulnerability classification model. While some studies leverage manual loss weight tuning [46–48] to determine the optimal loss weight for each task, Fu et al. [5] leverages multi-objective optimization to optimally determine the weights between different tasks. While these studies primarily emphasize enhancing the model through multi-task learning, our approach diverges by focusing on single-task learning and addressing the challenge of data imbalance in the CWE-ID classification task.

**Long-tailed learning** is used to learn a model on a highly imbalanced label distribution. Recently, Menon et al. [148] proposed a logit adjustment-based approach to adjust the model’s output logit based on the label frequencies. Focal Loss [145] adjusts the standard cross-entropy loss to reduce the relative loss for well-classified samples and focus more on rare samples that are misclassified during model training. In addition, class-balanced loss [146] and label-distribution-aware margin loss [147] also tackle long-tailed distribution via loss adjustment. Furthermore, Zhang et al. [178] explored prevalent re-sampling methods [170, 179, 180] and data augmentation techniques [181, 182] for effectively addressing long-tailed data challenges within the domain of visual recogni-

---

tion. It is worth noting that our VulExplainer approach can seamlessly integrate with these loss adjustments and re-sampling techniques tailored for long-tailed data distributions.

On the other hand, ensemble-based methods have been proposed to mitigate the long-tailed label distribution. For instance, Zhou et al. [183] proposed to train two neural branches, one learning from original label distribution while the other learning from frequency-reversed label distribution. Li et al. [149] proposed a BAGS approach to split long-tailed distribution into multiple more balanced sub-distributions. BAGS then learns multiple classification heads under a shared feature extractor, where each head is only trained on a specific sub-distribution. Xiang et al. [150] proposed an LFME approach that also splits data into multiple sub-groups to get a smaller class longtailness on each subset. LFME then learns an expert model on each subset and distills knowledge from all experts to build a unified student model.

While previous approaches proposed to divide a label distribution based on label frequencies [149, 150], these data division methods are not optimal for the vulnerability classification task since similar vulnerabilities can appear in different groups. In contrast, we propose a data division strategy based on CWE abstract types that result in more balanced distributions while keeping CWE-IDs with similar characteristics in the same group. Furthermore, we explore knowledge distillation via the self-attention of transformer models using a distillation token.

## 5.7 Threats to Validity

**Threats to the internal validity** relate to the hyperparameter settings when fine-tuning transformer models – GraphCodeBERT, CodeBERT, and CodeGPT. We use the default hyperparameter settings suggested by the original authors of each model and only tune the learning rate as transformer-based models are extremely expensive and consist of millions of parameters. To mitigate this threat, we report the hyperparameter settings in the paper and provide a public replication package [184] to ensure the reproducibility of our experiments.

**Threats to the external validity** relate to the generalizability of our VulExplainer. We conduct our experiment using a large-scale vulnerability dataset (i.e., the Big-Vul dataset [70]) consisting of thousands of vulnerable functions parsed from real-world software projects. Thus, our VulExplainer method is not necessarily to

---

generalize to other datasets. To mitigate this threat, we open-source our experimental dataset and data processing script in our public replication package [184]. However, other vulnerability datasets can be explored in future work.

In theory, our VulExplainer method can be applied to any transformer-based model. Nevertheless, we experiment with GraphCodeBERT, CodeBERT, and CodeGPT, which are the most common pre-trained transformer models for code-related classification tasks. Thus, our VulExplainer method is not necessarily to generalize to other transformer models. To mitigate this threat, we open-source all of the pre-trained models included in our experiments. However, other transformer models can be explored in future work.

## 5.8 Summary

In this chapter, we introduce a new data grouping approach based on CWE abstract types and a teacher-student learning framework to overcome the data imbalance issue of the software vulnerability classification task. By hierarchically grouping an imbalanced label distribution into multiple sub-distributions based on CWE abstract types, the sub-distributions become more balanced, and similar CWE-IDs are distributed in the same group. Thus, we can learn more accurate TextCNN teachers. However, they only perform well in each group respectively. We learn a transformer student model through our hierarchical knowledge distillation framework to generalize the knowledge of teachers to predict all CWE-IDs accurately. Through an extensive evaluation of 8,636 real-world vulnerabilities, our approach outperforms all of the baselines including source code transformer models and long-tailed learning approaches proposed in the vision domain. Last but not least, our approach can be applied to various Transformer-based SVCs without modifying the architecture but adding a special distillation token to the input.

---

## 6 Repair Vulnerabilities with Language Models (LMs)

© 2022 Association for Computing Machinery. This is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive version was published in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022), pp. 935–947, published 09 November 2022, <https://doi.org/10.1145/3540250.3549098>.

---

## 6.1 Introduction

Software vulnerabilities are security flaws, glitches, or weaknesses found in software systems that could be exploited by attackers to undertake malicious activities [106]. In particular, criminal groups may make use of unresolved security vulnerabilities in software to attack and damage a system to steal confidential information or extort assets, resulting in severe economic damage [185]. According to the statistics of the National Vulnerability Database (NVD), the number of vulnerabilities discovered per year is considerably increased five times from 4k+/year in 2011 to 20k+/year in 2021 [186].

Recently, researchers proposed various Artificial Intelligence (AI)-based methods to help under-resourced security analysts better understand the characteristics of vulnerabilities (e.g., vulnerability analysis) [26–29, 187–190] and find vulnerabilities faster (e.g., vulnerability predictions) [1, 14, 17, 18, 90, 91, 99, 108, 118, 161, 191–194]. For example, AI-based vulnerability prediction approaches are proposed to predict if a given function is vulnerable or not at the various granularity levels (e.g., function [14, 17, 18, 118], line [1, 74, 75, 94, 97, 100, 102, 108, 157]) using various types of information (e.g., graph properties [14, 18, 108, 191], semantic [1, 17], syntactic information [14, 18, 108, 191], and mutual information [157]). Such vulnerability prediction approaches only help security analysts to find, detect, and localize the location of vulnerabilities. However, security analysts still have to spend a huge amount of effort to manually fix or repair vulnerabilities [33, 34, 38].

Recently, Chen et al. [23] proposed VRepair [23], an automated vulnerability repair approach that leverages a Transformer-based Neural Machine Translation (NMT). VRepair is proposed to address various challenges of prior work in the area of automated program repairs (e.g., SequenceR [22], an RNN-based NMT model). However, VRepair is still inaccurate due to the following three limitations.

*Limitation ①:* VRepair is trained on a small bug-fix corpus of 23,607 C/C++ functions, which may generate suboptimal vector representations.

*Limitation ②:* VRepair leverages the word-level tokenization and the copy mechanism to handle Out-Of-Vocabulary (OOV), limiting its ability to generate new tokens that never appear in a vulnerable function but are newly introduced in the vulnerability repair.

*Limitation ③:* VRepair leverages a Vanilla Transformer (i.e., a basic Encoder-

---

Decoder Transformer architecture) which uses an absolute positional encoding, limiting the ability of its self-attention mechanism to learn the relative position information of code tokens within the input sequences.

*In this work*, we propose VulRepair, a T5-based Vulnerability Repair approach, aiming to address the aforementioned limitations of VRepair [23]. First, our VulRepair employs a pre-training CodeT5 component from a large codebase (i.e., CodeSearchNet+C/

C# [68, 132] with 8.35 million functions from 8 different Programming Languages) to generate more meaningful vector representation, employs BPE tokenization to handle Out-Of-Vocabulary (OOV) issues, and employs a T5 architecture that considers the relative position information in the self-attention mechanism. Through an extensive evaluation of our VulRepair on CVEFixes [195] and Big-Vul [70] datasets consisting of 8,482 vulnerability fixes from 1,754 large-scale software projects with over 180+ different CWE types spanning from 1999 to 2021, we address the following four research questions:

**(RQ1) What is the accuracy of our VulRepair for generating software vulnerability repairs?**

**Results.** Our VulRepair achieves a Perfect Prediction of 44%, which is 21% more accurate than VRepair [23] and 13% more accurate than CodeBERT [40].

**(RQ2) What is the benefit of using a pre-training component for vulnerability repairs?**

**Results.** Regardless of the model architectures, the PL/NL pre-training corpus improves the percentage of perfect predictions by 30%-38% for vulnerability repair approaches, highlighting the substantial benefits of using the pre-training component for vulnerability repair approaches.

**(RQ3) What is the benefit of using BPE tokenization for vulnerability repairs?**

**Results.** Regardless of the model architectures, the BPE subword tokenization improves the percentage of perfect predictions by 9%-14% for vulnerability repair approaches, highlighting the substantial benefits of using BPE tokenization for vulnerability repair approaches.

---

## (RQ4) What are the contributions of the components of our VulRepair?

**Results.** The pre-training component of our VulRepair is the most important component. Without a proper design of T5 architecture for our VulRepair, the performance can be decreased from 44% to 1%. This finding highlights that designing an NMT-based automated vulnerability repair approach is still a challenging task, which requires a deep understanding of modern Transformer architectures to achieve the highest possible %perfect predictions.

## 6.2 Background & Problem Motivation

Automated Vulnerability Repair (AVR) can be formulated as a Neural Machine Translation (NMT) task [196]. Formally speaking, the objective of an NMT-based AVR model aims to learn the mapping between a vulnerable function  $X_i$  and a vulnerability repair  $Y_i$  (i.e., the repair version of  $X_i$ ). In particular, an NMT-based model is composed of Encoder layers and Decoder layers, where the Encoder takes a sequence of code tokens as input to map a vulnerable function  $X_i = [x_1, \dots, x_n]$  into a fixed-length intermediate hidden state  $H = [h_1, \dots, h_n]$ . Then, the decoder takes the hidden state vector  $H$  as an input to generate the output sequence of tokens  $Y_i = [y_1, \dots, y_m]$ . We note that  $n$  (i.e., the length of the input sequence) and  $m$  (i.e., the length of the output sequence) can be different. To optimize the mapping, the parameters of the NMT-based model are updated using the training dataset with the following equation to maximize the conditional probability:

$$p(Y_i | X_i) = p(y_1, \dots, y_m | x_1, \dots, x_n) = \prod_{i=1}^m p(y_i | H, y_1, \dots, y_{i-1})$$

Previously, Recurrent Neural Networks (RNNs) [197] are widely used as an NMT model for various SE tasks, e.g., RNN-based automated program repair approaches in SequenceR [22] and Tufano et al. [198]. As the length of the source code grows, RNN-based models suffer from long-term dependencies among the input tokens, making the RNN-based models forget some past information for a long sequence of tokens (which is common in source code).

A Transformer-based NMT model, introduced by Google Brain [69], is an Encoder-Decoder architecture with a self-attention mechanism. Unlike RNNs, Transformers do not necessarily process the sequence of tokens in se-

---

quential order. Instead, the self-attention mechanism provides a context vector for any position in the input sequence (i.e., the context vector is used to provide a weight of the tokens that the model should pay attention to), allowing the Transformer-based NMT architecture to be more accurate than RNN-based NMT architecture. Therefore, Chen et al. [23] proposed VRepair [23] (accepted at IEEE TSE 2022), which is a Transformer-based NMT approach for automated vulnerability repair, aiming to address various limitations of RNN-based NMT approaches for automated program repairs [22, 198]. Specifically, VRepair consists of the following three steps:

**Step 1: Code Representation.** For an input vulnerable function, VRepair first leverages a word-level Clang tokenizer with a copy mechanism to tokenize a C function into a sequence of tokens. Then, a word embedding layer is used to generate a vector representation of each token in the sequence to capture the semantic information among the input tokens. Then, an absolute positional encoding layer is used to generate another vector representation of the same sequence that considers positional information among the input tokens. Finally, the two vector representations are added together to form the final code representation which will be used as the input vector of the encoder-decoder model.

**Step 2: An Encoder-Decoder Transformer.** Given the input vectors, VRepair leverages an encoder-decoder Transformer model [69] to generate vulnerability repairs. The representation first goes through a 6-layer Transformer encoder. Then, the output vector of the last Transformer encoder is fed to each of the six Transformer decoders. The output vector of the last decoder then goes through a linear layer with a softmax activation to obtain the final probability distribution of vocabulary used for repair generation.

**Step 3: Beam Search for Repair Generation.** Given the output vector of the Transformer decoder, VRepair then uses the beam search algorithm to generate 50 vulnerable repair candidates. However, there exist three major technical limitations.

**Limitation ①: VRepair is trained on a small bug-fix corpus of 23,607 C/C++ functions, which may generate suboptimal vector representations.** The quality of vector representation heavily relies on the language models of code being used. For the VRepair approach, Chen et al. [23] leverage a transfer learning technique in which VRepair is first pre-trained on a labelled bug-fix corpus to generate a vector representation, which is fed into a Transformer model. Then,

---

the Transformer model is fine-tuned on the vulnerability dataset to perform vulnerability repairs. However, their pre-training data contains a limited number of C/C++ functions. Thus, the pre-trained model of VRepair may not generate the most meaningful vector representation of the source code.

**Limitation ②: VRepair leverages the word-level tokenization and the copy mechanism to handle Out-Of-Vocabulary (OOV), limiting its ability to generate new tokens never appear in a vulnerable function but newly introduced in the vulnerability repair.** Hindle et al. [199] found that source code is far more natural and repetitive than natural languages. Unlike in natural language, software developers are free to create any tokens and can make them arbitrarily complex [83], leading to excessively large vocabulary size. Karampatsis et al. [83] raise concerns that statistical language models of source code often suffer from large vocabularies and Out-Of-Vocabulary (OOV) issues, which could severely affect the performance of the neural language models of source code. Therefore, VRepair overcomes the OOV problem by using word-level tokenization with a copy mechanism [200]. The copy mechanism aims to directly copy/reuse a rare token from the input sequence to the output sequence [200]. However, the word-level tokenization and the copy mechanism cannot reuse tokens that never appear in the vulnerable functions to the vulnerability repairs, limiting its ability to generate new code tokens.

**Limitation ③: VRepair leverages a Vanilla Transformer (i.e., a basic Encoder-Decoder Transformer architecture) which uses an absolute positional encoding, limiting the ability of its self-attention mechanism to learn the relative position information of code tokens within the input sequences.** In Step ①, VRepair leverages an absolute positional encoding layer to capture the position information (i.e., the position ID) of tokens used by the Transformer model in Step ②. This means that VRepair (the vanilla version of Transformer) requires an additional representation of absolute positions (i.e., the position ID in a sequence) to its input tokens to be added to the VRepair model. However, such absolute position information is not efficiently used in the self-attention mechanism [201], limiting the ability of its self-attention mechanism to be fully aware of the position of each token in a sequence. Such limitation could make the VRepair approach pay attention to incorrect code tokens (e.g., parenthesis instead of variable names), leading to inaccurate generation of vulnerability repairs.

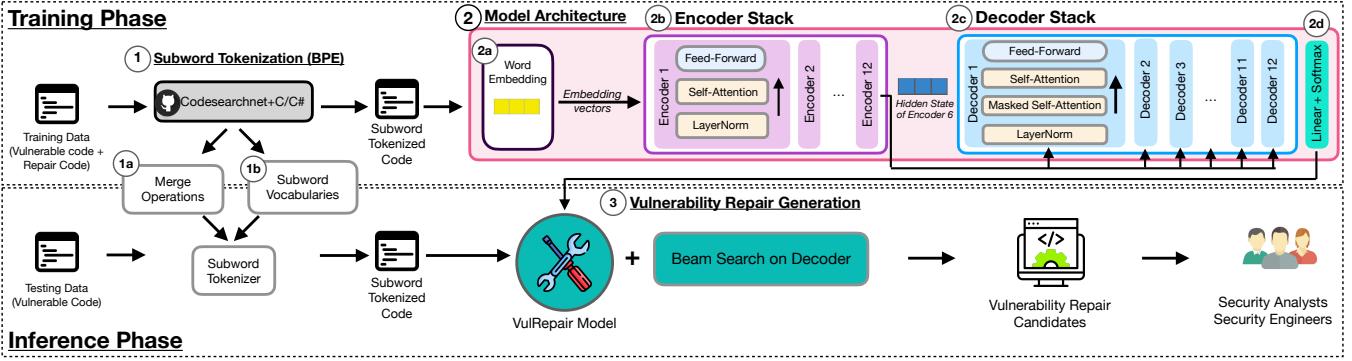


Figure 16: An overview architecture of our VulRepair.

### 6.3 VulRepair: A T5-based Vulnerability Repair Approach

In this section, we present our VulRepair architecture, which is a T5-based automated vulnerability repair approach.

**Overview.** Given a vulnerable function, in step ①, we perform subword tokenization using a Byte-Pairs Encoding (BPE) approach based on a CodeT5 pre-trained language model [132] to produce subword-tokenized functions (i.e., a list of subword code tokens for each function). In Step ②, we build a VulRepair model based on a T5 architecture [202]. For each subword-tokenized function, in Step ②a, VulRepair performs a word embedding to generate an embedding vector for each token and combine it into a matrix. Then, in Step ②b, the matrix is fed into the T5 encoder stack and the output of the last T5 encoder is fed into each T5 decoder in Step ②c. In Step ②d, the output of the T5 decoder stack is fed into a linear layer with softmax activation to generate the probability distribution of the vocabulary. Finally, in Step ③, we leverage beam search on top of the probability distribution of the vocabulary to generate the final candidates as a prediction. Below, we describe the details of each step.

#### 6.3.1 Code Representation

The code representation of each localized vulnerable function in our studied dataset consists of two main steps:

**① BPE Subword Tokenization.** In step ①, we leverage the Byte Pair Encoding (BPE) approach [50] to perform subword-level tokenization, which consists of two main steps. ①a generating merge operations to determine how a word should be split, and ①b applying merge operations based on the subword vocabularies.

---

Specifically, BPE will split all code tokens into sequences of characters and identify the most frequent symbol pair (e.g., the pair of two consecutive characters) that should be merged into a new symbol. BPE is an algorithm that will split rare tokens into meaningful subwords and preserve the common tokens (i.e., will not split the common words into smaller subwords) at the same time. For instance, the function name, *IsValidSize*, will be split into a list of subwords, i.e., ["*IsValid*", "*Size*"]. The rare word *IsValidSize* is split into two common words, *IsValid* and *Size*.

The use of BPE subword tokenization will help reduce the vocabulary size when tokenizing various tokens because it will split rare tokens into multiple subwords instead of adding the full tokens into the vocabulary directly. In this work, we apply a BPE tokenizer that is pre-trained on CodeSearchNet (CSN) [68] and a C/C# corpus extracted by Wang et al. [132]. The tokenizer is pre-trained in eight different programming languages (i.e., Ruby, JavaScript, Go, Python, Java, PHP, C, C#), which is suitable for tokenizing source code. To fit the code generation task, we add "<s>" and "</s>" tokens to represent the beginning of a sequence (BOS) and the end of a sequence (EOS). The "<pad>" token is used to pad the input sequence into the same length if needed. In addition, we add four special tokens (i.e., "<StartLoc>", "<EndLoc>", "<ModStart>", "<ModEnd>") into the vocabulary as an extra vocab ID, so they will not be split into subcomponents during tokenization. The use of pre-trained language models of source code will ensure that the vector representation being generated is more meaningful than VRepair, since it is pre-trained on a larger code corpus (i.e., CodeSearchNet+C/C#). In addition, the use of BPE will ensure that new identifiers that never appear in the vulnerable functions can be generated in the vulnerability repairs [203].

**2a Word Embedding.** Source code consists of multiple tokens where the meaning of each token heavily relies on the context (i.e., surrounding tokens) and the position of each token in a function. Therefore, it is important to capture the code context and its position within the function. The purpose of this step is to generate embedding vectors that capture the semantic meaning of code tokens and their position within a function. For each sub-word tokenized function, in Step 2a, we generate an [1x768] embedding vector for each subword token and combine it into a matrix to represent the meaningful relationship between a given code token and the other code tokens. To capture the semantic meaning of code tokens, we leverage word embedding vectors that are pre-trained on the same corpus as our pre-trained tokenizer discussed above. To capture the position of each

---

code token within the function, our VulRepair leverages relative position embedding which will be computed and added to key matrix and value matrix during self-attention calculation.

### 6.3.2 VulRepair Model Architecture

VulRepair is a T5-based model [202] which starts with an encoder stack and a decoder stack, and ends with a linear layer with softmax activation.

**2b An Encoder Stack.** In Step 2b, a stack of twelve layers of encoder blocks is implemented to derive the encoder hidden state used by the decoders. Similar to original Transformer Encoder [69], each encoder block starts with a Layer Normalization [204] where the activation is only rescaled and no additive bias is applied [202]. Each encoder block consists of two subcomponents: a multi-head self-attention layer with relative position encoding [201] followed by a feed-forward neural network. Each subcomponent (i.e., self-attention and FFNN) in each encoder has a residual connection around it and it is followed by a layer normalization step [69].

The self-attention mechanism [69] computes the relevant scores of each code token using the dot product operation where each token interacts with itself and other tokens once. The self-attention mechanism relies on three main vectors, Query, Key, and Value. The Query is a representation of the current code token used to score against all the other tokens based on their keys stored in the Key vectors. The attention scores of each token are obtained by taking the dot product between all of the Query vectors and Key vectors. The attention scores are then normalized to probabilities using the Softmax function to get the attention weights. Finally, the Value vectors can be updated by taking the dot product between the Value vectors and the attention weight vectors.

Different from VRepair that leverages an absolute positional encoding layer with a word embedding layer to capture the positional information in the input sequence, we use a relative positional encoding to efficiently consider the representations of the relative positions and the distances between tokens within the input sequence (i.e., the relation-aware self-attention mechanism). The self-attention used in our VulRepair is a scaled dot-product self-attention with relative position encoding. The self-attention operation is computed using four matrices, i.e.,  $Q$ ,  $K$ ,  $V$ , and  $P$ . The relative positional information,  $P$ , is supplied to the model as an additional component to the Key matrix and Value matrix as follows:  $\text{Attention}(Q, K, V) =$

---

$\text{softmax}\left(\frac{(Q(P+V))^T}{\sqrt{d_k}}\right)(V + P)$ , where  $P$  is an edge representation for the two inputs in dot-product operation to determine the positional information between tokens. Different from absolute positional encoding that leverages a fixed embedding for each position, the pairwise positional encoding produces a different learned embedding according to the offset between the  $K$  and  $Q$  in the self-attention operation. Therefore, it can effectively capture the relative information among tokens.

To capture richer semantic meanings of the input sequence, we use a multi-head mechanism to realize self-attention, which allows the model to jointly attend to the information from different code representation subspaces at different positions. For  $d$ -dimension  $Q$ ,  $K$ , and  $V$ , we split those vectors into  $h$  heads where each head has  $\frac{d}{h}$ -dimension. After all of the self-attention operation, each head will then be concatenated back again to feed into a fully-connected feed-forward neural network including two linear transformations with a ReLU activation in between. The multi-head mechanism can be summarized by the following equation:  $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$ , where  $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$  and  $W^O$  is used to linearly project to the expected dimension after concatenation.

**2c A Decoder Stack.** In Step 2c, a stack of twelve layers of decoder blocks is implemented to generate the vulnerability repairs based on the hidden states provided by the last encoder block. Similar to original Transformer Decoder [69], each decoder block starts with a Layer Normalization as in the encoder block. Each decoder block consists of three subcomponents: a masked multi-head self-attention layer with relative position encoding, a multi-head encoder-decoder self-attention with relative position encoding, and a feed-forward neural network. Same as the encoder block, each subcomponent in each decoder has a residual connection around it and it is followed by a layer normalization step. The masked self-attention is used during the training phase of our generation models to restrict the model to predict the next token without attending to the later context. Thus, the model can only attend to previous tokens during the generation. This follows the situation of the inference phase where the model will not have any later context and can only attend to previous tokens during the generation.

**2d Linear and Softmax Layer.** The Linear layer is a fully connected neural network that projects the vector produced by the decoder stack into a larger logits vector with the number of cells equals to the number of unique token in the

---

vocabulary. Then, the following Softmax layer transforms the value into a probability distribution that adds up to one, which would be used to generate the final output in Step ③.

### 6.3.3 Vulnerability Repair Generation

Given the output of softmax probabilities, in Step ③, we leverage beam search to select multiple vulnerability repair candidates for an input sequence at each timestep based on a conditional probability. The number of repair candidates relies on a parameter setting called Beam Width  $\beta$ . Specifically, the beam search selects the best  $\beta$  repair candidates with the highest probability using a best-first search strategy at each timestep. The beam search will be terminated when the EOS token (i.e., “</s>”) is generated.

## 6.4 Experimental Design

In this section, we present the motivation for our four research questions, our studied dataset, and our experimental setup.

### 6.4.1 Research Questions

The key goal of this work is to evaluate our VulRepair and compare it with two baseline approaches. Below, we present the motivation for the following four research questions.

**(RQ1) What is the accuracy of our VulRepair for generating software vulnerability repairs?** Recently, Chen et al. [23] proposed VRepair, an NMT-based automated vulnerability repair approach. However, as pointed out in Section 6.2, VRepair has three key limitations, leading to inaccurate vulnerability repair generation. To address these challenges, we propose our VulRepair approach. Thus, we formulate this RQ to investigate the accuracy of VulRepair when comparing to two competitive baseline approaches, i.e., VRepair (a Vanilla Transformer) [23] and CodeBERT (developed by Microsoft Research) [40, 205].

**(RQ2) What is the benefit of using a pre-training component for vulnerability repairs?** The quality of vector representation heavily relies on the language models of code being used. As pointed out in Section 6.2, VRepair is trained on a bug-fix corpus of 23,607 C/C++ functions. However, such a limited amount of data could lead to a suboptimal vector representation of code (i.e., Limitation ①, pre-training). In contrast, our VulRepair leverages a pre-trained language model

---

Table 14: Descriptive statistics of the studied dataset.

	1st Qt.	Median	3rd Qt.	Avg.
#Tokens in Vul. Func.	138	280	593	586
#Repaired Tokens	12	24	48	55
CC. of Vul. Func.	3	8	19	23

CC: Cyclomatic Complexity

of code that is pre-trained on CodeSearchNet+C/C# [68, 132] with 8.35 million functions from 8 different Programming Languages (i.e., Ruby, JavaScript, Go, Python, Java, PHP, C, C#). Thus, we formulate this RQ to investigate the impact of the pre-training component on the accuracy of automated vulnerability repair approaches.

**(RQ3) What is the benefit of using BPE tokenization for vulnerability repairs?** As pointed out in Section 6.2, VRepair leverages word-level tokenization with a copy mechanism. Such an approach may not be able to handle vulnerability repairs that have newly introduced tokens (i.e., Limitation ②, OOV problems). Recently, Karampatsis et al. [83] raise concerns that different tokenization approaches may have an impact on the accuracy of the language models of source code. However, the impact of tokenization approaches has not been investigated in the context of automated vulnerability repairs. Aligning with Karampatsis et al. [83], we formulate this RQ to investigate the impact of the tokenization component on the accuracy of vulnerability repair approaches.

**(RQ4) What are the contributions of the components of our VulRepair?** Our VulRepair involves various key components (i.e., BPE+Pre-Training+T5). However, little is known about what are the contributions of the components of our VulRepair and which component contributes the most to the accuracy of our VulRepair. Thus, we formulate this RQ to conduct an ablation study on the variants of our VulRepair, where each component is altered to others while having the same T5 architecture.

#### 6.4.2 Studied Dataset

In our experiment, we use the vulnerability repairs dataset, CVEFixes [195] and Big-Vul [70], that contains 8,482 vulnerability fixes (a pair of vulnerable C functions and vulnerable repairs). Table 14 presents the descriptive statistics of the experimental dataset.

---

To ensure a fair comparison with VRepair, we strictly follow the replication package provided by Chen et al. [23] to pre-process the experimental dataset. Each input sequence contains a special tag that specifies the CWE type of the sequence. Each vulnerable code snippet in the input sequences is labeled using the special tags “<StartLoc>” and “<EndLoc>”, where “<StartLoc>” indicates the beginning of the vulnerable code snippet, which will be ending with the special tag “<EndLoc>”. For the output labels, each repair code snippet is represented as the special tags “<ModStart>” and “<ModEnd>”, where “<ModStart>” indicates the beginning of the vulnerable repair and non-vulnerable context, which will be ending with the special tag “<ModEnd>”. The main purpose of adding such special tags to the tokenizer is to ensure that such special tags will not be treated as regular code tokens and will not be split by the tokenizer. Similarly, such special tags will help the model to pay attention to the areas of vulnerable code snippets and the vulnerability repair.

#### 6.4.3 Experimental Setup

**Split.** Same as Chen et al. [23], we split the experimental dataset into 70% of training, 10% of validation, and 20% of testing data.

**Model Implementation of Vulnerability Repair.** We build our VulRepair approach on top of two deep-learning Python libraries, i.e., Transformers [71] and PyTorch [72]. The Transformers library provides API access to the transformer-based model architectures and the pre-trained weight, while the PyTorch library supports the computation during the training process (e.g., back-propagation and parameter optimization).

**Model Training of our VulRepair.** We obtain the CodeT5 tokenizer and model pre-trained by Wang et al. [132] from the API of the Transformers library. We use our training dataset to fine-tune the pre-trained model to get suitable weights for our vulnerability repair task. The model is fine-tuned on an NVIDIA RTX 3090 graphic card and the training time is around 5 hours.

We use the cross-entropy loss ( $H(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x)$ ) to update the model and optimize between each position in the predicted sequence and each position in the ground-truth sequence where  $\mathcal{X}$  is the set of classes (i.e.,  $X$  is a set of possible tokens generated by the BPE tokenizer for our approach and any NMT-based models like VRepair),  $p$  is the ground truth probability distribution, and  $q$  is the predicted probability distribution. To obtain the best-fine-tuned

---

weights, we use the validation set to monitor the training process by epoch, and the best model is selected based on the optimal loss value against the validation set (not the testing set).

**Hyper-Parameter Settings for Fine-Tuning.** For the model architecture of our VulRepair approach, we use the default setting of CodeT5 [132], i.e., 12 Transformer Encoder blocks, 12 Transformer Decoder blocks, 768 hidden sizes, and 12 attention heads. During fine-tuning, the learning rate is set to  $2e^{-5}$  with a linear schedule where the learning rate decays linearly throughout the training process. We use backpropagation with AdamW optimizer [73] which is widely adopted to fine-tune Transformer-based models to update the model and minimize the loss function.

## 6.5 Experimental Results

**(RQ1) What is the accuracy of our VulRepair for generating software vulnerability repairs?**

**Approach.** To answer this RQ, we evaluate the accuracy of vulnerability repair approaches using the percentage of perfect predictions (%Perfect Predictions). The %Perfect Predictions measures the percentage of vulnerable functions that an approach can correctly generate a vulnerable repair that is exactly matched with ground-truth data (i.e., a human-written vulnerable repair). Then, we compare %Perfect Predictions of our VulRepair with the two baseline approaches as follows:

1. VRepair [23] uses a vanilla Encoder-Decoder Transformer model [69] for vulnerability repairs. VRepair is first trained on a labeled bug-fixing dataset and fine-tuned on a vulnerability dataset to generate vulnerability repairs;
2. CodeBERT [40] is an Encoder-only Transformer-based model that is pre-trained on a large codebase called CodeSearchNet [68], developed by Microsoft Research. CodeBERT consists of 12 Transformer Encoder Blocks plus six layers of Transformer Decoder for generation tasks. Mashhadi and Hemmati [205] leverage CodeBERT for automated program repair of Java bugs and present a substantial improvement over RNN-based models.

For each approach, we use the same experimental setup as Chen et al. [23] to evaluate the accuracy of our approach and the baseline approaches. Specifically, we leverage a beam width of 50 to generate 50 repair candidates for each

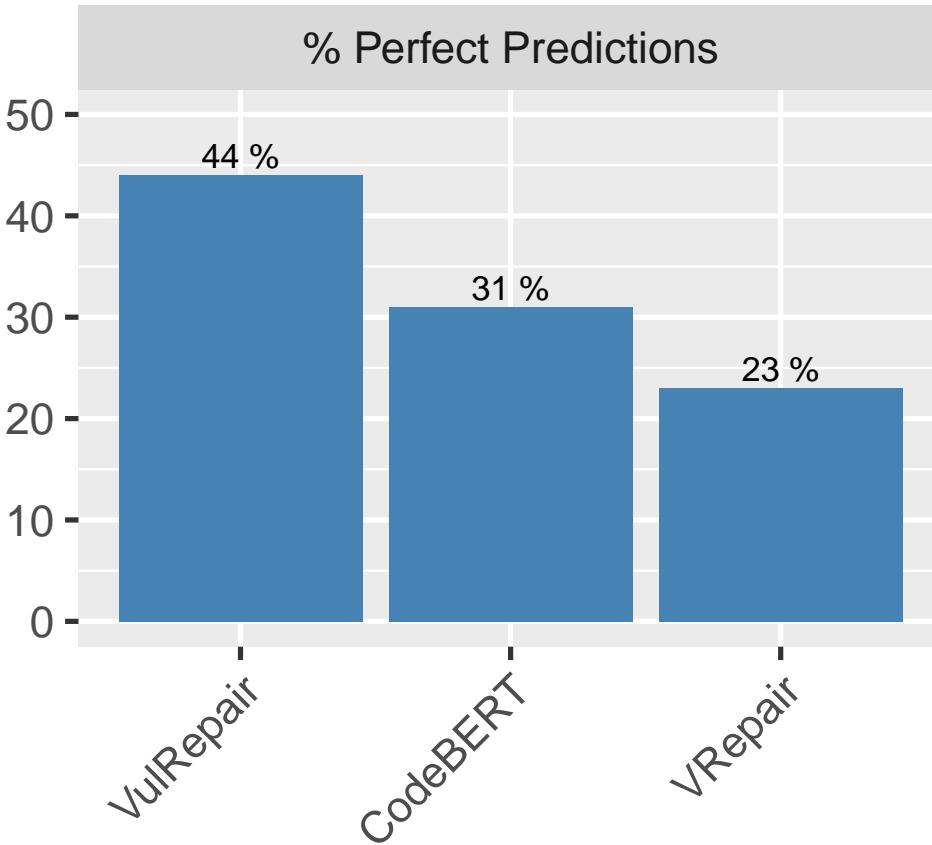


Figure 17: (RQ1) The experimental results of our VulRepair and the two baseline comparisons for vulnerability repairs. (↗) Higher % Perfect Predictions = Better.

vulnerable function in the testing dataset. Therefore, the %Perfect Predictions can be computed as the total number of correct predictions divided by the total number of functions in the testing dataset.

**Result.** Figure 17 presents the experimental results of our VulRepair and the two baseline approaches according to our evaluation measures (i.e., %Perfect Predictions).

**Our VulRepair achieves a Perfect Prediction of 44%, which is 13%-21% more accurate than the baseline approaches.** Figure 17 shows that CodeBERT achieves a Perfect Prediction of 31%, while VRepair achieves a Perfect Prediction of 23%, indicating that VulRepair is 13% ( $44\% - 31\%$ ) and 21% ( $44\% - 23\%$ ) better than CodeBERT and VRepair respectively. The 91% accuracy improvement over the VRepair [23] has to do with different improvements of our VulRepair to address various limitations of VRepair [23], i.e., using a pre-training model from a larger codebase (i.e., CodeSearchNet+C/C# [68, 132] with 8.35 million functions from 8 different Programming Languages) to generate more meaningful vector representation (Limitation ①), using BPE tokenization to han-

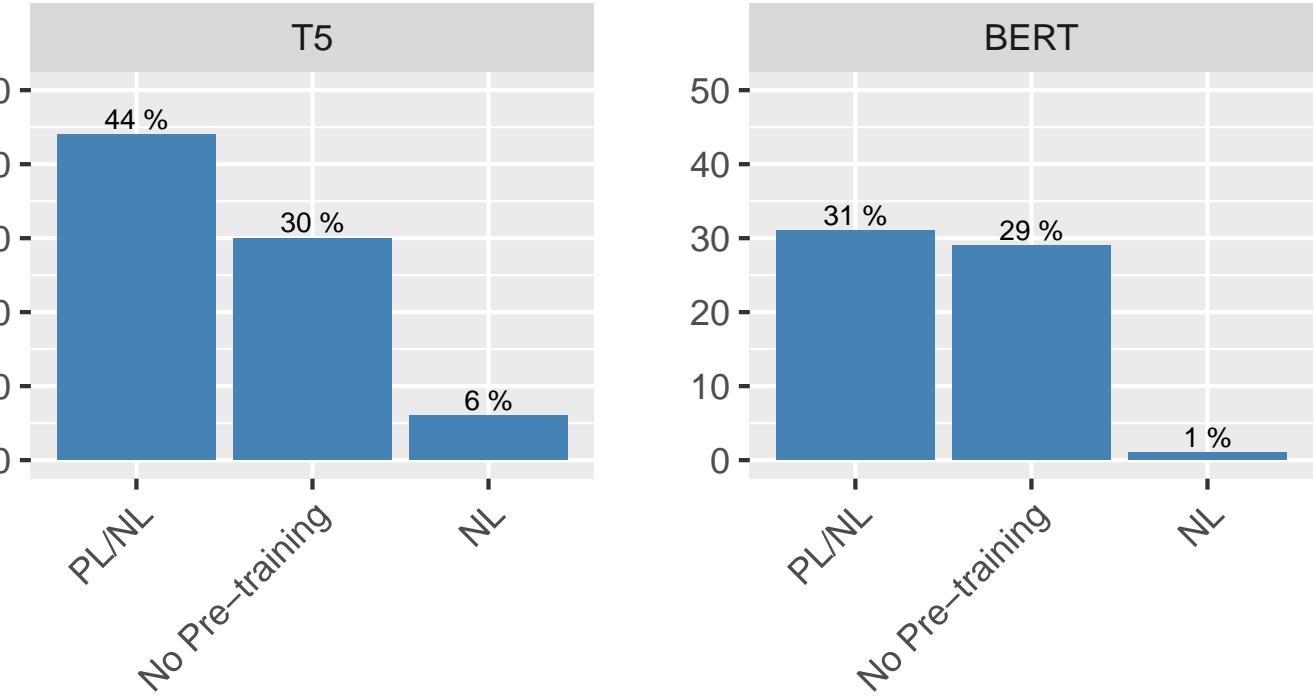


Figure 18: (RQ2) The experimental results of the ablation study with six different models. (↗) Higher % Perfect Predictions = Better.

dle Out-Of-Vocabulary (OOV) issues (Limitation ②), and using T5 architecture that considers the relative position information in the self-attention mechanism (Limitation ③). In contrast, VRepair [23] leverages word-level tokenization with a pre-trained model on a bug-fix corpus of 23,607 C/C++ functions and a vanilla Transformer (i.e., a default Transformer architecture).

#### (RQ2) What is the benefit of using a pre-training component for vulnerability repairs?

**Approach.** To answer this RQ, we aim to investigate the impact of the pre-training corpus component for vulnerability repairs. We note that VRepair leverages a vanilla Transformer, which performs model training in supervised learning (i.e., regular model training with label datasets). Thus, the model training process of VRepair is different from modern Transformer architectures like BERT and T5. In contrast, our VulRepair leverages T5 which performs a pre-training in unsupervised learning (i.e., labels are not required), i.e., the process of training a model for a general task (e.g., next word prediction) with a very large dataset. Thus, we only conduct an experiment of different pre-training corpus components with the T5 and BERT architectures. Specifically, we extend our experiment to systematically evaluate the following six variants of DL-based vulnerability repair approaches, i.e., 3 pre-training corpora (PL/NL, NL, No Pre-training)  $\times$  2 model architectures (T5, BERT).

- 
- **T5 + PL/NL (our VulRepair):** A T5 architecture that is pre-trained on both Programming Languages and Natural Language (PL/NL).
  - **T5 + NL:** The original T5 architecture that is pre-trained on NL only (e.g., Internet webpages).
  - **T5:** A T5 architecture without pre-training.
  - **BERT + PL/NL (original CodeBERT):** A BERT architecture that is pre-trained on PL/NL.
  - **BERT + NL:** The original BERT architecture that is pre-trained on NL only (e.g., Internet webpages).
  - **BERT:** A BERT architecture without pre-training.

Similarly, we evaluate the accuracy of these variants using the same evaluation measure (i.e., % Perfect Predictions).

**Result.** Figure 18 presents the experimental results of the benefits of using a large pre-training corpus for vulnerability repairs.

**Regardless of the model architectures, the PL/NL-based pre-training corpus improves the percentage of perfect predictions by 30%-38% for vulnerability repair approaches.** Figure 18 shows that the PL/NL pre-training corpus improves %Perfect Predictions by 30% (31% – 1%) for the BERT architecture and 38% (44% – 6%) for the T5 architecture when they are trained on the NL-only corpus (i.e., the original T5/BERT architecture provided by the Transformer library). This finding highlights the substantial benefits of the pre-training process on the larger codebase, i.e., CodeSearchNet+C/C# [68, 132] with 8.35 million functions from 8 different Programming Languages (i.e., Ruby, JavaScript, Go, Python, Java, PHP, C, C#) for vulnerability repairs. Nevertheless, when using the same PL/NL pre-training corpus, **the T5 architecture employed by our VulRepair still outperforms the BERT architecture.** Figure 18 shows that, when using the same PL/NL pre-training corpus, the T5 architecture outperforms the BERT architecture by 13% (44% – 31%), highlighting the substantial benefits of the Text-to-Text Encoder-Decoder Transformer (T5) for vulnerability repairs (i.e., code→code generation) over the Encoder-only Transformer (like BERT), which could be more suitable for other SE tasks (code→text) like code summarization.

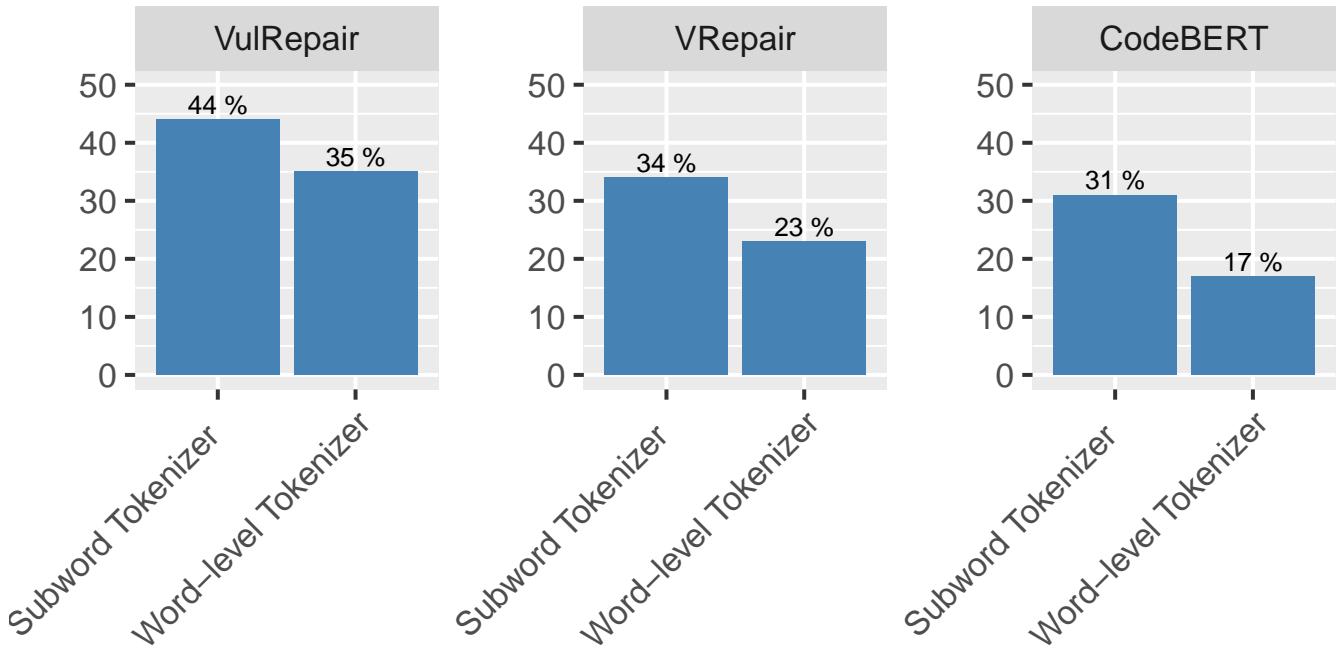


Figure 19: (RQ3) The experimental results of various approaches with different tokenization techniques for vulnerability repairs. ( $\nearrow$ ) Higher %Perfect Predictions = Better.

#### (RQ3) What is the benefit of using BPE tokenization for vulnerability repairs?

**Approach.** To answer this RQ, we aim to investigate the impact of the tokenization component for vulnerability repairs. To do so, for each approach, we alter only the tokenizer for each of the vulnerability repair approaches. Specifically, we extend our experiment to systematically evaluate the following six variants of DL-based vulnerability repair approaches, i.e., 2 tokenizers (subword tokenizer and word-level tokenizer)  $\times$  3 model architectures (VulRepair, CodeBERT, VRepair).

- **Subword Tokenizer + CodeT5 (our VulRepair):** BPE tokenizer with a CodeT5 model.
- **Word-level Tokenizer + CodeT5:** Word-level tokenizer with a CodeT5.
- **Subword Tokenizer + Vanilla Transformer:** BPE tokenizer with a Encoder-Decoder Transformer model.
- **Word-level Tokenizer + Vanilla Transformer (VRepair):** Word-level tokenizer with an Encoder-Decoder Transformer model and a copy mechanism for the OOV problem.

- 
- **Subword Tokenizer + CodeBERT (Original CodeBERT):** BPE tokenizer with a CodeBERT model.
  - **Word-level Tokenizer + CodeBERT:** Word-level tokenizer with a CodeBERT model.

Finally, we evaluate the accuracy of these variants using the same evaluation measure (i.e., % Perfect Predictions).

**Result.** Figure 19 presents the experimental results of the benefits of using BPE tokenization for vulnerability repairs.

**Regardless of the model architectures, the BPE subword tokenization improves the percentage of perfect predictions by 9%-14% for vulnerability repair approaches.** Figure 19 shows that the use of BPE subword tokenization improves %Perfect Predictions by 9% (44% – 35%) for VulRepair, 11% (34% – 23%) for VRepair, and 14% (31% – 17%) for CodeBERT. These results highlight the substantial benefits of using BPE tokenization for vulnerability repair approaches, addressing the Limitation ② of VRepair [23]. Nevertheless, our approach (BPE+CodeT5) is still top-performing for vulnerability repairs, which is 21% (44% – 23%) better than VRepair.

#### (RQ4) What are the contributions of the components of our VulRepair?

**Approach.** To answer this RQ, we aim to investigate the contribution of each component within VulRepair (Pre-training+BPE+T5) by examining the model accuracy of our VulRepair when each component is varied, comparing with a basic T5 (No Pre-training+Word-level+T5). Specifically, we extend our experiment to systematically evaluate the following four variants of T5-based vulnerability repair approaches, i.e., 2 pre-training strategies (pre-training, no pre-training)  $\times$  2 tokenizers (subword-level, word-level):

- **Pre-training + BPE + T5 (VulRepair):** A pre-trained T5 model with a BPE tokenizer.
- **Pre-training + Word-level + T5:** A pre-trained T5 model with a word-level tokenizer.
- **No Pre-training + BPE + T5:** A non-pre-trained T5 model with a BPE tokenizer.

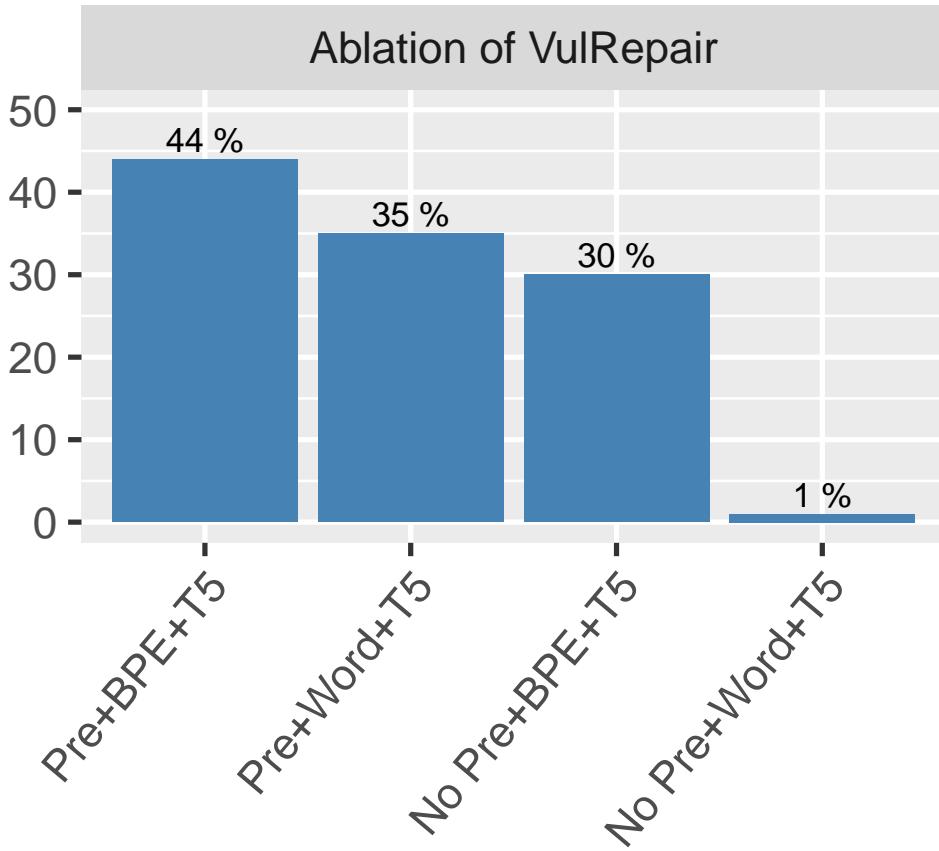


Figure 20: (RQ4) The ablation study result of VulRepair. ( $\nearrow$ ) Higher %Perfect Predictions = Better.

- **No Pre-training + Word-level + T5:** A non-pre-trained T5 model with a word-level tokenizer.

Similarly, we evaluate the accuracy of these variants using the same evaluation measure (i.e., % Perfect Predictions).

**Result.** Figure 20 presents the ablation study to evaluate the contributions of the components of our VulRepair.

**The pre-training component of our VulRepair is the most important component.** Within our VulRepair, the pre-training component contributes to 14% of the %Perfect Prediction. When comparing Pre+BPE+T5 and NoPre+BPE+T5 where the Pre-training component is eliminated, we observe a performance decrease from 44% to 30%, accounting for 14%. Within our VulRepair, the BPE component contributes to 9% of the %Perfect Prediction. When comparing Pre+BPE+T5 and Pre+Word+T5 where the BPE component is changed to the word level, we observe a performance decrease from 44% to 35%, accounting for 9%. Nev-

---

ertheless, without a proper design of T5 architecture for our VulRepair, the performance decreased from 44% to 1%. This finding highlights that designing an NMT-based automated vulnerability repair approach is a challenging task, which requires a deep understanding of modern Transformer architectures to achieve the highest possible percentage of perfect predictions.

## 6.6 Discussion

In this section, we perform additional analysis to further discuss the results of our VulRepair approach and provide some recommendations for future researchers.

### 6.6.1 What Types of CWEs that Our VulRepair Can Correctly Repair?

CWE (Common Weakness Enumeration) is a list of vulnerability weaknesses in software that can lead to security issues with its severity of risk, providing guidance to organizations and security analysts to best secure their software systems. To better understand the significance of our VulRepair on the practical usage scenarios, we perform a further investigation to better understand the Top-10 CWEs that can be correctly repaired by our VulRepair and the Top-10 most dangerous CWEs.<sup>2</sup> The Top-10 most dangerous CWEs are the most common and impactful issues experienced over the previous two calendar years. Such weaknesses are dangerous because they are often easy to find, exploit, and can allow adversaries to completely take over a system, steal data, or prevent an application from working.

**Our VulRepair can correctly repair 38% of the vulnerable functions affected by the Top-10 most dangerous CWEs (see Table 15).** We find that VulRepair achieves %Perfect Predictions as much as 53% for CWE-416 (Use After Free), 45% for CWE-20 (Improper Input Validation), and 33% for CWE-78 (OS Command Injection). Figure 21 shows that our VulRepair achieves 100% perfect predictions for the following CWEs (i.e., CWE-755, CWE-706, CWE-326, CWE-667, CWE-369, CWE-77, CWE-388, CWE-436, CWE-191). However, %Perfect Predictions still vary from 0% to 100%, depending on the CWE types in the dataset. That means the CWE types that achieve perfect predictions may not necessarily be the majority of CWEs in the dataset. Thus, we further analyze the % perfect predictions according to the majority of the CWEs in the dataset. We find that there exist many CWE types that our VulRepair cannot correctly generate vulnerability repairs (e.g., CWE-639, CWE-354, CWE-522) (see Figure 21). We

---

<sup>2</sup>[https://cwe.mitre.org/top25/archive/2021/2021\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html)

---

Table 15: (Discussion) The % Perfect Predictions of our VulRepair for the Top-10 Most Dangerous CWEs.

Rank	CWE Type	Name	%PP	Proportion
1	CWE-787	Out-of-bounds Write	30%	16/53
2	CWE-79	Cross-site Scripting	0%	0/1
3	CWE-125	Out-of-bounds Read	32%	54/170
4	CWE-20	Improper Input Validation	45%	68/152
5	CWE-78	OS Command Injection	33%	1/3
6	CWE-89	SQL Injection	20%	1/5
7	CWE-416	Use After Free	53%	29/55
8	CWE-22	Path Traversal	25%	2/8
9	CWE-352	Cross-Site Request Forgery	0%	0/2
10	CWE-434	Dangerous File Type	-	-
		TOTAL	38%	171/449

find that these CWE types are rare in our datasets with less than 5 functions, indicating that our VulRepair still cannot accurately repair for some types of rare vulnerabilities. Thus, *future researchers should further explore techniques to handle rare vulnerabilities (i.e., the low proportion of vulnerabilities in the training and testing dataset)*.

#### 6.6.2 How Do the Function Lengths and Repair Lengths Impact the Accuracy of Our VulRepair?

Although our VulRepair can correctly generate a considerable number of vulnerability repairs ( $\frac{745}{1,706}$ ) for various types of CWEs, there is a large number of 961 vulnerable functions that cannot be correctly generated. Thus, we perform a further investigation to analyze the accuracy of our VulRepair with respect to the repair length and the function length.

**The accuracy of our VulRepair depends on the size of the vulnerable functions and its difficulty to repair.** Table 16 shows that our VulRepair is most accurate for vulnerable functions that have less than 500 tokens and less than 20 repair tokens. Our VulRepair achieves the %Perfect Prediction of 64%-77%

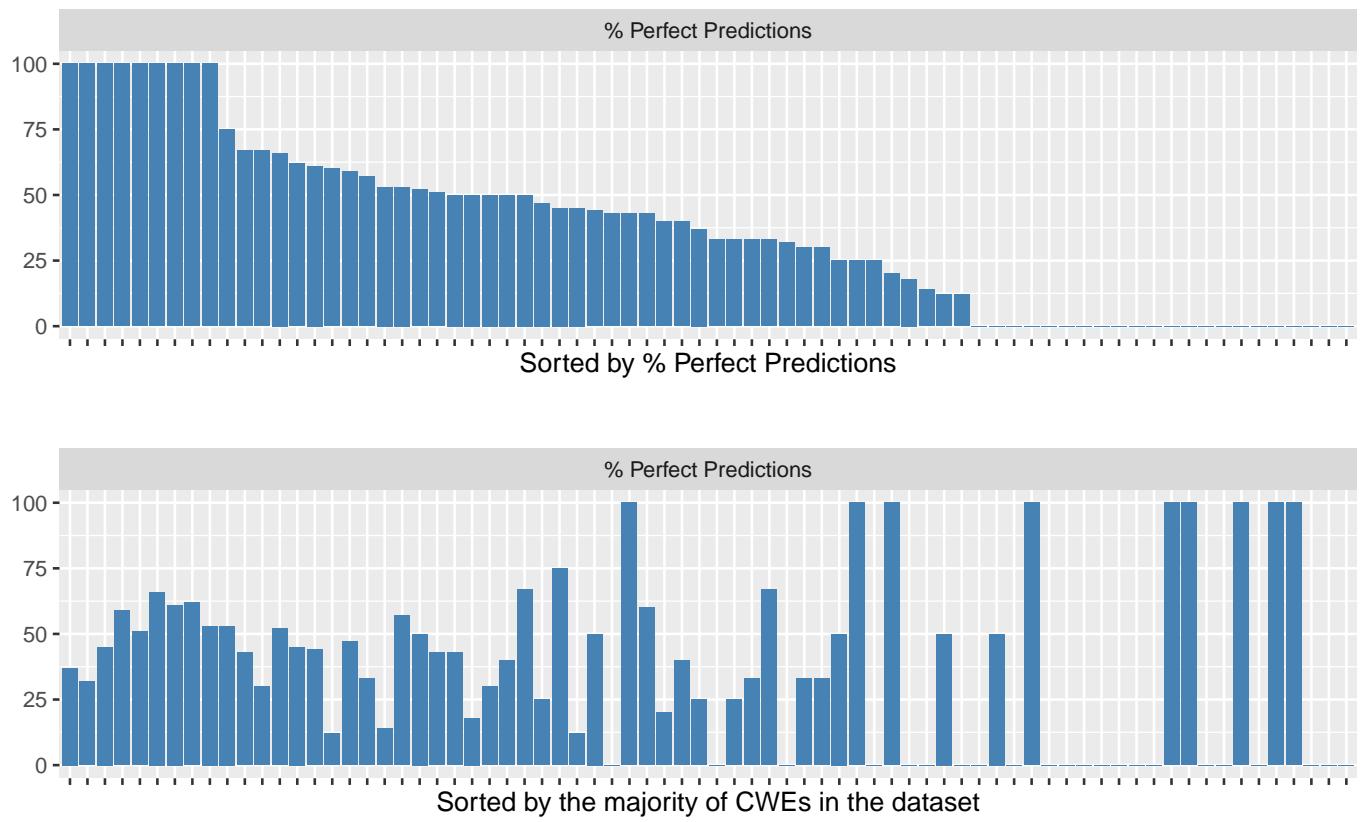


Figure 21: (Discussion) The %Perfect Predictions (y-axis) of our VulRepair according to each type of CWE (x-axis, sorted by % perfect predictions and sorted by the majority of CWEs in the dataset). Detailed statistics can be found in Appendix.

for vulnerable functions with less than 500 tokens, but the %Perfect Prediction is substantially decreased to 32% for vulnerable functions with greater than 500 tokens. The performance decrease for large functions (500+ tokens) has to do with the window size of the T5 architecture (i.e., limited to 512 tokens). For any vulnerable functions with greater than 512 tokens, such extra tokens will be truncated and will not be processed and learned by the models, leading to a negative impact on the accuracy of our VulRepair. Thus, *future researchers should further explore techniques that can handle larger functions (i.e., the functions with more than 512 tokens)*.

In addition, the repair difficulty (measured by #repair tokens in the vulnerability repair) is also impacting the accuracy of our VulRepair. We find that our VulRepair achieves the %Perfect Prediction of 63%-77% for vulnerable repairs with less than 10 repair tokens, but the %Perfect Prediction is substantially decreased to below 60% for vulnerable repairs with greater than 10 repair tokens. Thus, *future researchers should further explore techniques that can handle difficult repairs (i.e., repairs with more than 20 repair tokens)*.

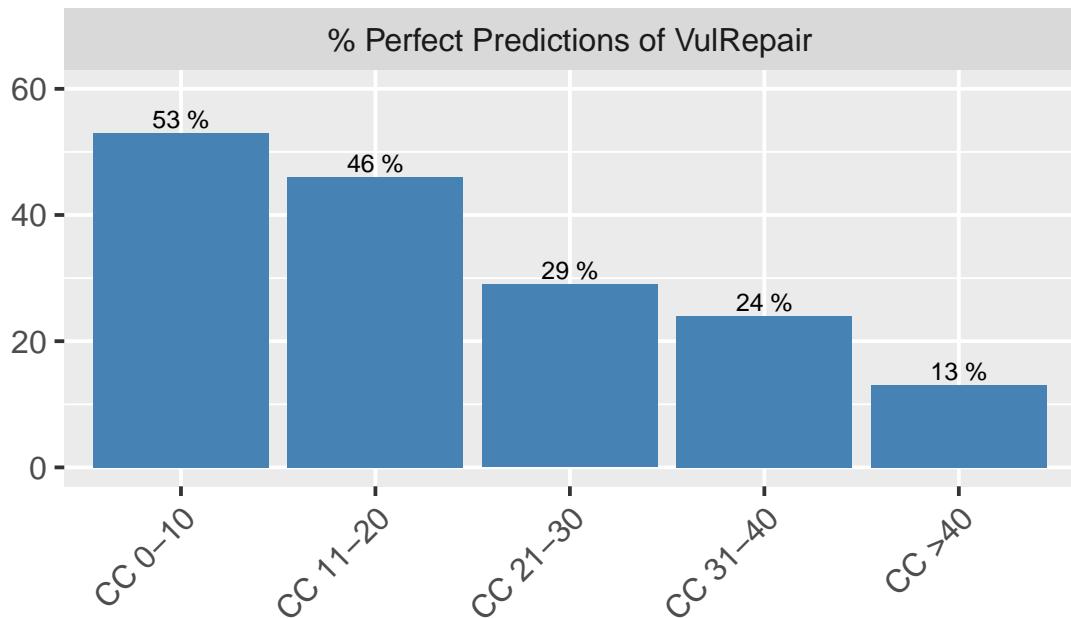


Figure 22: (Discussion) The accuracy of our VulRepair for various ranges of the Cyclomatic Complexity of the input vulnerable functions in the testing set. ( $\nearrow$ ) Higher % Perfect Predictions = Better.

### 6.6.3 How Does the Complexity of the Input Functions Impact the Accuracy of Our VulRepair?

It is possible that highly-complex vulnerable functions may be more difficult to generate repairs by our VulRepair than the others. Thus, we further investigate the accuracy of our VulRepair for various degrees of the complexity of the input functions. To do so, we measure the program complexity using a Cyclomatic Complexity measure. Cyclomatic Complexity (CC) is a quantitative measure of the number of linearly independent paths through a program’s source code.

**Our VulRepair achieves a higher accuracy for less-complex input functions than highly-complex input functions.** Figure 22 presents the accuracy of our VulRepair under different ranges of Cyclomatic Complexity (CC). We find that when the input functions are less complex (i.e., the CC is less than 20), our VulRepair can achieve %PP of 53% and 46%, which are better than the average accuracy (i.e., %PP of 44%). However, the accuracy of our VulRepair merely achieves %PP of 29%, 24%, and 13% when the input functions are more complex (i.e., the CC is higher than 20). Thus, *future researchers should further explore techniques that can handle highly-complex functions to further improve the accuracy of the NMT-based vulnerability repair models.*

Table 16: (Discussion) The % Perfect Predictions of our VulRepair according to the function length and the repair length.

		Function Lengths (#Tokens)					
		0-100	101-200	201-300	301-400	401-500	500+
Repair Lengths (#Repair Tokens)	0-10	77%	64%	75%	76%	67%	32%
	11-20	63%	56%	59%	43%	33%	32%
	21-30	50%	55%	56%	65%	56%	33%
	31-40	48%	53%	57%	42%	56%	15%
	41-50	54%	61%	53%	45%	20%	30%
	50+	48%	24%	32%	28%	16%	6%

#### 6.6.4 How Well Can Our VulRepair Handle the OOV Problem of Vulnerability Repairs?

To better understand how well can BPE handle the OOV problem of vulnerability repairs, we perform a further investigation to analyze the pairs of vulnerable function-repair where new tokens never appeared in the vulnerable function but appear in the vulnerable repairs (i.e., out-of-vocabulary). Among the 1,706 pairs in the testing dataset, we find that 37% of them (627 pairs) have new tokens that appear in the vulnerable repair, but never appeared in the vulnerable function. This means that the OOV problem is accounting for 37% of the testing dataset, indicating that the size of the OOV problems is considerably large and important. Among the 627 pairs with OOV problems, we find that 37% of the vulnerable functions ( $\frac{234}{627}$ ) can be correctly and automatically repaired by our VulRepair approach, indicating that our VulRepair can accurately generate vulnerability repairs when new are introduced in the repair version. On the other hand, VRepair which leverages the copy mechanism cannot accurately generate any vulnerability repairs when new tokens are introduced in the repair version, since the copy mechanism cannot reuse tokens that never appear in the vulnerable function to the vulnerability repairs. This finding highlights the importance of using BPE to handle the OOV problem for vulnerability repairs.

Nevertheless, we find that the correct vulnerability repairs (37%) often have new tokens ranging from 1 to 12 new tokens (avg=1.59), while the incorrect vulnerability repairs (the remaining 63%) have a relatively higher number of new tokens ranging from 1-100 new tokens (avg=4.85). This finding confirms that the cor-

---

rect generation of vulnerability repairs depends on its complexity and difficulty (#new tokens). Thus, *future researchers should further explore other techniques to address the OOV problem for more difficult types of vulnerability repairs.*

## 6.7 Related Work

**NMT-based APR.** Researchers proposed to leverage various Neural Machine Translation (NMT) approaches for Automated Program Repair (APR). For example, Chen et al. [22] proposed SequenceR, which is a vanilla version of Transformer with a copy mechanism to handle OOV. Jiang et al. [36] proposed CURE, which is a GPT architecture [206], pre-training on source code. Mashhadi and Hemmati [205] leveraged CodeBERT [40] to repair Java bugs automatically. Tufano et al. [198] leveraged an RNN-based NMT model to automatically generate repairs in the context of code review. Lutellier et al. [49] proposed CoCoNuT, which is a CNN-based NMT model to generate bug-fixes. Li et al. [207] proposed DLFix, which is a tree-based RNN architecture to generate bug-fixes. Thongtanunam et al. [203] proposed AutoTransform, which is a vanilla version of Transformer with a BPE tokenization to handle OOV. While NMT-based APR shares a similar concept of using NMT for a Code→Code task, NMT-based APR approaches [22, 36, 207] are designed to learn to generate a patch for bug-fixing purposes, which may not be related to other specific types of bugs like software vulnerabilities. In addition, such NMT-based APR approaches are designed to generate a patch that satisfies test cases.

Different from NMT-based APR, VulRepair aims to automatically generate vulnerability repairs that are exactly matched with the human-written repairs (i.e., the fix version).

**NMT-based AVR.** Researchers proposed NMT-based Automated Vulnerability Repair (AVR) approaches. For example, Chen et al. [23] proposed VRepair which leverages a word-level tokenizer and a vanilla Transformer model. Similar to VRepair, Chi et al. [35] proposed SeqTrans which relies on the same components as VRepair. Both VRepair and SeqTrans leverage a word-level tokenizer, however, Chen et al. [23] leverage the copy mechanism, and Chi et al. [35] leverage code normalization to solve the OOV problem.

Different from NMT-based AVR, our VulRepair is the first to leverage a T5 architecture with BPE and the pre-training of a large code corpus for automated vulnerability repairs. Our systematic and comprehensive evaluations also demon-

---

strate the substantial accuracy improvement of our approach to addressing various limitations of VRepair [23], highlighting the significant advancement of the NMT-based Automated Vulnerability Repair literature. Additional investigation in the Discussion section also provides recommendations for future researchers.

## 6.8 Threats to Validity

As for any empirical study, there are various threats to the validity of our results and conclusions.

*Threats to internal validity* are related to the degree to which our study minimizes systematic error. Our VulRepair consists of various hyperparameter settings (i.e., number of hidden layers, number of attention heads, and learning rate). Prior studies raise concerns that different hyperparameter settings may have an impact on the evaluation results, especially, for defect prediction models [90, 91]. However, finding an optimal hyperparameter setting can be very expensive given the large search space of the Transformer architecture. Instead, the goal of our work is not to find the best hyperparameter setting but to fairly compare the accuracy of our approach with the existing baseline approaches. Thus, the accuracy reported in the work serves as a lower bound of our approach, which can be even further improved through hyperparameter optimization. To mitigate this threat, we report the hyperparameter settings in the replication package to aid future replication studies.

*Threats to external validity* are related to the degree to which our findings can be generalized to and across other vulnerabilities and projects. Our VulRepair approach is evaluated on the CVEFixes [195] and Big-Vul [70] corpus, which consists of 8,482 vulnerability repairs from 180+ different CWEs. However, the results of VulRepair do not necessarily generalize to other CWEs and other datasets. Thus, other datasets can be explored in future work.

Recently, Liu et al. [208] suggested that the accuracy of an automated program repair approach should not be solely evaluated based on a perfect match. Instead, other measures should also be considered, e.g., the number of semantically correct repairs and the number of plausible patches. However, these two measures require test cases to evaluate whether the repairs can successfully pass the test cases or not. Unfortunately, there exists no test cases available in the experimental dataset that we used in this work. Thus, both measures cannot be evaluated. Nevertheless, future researchers should create new vulnerability

---

repair datasets where such repairs are reproducible and test case information is available.

## 6.9 Summary

In this chapter, we propose VulRepair, a T5-based automated software vulnerability repair approach. Through an extensive evaluation, we conclude that our VulRepair is considerably 13%-21% more accurate than VRepair and CodeBERT, highlighting the substantial advancement of NMT-based Automated Vulnerability Repairs. In addition, our VulRepair can accurately repair as many as 745 out of 1,706 real-world well-known vulnerabilities. Importantly, we find that our VulRepair can correctly repair 38% of the vulnerable functions related to the Top-10 most dangerous CWEs, e.g., CWE-416 (Use After Free), CWE-20 (Improper Input Validation), and CWE-78 (OS Command Injection), demonstrating the practicality and significance of our VulRepair for generating vulnerability repairs, helping under-resourced security analysts on fixing vulnerabilities.

Our additional analysis discovers important findings, leading to many open research challenges that future researchers should explore (e.g., to handle larger functions, handle rare types of vulnerabilities, to handle difficult repairs with many new tokens).

---

## **7 When Vulnerability Repair Meets Computer Vision – A Vision Transformer-Inspired Approach for Guiding Transformer Models to Focus on Vulnerable Code Areas and Generate Accurate Repair Patches**

© 2024 Association for Computing Machinery. This is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive version was published in ACM Transactions on Software Engineering and Methodology, 33(3), 1–29, published 15 March 2024, <https://doi.org/10.1145/3632746>.

---

## 7.1 Introduction

Software vulnerabilities are security flaws, glitches, or weaknesses found in software code that could lead to a severe system crash or be leveraged as a threat source by attackers [11]. According to the National Vulnerability Database (NVD), the number of vulnerabilities discovered yearly has increased from 6,447 in 2016 to 20,156 in 2021 and 18,017 vulnerabilities have been found in 2022. This trend indicates more vulnerabilities are being discovered and released every year, meaning that there will be more workloads for security analysts to track down and patch those vulnerabilities. In particular, it may take 58 days on average to fix a vulnerability based on vulnerability statistics reported in 2022 [209]. Recently, Deep Learning (DL)-based approaches have been proposed to automate the vulnerability repair process by learning the representation of vulnerable programs and generating repair patches accordingly, which may potentially accelerate manual security analysis processes. Specifically, the transformer architecture has been widely adopted to generate accurate vulnerability patches that repair the vulnerable code automatically [3, 23, 35, 210]. The attention-based transformer is shown to be more effective than RNNs because its self-attention mechanism learns global dependencies when scanning through each word embedding rather than processing input sequentially.

In the automated vulnerability repair (AVR) problem, a deep learning model consists of encoders to encode the code representations of the vulnerable function and the decoders generate repair code for vulnerable code areas in the function. Commonly, vulnerabilities in a function are caused by a few vulnerable code areas, hence previous studies have proposed various techniques to localize vulnerable code areas in a vulnerable function [1, 32, 109, 157]. For instance in Figure 23 and 24, the decoders only need to generate the repair code for specific vulnerable code areas. Thus, awareness and attention to the vulnerable code areas including vulnerable statements are crucially important. This further helps to guide an AVR model to emphasize and focus more on the vulnerable statements for producing better repairs. However, existing AVR approaches lack a mechanism to enhance awareness of vulnerable code areas during the vulnerability repair process. It is also challenging because vulnerable code areas can appear in different spatial locations. Toward this challenge, we observe that object detection in computer vision intuitively shares a similar concept to vulnerability repair because both approaches need to localize specific items in the input. Particularly, by linking the vulnerable code areas in a source code to the objects in an image, we hope to borrow the principles from the VIT-based objection de-

---

tection approaches [211–213] to propose a novel solution for the AVR problem.

In this work, we propose an AVR approach that can guide the encoders and decoders to focus more on vulnerable code areas during the repair process. In particular, our approach is inspired by the VIT-based approaches for object detection [211–213]. Figure 25 presents our analogy between object detection from the computer vision domain and vulnerability repair from the NLP domain. We connect detecting spatial objects in an image for predicting bounding boxes to localizing spatial vulnerable code areas in a vulnerable function for generating the corresponding repair code. Our model consists of a vulnerability repair encoder to produce code token embeddings for vulnerable functions and a vulnerability repair decoder to generate repair patches. As presented in Figure 25, the object queries are used in the VIT-based approaches for object detection aiming to attend to objects in an image for predicting the corresponding bounding boxes, on the other hand, we devise vulnerability queries (VQs) aiming to attend to the vulnerable code blocks in a source code for predicting repair tokens. Additionally, the cross-attention mechanism employed in the vulnerability repair decoder assists the VQs in cross-matching and paying more attention to the vulnerable code blocks.

To further strengthen the attention of the VQs to the vulnerable code areas and facilitate the repair generation for vulnerable code areas, we train an additional model to learn a vulnerability mask (VMs). The VMs are a probability distribution used to emphasize vulnerable code areas in vulnerable functions. We then incorporate the VMs with the cross-attention in our repair decoder that guides our VQs to attend more to vulnerable code areas and generate corresponding repairs. In addition, we apply the VMs to the self-attention of our repair encoder to pay attention to vulnerable code areas when encoding the representations of vulnerable functions. We name our approach VQM - *Vulnerability Repair Through Vulnerability Query and Mask*.

We conduct an experiment and compare our VQM approach with six competitive AVR baseline approaches (i.e., VRepair [23], VulRepair [3], TFix [210], CodeBERT [40], GraphCodeBERT [155], and SequenceR [22]). Through an extensive evaluation of our approach on 5,417 C/C++ vulnerable functions involving 2,095 different vulnerabilities spanning from 1999 to 2021, we empirically evaluate our approach by answering the following two research questions:

**(RQ1) What is the accuracy of our VQM approach for generating software**

---

## vulnerability repairs?

**Results.** Among all approaches included in our experiment, our VQM approach achieves the best percentage of perfect predictions of 32%, 43%, and 45% respectively when using beam=1,3,5 during repair generation.

### (RQ2) What are the contributions of each component of our VQM approach?

**Results.** Our method of applying vulnerability queries with vulnerability masks performs the best when compared with other variants in the ablation study. In addition, we find that using perfect vulnerability masks can achieve optimal performance, highlighting the effectiveness of our proposed vulnerability masks.

While our RQ1 and RQ2 delve into performance evaluations of our approach, the practical utility of AI-generated repairs for software developers remains a question unexplored. Consequently, in pursuit of this understanding, we address RQ3 by conducting a user study specifically aimed at gauging the perceptions of software developers possessing a security background towards AI-generated vulnerability repairs.

### (RQ3) Are AI-generated vulnerability repairs perceived as useful by software developers?

**Results.** Our survey study with 71 participants shows that 86% of participants perceive AI-generated vulnerability repairs as useful. In addition, 80% of them consider adopting AI-generated repairs if they are readily available and free of charge.

**The Novelty & Contributions** of this work are as follows:

- A novel vulnerability repair framework based on object detection that uses vulnerability queries to generate repair patches;
- A novel vulnerability mask that facilitates the repair model to locate vulnerable code tokens more accurately during vulnerability query;
- A comprehensive evaluation of our proposed approach against other AVR approaches using a benchmark dataset including real-world vulnerabilities; and
- An ablation study to assess the effectiveness of each component in our pro-

Vulnerable Function — CWE-787 (Out-of-bounds Write)	Repaired Function
<pre> 41 41 GPMF_ERR IsValidSize(GPMF_stream *ms, uint32_t size) 42 42 { 43 43     if (ms) 44 44     { 45 45         int32_t nestsize = (int32_t)ms-&gt;nest_size[ms-&gt;nest_level]; 46 46         if (nestsize == 0 &amp;&amp; ms-&gt;nest_level == 0) 47 47             nestsize = ms-&gt;buffer_size_longs; 50 50     } 51 51     return GPMF_ERROR_BAD_STRUCTURE; 52 52 }</pre>	<pre> 41 41 GPMF_ERR IsValidSize(GPMF_stream *ms, uint32_t size) 42 42 { 43 43     if (ms) 44 44     { 45 45         int32_t nestsize = (int32_t)ms-&gt;nest_size[ms-&gt;nest_level]; 46 46         uint32_t nestsize = (uint32_t)ms-&gt;nest_size[ms-&gt;nest_level]; 47 47         if (nestsize == 0 &amp;&amp; ms-&gt;nest_level == 0) 48 48             nestsize = ms-&gt;buffer_size_longs; 50 50     } 51 51     return GPMF_ERROR_BAD_STRUCTURE; 52 52 }</pre>
Subword-tokens of the vulnerable function $x_i = [t_1, \dots, t_n]$	Subword-tokens of the vulnerability repair $y_i = [r_1, \dots, r_k]$
<code>['GP', 'MF', ' ', 'ERR', 'IsValid', 'Size', '(', 'GP', 'MF', ' ', 'stream', '*', 'ms', ' ', 'uint', '32', ' ', 't', 'size', ')', '(', 'if', '(', 'ms', ')', 't', 'int', '32', ' ', 't', 'nest', 'size', '=', '(', 'int', '32', ' ', 't', ')', 'ms', ' ', '&gt;', 'nest', ' ', 'size', '[', 'ms', ' ', '&gt;', 'nest', ' ', 'level', ' ', 'level', ' ', 'level', ' ', 'level', ' ', 'if', '(', 'nest', ' ', 'size', ' ', '==', '0', '&amp;&amp;', 'ms', ' ', '&gt;', 'nest', ' ', 'level', ' ', 'level', ' ', 'level', ' ', 'if', '(', 'nest', ' ', 'size', ' ', '==', '0', ' ', 'ms', ' ', '&gt;', 'buffer', ' ', 'size', ' ', 'l', 'ongs', ' ', 'if', '(', 'size', ' ', '2', '&lt;', 'nest', ' ', 'size', ')', 'return', 'GP', 'MF', ' ', 'OK', ' ', ')', 'return', 'GP', 'MF', ' ', 'ERROR', ' ', 'BAD', ' ', 'STRUCT', 'URE', ' ', '']</code>	<code>[&lt;S2SV_ModStart&gt;, 'ms', ')', 't', 'uint', '32', ' ', 't', 'nestsize', '=', '(', 'uint', '32', ' ', 't', '&lt;S2SV_ModEnd&gt;', ')', 'ms', '&gt;']</code> <p>Note.  <code>'ms', ')', 't'</code> are context tokens before the fix  <code>)', 'ms', '&gt;'</code> are context tokens after the fix</p> <p>Those context tokens highlighted in blue are used to match the repair patches to the vulnerable parts in a vulnerable function.</p>

Figure 23: (CWE-787 Out-of-bounds Write) A real-world example [214] of vulnerability in a C function is caused by an inappropriate variable type definition, which could lead to serious security breaches or system crashes. The red tokens are vulnerable tokens; the green tokens are tokens used to repair; and the blue tokens are context tokens used to locate where the repair tokens should be implemented. It is worth noting that the model may not always match repairs to their correct locations when repeated context tokens are present. Nonetheless, in our experiments, we adopt the approach of Chen et al. [23] by utilizing three context tokens, which effectively align all repairs in our studied dataset. The left column presents the vulnerable function where below are sub-word tokens  $x_i$  used as input for our repair model. It can be seen that only some of the tokens highlighted in red (i.e., tokens corresponding to Line 45) are vulnerable. The right column presents the corresponding repaired function where below are sub-word tokens  $y_i$  as the repair patch output by our repair model.

posed approach.

- A user study to assess the usefulness of AI-generated vulnerability repairs from software developers’ perspective.

## 7.2 Our Proposed Approach

### 7.2.1 Usage Scenario

Imagine a software development team that is particularly concerned about identifying and addressing vulnerabilities within their functions and has decided to leverage our proposed VQM approach. In practice, they analyze source code using static analysis tools like Cppcheck [28] or deep learning-based vulnerability prediction tools [5] to identify potential vulnerabilities. However, such tools

Vulnerable Function — CWE-125 (Out-of-bounds Read)	Repaired Function
<pre> 809 809 static inline unsigned short ReadPropertyUnsignedShort(const 810 810   EndianType endian, 811 811   const unsigned char *buffer) 812 812   { 813 813     unsigned short 814 814     value; 815 815     if (endian == LSBEndian) 816 816       value=(unsigned short) ((buffer[1] &lt;&lt; 8)   buffer[0]); 817 817       return((unsigned short) (value &amp; 0xffff)); 818 818     } 819 819     value=(unsigned short) (((unsigned char *) buffer)[0] &lt;&lt; 8)   820 820       ((unsigned char *) buffer)[1]; 821 821     return((unsigned short) (value &amp; 0xffff)); 822 822 }</pre>	<pre> 809 809 static inline unsigned short ReadPropertyUnsignedShort(const 810 810   EndianType endian, 811 811   const unsigned char *buffer) 812 812   { 813 813     unsigned short 814 814     value; 815 815     if (endian == LSBEndian) 816 816       value=(unsigned short) ((buffer[1] &lt;&lt; 8)   buffer[0]); 817 817       return((unsigned short) (value &amp; 0xffff)); 818 818     value=(unsigned short) buffer[1] &lt;&lt; 8; 819 819     value = (unsigned short) buffer[0]; 820 820     return(value &amp; 0xffff); 821 821   } 822 822 }</pre>
<p>Subword-tokens of the vulnerable function  <math>x_i = [t_1, \dots, t_n]</math></p> <pre>[static', 'inline', 'unsigned', 'short', 'Read', 'Property', 'Unsigned', 'Short', '(', 'const', 'End', 'ian', 'Type', 'Endian', ')', 'const', 'unsigned', 'char', '**', 'buffer', ')', '{', 'unsigned', 'short', 'value', ':', 'if', '(', 'Endian', '==', 'L', 'SB', 'Endian', ')', '{', 'value', '=', '(', 'unsigned', 'short', ')', '(', '(', 'buffer', '[', '1', ')', '&lt;&lt;', '8', ')', ')', 'buffer', '[', '0', ')', '}', 'return', '(', '(', 'unsigned', 'short', ')', '(', 'value', '&amp;', '0', 'xffff', ')', ')', '}', 'value', '!=', '(', 'unsigned', 'short', ')', '(', '(', '(', '(', 'unsigned', 'char', '**', ')', 'buffer', '[', '0', ')', '&lt;&lt;', '8', ')', ')', '(', '(', 'unsigned', 'char', '**', ')', 'buffer', ')', '(', '1', ')', '}', 'return', '(', '(', 'unsigned', 'short', ')', '(', 'value', '&amp;', '0', 'ffff', ')', ')', '}', '}</pre>	<p>Subword-tokens of the vulnerability repair  <math>y_i = [r_1, \dots, r_k]</math></p> <pre>[&lt;S2SV_ModStart&gt;, 'unsigned', 'short', ')', '&lt;S2SV_ModEnd&gt;', 'buffer', '[', '1', '&lt;S2SV_ModStart&gt;', ')', '&lt;&lt;', '8', ':', 'value', '!=', '(', 'unsigned', 'short', ')', '&lt;S2SV_ModEnd&gt;', 'buffer', '[', '0', '&lt;S2SV_ModStart&gt;', '[', '0', ')', '&lt;S2SV_ModEnd&gt;', ')', 'return', '(', '&lt;S2SV_ModStart&gt;', ')', 'return', '(', '&lt;S2SV_ModEnd&gt;', 'value', '&amp;', '0', 'ffff', '&lt;S2SV_ModEnd&gt;', ')', ')', '}', '&lt;S2SV_ModStart&gt;', 'unsigned', 'short', ')', 'buffer', '&lt;S2SV_ModEnd&gt;', '[', '0', ')', '&lt;S2SV_ModStart&gt;', ')', '&lt;&lt;', '8', ':', 'value', '!=', '&lt;S2SV_ModEnd&gt;', '(', 'unsigned', 'short', '&lt;S2SV_ModStart&gt;', ')', 'return', '&lt;S2SV_ModStart&gt;', '&amp;', '0', 'ffff', ')', '&lt;S2SV_ModEnd&gt;', ')', '}', '}</pre>

Figure 24: (CWE-125 Out-of-bounds Read) A real-world example [215] of vulnerability in a C function. In the vulnerable function on the left, the `value` is calculated using `(buffer[1] << 8) | buffer[0]` (i.e., line 816), which shifts the second byte of the buffer by 8 bits to the left and then tries to combine it with the first byte. This operation could lead to accessing memory beyond the buffer's bounds and result in undefined behavior. A similar vulnerability also occurs in the second vulnerable block (i.e., lines 819-821). In the repaired function, the problematic byte-order conversion operations in both vulnerable blocks have been restructured to ensure proper handling of byte manipulation and boundary checks. The key change is in the handling of the byte-order conversion, where instead of performing the bit shift and combination in a single step, the code is split into separate steps. This ensures that the operations involving byte manipulation are performed sequentially and within the buffer's boundaries.

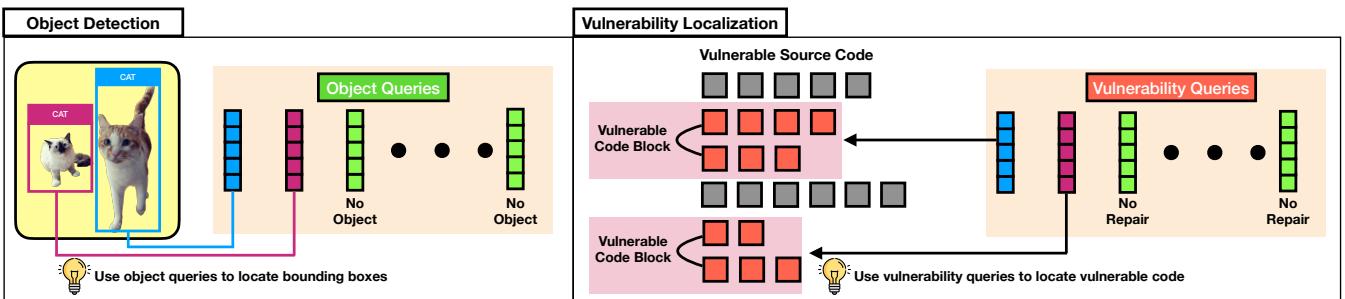


Figure 25: Intuitively, not all code tokens in a program need to be repaired and the repair can be in multiple areas. Similarly, not all pixels in an image have objects and the objects can appear in multiple locations in an image. Thus, in object detection, object queries are used in VIT-based approaches [211–213] to predict bounding boxes and locate objects. With a similar principle of object detection, we leverage vulnerability queries to attend more to the vulnerable code tokens in the vulnerable code areas and generate repairs for them.

cannot suggest vulnerability repairs. Thus, they leverage our VQM framework to obtain repair patches suggested by AI models. Finally, security experts on the team will validate the AI-generated patches before implementing them into their software system.

### 7.2.2 Problem statement

Similar to previous studies [3, 23], we focus on function-level vulnerability repair, assuming all vulnerabilities can be resolved within the function-level scope. Our vulnerability repair approach can handle fixing multiple vulnerable parts in a vulnerable function and satisfy three key behaviors (i.e., add, delete, and replace) to fix vulnerabilities. The special tokens, “<S2SV\_ModStart>” and “<S2SV\_ModEnd>” are added into repair patches to determine the behavior of add, delete, or replace, as detailed in Section 3.2 in Chen et al.’s work [23]. In particular, the model will learn to generate three context tokens to match the repair patches back to the vulnerable function and implement the repairs. For instance, in the right part of Figure 23, the first three repair tokens (“ms”, “)”, and “{”) and the last three repair tokens (“)”, “ms”, and “->”) are context tokens used to match the repair tokens to the vulnerable function. For simplicity, we use a vulnerable function with one vulnerable part as an example in Figure 23, however, the same repair manner can be repeated to fix vulnerable functions with multiple vulnerable parts.

Assuming we have a source code data set consisting of vulnerable source code functions along with corresponding repair patches that repair the vulnerable parts of those functions. We denote the data set as  $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$ , where  $x_i$  is a vulnerable function and  $y_i$  is its repair patch. Note that each  $y_i$  is not a com-

---

plete function but a patch used to repair the vulnerable part in the corresponding  $x_i$  as shown in Figure 23. The mapping between vulnerable functions,  $x_i$ , and repair patches,  $y_i$ , has been completed by Chen et al. [23] through parsing the code difference between the vulnerable and the fixed version of the source functions from real-world vulnerability datasets [70, 195]. In this work, we leverage BPE algorithm [50] to tokenize  $x_i$  and consider  $x_i$  as a sequence of code tokens denoted as  $x_i = [t_1, t_2, \dots, t_n]$  where the code token  $t_j, j = 1, \dots, n$  could be a clean token or vulnerable token (i.e., the tokens highlighted in red in Figure 23). Similarly, a repair patch  $y_i = [r_1, \dots, r_k]$  where  $y_i$  consists of  $k$  number of repair tokens  $r_j, j = 1, \dots, k$ . Each code token  $t_j$  and repair token  $r_j$  will be embedded into a vector for the model to learn its representation as detailed in Section 7.2.3. We define this problem as a sequence-to-sequence code generation task with an objective to capture vulnerable code tokens in  $x_i$  to generate corresponding repair patch  $y_i$ .

As presented in Figure 23, the vulnerable function *IsValidSize* only consists of one vulnerable code area (i.e., statement 45). In particular, those repair tokens (i.e.,  $[r_1, \dots, r_k]$ ) are only related to a few vulnerable tokens in the vulnerable function. Thus, it is a challenging task to generate repair tokens specifically for the vulnerable code area. To address this challenge, we propose vulnerability queries and masks to guide our repair model to pay more attention to the vulnerable code areas when generating their corresponding repair tokens. In what follows, we illustrate the technical details of our approach.

### 7.2.3 Vulnerability Repair Via Vulnerability Query and Mask

Our approach is inspired by the VIT-based approaches [211–213] for object detection where we link detecting spatial objects in an image for predicting bounding boxes to localizing vulnerable code tokens in a source code for generating the repair tokens. Our model consists of an encoder to produce code token embeddings for code tokens and a decoder to generate repair tokens.

Both encoder and decoder are developed based on the transformer architecture [69]. The main component of the encoder is multi-head self-attentions with the aim of learning code token embeddings. Similar to DeTR [211], the decoder utilizes both multi-head self-attentions and cross-attentions. The purpose of the cross-attentions is to cross-match vulnerability queries and their corresponding vulnerable code tokens in a vulnerable area. Ideally, when vulnerability queries achieve good matches with their vulnerable code tokens, they possess sufficient

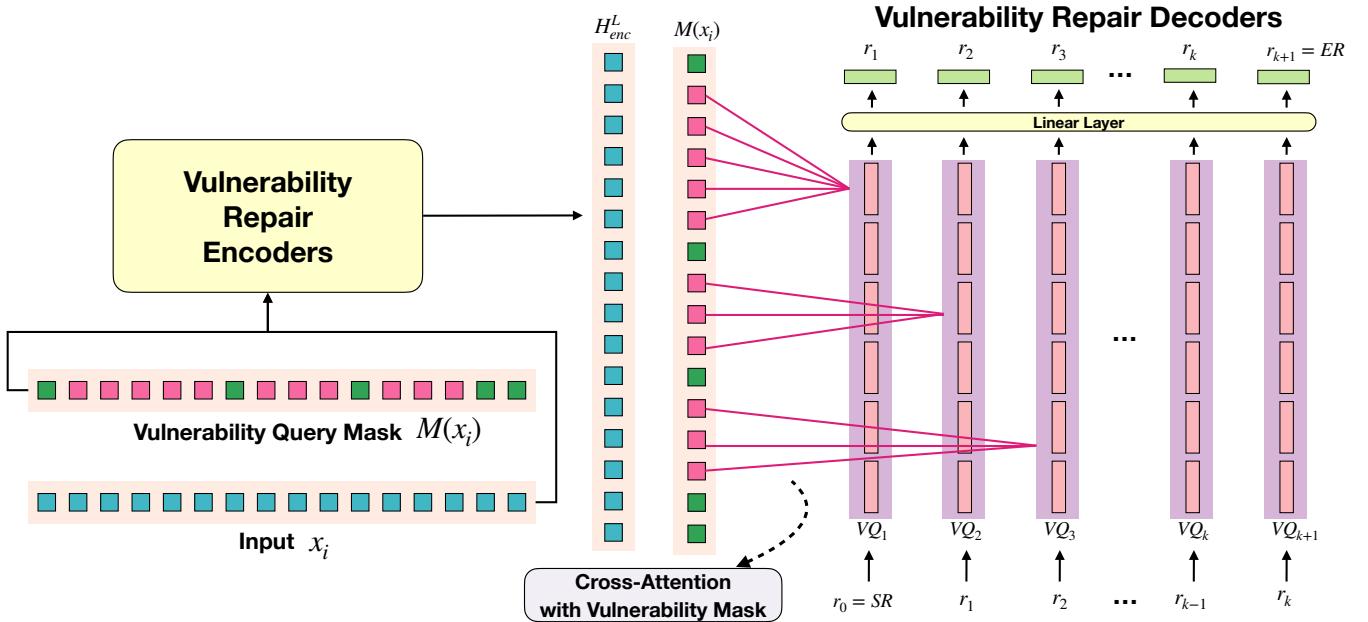


Figure 26: An overview architecture of our VQM approach. Input tokens  $x_i = [t_1, \dots, t_n]$  and vulnerability masks  $M(x_i)$  are input to encoders that output the embeddings of input tokens  $H_{enc}^L$ , where  $M(x_i)$  helps to emphasize the vulnerable embeddings. In decoders, each vulnerability query  $VQ_i$  is initialized from the previous repair token  $r_{i-1}$ , which is forwarded through multiple decoder layers followed by a linear layer to generate a repair token  $r_i$ . In each decoder, a cross-attention with  $M(x_i)$  to emphasize vulnerable embeddings is leveraged to cross-match  $VQ_i$  and  $H_{enc}^L$  and generate repairs corresponding to the vulnerable tokens.

---

information to generate repair tokens.

Additionally, to orient the matching process for attending more to vulnerable code tokens inside a source code, we propose to learn a vulnerability mask and apply it to both the encoder self-attention and decoder cross-attention mechanism. Particularly, we rely on the information on vulnerable tokens to train an additional model that outputs the possibility of a code token being a vulnerable token. We then base on these vulnerable scores to conduct a vulnerability mask.

In what follows, we present the technicality of the vulnerability repair encoder, the vulnerability repair decoder, and how to conduct and incorporate vulnerability masks into our framework.

#### 7.2.3.1 Vulnerability Repair Encoder

The purpose of the encoder is to produce code token embeddings for a given source code. Each token is embedded into a vector in  $\mathbb{R}^{d=768}$  by an embedding layer and input to the first encoder block. A stack of encoder blocks is leveraged to encode the representation for an embedded sequence through their self-attention layers followed by feed-forward neural networks, and each encoder block can be described as follows:

$$A^t = LN(MultiAttn(H_{enc}^{t-1})) + H_{enc}^{t-1}$$
$$H_{enc}^t = LN(FFN(A^t) + A^t)$$

where the hidden states from the previous encoder block  $H_{enc}^{t-1}$  forwards through a multi-head self-attention  $MultiAttn$  followed by a 2-layer feed-forward neural network  $FFN$ , and a layer normalization  $LN$ . The process will iterate until we obtain the last encoder hidden states  $H_{enc}^L$  to represent the vulnerable function. Here we note that  $L$  is the number of encoder blocks applied and  $H_{enc}^L$  contains the code token embeddings.

#### 7.2.3.2 Vulnerability Repair Decoder

Input to the vulnerability repair decoder is the vulnerability queries (VQ), each of which aims to match and capture information on vulnerable code tokens in a given source code.

The first VQ embeddings  $Q^0 = [q_1^0, \dots, q_k^0]$  are conducted and fed through several following decoder blocks. In each block, we apply both multi-head self-attention

---

and cross-attention as follows:

$$\begin{aligned}\hat{Q}^t &= LN(MultiAttn(Q^{t-1})) + Q^{t-1} \\ A_{cross}^t &= LN(CrossAttn(\hat{Q}^t, H_{enc}^L)) + Q^{t-1} \\ Q^t &= LN(FFN(A_{cross}^t) + A_{cross}^t)\end{aligned}$$

where  $H_{enc}^L$  is the encoder output.

It is worth noting that the cross-attention  $CrossAttn$  assists us in cross-matching the vulnerability query embeddings  $Q^t = [q_0^t, \dots, q_k^t]$  and the code token embeddings. If trained appropriately, the vulnerability query embeddings  $q_0^t, \dots, q_k^t$  attend and emphasize more the vulnerable code token embeddings in the vulnerable function, which finally contain sufficient information to generate the repair tokens.

Eventually, we obtain the output VQ embeddings  $Q^U = [q_0^U, \dots, q_k^U]$  where  $U$  is the number of the decoder blocks applied. On top of these VQ embeddings, we predict the repair tokens  $r_1, \dots, r_k$ . Specifically, we dedicate a linear layer on each VQ embedding  $q_0^U, \dots, q_k^U$  and aim to predict  $r_1, \dots, r_k$  and  $r_{k+1} = ER$  (i.e., the end repair token) by maximizing the likelihood with respect to a mini-batch of  $x_i$ :

$$p(y_i | x_i) = p(r_1, \dots, r_k | t_1, \dots, t_n) = \prod_{j=0}^k p(r_{j+1} | q_j^U) \quad (11)$$

where  $x_i = [t_1, \dots, t_n]$  is the source code and  $y_i = [r_1, \dots, r_k]$  is the corresponding repair patch.

The next question is how to initialize the first VQ embeddings  $Q^0 = [q_0^0, \dots, q_k^0]$ . Different from ViT-based object detection approaches [211–213], we do not initialize the first VQ embeddings  $Q^0 = [q_0^0, \dots, q_k^0]$  randomly. Indeed, we initialize  $Q^0 = [q_0^0, \dots, q_k^0]$  more informatively by setting  $q_0^0 = SR$  (i.e., the specific embedding for the start repairing token),  $q_j^0 = r_j, j = 1, \dots, k$ . By this informative initialization, we reframe the vulnerability repair problem as the task of generating repair patches in the source code.

The inference process is hence very natural. Given a source code  $x_i = [t_1, \dots, t_n]$ , we pass it through the vulnerability repair encoder to work out the encoder output  $H_{enc}^L$ . We start with the first VQ embedding  $q_0^0 = SR$  and feed to the vulnerability repair decoder to generate the first repair token  $r_1$ . We then set VQ embedding

---

$q_1 = r_1$  and feed it to the vulnerability repair decoder to generate the second repair token  $r_2$ . We repeat this process until we reach the *ER* token.

As mentioned before, the key factor to the success of our approach is how to accurately cross-match between the vulnerability queries and the vulnerable code tokens of a given source code. Currently, we expect that the cross-attention mechanism guided by maximizing the likelihood in Eq. (11) supports us in realizing this. To further strengthen the cross-matching, we learn a vulnerability mask that highly focuses on the vulnerable code tokens and then apply it to the encoder self-attention and the decoder cross-attention mechanism.

#### 7.2.3.3 Learning and applying vulnerability mask

In what follows, we present how to learn a vulnerability mask and then apply it to our model.

**Learning vulnerability mask** Note that for our dataset  $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$ , each vulnerable function  $x_i = [t_1, \dots, t_n]$  is a sequence of code token in which we know exactly the vulnerable scope or information if a code token  $t_j$  belongs to a vulnerable statement. In other words, we also possess the token-level vulnerable label  $v_i = [u_1, \dots, u_n]$  wherein  $u_j = 1$  means that the code token  $t_j$  belongs to a vulnerable statement and otherwise. For example, in the source code presented in Figure 23, the code tokens highlighted in red are the vulnerable code tokens labeled 1.

We now take advantage of this crucial information to learn vulnerability masks. Basically, we train an additional model to predict the vulnerability masks. Specifically, we leverage a pre-trained CodeBERT [40] model in learning the vulnerability masks. Each  $t_i$  in  $x_i$  is embedded into a vector in  $\mathbb{R}^{d=768}$  and forwarded through 12 layers of the BERT architecture. We then use a global max pooling layer and a sigmoid activation to obtain the probability mask  $m(x_i)$  and minimize the following cross-entropy loss with respect to a mini-batch of  $x_i$ :

$$H(x_i, v_i) = - \sum_{j=1}^n [u_j \log m_j(x_i) + (1 - u_j) \log (1 - m_j(x_i))] \quad (12)$$

Finally, to sharpen the vulnerability mask, we apply the following transformation

---

with a threshold value of 0.5:

$$M(x_i) = \frac{\beta}{1 + \exp\{-\alpha(m(x_i) - 0.5)\}}$$

where  $\alpha > 0$  and  $\beta > 0$  are two parameters to control the sharpness of the vulnerability mask.

In Figure 27, we visualize how  $\alpha$  and  $\beta$  affect the vulnerability masks. It can be seen that  $\alpha$  controls how fast the curve gets saturated and a large  $\alpha$  value forms a line approaching vertical at  $x = 0.5$ . On the other hand,  $\beta$  controls the gap between vulnerable and benign scores. We use a high value of  $\alpha = 1000$  so the model prediction threshold can approach a common threshold value of 0.5. Tokens are classified as vulnerable by the model if their prediction probability surpasses 0.5; otherwise, they are categorized as benign. We chose a relatively low value of  $\beta = 0.1$  to transform our vulnerability mask. Notably, a high  $\beta$  value would result in higher masking values. However, such high masking values may interfere with the hidden representation of our main repair encoders and decoders. Thus, we select a low  $\beta$  value to slightly adjust the self-attention weights and guide the repair model.

**Applying vulnerability mask to our model.** Our vulnerability mask finds application in rectifying vulnerable code segments requiring either “replace” or “delete” actions for resolution. In scenarios involving replacement, our vulnerability mask emphasizes the vulnerable section to be replaced by repair tokens. In the context of deletion, the mask underscores the vulnerable code slated for removal. However, it is important to note that in scenarios requiring “addition,” our vulnerability mask will not highlight anything, as no vulnerable code needs to be highlighted.

We incorporate our vulnerability mask into both the encoders’ self-attention output and the decoders’ cross-attention. For the encoder, we apply as follows:

$$\begin{aligned} A^t &= LN(MultiAttn(H_{enc}^{t-1}) + M(x_i) \otimes MultiAttn(H_{enc}^{t-1})) + H_{enc}^{t-1} \\ H_{enc}^t &= LN(FFN(A^t) + A^t) \end{aligned}$$

Here,  $\otimes$  denotes the element-wise product that produces the vector  $[M_j(x_i)B_j^t]_{j=1}^n$  where  $B^t = MultiAttn(H_{enc}^{t-1})$ .

For the cross-attention in the decoder, we apply as follows:

$$\hat{Q}^t = LN(MultiAttn(Q^{t-1})) + Q^{t-1}$$

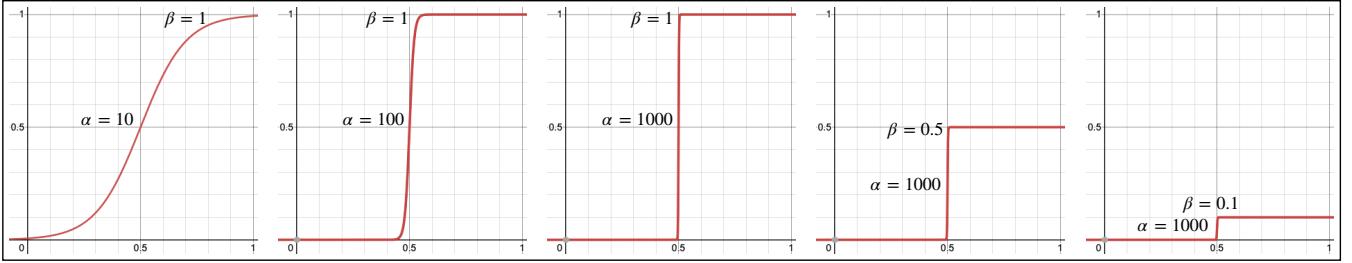


Figure 27: The plots of the vulnerability mask transformation to see how  $\alpha, \beta$  control the sharpness of vulnerability mask.

$$A_{cross}^t = LN(CrossAttn(\hat{Q}^t, H_{enc}^L + M(x_i) \otimes H_{enc}^L)) + Q^{t-1}$$

$$Q^t = LN(FFN(A_{cross}^t) + A_{cross}^t)$$

Finally, the entire framework of our approach encapsulated the vulnerability repair encoder, vulnerability repair decoder, and how to incorporate vulnerability masks are summarized in Figure 26.

## 7.3 Experimental Design

### 7.3.1 Research Questions

In this work, we aim to evaluate the effectiveness of our proposed VQM approach by answering our two research questions. In RQ1, we compare our approach with existing baseline methods for vulnerability repairs as described in Section 7.3.2. In RQ2, we focus on studying the proposed components in our VQM approach and present an ablation study. Below, we present the motivation for our two research questions.

**(RQ1) What is the accuracy of our VQM approach for generating software vulnerability repairs?** Transformer-based approaches have been leveraged for the automated vulnerability repair (AVR) problem [3, 23, 35]. While a vulnerable function may only consist of a few vulnerable codes to repair as shown in Figure 23, previous approaches can only implicitly learn the matching between vulnerable code areas and their repairs. On the other hand, we propose our VQM approach to explicitly guide the repair model to attend to those vulnerable areas and generate corresponding repairs. We formulate this RQ to assess the accuracy of VQM when comparing it to six other baseline approaches for the AVR introduced in Section 7.3.2.

**(RQ2) What are the contributions of each component of our VQM approach?**

---

Our VQM involves two key components, i.e., vulnerability queries (VQs) and vulnerability masks (VMs), to help the transformer model focus more on vulnerable code areas during the generation of repairs. However, little is known about the effectiveness of applying our proposed components on top of the transformer encoder-decoder model. Thus, we formulate this RQ and conduct an ablation study regarding the proposed VQs and VMs to assess their effects on the transformer model.

### 7.3.2 Baseline approaches

Our approach is evaluated against the leading automated vulnerability repair (AVR) methods, namely VRepair [23] and VulRepair [3], to assess its effectiveness. Moreover, we consider seq2seq-based AVR methods like TFix [210] and SequenceR [22] as baseline approaches for comparison. Additionally, we compare our approach with state-of-the-art pre-trained transformer models designed for source code, such as CodeBERT [40] and GraphCodeBERT [155], which are widely employed for addressing tasks related to source code. This comprehensive evaluation allows us to establish the advantages and novel contributions of our approach in the context of software vulnerability repair. The GPT2-CSRC approach is also included to compare our method with a decoder-only pre-trained model. In addition, we include automated program repair (APR) approaches such as CURE [36] and DLFix [207]. The details of baseline methods included in our evaluation are as follows:

- **VRepair**: A vanilla transformer architecture for the AVR task [23]. We replicate VRepair by following the instructions provided by Chen et al. to build and train the model.
- **VulRepair**: A T5-based approach [3] that relies on a large language model pre-trained on source code corpus [132]. We reproduce VulRepair using the repository provided by Fu et al..
- **TFix**: A T5-based approach [210] that relies on a large language model pre-trained on natural language corpus [216]. We reproduce TFix using the repository provided by Berabi et al..
- **SequenceR**: An RNN-based approach with bi-directional LSTM encoders and unidirectional LSTM decoders [22]. We replicate SequenceR by following the instructions provided by Chen et al. to build and train the model.

- 
- **CodeBERT**: A BERT-based large language model for source code [40], which has been leveraged to repair Java programs [205]. We reproduce CodeBERT using the repository provided by Feng et al..
  - **GraphCodeBERT**: An extensive version of CodeBERT by considering the Data Flow Graph (DFG) during training [155]. We reproduce GraphCodeBERT using the repository provided by Guo et al..
  - **GPT2-CSRC**: Utilizing a decoder-only model based on GPT-2 and a BPE tokenizer, GPT2-CSRC has undergone pre-training on an extensive dataset of approximately 17GB of C/C++ code. This dataset was curated from the top 10,000 most widely adopted Debian packages [56].
  - **CURE**: An automated program repair method driven by code awareness [36], this method harnesses the CoNut architecture [49] and integrates a decoder-only CodeGPT model alongside a BPE tokenizer. Moreover, during beam search, a code-aware search strategy is applied to enhance the output generation process.
  - **DLFix**: DLFix is a program repair technique built upon Tree-LSTM [207], a framework that utilizes abstract syntax trees (ASTs) as its input, standardized across variable names. To transform textual input into a vector space, Word2Vec embeddings [217] are used. Additionally, DLFix incorporates strategies such as patch re-ranking and program analysis filters to enhance the output generated by the model.

### 7.3.3 Experimental Dataset

We use the same experimental dataset provided by Chen et al. [23] to evaluate our approach. The dataset consists of Big-Vul [70] and CVEfixes [195] vulnerability fix corpus written in C/C++. The Big-Vul dataset was collected from 348 open-source GitHub projects by crawling the Common Vulnerabilities and Exposures (CVE) database. In total, Big-Vul contains 3,754 code vulnerabilities from 2002 to 2019. On the other hand, the CVEfixes dataset was constructed similarly to the Big-Vul, which consists of 5,365 vulnerabilities collected from 1,754 projects from 1999 to 2021. Specifically, we leverage both datasets pre-processed by Chen et al. [23] and obtain 5,417 samples spanning 2,095 different vulnerabilities (i.e., CVE-ID) after dropping null and duplicate samples.

---

Table 17: Training scheme of our VQM approach. Note. #: Scheme for training the mask prediction model; \*: Scheme for training the repair model.

Training	Data	$Seq_{enc}$	$Seq_{dec}$	Optim	Sch.	LR	Grad Clip	Bhz	Epo
#Pre-train	Bug Fix	512	N/A	AdamW	Linear	1e-4	1.0	16	75
#Fine-tune	Vul Fix	512	N/A	AdamW	Linear	1e-4	1.0	16	75
*Pre-train	Bug Fix	512	256	AdamW	Linear	1e-4	1.0	8	75
*Fine-tune	Vul Fix	512	256	AdamW	Linear	1e-4	1.0	8	75

### 7.3.4 Parameter Setting

We split the data into 70% for training, 10% for validation, and 20% for testing. We use a pre-trained T5 model provided by Wang et al. [132], which was pre-trained using multiple denoising objectives related to programming languages. The hyperparameter settings used to reproduce our mask model and repair model are presented in Table 17.

### 7.3.5 Model Training

Given that the existing vulnerability repair dataset only contains limited samples, pre-training on a larger bug fix dataset can further enhance the performance of a vulnerability repair model as demonstrated by Chen et al. [23]. The intuition is that the software vulnerability is a sub-domain of the software defect (i.e., bugs) domain which increases the transferability between the two tasks. Thus, for each model including ours, we first pre-train on the bug-fix dataset provided by Chen et al. [23], which consists of 23,607 samples to obtain more meaningful pre-trained weights for the vulnerability repair downstream task. Note that the bug fix dataset does not overlap with our experimental dataset introduced in Section 7.3.3. We report the details of our training settings in our replication package at <https://github.com/awsm-research/VQM>.

## 7.4 Experimental results

### (RQ1) What is the accuracy of our VQM approach for generating software vulnerability repairs?

**Approach.** To answer

this RQ, we evaluate the accuracy of vulnerability repair approaches using the percentage of perfect predictions (%PP) similar to previous AVR studies [3, 23]. If any of the repairs generated by the beam search is exactly the same as the ground-truth label (i.e., human-written vulnerable repair), it is considered as a

---

Table 18: (Main results) The comparison between our VQM approach and other baselines. Accuracy is presented in percentage. Beam=k shows the measure of %PP. We conducted the experiments five times with different random seeds and reported the mean performance plus minus standard deviation.

Methods	Beam=1	Beam=3	Beam=5
VQM( <b>Ours</b> )	<b>32.33±1.12</b>	<b>42.72±0.86</b>	<b>45.14±0.86</b>
VulRepair	29.65±1.27	39.85±1.31	42.79±1.15
TFix	15.41±1.96	26.7±1.69	30±1.77
GPT2-CSRC	11.93±0.71	19.27±0.65	24.77±0.81
CURE	11.19±0.53	20.55±1.03	26.06±0.65
GraphCodeBERT	9.15±0.43	16.83±0.85	21.38±0.54
CodeBERT	7.47±0.61	13.69±0.37	16.85±0.17
VRepair	5.36±0.55	10.31±0.29	13.12±0.53
DLFix	0.51±0.08	1.05±0.15	1.53±0.23
SequenceR	0.0±0	0.44±0.13	0.53±0.27

correct prediction. Thus, the overall %PP across all testing data is computed as the number of correct predictions divided by the number of testing samples. The %PP measures how much of the predictions generated by each approach can be applied to the vulnerable functions, where the quality and the applicability of those correct repairs are guaranteed by the human-written ground truths.

We use the dataset described in Section 7.3.3 and compare our proposed method with the baselines introduced in Section 7.3.2. To ensure the robustness of our experimental results, we ran our experiment five times for each approach by setting different random seeds. During beam search, we use  $beam \in [1, 3, 5]$  to evaluate all of the methods. Such beam settings lead to fewer repair candidates generated by the models, which would be more practical in real-world scenarios so developers will not need to inspect many repair candidates.

**Result.** The experimental results are presented in Table 18. **Regardless of the number of beams, our VQM method outperforms all baselines and achieves the best %PP.** When comparing only the top-1 repair candidates (i.e.,  $beam = 1$ ), our VQM is 2.68% (VulRepair) and 16.92% (TFix) better than pre-trained transformer encoder-decoder approaches, 21% (CURE) and 31% (DLFix) better than APR approaches, 23.18% (CodeBERT) and 24.86% (GraphCodeBERT) better than BERT-based approaches, and 20% better than the decoder-only approach, GPT2-CSRC.

The improvement of our VQM approach over previous state-of-the-art transformer encoder-decoder methods (e.g., VulRepair, TFix) has to do with our proposed

---

mechanism to help the repair model focus on vulnerable areas while generating the repairs. The decoders in VulRepair and TFix attend to each token embedding (including both vulnerable and benign tokens) encoded by encoders without explicitly learning the cross-match between vulnerable token embeddings and their repair token embeddings. On the other hand, we introduce vulnerability queries (VQs) and vulnerability masks (VMs) in our VQM to explicitly learn the cross-match between vulnerable token embeddings and their repair token embeddings. Our VQs and VMs help the decoder bias toward vulnerable token embeddings encoded by encoders during repair generation, hence leading to more accurate vulnerability repairs.

These results confirm that our proposed method of cross-matching vulnerability queries with vulnerable code tokens can help the model encode a more meaningful representation of a vulnerable function and decode the corresponding repair more accurately. In what follows, we provide a comprehensive ablation study of our proposed vulnerability queries and masks.

#### **(RQ2) What are the contributions of each component of our VQM approach?**

**Approach.** We introduce five variants of our approach to assess the effectiveness of our proposed vulnerability queries (VQs) and vulnerability masks (VMs) as follows:

- **Perfect Vulnerability Masking in Encoders and Decoders:** This method uses an identical architecture as our VQM, however, the perfect vulnerability masks (i.e., the exact location of each vulnerable token) are provided instead of predicted by a localization model.
- **Vulnerability Masks in Encoders:** This method only applies vulnerability masks on the self-attention output of each encoder to help the model focus more on vulnerable tokens when encoding the representations for a vulnerable function.
- **Vulnerability Masks in Decoders:** This method only applies vulnerability masks on the decoder cross-attention when cross-matching vulnerability queries and vulnerable code tokens to support the model to focus more on vulnerable tokens when generating repair tokens.
- **Without Vulnerability Masks:** This method is a plain transformer encoder-decoder architecture that applies no vulnerability masks.

Table 19: (Ablation results) The comparison between our proposed method and four other variants. Accuracy is presented in percentage.

Methods	Beam=1	Beam=3	Beam=5
Perfect Mask Enc + Perfect Mask Dec	33.76	44.31	46.88
Vul Mask Enc + Vul Mask Dec (ours)	<b>33.21</b>	<b>44.04</b>	46.06
Vul Mask Enc	32.75	43.49	<b>46.15</b>
Vul Mask Dec	32.84	43.85	45.69
w/o Vul Mask	29.82	39.72	43.67
with Vul Query Randomly Initialized	12.57	24.95	28.81
with No Bug Pre-trained Vul Mask	26.42	39.63	41.74

- **With Vulnerability Query Randomly Initialized:** This method applies vulnerability masks in both encoders and decoders while vulnerability queries are randomly initialized at the start of training.
- **With No Bug Pre-trained Vul Mask:** This method applies the vulnerability mask in both encoders and decoders while the vulnerability masks are only trained on the vulnerability repair dataset without pre-training on the bug-fix dataset.

In VQM, we train a separate model to predict VMs that apply to both encoders and decoders. In theory, the repair model should achieve better repair accuracy when applying more accurate VMs to its encoders and decoders. Thus, in the first variant, we aim to study whether leveraging perfect VMs (i.e., using the ground truths of vulnerability localization as VMs) can achieve better performance. Our proposed VMs can be applied to both the self-attention of encoders and the cross-attention of decoders. In particular, our VQM approach leverages VMs for both encoders and decoders. We further introduce three variants to study the effectiveness of VMs, i.e., (1) applying VMs to the self-attention of encoders; (2) applying VMs to the cross-attention of decoders; (3) without applying VMs; and (4) applying VMs trained only on the vulnerability repair dataset. Moreover, we proposed to initialize VQs based on the repair tokens in Section 7.2.3.2. We introduce a variant that randomly initializes VQs during training to compare with our proposed initialization method. In addition, the last variant is used to study the effectiveness of pre-training on the bug-fix corpus as described in Section 7.3.5.

We conduct the experiment using the dataset introduced in Section 7.3.3 and the same %PP measure as in RQ1 with  $beam \in [1, 3, 5]$ .

**Result.** The experimental results are shown in Table 19. Our approach to us-

---

Table 20: Compare our method with baselines, where all the methods are not pre-trained on the bug-fix data [23]. Accuracy is presented in percentage.

Methods	Beam=1	Beam=3	Beam=5
VQM	<b>5.32</b>	<b>8.81</b>	<b>9.72</b>
VulRepair	4.13	6.06	7.43
TFix	2.75	4.4	4.68
GraphCodeBERT	2.57	4.13	5.23
CodeBERT	1.56	2.29	2.75
VRepair	0.09	0.55	0.92
SequenceR	0	0	0

**ing vulnerability queries (VQs) along with vulnerability masks (VMs) inside both encoders and decoders achieves the best performance for  $beam \in [1, 3]$  and comparable performance for  $beam = 5$ .**

It can be seen that our proposed approach can achieve the best performance no matter the beam size when using the vulnerability localization ground truths as VMs (i.e., perfect masks). This result highlights the effectiveness of our proposed vulnerability masks. Moreover, applying the VMs is beneficial for both encoder self-attention and decoder cross-attention. It enhances the %PP by 2.93% when applied to encoders while gaining a %PP of 3.02% when applied to decoders, and the variants with VMs consistently outperform the variant without using any VMs. While applying the VMs on either encoders or decoders is beneficial, our proposed method to leverage the mask on both sides achieves better results for  $beam \in [1, 3]$  and comparable results for  $beam = 5$ .

Furthermore, “With No Bug Pre-trained Vul Mask” attains a beam 5 accuracy of 41.74%, registering a 4% reduction compared to our proposed method. These results underscore a crucial insight: the training process that encompasses the bug-fix dataset, featuring a broader spectrum of general bugs with a larger sample size, carries paramount significance.

In terms of the vulnerability queries (VQs), it can be seen that our approach to initialize the VQ embeddings based on repair tokens during training consistently outperforms the randomly initialized VQs. However, the random VQ embeddings method still outperforms baselines such as CodeBERT and GraphCodeBERT, highlighting the effectiveness of using vulnerability queries with cross-attention (as proposed in Section 7.2.3.2) for our vulnerability repair task.

---

Table 21: The %PP of our VQM approach across the top 25 most dangerous CWE-IDs in 2022. The %PP is shown based on the beam search results where Beam=5.

Rank	ID	%PP
1	CWE-787 (Out-of-bounds Write)	50% (13/26)
3	CWE-89 (SQL Injection)	100% (2/2)
4	CWE-20 (Improper Input Validation)	33% (24/72)
5	CWE-125 (Out-of-bounds Read)	42% (48/113)
6	CWE-78 (OS Command Injection)	33% (1/3)
7	CWE-416 (Use After Free)	31% (9/29)
8	CWE-22 (Path Traversal)	50% (1/2)
11	CWE-476 (NULL Pointer Dereference)	31% (11/36)
13	CWE-190 (Integer Overflow or Wraparound)	54% (19/35)
16	CWE-862 (Missing Authorization)	100% (1/1)
17	CWE-77 (Command Injection)	67% (2/3)
19	CWE-119 (Memory Corruption)	75% (223/296)
22	CWE-362 (Race Condition)	9% (3/34)
23	CWE-400 (Uncontrolled Resource Consumption)	55% (11/20)
	Average	55% (368/672)

In addition, the results shown in Table 20 indicate the effectiveness of pre-training on bug-fix corpus and correspond to the finding by Chen et al. [23] that the knowledge from the general bug fix corpus can be transferred to benefit the performance of AVR models. Our method still outperforms all baseline approaches.

## 7.5 Discussion

Our experiments have confirmed the performance advancement of our VQM approach over other AVR baseline approaches. In this section, we conduct further analysis to discuss whether our approach applies to repairing common vulnerabilities. Moreover, the data imbalance problem is common in vulnerability datasets [19, 20] where some vulnerability types (i.e., CWE-IDs) are common and easy to collect into the dataset while others are rare. Thus, we analyze the impact of imbalanced data frequencies across different CWE-IDs on our VQM approach. Last but not least, we analyze the impact of our vulnerability mask on repair decoders’ cross-attention to answer whether our vulnerability masks can truly help enhance the awareness of decoders’ vulnerability queries in vulnerable code areas as proposed in Section 7.2.3.

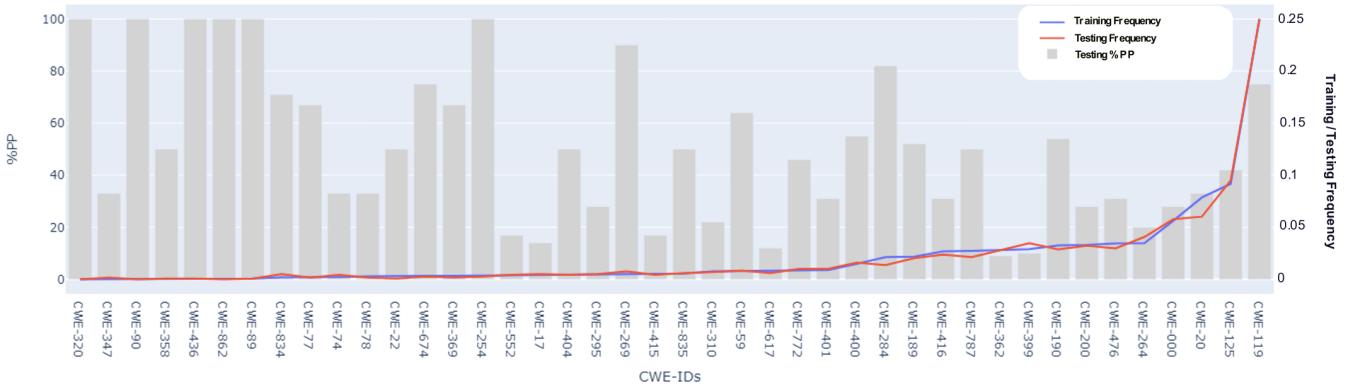


Figure 28: The performance analysis of our VQM accross different CWE-IDs. The bar chart represents the %PP while the blue line is the training frequency and the red line is the testing frequency across all vulnerability types. Note that the ticks of the Y axis on the left are for the %PP metric while those on the right are for the data frequency of each CWE-ID.

### 7.5.1 Can our VQM repair the common dangerous vulnerability types (i.e., CWE-IDs)?

To investigate whether our VQM approach can repair common dangerous real-world vulnerabilities, we evaluate our approach using testing data based on the 2022 CWE Top-25 Most Dangerous Software Weaknesses released by the CWE community [218]. Among the top 25 dangerous CWE-IDs, 14 of them are involved in our testing data. The results are presented in Table 21 with  $beam = 5$  (the repair model generates 5 repair candidates for each vulnerable function). We find that our approach can generate correct human-written repairs for 41% ( $\frac{87}{213}$ ) of the Top dangerous CWE-IDs. On average, our approach can correctly repair 55% ( $\frac{368}{672}$ ) of the vulnerable functions affected by the Top-25 most dangerous CWE-IDs, which is better than the average performance of our approach across all CWE-IDs (i.e., 45.14%). These results imply the potential applicability of our VQM approach which could be used to repair common vulnerabilities automatically.

### 7.5.2 How does our VQM perform across different vulnerability types that have different data frequencies in our experimental dataset?

We visualize the %PP across all CWE-IDs in our testing data as a bar graph to explore our VQM’s performance for different CWE-IDs. In addition, we show the frequency of each CWE-ID for both training and testing data as two line graphs to explore the relationship between the frequency of samples and the performance of our method. Note that the ticks of the Y axis on the left are for the %PP metric while those on the right are for the data frequency of each CWE-ID.

---

We found that the frequency of training and testing samples are not highly correlated with the performance of our method. This indicates that automated vulnerability repair (AVR) is a challenging problem in that high-frequency samples may not guarantee the repair model's performance.

As shown in Figure 28, the performance of our approach varies for each CWE-ID. Our approach performs well on some of the CWE-IDs that all testing samples can be correctly repaired. In particular, we found that our approach can achieve better accuracy for buffer-related errors such as CWE-119, CWE-190, CWE-787, and CWE-125, where our approach achieves 52% better than its average performance. To comprehensively assess the landscape, we also focus on two representative baseline methods, namely VulRepair and TFix, and evaluate their performances across various CWE-IDs. It is noteworthy that these baseline methods exhibit analogous characteristics to the performance trends of our approach. Specifically, both VulRepair and TFix showcase superior performance in the realm of buffer-related errors, surpassing their average performance by 53% and 47%, respectively.

These buffer-related CWE-IDs, which our approach and baseline methods can address with higher accuracy, are primarily centered around memory management concerns. These vulnerabilities encompass improper management of memory buffers, including arrays, strings, and other data structures, ultimately leading to potential memory corruption, crashes, or unauthorized memory access. Despite the criticality of these vulnerabilities and the complexity involved in identifying and mitigating them, they tend to arise from specific code segments and memory management practices.

These analyses collectively underscore the strides achieved by deep learning-based methodologies in advancing the proficiency of generating code repairs, particularly in the domain of buffer-related errors in C/C++ code.

In addition, we found that there are two vulnerability types (i.e., CWE-310 – Cryptographic Issues and CWE-552 – Files or Directories Accessible to External Parties) that can only be fixed by our VQM approach but not other baselines.

Compared to those buffer-related CWE-IDs, CWE-310 and CWE-552 address broader concerns related to cryptography and access controls. CWE-310 involves vulnerabilities related to cryptographic operations, which can be complex due to the intricacies of cryptography algorithms, key management, and proper

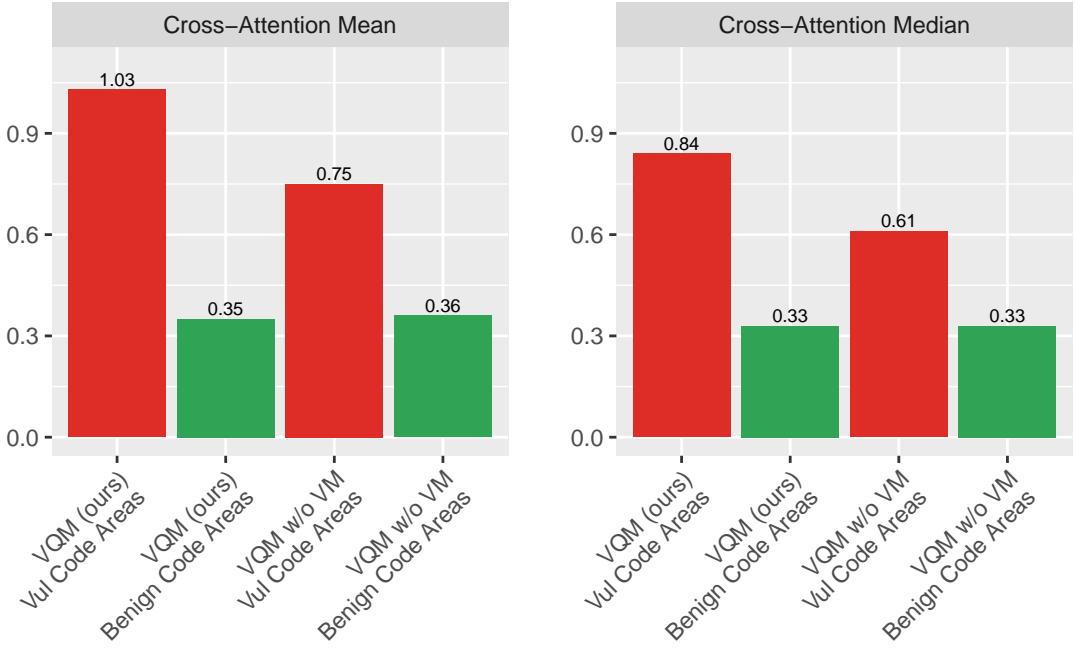


Figure 29: The comparison of the cross-attention weights of our VQM approach and the variant, VQM w/o VM. The red bars indicate the cross-attention weights for vulnerable code areas while the green bars indicate the cross-attention weights for benign code areas.

implementation of cryptographic functions. CWE-552 pertains to vulnerabilities related to the exposure of files or directories to external parties, which can involve complex access control mechanisms, permissions management, and overall system design. These findings underscore the potential of our proposed technique to comprehend code structures better, enhancing transformer encoders and decoders in generating accurate repairs for intricate cryptography and access control vulnerabilities.

Finally, it's worth noting that despite the introduction of our vulnerability query and masking techniques, certain vulnerabilities that are infrequent within our dataset pose a persistent challenge for accurate repair. For instance, our approach faced difficulties in effectively addressing vulnerabilities like CWE-444 (Inconsistent Interpretation of HTTP Requests) and CWE-285 (Improper Authorization). This highlights the challenge of automated vulnerability repair, demanding the model to address a wide spectrum of vulnerabilities, including those that manifest infrequently, thus obliging the model to glean insights from a limited dataset.

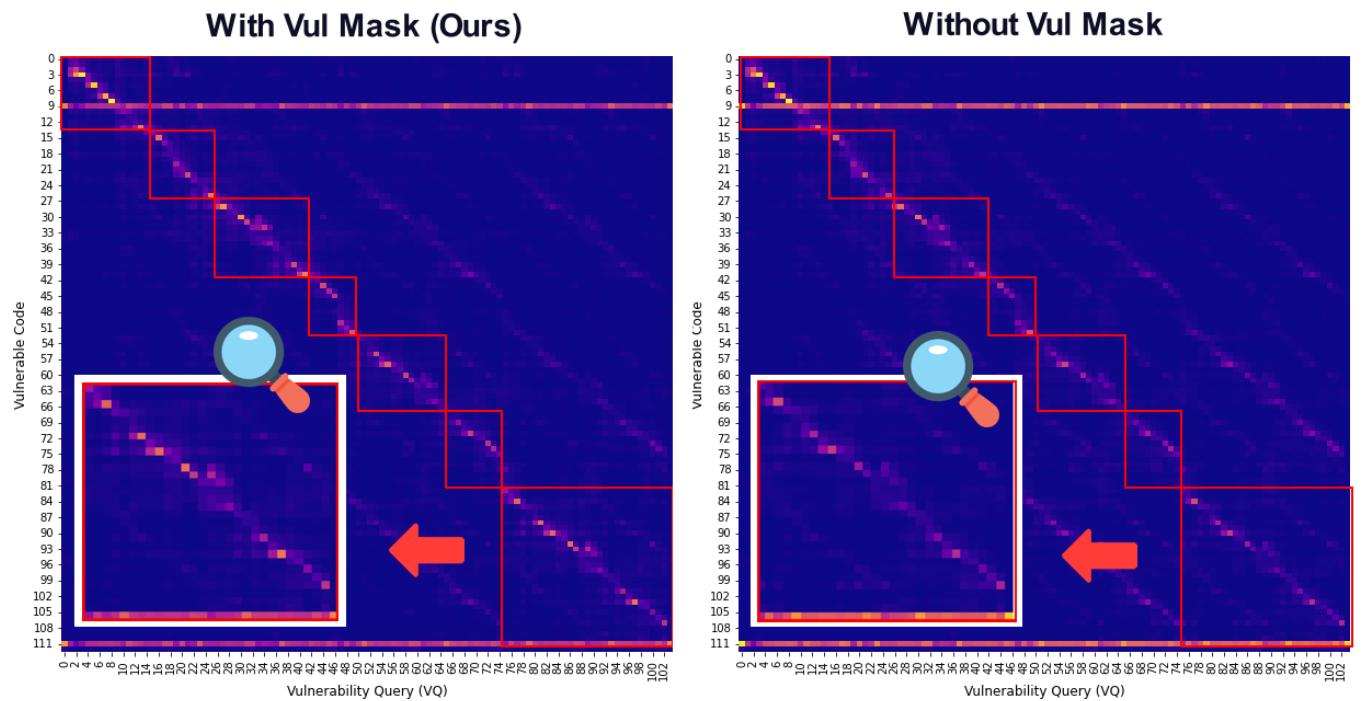


Figure 30: The visualization of the cross-attention weights between embeddings of generated repair patches (i.e., Vulnerability Queries - axis X) and embeddings of vulnerable functions (axis Y). The ground-truth vulnerable code areas are highlighted in red boxes. It can be seen that the cross-attention map is more highly activated for vulnerable code areas when applying vulnerability masks (VMs). In other words, our VMs help better distinguish the vulnerable and benign code areas.

---

### 7.5.3 Does our proposed vulnerability mask help highlight vulnerable code areas during vulnerability repair?

To investigate whether our proposed vulnerability mask (VMs) technique helps the repair decoders focus more on vulnerable code areas during repair generation, we analyze the cross-attention weights between the embeddings of vulnerable functions and embeddings of generated repair patches. We compare the cross-attention weights of our VQM approach with a variant without applying VMs (i.e., VQM w/o VM). We only remove the VM component to produce this variant while other components remain the same as our VQM. We analyze the cross-attention weights of each approach on correctly repaired testing samples.

The results are presented in Figure 29. We find that our VQM approach (applying VMs) has higher cross-attention mean (1.03 compared with the variant's 0.75) and median (0.84 compared with the variant's 0.61) weights for vulnerable code areas than the approach without applying VMs. Moreover, the cross-attention weights for benign code areas remain low and similar to the variant. For our VQM approach, the gaps between the cross-attention weights of the vulnerable code areas and benign code areas are 0.68 (1.03-0.35) and 0.51 (0.84-0.33) for mean and median, which is bigger than the variant's 0.39 (0.75-0.36) and 0.28 (0.61-0.33). These results confirm that our VMs can help our vulnerability queries in decoders focus more on vulnerable code areas and have better cross-attention contrast between vulnerable and benign code areas.

To demonstrate that our VMs can help decoders' cross-attention focus more on vulnerable code areas, we visualize the cross-attention maps of the two approaches, applying VMs (i.e., VQM) and without applying VMs. We visualize two vulnerable functions that are correctly repaired by both approaches in our testing set. We present the cross-attention visualization in Figure 30, where axis X is embeddings of the repair patches (i.e., Vulnerability Queries) while axis Y is embeddings of the vulnerable functions. Our approach with VMs has higher attention weights than the one without using VMs for the multi-scope (7 vulnerable code areas - the right part) vulnerable function. The visualization aligns with our analysis presented in Figure 29. This implies our VMs can help decoders focus more on vulnerable code areas when generating their corresponding repair codes through vulnerability queries, which may lead to better repair accuracy as demonstrated in our RQ1.

---

Table 22: (Discussion Results) The effectiveness of applying our vulnerability masks to a decoder-only transformer architecture.

Methods	Beam=1	Beam=3	Beam=5
GPT2-CSRC with Vulnerability Masks	<b>14.86</b>	<b>26.51</b>	<b>31.93</b>
GPT2-CSRC	11.93	19.27	24.77

#### 7.5.4 Can our proposed vulnerability mask be applied to decoder-only transformer architectures?

As our proposed vulnerability masks (VMs) are expressly designed to seamlessly integrate with the self-attention mechanism intrinsic to transformer models, the feasibility of their application to decoder-only transformers becomes evident. To explore the potential enhancement our VMs might confer upon decoder-only models, we proceed to integrate them with GPT2-CSRC [56], one of the prominent decoder-only program repair approaches. Notably, this integration involves applying our VMs to the self-attention mechanisms of the decoders, a strategy detailed in Section 7.2.3.3.

The experimental results are presented in Table 22. Evidently, the integration of our vulnerability masks (VMs) consistently yields performance enhancements in the context of the GPT2-CSRC approach. Noteworthy improvements are discernible across various beam widths, notably augmenting performance from 12% to 15% for  $beam = 1$ , from 19% to 27% for  $beam = 3$ , and from 25% to 32% for  $beam = 5$ . These results highlight the effectiveness of our proposed masking technique for the transformer’s self-attention mechanism.

## 7.6 A User Study of AI-generated Vulnerability Repairs

In addition to the performance evaluation of our approach in RQ1 and RQ2, we conducted a user study with 71 practitioners with software security backgrounds to evaluate the usefulness of AI-generated vulnerability repairs. We answered the following research question:

**(RQ3) Are AI-generated vulnerability repairs perceived as useful by software developers?** As we have comprehensively evaluated the performance in RQ1 and RQ2, it is essential to delve into the practical utility of AI-generated repairs for software practitioners, a vital aspect yet to be explored. To bridge this gap in knowledge, we diligently tackle RQ3 by conducting a user study tailored to assess the perceptions of security-aware software developers regarding AI-generated vulnerability repairs. This investigation seeks to unveil the practicality

The example presented in Part II-A	The AI-generated repair presented in Part II-B	
Example C Function	Vulnerable Function — CWE-787 (Out-of-bounds Write)	Repaired Function (The AI-generated Repair)
<pre> 1 1 static sk_sp&lt;SkImage&gt; unPremulSkImageToPremul (SkImage* input) { 2 2   SkImageInfo info = SkImageInfo::Make(input-&gt;width(), 3 3     input-&gt;height(), 4 4     kN32_SkColorType, kPremul_SkAlphaType); 5 5   RefPtr&lt;Uint8Array&gt; dstPixels = copySkImageData(input, 6 6     info); 7 7   if (!dstPixels) 8 8     return nullptr; 9 9   return newSkImageFromRaster( 10 10   info, std::move(dstPixels), 11 11   static_cast&lt;size_t&gt;(input-&gt;width()) * info.bytesPerPixel()); 12 12 } </pre>	<pre> 1 1 static sk_sp&lt;SkImage&gt; unPremulSkImageToPremul (SkImage* input) { 2 2   SkImageInfo info = SkImageInfo::Make(input-&gt;width(), 3 3     input-&gt;height(), 4 4     kN32_SkColorType, kPremul_SkAlphaType); 5 5   RefPtr&lt;Uint8Array&gt; dstPixels = copySkImageData(input, 6 6     info); 7 7   if (!dstPixels) 8 8     return nullptr; 9 9   return newSkImageFromRaster( 10 10   info, std::move(dstPixels), 11 11   static_cast&lt;size_t&gt;(input-&gt;width()) * info.bytesPerPixel()); 12 12 } </pre>	<pre> 1 1 static sk_sp&lt;SkImage&gt; unPremulSkImageToPremul (SkImage* input) { 2 2   SkImageInfo info = SkImageInfo::Make(input-&gt;width(), 3 3     input-&gt;height(), 4 4     kN32_SkColorType, kPremul_SkAlphaType); 5 5   RefPtr&lt;Uint8Array&gt; dstPixels = copySkImageData(input, 6 6     info); 7 7   if (!dstPixels) 8 8     return nullptr; 9 9   static_cast&lt;size_t&gt;(input-&gt;width()) * info.bytesPerPixel()); + 9   static_cast&lt;unsigned&gt;(input-&gt;width()) * info.bytesPerPixel()); 10 10 } </pre>

Figure 31: The left part shows the example vulnerable C function presented in Part II-A in our user study while the right part shows the example of AI-generated repair presented in Part II-B in our user study.

and real-world implications of our approach.

Following Kitchenham et al. [219], we conduct our study according to the following steps: (1) design and develop a survey, (2) recruit and select participants, and (3) verify data and analyze data. We explain the details of each step below.

### 7.6.1 Survey Design

**Step 1 – Design and development of the survey:** We designed our survey as a cross-sectional study where participants provided their responses at one fixed point in time. The survey consists of 8 closed-ended questions and 3 open-ended questions. For closed-ended questions, we use multiple-choice questions and a Likert scale from 1 to 5. Our survey consists of two parts: preliminary questions and participants’ perceptions of AI-generated software vulnerability repairs.

**Part I: Demographics.** The survey commences with a query, “(D1) What is your role in your software development team?”, to ensure that our survey captures responses from the intended target participants. Subsequently, the survey features a demographics question, “(D2) What is the level of your professional experience?”, aimed at ensuring a diverse distribution of responses across software practitioners with varying degrees of professional experience.

**Part II-A: Manual Vulnerability Repair.** To simulate a realistic vulnerability analysis scenario, we presented an example vulnerable C function to the participants as depicted in the left part of Figure 31. We then asked the participants to examine whether the function is vulnerable and propose a fix if required. Notably, we prepared ten different examples of vulnerable C functions which were spread equally to groups of participants to ensure that our survey is not biased toward a

---

specific vulnerable function.

Precisely, four inquiries were presented to the participants, commencing with “(Q1) Do you think the C function presented in Figure 1 is a vulnerable function or not?”; followed by “(Q2) Which line of code do you think is vulnerable?”; then “(Q3) Please suggest a fix to patch the vulnerable line.”; and concluded with “(Q4) How long did it take for you to identify whether the function is vulnerable and propose a fix to the vulnerable function (if required)?”.

**Part II-B: Participants’ Perception of AI-generated Vulnerability Repairs.** As illustrated by the experimental results presented in Section 7.4, our approach consistently attains the highest perfect repair accuracy, surpassing other baseline vulnerability repair methods by a significant margin. Given this demonstrated superiority, we selected our VQM approach as the representative AI-generated vulnerability repair method for this user study.

To assess the participants’ perception regarding AI-generated vulnerability repairs, we presented the repairs generated by our VQM approach as shown in the right part of Figure 31.

Precisely, five inquiries were presented to the participants, “(Q5) Do you think the AI-generated vulnerability repair by our approach is correct or not?”; followed by “(Q6) How do you perceive the usefulness of AI-generated vulnerability repairs?”; then “(Q7) Please justify your answer to Q6.”; then “(Q8) Would you consider adopting AI-generated vulnerability repair techniques if they are integrated into your software development IDEs (e.g., VSCode) for free with no conditions?”; and concluded with “(Q9) What is your expectation of AI-generated vulnerability repairs and how can we improve them?”

We employed Google Forms as the platform for our online survey administration. Each participant was greeted with a comprehensive introductory statement upon accessing the landing page. This statement elucidated the study’s objectives, rationale for participant selection, potential advantages and risks, and the commitment to safeguarding confidentiality. The survey was designed to be succinct, with an estimated completion time of around 15 minutes, and ensured complete anonymity for all respondents. Importantly, our survey underwent a rigorous evaluation process and received ethical approval from the Monash University Human Research Ethics Committee (MUHREC ID: 40251).

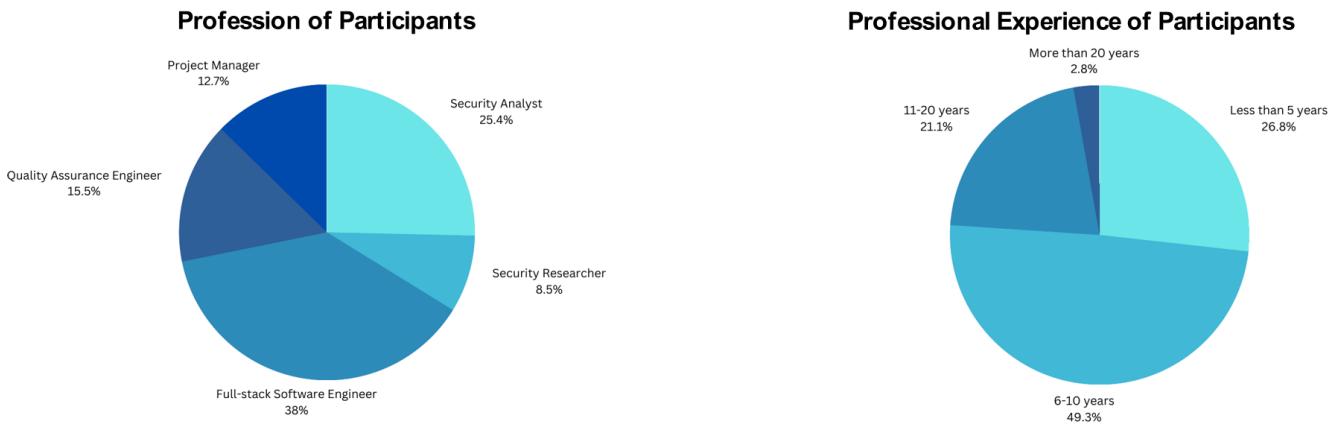


Figure 32: The demographics of our survey participants in terms of their profession and professional experience.

**Step 2: Recruit and select participants:** We recruited developers who have software engineering and/or software security expertise through LinkedIn and Facebook platforms. We received hundreds of Expressions of Interest in one week. We then randomly split our target audience into ten groups, where each group was given one distinct example vulnerable function while all of the questions were identical across all groups (as described in Part II-A). Finally, we obtained a total of 82 responses in one week.

**Step 3: Verify data and analyze data:** To ascertain the completeness of survey responses, particularly regarding open-ended questions, we conducted a thorough manual review. We filtered out 11 invalid responses (e.g., should any of the open-ended questions remain unanswered or if the responses are incomprehensible) out of a total of 82 responses. Thus, we included the remaining 71 responses for analysis. Closed-ended responses were quantitatively analyzed and presented using Likert scales through stacked bar plots. Additionally, we performed an in-depth manual analysis of open-ended question responses to gain a better understanding of participants' insights.

### 7.6.2 Survey Results

**Part I: Demographics.** Figure 32 presents the overall respondent demographic. In terms of the profession of the participants, 25% ( $\frac{18}{71}$ ) of them are security analysts, 9% ( $\frac{6}{71}$ ) of them are security researchers, 38% ( $\frac{27}{71}$ ) of them are full-stack software engineers, while the other 28% ( $\frac{20}{71}$ ) are software quality assurance engineers and software project managers. In terms of the level of their professional experience, 27% ( $\frac{19}{71}$ ) of them have less than 5 years of experience, 49% ( $\frac{35}{71}$ )

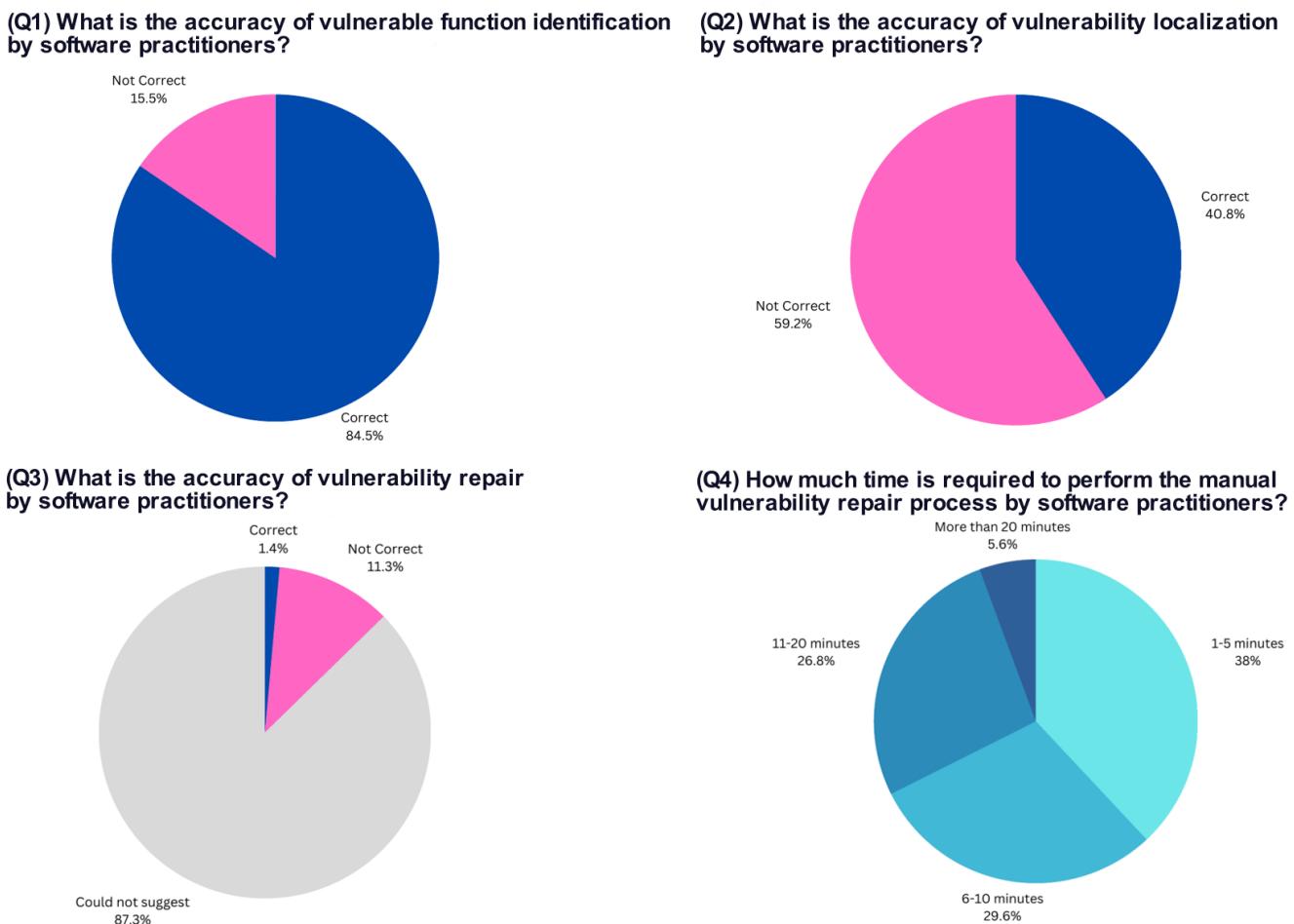


Figure 33: (Survey Results) A summary of the survey questions (i.e., Part II-A: Q1-Q4) and the results obtained from 71 participants.

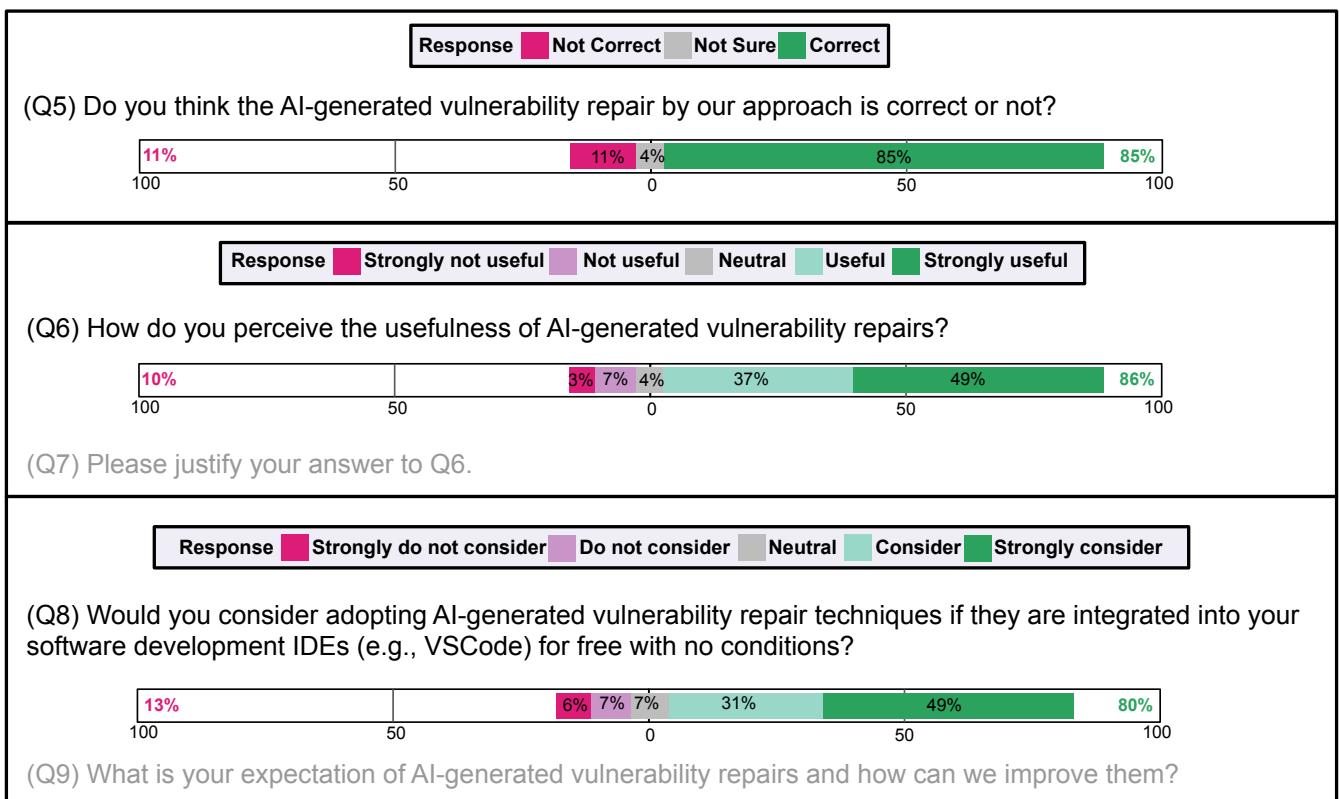


Figure 34: (Survey Results) A summary of the survey questions (i.e., Part II-B: Q5-Q9) and the results obtained from 71 participants.

---

have 6-10 years of experience, 21% ( $\frac{15}{71}$ ) have 11-20 years of experience, while the other 3% ( $\frac{2}{71}$ ) has more than 20 years of experience.

**Part II-A: Manual Vulnerability Repair.** Figure 33 summarizes the answers to (Q1)-(Q4) regarding manual vulnerability repair from the participants. In particular, 85% ( $\frac{60}{71}$ ) of them can correctly identify that the given function is vulnerable, 41% ( $\frac{29}{71}$ ) of them can correctly locate the vulnerable statement, while only 1% ( $\frac{1}{71}$ ) of them can suggest the correct repair. **Our findings highlight the complexity of vulnerability repair in contrast to the identification and localization of vulnerabilities.** A significant majority, accounting for over 87% of the participants, encountered challenges when tasked with proposing vulnerability repairs. Additionally, among those participants who did attempt to provide repair suggestions, most of them were incorrect.

In the context of identifying, locating, and suggesting repairs for vulnerabilities, 38% ( $\frac{27}{71}$ ) spent 1-5 minutes, 30% ( $\frac{21}{71}$ ) took 6-10 minutes, and 32% ( $\frac{23}{71}$ ) required over 10 minutes to complete these tasks. **This highlights the time-consuming nature of vulnerability repair, with 62% of participants requiring over 10 minutes to address a single vulnerable function containing 5-15 statements.**

**Part II-B: Participants' Perception of AI-generated Vulnerability Repairs.** Figure 34 summarizes the answers to (Q5)-(Q9) regarding participants' perception of AI-generated vulnerability repairs provided by our approach. **As presented in (Q5) results, the majority of participants demonstrated the ability to accurately assess the correctness of AI-generated vulnerability repairs, with 85% ( $\frac{60}{71}$ ) providing correct evaluations.** However, 11% ( $\frac{8}{71}$ ) of participants failed to correctly identify the accuracy of the repairs, while 4% ( $\frac{3}{71}$ ) expressed uncertainty regarding the correctness of the AI-generated repairs.

Participants have actively engaged in the manual vulnerability repair process encompassing questions (Q1) to (Q4). In (Q5), we simulate the vulnerability repair workflow involving AI tools, wherein users are required to evaluate the correctness of AI-generated repairs before implementing them. The outcomes of (Q5) provide further evidence of the participants' proficiency in assessing AI-generated repairs. Hence, we proceed to evaluate their perspective on the usefulness of AI-generated vulnerability repairs, drawing from their professional expertise in vulnerability repair.

**(Q6) How do you perceive the usefulness of AI-generated vulnerability re-**

---

pairs?

**Findings.** 86% of the participants perceived that the vulnerability repairs generated by our approach are useful due to various reasons stated in (Q7):

- **Efficiency & Productivity** – R8 (a security analyst with 6-10 years of experience): *The AI systems can quickly and efficiently scan large code bases for vulnerabilities and can also identify vulnerabilities that human developers may miss*; R56 (a security analyst with 6-10 years of experience): *AI-generated vulnerability repairs can enhance the efficiency and effectiveness of security teams by automating the identification and resolution of vulnerabilities, allowing them to focus on more complex security tasks and reducing the time required to patch potential weaknesses*; R22 (a full-stack software engineer with 6-10 years of experience): *Its automation and data analysis saves time and effort, boosting productivity. It also speeds up scientific discovery and development in various fields*; and R12 (a full-stack software engineer with 6-10 years of experience): *Throughout my seven years of experience, AI-generated vulnerability repairs have been very useful as they help a lot. It helps to reduce the backlog of vulnerabilities that need to be fixed, and it can also help to ensure that vulnerabilities are fixed quickly*.
- **Timeliness & Scalability** – R29 (a full-stack software engineer with less than 5 years of experience): *AI can rapidly identify and respond to vulnerabilities, which is critical in a constantly evolving threat landscape* and R41 (a security researcher with less than 5 years of experience): *AI can handle large and complex systems, making it an effective tool for identifying and addressing vulnerabilities at scale, which may be challenging for human teams*.
- **Applicability** – R18 (a security analyst with 6-10 years of experience): *The repaired function is a good one, and it shows and indicates some compatibility with the vulnerable functions. In this way, adopting AI-generated vulnerability repair plays an important role in ensuring proper functioning for greater results outcomes* and R20 (a full-stack software engineer with 11-15 years of experience): *Useful in the sense that it allows for more successful and quick corrections. Secondly, it provides a fast learning system for developers*.

On the other hand, we observed that over 12% ( $\frac{9}{71}$ ) of participants expressed apprehensions about human involvement or potential ethical considerations, de-

---

spite recognizing the usefulness of AI-generated repairs:

- R32 (a security analyst with 6-10 years of experience): *In summary, AI-generated vulnerability repairs can be a valuable aid in the security process, but they should be used alongside human expertise and thorough testing to ensure the effectiveness and security of the repairs;*
- R33 (a security analyst with 6-10 years of experience): *However, it's important to exercise caution with AI-generated repairs. While AI algorithms can suggest potential fixes, they may not always produce the most effective or secure solutions. Human expertise is still necessary to validate and test the proposed repairs, ensuring that they don't introduce new vulnerabilities or have unintended consequences;*
- R40 (a security analyst with 6-10 years of experience): *AI-generated vulnerability repairs can be a valuable tool in cybersecurity, offering speed and scalability. However, they should be part of a broader security strategy that includes human expertise, ongoing monitoring, and ethical considerations;*
- R43 (a full-stack software engineer with 6-10 years of experience): *AI-based vulnerability repairs have the potential to be highly useful in enhancing cybersecurity, especially in terms of speed. However, Human security professionals should provide oversight, review AI-generated repairs, and make critical decisions.*

**(Q8) Would you consider adopting AI-generated vulnerability repair techniques if they are integrated into your software development IDEs (e.g., VSCode) for free with no conditions?**

**Findings.** **80% of the participants expressed a willingness to embrace AI-generated vulnerability repair techniques if they are readily available and free of charge.** Furthermore, the participants' expectations regarding AI-generated vulnerability repairs, as indicated in their responses to (Q9), can be summarized as follows:

- **Accuracy** – R4 (a security analyst with 6-10 years of experience): *My expectation of AI-generated vulnerability repairs is that they will become more advanced and effective in detecting and patching vulnerabilities in software. To improve them, we can focus on enhancing the accuracy and speed of vulnerability detection algorithms, ensuring compatibility with different software*

---

*development environments, and continuously updating the AI models with the latest threat intelligence and R5 (a security analyst with 6-10 years of experience): Continuous learning and adaptation: AI models should be regularly updated and fine-tuned based on feedback and new security insights to enhance their accuracy and effectiveness.*

- **Data Integrity** – R69 (a security analyst with 6-10 years of experience): *To improve the quality of AI-generated repairs, I think it will be important to have robust datasets that cover a wide range of vulnerabilities and potential solutions.*
- **Availability** – R28 (a security analyst with 6-10 years of experience): *It needs to be integrated into more reachable software.*
- **Explainability** – R33 (a security analyst with 6-10 years of experience): *Transparency and explainability: Providing clear explanations of how AI-generated repairs are generated can help developers understand and trust the suggested fixes; R42 (a quality assurance engineer with less than 5 years of experience): AI-generated vulnerability repairs should provide clear explanations for their decisions, allowing human operators to understand and trust the recommendations.; and R53 (a full-stack software engineer with more than 20 years of experience): In my opinion, AI-generated vulnerability repairs have a lot of potential, but there are some challenges that need to be addressed before they can be widely used and trusted. One challenge is ensuring the accuracy of the repairs. Another challenge is the need for explainability and transparency. Currently, many AI systems are "black boxes," meaning that it's difficult to understand how they arrive at their decisions. I think improving explainability and transparency will be key to building trust in the technology.*

**Survey Summary.** Our survey study with 71 software practitioners provides valuable insights into the perception of AI-generated vulnerability repairs. In particular, 86% ( $\frac{61}{71}$ ) of the participants found these repairs useful attributing their value to increased efficiency, productivity, timeliness, scalability, and broad applicability. However, experienced security analysts voiced concerns regarding the potential ethical implications and the need for human oversight. Furthermore, participants expressed expectations for improvement, emphasizing the importance of enhancing repair model accuracy, data integrity, method availability (e.g., integrating it into common software development IDEs), and the transparency and explainability of AI models. These findings underscore the poten-

---

tial and importance of AI-generated vulnerability repairs in the software security landscape while highlighting areas for further development and refinement.

## 7.7 Related work

Machine learning (ML)-based techniques have been proposed to automate various software engineering-related tasks such as agile planning [103], code review [203, 220–223], code completion [224], defect prediction [74, 94, 102], and test case generation [225]. In particular, ML-based vulnerability prediction approaches have also been proposed to help security analysts predict vulnerabilities [1, 108, 226–228], explain their vulnerability types [2], estimate their severity [5], and recommend repair patches [3, 23].

In this work, we focus on the **Automated Vulnerability Repair (AVR)** task that uses machine learning models to generate repair patches for vulnerable C/C++ functions. In particular, our AVR task shares similarities with the widely recognized Automated Program Repair (APR) task, yet it stands apart through two distinct aspects. Firstly, the AVR task is notably more domain-specific, directed towards addressing vulnerabilities rather than general defects. Secondly, instead of generating complete repaired functions as output, the AVR task requires models to generate repair patches that exclusively address the vulnerable code regions within vulnerable functions. This design curtails output length and alleviates the model’s burden of generating extensive sequences, thereby optimizing the repair process. It is worth noting that the general program repair shares a similar nature to vulnerability repair where defective functions only consist of a few defective code statements that need to be repaired. Thus, our AVR configuration and the methodology of VQM have the potential for adaptation to tackle the APR task that focuses on general bug fixing. It is important to acknowledge that vulnerability repair resides within the larger domain of program repair. Nevertheless, in this work, our focus remains dedicated to the specialized domain of vulnerability repair.

RNN-based models such as SequenceR [22] have been proposed to encode the vulnerable programs and decode corresponding repairs sequentially. SequenceR used Bi-LSTMs as encoders with unidirectional LSTMs to generate repairs. Recently, attention-based Transformer models have been leveraged in the AVR domain, which was shown to be more accurate than RNNs. For instance, VRepair [23] relied on an encoder-decoder Transformer with transfer learning using the bug-fix data to boost the performance of the vulnerability repair on C/C++

---

programs. SeqTrans [35] constructed code sequences by considering data flow dependencies of programs and leveraged an identical architecture as VRepair. In addition, Berabi et al. [210] proposed to use a T5 model pre-trained on natural language corpus (i.e., T5-large [216]) to fix JavaScript programs and Fu et al. [3] utilized a T5 model pre-trained on source code (i.e., CodeT5 [132]) to repair C/C++ programs. Mashhadi et al. [205] applied the CodeBERT [40] model to repair Java bugs. Those large pre-trained language models have demonstrated strong improvement over RNNs and non-pretrained transformers because the pre-training steps help the models gain better initial weights for the vulnerability repair downstream task than training from scratch. On the other hand, DL-Fix [207] and CURE [36] were proposed to generate vulnerability repairs that satisfy test cases. Thus, complete repaired functions are required to train and evaluate models and the problem statement is different from ours described in Section 7.2.2. Different from the sequence-based methods mentioned above, Dinella et al. [229] proposed to learn the graph transformation based on the Abstract Syntax Tree (AST) of source code, which used GNNs to represent the program and LSTMs to generate repairs for JavaScript programs.

Previous approaches mainly focus on leveraging seq2seq models for the AVR task. In particular, transformer-based methods have achieved promising performance. As illustrated in Section 7.2.2, a vulnerable function usually consists of only a few vulnerable code areas that cause the vulnerability. Nevertheless, existing transformer-based approaches lack a mechanism to capture those vulnerable code areas during the repair generation. Thus, we propose a mechanism to help decoders focus more on vulnerable code areas during the repair. Specifically, we extend the ViT-based approaches for object detection (e.g., DeTR [211]) and build our own vulnerability repair approach with vulnerability queries and masks to guide decoders to focus more on vulnerable code areas during vulnerability repairs.

## 7.8 Threats to Validity

Similar to other empirical studies related to deep learning models, there are various threats to the validity of our results and conclusions.

*Threats to internal validity* relate to the stochastic gradient descent process to update network weights for deep learning models during training. To mitigate this threat, we explicitly set the random seeds to ensure reproducibility and report the hyperparameter settings in the replication package to support future replication

---

studies. In addition, we repeated our main experiment five times with different random seeds to ensure the soundness of our experimental results and findings.

We acknowledge another internal threat to the validity of our experiments arising from the assumption that our AI-based vulnerability repairs operate on confirmed vulnerable functions. While this assumption simplifies our experimental setup, it may not fully represent the complexity of real-world scenarios where identifying vulnerabilities is an integral part of the process. However, the vulnerability detection process is beyond the scope of this work. Thus, it is important to consider this limitation when interpreting the results of our experiments. Finally, it is paramount for future research endeavors to develop an end-to-end pipeline that encompasses vulnerability detection.

*Threats to external validity* relate to whether our VQM approach can be generalized to other vulnerabilities and projects not included in our studied dataset, and programming languages other than C/C++. Our approach is evaluated on the dataset provided by Chen et al. [23] consisting of CVEFixes [195] and Big-Vul [70] vulnerability corpus. While our studied dataset includes various vulnerabilities written in C/C++ across different software projects, our VQM approach does not necessarily generalize to other data and programming languages. Since our approach is not programming language-specific, it could be trained on other vulnerabilities written in programming languages other than C/C++ or from other projects without any modification of our approach. Thus, other datasets can be explored in future work.

## 7.9 Summary

In this work, inspired by VIT-based object detection approaches in the computer vision domain, we have introduced a new AVR method named VQM to enhance awareness and attention to vulnerable code areas in a vulnerable function for producing better repairs. In our repair model, we cross-match vulnerability queries (VQs) and their corresponding vulnerable code areas and their corresponding repairs through the cross-attention mechanism. To strengthen such cross-matchings and guide decoders to pay more attention to vulnerable code areas, we propose to learn a vulnerability mask (VM) and incorporate it into the cross-attention. Additionally, we apply our VMs in the self-attention of encoders to guide our model to focus more on vulnerable code areas when learning the embeddings of a vulnerable function. Through an extensive evaluation of 5,417 real-world vulnerabilities, our experiment confirms the advancement of our approach

---

over all of the baseline AVR approaches. Moreover, the additional analysis of our experimental results highlights the applicability of our VQM approach, which can accurately repair common dangerous vulnerabilities. Last but not least, our survey study with 71 software practitioners highlights the significance and usefulness of AI-generated vulnerability repairs in the realm of software security.

---

## **8 AIBugHunter – A Practical Vulnerability Analysis Tool for Locating, Explaining, Estimating, and Repairing Software Vulnerability**

© 2024 Springer Nature Switzerland AG. This is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive version was published in Empirical Software Engineering, **29**(1), 4, published 20 November 2023, <https://doi.org/10.1007/s10664-023-10346-3>.

---

## 8.1 Introduction

Software vulnerabilities are weaknesses in an information system, security procedures, internal controls, or implementations that could be exploited or triggered by a threat source [59]. Such unresolved weaknesses result in extreme security or privacy risks. According to the research conducted by WhiteSource [230] on open source vulnerabilities in the past 10 years (including multiple sources like the National Vulnerability Database (NVD), security advisories, GitHub issue trackers, etc.), C has the highest number of vulnerabilities out of all seven reported languages (i.e., C, PHP, Java, JavaScript, Python, C++, Ruby), accounting for 47% of all reported vulnerabilities. Buffer errors (e.g., CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer) are the most common vulnerabilities in C and C++. It is worth noting that this group of vulnerabilities related to memory corruption could often have critical consequences such as system crashes and sensitive information disclosure. In particular, our proposed software vulnerability classification approach can correctly identify 79% of the CWE-119 buffer error as shown in Table 24 (see Rank 17).

Recently, the shift-left testing concept (i.e. move software testing earlier in project timelines) has been proposed to try to perform software testing at earlier stages of development, instead of testing applications during late phases of development. Thus, vulnerabilities related to fundamental features, such as buffer errors, could ideally be found and fixed earlier. DevSecOps has also been proposed to extend the idea of DevOps by integrating security into DevOps initiatives [231]. DevSecOps aims to examine application security from the start of development by automating some security gates and selecting the right tools to continuously integrate security in the DevOps workflow. For example, program analysis(PA)-based tools can be integrated into IDEs, such as Visual Studio Code (VS Code), to detect such vulnerabilities during coding. However, these methods usually rely on pre-defined vulnerability patterns and struggle to detect specific types of vulnerability. Croft et al. [30] demonstrated that Machine Learning(ML)-based techniques are more accurate than PA-based tools in detecting file-level vulnerabilities. Our own previous study showed that our ML-based LineVul approach is more accurate than the PA-based Cppcheck tool [28] on line-level vulnerability prediction [1, 74, 94]. ML-based methods learn vulnerability patterns based on historical vulnerability data instead of relying on pre-defined patterns. Thus, ML-based approaches can capture more kinds of vulnerabilities and be more easily extended as new vulnerabilities emerge. PA-based tools, such as Checkmarx [29], have been integrated into software development workflow to support

---

security diagnosis during development. However, to date, ML-based tools have not been integrated as security tools to help detect security issues during software development.

In this work, we propose an ML-based vulnerability analysis tool, AIBugHunter, to bridge the critical gap between ML-based security tools and software practitioners. AIBugHunter is integrated into a modern IDE (i.e., VS Code) – to fulfill the concept of shift-left testing and to support real-time security inspection during software development. In particular, given developers' source code written in C/C++, our AIBugHunter can (1) locate vulnerabilities, (2) classify vulnerability types, (3) estimate vulnerability severity, and (4) suggest repairs. We integrate our previous work LineVul [1] and VulRepair [3] for AIBugHunter to achieve automated vulnerability localization and repairs. In this work, we further propose a multi-objective optimization (MOO)-based approach to optimize the multi-task learning scenario and help our AIBugHunter accurately identify vulnerability types (i.e., CWE-IDs, and CWE-Types) and explain the detected vulnerabilities. In addition, a transformer-based approach is proposed to help AIBugHunter estimate the vulnerability severity (i.e., CVSS Score) which could be beneficial for the prioritization of security issues.

We evaluate our proposed MOO-based vulnerability classification and severity estimation approaches on a large dataset that consists of 188k+ C/C++ functions including various vulnerability types and severity. We found that our MOO-based vulnerability classification approach outperforms other baseline methods and achieves the accuracy of 65% (demonstrated in RQ1) and 74% (demonstrated in RQ2) for classifying CWE-ID and CWE-Types respectively. In addition, our transformer-based severity estimation approach outperforms other baseline methods and achieves the best mean squared error (MSE) and mean absolute error (MAE) measures (demonstrated in RQ3). We evaluate our AIBugHunter via qualitative evaluations including (1) a survey study to obtain software practitioners' perceptions of our AIBugHunter tool; and (2) a user study to investigate the impact that our AIBugHunter could have on developers' productivity in security aspects. Our survey study shows that predictions provided by AIBugHunter are perceived as useful by 47%-86% of participated software practitioners and 90% of participants will consider adopting our AIBugHunter. Moreover, our user study indicates that AIBugHunter could save developers' time spent on security analysis that could potentially enhance security productivity during software development (demonstrated in RQ4).

---

The main contributions of this work include:

1. AIBugHunter, a novel ML-based software security tool for C/C++ that is integrated into the VS Code IDE to bridge the gap between ML-based vulnerability prediction techniques and software developers and achieve real-time security inspection;
2. A quantitative evaluation of AIBugHunter on a large dataset showing its high precision and recall;
3. A qualitative survey study of AIBugHunter with 21 software practitioners demonstrating both its practicality and potential acceptance;
4. A qualitative user study of AIBugHunter with 6 software practitioners demonstrating AIBugHunter could enhance practitioners' productivity in combating security issues during software development;
5. A multi-objective optimization approach for vulnerability classification that optimizes the multi-task learning scenario for classifying the vulnerability types for vulnerable functions written in C/C++; and
6. A transformer-based approach to estimate vulnerability severity for vulnerable functions written in C/C++.

We make available our datasets, scripts including data processing, model training, model evaluation, and experimental results related to our approach in a GitHub repository: (<https://github.com/awsm-research/AIBugHunter>). Additionally, AIBugHunter is available at Visual Studio Code marketplace (<https://marketplace.visualstudio.com/items?itemName=AIBugHunter.aibughunter>).

## 8.2 AIBugHunter: Our Approach

We provide an overview of our AIBugHunter, an ML-based vulnerability prediction tool as a plug-in in Visual Studio Code (VS Code). The main purpose of our AIBugHunter is to bridge the gap between ML-based vulnerability prediction techniques and software developers by providing a security plug-in in IDE to present more security information during software development.

### 8.2.1 AIBugHunter security tool

As a security tool integrated into VS Code, AIBugHunter first scans the file opened by developers and parses the whole file into multiple separate functions. For

The screenshot shows a code editor window for a file named 'paper.cpp'. A tooltip is displayed over line 9 of the code, highlighting a potential vulnerability. The tooltip contains the following text:

```

C paper.cpp > unPremulSkImageToPremul(SkImage *)
1 static sk_sp<SkImage> unPremulSkImageToPremul(SkImage* input) {
2 SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
3 kN32_S [Severity: High (7.11)] Line 9 may be vulnerable with CWE-787 (Out-of-bounds Write | Abstract
4 RefPtr Type: Base) AIBugHunter(More Details)
5 if (!d
6 return The software writes data past the end, or before the beginning, of the intended buffer. (More Details)
7 return View Problem Quick Fix... \(Ctrl+.\) 5
8 info,
9 static_cast<size_t>(input->width()) * info.bytesPerPixel());

```

The status bar at the bottom of the editor also displays the same vulnerability information. The UI is annotated with numbers 1 through 5, corresponding to the following elements:

- 1: The number '1' is located next to the status bar message 'Line 9 may be vulnerable with CWE-787 (Out-of-bounds Write | Abstract Type: Base)'.
- 2: The number '2' is located next to the status bar message 'Severity: High (7.11)'.
- 3: The number '3' is located next to the status bar message 'AIBugHunter([More Details](#))'.
- 4: The number '4' is located next to the status bar message '1 of 2 problems'.
- 5: The number '5' is located next to the 'Quick Fix... (Ctrl+.)' button in the tooltip.

Figure 35: The user interface of our AIBugHunter.

each function, our AIBugHunter performs the following 4 steps:

1. Localize the vulnerable lines (LineVul);
2. Classify the vulnerability types (proposed in this work);
3. Estimate the vulnerability severity (proposed in this work); and
4. Suggest the repair patches (VulRepair).

where LineVul [1] locates vulnerable lines; our approach predicts types and severity; and VulRepair [3] suggests repairs.

In AIBugHunter, we use LineVul and VulRepair from our previous works. These models were trained using the extensive Big-Vul dataset offered by Fan et al. [70] and the CVEFixes dataset provided by Bhandari et al. [195]. We illustrate both of them as follows:

LineVul is among the first to predict line-level vulnerabilities using the transformer model and its self-attention mechanism. Given a C/C++ function as input, first, LineVul leverages a BPE tokenizer to tokenize the function into subword tokens and mitigate the out-of-vocab problem. Second, LineVul leverages transformer encoders [69] to learn the representation of those tokens, which can better tackle the long-term dependencies among tokens than previously proposed RNN-based methods [32]. Third, LineVul uses a linear classification head to predict function-level vulnerability prediction based on the learned representations. LineVul uses intrinsic model interpretation to localize line-level vulnerabilities. In particular,

---

LineVul summarizes the self-attention scores of each line in the function and ranks the line scores to place potentially vulnerable lines on the top. Our previous work [1] has demonstrated that LineVul achieves the best accuracy for both function-level and line-level vulnerability prediction and is the most cost-effective approach to localize line-level vulnerabilities when compared with other baseline methods.

VulRepair is among the first to leverage a large pre-trained language model for the automated vulnerability repair (AVR) problem. Given a vulnerable C/C++ function as input, instead of using word-level tokenization as previous work [23], VulRepair leverages a BPE tokenizer to tokenize the function into subword tokens and address the potential OOV problem. VulRepair uses a pre-trained encoder-decoder T5 architecture where encoders encode the representation of the vulnerable function and decoders generate the corresponding repair patches. In particular, the relative position encoding of T5 used by VulRepair improves the absolute position encoding of the vanilla transformer used in previous work [23]. VulRepair was evaluated using the human-written repairs as ground-truth labels where a repair generated by VulRepair is considered correct if it is identical to the labels. Our previous work [3] has demonstrated that VulRepair substantially improves the performance of previous works for the AVR problem.

### 8.2.2 Example Usage

Consider the situation where an opened file contains one function written in C++, shown in Fig 35. This example uses a real-world "out-of-bounds write" vulnerability [66] that is considered the most dangerous vulnerability in 2021 [218]. Fig 35 shows the "*unPremulSkImageToPremul*" function. AIBugHunter has analyzed this and considered it as a vulnerable function. This is due to the variable type "size\_t" being misused, causing an "out-of-bounds write" vulnerability (i.e., CWE-787) at line number 9.

As shown in Fig 35, AIBugHunter first takes the whole function as an input and sends it to its backend models, LineVul [1]. The LineVul algorithm identifies that the 9th line of the "*unPremulSkImageToPremul*" function is a vulnerable line, as annotated by ①. Our approach further classifies this function as a vulnerability of CWE-787, shown as ②, with a Base type shown in ③.

This function is predicted as being of a high severity with a CVSS score of 7. This is shown as ④. Finally, we use our backend tool, VulRepair [3], to generate repair

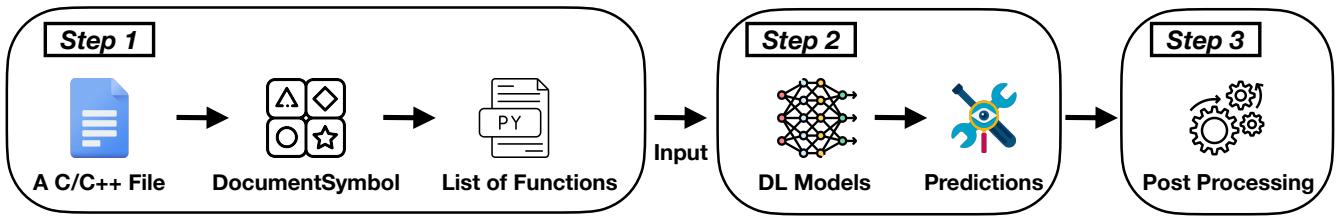


Figure 36: The back-end implementation of our AIBugHunter.

patches. This patch will be used to replace the vulnerable line. The developer can select this option by clicking on the "Quick Fix" button, shown as ⑤.

### 8.2.3 AIBugHunter Implementation

We developed our AIBugHunter extension using the VS Code Extension API provided by Microsoft to gain symbol information and utilize other VS Code IDE features. AIBugHunter is mainly written in TypeScript following the boilerplates provided by the VS Code extension generator. Being a plain VS Code extension, our package's operations are driven by a Node.js engine. In what follows, we introduce the front-end and back-end implementation details of AIBugHunter.

#### 8.2.3.1 Front-End Implementation

The UI elements of AIBugHunter are defined by the VSCode API provided by Microsoft, the backend of which is Node.JS, and is interacted using the Type-Script language. When a user opens a C/C++ file, AIBugHunter extracts each function from the source code using the “symbols” information available through VSCode API, and builds a list of parsed functions to be passed into the DL models introduced in the following section. The back-end will return the generated predictions using the API provided, and the relevant information is displayed on the UI as “diagnostics”. This enables the extension to indicate the specific line to fix using underlines, display hover messages, provide a link to the CWE page, provide a “Quick Fix” button for repair candidates, and offer other error messages in the interface. AIBugHunter presents its vulnerability predictions and explanatory information as shown in Fig 35.

---

### 8.2.3.2 Back-End Implementation

The back end consists of three main steps as summarized in Fig 36. First, the data preparation step is to construct data for DL models. Second, the DL models inference step for (1) locating line-level vulnerabilities, (2) classifying vulnerability types, (3) estimating vulnerability severity, and (4) suggesting repairs. Third, the post-prediction processing step is used to prepare information and present it in the UI.

**Step 1: Data Preparation.** When a C/C++ file is opened, VSCode automatically analyzes it and generates a “DocumentSymbol”, which is a collection of symbols in the document such as variables, classes, and functions. We preserve only the collection of functions to construct a list of functions parsed from the document, where each parsed function undergoes formatting to remove comments. Note that all the modifications are recorded as a position delta to correctly map the prediction results to the original code.

**Step 2: DL Model Inference.** The model inference consists of two steps to obtain all the predictions to present in the front end as described below:

**Step 2a.** Send the list of functions from the data preparation step to the line-level vulnerability detection model’s inference API endpoint (or flag in local inference mode). This will return a JSON that tells if individual functions are vulnerable or not (binary), and scores on each line of the function that determine which line the modifications are required to fix the vulnerability.

**Step 2b.** For functions that were predicted vulnerable in the previous step are now sent to three additional DL models. For each function, the first model will return a CWE-ID indicating the vulnerability type; the second model will return a CVSS score indicating the severity; and the third model will return an annotated piece of “patch code” as suggested repairs.

**Step 3: Post-Prediction Processing.** All the predictions from the model inference step are processed according to the user configuration. Additionally, for functions predicted as vulnerable, we fetch the vulnerability description from MITRE ATT&CK [43] based on the predictions to provide in-depth details of the predicted vulnerability and an accessible link to the official page of the specific CWE-ID. Finally, the organized information is displayed on the interface via the VSCode extension API.

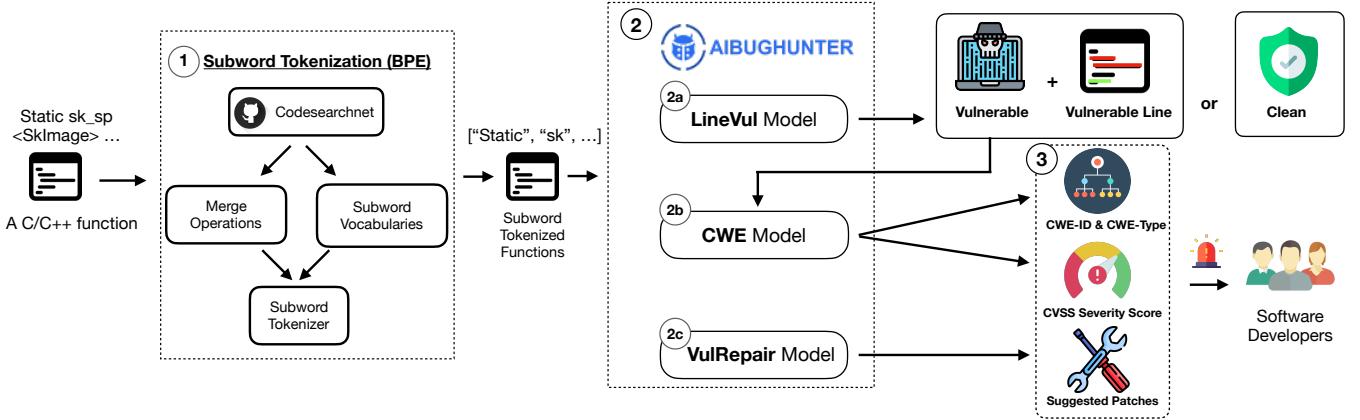


Figure 37: An overview architecture of our approach.

### 8.3 Learning to predict vulnerability type and severity

Our approach is a vulnerability prediction framework consisting of three different inference tasks. As shown in Fig 37, given a C/C++ function, we first tokenize raw input into code tokens through Byte Pair Encoding (BPE) in Step ①. In Step ②, the tokenized function is then input to a LineVul model proposed in our previous work [1] to predict vulnerable lines in the input function. If vulnerable lines exist in the function, our approach further predicts vulnerable types (i.e., CWE-ID and CWE-Type) and severity (i.e., CVSS severity score) of the vulnerable function as shown in step ②b. Furthermore, the vulnerable function is also input to the VulRepair [3] model to generate suggested repair patches as shown in step ②c. Finally in Step ③, AIBugHunter integrates the predictions from LineVul, our approach, and VulRepair models and present them to software developers in the IDE. We refer readers to our previous work [1] for more technical details about BPE tokenization and the Transformer architecture of our approach.

In this section, we introduce key new components in our AIBugHunter approach over our prior works. Given a vulnerable function, we aim to predict its vulnerability types, where CWE-ID and CWE-Type are available categorizations provided by the MITRE Corporation [37]. CWE-Type is a higher-level of vulnerability category, where each CWE-Type may contain multiple similar CWE-IDs. Since CWE-ID and CWE-Type are highly correlated labels, we learn a shared CodeBERT model through multi-objective optimization as described in Section 8.3.1.

To predict the severity of vulnerabilities, we leverage a separate CodeBERT model instead of sharing the same model with the CWE classification task. This

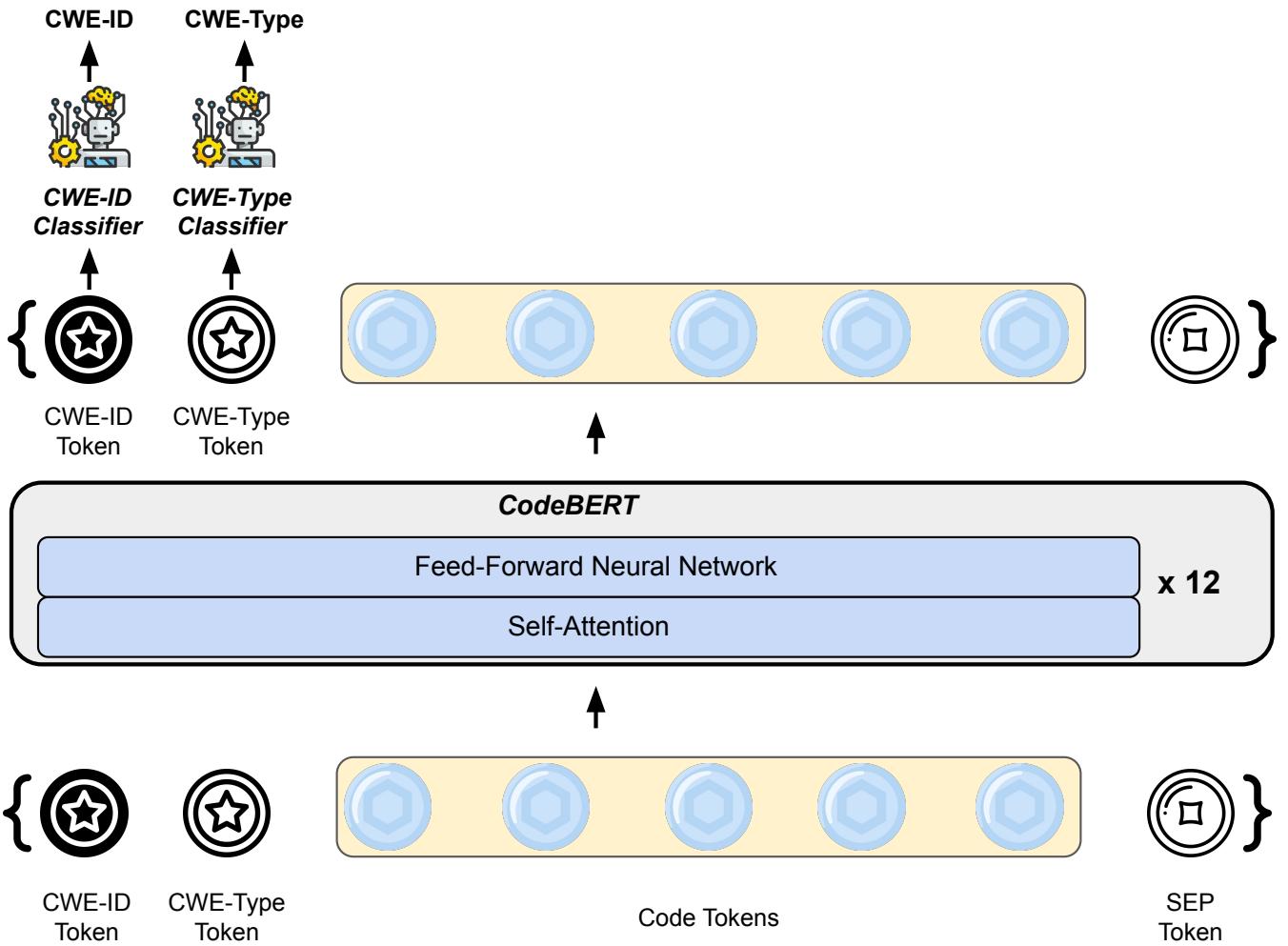


Figure 38: An overview architecture of multi-objective CWE classification.

is due to (1) the CVSS severity score being a regression task that is different from the CWE classification; and (2) the CVSS severity score being determined using metrics provided by Common Vulnerability Scoring System (CVSS) [45] rather than based on vulnerability types. Thus the vulnerability types and severity scores are not necessarily highly correlated. In the following paragraphs, we describe in detail our approach for CWE classification, followed by severity regression.

### 8.3.1 Multi-Objective CWE Classification

In this section, we introduce our novel multi-objective approach that is used to predict the CWE-ID and CWE-Type of a vulnerable function.

---

### 8.3.1.1 Sequence Representation

As shown in Fig 38, instead of using only one “[CLS]” token as a normal BERT model, our approach leverages two special tokens (one “[CLS]” token for CWE-ID classification and the other “[CLS\_TYPE]” token for CWE-Type classification) along with a “[SEP]” token represents the end of a sequence. All of the special tokens are added during the subword tokenization process as described in our previous work [1].

The intuition behind using two special tokens for different tasks is the success of DeiT. Touvron et al. [154] leveraged two special tokens to distill knowledge from a Transformer-based model for image classification tasks. In DeiT, one special token learns from the ground-truth labels while the other learns from the prediction generated by the teacher model to distill knowledge from it. Similarly, our CWE-ID class token is responsible for the CWE-ID prediction and learns from ground-truth labels of CWE-ID while our CWE-Type class token focuses on CWE-Type prediction and learns from ground-truth labels of CWE-Type.

### 8.3.1.2 Two Non-Shared Classification Heads

Similar to DeiT [154], our approach uses two non-shared classification heads to generate predictions for two different tasks. Each classification head consists of two linear layers with dropout layers in between. Both heads rely on a softmax layer to activate the probabilities of each label which is the final prediction by our approach. The parameters of the two heads are non-shared, so they can map the representation of their own special token (i.e., the class token of CWE-ID and CWE-Type) to the prediction without conflicting with each other.

The reasons for having two non-shared classification heads are (i) the number of classes for CWE-IDs is different from the number of classes for CWE-Type and (ii) we aim that each classification head can focus and specialize for each task (CWE-IDs or CWE-Type) to obtain better performances. Thus, we use separate non-shared heads to classify CWE-IDs and CWE-Type respectively. In concurring with our design, the experiment results in Fig 40 show that our multi-objective method with a shared transformer architecture achieves the best performance among other baseline methods.

### 8.3.1.3 Multi-Objective Optimization

The problem solved by our approach can be considered as a multi-task learning (MTL) problem with an input space of  $X$  and a collection of task spaces  $\{y^T\}$  where  $T$  is the number of tasks. Specifically, we have a large vulnerability dataset with data points  $\{x_i, y_i^1, y_i^2\}_i \in [N]$  where  $x_i$  is a vulnerable function,  $y^1$  is a CWE-ID label,  $y^2$  is a CWE-Type label, and  $N$  is the number of data points.

To optimize the parameters of a multi-task model, we need to minimize both loss functions yielded by CWE-ID and CWE-Type labels so the model can infer both labels given the same input. Although the weighted summation is intuitively appealing as shown in Equation 13, obtaining such weighted summation of loss functions for multi-task learning requires an expensive grid search over various scalings or the use of a heuristic such as [232, 233] to find out the optimal values of  $W_1$  and  $W_2$ .

$$\mathcal{L}_{Total} = W_1 \mathcal{L}_{ID} + W_2 \mathcal{L}_{Type} \quad (13)$$

Alternatively, our approach relies on the approach proposed by Sener et al. [234] where the MTL problem is formulated as multi-objective optimization (MOO): optimizing a collection of possibly conflicting objectives. The training objective of our approach can be specified using a vector-valued loss  $L$ :

$$\min L(\theta^{sh}, \theta^1, \theta^2) = \min (\hat{\mathcal{L}}^1(\theta^{sh}, \theta^1), \hat{\mathcal{L}}^2(\theta^{sh}, \theta^2)) \quad (14)$$

where  $L$  is the combined cross-entropy (CE) loss (described in Equation 1) from both tasks computed by MOO,  $\hat{\mathcal{L}}^1$  is the CE loss of the CWE-ID classification task,  $\hat{\mathcal{L}}^2$  is the CE loss of the CWE-Type classification,  $\theta^{sh}$  is parameters of shared 12-layer CodeBERT,  $\theta^1$  is parameters of the CWE-ID classification head, and  $\theta^2$  is parameters of the CWE-Type classification head as shown in Fig 38. In short, we aim to minimize all of the parameters (i.e.,  $\theta^{sh}, \theta^1, \theta^2$ ) during gradient descent simultaneously.

To fulfill the objective Equation 14 during the training phase of our approach, we leverage the same gradient update process as proposed by Sener et al. [234]. As shown in Algorithm 1, we first update the task-specific parameters (i.e.,  $\theta^1$  and  $\theta^2$ ) through the gradient descent algorithm. We then apply the Frank-Wolfe solver

Example 1 - A vulnerable function with a low CVSS score.		Example 2 - A vulnerable function with a high CVSS score.	
Example Code	<pre>void ahci_uninit(AHCIState *s) {     g_free(s-&gt;dev); }</pre>	Example Code	<pre>const Chapters::Display* Chapters::Atom::GetDisplay(int index) const {     if (index &lt; 0)         return NULL;     if (index &gt;= m_displays_count)         return NULL;     return m_displays + index; }</pre>
CVSS Score	1.9	CVSS Score	10.0
Confidentiality Impact	None (There is no impact to the confidentiality of the system.)	Confidentiality Impact	Complete (There is total information disclosure, resulting in all system files being revealed.)
Integrity Impact	None (There is no impact to the integrity of the system)	Integrity Impact	Complete (There is a total compromise of system integrity. There is a complete loss of system protection, resulting in the entire system being compromised.)
Availability Impact	Partial (There is reduced performance or interruptions in resource availability.)	Availability Impact	Complete (There is a total shutdown of the affected resource. The attacker can render the resource completely unavailable.)
Access Complexity	Medium (The access conditions are somewhat specialized. Some preconditions must be satisfied to exploit)	Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)	Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None	Gained Access	None
Vulnerability Type(s)	Denial Of Service	Vulnerability Type(s)	Denial Of Service Execute Code Overflow Memory corruption

Figure 39: Two concrete examples of high and low CVSS severity scores.

(please refer to the original paper written by Sener et al. [234] for details) to find a common descent direction to satisfy our training objective. We then apply the solution of the Frank-Wolfe solver to update the shared parameters (i.e.,  $\theta^{sh}$ ) through the gradient descent algorithm. With such a gradient update process, all of the parameters (i.e.,  $\theta^{sh}$ ,  $\theta^1$ , and  $\theta^2$ ) can be updated at the same time without conflicting with each other.

### Algorithm 1 Gradient Update Equations for MTL

```

1: for  $t = 1$  to  $T$  do
2:    $\theta^t = \theta^t - \eta \nabla_{\theta^t} \hat{\mathcal{L}}^t(\theta^{sh}, \theta^t)$             $\triangleright$  Gradient descent on task-specific parameters(i.e.,  $\theta^1$ ,  $\theta^2$ )
3: end for
4:  $\alpha^1, \dots, \alpha^T = FRANKWOLFESOLVER(\theta)$             $\triangleright$  Solve to find a common descent direction
5:  $\theta^{sh} = \theta^{sh} - \eta \sum_{t=1}^T \alpha^t \nabla_{\theta^{sh}} \hat{\mathcal{L}}^t(\theta^{sh}, \theta^t)$             $\triangleright$  Gradient descent on shared parameters(i.e.,  $\theta^{sh}$ )

```

### 8.3.2 CVSS Severity Score Estimation

We used Version 3.1 of the CVSS score which has a range of 0-10. Below, we provide two concrete examples and present the difference between high and low severity scores. It can be seen that the CVSS scores were assigned based on different measures such as confidentiality impact, integrity impact, availability impact, access complexity, authentication, gained access, etc. A low CVSS score (see Example 1 in Fig 39) usually has None or Partial impact to the confidentiality, integrity, and availability aspects of the software system. In contrast, a high CVSS score (see Example 2 in Fig 39) usually corresponds to higher impact such as Complete impact where there could be total information disclosure, total compromise of system integrity, and total shutdown of the affected resource.

---

As the pre-trained CodeBERT model has been effectively used for vulnerability-related tasks [1, 108], we rely on CodeBERT to obtain word embeddings for each vulnerable function. We add a linear layer as a regression head on top of CodeBERT, which returns one value for each vulnerable function as a severity score prediction. We minimize the Mean Square Error (MSE) loss as described in Equation 15 to train the severity regression model:

$$\mathcal{L}_{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (15)$$

where  $y_i$  is a ground-truth severity score and  $\hat{y}_i$  is a prediction of the model.

## 8.4 A Quantitative Evaluation of AIBugHunter

In this section, we present a quantitative evaluation of AIBugHunter. We present our three research questions, our studied dataset, our experimental setup, and answers to our first three research questions along with their experimental results.

### 8.4.1 Research Questions

The empirical evaluation of LineVul and VulRepair backend components used in our AIBugHunter have been presented in our previous works. To evaluate our new proposed approach for vulnerability type and severity prediction, we conduct a new set of experiments to compare our proposed method with existing baseline approaches. Through an extensive evaluation of our approach on 8,783 C/C++ vulnerable functions including 88 different types of vulnerabilities, we answer the following three research questions:

**(RQ1) How accurate is our approach for predicting vulnerability IDs (i.e., CWE-IDs)?** We focus on CWE-ID multi-class classification and compare our approach with four baseline models. Our approach achieves a multiclass accuracy of 0.65, which is 10%-141% more accurate than other baseline approaches with a median improvement of 86%.

**(RQ2) How accurate is our approach for predicting vulnerability types (i.e., CWE abstract types)?** We focus on CWE-Type multiclass classification and compare our approach with the same four baseline models described in RQ1. Our approach achieves a multiclass accuracy of 0.74, which is 3%-45% more

---

accurate than other baseline approaches with a median improvement of 23%.

### (RQ3) How accurate is our approach for predicting vulnerability severity?

We focus on the CVSS severity score regression task and compare our approach with 3 baseline approaches. Our approach achieves an MSE of 1.8479 and an MAE of 0.8753, which are better than the baseline approaches.

#### 8.4.2 Studied Dataset

To ensure a fair comparison with the previous work, we use the existing benchmark dataset [70]. We did not further parse data from 2020 to 2022 as previous studies did not publish scripts to collect datasets. When implementing our data collection scripts, the collected data may not be the same as used by previous works, posing potential threats to internal validity. Nevertheless, we encourage future studies to evaluate our approach on more recent datasets once available.

As this work is an extended version of our previous work [1], we use the same experimental dataset (i.e., Big-Vul [70]) to evaluate the performance of our approach on vulnerable functions. The Big-Vul dataset is collected from 348 open-source Github projects, which includes 91 different CWEs from 2002 to 2019, and nearly 11k of C/C++ vulnerable functions. Given a large number of vulnerable functions from diverse projects and timeframes, the Big-Vul dataset is a suitable dataset to evaluate whether our vulnerability classification and CVSS score estimation approaches can generalize well to the diverse samples. Other vulnerability datasets such as the Devign dataset [18] are not selected because the CWE-ID and CVSS score information are not provided.

#### 8.4.3 Experimental Setup

**Data Splitting.** Similar to our previous work [1], we split the dataset into 80% of training data, 10% of validation data, and 10% of testing data. We randomly split the data into three similar distributions so different vulnerability types are equally represented in training, validation, and testing sets. We also ensure that CWE-IDs appearing in the testing set should also appear in the training set.

**Data Preprocessing.** To satisfy the scenario of CWE classification tasks and the severity score regression task, we only keep the vulnerable functions with known CWE-ID, CWE-Type, and CVSS scores. Table 23 presents the descriptive statistics of our studied dataset after removing non-vulnerable functions. After data filtering, we keep 8,783 C/C++ functions with 88 different CWE-IDs, 6 differ-

---

Table 23: Descriptive statistics of our studied datasets that describes the distribution of the severity score, and the distributions of cardinalities of CWE-ID and CWE-Type.

	Mean	Median	Std.	1st Quantile	3rd Quantile	Min	Max
CWE-ID Cardinality	100	9	281	3	49	1	2127
CWE-Type Cardinality	1255	415	1491	138	1827	1	4437
Severity Score	6.18	6.8	1.95	4.6	7.5	1.2	10.0

ent CWE-Types, and CVSS scores (labeled based on CVSS version 3.1 [235]) ranging from 1.2-10.0. Note that CWE-IDs and CWE-Types are many-to-one mappings where each CWE-ID has one CWE-Type but each CWE-Type may correspond to many CWE-IDs.

**Multi-objective Classification Model Implementation.** We leverage the pre-trained CodeBERT model as a backbone encoder to generate the shared representation of CWE-ID and CWE-Type classification tasks using the Transformers library in Python. We then add two classification heads on top of the backbone, one predicting the CWE-ID and the other predicting the CWE-Type. Note that the parameters in the backbone are shared by both tasks, however, the parameters in each classification head are task-specific. We leverage two cross-entropy loss functions (i.e.,  $CE_{ID}$  and  $CE_{Type}$ ) and implement the multi-objective optimization process based on the implementation provided by Sener et al. [234] to fine-tune the CodeBERT model under the multi-task setting of CWE-ID and CWE-Type. The multi-objective loss is implemented as described in Section 8.3.1.3 where each cross-entropy loss is implemented based on Equation 16. We use the PyTorch library to update the model and optimize the loss functions.

$$\mathcal{L}_{CE}(p, q) = - \sum_x p(x) \log_q(x) \quad (16)$$

**Severity Regression Model Implementation.** We leverage a pre-trained CodeBERT model with a regression head for CVSS score regression. The model is implemented with the Transformers library and trained using the PyTorch library. We use the Mean Squared Error (MSE) loss to update the model during training as Equation 15.

**Hyperparameter Settings for Fine-Tuning.** We use the default setting of CodeBERT, i.e., 12 Transformer Encoder blocks, 768 hidden sizes, and 12 attention heads. We follow the same fine-tuning strategy provided by Feng et al. [40].

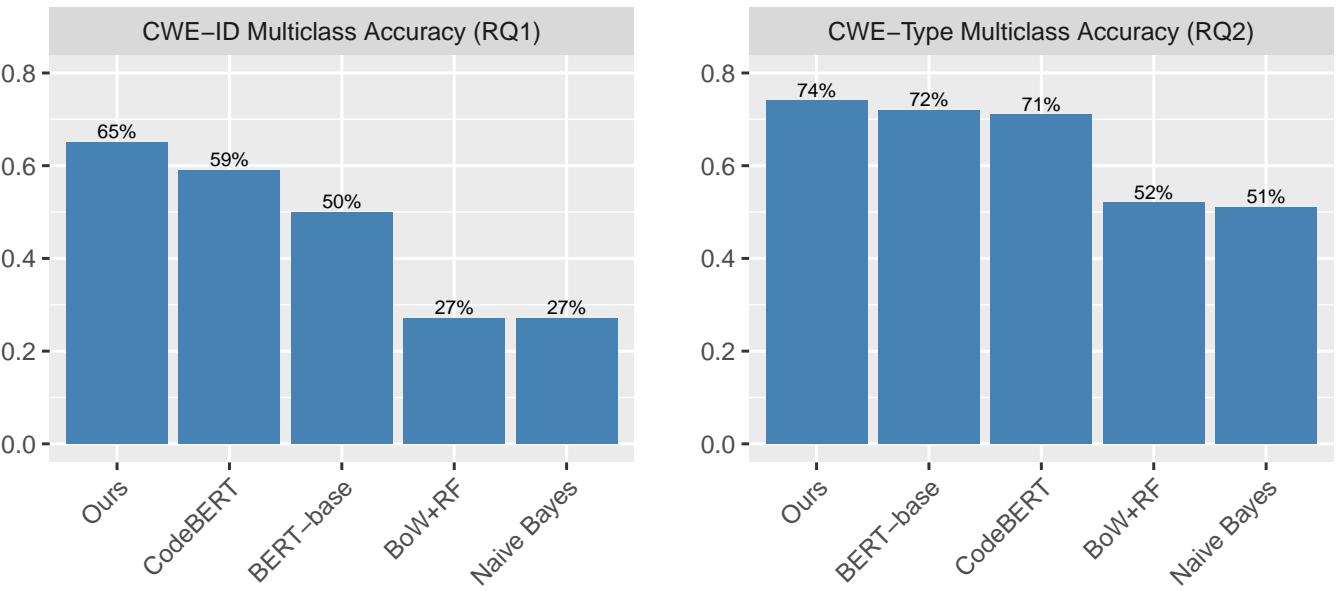


Figure 40: (RQ1 and RQ2) The Multiclass Accuracy of our approach and four other baselines. (↗) Higher Multiclass Accuracy = Better.

During training, the learning rate is set to 2e-5 with a constant schedule. We use backpropagation with AdamW optimizer [73] which is widely adopted to fine-tune Transformer-based models to update the model and minimize the loss function. The best model is selected based on the validation data, which will perform inference on testing data as final evaluation results.

**Execution Environment.** All of the experiments are run under the Ubuntu 20.04 system with an AMD Ryzen 9 5950X CPU with 16C/32T, 64GB RAM, and an NVIDIA GTX 3090 GPU (24GB of memory).

#### 8.4.4 Experimental Results

##### RQ1: How accurate is our approach for predicting vulnerability IDs (i.e., CWE-IDs)?

**Approach.** To answer this RQ, we focus on CWE-ID multi-class classification and compare our approach with four baseline models, described as follows:

1. BERT models pre-trained on natural language (i.e., BERT-base [63]), which have been adopted for CWE classification tasks [20, 144].
2. BERT models pre-trained on programming language (i.e., CodeBERT [40]), which have been applied to software vulnerability prediction [1, 177, 236].
3. BoW+RF uses Bag of Words as features together with a Random Forest model for CWE-ID classification [19, 165].

- 
4. BoW+NB uses Bag of Words as features together with a Naive Bayes model for CWE-ID classification [164].

The pre-trained BERT-based language models are selected because previous studies such as [144] and [237] have used them to achieve promising results on the CWE-ID classification tasks. The Random Forest and Naive Bayes models are selected because they are important machine learning-based methods for CWE-ID classification tasks proposed in previous studies.

We evaluate our approach based on the multiclass accuracy which is computed as  $\frac{\text{Correctly Predicted Testing Data}}{\text{Total Testing Data}}$ .

**Results.** Fig 40 presents the experimental results of our approach and the four baseline approaches according to the multiclass accuracy.

**Our approach achieves an accuracy of 0.65, which is 10%-141% more accurate than other baseline approaches with a median improvement of 86%. These results confirm that our approach is more accurate than other baseline approaches for CWE-ID classification.**

We use CodeBERT as our backbone architecture, however, our approach outperforms the CodeBERT model by 6%. Our approach can learn knowledge from two perspectives based on the class of CWE-ID and the class of CWE-Type where both classes describe the same vulnerable function. The correlated information between the two kinds of labels further benefits our method. On the other hand, the CodeBERT method only learns from CWE-ID labels. In other words, the comparison between our approach and CodeBERT highlights the advancement of using labels from both tasks (i.e., CWE-ID and CWE-Type) with multi-objective optimization. In short, our results demonstrate that **the multi-task learning with multi-objective optimization using both CWE-ID and CWE-Type labels outperforms other baselines that are only trained using CWE-ID labels.**

We analyze the performance of our approach on 879 testing samples. First, 567 of 879 (65%) are correctly predicted. On the other hand, 312 samples are misclassified. Among the 312 misclassified samples, we find that 89 of 312 (35%) were predicted as close to the ground truth (i.e., incorrectly predict the CWE-ID, but correctly predict the CWE-Type). This means our approach can at least correctly predict the vulnerability type for 75%  $\frac{(567+89)}{879}$  of testing samples (outper-

Table 24: (RQ1 Discussion) The Accuracy of our approach for the Top-25 Most Dangerous CWEs ([https://cwe.mitre.org/top25/archive/2021/2021\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html)).

Rank	CWE-ID	Name	Accuracy	Proportion
1	CWE-787	Out-of-bounds Write	43%	9/21
2	CWE-79	Cross-site Scripting	29%	2/7
3	CWE-125	Out-of-bounds Read	67%	44/66
4	CWE-20	Improper Input Validation	66%	71/107
7	CWE-416	Use After Free	52%	15/29
8	CWE-22	Path Traversal	0%	0/4
9	CWE-352	Cross-Site Request Forgery	0%	0/1
12	CWE-190	Integer Overflow	68%	21/31
14	CWE-287	Improper Authentication	0%	0/2
15	CWE-476	NULL Pointer Dereference	41%	7/17
17	CWE-119	Improper Restriction	79%	180/228
18	CWE-862	Missing Authorization	0%	0/1
20	CWE-200	Exposure of Sensitive Info	62%	26/42
22	CWE-732	Incorrect Permission Assignment	86%	6/7
23	CWE-611	Improper Restriction	50%	1/2
25	CWE-77	Improper Neutralization	0%	0/1
			67%	382/566

form all other baselines), highlighting the potential usefulness of our approach in practice.

To investigate whether our approach can classify dangerous real-world vulnerabilities, we evaluate our approach on Top-25 most dangerous CWE-IDs [218] in the testing set to understand the significance of our approach in the practical usage scenario. Table 24 presents the accuracy of our approach on Top-25 most dangerous CWE-IDs. **We find that our approach can correctly predict 67% of the vulnerable functions affected by the Top-25 most dangerous CWE-IDs, which is better than the average performance of our approach(i.e., 65%).**

In addition, Fig 41 presents our method’s accuracy for each CWE-ID in the testing set. It can be seen that the accuracy of our approach is not highly correlated to training or testing data frequencies. Our approach performs well on some of the CWE-IDs with low frequencies such as CWE-754 while having challenges generalizing to other low frequencies CWE-IDs such as CWE-94. However, those CWE-IDs that cannot be identified by our approaches are all CWE-IDs that rarely occur in the dataset. This highlights the challenge of imbalanced data in the CWE-ID classification task where some CWE-IDs are common (e.g., CWE-119)

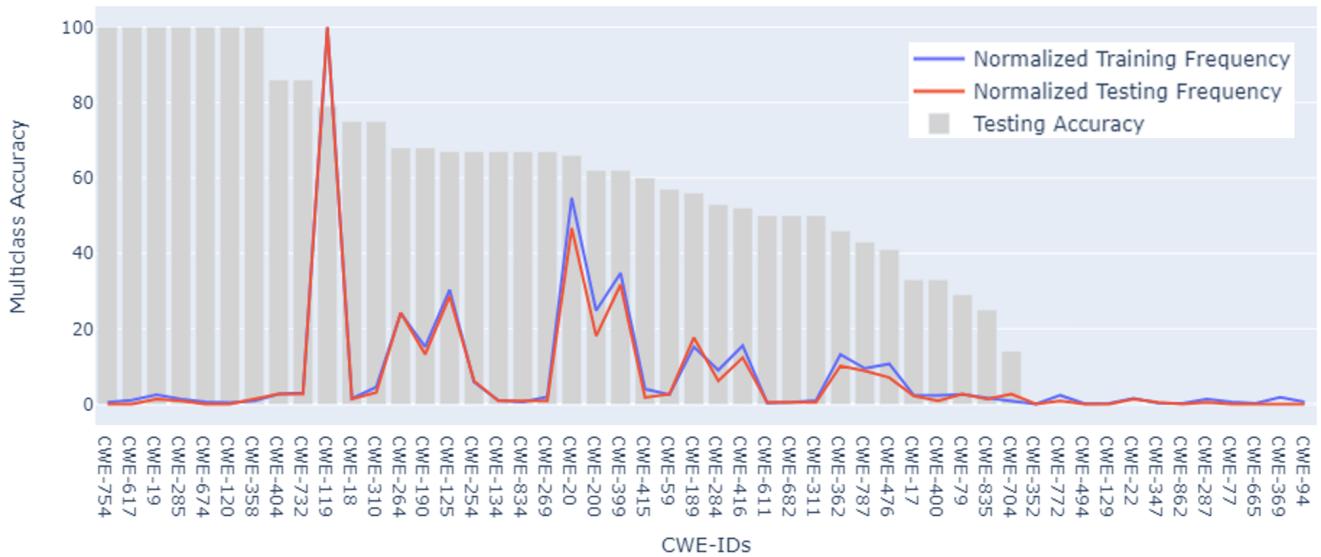


Figure 41: (RQ1 Discussion) Our method’s Multiclass Accuracy of CWE-ID classification for each CWE-ID in the testing set. The accuracy is shown in percentage.

and easy to collect while other CWE-IDs can be rare (e.g., CWE-369) and difficult to collect. Those rare CWE-IDs are more prone than common CWE-IDs to be misclassified by our approach due to not enough training samples. Thus, future researchers may explore new techniques to solve this imbalance problem.

Last but not least, we found that the complexity of vulnerabilities may also affect the performance of our approach. In particular, our approach achieves an accuracy of 86% for the least complex CWE-IDs that are under the CWE-Type of the “class weakness”. Class weaknesses typically describe issues in terms of 1 or 2 of the following dimensions: behavior, property, and resource [238]. For instance, the class weakness of “Uncontrolled Resource Consumption” (CWE-400) describes an issue (Uncontrolled) with a behavior (Consumption) associated with any type of resource. However, our approach only achieves an accuracy of 51% for the most complex CWE-IDs that are under the CWE-Type of the “variant weakness”. Variant weaknesses typically describe issues in terms of 3 to 5 of the following dimensions: behavior, property, technology, language, and resource [239]. For instance, the variant weakness of “Use After Free” (CWE-416) describes an issue (Referencing memory after it has been freed) with a specific resource (Memory) with specific languages (C/C++). These results highlight the challenge of classifying those complex vulnerability types such as the variant weakness.

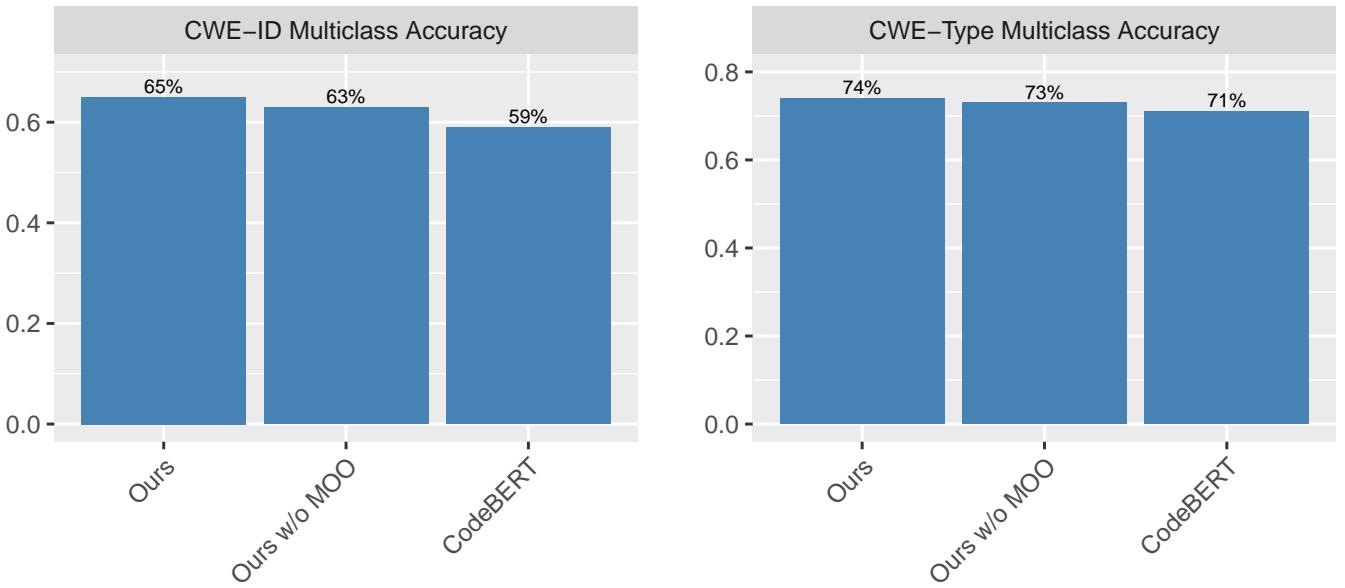


Figure 42: The Multiclass Accuracy of our approach, our approach w/o MOO, and single-task CodeBERT. ( $\nearrow$ ) Higher Multiclass Accuracy = Better.

Table 25: (RQ2) The Multiclass Accuracy of each CWE type for all of the approaches evaluated in RQ2.

Methods	CWE Abstract Types						Overall
	Class	Category	Variant	Base	Pillar	Deprecated	
Ours	85.55%	67.01%	<b>62.86%</b>	60.8%	52.94%	<b>50%</b>	<b>74%</b>
BERT-base	84.42%	<b>68.53%</b>	34.29%	55.11%	<b>64.71%</b>	<b>50%</b>	72%
CodeBERT	78.78%	63.95%	42.86%	<b>71.02%</b>	52.94%	30%	71%
BoW+RF	74.49%	29.44%	14.29%	33.52%	5.88%	10%	52%
Naive Bayes	<b>99.77%</b>	3.55%	-	-	-	-	51%

Table 26: (RQ2 Discussion) The analysis of how different function lengths affect the multi-class accuracy of our approach for CWE-ID and CWE-Type prediction tasks. Note. Function lengths counted by number of tokens in a tokenized function.

Function Length (Tokens)	CWE-ID Accuracy	CWE-Type Accuracy
0-100	84%	85%
101-200	72%	81%
201-300	69%	71%
301-400	60%	69%
401-500	61%	70%
>500	57%	72%

---

**RQ2: How accurate is our approach for predicting vulnerability types (i.e., CWE abstract types)?**

**Approach.** To answer this RQ, we focus on CWE-Type multiclass classification and compare our approach with the same four baseline models described in RQ1. We adopt the same measure as mentioned in RQ1 to evaluate our approach.

**Results.** Fig 40 presents the experimental results of our approach and the four baseline approaches according to the multiclass accuracy.

**Our approach achieves an accuracy of 0.74, which is 3%-45% more accurate than other baseline approaches with a median improvement of 23%.** These results confirm that our approach is more accurate than other baseline approaches for CWE-Type classification.

Our approach performs the best, which is the only model that leverages both CWE-Type and CWE-ID labels during training. The improvement of our approach compared with other baseline approaches is 3%-45% which is not as significant as the improvement demonstrated in RQ1 (10%-141%). The difference in improvements implies that while leveraging both labels can benefit performance for both CWE-ID and CWE-Type classification tasks, our method is more beneficial for the CWE-ID classification.

In addition, Table 25 presents detailed accuracy for each CWE-Type. It can be seen that the performance depends on the number of samples and varies for each type. Nevertheless, our approach has the best overall accuracy and is the only approach that achieves at least 50% of accuracy for each CWE-Type.

In Section 8.3.1.3, we proposed leveraging Multi-Objective Optimization (MOO) instead of taking the weighted summation of loss functions for gradient descent. We now further evaluate whether MOO can help our approach learn better on multi-task learning. Specifically, we compare our approach (using MOO) with a variant method (without using MOO) that leverages a weighted summary of the loss function during gradient descent. The loss function of the weighted summary version of our approach is described as Equation 13. We set  $W_1$  and  $W_2$  to 0.5 so both tasks contribute equally to the total loss. To ensure a fair comparison, we only switch the MOO component of our approach and adopt the same model architecture, hyperparameters, and training strategy for both approaches.

Fig 42 presents the accuracy of our approach, the variant approach, and the

---

single-task CodeBERT. We find that the multi-task learning framework is always better than the CodeBERT which only learns from a single task, and our approach performs the best on both tasks. Our approach can achieve an accuracy of 63%-65% and 73%-74% on CWE-ID and CWE-Type classification respectively while single-task CodeBERT only achieves an accuracy of 59% and 71%. This result confirms that (1) leveraging multi-task learning on two correlated tasks may benefit the model performance on both tasks and (2) the MOO approach used by our approach can learn a model with higher accuracy than the weighted summary approach.

Furthermore, we analyze the impact of function length on our approach for CWE-ID and CWE-Type classification. According to Table 26, when the function length is short, e.g., consisting of 0-100 tokens, our tool can have better 84% and 85% accuracy respectively. However, the performance decreases as functions become longer, for functions consisting of more than 500 tokens, the accuracy becomes 57% and 72% respectively. These results highlight the challenge of tackling long sequences for vulnerability classification tasks. Thus, future researchers should further explore techniques that can classify difficult longer vulnerable functions.

### RQ3: How accurate is our approach for predicting vulnerability severity?

**Approach.** To answer this RQ, we focus on the CVSS severity score regression task and compare our approach with 3 baseline approaches as follows:

1. BERT models pre-trained on natural language (i.e., BERT-base [63]).
2. BoW+RF uses Bag of Words as features together with a Random Forest model for severity score regression [19, 165].
3. BoW+LR uses Bag of Words as features together with a Linear Regression model for severity score regression.

We evaluate our approach based on Mean Squared Error (MSE) 15 and Mean Absolute Error (MAE) where MSE penalizes the predictions that are far from true values through the square of Euclidean distance and MAE measures the exact distance between predicted values and ground-truth values as  $MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$ .

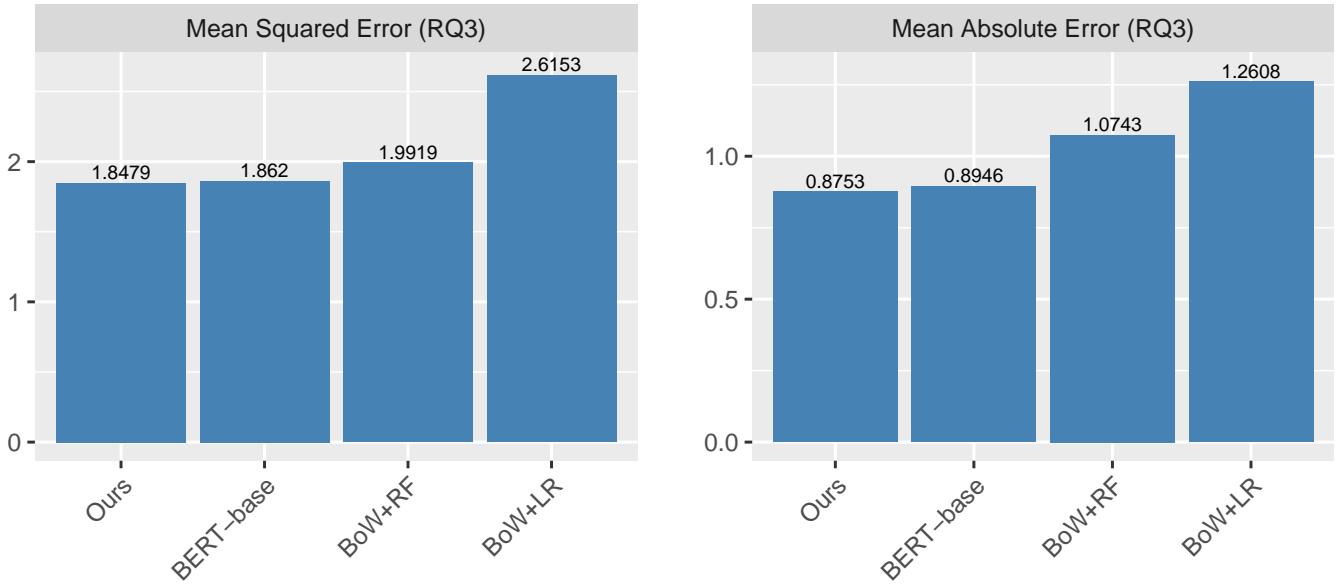


Figure 43: (RQ3) The Mean Squared Error (MSE) and Mean Absolute Error (MAE) of our approach and three other baselines. (↓) Lower MSE, MAE = Better.

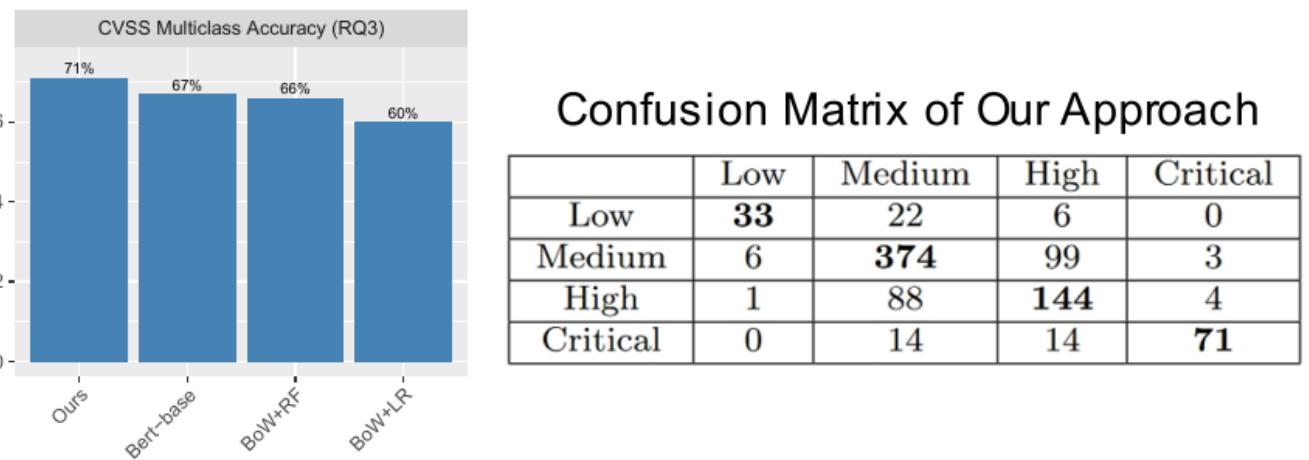


Figure 44: (RQ3 Discussion) The left part is the multi-class accuracy of the CVSS score for each approach evaluated in RQ3. The right part is the confusion matrix of our approach. Note that each class of CVSS is directly mapped from the CVSS score as shown at the bottom of the confusion matrix table.

---

**Results.** Fig 43 presents the experimental results of our approach and the three baseline approaches according to the MSE and MAE loss.

**Our approach achieves an MSE of 1.8479 and an MAE of 0.8753, which are better than all of the baseline approaches.** These results confirm that our approach can predict the most accurate severity scores.

Our approach **achieves 0.1440 and 0.7674 less MSE than the baselines using traditional Bag of Words and Machine Learning algorithms (i.e., BoW+RF, BoW+LR)**. This result highlights the advancement of leveraging a BPE tokenization and pre-trained word embedding with a Transformer-based architecture. The word embeddings with the self-attention mechanism [69] in the Transformer model can learn the semantic features of input source code while the traditional BoW approach only considers the word frequencies when representing source code. Thus, our approach learns a more accurate mapping between a vulnerable function and its corresponding severity score.

Our approach **achieves 0.0141 less MSE and 0.0193 less MAE than the BERT-base (pre-trained on natural language) model**. This result confirms that leveraging a BERT architecture pre-trained using programming languages (our approach) can improve the one pre-trained using natural language.

To investigate whether each approach can accurately predict the severity of vulnerable functions, we map the CVSS score into four classes of severity based on the CVSS protocol, i.e., low, medium, high, and critical as detailed in Fig 44. Our approach achieves an accuracy of 0.71, which is 6%-18% better than other baselines. The result confirms that our approach can correctly predict the severity class for 71% of vulnerable functions in testing data.

To further investigate our approach's performance, we present our approach's confusion matrix in Fig 44. It can be seen that our approach neither estimates low severity as critical (last row, first column) nor estimates critical severity as low (first row, last column). Furthermore, the last column shows that when our approach predicts a critical severity, the accuracy is 91%. Nevertheless, the most common error of our approach is predicting samples to the close class such as estimating a medium severity as high severity, which highlights the challenge of the CVSS severity estimation task.

---

## 8.5 Qualitative Evaluations of AIBugHunter

We conducted qualitative evaluations including (1) a survey study to obtain software practitioners' perceptions of our AIBugHunter tool; and (2) a user study to investigate the impact that our AIBugHunter could have on developers' productivity in security aspects, to answer the following research question:

**(RQ4) How do the software practitioners perceive the usefulness of our AIBugHunter?** According to our survey study, each kind of vulnerability prediction provided by our AIBugHunter is perceived as useful by 47%-86% of participated software practitioners. Furthermore, 90% of participants consider adopting our AIBugHunter if it is freely available in an IDE without conditions. Moreover, our user study shows that our AIBugHunter could save developers' time spent on security analysis that could enhance security productivity during software development.

### 8.5.1 A Qualitative Survey Study

Following Kitchenham et al. [219], we conduct our study according to the following steps: (1) design and develop a survey, (2) recruit and select participants, and (3) verify data and analyze data. We explain the details of each step below.

#### 8.5.1.1 Survey Design

**Step 1: Design and development of the survey:** We designed our survey as a cross-sectional study where participants provided their responses at one fixed point in time. The survey consists of 6 closed-ended questions and 5 open-ended questions. For closed-ended questions, we use multiple-choice questions and a Likert scale from 1 to 5. Our survey consists of two parts: preliminary questions and developers' perceptions of AI-based software vulnerability predictions.

*Part I: Demographics.* The survey starts with a question, ("(D1) What is your role in your software development team?"), to ensure that our survey results are obtained from the right target participants. Then, the survey is followed by a demographics question, ("(D2) What is the level of your professional experience?"), to ensure our survey is distributed across software practitioners with different levels of professional experience.

*Part II: Vulnerability predictions generated by our AIBugHunter.* We then ask about software practitioners' perceptions of AI-based vulnerability predictions.

---

Specifically, we present an example visualization of a prediction generated by AIBugHunter as shown in Figure 35. Then, we ask four questions, i.e., (“(Q1) How do you perceive the usefulness of the recommended location of the vulnerability (i.e., line number)?”), (“(Q2) How do you perceive the usefulness of the vulnerability severity prediction?”), (“(Q3) How do you perceive the usefulness of the vulnerability type prediction (i.e., CWE-ID and CWE-Type)?”), and (“(Q4) How do you perceive the usefulness of the “Quick Fix” button which will replace a vulnerable line with the suggested repair on click?”) Each question is followed by an open question for the rationale.

We use Google Forms to conduct our survey in an online setting. Each participant is provided with an explanatory statement on the landing page that describes the purpose of the study, why the participant was chosen for this study, possible benefits and risks, and confidentiality. The survey takes approximately 10 minutes to complete and is completely anonymous. Our survey has been rigorously reviewed and approved by the Monash University Human Research Ethics Committee (MUHREC ID: 35047).

**Step 2: Recruit and select participants:** We recruit developers who have software development experience through LinkedIn and Facebook platforms. We send a survey invitation to the target groups via direct message. To mitigate potential bias introduced by the participant groups, we selected participants with different software engineering-related professions, different lengths of professional experience, and different organizations. Finally, we obtained a total of 22 responses over a two-week period of recruitment.

**Step 3: Verify data and analyze data:** To verify the completeness of the response in our survey (i.e., whether all questions were appropriately answered), we manually reviewed all of the open-ended questions. Finally, we obtained a set of 21 valid responses. We present the results of closed-ended responses in a Likert scale with stacked bar plots. We manually analyze the responses to the open-ended questions to better understand the in-depth insights.

#### 8.5.1.2 Survey Results

Fig 45 summarizes the survey results, we describe each question in detail in the following.

**Respondent demographics.** Fig 46 presents the overall respondent demo-

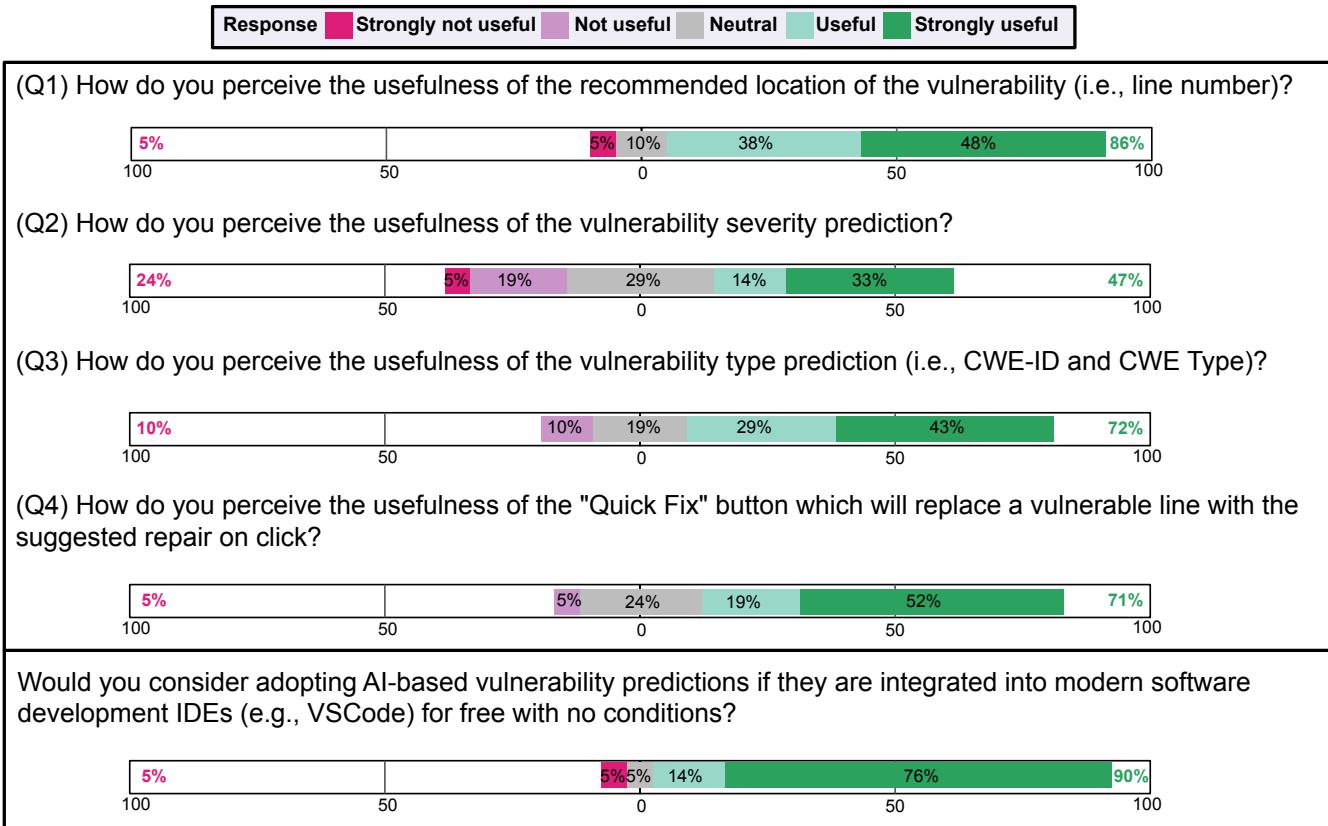


Figure 45: (Q1-Q4) A summary of the survey questions and the results obtained from 21 participants.

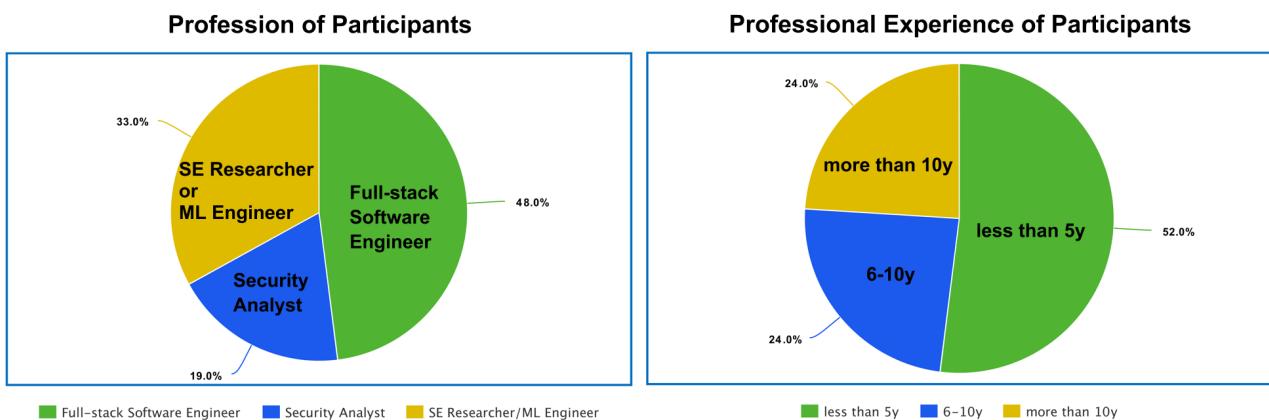


Figure 46: The demographics of our survey participants in terms of their profession and professional experience.

---

graphic. In terms of the profession of the participants, 48% ( $\frac{10}{21}$ ) of them are full-stack software engineers, 19% ( $\frac{4}{21}$ ) of them are security analysts, while the other 33% are software engineering researchers, machine learning engineers, etc. In terms of the level of their professional experience, 52% ( $\frac{11}{21}$ ) of them have less than 5 years of experience, 24% ( $\frac{5}{21}$ ) have 6-10 years of experience, while the other 24% have more than 10 years of experience.

**(Q1) How do you perceive the usefulness of the recommended location of the vulnerability (i.e., line number)?**

**Findings.** 86% of the respondents perceived that the prediction of the vulnerability location is useful due to various reasons:

- Explicitly localize the vulnerability (R1: *I think it is useful to know which line does the vulnerability is located.*, R7: *It can help you quickly identify where the vulnerability is.*, R15: *It is helpful to know the reason for vulnerability and line number.*)
- Reduce time spent on code review (R4: *Speed to fix for developers.*, R11: *This would decrease code review time when I want to check for security breaches.*, R12: *This is something it would take me a lot of time debugging to figure out.*)
- Support debugging process (R6: *Been useful in debugging code*, R9: *Helps with quick resolution of bugs/vulnerabilities.*)

**(Q2) How do you perceive the usefulness of the vulnerability severity prediction?**

**Findings.** 47% of the respondents perceived that the prediction of the vulnerability severity score is useful due to various reasons:

- Prioritization of vulnerability repairs (R3: *Having it will allow me to prioritize fixing high-impact vulnerabilities before looking at things that don't matter as much.*, R8: *Vulnerability score will help me prioritize the fix.*, R11: *This would help me prioritize which part of the code I should fix first.*)
- Risk management (R4: *Modeling business risk is very useful for overall software development planning.*, R9: *Determines the magnitude of the vulnerability against the risks involved.*, R13: *Just like vulnerability scanning tools,*

---

*(it is helpful to know how bad it is and decide the further steps, so yes, it is useful.)*

**(Q3) How do you perceive the usefulness of the vulnerability type prediction (i.e., CWE-ID and CWE-Type)?**

**Findings.** 72% of the respondents perceived that the predictions of CWE-ID and CWE-Type are useful due to various reasons:

- Help understands the vulnerability (R3: *It is important to understand what the vulnerability is before you fix it.*, R9: *Helps to identify common weaknesses and resolve them easily.*, R11: *This would help me understand which kind of security breach I might be facing.*, R15: *It helps understand the problem.*, R19: *Easier to know what the problem is.*)
- Align with security practices (R4: *Aligns well with security practices.*, R13: *A quick classification would help solve the problem more efficiently.*)

**(Q4) How do you perceive the usefulness of the “Quick Fix” button which will replace a vulnerable line with the suggested repair on click?**

**Findings.** 71% of the respondents perceived that the “Quick Fix” button that suggests the vulnerability repair is useful due to various reasons:

- Reduce time spent on vulnerability repairs (R6, R15: *It saves time.*, R9: *Save hours to resolve an issue.*, R11: *This would save a lot of time. I can also modify the suggested codes if I want to.*)
- Help with vulnerability repair implementation (R12: *Having a potential fix helps me think through the fix I would like to implement even if I do not use the suggested fix.*)

**Survey Summary.** Our survey study with 21 software practitioners found that all kinds of vulnerability predictions provided by our AIBugHunter are perceived as useful. Specifically, the vulnerable line prediction reduces the time required to locate vulnerability while severity score prediction helps developers prioritize their workloads. Moreover, the prediction of the vulnerability type helps developers understand the vulnerability and the repair recommendation suggested by

	Demographics			Test	W/O AIBugHunter				With AIBugHunter				Satisfaction
	Years in SE	Years in SS	Degree		CWE	Task 1	Task 2	Task 3	Task 4	Task 1	Task 2	Task 3	Task 4
P1	6	0	Bachelor	Don't know	8	9	10	12	1	2	3	3	5
P2	10	1	Master	Knows	5	7	9	10	1	1	2	2	5
P3	10	3	Bachelor	Knows	5	5	8	11	1	1	2	2	5
P4	4	0.5	Bachelor	Knows	N/A	14	14	N/A	1	1	1	3	4
P5	10	3	PhD	Knows	5	5	7	10	3	1	3	4	5
P6	5	2	Bachelor	Knows	12	12	13	15	1	1	1	4	4
SE: Software Engineering			Correct Answer				Incorrect Answer						
SS: Software Security			CWE: Common Weakness Enumeration										

Figure 47: The experimental results of our user study with six participants. Wherein the first task was to locate the vulnerability, the second task was to explain the vulnerability type, the third task was to estimate the vulnerability severity, and the fourth task was to suggest repairs. The time was measured in minutes and the satisfaction ranged from 1 (highly dissatisfied) to 5 (highly satisfied).

the “Quick Fix” button helps developers come up with repair implementation. Finally, we found that **90% of the respondents consider adopting an AI-based vulnerability prediction approach such as AIBugHunter** if it is publicly available for free in a modern IDE (e.g., Visual Studio Code), highlighting the practical need for our AI-based vulnerability prediction approach.

### 8.5.2 A Preliminary User Study

We conducted a user study to assess the impact that AIBugHunter may have on developers’ productivity in security analysis. To do so, we choose *single-subject experimental designs* as our research methodology—a type of research methodology characterized by repeated assessment of a particular phenomenon (often a behavior) over time. The single-subject experiment is useful when researchers are attempting to observe the behavior of an individual or a small group of individuals and wish to document that observation. In particular, we run the user study with two groups, i.e., a control group (i.e., the group of participants that do not have access to AIBugHunter) and a treatment group (i.e., the group of participants that have access to AIBugHunter). First, we assign a vulnerable C/C++ function to a participant to perform given tasks. The tasks are to locate, estimate severity, explain its type, and suggest repairs. In a well-designed experiment, all variables apart from the treatment should be kept constant between the two

---

groups, allowing us to correctly measure the entire effect of the treatment without interference from confounding variables. With this methodology, our results will not be affected by different participants' expertise and task difficulty (which is commonly affected by randomized control trials). In what follows, we illustrate our user study design followed by the results.

#### 8.5.2.1 User Study Design

**(Step 1) Design and develop a user study.** Our user study is face-to-face where each participant participated individually. Our user study consists of three parts, (1) demographic questions, (2) user study, and (3) survey questions to seek feedback after using AIBugHunter.

In the demographic questions, we asked about the participants' education and experience in software engineering and software security to ensure that we approached the right target group of participants.

In the user study, we used a real-world vulnerable C function in our experiment [240]. The `jpeg_size` function from a PDF generation library caused a buffer overread vulnerability (i.e., CWE-125) due to an inappropriate data bounding check. In particular, we designed our main experiment into two parts. The participants were asked to diagnose the vulnerable C function without using our AIBugHunter tool in the first part while they were asked to diagnose the vulnerable function with the help of our AIBugHunter tool in the second part. In each part, the participants were required to complete four tasks within 15 minutes, i.e., (1) locate vulnerability, (2) explain the vulnerability type, (3) estimate vulnerability severity, and (4) suggest repairs.

In the survey questions, we asked about the participants' satisfaction regarding our AIBugHunter tool using a Likert scale ranging from 1 to 5 followed by an open-ended question for justification. Last but not least, our experiment has been rigorously reviewed and approved by the Monash University Human Research Ethics Committee (MUHREC ID: 36037).

**(Step 2) Recruit and select participants.** We recruited software developers and researchers who have software engineering and/or software security expertise. To ensure the diversity of our participants, we select participants from a diverse set of professional experiences and occupations. Finally, we recruited a total of 6 participants to participate in our user study. Each participant will receive a gift

---

card of \$20 as a token of appreciation.

**(Step 3) Conduct the user study.** We conducted the user study as mentioned in Step 1. We also video-recorded during the user study with permission from the participants. Finally, for each participant, we analyzed the time spent on each task between the two groups (i.e., control vs treatment). Then, we manually analyzed the responses to the open-ended questions to better understand the in-depth insights from the participants.

#### 8.5.2.2 User Study Results

**Participant Demographics.** The education level of our participants varies from bachelor's, master's, to Ph.D. degrees, while the professional experience in software engineering and software security varies from a few months to 10 years, ensuring that the results are not bounded to specific groups of participants.

**Main Findings. Our AIBugHunter can reduce the time spent on detecting, locating, estimating, explaining, and repairing vulnerabilities from 10-15 minutes to 3-4 minutes (see Figure 47).** Without using AIBugHunter, the results show that the majority of the participants cannot provide accurate answers to the given tasks, which indicates that the vulnerability analysis task is challenging and time-consuming. With the use of AIBugHunter, the results show that all of the participants were able to provide accurate answers to the given tasks within 4 minutes. This finding implies that AIBugHunter could possibly enhance developers' productivity in combating cybersecurity issues during the software development lifecycle. Last but not least, all of the participants rated our AIBugHunter as satisfied or highly satisfied due to reasons as follows:

- P1: *It is seamlessly integrated into my development environment.*
- P3: *It exceeds my expectations for automated tools.*
- P4: *Detect the vulnerability down to line-level and provide CWE information.*
- P5: *Identify the vulnerability fast.*

#### 8.5.3 The implications of AIBugHunter to researchers and practitioners

In this section, we discuss the broader implications of our AIBugHunter tool for researchers and practitioners. For practitioners, our AIBugHunter tool can help security practitioners locate vulnerabilities, identify vulnerability types, estimate

---

vulnerability severity, and suggest vulnerability repairs. These AI-powered security intelligence features can produce significant benefits to practitioners. This includes potentially increasing developers' productivity, increasing the security of their software systems, and reducing overall software development costs. For researchers, our AIBugHunter tool is among the first proof-of-concept AI-powered security intelligence tools with numerous features combined into one tool. Many static analysis tools can only perform vulnerability detection, not repairs. Instead, we present how such important features could be integrated into a VS Code Extension. The results of our user study also highlight the usability of our tool and its substantial potential benefits for the software engineering community.

## 8.6 Threats to Validity

### 8.6.1 Construct Validity

**Threats to the construct validity** relate to the potential bias of our survey study and user study. In our survey study and user study, we recruited 21 and 6 participants respectively from different professions such as software engineers and security analysts. However, the results of our two studies could still be biased towards our participants and the results do not necessarily generalize to other audiences. To mitigate this threat for our survey study, we spread our survey on social platforms such as Facebook and LinkedIn to ensure diverse participant demographics. To mitigate this threat for our user study, we recruited software practitioners with different backgrounds and professional experiences for our user study.

The goal of our survey study and user study is to investigate the usefulness of the tool. Thus, we only focus on correct predictions when designing our survey study and user study. However, our AIBugHunter could also return incorrect predictions. Thus, an extended user study is also required to fully evaluate the impact of our AIBugHunter by including both correct and incorrect predictions. Since this research question requires a rigorous user study and a different methodology than we use in this article, we plan to investigate this in future work.

Furthermore, the maturity of AIBugHunter is still at the early stage of development and is not yet ready for commercialization. Our user study experiment was conducted as a preliminary analysis. Thus, the findings are only limited to our studied group, and may not be generalized to other participants, users, software systems, and organizations. Therefore, an extensive evaluation of AIBugHunter is

---

still needed.

### 8.6.2 Internal Validity

**Threats to the internal validity** relate to our choice of hyperparameter settings (i.e., optimizer, scheduler, learning rate, etc.) of our models to classify vulnerability types and estimate vulnerability severity. Finding a set of optimal hyperparameter settings of the CodeBERT model is extremely expensive due to a large number of trainable parameters in CodeBERT and the large search space of the Transformer architecture. Thus, we leverage the default setting of CodeBERT as reported by Feng et al. [40]. Hence, our results serve as a lower bound for our approach, which can be further improved through hyperparameter optimization [89, 91]. To mitigate this threat, we report the hyperparameter settings in the replication package to support future replication studies.

### 8.6.3 External Validity

**Threats to the external validity** relate to the generalizability and applicability of our AIBugHunter. The models used in AIBugHunter were trained using BigVul [70] and CVEFixes [195] datasets consisting of C/C++ source code. Thus, our models do not necessarily generalize to other data and programming languages. However, the AIBugHunter tool could be used with other programming languages as it is designed to adopt any deep learning models. Nevertheless, future work could explore the effectiveness of the AIBugHunter tool in other programming languages when other models are used.

## 8.7 Related Work

We discuss key previous studies of ML-based vulnerability prediction and multi-task learning for software vulnerability prediction. We compare our approach with previous methods and illustrate the difference.

### 8.7.1 ML-Based Vulnerability Type Classification

Multiple ML-based approaches have been proposed to automate the CWE-ID classification task [164–166]. [166] constructed a Huffman Tree SVM , [164] used a Naive Bayes model, and [165] leveraged a Random Forest model to automate the CWE-ID classification task. All of these approaches rely on the Bag of Words technique, while such a method can embed textual input features into numeric

---

vector space, such embedding based on word counting can not capture enough semantic information of input.

Instead of using CVE entries as input, [19] leveraged ML-based models to classify CWE-IDs for vulnerability security patches based on the features extracted from security patches. However, defining such hand-crafted features is time-consuming and may require much effort.

Recently, researchers have proposed DL-based models that learn the input representation through neural networks to better capture the semantic features of the input. [21] proposed ThreatZoom, a Hierarchical Neural Network that considers the hierarchical nature of CWE-ID. [144] leveraged the BERT architecture to learn textual features through the self-attention mechanism.

Previous studies focus on mapping either CVE entries (i.e., vulnerability description) or security patches into CWE-ID, however, such input features are not available during the software development stage, thus they are not compatible with our AIBugHunter, where it requires an ML model to predict based on the source code written by developers. In contrast, our approach only takes vulnerable source code without any description as input and predicts the corresponding CWE-ID. Therefore, it can support our AIBugHunter to generate vulnerability predictions based on the code written by developers.

#### 8.7.2 Multi-Task Learning for Software Vulnerability Prediction

[241] used three ML ensemble classifiers to predict CVSS characteristics based on vulnerability description. [46] proposed DeepCVA which uses multiple GRUs and a shared embedding layer as a multi-task learning framework for commit-level vulnerability assessment. [48] leveraged a Bi-LSTM as a shared feature extractor with multiple classifiers to predict different Common Vulnerability Scoring System (CVSS) characteristics based on vulnerability description. [47] used a shared BERT architecture with two prediction heads to learn a multi-task model that supports CVSS severity score classification and regression.

Some of these studies leveraged a shared architecture [47, 48] that can learn from labels of different tasks that are correlated, hence may help improve the model performance. Nevertheless, all of these studies relied on the weighted summation of loss functions during gradient descent, i.e., (1) averaging the loss of each task [46], (2) tuning loss weights of each task [47], (3) summarizing loss

---

of each task [48]. Such a weighted summation approach may not find the optimal solution when updating the shared model, for instance, the updated parameters are better for one task but not the other as discussed in Section 8.3.1.3. In contrast, our approach finds an optimal collection of parameters that benefits all tasks simultaneously during gradient descent that can optimize a collection of possibly conflicting objectives. To the best of our knowledge, this paper is among the first to leverage multi-objective optimization to learn a DL model for the software vulnerability classification task.

## 8.8 Summary

In this chapter, we integrate our previous works and further propose AIBugHunter, an ML-based vulnerability prediction tool to (1) localize vulnerabilities, (2) classify vulnerability types, (3) estimate vulnerability severity, and (4) suggest repairs. We present a new approach based on multi-objective optimization to classify vulnerability types and learn a regression CodeBERT model to estimate vulnerability severity. To the best of our knowledge, this work is among the first to deploy an ML-based vulnerability prediction tool to the VS Code IDE. Our AIBugHunter realizes real-time vulnerability prediction during software development, which may help integrate security approaches into the software development life cycle. Our empirical survey study with 21 software practitioners confirms that our AIBugHunter is perceived as useful; and our user study indicates that our AIBugHunter could help reduce developers' time spent on security analysis, which could enhance developers' productivity in combating security issues during software development.

---

## **9 ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?**

© 2023 IEEE. This is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive version was published in 2023 30th Asia-Pacific Software Engineering Conference (APSEC), pp. 632–636, published 02 April 2024, <https://doi.org/10.1109/APSEC60848.2023.00085>.

---

## 9.1 Introduction

Software vulnerabilities are weaknesses or flaws in software code that can be exploited by attackers to compromise the security of a system, gain unauthorized access, or cause unintended behavior. Recently, there have been advancements in employing language models for source code (e.g., CodeBERT, GraphCodeBERT, and CodeT5) to automatically achieve the following tasks: (1) pinpoint vulnerable functions and statements [1] within source code; (2) recognize vulnerability types to explain detected vulnerabilities [5]; (3) estimate the severity of vulnerabilities [5]; and (4) suggest repair patches [3, 23]. In particular, a deep learning-based software security tool named AIBugHunter was proposed in VSCode that achieves promising results for the aforementioned four vulnerability tasks using multiple fine-tuned language models for source code [5].

On the contrary, large language models (LLMs) like ChatGPT have effectively demonstrated their competence in tasks related to code, such as the simulation of system behavior from provided requirements, the formulation of API specifications, and the discernment of implicit assumptions within code [51]. Leveraging ChatGPT’s considerable scale, with 175 billion parameters for gpt-3.5-turbo [52] and 1.7 trillion parameters for gpt-4 [53], offers the potential for its application in vulnerability-related tasks. However, to the best of our knowledge, no comprehensive studies have been conducted to evaluate the entire vulnerability workflow, spanning from detecting vulnerabilities and explaining their types to estimating their severity and repair suggestions.

In this work, we conduct a thorough analysis to assess ChatGPT’s ability for the four vulnerability prediction tasks mentioned above. Noteworthy is the fact that ChatGPT’s 1.7 trillion parameters surpass the count of parameters in source code-oriented pre-trained language models like CodeBERT and GraphCodeBERT by nearly 14,000 times. Therefore, the prevalent approach to utilizing ChatGPT involves furnishing it with appropriate prompts and task examples, rather than engaging in fine-tuning for these specific downstream tasks. It is important to note that the model parameters of ChatGPT remain proprietary by OpenAI, thereby precluding the possibility of fine-tuning its parameters for vulnerability tasks.

Thus, we compare prompting ChatGPT with other fine-tuned language models specifically designed for source code purposes. We conduct experiments to compare two versions of ChatGPT (i.e., gpt-3.5-turbo and gpt-4) with four compet-

---

itive baseline approaches (i.e., AIBugHunter [5], CodeBERT [40], GraphCodeBERT [155], and VulExplainer [2]) designed for software vulnerability on four different vulnerability tasks. Through an extensive evaluation of ChatGPT on two vulnerability datasets (i.e., Big-Vul [70] and CVEFixes [195]) encompassing over 190,000 C/C++ functions, we answer the following four research questions:

**(RQ1)** How accurate is ChatGPT for function and line-level vulnerability predictions?

**Results.** ChatGPT achieves F1-measure of 10% and 29% and top-10 accuracy of 25% and 65%, which are the lowest compared with other baseline methods.

**(RQ2)** How accurate is ChatGPT for vulnerability types classification?

**Results.** ChatGPT achieves the lowest multiclass accuracy of 13% and 20%, which is 45%-52% lower than the best baseline.

**(RQ3)** How accurate is ChatGPT for vulnerability severity estimation?

**Results.** ChatGPT gave the most inaccurate severity estimation with the highest mean squared error (MSE) of 5.4 and 5.85 while other baseline methods achieved MSE of 1.8 to 1.86.

**(RQ4)** How accurate is ChatGPT for automated vulnerability repair?

**Results.** ChatGPT failed to generate any correct repair patches while other baselines correctly repaired 7%-30% of vulnerable functions.

**Novelty & Contributions.** This work represents one of the pioneering pilot studies that comprehensively assess ChatGPT’s (gpt-3.5-turbo and gpt-4) performance in vulnerability prediction, vulnerability type identification, severity estimation, and patch recommendation. In addition, we conduct comparative analyses with other state-of-the-art language models, specifically fine-tuned for software vulnerability-related tasks.

## 9.2 Related Work

Recently, researchers have been investigating the applicability of ChatGPT for software vulnerability tasks. Cheshkov et al. [242] investigated the base ChatGPT (gpt-3.5-turbo) performance for vulnerability prediction and classification using 120 samples across five different CWE-IDs. Zhang et al. [54] designed suitable prompts for ChatGPT to enhance its performance for vulnerability prediction. On the other hand, Napoli et al. [55] investigated ChatGPT’s performance

---

for the smart contracts vulnerability correction task. Some previous studies have evaluated ChatGPT’s performance for the automated program repair (APR) task where the model was asked to fix general bugs [56–58]. In particular, Pearce et al. [56] assessed large language models’ performance for the program repair, however, the most advanced two versions of ChatGPT were not included in their experiments.

### 9.3 Problem Statement & Prompt Design

In this section, we introduce the problem statements of the four vulnerability tasks, i.e., (1) function and line-level software vulnerability prediction (SVP), (2) software vulnerability classification (SVC), (3) severity estimation, and (4) automated vulnerability repair (APR). After each problem statement, we illustrate how we design the prompts for ChatGPT to perform the prediction task.

#### 9.3.1 Prompt ChatGPT for Software Vulnerability Prediction

**Problem.** We formulate vulnerability prediction as a binary classification task where the model predicts whether the input source code function is vulnerable. For vulnerable functions, we formulate the line-level vulnerability localization task as a ranking problem, where the model ranks vulnerable statements on the top to reduce the manual analysis workload for security analysts.

**Prompt.** We present example prompts for function and line-level vulnerability prediction in Fig 48. In the initial prompt, we provide ChatGPT with a task description focusing on function-level predictions, along with a clear instruction for return. In the subsequent prompt, we inform ChatGPT that the given function is vulnerable and request it to rank the top 10 most vulnerable-prone statements from the given function. We provide a return template, anticipating that ChatGPT will generate an output consisting of a line number accompanied by its corresponding code statement as predictions.

#### 9.3.2 Prompt ChatGPT for Software Vulnerability Classification

**Problem.** We formulate vulnerability classification as a multiclass classification task where the model identifies a CWE-ID for an input vulnerable function. Common Weakness Enumeration Identifier (CWE-ID) is a community-developed list of common software weaknesses and vulnerabilities [37], which allows security professionals to categorize and communicate about security issues in a standardized manner.



Predict whether the C/C++ function below is vulnerable. Strictly return 1 for a vulnerable function or 0 for a non-vulnerable function without further explanation.

<input function to be assessed>



Prediction: 1



The following C/C++ function is vulnerable. Predict which 10 lines are the most vulnerable-prone.

Return template:

Line 1: <code statement>  
Line 2: <code statement>  
Line n: <code statement>

Generate the ten most vulnerable-prone code statements (with the line number) from the vulnerable function without any explanation.

<input vulnerable function to be assessed>



Line 3: <code statement>  
Line 2: <code statement>  
Line 10: <code statement>  
Line 15: <code statement>  
Line 7: <code statement>  
Line 8: <code statement>  
Line 5: <code statement>  
Line 12: <code statement>  
Line 13: <code statement>  
Line 9: <code statement>



Figure 48: An example prompt for function and line-level vulnerability prediction.



The following C/C++ function is confirmed vulnerable, identify the CWE-ID of this vulnerability. The potential CWE-IDs are ['CWE-264', 'CWE-119', 'CWE-787', ..., CWE-20]. Strictly return one of the CWE-IDs from the list.

<input vulnerable function to be assessed>



CWE-787



Figure 49: An example prompt for CWE-ID classification.



The following C/C++ function is confirmed vulnerable, estimate the CVSS severity score (Version 3.1) of this vulnerable function. The potential severity score is a float between 0 to 10. Strictly return a severity score estimation without any further explanation.

<input vulnerable function to be assessed>



7.5



Figure 50: An example prompt for severity estimation.

**Prompt.** We present example prompts for CWE-ID classification in Fig 49. In particular, we inform ChatGPT that the input function is vulnerable and request it to identify its corresponding CWE-ID. Additionally, we limit the classification scope by providing a list of potential CWE-IDs to ensure a fair comparison with other fine-tuned models.

### 9.3.3 Prompt ChatGPT for Vulnerability Severity Estimation

**Problem.** We formulate vulnerability severity estimation as a regression task where the model predicts a continuous value based on input vulnerable functions to estimate their severity. CVSS (Common Vulnerability Scoring System) severity score is a standardized numerical system used to assess the seriousness of security vulnerabilities in software and systems. We use CVSS version 3.1 ranging from 0 to 10.

**Prompt.** We present example prompts for severity estimation in Fig 50. We inform ChatGPT that the input function is vulnerable and specify the CVSS version and the output range to make it generate a severity estimation for the given function.

---

### 9.3.4 Prompt ChatGPT for Automated Vulnerability Repair

**Problem.** We formulate vulnerability repair as a sequence-to-sequence generation task where the model generates corresponding repair patches for input vulnerable functions.

**Prompt.** We present example prompts for vulnerability repair in Fig 51. Given that model outputs are repair patches designed by Chen et al. [23] instead of the complete repaired program, we provide three repair examples in each prompt to make ChatGPT comprehend our repair task. We then request ChatGPT to create repair patches for vulnerable functions using the templates provided in those examples.

## 9.4 Experimental Design and Results

In this section, we introduce our experimental datasets selected for each vulnerability task followed by the parameter settings and hardware environment used to reproduce the baseline language models fine-tuned for vulnerability tasks. Finally, we present our experimental approach along with the results for each research question.

### 9.4.1 Experimental Datasets

We use the Big-Vul dataset constructed by Fan et al. [70] to evaluate vulnerability prediction (RQ1), classification (RQ2), and severity estimation (RQ3). Big-Vul has been widely adopted for software vulnerability tasks [1, 5], which comprises 188k C/C++ functions gathered from 348 Github projects, encompassing 3,754 code vulnerabilities across 91 types. Each vulnerable function is labeled with a CWE-ID and a CVSS severity score. Its data distribution mirrors real-world conditions, with a vulnerable-to-benign function ratio of 1:20. Similar to previous studies [1, 5], we split the data into 80% for training, 10% for validation, and 10% for testing.

For the automated vulnerability repair (RQ4), we leverage the Big-Vul and CVE-Fixes [195] datasets pre-processed by Chen et al. [23]. The dataset has been adopted to evaluate vulnerability repair approaches [3, 23], which contains 5.5k pairs of vulnerable functions and their repair patches. Similar to previous studies [3, 23], we split the data into 70% for training, 10% for validation, and 20% for testing.



Example Vulnerable Function 1:  
<sample input vulnerable function>

Example Repair Patch 1:  
<sample output repair patches>

Example Vulnerable Function 2:  
<sample input vulnerable function>

Example Repair Patch 2:  
<sample output repair patches>

Example Vulnerable Function 3:  
<sample input vulnerable function>

Example Repair Patch 3:  
<sample output repair patches>

Generate repair patches for the following vulnerable function.  
The return format should strictly follow the Example Repair Patch provided above.

<input vulnerable function to be repaired>

The screenshot shows a code editor interface with a light gray background. In the top left corner, there is a green circular icon with a white swirl pattern. To its right, the text "Repair Patches:" is displayed. In the top right corner, there are three small icons: a copy icon, a thumbs up icon, and a thumbs down icon. Below this header, there is a dark gray rectangular input field containing the letter "c". To the right of this field is a "Copy code" button with a clipboard icon. At the bottom of the input field, there is some partially visible C code: "<S2SV\_ModStart> t , const vpx\_prob <S2SV\_ModEnd> \* context\_tree , <S2SV\_ModS".

Figure 51: An example prompt for automated vulnerability repair.

---

#### 9.4.2 Parameter Settings and Execution Environment

We replicate the baseline methods using the original authors' specified parameter settings, running experiments on a Linux machine equipped with an AMD Ryzen 9 5950X processor, 64 GB RAM, and an NVIDIA RTX 3090 GPU. The ChatGPT prompting was completed via paid API access provided by OpenAI [125].

#### 9.4.3 Experimental Results

##### (RQ1) How accurate is ChatGPT for function and line-level vulnerability predictions?

**Approach.** To answer this RQ, we focus on the function and line-level vulnerability predictions and compare ChatGPT (i.e., gpt-3.5-turbo and gpt-4) with three other fine-tuned baseline language models as follows:

1. AIBugHunter: A recently proposed deep learning-based software security tool that utilizes fine-tuned language models [1, 3] to perform vulnerability prediction, classification, severity estimation, and repair [5].
2. CodeBERT: A language model pre-trained for tasks related to source code, CodeBERT underwent pre-training using the CodeSearchNet dataset [68], encompassing various programming languages. CodeBERT has demonstrated its capability to perform tasks associated with source code [40].
3. GraphCodeBERT: A language model that was also pre-trained on the CodeSearchNet dataset to perform source code-related tasks. Notably, when forming the input for the model, GraphCodeBERT considers the data flow graph in addition to source code tokens [155].

We selected CodeBERT and GraphCodeBERT because they are widely recognized language models pre-trained on the CodeSearchNet [68] dataset, which encompasses diverse programming languages, making them well-suited for source code-related tasks. Similar to the previous study [1], we report F1-measure, precision, and recall to evaluate function-level performance. For line-level performance, we report top-10 accuracy that measures the percentage of vulnerable functions where at least one actual vulnerable line appears in the model's top-10 ranking. This metric has been previously used to evaluate the prediction of line-level vulnerability prediction [1].

**Result.** Fig 52 presents the experimental results of function and line-level vulnerability prediction. **ChatGPT failed to accurately predict at the function**

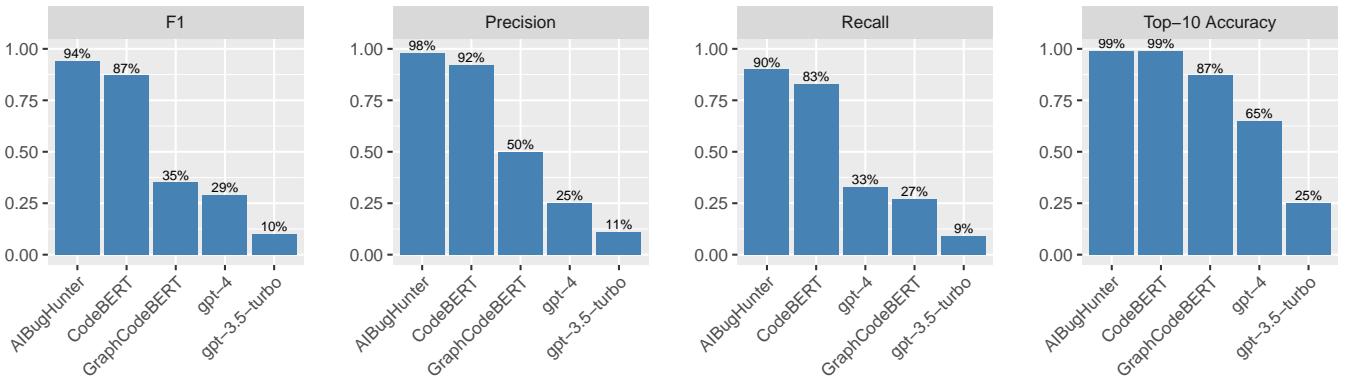


Figure 52: (RQ1) The experimental results of function-level and line-level vulnerability prediction. (↗) For all metrics, higher = better.

**level with an F1-measure of 10% and top-10 accuracy of 25% at the statement level.** The leading-edge gpt-4 with 1.7 trillion parameters achieves an F1-measure of 29% along with a top-10 accuracy of 65%. In contrast, the fine-tuned AIBugHunter achieves an F1-measure of 94% along with a top-10 accuracy of 99% with only 120 million parameters. Despite gpt-4’s extensive model size and pre-training data, it faced challenges in generalizing the vulnerability prediction task without undergoing fine-tuning. These results highlight that the vulnerability prediction task requires models to learn domain-specific knowledge (e.g., vulnerability patterns) and fine-tuning is still required for large language models despite their significant model size.

#### (RQ2) How accurate is ChatGPT for vulnerability types classification?

**Approach.** To answer this RQ, we focus on the vulnerability classification task where we aim to identify CWE-IDs for vulnerable functions. We compare ChatGPT (i.e., gpt-3.5-turbo and gpt-4) with three fine-tuned baseline language models introduced in RQ1. Additionally, we include VulExplainer [2] which leverages language models with a distillation framework to mitigate the data imbalances in the CWE-ID classification task. Similar to previous studies [2, 5], we use the multiclass accuracy measure to evaluate the performance of each method.

**Result.** Fig 53 presents the experimental results of CWE-ID classification. **The accuracy of ChatGPT in correctly identifying CWE-IDs for vulnerable functions is limited, standing at a mere 13%.** The gpt-3.5-turbo and gpt-4 achieve 13%-20% accuracy while the fine-tuned language model baselines achieve 62%-65%. These findings suggest that the accurate identification of CWE-ID for a vulnerable function requires the model to learn to map specific patterns (e.g., buffer overflow) in vulnerable functions to a CWE-ID. However, ChatGPT has not ade-

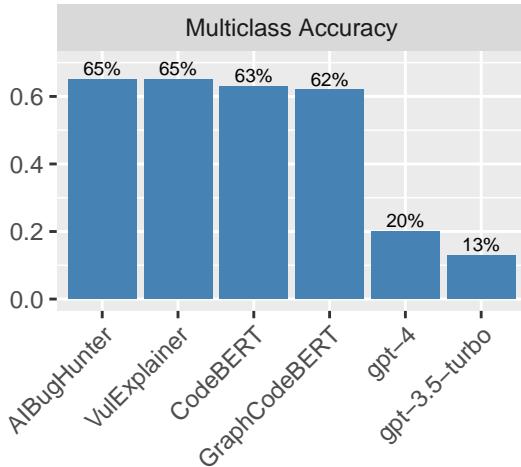


Figure 53: (RQ2) The experimental results of vulnerability type (i.e., CWE-ID) classification. (↗) Higher Multiclass Accuracy = better.

quately acquired such knowledge during the pre-training phase of ChatGPT so a fine-tuning stage is still required to boost its performance.

#### (RQ3) How accurate is ChatGPT for vulnerability severity estimation?

**Approach.** To answer this RQ, we focus on predicting the CVSS score of vulnerable functions. We compare ChatGPT (gpt-3.5-turbo and gpt-4) with the three baselines introduced in RQ1, where CodeBERT has been shown to be effective for severity estimation [5]. Similar to the previous study [5], we use Mean Squared Error (MSE) and Mean Absolute Error (MAE) to assess the performance of each method.

**Result.** Fig 54 presents the experimental results of the severity score estimation. **ChatGPT failed to accurately estimate the CVSS severity score, resulting in an MSE of 5.4 and an MAE of 1.84.** The gpt-3.5-turbo and gpt-4 have MSE of 5.4 and 5.85 while the fine-tuned language model baselines achieve 1.8-1.86. Similar to vulnerability prediction and classification tasks, accurately estimating severity scores also demands software security expertise that ChatGPT has not acquired during its extensive pre-training phase, hindering its ability to provide accurate predictions in this context.

#### (RQ4) How accurate is ChatGPT for automated vulnerability repair?

**Approach.** To answer this RQ, we focus on generating vulnerability repair patches for vulnerable functions. We compare ChatGPT (i.e., gpt-3.5-turbo and gpt-4) with the three baseline methods introduced in RQ1. Notably, AIBugHunter

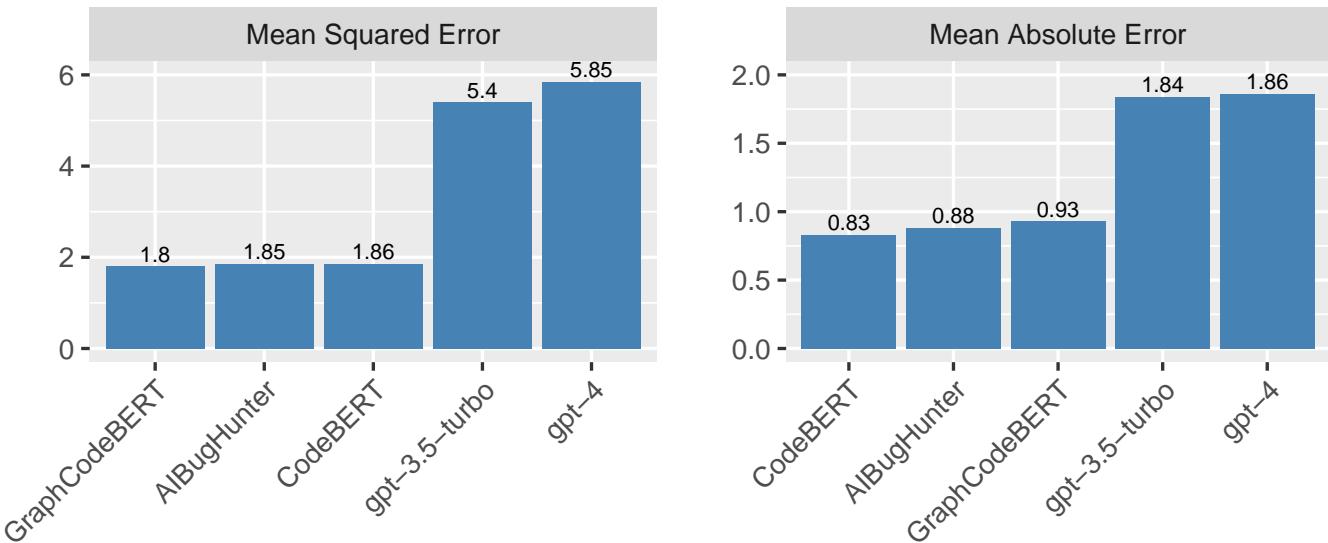


Figure 54: (RQ3) The experimental results of vulnerability severity estimation. (↓) Lower MSE, MAE = better.

leverages the VulRepair [3] model for repair patches generation, which achieves state-of-the-art results in the vulnerability repair problem. Similar to previous studies [3, 23], we use the percentage of perfect prediction (%PP) measure to assess the performance of each method. Only if the generated repair patches are identical to the ground-truth patches, we count it as a correct prediction. The %PP is computed as  $\frac{\text{total correct predictions}}{\text{total testing samples}}$ . We use greedy decoding to return one repair candidate for fine-tuned language models and ensure a fair comparison with ChatGPT. Furthermore, we incorporate BLEU and METEOR scores to evaluate the degree of similarity between the patches produced by the model and the actual patches.

**Result.** Fig 55 presents the experimental results of the vulnerability repair. **ChatGPT failed to generate correct repair patches for all of the vulnerable functions in our testing data.** In contrast, the fine-tuned language model baselines can correctly repair 7%-30% of the testing function. The BLEU and METEOR scores further demonstrate that repair patches generated using baseline methods exhibit greater proximity to the true patches compared to those generated through ChatGPT methods. These results indicate that vulnerability repair is a more challenging task compared with other vulnerability prediction tasks, where ChatGPT struggles to generate correct repairs for vulnerable functions. Thus, a fine-tuning step using domain-specific data is crucial for ChatGPT to generalize its ability for the vulnerability repair task.

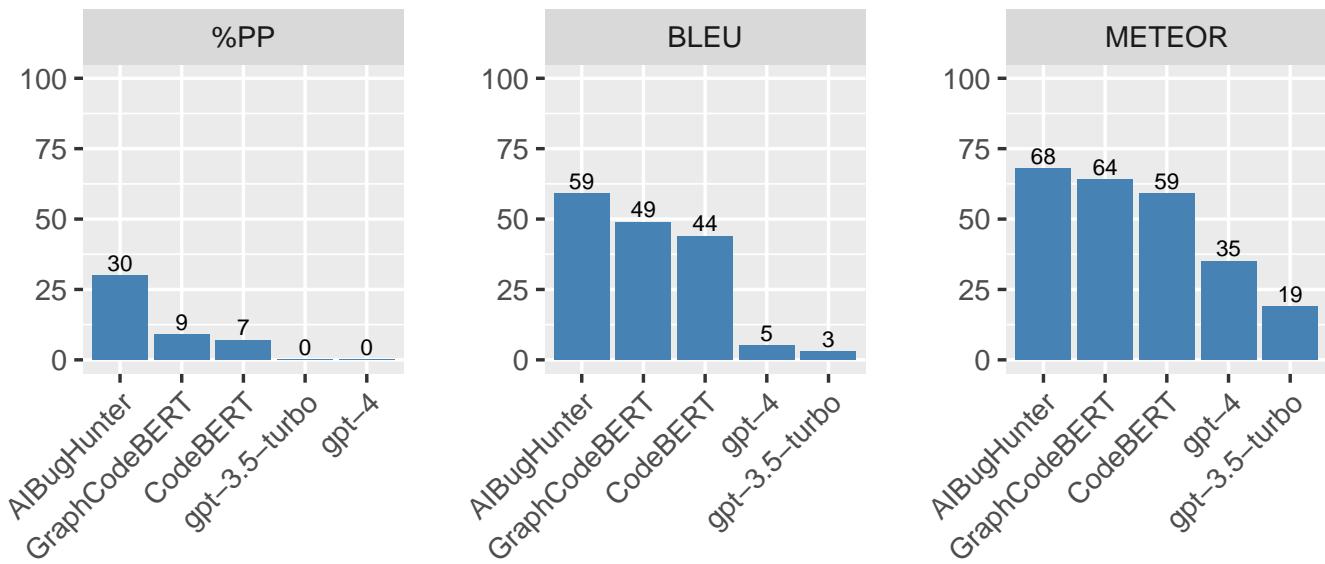


Figure 55: (RQ4) The experimental results of automated vulnerability repair. ( $\nearrow$ ) Higher %PP, BLEU, METEOR = better.

## 9.5 Summary

In this chapter, we empirically evaluate the performance of prompting two versions of ChatGPT (gpt-3.5-turbo and gpt-4) for four common vulnerability tasks: locating vulnerabilities, identifying vulnerability types, estimating severity scores, and suggesting repair patches. We compare the performance of ChatGPT with other pre-trained language models that have significantly smaller model sizes than ChatGPT but have been fine-tuned to perform software vulnerability prediction tasks. Through an assessment encompassing over 190,000 real-world C/C++ functions, ChatGPT yielded the least favorable outcomes for all vulnerability tasks, notably struggling to generate accurate patches for the vulnerability repair task. These findings highlight the imperative of possessing security expertise in addressing software vulnerability prediction tasks, a facet not assimilated by ChatGPT during its extensive pre-training phase. Thus, an additional round of fine-tuning stands as a pivotal requirement for ChatGPT to effectively generalize and undertake software vulnerability tasks.

---

## **10 Paving the Road Ahead: AI-driven Security in DevSecOps – Current Landscape, Challenges, and Future Research Opportunities**

\* **The work in this chapter is accepted** by the journal of ACM Transactions on Software Engineering and Methodology (TOSEM) journal, 2025.

---

## 10.1 Introduction

The traditional software development lifecycle (SDLC) adopts a sequential and siloed approach, with distinct phases executed in a linear fashion, resulting in limited collaboration, slow feedback loops, increased risk of defects, difficulty in managing changes, lack of agility, and increased costs and time-to-market. To address these limitations, DevOps emerged over a decade ago, combining development (Dev) and operations (Ops) to integrate individuals, processes, and technology across application planning, development, delivery, and operations. Moreover, DevOps facilitates coordination and collaboration among previously segregated roles like development, quality engineering, and security [243]. DevOps has been a widely adopted SDLC [244], with organizations generally expressing satisfaction and positivity regarding their transition to DevOps practices [245].

While DevOps improves collaboration, automation, and agility in software development and operations, it often overlooks security considerations until later stages of the development process [246]. This delayed focus on security can lead to vulnerabilities and risks being introduced into the software, potentially exposing organizations to cyber threats and compliance issues [247]. To overcome these security limitations and ensure the secure delivery of software products while preserving DevOps agility, the concept of DevSecOps (Development, Security, and Operations) emerged. DevSecOps involves considering application and infrastructure security from the outset and automating certain security gates to prevent slowdowns in the DevOps workflow [231]. However, incorporating security practices into DevOps to achieve DevSecOps poses several challenges. Common security practices, such as security code review [248], often demand significant manual effort, potentially hindering the agility of DevOps. While automated tools like static application security testing (SAST) can assist in identifying vulnerabilities during code review, they still face limitations in practical effectiveness, especially in real-world scenarios [249]. Additionally, recent research advocates for further automation of traditionally manual security practices to better align with rapid software deployment cycles [250].

In recent years, the rapid advancement of artificial intelligence (AI) technologies has significantly impacted software development and cybersecurity. As organizations increasingly adopt DevSecOps practices [251], integrating security into every phase of the software development lifecycle becomes crucial. AI-driven security approaches are gaining traction in research and are seen as vital tools

---

in this integration. According to Google’s 2023 State of DevOps Report, DevOps teams recognize AI’s potential in enhancing data analysis, automating security tasks, and improving bug identification [252]. This growing emphasis on AI within the DevSecOps framework reflects the industry’s commitment to embedding security more deeply into DevOps processes, ensuring that security measures are proactive and continuous throughout the development lifecycle.

We observe that several systematic literature reviews (SLRs) have explored the DevSecOps domain from diverse perspectives. As presented in Table 27, one SLR focused on AI-driven approaches (i.e., machine learning and deep learning) specifically for the operation and monitoring step in DevSecOps [253], while other studies, not primarily focused on AI, covered all steps of DevSecOps [246, 250, 254–259]. However, none of the previous studies reviewed AI-driven approaches for all steps in DevSecOps comprehensively. Despite the growing intersection of AI and DevSecOps, our analysis reveals a notable absence of an SLR specifically examining literature concerning AI-driven methodologies and tools aimed at automating and enhancing the security aspect of DevOps that helps achieve the DevSecOps paradigm. To bridge this gap, our research aims to contribute to this critical intersection by providing a comprehensive landscape of AI techniques applicable to DevSecOps and identifying future opportunities for enhancing security, trust, and efficiency in software development processes.

In this work, we *define AI approaches as those employing machine learning or deep learning algorithms*. We iteratively defined our search string and searched papers from top-tier software engineering and security conferences and journals, focusing on those published between 2017 and 2023. This timeframe was chosen due to the significant advancements in AI and Large Language Models (LLMs) since the proposal of the transformer architecture in 2017. Our automated literature-searching process yielded a collection of 1,683 papers, from which we manually reviewed and filtered out 1,595 papers based on our selection criteria. Moreover, our snowballing search identified 11 additional papers. We systematically analyzed the collected 99 papers and presented two key aspects: Firstly, we identified existing AI-driven security methods that can be integrated into the DevOps workflow. Secondly, we delved into the challenges and future research opportunities arising from the current landscape of AI-driven security methods. Based on our SLR, this work presents the following contributions:

- **Landscape of AI-Driven Security Approaches for DevSecOps:** We com-

---

prehensively reviewed 99 AI-driven security approaches relevant to DevSecOps, introducing the specific problems they address, detailing the proposed AI methodologies, and presenting the benchmarks used for their evaluation. Notably, we identified and categorized 65 benchmarks, including both synthetic and real-world datasets, which are crucial for assessing the effectiveness of these AI-driven approaches within the DevSecOps paradigm.

- **Identification of Challenges and Future Research Opportunities:** We systematically identified 15 key challenges faced by state-of-the-art AI-driven security approaches, drawing insights from both AI and software engineering perspectives. From an AI standpoint, challenges such as data quality and model explainability are emphasized, while from the software engineering perspective, issues like the complexity of securing Cyber-Physical Systems (CPS), particularly the need to address multiple simultaneous threats across various CPS layers, are highlighted. Correspondingly, we derived 15 future research directions that aim to address these challenges and enhance AI-driven security approaches within the DevSecOps paradigm.

## 10.2 Background and Related Work

In this section, we define the DevOps process of this study and present an overview of each step in DevOps. We then compare our study with existing reviews on DevSecOps.

### 10.2.1 DevOps

To identify the five steps in the DevOps workflow used in this work, the first author initially conducted a review of the relevant literature to explore various definitions of the DevOps process from both industry and academic articles. The first author then discussed the initial findings with the other two authors to incorporate their insights: the second author contributed industry experience, while the third author provided an academic perspective. After synthesizing these inputs, the authors collectively decided to adopt the definition used by Microsoft Microsoft [260], which is widely recognized and implemented in the industry. This choice was made to ensure alignment with widely accepted industry practices and to adopt a well-established framework that effectively captures the essential stages of the DevOps process.

To identify the 12 security tasks within the DevOps workflow, we relied on the collective expertise of the three authors in AI and software security. The first au-



Figure 56: The overview of the DevOps workflow and the identified security tasks relevant under each step in the DevOps process.

---

thor was responsible for providing an initial overview of potential security tasks based on existing knowledge and relevant software engineering and software security literature. This overview was then discussed and refined through iterative brainstorming sessions and online meetings among the three authors. Through these discussions, we ensured that each task was relevant to specific security aspects of individual DevOps steps. This iterative and collaborative approach was essential for mitigating potential biases. The authors reached a consensus on the 12 identified tasks before proceeding with the literature search and further review, ensuring that the identified tasks were well-founded and relevant to our study.

For consistency, we followed the definition provided by Microsoft [260] throughout this work and determined DevOps as a five-step workflow (depicted in Figure 56): (1) Plan, (2) Development, (3) Code Commit, (4) Build, Test and Deployment, and (5) Operation and Monitoring. Below, we introduce each step in detail.

**Plan.** In the “Plan” step of DevOps, teams define project goals, requirements, and timelines. This phase establishes the initial roadmap for the project, involving activities such as gathering user stories, prioritizing features, and assigning tasks. During this phase, the team identifies project objectives and security needs, engaging in threat modeling to grasp security vulnerabilities and plan security measures accordingly [261]. Furthermore, software impact analysis is conducted to identify entities directly or indirectly influenced by a change [262]. This involves the process of assessing and estimating the potential consequences before implementing a modification in the deployed product [263].

**Development.** In the “Development” step of DevOps, software engineers are responsible for implementing the features and functionalities outlined in the requirements and specifications established during the planning phase. In this step, developers operate within an Integrated Development Environment (IDE) where they use static analysis tools (e.g., Checkmarx [29], Flawfinder [26], and Snyk [264]) to scan for potential errors and vulnerabilities before compiling the code.

**Code Commit.** In the “Code Commit” step of DevOps, developers use version control systems like Git to commit their code changes, facilitating collaboration and tracking changes. This step is integral to the Continuous Integration/Continuous Deployment (CI/CD) pipeline, where CI involves the regular integration of developers’ work into the main branch of the version control system

---

[265], while CD automates the deployment of software changes to production without human intervention [266]. Furthermore, dependency management plays a critical role in this step, as modern software often relies heavily on third-party code, such as external libraries, to streamline development processes [267]. However, this practice can introduce dependency vulnerabilities [268], underscoring the importance of effectively managing external libraries and dependencies to ensure the reliability and security of the software product. For instance, Dependabot [269] in GitHub helps users monitor their software dependencies and issues security alerts to users if finding vulnerable dependencies.

**Build, Test, and Deployment.** In the “Build and Test” step of DevOps, software code undergoes compilation and rigorous testing to ensure functionality and reliability. Depending on the organization’s infrastructure and DevOps practices, these processes may occur either on-premises or in the cloud. In software systems, various configurations are used to control features, endpoints (e.g., cache server addresses), security, fault tolerance, tunable behaviors (e.g., timeouts, throttling limits) and so on [270]. Thus, configuration validation tools are used to ensure the proper configuration of cloud environments. Infrastructure as Code (IaC) simplifies infrastructure management by provisioning consistent environments through machine-readable code, eliminating the need for manual provisioning and management of servers and other components during application development and deployment [271]. Various tools and platforms support IaC, including Terraform, Cloudify, Docker Swarm, Kubernetes, Packer, and Chef, Ansible, Puppet for configuration management [272]. Then, the validated software code is deployed to the production environment to make it available for end-users, involving customization, configuration, and installation [273]. This phase involves automating the deployment process to ensure consistency and reliability [274].

**Operation and Monitoring.** In the “Operation and Monitoring” step of DevOps, the focus shifts to maintaining and monitoring the deployed software to ensure optimal performance and security. This phase involves leveraging actionable intelligence and employing data-driven, event-driven processes to promptly identify, evaluate, and respond to potential risks [260]. Log analysis is commonly used to detect and diagnose abnormal behavior, enhancing system reliability using data from application logs and runtime environments [275]. Additionally, anomaly detection involves identifying uncommon and unexpected occurrences over time [276]. Hagemann and Katsarou [277] categorize methods for detect-

Table 27: Comparison with other related reviews focusing on DevSecOps. **#P** - total number of reviewed papers, **SLR** - Is the study a systematic literature review?, **A** - Does the study focus on the machine learning and deep learning approaches for DevSecOps?, **B** - Does the study encompass all steps in DevSecOps?

Reference	Year	Focus	Time	#P	SLR	A	B
Myrbakken and Colomo-Palacios [246]	2017	This study explores the definition, characteristics, benefits, and evolution of DevSecOps while also pointing out the challenges in its adoption.	2014-2017	52			✓
Prates et al. [254]	2019	This study reviews important metrics for monitoring the DevSecOps process.	2013-2019	13			
Mao et al. [255]	2020	This study reports the state-of-the-practice of DevSecOps and calls for academia to pay more attention to DevSecOps.	2013-2019	141			✓
Bahaa et al. [253]	2021	This study reviews machine learning approaches on the detection of IoT attacks.	2016-2020	49	✓	✓	
Leppänen, Honkaranta, and Costin [256]	2022	This study reviews security challenges and practices for DevOps software development.	2015-2019	18	✓		✓
Akbar et al. [257]	2022	This study identifies and prioritizes the challenges associated with implementing the DevSecOps process.	-2022	46			✓
Naidoo and Möller [258]	2022	This study presents a socio-technical framework of DevSecOps based on a systematic literature review.	2016-2020	26	✓		✓
Rajapakse et al. [250]	2022	This study reviews the challenges faced by practitioners in DevSecOps adoption, assesses solutions in the literature, and highlights areas requiring future research.	2011-2020	54	✓		✓
Valdés-Rodríguez et al. [259]	2023	The study reviews methods or models suitable for integrating security into the agile software development life cycle.	2018-2023	39	✓		✓
<b>This study</b>	<b>2024</b>	<b>Our study introduces a landscape of state-of-the-art AI-driven security methodologies and tools tailored for DevSecOps, while also pinpointing the challenges faced by these AI-driven approaches and deriving potential avenues for future research.</b>	<b>2017-2023</b>	<b>99</b>	<b>✓</b>	<b>✓</b>	<b>✓</b>

ing anomalies into three groups: machine learning, deep learning, and statistical approaches. Finally, feedback loops are established to gather insights from the aforementioned monitoring activities, enabling continuous improvement and refinement of the DevOps pipeline and security measures.

### 10.2.2 Other Reviews in DevSecOps

We are aware that there are existing literature reviews of the DevSecOps process, and we have identified 9 related reviews, which are summarized in Table 27. Myrbakken and Colomo-Palacios [246] conducted one of the early literature reviews of DevSecOps, focusing on providing an overview and defining the DevSecOps process, considering DevSecOps as a relatively new concept at the time. Prates et al. [254] reviewed DevSecOps metrics that can be used to monitor the process. For example, defect density and defect burn rate [278] can be employed in the “Development” step to monitor the quality of code being produced and the efficiency of defect resolution over time.

Recent studies have identified several challenges when implementing the DevSecOps paradigm [256, 257]. For instance, Akbar et al. [257] pointed out challenges such as the use of immature automated deployment tools and a lack of software security awareness. Furthermore, Rajapakse et al. [250] organized challenges and solutions in their SLR based on existing literature, presenting future research opportunities in DevSecOps for unresolved problems. Additionally, Valdés-Rodríguez et al. [259] discussed current trends in existing methods for software development involving security. On the other hand, Bahaa et al. [253] concentrated on machine learning/deep learning approaches for Internet

---

of Things (IoT) attacks, while Naidoo and Möller [258] delved into the socio-technical perspective of DevSecOps.

In contrast to previous reviews, our SLR specifically focuses on the emerging trend of AI-driven security approaches within the DevSecOps landscape. While prior works such as Rajapakse et al. [250] have thoroughly categorized challenges related to People, Practices, Tools, and Infrastructure in DevSecOps, our objective is distinct: to categorize AI-driven security approaches, pinpoint the challenges they present, and uncover future research prospects specifically related to AI-driven methods in DevSecOps. To our knowledge, this work represents one of the first SLRs that focuses on AI-driven security approaches across every step of the DevSecOps process.

## 10.3 Approach

This systematic literature review (SLR) follows the principles outlined by Keele et al. [279] and Kitchenham, Madeyski, and Budgen [280], a framework widely adopted in the context of DevSecOps-related SLRs [250, 257, 281, 282]. Our methodology encompasses three stages: (1) formulation of the review plan, (2) execution of the review, and (3) comprehensive examination of the review outcomes. In the following sections, we introduce each of these steps in detail.

### 10.3.1 Research Questions

For a thorough understanding of AI-driven security approaches in DevSecOps, it is crucial to explore the existing AI techniques applicable to each DevSecOps step. Additionally, identifying challenges within these AI-based security techniques is essential for deriving future research directions aimed at further enhancing these techniques. Thus, this systematic literature review aims to address the following research questions:

- **RQ1: What are the existing AI methods and tools employed at each stage of the DevSecOps process, and what specific security challenges do they address?**
- **RQ2: What challenges and future research opportunities exist for AI-driven DevSecOps?**

Table 28: The overview of the selected software engineering and security conferences and journals.

<b>Software Engineering Conference</b>	<b>Acronym</b>	<b>CORE Rank</b>	<b>#P</b>
International Conference on Software Engineering	ICSE	A*	12
Foundations of Software Engineering	FSE	A*	12
Automated Software Engineering Conference	ASE	A*	14
Mining Software Repositories	MSR	A	5
Software Analysis, Evolution and Reengineering	SANER	A	5
International Symposium on Software Testing and Analysis	ISSTA	A	1
International Conference on Software Maintenance and Evolution	ICSME	A	1
Evaluation and Assessment in Software Engineering	EASE	A	0
Empirical Software Engineering and Measurement	ESEM	A	1
<b>Software Engineering Journal</b>	<b>Acronym</b>	<b>Impact Factor</b>	<b>#P</b>
Transactions on Software Engineering	TSE	7.4	11
Transactions on Software Engineering and Methodology	TOSEM	4.4	4
Information and Software Technology	IST	3.9	5
Empirical Software Engineering	EMSE	3.8	5
Journal of Systems and Software	JSS	3.5	5
<b>Security Conference</b>	<b>Acronym</b>	<b>CORE Rank</b>	<b>#P</b>
Network and Distributed System Security Symposium	NDSS	A*	2
Symposium on Security and Privacy	SP	A*	1
Computer and Communications Security	CCS	A*	5
USENIX Security Symposium	USENIX	A*	0
<b>Security Journal</b>	<b>Acronym</b>	<b>Impact Factor</b>	<b>#P</b>
Transactions on Dependable and Secure Computing	TDSC	7.3	8
Transactions on Information Forensics and Security	TIFS	7.2	2

### 10.3.2 Literature Search Strategy

We follow the iterative approach outlined by Kitchenham, Madeyski, and Budgen [280] to develop the search string for this study. For each of the five steps in DevOps, as defined in Section 10.2.1, we first identify the common security processes associated with that step. Next, we combine these security processes with AI-related terms to form the search string for AI-driven security approaches for that step. In particular, the full list of security tasks at each step of DevOps, which this study concentrates on, is presented in Figure 56. For instance, in the planning step, common security processes include threat modeling and software impact analysis. We extract the keywords “threat modeling” and “software impact analysis” with AI-related terms, resulting in a search string such as (*threat modeling OR software impact analysis*) AND (*AI-related terms*). This iterative process is repeated for each step in the DevOps lifecycle. This process will result in a set of DevSecOps activity keywords and a set of AI-related keywords. The complete set of search keywords is as follows:

- *Keywords related to DevSecOps activities: Threat Modeling, Software Impact Analysis, Static Application Security Testing, SAST, Software Vulnerability Detection, Software Vulnerability Prediction, Software Vulnerability Classification, Automated Vulnerability Repair, Automated Program Repair,*

---

*Dependency Management, Dependency Vulnerability, Package Management, CI/CD Secure Pipeline, Software Defect Prediction, Defect Prediction, SDP, Continuous Integration, Continuous Deployment, Configuration Validation, Infrastructure Scanning, Infrastructure as Code, IaC, Log Analysis, Anomaly Detection, Cyber-Physical Systems*

- *Keywords related to AI: Artificial Intelligence, AI, Deep Learning, DL, Machine Learning, ML, LLM, Large Language Model, Language Model, LM, Natural Language Processing, NLP, Transformer, Supervised Learning, Semi-supervised Learning, Unsupervised Learning*

After an iterative refinement process, our final search string is as follows:

```
“([Threat Modeling] OR [Software Impact Analysis] OR [Static Application Security Testing] OR [SAST] OR [Software Vulnerability Detection] OR [Software Vulnerability Prediction] OR [Software Vulnerability Classification] OR [Automated Vulnerability Repair] OR [Automated Program Repair] OR [Dependency Management] OR [Dependency Vulnerability] OR [Package Management] OR [CI/CD Secure Pipeline] OR [Software Defect Prediction] OR [Defect Prediction] OR [SDP] OR [Continuous Integration] OR [Continuous Deployment] OR [Configuration Validation] OR [Infrastructure Scanning] OR [Infrastructure as Code] OR [IaC] OR [Log Analysis] OR [Anomaly Detection] OR [Cyber-Physical Systems]) AND ([Artificial Intelligence] OR [AI] OR [Deep Learning] OR [DL] OR [Machine Learning] OR [ML] OR [LLM] OR [Large Language Model] OR [Language Model] OR [LM] OR [Natural Language Processing] OR [NLP] OR [Transformer] OR [Supervised Learning] OR [Semi-supervised Learning] OR [Unsupervised Learning])”
```

We use Harzing’s Publish or Perish software for our automated search process [283] with the Google Scholar search engine, aiming to gather high-quality and impactful research papers for our review. To achieve this, we target both top-tier software engineering (SE) conferences/journals and reputable software security venues. The targeted venues are summarized in Table 28. Specifically, we focus on 9 SE conferences, all of which are ranked either CORE A\* or CORE A according to International CORE Conference Rankings (ICORE) [284], along with 5 SE journals with impact factors (IFs) ranging from 3.5 to 7.4. Additionally, we include 4 security conferences ranked CORE A\* and 2 security journals with IFs of 7.2 and 7.3. We incorporate SE and security journals with IFs spanning from 3.5 to 7.4 to strike a balance between diversity and reputation. While journals with

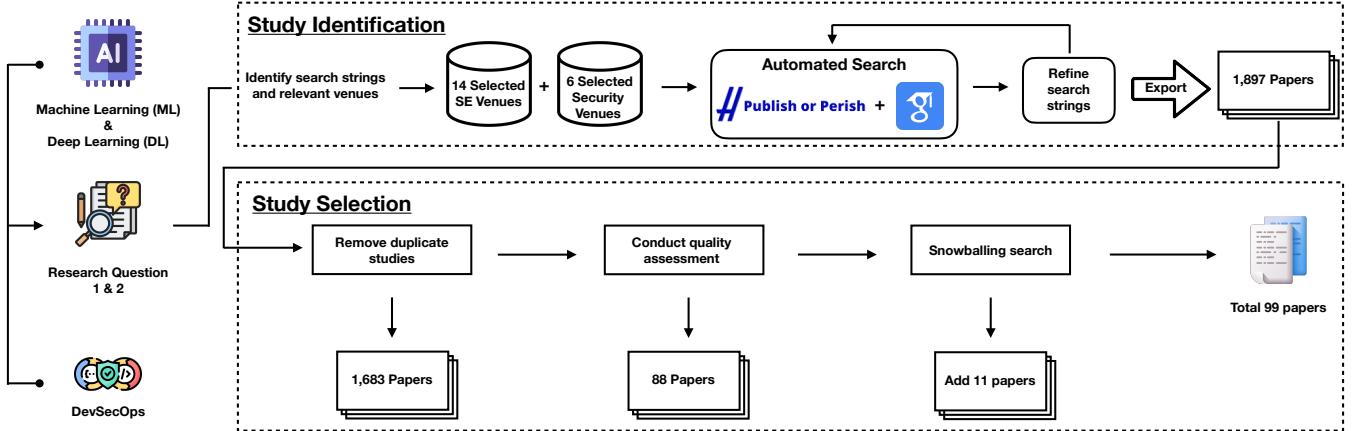


Figure 57: The overview of our study identification and selection process.

a higher IF enhance visibility and credibility, those with a lower IF broaden our literature search.

It is worth noting that we deliberately chose the top-tier SE and security conferences and journals to ensure the quality and impact of the studies included in our review. Our comprehensive search process led to a collection of 1,683 unique papers across 20 distinguished venues, ultimately identifying 88 studies that met our rigorous criteria and were deemed suitable for inclusion. This substantial number of papers demonstrates the thoroughness of our review and compares favorably with similar SLRs outlined in Table 27. While we acknowledge the potential for overlooking relevant studies by excluding lower-ranking venues, our deliberate emphasis on esteemed SE and security venues enables us to capture emerging trends and valuable insights from reputable sources. Notably, this approach aligns with the strategies adopted by other SLRs in the software engineering field such as [285].

This work reviews AI-based security approaches that can be integrated into the DevOps process to achieve the DevSecOps paradigm. It is worth noting that the recent advancements of AI such as language models (LM) and large language models (LLM) stem from the transformer architecture published in 2017 by Vaswani et al. [69]. Thus, we focus our search on papers published from 2017 until the end of 2023 to examine the state-of-the-art AI-based security methods.

#### 10.3.3 Literature Selection: Inclusion-Exclusion Criteria and Quality Assessment Criteria

As presented in Table 29, we have formulated three inclusion criteria (IC) and five exclusion criteria (EC) to ensure that the papers selected are qualified and highly

---

Table 29: The inclusion-exclusion criteria and quality assessment criteria.

<b>Inclusion Criteria</b>	
IC-1	The paper must be peer-reviewed and published at a journal, conference, or workshop
IC-2	The paper focuses on machine learning/deep learning for security purposes for any of the step included in DevOps
IC-3	The paper with accessible full text
<b>Exclusion Criteria</b>	
EC-1	The paper is a duplicate or continuation of another study already included in the review
EC-2	The paper written in languages other than English
EC-3	The paper is a literature review
EC-4	The paper is a replication study
EC-5	Studies from sources such as books, theses, technical reports, monographs, keynotes, panels, or venues that do not undergo a full peer-review process
<b>Quality Assessment Criteria</b>	
QAC-1	The paper presents empirical results or case studies demonstrating the effectiveness of machine learning/deep learning techniques in improving security practices within DevSecOps
QAC-2	The research objective is described
QAC-3	The paper describes techniques or methodologies
QAC-4	The paper describes evaluation or validation methods
QAC-5	The paper presents the study results

relevant to this study. In addition, a well-crafted quality assessment can help prevent biases introduced by low-quality studies [286] and Kitchenham, Madeyski, and Budgen [280] also suggested that such a process should be considered mandatory in any systematic review to avoid research bias. Thus, in Table 29, we further outline five quality assessment criteria (QAC) aimed at assessing the relevance, clarity, validity, and significance of included papers.

Specifically, we employ a binary scale (Yes/No) to evaluate each IC, EC, and QAC for every paper. Papers failing to meet any criteria are excluded from our study. We initially reviewed the title and abstract of each of the 1,683 papers exported from our automated search process after deduplication. However, in some cases, we need to assess the full text to make a decision. Despite the presence of our search string terms in the title, abstract, or keywords of certain papers, it remains unclear how their content relates to the focus of our SLR. For example, while some papers appear to develop an automated security approach for a specific step in DevOps based on their title and abstract, a full-paper inspection reveals that the proposed approach is not relevant to either machine learning (ML) or deep learning (DL). Thus, to ensure that a paper is adequately aligned with the focus of our review (i.e., ML or DL-based security approaches for DevSecOps), we conducted a comprehensive full-text review following the initial assessment of titles and abstracts. By adhering to these steps, we can effectively filter out papers that do not address ML and DL for DevSecOps. Specifically, we excluded 1,584 irrelevant papers based on our IC, EC, and QAC, resulting in a collection of 88 papers selected for this study.

---

#### 10.3.4 Snowballing Search

To expand our search for potentially relevant primary studies, we employed a snowballing approach. This method involves not only examining the reference lists and citations of papers but also systematically tracking where papers are referenced and cited. This dual approach, known as backward and forward snowballing, allows us to thoroughly explore relevant literature beyond the initial set of papers.

Before initiating the snowballing procedure, it is essential to curate a collection of initial papers. In this study, the initial paper collection includes 88 papers after the quality assessment. We performed forward and backward snowballing with deduplication and the full study selection process. Consequently, we obtained an additional 11 papers via our snowballing search.

#### 10.3.5 Data Extraction and Analysis

Mike: below are data analysis We obtained 99 relevant papers after searching, filtering, and snowballing. Figure 58 presents an overview of the distribution of our selected papers. All of the selected papers are peer-reviewed by the venues listed in Table 28. Figure 58 (a) illustrates the distribution of papers across various venues, revealing ASE as the most prevalent venue with a contribution of 14% of the total, followed closely by ICSE and FSE, each accounting for 12%. TDSC follows with 8%, while IST, JSS, EMSE, MSR, SANER, and CCS each contribute 5% respectively. TOSEM contributes 4%, whereas NDSS and SP have the smallest contributions, at 2% and 1% respectively.

In Figure 58 (b), we present the paper trend over the years. Notably, the number of papers annually shows a steady linear increase from 2017 to 2022. However, there is a significant jump from 14 papers in 2022 to 39 papers in 2023. This sudden surge could be attributed to the recent advancements in generative AI and the escalating concerns surrounding software security. Figure 58 (c) presents the paper distributions of SE venues versus security venues where SE venues account for the majority of 82% of papers while security venues account for the remaining 18%.

Figure 58 (d) illustrates the distribution of papers across each step in DevOps. Notably, we found no relevant papers discussing an AI-driven security approach in the planning step of DevOps. This could be attributed to the nature of activities involved in this stage, such as threat modeling and impact analysis, which may

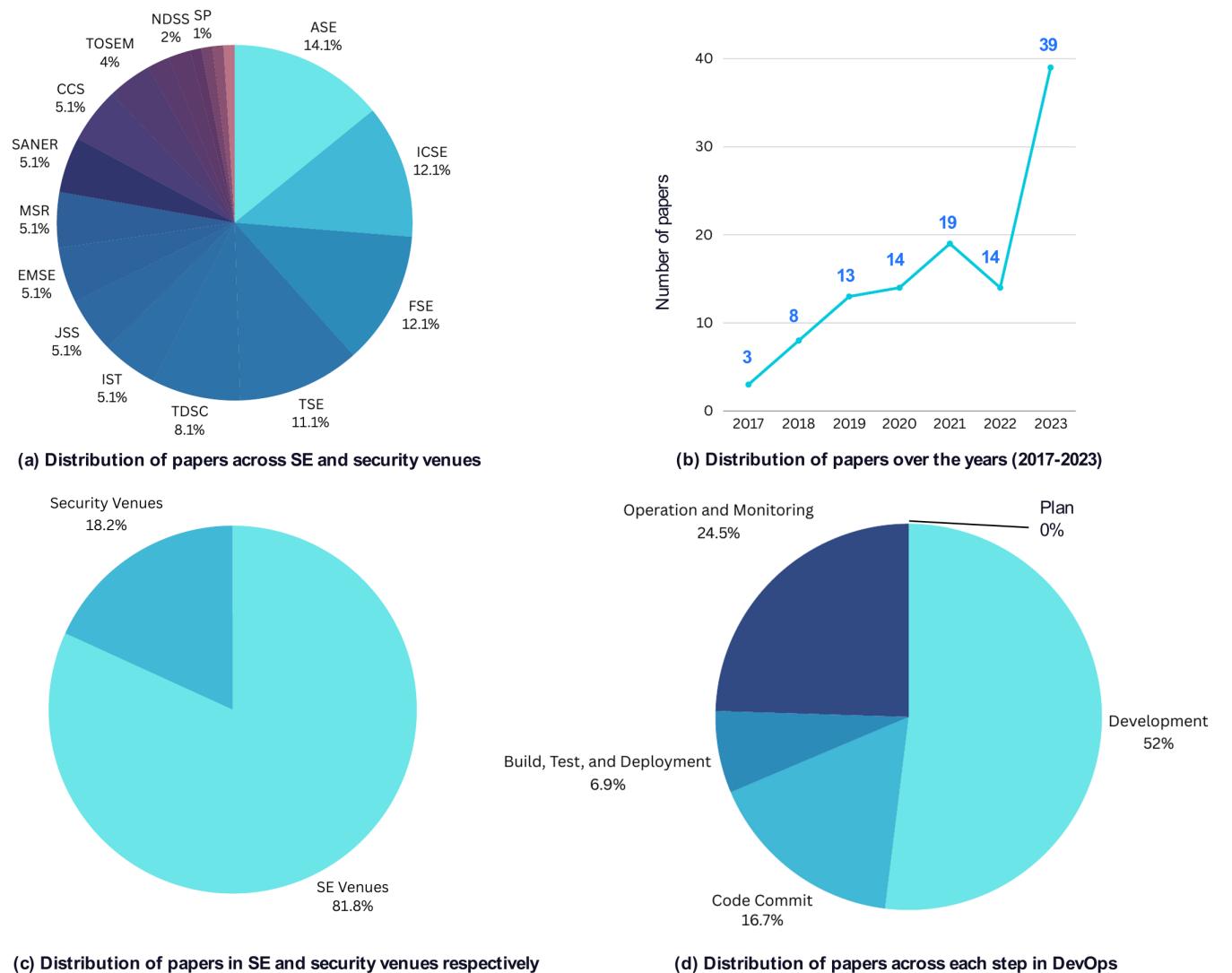


Figure 58: The overview of the distributions of the selected 99 papers.

---

require a higher degree of expertise and human intervention rather than relying on AI algorithms. The majority of studies, accounting for 52%, focus on the Development step in DevOps. In this step, AI-driven security approaches can directly assist software developers by detecting vulnerabilities in their source code, providing explanations, and suggesting repairs. Following this, 24% of the studies concentrate on the Operation and Monitoring step in DevOps. During this phase, AI-driven security approaches can learn from historical data and help monitor and detect anomalies occurring in software systems. Additionally, 17% of studies focus on developing AI-driven approaches for securing the Code Commit step in DevOps, while the remaining 7% concentrate on the Build, Test, and Deployment step in DevOps.

During the full-text review, we conducted data extraction to gather all relevant information necessary for a comprehensive and insightful response to our two research questions outlined in Section 10.3.1. This extraction phase involved collecting data on various AI-driven approaches proposed for security tasks associated with each step in DevOps, as outlined in Figure 56. With this compiled data, we systematically analyzed the relevant aspects of AI-driven approaches aimed at enhancing the security aspect of DevOps.

#### 10.3.6 Data Synthesis and Mapping

To explore the landscape of AI-driven security approaches in DevSecOps, we conducted thematic analysis, following the process outlined by Braun and Clarke [287] to synthesize the extracted data. Our analysis encompassed 99 papers relevant to our research questions (RQs), and we systematically categorized key aspects of AI-driven security within the DevOps process presented in Figure 56. This iterative process spanned approximately three months, during which the first author conducted the primary open coding and mapping for both RQs. Throughout this period, the first author engaged in close discussions with the second author via an internal Slack channel, while the second and third authors provided feedback during weekly meetings. This feedback was instrumental in refining the codes and ensuring that they accurately represented the data. Through this collaborative process, the three authors gradually reached a consensus on the content and themes involved in both RQs, with any discrepancies resolved through discussion and iterative refinement.

**Data Synthesis For RQ1.** The data were imported into Google Sheets, where we conducted an open coding process [288]. This involved systematically break-

---

ing down the key information extracted from each paper—such as the security tasks addressed, the problems solved, the neural network architectures employed, and the benchmarks used—into smaller, meaningful components. These components were then labeled with codes, representing specific categories or themes that emerged from the data [289]. The coding process was iterative, with initial codes refined through multiple rounds of analysis to ensure they accurately captured the essence of the data. Finally, we summarized existing approaches and benchmarks into overview tables for each security task in DevOps to provide a comprehensive comparison of the methodologies and evaluation benchmarks used across the studies.

**Data Synthesis and Mapping For RQ2.** We adopted an identical open coding process as in RQ1 to systematically identify and categorize the challenges encountered by AI-driven security approaches. These challenges were derived directly from the coded data of the reviewed papers, ensuring that our findings were firmly grounded in the existing literature rather than subjective interpretations. By grouping these challenges into distinct thematic categories, we identified overarching themes that reflect the current state of the field. We also thematically mapped the challenges to the proposed potential avenues for future research. In doing the mapping, we first considered how the proposed research directions addressed specific challenges, as evidenced by the existing literature. Next, we evaluated the feasibility and relevance of these proposed directions by examining how they aligned with the identified challenges and the current gaps in the literature. This involved a careful review of existing studies to ensure that each proposed direction was well-grounded in the challenges it aimed to address. Finally, we summarized the challenge themes along with their proposed directions into an overview table, providing a clear and concise reference that highlights the relationship between current challenges and future research opportunities.

Through this multi-layered coding structure in Google Sheets in RQ1 and RQ2, we were able to systematically synthesize the data, leading to meaningful insights into both the present landscape and future opportunities in AI for DevSecOps.

## 10.4 RQ1: AI Methods Overview For DevSecOps

In this section, we introduce AI-driven security methodologies and tools for DevSecOps. We derived insights from 99 collected literature sources to address our RQ1. Each subsection focuses on AI-driven approaches for specific steps in DevOps. These steps may encompass multiple security-related tasks. In each

---

subsection, we begin with a summary table that encapsulates the results related to our RQ1 for that particular step. Furthermore, given the importance of benchmarks in evaluating the effectiveness of AI-driven approaches, we present tables summarizing these security benchmarks before delving into the specific security tasks. This provides an overview of the security benchmarks previously used and offers readers detailed information such as which benchmarks are most commonly used and whether each benchmark is synthetic or real-world.

#### 10.4.1 Plan

##### 10.4.1.1 Threat Modeling

Threat modeling is an engineering technique employed to identify threats, attacks, vulnerabilities, and countermeasures that may impact an application. It aids in shaping the application's design, meeting the organization's security objectives, and minimizing risk [290]. In the planning step of DevSecOps, threat modeling proactively identifies security concerns and integrates security measures into the development process from the outset. One widely adopted framework for threat modeling is STRIDE proposed by Howard and Lipner [291], which categorizes threats into six main types: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service (DoS), and Elevation of Privilege. On the other hand, the DREAD framework proposed by [292] provides a structured approach to assess the severity of identified threats. DREAD stands for Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability. This framework allows teams to assign scores to each criterion, typically on a scale from 0 to 10, to evaluate the potential impact of a threat. By considering factors such as the potential damage caused, the ease of exploitation, and the number of users affected, teams can prioritize their security efforts and focus on addressing the most critical vulnerabilities first. Both STRIDE and DREAD threat modeling are typically conducted by human security experts, often in collaboration with developers, architects, and other stakeholders involved in the software development process. These collaborative processes ensure that security considerations are integrated into DevOps planning throughout the development lifecycle and that potential threats are identified, assessed, and mitigated at an early stage of development.

##### 10.4.1.2 Software Impact Analysis

Software impact analysis is a crucial process in DevSecOps that analyzes, predicts, and estimates the potential consequences before a change in the deployed

---

product [263]. Impact analysis integrates security considerations into the planning phase of DevOps by analyzing potential unexpected side effects of decisions or changes within a system and identifying potentially affected areas. This process starts by identifying impacted modules and functionality, describing proposed changes, and delineating affected areas. Risk assessment is used to evaluate potential risks associated with each change, such as performance changes, security vulnerabilities, and compatibility issues, often using a qualitative scale or numerical scoring system [293]. In particular, Turver and Munro [294] introduced a technique for the early detection of ripple effects based on a simple graph-theoretic model of documentation and the themes within the documentation. This technique aims to provide a more accessible and effective approach to assessing the impact of changes, particularly during the early stages of a project when source code understanding may be limited. Gethers et al. [295] introduced an adaptive approach for conducting impact analysis from a change request to the source code. The approach begins with a textual change request, such as a bug report. It uses a single snapshot (release) of the source code which is indexed using Latent Semantic Indexing to estimate the impact set. In particular, the analysis encompasses information retrieval, dynamic analysis, and data mining of previous source code commits. They evaluated their approaches using open-source software systems and showed significant improvement in their combined approach over standalone approaches.

Despite the critical role of threat modeling and impact analysis, no relevant literature on AI-driven security approaches was found during our literature search. Thus, we have presented common approaches used in these areas. In RQ2, we will highlight AI-driven approaches found outside of academic literature, such as those developed by Aribot, Validata, and Copilot for Security. Additionally, we will outline future research directions, suggesting how researchers can build upon these industry practices to advance AI-driven methods for threat modeling and impact analysis within DevSecOps.

#### 10.4.2 Development

We provide an overview of current AI-driven security approaches for the development phase of DevSecOps in Table 30. Below, we delve into each associated security task in detail.

Table 30: An overview of AI-driven approaches for security-related tasks in the development step.

DevOps Step	Identified Security Task	AI Method	ML	DL	Reference	
Development	Software Vulnerability Detection	RNN	✓		[16, 17, 93, 296, 297]	
		TextCNN	✓		[298]	
		GNN	✓		[14, 18, 32, 108, 299–305]	
		Node2Vec	✓		[306]	
Testing		LM for code	✓		[1, 109, 307]	
		LM + GNN	✓		[109, 123]	
Software Vulnerability Classification	ML algorithms	✓		[19, 165]		
	RNN	✓		[16, 308]		
	TextRCNN	✓		[309]		
	Transformer	✓		[144]		
Automated Vulnerability Repair	LM	✓		[20, 310]		
	LM for code	✓		[2, 5]		
	LM for code + RNN	✓		[311]		
	ML algorithms	✓		[312]		
Deployment	Security Tools in IDEs	CNN	✓		[49]	
		RNN	✓		[22, 313]	
		Tree-based RNN	✓		[207]	
		GNN	✓		[229]	
		Transformer	✓		[35, 314–316]	
		LM for code	✓		[3, 4, 36, 56, 205, 210, 317–321]	
		LM-based security tool	✓		[5]	
	Containerization	ML algorithms			[296]	
		CNN			[49]	
		RNN			[22, 313]	
		Tree-based RNN			[207]	

Table 31: Benchmarks used in evaluating AI-driven software vulnerability detection.

Benchmark	Year	Granularity	Programming Language	Real-World	Synthesis	Evaluation Studies
Firefox [322]	2013	File	C, C++	✓		[296]
Android [323]	2014	File	Java	✓		[296]
Draper [17]	2018	Function	C, C++	✓	✓	[17, 303]
Vuldeepecker [16]	2018	Code Gadget	C, C++	✓	✓	[16]
Du et al. [324]	2019	Function	C, C++	✓		[324]
Devign [18]	2019	Function	C, C++	✓		[18, 32, 303, 306]
FUNDED [191]	2020	Function	C, Java, Swift, PHP	✓	✓	[301]
Big-Vul [70]	2020	Function/Line	C, C++	✓		[1, 32, 108, 123, 303, 304, 307]
Reveal [14]	2021	Function	C, C++	✓		[14, 32, 306]
Cao et al. [299]	2021	Function	C, C++	✓		[299]
D2A [113]	2021	Function	C, C++	✓		[109, 301, 303]
Deepwukong [325]	2021	Function	C, C++	✓	✓	[298, 325]
Vuldeelocator [93]	2021	Line	C, C++	✓	✓	[93]
VulCNN [326]	2022	Function	C, C++	✓	✓	[298, 326]
VUDENC [297]	2022	Token	Python	✓		[297]
DeepVD [302]	2023	Function	C, C++	✓		[302]
VulChecker [300]	2023	Instruction	C, C++	✓		[300]

---

#### 10.4.2.1 Software Vulnerability Detection

Machine learning and deep learning-based vulnerability detection (VD) have been proposed to predict potential vulnerabilities in developers' source code. These detection approaches achieve improved accuracy from traditional static analysis methods without requiring the compilation of developers' code [30]. Leveraging these VD methods during development, developers can proactively identify vulnerabilities, facilitating a “shift-left” in security testing—from the testing/deployment phase to the development phase. This proactive integration aligns with the principles of DevSecOps, embodying the paradigm during the “Development” step. We summarize the benchmarks used to evaluate the AI-driven VD in Table 31. In what follows, we present current AI-driven vulnerability detection methods, explore the problems they address, and discuss how each approach differs or is similar to the others.

**Existing AI methods.** Various AI-driven VD methods have been proposed to predict vulnerabilities on different granularities (i.e., file, function, and line levels) and programming languages (i.e., C, C++, Java, Python, Swift, Php). Below, we introduce the model architecture employed by existing VD approaches, the granularities and programming languages they target, the problems these approaches aim to solve, and how each approach either differs from or complements the others.

Dam et al. [296] focused on the challenges inherent in manually designing code features, such as complexity metrics or frequencies of code tokens, which represent characteristics of potentially vulnerable code. These manually designed features often struggle to sufficiently capture both the semantic and syntactic representations of source code, a crucial capability for building accurate prediction models. To overcome these limitations, Dam et al. [296] leverages a Long Short Term Memory (LSTM) model to sequentially learn both semantic and syntactic features of the code. This model aims to capture intricate patterns in source code files that indicate vulnerabilities. The LSTM-based approach targets file-level VD and has been evaluated using benchmarks from Android applications [323] and the Firefox application [322], highlighting a focus on Java, C, and C++ programming languages.

Most recent work focuses on function-level granularity in VD, as it can save more manual effort compared to file-level analysis, where each file may still contain large code blocks requiring inspection. This finer granularity enhances efficiency

---

and precision in identifying vulnerabilities. For instance, Du et al. [324] proposed the LEOPARD framework which uses program metrics and avoids the need for prior knowledge of known vulnerabilities. This framework seeks to overcome the limitations of machine learning-based methods, which struggle with imbalanced data, and pattern-based methods, which require prior knowledge. LEOPARD was evaluated on a benchmark comprising 11 real-world open-source projects, including BIND, FFmpeg, Linux, OpenSSL, and Wireshark written in C and C++. LEOPARD’s metric-based grouping and ranking can serve as a pre-filtering step, potentially enhancing the efficiency and accuracy of DL models. Similar to Dam et al. [296], Russell et al. [17] leveraged an RNN-based representation learning approach that treated a code function as a sequence of tokens to detect vulnerabilities on the function level. To enhance existing labeled datasets, the authors compiled a vast dataset of millions of open-source functions in C and C++, labeling them with findings from three different static analyzers that indicate potential exploits. They then evaluated the proposed approach using a benchmark parsed from real-world projects, including the SATE IV Juliet Test Suite, the Debian Linux distribution, and public Git repositories on GitHub.

Another mainstream method involves considering the graph structures of source code, such as data flow graphs (DFG), control flow graphs (CFG), or abstract syntax trees (AST). In this context, a code function is treated as a graph with nodes and edges, and graph neural networks (GNNs) are employed to learn the graph representations for making vulnerability predictions.

In particular, Zhou et al. [18] proposed the Devign method to use DFG, CFG, and AST from the code property graph (CPG) [134]. This method leverages GNNs to analyze rich code semantic representations and includes a novel convolutional (Conv) module to improve graph-level classification. Evaluated on the C/C++ dataset from four large-scale open-source projects (i.e., FFmpeg, Wireshark, QEMU, and Linux Kernel), Devign significantly outperforms existing methods, marking one of the first applications of GNNs for VD and highlighting the benefits of graph-based approaches. Furthermore, Chakraborty et al. [14] investigated the significant drop in performance of state-of-the-art DL techniques for automated VD in real-world scenarios, highlighting issues with training data and model design that often lead to high false positives and negatives. To address these challenges, they proposed the Reveal approach, which integrates code property graphs and GNNs similar to Devign [18]. Additionally, Reveal employs techniques such as SMOTE [327] for class imbalance mitigation and triplet loss

---

[328] to enhance the separation between vulnerable and neutral examples. This approach was evaluated on a C/C++ benchmark including Chromium and Debian source code repository. Similarly, Zhang et al. [301] leveraged code property graphs and GNNs to explore cross-domain VD, mitigating the challenge of tedious and time-consuming annotation of large-scale software code, which requires significant effort from domain experts. The proposed method was designed for C/C++ languages and was evaluated on several benchmarks, including Devign [18], Reveal [14], D2A [113], and FUNDED [191].

Cao et al. [299] found that existing GNNs often struggle with incorporating both incoming and outgoing edges into a node simultaneously. Thus, they proposed a bidirectional GNN-based approach that integrates code property graphs to enhance VD. This approach was evaluated on a benchmark including C/C++ open-source projects such as Linux Kernel, FFmpeg, Wireshark, and Libav. Notably, Mirsky et al. [300] found that Existing VD approaches often fail to classify the type of vulnerability found, which leaves developers without crucial information on what specific vulnerabilities are present. Thus, they proposed VulChecker, a tool that can precisely locate vulnerabilities in source code as well as classify their type (CWE). To accomplish this, we propose a new program representation named enriched program dependency graph (ePDG), program slicing strategy, and applied gated graph recurrent neural networks (GRNN) to utilize all of the code’s semantics and improve the reach between a vulnerability’s root cause and manifestation points. This approach was evaluated on an augmented Juliet C/C++ benchmark. While these graph-based representations effectively capture program dependencies and semantics, Wang et al. [302] found that they often include numerous irrelevant entities and dependencies that do not enhance the model’s ability to identify vulnerabilities. To mitigate this, Wang et al. [302] proposed, DeepVD, a graph-based neural network VD model that aims to leverage features emphasizing class separation between vulnerability and benign categories. This approach was evaluated on their collected C/C++ benchmark including 303 different projects and 13,130 vulnerable functions.

Yuan et al. [306] found that existing DL-based approaches are limited to single functions, failing to leverage inter-function information. To address this, they proposed the Behavior Graph Model, which extracts abstract behaviors and relationships between functions to enhance DL-based VD. This method was evaluated on two C/C++ benchmarks: Devign [18] and Reveal [14]. In addition, Zhang et al. [307] found DL models struggle with long code snippets due to input length lim-

---

itations, which impede their ability to fully represent extensive vulnerable code. To mitigate this, they proposed to decompose long code snippets into multiple execution paths from the Control Flow Graph (CFG). By leveraging a pre-trained code model and convolutional neural network with intra- and inter-path attention, the method focuses on learning representations of these shorter paths. This strategy allows the model to better handle long sequences by processing each path individually, thus improving its ability to detect vulnerabilities in lengthy code. This approach was evaluated using three C/C++ benchmarks: Devign [18], Reveal [14], and Big-Vul [70].

Cai et al. [298] based their method on complex network analysis theory to convert the CPG into an image-like matrix and used the TextCNN model to mitigate the scalability issue. The proposed approach was evaluated on two benchmarks (i.e., VulCNN [326] and Deepwukong [325]) derived from the Software Assurance Reference Dataset (SARD) and CVE vulnerability databases. Moreover, Wu et al. [303] employed vulnerability-specific inter-procedural slicing algorithms to capture the semantics of various types of vulnerabilities and used GNNs to learn and understand these vulnerability semantics. They evaluate their approach on common C/C++ vulnerability benchmarks such as Devign [18], Draper [17], Big-Vul [70], and D2A [113]. Finally, Steenhoek, Gao, and Le [123] combined the recently advanced large language models (LLMs) with data flow graph (DFG) to further improve the prediction accuracy on the Big-Vul [70] benchmark for C/C++ languages.

Nevertheless, these VD methods still operate on the function level, which may still consist of multiple lines of code that need to be manually inspected by developers. To address this issue, prior works have proposed various line-level VD approaches. Notably, the VulDeePecker proposed by Li et al. [16] marks the initial stride towards fine-grained VD. This approach introduces the concept of code gadgets, aiming to encompass a finer-grained code representation beyond program or function levels. VulDeePecker was evaluated on their curated benchmark including 19 popular C/C++ open-source projects. Li, Wang, and Nguyen [32] proposed IVDetect which leverages GNNs to predict on function level and used GNNExplainer [65] to interpret model predictions to locate fine-grained vulnerabilities for C/C++ languages. IVDetect was evaluated on Devign [18], Reveal [14], and Big-Vul [70] benchmarks. Wartschinski et al. [297] proposed VUDENC which uses Word2Vec embedding in conjunction with an RNN-based model to predict vulnerabilities within a few lines of code. VUDENC was evaluated on

---

their Python benchmark collected from 1,009 vulnerability-fixing commits. In addition, Zou et al. [304] proposed a multi-granularity VD that can predict function and slice-level C/C++ vulnerabilities, which was evaluated on the Big-Vul [70] benchmark.

Recent works have proposed AI-driven methods that pinpoint vulnerable lines in source code, identifying not just vulnerable functions but also the specific code lines associated with vulnerabilities, aiming to reduce the effort required for manual security code review. Li et al. [93] proposed VulDeeLocator which relies on intermediate code to accommodate extra semantic information with the BiRNN model to locate vulnerable lines for C/C++ languages. VulDeeLocator was evaluated on their curated benchmark involving real-world projects and synthetic and academic programs parsed from SARD. However, most fine-grained VD approaches rely on RNN-based models, which have limited capability to learn long-term code dependencies in lengthy source code functions. To address this challenge, Fu and Tantithamthavorn [1] proposed LineVul which uses a transformer architecture for VD, which retains better long-term dependencies than RNN-based models and utilizes intrinsic self-attention scores to identify line-level vulnerabilities. Alternatively, Hin et al. [108] employed CodeBERT embeddings [40] with GNNs, treating line-level VD as a node classification task. Both approaches focus on C/C++ languages and were evaluated on the Big-Vul benchmark. Similarly, Ding et al. [109] combined a pre-trained transformer model with GNNs to locate C/C++ vulnerabilities at the line level. This approach was evaluated on the synthetic Juliet test suite v1.3 and real-world D2A benchmarks [113]. Dong et al. [305] advanced the use of GNNs for line-level vulnerability detection by incorporating sub-graph embeddings. This approach, evaluated on both real-world projects from the National Vulnerability Database (NVD) and synthetic projects from SARD, builds upon previous GNN-based methods that leveraged graph representations to enhance VD.

In summary, recent advancements in fine-grained, line-level VD showcase a diverse array of methodologies, each addressing specific limitations of earlier approaches. Early works like VulDeePecker [16] introduced the concept of code gadgets to provide a finer-grained representation beyond function levels, while IVDetect [32] used GNNs and GNNExplainer for function-level predictions and fine-grained vulnerability localization. VUDENC’s [297] multi-granularity method further refined this by focusing on vulnerabilities within smaller code segments or slices. More recent approaches have shifted towards leveraging transform-

---

Table 32: Benchmarks used in evaluating AI-driven vulnerability type classification.

Benchmark	Year	Granularity	Programming Language	Real-World	Synthesis	Evaluation Studies
$\mu$ VulDeePecker [308]	2019	Code Gadget	C, C++	✓	✓	[308]
TreeVul [311]	2023	Commit	C, C++, Java, and Python	✓		[311]

ers and advanced embeddings: LineVul Fu and Tantithamthavorn [1] employed transformers to capture long-term dependencies, while CodeBERT and GNNs were combined by Hin et al. [108] for improved line-level predictions. Similarly, Ding et al. [109] integrated transformers with GNNs to enhance vulnerability localization, and Dong et al. [305] advanced GNN-based methods by incorporating sub-graph embeddings. Collectively, these methods highlight a progressive refinement in VD, moving from broad file and function-level predictions to more precise line-level detections, each contributing unique insights into improving vulnerability identification and reducing manual review efforts.

#### 10.4.2.2 Software Vulnerability Classification

AI-driven methods hold the promise of predicting vulnerability types by analyzing the given vulnerable source code. These predictions explain detected vulnerable source code, furnishing developers with valuable insights. By employing this approach, developers can prioritize addressing critical vulnerability types promptly. In practical terms, integrating these AI-driven automation methods directly into developers' Integrated Development Environments (IDEs) has the potential to furnish real-time vulnerability insights during the development stage. This integration aligns with the DevSecOps concept, reflecting the principle of incorporating security into the "Development" step. We summarize the benchmarks used to evaluate the AI-driven vulnerability type classification in Table 32.

**Existing AI methods.** We observed that a notable number of studies utilizing AI-driven methods concentrated on categorizing vulnerability types and characteristics by analyzing the input from vulnerability descriptions [20, 144, 165, 309, 310]. Nevertheless, these studies fall outside the scope of our review because information regarding vulnerability descriptions might not be accessible in the initial phases of software development.

Instead, our focus centers on AI-driven approaches that use plain source code as input to predict vulnerability types, which can explain vulnerability types by scanning developers' source code. Despite efforts, the data imbalance challenge in vulnerability classification persists. In particular, some vulnerabilities

---

such as buffer-related errors are common while other vulnerabilities rarely occur. While Das et al. [20] incorporated data augmentation [44], the performance of their transformer model showed no significant improvement. Similarly, Wang et al. [19] addressed data imbalance by focusing on the top 10 frequency CWE-IDs, yet this approach limited the model’s ability to identify rare vulnerability types.

To mitigate the data imbalance issue, Fu et al. [2] proposed a hierarchical knowledge distillation framework named VulExplainer. VulExplainer involves grouping the imbalanced dataset into subsets based on CWE abstract types, creating more balanced subsets consisting of similar CWE-IDs. Separate TextCNN teacher models are trained for each subset, but they can only predict specific CWE-IDs within their subset. To address this limitation, a comprehensive transformer student model predicting all CWE-IDs is then developed through knowledge distillation. VulExplainer was evaluated using the Big-Vul [70] benchmark consisting of real-world C/C++ projects.

On the other hand, Fu et al. [5] proposed leveraging the pre-trained language model, CodeBERT [40], and trained it with multi-objective optimization (MOO) for CWE-ID classification. Unlike traditional multi-task learning methods, which simply aggregate losses, MOO aims to optimize multiple objectives concurrently, such as CVSS severity score estimation, thereby enhancing the model’s overall performance. The proposed MOO-based model was evaluated using the C/C++ Big-Vul benchmark [70], demonstrating improvements over conventional multi-task learning by more effectively handling correlated tasks.

Pan et al. [311] observed that many methods for identifying security patches in open-source software (OSS) merely confirm the existence of a patch without classifying the specific type of vulnerability it addresses. To address this gap, they introduced TreeVul, a hierarchical and chained architecture designed to leverage the structured nature of CWE-IDs for more nuanced vulnerability classification. TreeVul utilizes CodeBERT for source code representation and multiple LSTM encoders to generate predictions across different levels of the CWE hierarchy. This approach allows TreeVul to offer developers detailed predictions for both high-level and low-level CWE-IDs. Evaluated on a benchmark encompassing C/C++, Java, and Python, TreeVul enhances the ability to pinpoint and address specific vulnerabilities, thus improving the precision of patch management and supporting more targeted security fixes.

Existing deep learning methods for vulnerability detection often only determine

---

whether a program contains vulnerabilities but fail to classify the specific types. Thus, different from the approaches mentioned above, Zou et al. [308] combined vulnerability detection and type classification into an integrated multi-class detection task. Specifically, they introduced  $\mu$ VulDeePecker which introduces the concept of code attention and considers control-dependence when extracting code gadget [16]. The bidirectional LSTM was trained to detect and classify 40 different vulnerability types. They developed a new benchmark from SARD and NVD, featuring 116 CWE-IDs and 33,409 C/C++ programs, to evaluate  $\mu$ VulDeePecker. This benchmark demonstrated that  $\mu$ VulDeePecker significantly improves multiclass classification and detection capabilities by integrating control dependence in addition to data dependence.

In summary, the reviewed studies on AI-driven vulnerability classification address various aspects of the problem but differ in their approaches and focus areas. Some research [20, 144, 165, 309, 310] concentrates on leveraging vulnerability descriptions for classification, which may not always be available in early development stages. Other works such as [2], tackle the issue of data imbalance by introducing hierarchical knowledge distillation to better handle rare vulnerability types. This method creates balanced subsets of CWE-IDs to enhance classification accuracy. In contrast, Fu et al. [5] advances multi-task learning with multi-objective optimization (MOO) to improve CWE-ID classification by optimizing multiple objectives simultaneously, demonstrating its effectiveness over traditional multi-task learning methods. Meanwhile, Pan et al. [311] introduces Tree-Vul, which utilizes a hierarchical approach to classify vulnerabilities across different CWE tree levels, offering developers more detailed and actionable insights. Lastly, Zou et al. [308] integrates vulnerability detection and type classification into a unified multi-class task, using code attention and control-dependence to enhance both detection and classification. Collectively, these approaches complement each other by addressing distinct challenges in vulnerability classification, from data imbalance to granularity of vulnerability types, thus offering more practical and accurate automated vulnerability classification approaches for enhancing software security.

#### 10.4.2.3 Automated Vulnerability Repair

The progress in sequence-to-sequence (seq2seq) learning within the realm of deep learning has facilitated significant advancements, particularly in the development of AI-driven automated programs and vulnerability repair approaches. These innovative solutions now offer the capability to automatically recommend

---

Table 33: Benchmarks used in evaluating AI-driven program and vulnerability repairs.

Benchmark	Year	Programming Language	Real-World	Synthesis	Evaluation Studies
Defects4J [329]	2014	Java	✓		[22, 36, 49, 207, 315, 316]
ManyBugs [330]	2015	C	✓		[49]
BugAID [331]	2016	JavaScript	✓		[49]
QuixBugs [332]	2017	Java, Python	✓		[36, 49]
CodeFlaws [333]	2017	C	✓		[49]
Bugs.jar [334]	2018	Java	✓		[207]
SequenceR [22]	2019	Java	✓		[22]
Bugs2Fix [335]	2019	Java	✓		[317]
ManySStuBs4J [336]	2020	Java	✓		[205]
Hoppity [229]	2020	JavaScript	✓		[229]
CodeXGLUE [156]	2021	Java	✓	✓	[319]
TFix [210]	2021	JavaScript	✓		[210, 319]
VRepair [314]	2022	C, C++	✓		[3, 4, 314]
Namavar, Nashid, and Mesbah [313]	2022	JavaScript	✓		[313]
Pearce et al. [56]	2023	C, C++	✓	✓	[56]
Function-SStuBs4J [321]	2023	Java	✓		[321]
InferFix [320]	2023	Java, C#	✓		[320]

fixes for vulnerable or buggy programs, addressing the time-consuming and labor-intensive nature of manual code repair. Program repair models take source code as input, which allows for potential integration with developers' IDEs, providing near real-time code repair suggestions during the development phase. This integration automates the vulnerable code repair process and incorporates security into the "Development" step. We summarize the benchmarks used to evaluate AI-driven vulnerability and program repair approaches in Table 33.

**Existing AI methods.** Several approaches have been suggested for program repair and vulnerability repair based on deep learning (DL), with program repair targeting general software defects and vulnerability repair addressing security-related weaknesses. Due to the inherent similarities between these tasks, transfer learning can be employed to enhance the model's ability to generalize across both as demonstrated by Chen, Kommrusch, and Monperrus [314]. Consequently, the subsequent discussion encompasses a review of both DL-based program repair and vulnerability repair.

Automated program repair has traditionally relied on static and dynamic analysis techniques like Angelix [337] and CapGen [338], which, despite their progress, are limited to simple, small fixes and depend heavily on intelligent design and domain-specific knowledge. To address these limitations, Chen et al. [22] introduced SequenceR, a language-agnostic, DL-based approach for program repair, focusing specifically on one-line fixes. SequenceR employs an RNN-based sequence-to-sequence (seq2seq) network architecture with a copy mechanism [339] to handle rare or unseen tokens and constructs an abstract buggy context

---

to capture long-range dependencies. It was evaluated using a benchmark curated from CodRep [340] and Bugs2Fix [335], which includes past Java commits from open-source projects and emphasizes the collection of one-line bug fixes.

Recently, considerable research attention has been directed towards employing transformer models for both program repair and vulnerability repair [3, 35, 56, 205, 210, 314, 317–319]. This shift from RNN-based to transformer-based models is driven by the limitations of RNNs in handling long-range dependencies and their sequential processing nature, which can lead to inefficiencies and difficulties in capturing complex code patterns. With their attention mechanisms, transformers [69] allows for the modeling of long-range dependencies more effectively by considering all tokens simultaneously rather than sequentially like RNNs. This parallel processing capability and the ability to capture contextual information across different parts of the code make transformers particularly well-suited for tasks like program repair and vulnerability detection, where understanding the broader context is crucial. Consequently, transformer-based models have demonstrated superior performance and scalability compared to their RNN-based predecessors.

However, transformers are more complex than RNNs in terms of the number of parameters and the architectural intricacies, requiring a greater amount of training data to achieve optimal performance. This requirement poses a challenge, as vulnerability data is significantly scarcer than general bug data. To address this, Chen, Kommrusch, and Monperrus [314] explored the effectiveness of domain adaptation using a vanilla transformer model, demonstrating that pre-training on a larger general bug benchmark followed by fine-tuning on a vulnerability repair dataset can enhance performance. Their approach was evaluated on a curated C/C++ benchmark prepared from the CVEFixes [195] and Big-Vul [70] datasets. Similarly, Chi et al. [35] leveraged a vanilla transformer model with data-flow dependencies as input code sequences as input, which was evaluated on a Java benchmark [341].

Building on the concept of pre-training transformers on extensive datasets, another elegant solution to the data scarcity issue in vulnerability repair involves leveraging pre-trained transformers that have been trained on terabytes of code and natural language. For instance, Mashhadi and Hemmati [205] focused on fixing Java programs using the pre-trained CodeBERT model [40], evaluating their approach on a Java benchmark [336]. Similarly, Fu et al. [3] utilized the CodeT5 model [132] to repair C/C++ vulnerabilities, assessing the effectiveness

---

of language models and different tokenizers on the C/C++ benchmark proposed by Chen, Kommrusch, and Monperrus [314]. These studies illustrate the potential of pre-trained transformers in overcoming the limitations of data scarcity, providing a more robust and scalable solution for automated vulnerability repair and program repair tasks.

Nevertheless, Hao et al. [317] found that current pre-trained transformer code language models (CLMs) for program repairs still have a low success rate. This is primarily because these CLMs are typically developed for general coding purposes, and their potential for program repair applications has yet to be fully explored. To address this, they proposed APRFiT, a general curricular fine-tuning framework designed to improve the success rate of CLMs for program repair. This framework was evaluated on the Bugs2Fix benchmark [335]. On the other hand, Zirak and Hemmati [319] focused on using domain adaptation to enhance existing LM-based approaches such as TFix [210] and CodeXGLUE in cross-project scenarios, evaluating their effectiveness on JavaScript and Java benchmarks [156]. These efforts collectively highlight the need for specialized fine-tuning and domain adaptation strategies to fully realize the potential of pre-trained transformers in program and vulnerability repair tasks.

Fu et al. [4] argued that the majority of vulnerable code consists of only a small portion that needs repair, suggesting that decoders should focus solely on the statements requiring fixes during decoding. Inspired by vision transformers, they introduced a vulnerability masking technique designed to guide the vulnerability repair model toward focusing on vulnerable code blocks during the decoding process. This approach was evaluated on the C/C++ vulnerability benchmark proposed by Chen, Kommrusch, and Monperrus [314]. In a similar vein, Zhu et al. [315] proposed a syntax-guided edit decoder for program repair, which generates edits instead of modified code, offering an efficient representation of small modifications. This method was evaluated on the Defects4J v2.0 benchmark, which consists of Java bugs. These approaches emphasize the importance of targeted repairs and efficient representations for program and vulnerability repairs.

Namavar, Nashid, and Mesbah [313] noted the lack of systematic studies on the effect of code representation on learning performance in program repair. To address this gap, they conducted a controlled experiment to examine the impact of various code representations on model accuracy and usefulness in deep learning-based program repair, using their curated JavaScript benchmark.

---

Despite promising performance, Zhu et al. [316] identified a common limitation among deep learning models: the generation of numerous untypable patches. This issue arises because existing models often do not account for the constraints imposed by typing rules. To mitigate this problem, Zhu et al. [316] introduced Tare, a type-aware model for neural program repair that learns typing rules to generate more accurate patches. This approach was evaluated on the Defects4J [329] and QuixBugs [332] benchmarks, which include Java and Python code, respectively. These studies underscore the importance of code representation and type awareness in enhancing the effectiveness of automated program repair models.

Prior works have introduced context-aware program repair approaches to address the limitations in learning bug-fixing code changes and the context of the surrounding source code. For instance, Li, Wang, and Nguyen [207] proposed DLFix, which employs a tree-based RNN to learn the context surrounding code fixes. This approach was evaluated on the Defects4J [329] and Bugs.jar [334] Java benchmarks, demonstrating its effectiveness in capturing contextual information. Similarly, Lutellier et al. [49] leveraged CNN layers to extract hierarchical features, allowing for better modeling of source code at different granularity levels, such as statements and functions. This method was evaluated on five common benchmarks: Defects4J (Java) [329], QuixBugs (Java, Python) [332], Many-Bugs (C) [330], BugAID (JavaScript) [331], and CodeFlaws (C) [333]. Another notable approach is CURE, proposed by Jiang, Lutellier, and Tan [36], which uses a code-aware search strategy to find correct fixes by focusing on compilable patches and those close in length to the buggy code, thus addressing issues related to the search space and syntax rules. CURE was evaluated on the Defects4J [329] and QuixBugs [332] Java benchmarks. On the other hand, Dinella et al. [229] represented code as a graph and leveraged GNNs to learn the context and fix bugs, showcasing their approach on a proposed JavaScript benchmark. These context-aware approaches collectively highlight the importance of incorporating surrounding code context to enhance the accuracy and effectiveness of automated program and vulnerability repair models.

Certain studies concentrate on addressing specific categories of software bugs or vulnerabilities and repairing warnings issued by static analysis tools automatically. For instance, Utture and Palsberg [342] focused on fixing resource leak warnings and proposed RLFixer, which was evaluated on warnings generated by popular Java resource-leak detectors. Similarly, Marcilio et al. [343] aimed to au-

---

tomatically generate fix suggestions for common warnings issued by static code analysis tools, providing a practical solution to common software issues identified by these tools. Siddiq et al. [312], on the other hand, focused on fixing SQL injection vulnerabilities, highlighting the importance of addressing specific security threats. In a different approach, Pearce et al. [56] investigated the effectiveness of zero-shot large language models (LLMs) for vulnerability repair and evaluated their approach on a collected C/C++ benchmark consisting of both synthetic and real-world vulnerabilities.

Finally, it is worth noting that some studies have developed end-to-end methods for both vulnerability detection and repair [320, 321, 344]. Mesecan et al. [344] presented HyperGI which can detect, localize, and repair information leakage. Ni et al. [321] used multi-task learning to construct a comprehensive end-to-end model for both defect prediction and program repair. This approach was evaluated on their collected Java benchmark, Function-SStuBs4J. Jin et al. [320] proposed InferFix which relies on a static analysis tool, Infer, to detect and locate vulnerabilities, then leveraged large language models (LLMs) to generate corresponding repairs. They also curated a dataset of bugs extracted by executing the Infer static analyzer on the change histories of thousands of Java and C# repositories to evaluate their approach.

#### 10.4.2.4 Security Tools in IDEs.

Static analysis tools rely on predefined patterns to assist developers in identifying potential vulnerabilities in source code. Similarly, integrating deep learning (DL)-based vulnerability prediction approaches into security analysis tools and deploying them to developers' Integrated Development Environments (IDEs) is becoming increasingly feasible. Previous research has shown that DL models can surpass static analysis tools in terms of accuracy in detecting and locating vulnerabilities [1, 30]. Moreover, they exhibit the capability to identify specific vulnerability types and propose corresponding repairs. The incorporation of AI-driven security tools into IDEs streamlines developers' workflows by automating the identification of security issues in their code. Such automation integrates security considerations into the development stage, aligning with the principles of DevSecOps. Below, we explore existing AI-driven security tools within IDEs, examine the challenges they have encountered, and discuss potential avenues for future research.

**Existing AI methods.** We encountered a limited number of search results while

---

exploring AI-driven software security tools. Notably, although deep learning-based approaches have demonstrated superior effectiveness compared to static analysis tools [30], the predominant tools in the current landscape continue to rely on static analysis and predefined patterns. Despite this trend, our investigation uncovered open-source and commercial AI-driven security tools (AIBugHunter [5] and Snyk IDE [264]) designed to assist developers in identifying security issues during development.

Fu et al. [5] introduced AIBugHunter, a deep learning-based security tool designed for C/C++ to assist developers in automating security aspects of their development process. This tool boasts several key capabilities, including the detection and pinpointing of vulnerabilities at the line level, explanation of detected vulnerability types, estimation of vulnerability severity, and suggestions for corresponding repairs. Each of these functionalities is powered by a dedicated language model. AIBugHunter was evaluated on the Big-Vul benchmark [70] and operates as an open-source tool that is accessible as a VSCode extension. According to the results of a user experiment conducted by Fu et al. [5], AIBugHunter can reduce the time spent on detecting, locating, estimating, explaining, and repairing vulnerabilities from 10-15 minutes to a mere 3-4 minutes.

Snyk IDE [264] is a commercial security tool supporting multiple programming languages such as Apex, C++, Go, Java, Kotlin, JavaScript, .NET, PHP, Python, Ruby, Scala, Swift, TypeScript, and VB.NET. Through static code scans, it provides developers with crucial information, pinpointing vulnerabilities, offering explanations of their types, and suggesting actionable fixes. Snyk is powered by DeepCode AI, which consists of multiple AI models trained on security-specific data parsed from millions of open-source projects by security researchers. Moreover, Snyk IDE provides flexible plans, including both free and paid options. It seamlessly integrates as a security plugin across various popular IDEs like JetBrains, Visual Studio Code, Eclipse, and Visual Studio. MongoDB, a leading database technology company, has enhanced its security posture by integrating Snyk into its development workflow [345]. Snyk's dashboard and automated security features are embedded within MongoDB's existing GitHub, Slack, and Jira tools, allowing developers to efficiently manage open-source vulnerabilities. This integration streamlines the identification and remediation of security issues, significantly reducing manual checks and accelerating response times. By leveraging Snyk's automated solutions, MongoDB has achieved greater efficiency in securing third-party dependencies, illustrating the effectiveness of incorporating

Table 34: An overview of AI-driven approaches for security-related tasks in the code commit step.

DevOps Step	Identified Security Task	AI Method	ML	DL	Reference
Code Commit	CI/CD Secure Pipelines	ML algorithms	✓		[74, 348–355]
		Explainable ML	✓		[102]
		RNN		✓	[94, 356]
		Tree-based RNN		✓	[357]
		Transformer		✓	[358]
		JIT-SDP tool	✓		[100, 359]
		Change analysis tool	✓		[360]
		LM		✓	[361]
		LM for code		✓	[362]

security tools directly into developer environments.

Last but not least, GitLab Duo [346] exemplifies an AI-driven approach for enhancing security from the development stage through the CI/CD process in an industrial context. GitLab Duo offers several practical features throughout the software development lifecycle, including automating comprehensive merge request descriptions, providing natural language explanations of code for QA testers, performing root cause analysis of pipeline errors, and facilitating vulnerability resolution. By leveraging generative AI, GitLab Duo helps engineers understand and address vulnerabilities within the development window, minimizing context-switching and ensuring that security measures are seamlessly integrated into the CI/CD process. A notable case study is Cube, a software development company based in the Netherlands, which adopted GitLab Duo to increase efficiency and speed in creating secure software [347]. Cube, serving diverse industries such as energy and real estate, uses GitLab Duo's features like Code Suggestions and AI-powered chat to save time and improve software security. By integrating these AI-driven tools, Cube has enhanced its ability to deliver secure software applications, demonstrating how industrial adoption of generative AI can complement academic approaches to achieve secure software development.

### **10.4.3 Code Commit**

We provide an overview of current AI-driven security approaches for the code commit phase of DevSecOps in Table 34. Below, we delve into each associated security task in detail.

#### **10.4.3.1 Dependency Management**

The issue of vulnerable dependencies is widely recognized in software ecosystems.

---

tems due to the extensive interconnection of free and open-source software libraries [363]. Thus, dependency management is crucial in DevSecOps as it helps developers handle and organize the external components and libraries that a software project relies on to function correctly. This process involves identifying, tracking, and managing the dependencies between different modules, libraries, or packages within a software application. Effective dependency management ensures secure and resilient software development practices in DevSecOps, mitigating the risks associated with vulnerable dependencies. Dependency management includes package managers like npm (Node.js), pip (Python), Maven (Java), and NuGet (.NET), which automate the installation, configuration, and versioning of dependencies. In addition, there exist commercial tools that provide visibility into dependencies and help identify and mitigate security vulnerabilities. For instance, Sonatype Nexus [364] is a repository manager used for managing software components. This tool allows organizations to store and retrieve artifacts securely and efficiently. Nexus supports various package formats and integrates with popular build tools and CI/CD pipelines, providing dependency management, security scanning, and artifact promotion capabilities. Black Duck [365] is a software composition analysis (SCA) platform designed to manage risks linked to open-source and third-party software components. Black Duck scans codebases, detects open-source components, and assesses their compliance with licenses, security vulnerabilities, and overall quality. It integrates with development tools for proactive dependency management. Snyk [366] also offers a dependency management tool, which is a developer-centric approach to application security, seamlessly integrating into existing DevOps workflows. This tool offers a Dependency Tree View for identifying dependencies and their vulnerabilities, automatically updating them as they evolve. With features like automated scanning within IDEs, comprehensive vulnerability databases, and prioritized remediation, Snyk empowers developers to proactively manage their risk exposure and maintain the integrity of their software projects.

Despite the critical role of effective dependency management, our review of AI-driven security approaches encountered challenges in finding relevant literature. Thus, we have introduced common dependency management practices. In RQ2, we will highlight AI-driven approaches found outside of academic literature followed by future research directions, suggesting how researchers can build upon these industry practices to advance AI-driven methods for dependency management within DevSecOps.

---

Table 35: Benchmarks used in evaluating AI-driven just-in-time (JIT) software defect prediction.

Benchmark	Year	Granularity	Programming Language	Real-World	Synthesis	Evaluation Studies
PROMISE [367]	2007	Commit	Java	✓		[357]
Kamei et al. [368]	2012	Commit	C, C++, Java, JavaScript, Perl	✓		[348]
Qt & OpenStack [369]	2018	Commit/Line	C++, Python	✓		[74, 102]
Cabral et al. [350]	2019	Commit/File	Java, JavaScript, Python	✓		[350]
Yan et al. [353]	2020	Commit/File	Java	✓		[353, 358]
Wattanakriengkrai et al. [75]	2020	Commit	Java	✓		[94]
Suh [355]	2020	Commit/File	JavaScript, PHP	✓		[355]

#### 10.4.3.2 CI/CD Secure Pipelines

In the Code Commit step of DevSecOps, ensuring the security of the Continuous Integration/Continuous Deployment (CI/CD) pipeline is crucial. A Secure CI/CD Pipeline involves implementing security measures, starting when code is committed [370]. This may include integrating security checks, such as Just-in-Time (JIT) defect prediction approaches, to identify potential vulnerabilities in code changes as soon as they are submitted. The accuracy of these JIT defect prediction models can be influenced by the methods used to identify defects in the underlying datasets, as different approaches may yield varying predictive performance [371]. Additionally, establishing robust issue-report-to-fix-commit links enhances security by facilitating the swift resolution of reported issues, ensuring that security fixes are promptly applied to the codebase. By leveraging AI-driven techniques in JIT defect prediction and issue-report-to-fix-commit links, organizations can proactively address security concerns at the code commit stage, aligning with the principles of DevSecOps. We summarize the benchmarks used to evaluate the JIT defect prediction in Table 35.

**Existing AI methods.** With the increased interest in continuous deployment, a variant of software defect prediction called Just-in-Time (JIT) Software Defect Prediction (SDP) focuses on predicting whether each incremental software change (e.g., a commit) is defective [372]. JIT-SDP typically analyzes project metrics and code metrics, such as code complexity, lines of code, code churn, and the developer's experience, all collected from historical data of past commits. By leveraging these factors, teams can identify potential issues before they escalate. Integrating JIT-SDP into the CI/CD pipeline enables teams to proactively detect security vulnerabilities in code commits, such as improper input validation or insecure coding practices, allowing for timely mitigation and remediation.

A majority of studies have focused on building ML models for JIT-SDP. For instance, Chen et al. [348] proposed the supervised method MULTI, which aims to reduce manual effort from developers by maximizing the number of identified

---

buggy changes while minimizing efforts in software quality assurance activities. MULTI was evaluated on a benchmark consisting of five programming languages (i.e., C, C++, Java, JavaScript, and Perl) from six open-source projects: Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla, and PostgreSQL. Similarly, Li et al. [349] leveraged semi-supervised learning with a greedy strategy in unit code inspection effort to rank changes according to their tendency to be defect-prone. This approach was evaluated on the same benchmark as MULTI [348]. While both studies focus on the same task of JIT-SDP and aim to reduce manual effort, they explore different methodologies: MULTI uses multi-task supervised learning, whereas Li et al. [349] employs a semi-supervised approach.

Addressing challenges such as the class imbalance problem has also been a focus. Cabral et al. [350] proposed a class imbalance learning algorithm to improve existing ML models for JIT-SDP and evaluated on their collected benchmark with ten open-source GitHub projects including Java, JavaScript, and Python languages. Alternatively, Tessema and Abebe [351] suggested using change request information to further enhance ML-based JIT-SDP.

In the context of JIT-SDP, addressing challenges like the class imbalance problem is crucial for improving prediction accuracy, as highlighted by Cabral et al. [350] and Tessema and Abebe [351]. However, as ML models become increasingly complex, the need for explainability in these models has grown significantly. Explainable JIT-SDP is essential because it allows developers to understand the reasoning behind predictions, which can build trust in the models and facilitate the debugging and refinement process. Moreover, understanding the model's decision-making process can help in identifying and addressing biases or errors in the predictions, ultimately leading to more reliable software quality assurance. This connection to explainable AI is exemplified by the PyExplainer tool proposed by Pornprasit et al. [102], who explored rule-based explanations [373] to explain the predictions made by black-box ML models. PyExplainer was evaluated on a benchmark of two large-scale open-source software projects (i.e., Openstack and Qt) provided by McIntosh and Kamei [369]. The explainability of ML models not only enhances their usability but also aligns with the overarching goal of reducing manual effort in software quality assurance by making the models' outputs more interpretable and actionable for developers.

Fine-grained JIT-SDP models have emerged to address the limitations of traditional JIT-SDP by providing more precise bug localization, which aims to identify not just buggy commits but also the specific files or lines within a commit that

---

are defective. These models enhance the efficiency of debugging and reduce the manual effort required by developers. For instance, Pascarella, Palomba, and Bacchelli [352] proposed a method to locate buggy files, while Pornprasit and Tantithamthavorn [74] and Yan et al. [353] focused on identifying buggy lines within a commit by ranking the prediction scores of ML models. Yan et al. [353] evaluated their bug localization method on a benchmark consisting of Java programs. In contrast, Pornprasit and Tantithamthavorn [74] assessed their JITLine approach using a benchmark that included projects from both the Qt and Open-Stack ecosystems, as detailed by McIntosh and Kamei [369].

Furthermore, Qiu et al. [358] and Pornprasit and Tantithamthavorn [94] leveraged Recurrent Neural Networks (RNNs) to capture the semantic information of source code, aiming to locate buggy files within a commit and subsequently localize defective lines by ranking prediction scores. These approaches were evaluated on the Java benchmark prepared by Yan et al. [353] and the benchmark by Wattanakriengkrai et al. [75], respectively. Dam et al. [357] utilized an abstract syntax tree (AST) to represent source code and employed tree-based LSTM networks to effectively capture both the syntax and semantics at multiple levels. This method was evaluated on the PROMISE benchmark [367], which consists of Java projects, demonstrating the potential for more accurate prediction models through a deeper understanding of source code structure and semantics.

Some prior works developed JIT-SDP tools for integration into CI/CD pipelines. For example, Qiu et al. [359] proposed JITO, integrated into IntelliJ IDEA, to detect defective code changes and locate exact defect lines. Similarly, Khanan et al. [100] developed JITBot, integrated into CI/CD pipelines, providing defect prediction in GitHub commits along with explanations to clarify reasons and mitigation plans. In addition, Mehta et al. [360] introduced Rex, a change analysis tool that leverages machine learning (ML) models and program analysis. Rex learns change rules that capture dependencies between different regions of code or configuration, based on patterns observed in commit logs spanning several months. When an engineer modifies a subset of files within a change rule, Rex suggests additional changes to ensure consistency and completeness. The tool has been effectively implemented within services such as Office 365 and Azure, impacting over 5,000 changes in the system.

On the other hand, issue-report-to-fix-commit links are critical for security, aiding in understanding code changes and assessing security implications. In partic-

---

ular, establishing robust links between issues and fixes helps in tracking which code changes address specific vulnerabilities or bugs. This traceability ensures that identified issues are addressed systematically and that fixes are correctly applied, which is essential for maintaining security [361]. Manual maintenance of these links is error-prone, potentially leading to vulnerabilities. Automatic link recovery methods have been proposed, but traditional classifiers such as Relink [374] may struggle due to limited positive links and dependency on their number for generating negative links. To address this, Sun et al. [354] formulated the missing link problem as a model learning problem and trained a machine learning (ML) classifier. In contrast, Ruan et al. [356] proposed leveraging recurrent neural networks (RNNs) to learn the semantic representation of natural language descriptions and code in issues and commits, and the semantic correlation between issues and commits. Recently, Lan et al. [361] suggested using language models such as BERT to further enhance performance. However, Zhang et al. [362] argued that the substantial size of language models like CodeBERT poses challenges in terms of computational resources and efficiency. To address this, they proposed a knowledge distillation approach to transfer the knowledge from the large CodeBERT model to a more compact model. This distilled model aims to retain competitive performance while significantly reducing the computational demands associated with training and deploying large-scale language models.

Prior works have proposed multiple AI-driven approaches to automatically identify security-related commits. For instance, Suh [355] used machine learning models such as Support Vector Machine (SVM) to predict whether a commit is likely to be reverted based on features extracted from the revision history of a codebase. This approach was motivated by the need to address the challenges associated with continuous integration and deployment at scale, where quickly identifying time-sensitive bugs is crucial. Evaluated on a benchmark collected from the Wayfair website repositories, which include code written in PHP, JavaScript, and Mustache templates, this method helps streamline the detection of problematic commits and enhances the overall efficiency and security of the development process.

#### 10.4.4 Build, Test, and Deployment

We provide an overview of current AI-driven security approaches for the build, test, and deployment phase of DevSecOps in Table 36. Below, we delve into each associated security task in detail.

Table 36: An overview of AI-driven approaches for security-related tasks in build, test, and deployment steps.

DevOps Step	Identified Security Task	AI Method	ML	DL	Reference
Build, Test, and Deployment	Configuration Validation	ML algorithms FFNN GAN	✓ ✓ ✓	✓ ✓	[375–377] [378] [379, 380]
	Infrastructure Scanning	ML algorithms Word2Vec-CBOW	✓	✓	[381–383] [384]

Table 37: Benchmarks used in evaluating AI-driven configuration validation.

Benchmark	Year	Real-World	Synthesis	Evaluation Studies
SPLConqueror [385]	2015	✓		[376, 378, 379]
ACTGAN [380]	2019	✓		[380]

#### 10.4.4.1 Configuration Validation

Configuration validation is a critical aspect of DevSecOps. It ensures that the configurations of software systems, including parameters and settings, are accurate, optimal, and secure. Common approaches to configuration validation include manual inspection, automated testing, and performance estimation models. The importance of configuration validation lies in its direct impact on system reliability, performance, and security. Misconfigurations can lead to vulnerabilities and system failures, hence making robust validation processes essential for safeguarding software systems against potential threats [386]. We summarize the benchmarks used to evaluate the AI-driven configuration validation in Table 37.

**Existing AI methods.** Manual configuration tuning in complex software systems presents a significant security challenge due to the extensive array of parameters for users to configure. However, understanding the intricacies of the software required for effective tuning often exceeds typical user capabilities [380]. This knowledge gap increases the risk of misconfigurations, leaving systems vulnerable to security breaches. To address this challenge, leveraging AI-driven performance estimation can provide valuable insights into the impact of various configuration settings on system performance and security, empowering users to make informed decisions.

In particular, deep learning-based approaches have been proposed. Ha and Zhang [378] proposed DeepPerf, which uses a deep feedforward neural network with sparsity regularization and efficient hyperparameter tuning to predict performance with high accuracy from a small set of configurations. DeepPerf

---

was evaluated on a real-world benchmark [385] including systems such as Dune MGS, HIPAcc, and HSMGP, among others. These systems span various domains, are written in different programming languages, and have diverse configurations, both binary and numeric, which support configuration at either compile time or load time. Shu et al. [379] proposed Perf-AL, which employs generative networks with adversarial learning, comprising a generator and a discriminator. These networks iteratively refine the prediction model through competition until the predicted values converge toward the ground truth distribution. Perf-AL was evaluated on the benchmark presented by Siegmund et al. [385]. Additionally, Cheng, Ying, and Wang [375] proposed to combine multi-objective optimization and a performance prediction model to search for an optimal configuration for Spark deployed in the public cloud.

Recently, Xia, Ding, and Shang [376] introduced CoMSA, a Modeling-driven Sampling Approach based on XGBoost, which prioritizes configurations with uncertain performance predictions for further training. CoMSA is adaptable to scenarios with or without historical performance testing results because not all software projects maintain such records. CoMSA was evaluated on a similar benchmark as DeepPerf including software systems such as LRZIP, LLVM, x264, and SQLite. Many existing approaches struggle with performance estimation due to the challenge of limited sample sizes, which impacts model accuracy, especially under tight tuning time constraints. Instead of solely focusing on improving the performance estimation models, Bao et al. [380] proposed ACTGAN which aims to capture hidden structures within good configurations and use this knowledge to generate potentially better configurations. ACTGAN was evaluated on their collected benchmark including eight widely used software systems like Kafka and Spark.

On the other hand, container orchestrators (CO) are crucial for managing container clusters in virtualized infrastructures. Securing CO is challenging due to numerous configurable options and manual configuration is prone to errors and time-consuming. Thus, Haque, Kholoosi, and Babar [377] proposed KGSecConfig, a machine learning-based approach for automating the security configuration of Container Orchestrators (CO), such as Kubernetes, Docker, Azure, and VMWare. It leverages knowledge graphs to systematically capture, link, and correlate heterogeneous and multi-vendor configuration options into a unified structure. By employing keyword and learning models, KGSecConfig enables the automated extraction of secured configuration options and concepts, aiding in

---

Table 38: Benchmarks used in evaluating AI-driven infrastructure security scanning.

Benchmark	Year	Real-World	Synthesis	Evaluation Studies
Rahman and Williams [381]	2018	✓		[381]
Rahman and Williams [382]	2019	✓		[382]
Dalla Palma et al. [383]	2021	✓		[383]
Borovits et al. [384]	2022		✓	[384]

the mitigation of misconfigurations in CO environments.

#### 10.4.4.2 Infrastructure Scanning

In DevSecOps, Infrastructure Scanning plays a crucial role in ensuring the security and compliance of software systems. With the rise of Infrastructure as Code (IaC) tools like Ansible, Chef, and Puppet, the process of provisioning and configuring infrastructure has become more automated and scalable. These tools allow developers and operations teams to define infrastructure configurations as machine-readable code, enabling consistent, repeatable deployments across environments. Thus, IaC is a key DevOps practice and a component of continuous delivery [387]. However, during the development of IaC scripts, practitioners might unknowingly introduce security smells (e.g., hard-coded passwords). These recurring coding patterns signal security weaknesses that could lead to security breaches [388]. By integrating AI-driven IaC methods, organizations may proactively identify and mitigate security risks in their IaC scripts. This saves developers' manual security inspection efforts and strengthens the overall security posture of their systems. We summarize the benchmarks used to evaluate the AI-driven IaC scanning in Table 38.

**Existing AI methods.** Similar to other source code artifacts, Infrastructure as Code (IaC) scripts may contain defects that hinder their proper functionality. To automate the defect prediction process and reduce manual inspection, Rahman and Williams [381] leveraged text-mining techniques, such as Bag-of-Words (BoW) and TF-IDF, to extract features from IaC scripts and predict defective ones using machine learning (ML) models. They evaluated their method on their collected benchmark including Puppet IaC scripts. Rahman and Williams [382] further conducted qualitative analysis on defect-related commits extracted from open-source software repositories to identify source code characteristics correlated to defective IaC scripts. They evaluated their IaC defect prediction model on their collected benchmark including open-source repositories maintained by Mirantis, Mozilla, OpenStack, and Wikimedia Commons. They then surveyed practitioners to gauge their agreement with the identified characteristics, using

Table 39: An overview of AI-driven approaches for security-related tasks in the operation and monitoring step.

DevOps Step	Identified Security Task	AI Method	ML	DL	Reference
Operation and Monitoring	Log Analysis and Anomaly Detection	ML algorithms RNN RNN-based AE GNN Transformer Explainable DL Diffusion Model	✓		[389–394] [393, 395–402] [401, 403] [404] [405] [406, 407] [408]
	Cyber-Physical Systems	ML algorithms RNN + GNN GAN VAE Transformer LM + RNN	✓		[409, 410] [411] [412] [413] [414] [415]

them as features to construct their ML IaC defect prediction model. Similarly, Dalla Palma et al. [383] also suggested an ML-based technique to predict defects in IaC scripts. Their models rely on various metrics, such as lines of code, IaC-specific metrics like the number of configuration tasks, and process metrics such as the number of commits to a file, computed from the collected IaC scripts to predict their proneness to failure. The study was particularly implemented and targeted for their collected Ansible-based benchmark.

On the other hand, Borovits et al. [384] focused on the linguistic anti-patterns in IaC. Linguistic anti-patterns are recurring poor practices concerning inconsistencies in the naming, documentation, and implementation of an entity. They impede the readability, understandability, and maintainability of source code. In particular, they proposed FINDICI, a deep learning (DL)-based approach for anti-pattern detection in IaC. They build and use the abstract syntax tree of IaC code units to create code embeddings used by DL models to detect inconsistent IaC code units. They also evaluated their approach using an Ansible-based synthesis benchmark.

#### 10.4.5 Operation and Monitoring

We provide an overview of current AI-driven security approaches for the operation and monitoring phase of DevSecOps in Table 39. Below, we delve into each associated security task in detail.

---

Table 40: Benchmarks used in evaluating AI-driven log analysis and anomaly detection.

Benchmark	Year	Real-World	Synthesis	Evaluation Studies
Yahoo! Webscope [416]	2006	✓	✓	[390]
BGL [417]	2007	✓		[393, 399, 400, 402, 405]
HDFS [418]	2009	✓		[393, 395–397, 399–404]
ADFA-LD [419]	2013		✓	[389]
SDS [420]	2015		✓	[390]
UNSW-NB15 [421]	2015		✓	[403]
OpenStack [395]	2017	✓		[395, 399]
Microsoft [396]	2019	✓		[396]
LogHub [422]	2020	✓		[391, 402]
Studiawan, Sohel, and Payne [398]	2020	✓		[398]
Yang et al. [392]	2023	✓		[392]

#### 10.4.5.1 Log Analysis and Anomaly Detection

AI-driven approaches are pivotal in the detection and mitigation of anomalies in system logs. These techniques enable organizations to effectively identify irregularities and address potential issues. Additionally, explainable AI (XAI) contributes significantly by elucidating the underlying causes of anomalies, thus supporting informed decision-making. The scope of AI-driven anomaly detection extends beyond system logs to include cloud services, enhancing operational resilience and aligning with the principles of the DevSecOps paradigm. We summarize the benchmarks used to evaluate the AI-driven log analysis and anomaly detection in Table 40. The following sections present a comprehensive literature review on these topics.

**Existing AI methods.** Some studies have concentrated on machine learning (ML) models such as support vector machines (SVM) due to their lower computation and time requirements compared to deep learning (DL) models. For instance, Khreich et al. [389] integrated frequency and temporal information from system call traces using a one-class SVM, which preserves temporal dependencies among these events. The proposed approach was evaluated on the ADFALD (Australian Defence Force Academy Linux Dataset) benchmark with synthesis data based on real-world attacking scenarios.

Moreover, Cid-Fuentes, Szabo, and Falkner [390] observed that certain anomaly detectors rely on historical failure data and cannot adapt to changes in system behavior at runtime. To address this limitation, they developed a model of system behavior at runtime using SVM, thereby eliminating the need for historical failure data and enabling adaptation to behavior changes. They evaluated their approach on the SDS (Synthetic Distributed System) [420] and Yahoo! Webscope benchmarks [416] consisting of both synthetic and real-world data. Han

---

et al. [391] found that many machine learning algorithms, such as SVM and Logistic Regression, assume clean data and suffer from high training times. Thus, they proposed the Robust Online Evolving Anomaly Detection (ROEAD) framework, which incorporates a Robust Feature Extractor (RFE) to adapt to noise and an Online Evolving Anomaly Detection (OEAD) component for dynamic parameter updates. Their proposal was evaluated on the LogHub [422] benchmark. More recently, Yang et al. [392] used traditional Principal Component Analysis (PCA) and achieved comparable effectiveness to advanced supervised/semi-supervised DL-based techniques while demonstrating better stability under insufficient training data. Their approach was evaluated on their collected benchmark consisting of HDFS logs from Hadoop jobs [418], BGL logs from the Blue Gene/L supercomputer [417], Spirit logs from the Spirit supercomputer [423], NC logs from a university network center, and MC logs from a motor corporation’s online service system.

Due to recent advancements in deep learning (DL), numerous studies on log-based anomaly detection have introduced various DL-based approaches. For instance, Du et al. [395] proposed DeepLog, one of the pioneering Recurrent Neural Network (RNN) models that treat system logs as natural language sequences to autonomously learn patterns from normal execution log files. DeepLog was evaluated on HDFS [418] and their curated OpenStack log benchmarks. Following this, several other RNN-based detectors emerged, including those by Zhang et al. [396], who leveraged attention-based Bi-LSTM, and Meng et al. [397], who used the same architecture with their proposed template2Vec to extract semantic and syntax information from log templates. The former was evaluated on the HDFS [418] benchmark and log data collected from Microsoft’s system, while the latter was evaluated on the HDFS benchmark. Studiawan, Sohel, and Payne [398] employed Gated Recurrent Units (GRU) alongside sentiment analysis to identify negative sentiment indicative of anomalous activities in operating system (OS) logs. They evaluated their approach on the collected benchmark from four public datasets of OS and system logs: nps-2009-casper-rw, DFRWS 2009, Honeynet Forensic Challenge 7 2011, and BlueGene/L. Furthermore, Zhou et al. [399] proposed to use sentence embedding in their DeepSyslog method to capture the contextual and semantic relationships between log events, addressing the limitations of traditional methods that rely on incomplete and unstructured log data, which often leads to missed anomalies and false alarms. DeepSyslog was evaluated on HDFS [418], BGL [417], and OpenStack [395] benchmarks. Wang et al. [400] introduced an online learning paradigm and used LSTM to handle

---

incoming new and unstable log data and evaluated it on HDFS [418] and BGL [417] benchmarks. In addition, Yuan, Liu, and Zhu [403] leveraged LSTM-based autoencoder to reconstruct discrete event logs and showed that their approach can detect not only sequences that include unseen or rare events but also structurally abnormal sequences, addressing the fundamental limitation of predicting upcoming events which often results in high false positives due to an inability to fully exploit sequence characteristics. They evaluated their approach on UNSW-NB15 [421] traffic logs and HDFS console logs [418].

Du et al. [401] focused on the challenge of continuously updating anomaly detection systems with new information over time. They introduced a lifelong learning framework called unlearning, which adjusts the model upon labeling false negatives or false positives post-deployment. This framework addressed both the challenge of exploding loss in anomaly detection and catastrophic forgetting in lifelong learning. They evaluated their approach on multiple benchmarks such as HDFS [418] and Yahoo Network Traffic [424]. Furthermore, Li et al. [402] identified challenges in analyzing interleaved logs in modern distributed systems. They proposed SwissLog as a solution to these challenges. The issues include the absence of log dependency mining, variability in log formats, and difficulty in non-intrusive performance issue detection. SwissLog tackles these by constructing ID relation graphs, grouping log messages by IDs, using an online data-driven log parser, and applying an attention-based Bi-LSTM model and heuristic searching algorithm for anomaly detection and localization. SwissLog was evaluated on common log benchmarks such as HDFS [418] and LogHub [422].

Some existing approaches, as discovered by Le and Zhang [405], rely on a log parser to convert log messages into log events, which are then used to create log sequences. However, errors in log parsing can negatively impact the performance of anomaly detectors based on unsupervised or supervised machine learning models. Thus, they introduced NeuralLog, which employs a transformer architecture and eliminates the need for log parsing. They used subword tokenization to address the out-of-vocabulary (OOV) issue. NeuralLog was evaluated on BGL [417] and LogHub [422] benchmarks. Instead of treating system logs as natural language sequences which might reduce anomaly detectors' sensitivity to the log flaws and the concurrency of multiple anomalies, Li et al. [404] transformed log record sequences into log event graphs using event semantic embedding and event adjacency matrix. An attention-based Gated Graph Neural Network (GGNN) model was then used to capture semantic information for

---

Table 41: Benchmarks used in evaluating AI-driven cyber-physical system (CPS) security approaches.

Benchmark	Year	Real-World	Synthesis	Evaluation Studies
Gas Pipeline Dataset [427]	2015	✓		[411]
SWaT [428]	2016	✓		[410, 411]
WADI [429]	2017	✓		[411]
BATADAL [430]	2018		✓	[411]
MSDS [431]	2023	✓		[414]

anomaly identification. The proposed approach was evaluated on the HDFS [418] benchmark. Finally, Wu, Li, and Khomh [393] presented a comprehensive study investigating the effectiveness of different representations used in machine learning and deep learning models for log-based anomaly detection using common benchmarks such as HDFS [418] and BGL [417].

Some studies have delved into explainable AI (XAI) for anomaly detection in securing software systems. For instance, Han et al. [406] introduced an interpretation methodology tailored for unsupervised deep learning models specifically designed for security systems. Their approach formulates anomaly interpretation as an optimization problem, seeking to identify the most significant differences between anomalies and a normal reference. Furthermore, the interpretations underwent validation through feedback from human security experts. Additionally, Aguilar et al. [407] proposed a decision tree-based autoencoder aimed at anomaly detection, which offers insights into its decisions by exploring correlations among various attribute values.

Given the complexity of managing diverse services in cloud environments, there is a need for automated anomaly detection mechanisms that are easy to set up and operate without requiring extensive knowledge of individual services [394]. For instance, Sauvanaud et al. [394] leveraged machine learning models to aid providers in diagnosing anomalous virtual machines (VMs). Recently, Lee et al. [408] proposed Maat, a framework for anticipating cloud service performance anomalies based on a conditional diffusion model [425]. Maat adopts a two-stage paradigm for anomaly anticipation, consisting of metric forecasting and anomaly detection on forecasts. It employs a conditional denoising diffusion model for multi-step forecasting and extracts anomaly-indicating features based on domain knowledge, followed by the application of isolation forest [426] with incremental learning to detect upcoming anomalies, thus uncovering anomalies that better conform to human expertise.

---

#### 10.4.5.2 Cyber-Physical Systems

Cyber-physical systems (CPS) integrate sensing, computation, control, and networking into physical objects and infrastructure, establishing connections among them and with the Internet to facilitate seamless interaction and automation [432]. Given the critical need for high security in CPS to ensure safe operation, anomaly detection, relying on data analysis and learning, emerges as a key security technology. The principles of DevSecOps, prioritizing security at every stage of software development, intersect with CPS, particularly when software interacts with physical systems or infrastructures. In such scenarios, the integration of AI-driven approaches for anomaly detection and security enhancement in CPS aligns with DevSecOps goals, aiming to seamlessly integrate security into the development and deployment processes. We summarize the benchmarks used to evaluate the AI-driven CPS security approaches in Table 41.

**Existing AI methods.** Prior works have proposed leveraging Digital Twins, which are digital replicas of physical entities [433], to train ML/DL models for anomaly detection. Digital twins are particularly useful for anomaly detection in cyber-physical systems (CPS) due to their ability to create virtual replicas of physical systems. For example, LATTICE [411] is a digital twin-based anomaly detection method employing deep curriculum learning. It assigns difficulty scores to each sample and uses a training scheduler to sample batches of training data based on these scores, facilitating learning from easy to difficult data. This approach has shown to be more effective than their previous DL-based proposal, ATTAIN [412]. LATTICE was evaluated on multiple CPS benchmarks including Secure Water Treatment (SWaT) [428], Water Distribution (WADI) [429], Battle of Attack Detection Algorithms (BATADAL) [430], Gas Pipeline Dataset [427]. Additionally, Xu et al. [415] identified challenges related to data complexity and insufficiency within CPS, particularly in train control and management systems. Consequently, they proposed employing a language model (LM) with an LSTM architecture to understand complex data, supplemented by a knowledge distillation technique to learn from out-of-domain datasets, addressing the issue of data insufficiency. They evaluated their approach using two TCMS (train control management system) benchmarks.

On the other hand, Xi et al. [409] found that existing anomaly detection methods in CPS, such as AutoEncoder (AE) [434] or Generative Adversarial Network (GAN) [435], often overlook implicit correlations between data points, like the relationship between vehicle speed and obstacle position in the Intelligent Cruise Control

---

System (ICCS), resulting in suboptimal performance. Hence, they proposed an adaptive unsupervised learning method incorporating a Gaussian Mixture Model (GMM), dynamically constructing and updating data correlations via KNN and dynamic graph techniques. They evaluated their approach using benchmarks related to smart grids and smart home systems. Lin et al. [410] focused on Industrial Control Systems (ICS) such as water and power and leveraged the Bayesian network to discover dependencies between sensors and actuators and recognize irregular dependencies. Lin et al. [410] focused on Industrial Control Systems (ICS) like water and power systems, leveraging Bayesian networks to uncover dependencies between sensors and actuators and identify irregularities. They evaluated their approach on the SWaT [428] benchmark, demonstrating its effectiveness in recognizing abnormal dependencies.

On the other hand, microservice anomaly detection is vital for system reliability in a microservice architecture. Xie et al. [413] focused on anomaly detection in traces within a microservice architecture. Traces record inter-microservice invocations and are essential for diagnosing system failures. They suggested a group-wise trace anomaly detection algorithm, which categorizes traces based on shared sub-structures and employs a group-wise variational autoencoder to obtain structural representations, effectively reducing system detection overhead and outperforming existing methods that analyze each trace individually without considering the structural relationships between them. They evaluated their proposed approach on a real-world benchmark from eBay's microservices system. Huang et al. [414] claimed that the main challenge arises from integrating multiple data modalities (e.g., metrics, logs, and traces) effectively. To address this, they proposed extracting and normalizing features from metrics, logs, and traces, integrating them using a graph representation called MST (Microservice System Twin) graph. A transformer architecture with spatial and temporal attention mechanisms is then employed to model inter-correlations and temporal dependencies, enabling accurate anomaly detection. They evaluated their approach on the MSDS benchmark [431], which includes distributed traces, application logs, and metrics from an OpenStack-based AI analytics system.

## 10.5 RQ2: Challenges and Research Opportunities in AI-Driven DevSecOps

In the previous section, we reviewed existing AI-driven security methodologies and tools for DevSecOps to address our RQ1. Now, we will introduce themes of challenges encountered by prior studies and derive future research opportu-

Table 42: (RQ2) The overview of the 15 challenges and future research opportunities derived from previous studies.

DevOps Step	Identified Security Task	Themes of Challenges	Research Opportunity
Plan	Threat Modeling Software Impact Analysis	- -	- -
Development	Software Vulnerability Detection Software Vulnerability Classification Automated Vulnerability Repair Security Tools in IDEs	C1-1 - Data Imbalance C4 - Cross Project C5 - MBU Vulnerabilities C6 - Data Quality C1-2 - Data Imbalance C7 - Incompleted CWE Tree C2-1 - Model Explainability C8 - Sequence Length and Computing Resource C9 - Loss of Pre-Trained Knowledge C10 - Automated Repair on Real-World Scenarios C3-1 - Lack of AI Security Tooling in IDEs	R1-1 - Data augmentation and logit adjustment R4 - Evaluate cross-project SVD with diverse CWE-IDs R5 - Evaluate SVD on MBU vulnerabilities R6 - Address data inaccuracy from automatic data collection. R1-2 - Meta-learning and LLMs R7 - Develop advanced tree-based SVC R2-1 - Evidence-based explainable AI (XAI) R8 - Explore transformer variants that can process longer sequences R9 - Explore different training paradigms during fine-tuning R10 - Address limitations of LLMs R3-1 - AI tool deployment and comprehensive tool evaluation
Code Commit	CI/CD Secure Pipelines	C2-2 - Model Explainability C3-2 - Lack of AI Security Tooling in CI/CD C11 - The Use of RNNs	R2-2 - Explainable AI (XAI) for DL Models R3-2 - AI tool deployment in CI/CD pipelines R11 - Explore LMs and LLMs
Build, Test, and Deployment	Configuration Validation Infrastructure Scanning	C12 - Complex Feature Space C3-3 - Lack of AI Security Tooling for Infrastructure Scanning C13 - Manual Feature Engineering	R12 - Explore transformer for tabular data R3-3 - AI tool deployment and post-deployment evaluation R13 - Explore DL-based techniques
Operation and Monitoring	Log Analysis and Anomaly Detection Cyber-Physical Systems	C2-3 - Model Explainability C14 - Normality Drift for Zero-Positive Anomaly Detection C15 - Monitoring Multiple Cyber-Attacks Simultaneously	R2-3 - Explainable AI (XAI) for ML Models R14 - Enhance normality drift detection R15 - Distributed anomaly detection and multi-agent systems

nities to answer our RQ2. To begin, we found that some of these challenges are shared across multiple security tasks related to the DevOps process. Thus, we will first illustrate these common challenges along with their associated research opportunities in the following section. After that, we will proceed to introduce challenges and opportunities specific to each security task in the DevOps process. For clarity, we will use “C” followed by a number to represent a challenge and “R” followed by a number to represent the corresponding research opportunity. Our answers to RQ2 are summarized in Table 42.

### 10.5.1 Common Challenges

We identified three common challenges: (C1) Data Imbalance, (C2) Model Explainability, and (C3) Lack of AI Security Tooling. Based on our investigation of previous literature, we found that these challenges are shared by multiple security tasks related to the DevOps process. In what follows, we introduce these common challenges and discuss the associated research opportunities.

**C1-1 - Data Imbalance in Software Vulnerability Detection (In DevOps Development).** Chakraborty et al. [14] observed that the performance of deep learning (DL)-based VD approaches could drop 73% of the F1-score due to the data imbalance issue. Thus, Yang et al. [436] further investigated the impact of data sampling on the effectiveness of existing state-of-the-art (SOTA) DL-based VD approaches. Their discovery revealed that, in DL-based VD, employing over-sampling proves more beneficial than under-sampling. Despite this observation, their experimental findings indicate a persistent challenge: a notable proportion of cases (ranging from 33% to 58%) where decisions were not determined by the presence of vulnerable statements. Consequently, the issue of data im-

---

balance persists, with models continuing to prioritize non-vulnerable code statements when arriving at a vulnerable decision.

**R1-1 - Data Augmentation and Logit Adjustment.** Regarding the research opportunities of data imbalance, Yang et al. [436] suggested that future research explore data augmentation, emphasizing its potential value. Their results indicated that employing a straightforward repetition strategy could enhance the performance of models. Furthermore, the issue of data imbalance is also recognized in the computer vision domain, and proven methods like logit adjustment [148] have demonstrated effectiveness in addressing imbalances in image classification. The application of such methods holds the potential to improve the performance of vulnerability detection.

**C1-2 - Data Imbalance in Software Vulnerability Classification (In DevOps Development).** We found that current vulnerability classification methods still suffer from the data imbalance challenge where models have limited performance on the vulnerability types that infrequently occur. For instance, VulExplainer [2] can correctly identify 67%-69% for common CWE-IDs while the performance drops to 49%-56% for rare CWE-IDs. Moreover, the MOO-based vulnerability classification [5] could not correctly identify some of the infrequent vulnerabilities such as CWE-94 (Improper Control of Generation of Code).

**R1-2 - Meta-Learning and LLMs.** Fu et al. [2] showed that their VulExplainer approach outperformed the commonly used data imbalance techniques such as focal loss [145] and logit adjustment [148]. However, the performance on infrequent vulnerability types still has plenty of room to improve. Thus, future research should explore other techniques to address the data imbalance issue. For instance, meta-learning involves training models to learn a higher-level strategy or set of parameters that enable them to quickly adapt to new, unseen tasks with limited data. It might be suitable for imbalanced data because the exposure to diverse tasks during meta-training helps the model generalize effectively, allowing it to adapt to tasks with imbalanced class distributions. The transfer of knowledge across tasks and the few-shot learning nature of meta-learning contribute to improved performance in scenarios where certain classes have limited samples. Such an approach could also be integrated with few-shot learning of large language models (LLMs).

**C2-1 - Model Explainability in Automated Vulnerability Repair (In DevOps Development).** The advancement of language models has dramatically im-

---

proved the accuracy of programs and vulnerability repair due to the substantial model size and training data. While recent studies focus on performance improvement, the predictions offered by those models are not explainable, posing challenges in establishing trust between the models and users. As highlighted by Winter et al. [437], trust in program repair is a crucial problem, in their empirical study involving Bloomberg developers, the sentiment conveyed was that an automated repair tool should demonstrate its reliability and foster trust with developers.

**R2-1 - Evidence-based XAI.** Given the complex structure of Large Language Models (LLMs), characterized by multiple hidden layers and a large number of parameters, developing intrinsically explainable AI to explain their repair predictions poses a significant challenge. Nonetheless, an avenue worth exploring involves leveraging the model's self-attention mechanism to ascertain if it can offer meaningful explanations. In addition, future studies could delve into evidence-based explainable AI (XAI), wherein the repair model not only presents its generated fix to end users but also showcases a similar repair case from its training data. This approach aims to establish trust with users, drawing inspiration from its successful application in explaining the story point estimation model in agile software development [103].

**C2-2 - Model Explainability in Software Defect Prediction (In DevOps Code Commit).** Our investigation revealed that the majority of just-in-time (JIT) software defect prediction (SDP) methods based on deep learning (DL) primarily emphasize enhancing performance and granularity [74, 94, 352, 353]. While more accurate and finer-grained SDP methods are beneficial for constructing robust and cost-effective DL-based SDP solutions, there is a noticeable lack of attention to the explainability of these DL-based SDPs. This lack of focus on explainability presents a challenge to the trustworthiness of DL-based SDPs.

**R2-2 - XAI for DL Models.** Given the complexity of DL models, explaining them is more challenging compared to machine learning (ML) models. Nonetheless, future studies could explore the use of extrinsic explanations such as Layer Integrated Gradient (LIG) [76], DeepLift [78, 79], DeepLiftSHAP [80], and GradientSHAP [80], which rely on gradients or their approximations to assess feature importance. Additionally, intrinsic explanations, such as the self-attention outputs from transformer architectures, could also highlight significant features contributing to model predictions. Finally, exploring case-based reasoning [438], which uses similar predictions retrieved from the training data as supporting evidence

---

to explain model predictions, is another promising avenue to consider.

**C2-3 - Model Explainability in Log Analysis and Anomaly Detection (In DevOps Operation and Monitoring).** Our investigation reveals that while the majority of AI-driven anomaly detection approaches focused on performance improvement, only a few studies [406, 407] prioritized model explainability. Explainable AI (XAI) is crucial for anomaly detection models due to several reasons according to Han et al. [406]. Firstly, without detailed explanations for system decisions, security operators struggle to establish trust, leading to unreliable outputs. Secondly, the black-box nature of deep learning models makes it difficult to diagnose and address system mistakes, hindering effective troubleshooting. Additionally, integrating human expertise into security systems is challenging with opaque models, limiting human-in-the-loop capabilities and feedback incorporation. Thus, the lack of focus on explainability presents a significant challenge for AI-driven anomaly detection systems.

**R2-3 - XAI for ML Models.** To enhance the explainability of machine learning (ML) models in anomaly detection, future research can explore various interpretability methods. Approaches such as SHAP [80], LIME [82], and Breakdown [439] have demonstrated success in tasks like software defect prediction for understanding model decisions [440]. Additionally, recent advancements in explainable AI, such as the AIM framework proposed by Vo et al. [441], show potential for outperforming traditional XAI approaches like LIME and L2X, particularly in tasks like sentiment analysis. On the other hand, causal inference holds the potential to make machine learning models more interpretable [442]. By determining cause-and-effect relationships between variables, causal inference techniques offer insights into AI-driven anomaly detection. Overall, integrating causal inference methods into anomaly detection frameworks holds promise for enhancing model interpretability. By uncovering causal relationships between input features and detected anomalies, these techniques offer deeper insights into system behavior, enabling more informed decision-making by software developers and security analysts.

**C3-1 - Lack of AI Security Tooling in IDEs (In DevOps Development). Challenge - Lack of AI-driven Tools in IDEs** A significant hurdle in the current landscape is the limited availability of AI-driven security tools in developers' IDEs, posing a challenge to the widespread adoption of advanced security measures in software development. The scarcity of such tools restricts developers from harnessing the full potential of artificial intelligence in enhancing security practices.

---

Additionally, there is a notable absence of a comprehensive user study that thoroughly evaluates the performance and effectiveness of existing AI-driven security tools, including prominent ones like AIBugHunter [5] and Snyk [264]. The lack of a comprehensive user study hinders a nuanced understanding of the strengths, weaknesses, and overall impact of these tools, impeding efforts to establish robust security practices in the evolving landscape of software development.

**R3-1 - AI Tool Deployment and Comprehensive Tool Evaluation.** We outline potential avenues for research in the domain of AI-driven security tools. Considering the extensive literature and varied techniques available for vulnerability detection, classification, and repair, future studies could focus on seamlessly integrating existing methods into developers' IDEs to enhance their practical applicability. For instance, commercial tools such as Code Scanning Autofix powered by GitHub Copilot are available in GitHub to help developers automatically fix vulnerable code in their pull requests. This tool is an expansion of Code Scanning that provides users with targeted recommendations to help them fix code scanning alerts in pull requests and avoid introducing new security vulnerabilities. The potential fixes are generated automatically by large language models (LLMs) using data from the codebase, the pull request, and from CodeQL analysis [443]. In particular, when a vulnerability is discovered in supported languages (i.e., JavaScript, TypeScript, Python, and Java), Autofix will generate a natural language explanation of the suggested fix, along with a preview of the code suggestion that the developer can accept, edit, or dismiss. Moreover, these code suggestions can include changes across multiple files and the dependencies that should be added to the project [444]. Furthermore, a comprehensive evaluation of these AI-driven security tools is imperative. This involves soliciting and analyzing user feedback, particularly the scenarios where the tools generate inaccurate predictions. Furthermore, it is crucial to investigate the robustness of AI-driven security tools. For example, a robust security tool should ideally predict the repaired program as benign. However, the existing literature does not definitively address whether a program continues to be predicted as vulnerable even after undergoing a successful repair. Exploring this aspect would contribute valuable insights into the efficacy of AI-driven security tools in practical scenarios.

**C3-2 - Lack of AI Security Tooling in CI/CD (In DevOps Code Commit).** Our investigation revealed that the majority of just-in-time (JIT) SDP tools used in CI/CD environments, such as JITO [359] and JITBot [100], primarily rely on machine learning models. Despite proposed deep learning (DL)-based SDP

---

methodologies, there is an absence of DL-based tools integrated into CI/CD pipelines. This absence impedes the practical adoption of DL-based approaches.

**R3-2 - AI Tool Deployment in CI/CD Pipelines.** In contrast to machine learning (ML) models, which depend on manually predefined metrics for predicting software defects, DL models can learn source code representations directly from developers' code without the need for extensive feature engineering efforts. To enhance their practical adoption, future research should focus on integrating existing DL-based approaches for JIT SDP into CI/CD pipelines. Additionally, conducting a comprehensive evaluation of these DL-based tools is essential. This evaluation should encompass not only performance assessment post-deployment but also a large-scale user study to gather feedback from developers.

**C3-3 - Lack of AI Security Tooling for Infrastructure Scanning (In DevOps Build, Test, and Deployment).** Our investigation has uncovered a gap between AI-driven infrastructure scanning approaches and their practical adoption. Although several machine learning (ML)-based methods have been proposed to detect defective Infrastructure-as-Code (IaC) scripts [381–384], they have yet to be deployed as software tools for developers. The lack of deployment hinders their practical adoption. Furthermore, the precision and practicality of these tools post-deployment remain unexplored.

**R3-3 - AI Tool Deployment and Post-Deployment Evaluation.** Future research should focus on bridging the gap between proposed AI-driven methods for detecting defective Infrastructure-as-Code (IaC) scripts and their practical deployment as software tools for developers. Exploring strategies to facilitate the deployment of these tools, such as developing user-friendly interfaces and integration into existing development workflows, could enhance their practical adoption. Additionally, investigating the precision and practicality of these tools post-deployment is crucial to assess their effectiveness in real-world scenarios. By addressing these research opportunities, advancements can be made towards empowering developers with effective and efficient tools for enhancing the security of Infrastructure-as-Code.

Below, we begin introducing the challenges and research opportunities specific to each security task in the DevOps process. Since we found no relevant literature discussing AI-driven approaches in the planning step of DevOps, we will start with the development step.

---

## 10.5.2 Plan

### 10.5.2.1 Threat Modeling & Impact Analysis

Since our literature review did not reveal relevant studies on Threat Modeling and Impact Analysis, we were unable to identify challenges directly from the literature. Instead, we examine existing AI-driven approaches used in industry and explore how future research could build upon these practices to advance the field.

**Existing AI Applications in Industry.** The recent advancement of AI models presents promising opportunities for facilitating threat modeling and impact analysis in cybersecurity. For instance, an AI-driven commercial tool named Aribot for threat modeling has been introduced by Aristiun [445]. Aribot automates the threat modeling process through various functionalities. It automatically generates Infrastructure-as-Code templates, addressing public cloud-specific threats effectively. It ensures traceable security requirements throughout the lifecycle, facilitating comprehensive security coverage. Aribot also simplifies compliance adherence by mapping security requirements to frameworks such as the National Institute of Standards and Technology (NIST). Additionally, it enhances transparency and accountability by reporting and tracking records remediation efforts by development teams, providing real-time updates on implementation status without requiring manual intervention. Another AI-driven commercial tool for impact analysis has been introduced by Validata [446]. This AI-based solution automatically delivers crucial information for applications, expediting upgrades and patches while predicting the impact of new product versions before release. It empowers users to effortlessly analyze change impact on quality, performance, resource capacity, and costs within hours, automatically applying recommended changes through a corrective action plan.

Moreover, Microsoft [447] has recently introduced Copilot for Security. Informed by large-scale data and threat intelligence, including more than 78 trillion security signals processed by Microsoft each day. Copilot is coupled with large language models (LLMs) to deliver tailored security insights. It offers an interactive interface to support security practitioners during impact analysis and threat modeling with relevant security knowledge. Their randomized controlled trial indicated that experienced security analysts were 22% faster with Copilot, they were 7% more accurate across all tasks when using Copilot, and 97% said they want to use Copilot the next time they do the same task [448]. Copilot serves as the first critical step in leveraging generative AI to support security practitioners in their

---

workflow.

In summary, the adoption of AI in threat modeling and impact analysis presents significant advantages for the planning step in DevSecOps. Tools like Aribot, Validata, and Copilot for Security showcase the potential of AI to automate these processes effectively. By leveraging the capabilities of AI to assess risks, identify vulnerabilities, and prioritize security measures, organizations could make informed decisions and allocate resources more efficiently.

**Future Research Directions.** To build on existing AI-driven tools like Aribot, Validata, and Copilot for Security, research can focus on a few key areas critical to enhancing DevSecOps planning. First, there is a need to refine AI models to enable dynamic updates in threat modeling and impact analysis as part of continuous integration/continuous deployment (CI/CD) pipelines. Given the iterative nature of DevSecOps, these updates would ensure that security assessments remain relevant and responsive to ongoing code changes. Additionally, the creation of benchmark datasets is essential for measuring the effectiveness of these AI-driven approaches. Such benchmarks would help identify gaps in current tools, guiding future improvements and providing a standardized basis for evaluating new solutions. Finally, research into the ethical implications of deploying AI in security is crucial. This includes ensuring that AI models are used responsibly, with attention to privacy, fairness, and transparency, which will be vital for the wider adoption of AI-driven security measures in DevSecOps environments.

### 10.5.3 Development

#### 10.5.3.1 Software Vulnerability Detection

While the progress in code pre-trained language models enhances the F1-score of function-level vulnerability detection, reaching up to 96.5% [123], there remain several challenges that must be addressed in the current landscape of AI-driven vulnerability detection. We describe challenges and potential research directions in the following.

**C4 - Cross Project.** Most of the VD studies considered the mixed project (models are trained and tested on combined projects) scenario when evaluating their proposed approaches. However, in an empirical study evaluating SOTA DL-based VD approaches, Steenhoek et al. [449] observed a decline in model performance during detection under the cross-project scenario (models are trained on

---

one set of projects and tested on completely different, non-overlapping projects). Specifically, the F1-score of function-level detection models experienced a reduction ranging from 11% to 32%. This underscores the challenge posed by cross-project vulnerability detection and emphasizes the limited generalizability of current methodologies. The findings emphasize the need for advancements in methodologies capable of generalizing effectively across various projects, especially in cross-project scenarios.

**R4 - Cross-Project SVD with Diverse CWE-IDs.** Zhang et al. [301] and Liu et al. [450] both proposed to use deep domain adaptation to address cross-project VD in C languages. Subsequent research avenues could delve into cross-programming language VD and explore cross-project VD for other programming languages. In particular, Liu et al. [450] mainly focused on CWE-119 and CWE-399 while other dangerous CWE-IDs [218] should be considered in future studies. Furthermore, the study concentrated on graph attention networks, yet there is potential for exploration of other graph neural networks (GNNs) and large language models (LLMs) for effective cross-project VD.

**C5 - MBU Vulnerabilities.** Sejfia et al. [451] pointed out that state-of-the-art deep learning-based vulnerability detection (VD) approaches concentrate on individual base units, assuming that vulnerabilities are confined to a single function. However, vulnerabilities may extend across multiple base units (MBU). They found that existing DL-based detectors do not work as well in detecting all comprising parts of MBU, which poses a challenge in predicting MBU vulnerabilities. Thus, future research should contemplate adapting their evaluation methods to incorporate the presence of MBU vulnerabilities.

**R5 - Evaluate SVD on MBU Vulnerabilities.** The majority of studies on function-level VD [1, 14, 325] have typically assessed performance based on the number of correctly detected vulnerable functions. However, it is important to note that a single vulnerability might encompass multiple vulnerable functions. Therefore, merely identifying one vulnerable function does not necessarily equate to the comprehensive detection of the entire vulnerability. Acknowledging this, Sejfia et al. [451] emphasized the need for future research to account for the scenario of Multiple Base Unit (MBU) vulnerabilities during the training and testing phases of DL-based VD. This requires refining evaluation metrics and methodologies to align with MBU vulnerabilities. Consequently, there is a critical call for an enhanced understanding and consideration of MBU vulnerabilities to further advance the field of vulnerability detection.

---

**C6 - Data Quality.** Finally, Croft, Babar, and Kholoosi [139] noted data quality concerns related to accuracy, uniqueness, and consistency in widely used vulnerability datasets. Their findings revealed that a substantial portion, ranging from 20% to 71%, of labels in real-world datasets were inaccurately assigned. This inaccuracy had the potential to significantly impact the performance of resulting models by up to 65%.

**R6 - Addressing Data Inaccuracy from Automatic Data Collection.** Croft, Babar, and Kholoosi [139] pointed out that automatic data collection often leads to data inaccuracy. Common vulnerability datasets [70, 113] were constructed by using the code changes information to recover the vulnerable version of a method. However, the vulnerability fixing commits or vulnerable lines could be incorrectly selected. Thus, it is important for future research to devise semantic filters or heuristics—methods or rules designed to analyze and interpret the meaning of code changes. This is crucial for precisely pinpointing lines that signify vulnerability fixes, and addressing the data inaccuracies stemming from the automatic data collection process.

#### 10.5.3.2 Software Vulnerability Classification

In our examination of AI-driven methods explaining vulnerability types to developers, we found that a significant amount of studies focus on vulnerability detection, with limited attention given to vulnerability classification. Below, we outline the challenge identified in current AI-driven vulnerability classification approaches followed by future research opportunities.

**C7 - Incompleted CWE Tree.** The TreeVul approach [311] currently relies exclusively on parent-child relations within the Common Weakness Enumeration (CWE) hierarchy for conducting top-down searches. This approach excludes the consideration of other potentially valuable relations, such as PeerOf, which could enhance the model's effectiveness. In addition, TreeVul only considers CWE categories with depth $\leq 3$  while some important categories may be located at depth $>3$ . These incomplete considerations of CWE tree structure may hinder TreeVul's ability to generalize to additional CWE-IDs.

**R7 - Advanced Tree-based SVC.** According to Pan et al. [311], TreeVul did not encompass the entire CWE tree. Consequently, future investigations may broaden the TreeVul approach by incorporating extra relations, such as PeerOf. Nevertheless, this procedure views the CWE structure as a graph, introducing a

---

higher level of complexity compared to the hierarchical tree structure considered by Pan et al. [311]. Thus, emphasis could be placed on streamlining the transformation process to reduce the complexities of converting the CWE tree into a graph. Furthermore, optimizing the TreeVul approach to dynamically determine the appropriate level for concluding top-down searches beyond the predefined depth-3 CWE categories offers an opportunity to enhance the model’s adaptability. This optimization could involve developing a mechanism to assess confidence levels, allowing the model to automatically adjust its search depth based on contextual factors for each specific input. Such advancements could improve the precision and flexibility of AI-driven vulnerability classification.

#### 10.5.3.3 Automated Vulnerability Repair

Recent advancements in transformer architecture and pre-trained language models for code have shown improvements over RNN-based models in vulnerability repair and program repair tasks. Nevertheless, beyond model architecture and performance, there are additional aspects that demand attention and further improvement, especially when considering their integration into real-world projects. Below, we present challenges derived from previous studies with potential research opportunities.

**C8 - Sequence Length and Computing Resource.** Most code pre-trained language models (LMs) have an input length limit of 512 subword tokens for base-size models and 1024 for large-size models. Thus, LMs may not fully comprehend long code programs due to the input length limit of LMs, which constrains the repair capability. Furthermore, the increase in output length not only impacts the performance of repair capability but also introduces a well-known challenge associated with long sequences in the NMT model. This difficulty arises from the limitations imposed by Markov chain assumptions and probabilistic constraints, where the model encounters challenges in maintaining coherence and capturing intricate dependencies over extended sequences. In particular, Fu et al. [3] observed that the repair model demonstrated a repair accuracy of 77% when both input and output lengths were below 100 and 10, respectively. However, the model’s accuracy drastically dropped to 7% when the input and output lengths exceeded 500 and 50, respectively. In addition, Huang et al. [452] emphasized that the substantial size of language models can place a burden on computing resources, which hinders the generation of candidate patches. For example, during patch synthesis on the Defects4J dataset, the maximum beam size is set at 200, limiting the generation to only 200 patches for each bug.

---

**R8 - Transformer Variants That Process Longer Sequence.** Transformer architectures have led to remarkable progress in automated repair applications. However, despite their successes, modern transformers rely on the self-attention mechanism, whose time- and space-complexity is quadratic in the length of the input that requires substantial computing resources when encountering long sequences. Recently, many alternative architectures have been proposed to mitigate the long sequence challenge and computing burden of transformers. For example, Beltagy, Peters, and Cohan [453] presented Longformer which uses global and local attention windows to mitigate the memory and time bottleneck of the self-attention mechanism from  $O(ns \times ns)$  to  $O(ns \times w)$ , with  $ns$  being the sequence length and  $w$  being the average window size. In particular, the base-size Longformer can process up to 4,096 subword tokens. On the other hand, Sun et al. [454] proposed Retentive Network (RetNet) as a new foundation architecture for large language models. The chunkwise recurrent representation of RetNet facilitates efficient long-sequence modeling with linear complexity. Thus, future studies may explore their efficacy in the context of program and vulnerability repair.

**C9 - Loss of Pre-Trained Knowledge.** Most language model-based repair approaches follow a paradigm of taking the pre-trained checkpoint and further fine-tuning the model to fit the downstream program and vulnerability repair tasks. Nevertheless, according to Huang et al. [452], after fine-tuning, pre-trained models may experience a reduction in the knowledge gained during pre-training when compared to zero-shot learning (i.e., without fine-tuning). This could be attributed to conflicting training objectives between the unsupervised masked language modeling (MLM) of pre-training and the supervised neural machine translation (NMT) of fine-tuning.

**R9 - Explore Different Training Paradigms.** Huang et al. [452] suggested two research directions to address this challenge. Future research could investigate strategies to alleviate catastrophic forgetting [455] for program and vulnerability repair tasks. Additionally, exploring both neural machine translation (NMT) and masked language modeling (MLM) training paradigms during the fine-tuning stage is a potential direction. Notably, AlphaRepair [456] adopts a cloze task (MLM) approach instead of a translation task (NMT), predicting the token at the mask location based on contextual tokens. While employing MLM during fine-tuning could be advantageous as it aligns with the training objective of the model's pre-training stage, the differentiation in repair efficacy between the two paradigms

---

(NMT and MLM) remains unclear.

**C10 - Automated Repair on Real-World Scenarios.** Pearce et al. [56] found that large language models (LLMs) can perfectly repair all of their synthetic and hand-crafted vulnerability scenarios. However, LLMs were not sufficiently reliable when producing automatic fixes for the real-world data in their qualitative analysis. Furthermore, they underscored the limitation of the current repair approach, which is confined to addressing issues within a single location in a single file.

**R10 - Addressing Limitations of LLMs.** Addressing the limitations of large language models (LLMs) in real-world vulnerability scenarios presents promising avenues for future research. Pearce et al. [56] identified the significant performance gap of LLMs in repairing synthetic and real-world vulnerability scenarios. Thus, understanding and mitigating the factors contributing to the discrepancy in performance between synthetic and real-world scenarios would be a valuable direction for further investigation, enabling the development of more robust automatic fixes in practical cybersecurity contexts. Furthermore, exploring multi-location and multi-file repair strategies and techniques could offer valuable insights.

#### 10.5.4 Code Commit

##### 10.5.4.1 Dependency Management

Since our literature review did not reveal relevant studies on Dependency Management, we were unable to identify challenges directly from the literature. Instead, we examine existing AI-driven approaches used in industry and explore how future research could build upon these practices to advance the field.

**Existing AI Applications in Industry.** Recently, a startup company Infield introduced an AI-driven commercial tool designed to automate software dependency management and strengthen DevOps security [457]. The tool continuously monitors recommended updates of open-source components, provides step-by-step guidance for achieving the ideal status, and gathers unstructured information about open-source dependencies and their upgrades. This data is then structured to help users manage their backlog of upgrades efficiently, allowing them to prioritize upgrades based on risk and effort. For instance, the tool can be connected with users' codebase in GitHub, scans their code to determine the underlying dependencies, and recommends the steps needed to upgrade safely for their codebase. The tool offers a human-assisted approach to dependency

---

management, helping organizations overcome the challenges of maintaining dependencies in a rapidly evolving ecosystem.

**Future Research Directions.** The introduction of AI-driven tools like Infield's dependency management solution marks a significant advancement in automating and enhancing security in DevSecOps. However, there remains substantial potential for further innovation, particularly in integrating Retrieval-Augmented Generation (RAG) techniques with Large Language Models (LLMs). One promising direction is to augment LLMs with a pre-built knowledge base containing comprehensive information on software dependencies, vulnerabilities, and best practices. In this setup, when a software update is identified, developers or security practitioners could use the RAG-based LLM system to query the knowledge base and receive tailored security recommendations for the specific dependencies involved. This approach would empower users to make more informed decisions, ensuring more effective and secure management of software dependencies. Additionally, developing a benchmark dataset specifically for evaluating AI-driven dependency management approaches is crucial. This dataset would allow researchers to systematically assess the effectiveness of RAG-based LLMs in real-world scenarios, identifying gaps and opportunities for improvement. By focusing on these areas, future research can build on existing AI applications and drive advancements in dependency management within the DevSecOps lifecycle.

#### 10.5.4.2 CI/CD Secure Pipelines

AI-driven just-in-time (JIT) software defect prediction (SDP) approaches have been proposed and deployed into CI/CD pipelines for adoption [100, 359]. However, our investigation uncovers several challenges that must be addressed to further enhance these AI-driven approaches. Below, we present the challenge derived from previous JIT SDP studies with potential research opportunities.

**C11 - The Use of RNNs.** We discovered that many DL-based JIT SDP methods primarily rely on RNNs [94, 357]. However, pre-trained transformer architectures and language models (LMs) demonstrate promise for SDP, building on their successful track record in vulnerability detection [449], a closely related domain. RNNs excel at learning source code representations and predicting defects without relying on predefined metrics. Nonetheless, RNNs process input sequentially and struggle with long sequences, which can lead to suboptimal results compared to transformer-based models.

---

**R11 - Explore LMs and LLMs.** In light of the successful application of language models (LMs) for software vulnerability detection [1, 449], future research could explore their potential for JIT SDP. LMs like CodeBERT [40] and CodeT5 [131, 132] are transformer architectures pre-trained on source code, featuring self-attention mechanisms that excel in capturing semantic nuances and handling longer sequences compared to RNNs. Moreover, these LMs are pre-trained on extensive source code datasets encompassing various programming languages, enabling them to generate better source code representations compared to RNNs and enhancing SDP effectiveness. Furthermore, investigating the efficacy of large language models (LLMs) such as GPT-4 [458] and Code Llama [459] for SDP is a valuable research direction. These models are pre-trained extensively to understand source code. Thus, future research could explore strategies for leveraging these LLMs and deploying them to build secure CI/CD pipelines.

## 10.5.5 Build, Test, and Deployment

### 10.5.5.1 Configuration Validation

AI-driven methodologies have been suggested for automated verification of optimal and secure system configurations. Nevertheless, the complexity of software systems, characterized by an extensive array of configuration options, poses a challenge for machine learning (ML) and deep learning (DL) models. Below, we present the challenge derived from previous studies with potential research opportunities.

**C12 - Complex Feature Space.** Machine learning (ML) models for extracting insights from numeric and categorical features is a widely adopted strategy in configuration validation processes. Contemporary AI-driven performance prediction models frequently rely on conventional machine learning models like XGBoost [376] or basic feed-forward neural networks such as Deep-Perf [378] to analyze tabular configuration data and predict system performance. Nonetheless, software systems often encompass an extensive array of configuration options. For instance, one of the datasets used to evaluate Deep-Perf consists of 13,485 valid configurations [378]. The extensive configuration size results in intricate relationships among features and an expansive feature space. Consequently, traditional ML models and simple feed-forward neural networks may struggle to generalize to unseen configurations or capture subtle patterns and dependencies within the data, especially in high-dimensional spaces such as configuration datasets.

**R12 - Transformers for Tabular Data.** Addressing the challenge of capturing

---

intricate relationships and dependencies within high-dimensional configuration datasets presents numerous future research opportunities in performance prediction. One promising avenue for exploration involves leveraging more advanced model architectures, such as transformer-based models [69], to enhance the predictive capabilities of performance prediction models. Transformer architectures, renowned for their success in natural language processing tasks, offer the potential to effectively capture complex patterns and dependencies within tabular data like configuration datasets. By adapting transformer models to handle tabular data, researchers can explore their ability to learn from the sequential and relational information present in configuration parameters and accurately predict system performance. In addition, researchers can explore semi-supervised learning techniques [460]. These methods leverage both labeled and unlabeled data, helping overcome the scarcity of labeled training instances in performance prediction. In summary, these research directions could improve the effectiveness of AI-driven configuration validation processes in complex software systems.

#### 10.5.5.2 Infrastructure Scanning

AI-driven infrastructure scanning approaches have been proposed for detecting insecure Infrastructure-as-Code (IaC) scripts during the deployment phase of DevOps. Nevertheless, there remains room for improvement and identified gaps. Below, we outline the challenge derived from previous studies with potential research opportunities.

**C13 - Manual Feature Engineering.** We found that some AI-driven approaches still rely on manual feature engineering with machine learning (ML) models to predict insecure IaC scripts [381–383], which can demand substantial effort, especially as the project scales up. It is necessary to prepare a predetermined set of features before model training; for instance, Dalla Palma et al. [383] prepared a set of 108 features. Subsequently, the feature selection process is essential to filter out irrelevant features, and techniques such as normalization will be adopted to optimize ML model performance. This time-consuming process could hinder the practical adoption of AI-driven infrastructure scanning approaches.

**R13 - Explore DL-based Techniques.** Considering the recent advancement of deep learning (DL) and language models (LMs), it is feasible to use DL models for detecting defective IaC scripts. Unlike traditional machine learning (ML) models, DL models allow direct input of IaC scripts, thereby eliminating the need for manual feature engineering. DL models are capable of learning both semantic and

---

syntactic features of IaC scripts. They can identify insecure patterns within IaC scripts based on historical data and predict defective scripts in future instances. Additionally, LMs have proven successful in various software security tasks, such as vulnerability detection [1, 449] and repairs [3, 210, 317]. Given that LMs are pre-trained on vast amounts of source code data, it is viable to fine-tune them for accurately detecting insecure IaC scripts. Hence, future research could explore the application of DL models to address the time-consuming challenges associated with manual feature engineering in detecting insecure IaC scripts.

### 10.5.6 Operation and Monitoring

#### 10.5.6.1 Log Analysis and Anomaly Detection

Several AI-driven anomaly detection approaches have been developed to detect anomalies in software systems. However, our investigation reveals the challenge that requires attention and resolution. Below, we introduce the challenge derived from previous studies with potential research opportunities.

**C14 - Normality Drift for Zero-Positive Anomaly Detection.** In anomaly detection, zero-positive classification is commonly used because anomalies are typically rare and undefined. Zero-positive classification trains models on normal behavior, allowing them to generalize to unseen anomalies and adapt to changing data distributions. However, Han et al. [461] recognized a normality drift problem for the zero-positive classification used in anomaly detection. Normality drift refers to the phenomenon where the underlying distribution of normal (non-anomalous) data changes over time in a dataset used for training AI-driven anomaly detections. In other words, the characteristics of normal behavior exhibited by the system or environment being monitored may evolve or shift gradually or abruptly, leading to discrepancies between the training data and the data encountered during deployment or inference. This drift can adversely affect the performance of anomaly detection systems, as models trained on historical data may become less effective at identifying anomalies from the new normal behavior. Thus, addressing normality drift is crucial for maintaining the effectiveness and reliability of anomaly detection systems.

**R14 - Enhance Normality Drift Detection.** Han et al. [461] introduced OWAD as a solution to combat the challenge of normality drift encountered in deep learning-based anomaly detection for security applications. OWAD serves as a framework designed to detect, explain, and adapt to normality shifts at the distribution level, departing from sample-level explanations like CADE [462] which

---

fail to comprehensively address the holistic shift in distribution, thereby limiting their applicability in understanding the overall normality drift. This development paves the way for numerous future research opportunities in anomaly detection. Particularly, there exists potential for exploring innovative adaptation strategies within OWAD aimed at enhancing its efficacy in adapting to evolving data distributions. This could entail investigating reinforcement learning approaches or meta-learning techniques to dynamically adjust model parameters in response to shifting distributions. Future investigations can also concentrate on refining OWAD's adaptation mechanisms to ensure effectiveness across diverse security environments.

#### 10.5.6.2 Cyber-Physical Systems

AI-driven approaches have been proposed to address security concerns in cyber-physical systems (CPS) [409–412, 415, 434, 435]. However, our investigation reveals that current approaches to CPS security primarily target individual cyber-attacks, neglecting the complex scenario of encountering multiple simultaneous threats across diverse CPS layers. Below, we describe this challenge and highlight potential avenues for future research.

**C15 - Monitoring Multiple Cyber-Attacks Simultaneously.** CPS consists of multiple layers representing distinct components and functionalities crucial for system operation. These layers typically include the physical layer encompassing hardware infrastructure, the communication layer facilitating data exchange between components, the control layer governing system behavior, and the data processing layer analyzing collected data. However, current approaches in CPS security often focus on addressing individual cyber-attacks, overlooking the reality of facing multiple simultaneous threats across various CPS layers [409–412, 415, 434, 435]. For instance, attackers may simultaneously disrupt communication networks, manipulate control algorithms, and tamper with data analytics results in the smart grid CPS, posing complex challenges for system security and resilience. As cyber threats grow in sophistication and diversity, the need to coordinate responses and monitor ongoing attacks concurrently becomes paramount. However, achieving this in real time presents substantial challenges due to the intricate and interconnected nature of CPS environments.

**R15 - Distributed Anomaly Detection and Multi-Agent Systems.** Future research could focus on developing distributed detection and response mechanisms within CPS environments, empowering individual components to detect

---

and respond to cyber threats in real time. This approach decentralizes the security architecture and reduces reliance on centralized control systems. By integrating AI-based anomaly detections, these mechanisms could effectively identify unusual patterns or behaviors indicative of cyber attacks across various CPS layers. Multi-agent systems (MAS) hold promise for addressing CPS attacks across different layers due to their decentralized and collaborative nature. MAS consists of multiple autonomous agents, each capable of independent decision-making and action. For example, Lyu and Brennan [463] introduced a two-layer architecture modeling framework for CPS, demonstrating how it enables real-time adaptation in dynamic industrial automation environments. Moreover, integrating AI techniques with MAS could facilitate collaborative decision-making and resource allocation among autonomous agents distributed across different CPS components or layers. By leveraging reinforcement learning (RL), agents can dynamically adapt their strategies based on feedback from the environment, allowing them to respond to emerging cyber threats with agility and precision. For instance, Ibrahim and Elhafiz [464] presented a case study of a smart grid and investigated CPS security using RL. In summary, AI-driven MAS could adaptively adjust defense strategies based on evolving threat landscapes and system conditions, enhancing the effectiveness of cyber defense operations within CPS environments.

## 10.6 Threats to Validity

Our systematic literature review (SLR) was conducted in alignment with the guidelines outlined by Keele et al. [279] and Kitchenham, Madeyski, and Budgen [280]. However, like any SLR, our review also has certain limitations. Below, we provide a discussion of the external, internal, and construct threats to the validity of our SLR, along with corresponding mitigation strategies.

*Threats to construct validity* relate to the process of selecting the 12 security tasks for our review as presented in Figure 56. This process involves subjective judgment that could potentially influence the construct validity of our findings. The identification and refinement of tasks were based on the collective expertise of the three authors, which, while informed, inherently involved subjective elements. This subjectivity could affect how well the tasks represent the actual security concerns within the DevOps workflow. To mitigate this threat, we employed a rigorous approach involving multiple authors rather than relying on a single individual's perspective. The first author initially provided an overview of potential tasks, which was then critically discussed and refined through iterative

---

brainstorming sessions and online meetings with the other two authors. This collaborative process ensured a broad range of insights and perspectives were considered, thereby enhancing the accuracy and relevance of the identified tasks.

*Threats to external validity* relate to the search string, the filtering process, and the selection of venues in this SLR aimed at identifying literature related to AI-driven methodologies and tools for DevSecOps. It is possible that our search string missed studies that should have been included in our review, potentially due to a missed term or a combination of terms that may have returned more significant results. Given that our study focuses on two areas—AI (specifically, machine learning and deep learning) and security methodologies and tools for integration into DevOps—we employ a systematic approach to testing different combinations of AI-related terms and security tasks as presented in Section 10.3.2. This involves experimenting with variations in search strings, including synonyms, related terms, and alternative phrasings. By doing so, we aim to increase the likelihood of identifying relevant studies that may have been overlooked initially. Furthermore, we leverage our understanding of the domain to refine and optimize our search strategy iteratively to mitigate this threat.

While conducting this systematic literature review (SLR), we acknowledge the potential for selection bias in the studies included. To mitigate this threat, we employ rigorous inclusion and exclusion criteria predefined before initiating the filtering process as illustrated in Table 29. Furthermore, we implement a snowballing search strategy to supplement our initial searches as presented in Section 10.3.4, thereby capturing papers that may have been overlooked initially.

The selection of venues for this SLR plays a crucial role in ensuring the reliability of our findings. In our approach, we deliberately include top-tier SE and security conferences and journals, focusing on venues with CORE A or CORE A\* rankings. By prioritizing these esteemed venues, known for their rigorous peer-review processes and high academic standards, we aim to uphold the quality and credibility of the literature surveyed. While we acknowledge the possibility that our exclusion of lower-ranking venues may have resulted in the omission of relevant studies, our deliberate focus on top SE and security venues allows us to capture emerging trends and insights from reputable sources. To provide transparency and accountability in our venue selection process, we disclose the selected venues in Table 28, enabling readers to assess our review methodology.

*Threats to internal validity* relate to the potential absence of literature discussing

---

AI-driven methods or tools for specific security tasks within the DevSecOps process. Notably, despite our comprehensive search efforts, we encountered a scarcity of studies addressing AI applications for threat modeling and impact analysis in the plan step, as well as dependency management in the code commit step. This gap in the literature poses a potential limitation to the comprehensiveness of our review. To mitigate this, we have examined common approaches for these three security tasks within the context of RQ1 and explored AI-driven applications in industry along with future research directions in RQ2. This integrated approach provides a comprehensive overview of current practices and underscores their relevance to DevSecOps.

Another threat to internal validity in our study arises from the fact that the themes and challenges were derived directly from the reviewed literature. This methodology means that some universally important challenges, such as data quality and model explainability, may only be associated with specific security tasks in DevSecOps if they were explicitly discussed in the literature we analyzed. Consequently, while our study highlights key challenges within the context of specific security tasks, there may be broader applicability of these challenges across other tasks that are not discussed in this work.

## 10.7 Summary

In this chapter, we present a systematic literature review (SLR) focusing on AI-driven security approaches tailored for DevSecOps. We collect papers from high-impact software engineering and security venues, analyzing 99 publications to gain insights into the field. Our SLR identifies 12 security tasks critical to DevOps and examines various AI-driven security approaches, tools, and 65 benchmarks used to evaluate these approaches. Additionally, we uncover 15 significant challenges confronting state-of-the-art AI-driven security methodologies and derive promising avenues for future research. This SLR lays the groundwork for motivating the challenges and future research directions beyond this thesis. While this thesis primarily focuses on advancing deep learning-based vulnerability detection, classification, and repair within the software development phase of DevSecOps, this SLR paves the way for expanding my research to explore AI-driven security approaches across other critical phases of DevSecOps in the future.

## 11 Conclusion

---

Software vulnerabilities are widespread and can lead to significant revenue loss for businesses due to breaches and system failures. Ensuring the safety and security of software systems, however, presents a formidable challenge. Security analysts must engage in labor-intensive processes to (1) detect and locate vulnerabilities, (2) identify the specific types of these vulnerabilities, and (3) propose and implement repair patches to mitigate them. This workflow demands considerable manual effort, making it difficult for under-resourced security teams to keep up with emerging threats. The primary goal of this research is to alleviate this burden by developing deep learning (DL)-based automated approaches. These approaches are designed to assist security analysts by accurately detecting and localizing vulnerabilities within source code, explaining the types of detected vulnerabilities, and suggesting corresponding repairs. To achieve this goal, we raise four research questions in Chapter 1. We discuss the challenges and problems of the existing DL-based software vulnerability detection (SVD), software vulnerability classification (SVC), and automated vulnerability repair (AVR) approaches in chapter 2. Then, to detect and locate vulnerabilities more accurately down to the line level, we present LineVul in Chapter 3 and Optimatch in Chapter 4. To accurately identify vulnerability types and explain detected vulnerabilities, we present a multi-objective optimization(MOO)-based SVC in Chapter 8 and a transformer-based hierarchical distillation approach in Chapter 5 to mitigate the data imbalance issue in SVC. To automatically recommend the repairs for vulnerable source code, we introduce a T5-based AVR, VulRepair, in Chapter 6. Moreover, in Chapter 7, we introduce VQM (vulnerability query and masking mechanism), which focuses on highlighting vulnerable code areas within self-attention during both the encoding of vulnerable source code and the decoding of the corresponding repair patches to enhance the accuracy of vulnerability fixes. In Chapter 8, we then integrate and deploy our proposed SVD, SVC, and AVR methods into a VSCode extension, AIBugHunter, that can (1) detect and locate vulnerabilities, (2) explain their vulnerability types, (3) estimate their severity, and (4) suggest repair patches to security analysts and software developers in IDE. In Chapter 9, we assess the performance of ChatGPT, specifically gpt-3.5-turbo and gpt-4, across the four prediction tasks. Our findings suggest that, despite their substantial model size and pre-training data, further fine-tuning is necessary for ChatGPT to effectively adapt to all vulnerability prediction tasks. In conclusion, this thesis has primarily focused on advancing deep learning-based methods for vulnerability detection, classification, and repair within the software development phase of DevSecOps. However, to address the security needs of other phases within the DevSecOps lifecycle, such

---

as planning, testing, deployment, and monitoring, additional research is required. In Chapter 10, we present a systematic literature review (SLR) that examines existing AI-driven security approaches across the DevSecOps pipeline. This review identifies key challenges and derives future research opportunities to extend the impact of AI-driven solutions beyond the software development phase, thereby covering the broader DevSecOps framework.

---

## References

- [1] Michael Fu and Chakkrit Tantithamthavorn. "LineVul: A Transformer-based Line-Level Vulnerability Prediction". In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE. 2022.
- [2] Michael Fu et al. "VulExplainer: A Transformer-based Hierarchical Distillation for Explaining Vulnerability Types". In: *IEEE Transactions on Software Engineering* (2023).
- [3] Michael Fu et al. "VulRepair: a T5-based automated software vulnerability repair". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2022, pp. 935–947.
- [4] Michael Fu et al. "Vision Transformer-Inspired Automated Vulnerability Repair". In: *ACM Transactions on Software Engineering and Methodology* (2023).
- [5] Michael Fu et al. "AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities". In: *Empirical Software Engineering* 29.1 (2024), p. 4.
- [6] Michael Fu et al. *ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?* 2023. arXiv: [2310.09810 \[cs.SE\]](https://arxiv.org/abs/2310.09810).
- [7] Michael Fu, Jirat Pasuksmit, and Chakkrit Tantithamthavorn. "AI for DevSecOps: A Landscape and Future Opportunities". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2025).
- [8] OpenAI. *ChatGPT (gpt-3.5-turbo/gpt-4o-mini/gpt-4o)*. Accessed: 2024-09-03. 2024. url: <https://chat.openai.com/>.
- [9] Kyle. *Example C++ Vulnerable Function - unPremulSkImageToPremul*. [https://github.com/kwyatt/webrtc\\_src\\_third\\_party/commit/a87bf9b1dbdc6e0fb80091a901ab7d5a05d64ecd.2016](https://github.com/kwyatt/webrtc_src_third_party/commit/a87bf9b1dbdc6e0fb80091a901ab7d5a05d64ecd.2016).
- [10] GoPro. *Example C Vulnerable Function - IsValidSize*. <https://github.com/gopro/gpmf-parser/commit/341f12cd5b97ab419e53853ca00176457c9f1681>. 2019.
- [11] CSRC. *Definition of Software Vulnerability*. [https://csrc.nist.gov/glossary/term/software\\_vulnerability](https://csrc.nist.gov/glossary/term/software_vulnerability). 2022.
- [12] ASIC. *Guidance for consumers impacted by the Optus data breach*. <https://asic.gov.au/about-asic/news-centre/news-items/guidance-for-consumers-impacted-by-the-optus-data-breach/>. 2022.
- [13] Medibank. *Cyber event updates and support - Medibank*. <https://www.medibank.com.au/health-insurance/info/cyber-security/>. 2022.
- [14] Saikat Chakraborty et al. "Deep learning based vulnerability detection: Are we there yet". In: *IEEE Transactions on Software Engineering (TSE)* (2021).
- [15] Zhen Li et al. "SySeVR: A framework for using deep learning to detect software vulnerabilities". In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2021).
- [16] Zhen Li et al. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *arXiv e-prints* (2018), arXiv-1801.
- [17] Rebecca Russell et al. "Automated vulnerability detection in source code using deep representation learning". In: *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE. 2018, pp. 757–762.
- [18] Yaqin Zhou et al. "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, pp. 10197–10207.

- 
- [19] Xinda Wang et al. “A machine learning approach to classify security patches into vulnerability types”. In: *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2020, pp. 1–9.
- [20] Siddhartha Shankar Das et al. “V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities”. In: *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2021, pp. 1–12.
- [21] Ehsan Aghaei, Waseem Shadid, and Ehab Al-Shaer. “Threatzoom: Hierarchical neural network for cves to cws classification”. In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2020, pp. 23–41.
- [22] Zimin Chen et al. “Sequencer: Sequence-to-sequence learning for end-to-end program repair”. In: *IEEE Transactions on Software Engineering* 47.9 (2019), pp. 1943–1959.
- [23] Zimin Chen, Steve Kommarusch, and Martin Monperrus. “Neural Transfer Learning for Repairing Security Vulnerabilities in C Code”. In: *IEEE Transactions on Software Engineering* (2021).
- [24] *Cybercrime To Cost \$10.5 Trillion Annually By 2025*. <https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/>. 2016.
- [25] *THE COST OF CYBERCRIME*. [https://www.accenture.com/\\_acnmedia/pdf-96/accenture-2019-cost-of-cybercrime-study-final.pdf](https://www.accenture.com/_acnmedia/pdf-96/accenture-2019-cost-of-cybercrime-study-final.pdf). 2019.
- [26] *Flawfinder*. <https://dwheeler.com/flawfinder/>. 2022.
- [27] *RATS*. <https://code.google.com/archive/p/rough-auditing-tool-for-security/>. 2022.
- [28] Daniel Marjamäki. *Cppcheck*. <https://cppcheck.sourceforge.io/>. 2007.
- [29] Checkmarx. *Checkmarx*. <https://checkmarx.com/>. 2006.
- [30] Roland Croft et al. “An empirical study of rule-based and learning-based approaches for static application security testing”. In: *In the Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2021, pp. 1–12.
- [31] Mianxue Gu et al. “Hierarchical Attention Network for Interpretable and Fine-Grained Vulnerability Detection”. In: *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2022, pp. 1–6.
- [32] Yi Li, Shaohua Wang, and Tien N Nguyen. “Vulnerability detection with fine-grained interpretations”. In: *29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*. Association for Computing Machinery, Inc. 2021, pp. 292–303.
- [33] Tom Britton et al. “Reversible debugging software-quantify the time and cost saved using reversible debuggers”. In: *University of Cambridge* (2013).
- [34] Dongliang Mu et al. “Understanding the reproducibility of crowd-reported security vulnerabilities”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 919–936.
- [35] Jianlei Chi et al. “Seqtrans: Automatic vulnerability fix via sequence to sequence learning”. In: *IEEE Transactions on Software Engineering* (2022).
- [36] Nan Jiang, Thibaud Lutellier, and Lin Tan. “Cure: Code-aware neural machine translation for automatic program repair”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1161–1173.
- [37] CWE. *Common Weakness Enumeration (CWE)*. <https://cwe.mitre.org/index.html>. 2006.

- 
- [38] Fixing Vulnerabilities Costs 100x More If You Don't Understand the Weakness. [https://medium.com/@CWE\\_CAPEC/fixing-vulnerabilities-costs-100x-more-if-you-dont-understand-the-weakness-c3877f68d8a6](https://medium.com/@CWE_CAPEC/fixing-vulnerabilities-costs-100x-more-if-you-dont-understand-the-weakness-c3877f68d8a6). 2020.
- [39] Microsoft. *Visual Studio Code*. <https://code.visualstudio.com/>. 2022.
- [40] Zhangyin Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. 2020, pp. 1536–1547.
- [41] Yonghee Shin and Laurie Williams. “An empirical model to predict security vulnerabilities using code complexity metrics”. In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 2008, pp. 315–317.
- [42] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. “Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista”. In: *2010 Third international conference on software testing, verification and validation*. IEEE. 2010, pp. 421–428.
- [43] MITRE. *MITRE corporation*. <https://www.mitre.org/>. 2022.
- [44] Jason Wei and Kai Zou. “EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 6382–6388.
- [45] CVSS. *Common Vulnerability Scoring System (CVSS)*. <https://nvd.nist.gov/vuln-metrics/cvss>. 2003.
- [46] Triet Huynh Minh Le et al. “Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 717–729.
- [47] Ion Babalau et al. “Severity Prediction of Software Vulnerabilities based on their Text Description”. In: *2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE. 2021, pp. 171–177.
- [48] Xi Gong et al. “Joint prediction of multiple vulnerability characteristics through multi-task learning”. In: *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE. 2019, pp. 31–40.
- [49] Thibaud Lutellier et al. “Coconut: combining context-aware neural translation models using ensemble for program repair”. In: *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 2020, pp. 101–114.
- [50] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *54th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics (ACL). 2016, pp. 1715–1725.
- [51] Jules White et al. “Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design”. In: *arXiv preprint arXiv:2303.07839* (2023).
- [52] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.
- [53] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: [2303.08774 \[cs.CL\]](https://arxiv.org/abs/2303.08774).
- [54] Chenyuan Zhang et al. “Prompt-Enhanced Software Vulnerability Detection Using Chat-GPT”. In: *arXiv preprint arXiv:2308.12697* (2023).
- [55] Emanuele Antonio Napoli and Valentina Gatteschi. “Evaluating ChatGPT for Smart Contracts Vulnerability Correction”. In: *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2023, pp. 1828–1833.

- 
- [56] Hammond Pearce et al. "Examining zero-shot vulnerability repair with large language models". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 2339–2356.
- [57] Chunqiu Steven Xia and Lingming Zhang. "Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT". In: *arXiv preprint arXiv:2304.00385* (2023).
- [58] Dominik Sobania et al. "An analysis of the automatic bug fixing performance of chatgpt". In: *arXiv preprint arXiv:2301.08653* (2023).
- [59] Arnold Johnson et al. "Guide for security-focused configuration management of information systems". In: *NIST special publication 800.128* (2011), pp. 16–16.
- [60] *ProxyLogon Flaw*. <https://proxylogon.com/>. 2022.
- [61] *Microsoft Exchange Flaw: Attacks Surge After Code Published*. <https://www.bankinfosecurity.com/ms-exchange-flaw-causes-spike-in-trdownloader-gen-trojans-a-16236>. 2021.
- [62] Mukesh Kumar Gupta, MC Govil, and Girdhari Singh. "Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey". In: *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*. IEEE. 2014, pp. 1–5.
- [63] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 4171–4186.
- [64] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016).
- [65] Rex Ying et al. "Gnnexplainer: Generating explanations for graph neural networks". In: *Advances in neural information processing systems (NeurIPS) 32* (2019), p. 9240.
- [66] *CWE-787*. <https://cwe.mitre.org/data/definitions/787.html>. 2022.
- [67] Yinhan Liu et al. "Roberta: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).
- [68] Hamel Husain et al. "Codesearchnet challenge: Evaluating the state of semantic code search". In: *arXiv preprint arXiv:1909.09436* (2019).
- [69] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems (NeurIPS)*. 2017, pp. 5998–6008.
- [70] Jiahao Fan et al. "AC/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 508–512.
- [71] Thomas Wolf et al. "Huggingface's transformers: State-of-the-art natural language processing". In: *arXiv preprint arXiv:1910.03771* (2019).
- [72] R. Collobert, K. Kavukcuoglu, and C. Farabet. "Torch7: A Matlab-like Environment for Machine Learning". In: *BigLearn, NIPS Workshop*. 2011.
- [73] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization". In: *International Conference on Learning Representations*. 2018.
- [74] Chanathip Pornprasit and Chakkrit Tantithamthavorn. "JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction". In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2021.
- [75] Supatsara Wattanakriengkrai et al. "Predicting defective lines using a model-agnostic technique". In: *IEEE Transactions on Software Engineering (TSE)* (2020).
- [76] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic attribution for deep networks". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 3319–3328.

- 
- [77] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. “Deep inside convolutional networks: Visualising image classification models and saliency maps”. In: *arXiv preprint arXiv:1312.6034* (2013).
- [78] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. “Learning important features through propagating activation differences”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 3145–3153.
- [79] Marco Ancona et al. “Towards better understanding of gradient-based attribution methods for Deep Neural Networks”. In: *6th International Conference on Learning Representations (ICLR)*. Arxiv-Computer Science. 2018.
- [80] Scott M Lundberg and Su-In Lee. “A unified approach to interpreting model predictions”. In: *Proceedings of the 31st international conference on neural information processing systems*. 2017, pp. 4768–4777.
- [81] Chris Parnin and Alessandro Orso. “Are automated debugging techniques actually helping programmers?” In: *Proceedings of the 2011 international symposium on software testing and analysis*. 2011, pp. 199–209.
- [82] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “” Why should i trust you?” Explaining the predictions of any classifier”. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1135–1144.
- [83] Rafael-Michael Karampatsis et al. “Big code!= big vocabulary: Open-vocabulary models for source code”. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE. 2020, pp. 1073–1085.
- [84] Suraj Yatish et al. “Mining Software Defects: Should We Consider Affected Releases?” In: *ICSE*. 2019, pp. 654–665.
- [85] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. “AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models”. In: *ICSME*. 2018, pp. 92–103.
- [86] Chakkrit Tantithamthavorn. “Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Modelling”. In: *Companion Proceeding of the International Conference on Software Engineering (ICSE)*. 2016, pp. 867–870.
- [87] Chakkrit Tantithamthavorn et al. “The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models”. In: *ICSE*. 2015, pp. 812–823.
- [88] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E Hassan. “The Impact of Correlated Metrics on Defect Models”. In: *IEEE Transactions on Software Engineering* (2021).
- [89] Chakkrit Tantithamthavorn et al. “The Impact of Automated Parameter Optimization on Defect Prediction Models”. In: *TSE* (2019).
- [90] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. “The impact of class rebalancing techniques on the performance and interpretation of defect prediction models”. In: *IEEE Transactions on Software Engineering* 46.11 (2018), pp. 1200–1219.
- [91] Chakkrit Tantithamthavorn et al. “Automated Parameter Optimization of Classification Techniques for Defect Prediction Models”. In: *ICSE*. 2016, pp. 321–332.
- [92] Chakkrit Tantithamthavorn et al. “Comments on “Researcher Bias: The Use of Machine Learning in Software Defect Prediction””. In: *TSE* 42.11 (2016), pp. 1092–1094.
- [93] Zhen Li et al. “Vuldeelocator: a deep learning-based fine-grained vulnerability detector”. In: *IEEE Transactions on Dependable and Secure Computing* (2021).

- 
- [94] Chanathip Pornprasit and Chakkrit Tantithamthavorn. “DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction”. In: *IEEE Transactions on Software Engineering* (2022).
- [95] Hoa Khanh Dam et al. “Automatic feature learning for vulnerability prediction”. In: *arXiv preprint arXiv:1708.02368* (2017).
- [96] Chakkrit Tantithamthavorn and Jirayus Jiarpakdee. “Explainable AI for Software Engineering”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 1–2.
- [97] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. “Actionable Analytics: Stop Telling Me What It Is; Please Tell Me What To Do”. In: *IEEE Software* 38.4 (2021), pp. 115–120.
- [98] Chakkrit Kla Tantithamthavorn and Jirayus Jiarpakdee. “Explainable ai for software engineering”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 1–2.
- [99] Jirayus Jiarpakdee et al. “An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models”. In: *IEEE Transactions on Software Engineering (TSE)* (2020), To Appear.
- [100] Chaiyakarn Khanan et al. “JITBot: An Explainable Just-In-Time Defect Prediction Bot”. In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2020, pp. 1336–1339.
- [101] Dilini Rajapaksha et al. “SQAPlanner: Generating data-informed software quality improvement plans”. In: *IEEE Transactions on Software Engineering* (2021).
- [102] Chanathip Pornprasit et al. “PyExplainer: Explaining the Predictions of Just-In-Time Defect Models”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 407–418.
- [103] Michael Fu and Chakkrit Tantithamthavorn. “GPT2SP: A Transformer-Based Agile Story Point Estimation Approach”. In: *IEEE Transactions on Software Engineering* (2022).
- [104] Sarthak Jain and Byron C Wallace. “Attention is not Explanation”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 3543–3556.
- [105] Sarah Wiegreffe and Yuval Pinter. “Attention is not not Explanation”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 11–20.
- [106] National Vulnerability Database. <https://nvd.nist.gov/>. 2022.
- [107] NIST. Apache Struts Vulnerability (CVE-2021-31805). <https://nvd.nist.gov/vuln/detail/CVE-2021-31805>. 2022.
- [108] David Hin et al. “LineVD: Statement-level Vulnerability Detection using Graph Neural Networks”. In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE. 2022.
- [109] Yangruibo Ding et al. “VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements”. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 959–970.
- [110] CWE. 2023 CWE Top 25 Most Dangerous Software Weaknesses. [https://cwe.mitre.org/top25/archive/2023/2023\\_top25\\_list.html](https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html). 2023.

- 
- [111] Aaron Van Den Oord, Oriol Vinyals, et al. “Neural discrete representation learning”. In: *Advances in neural information processing systems (NeurIPS) 30* (2017).
  - [112] Jean Feydy et al. “Interpolating between optimal transport and mmd using sinkhorn divergences”. In: *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR. 2019, pp. 2681–2690.
  - [113] Yunhui Zheng et al. “D2a: A dataset built for ai-based vulnerability detection methods using differential analysis”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-Szheng2021d2aEIP)*. IEEE. 2021, pp. 111–120.
  - [114] Saurabh Pujar et al. “Analyzing source code vulnerabilities in the D2A dataset with ML ensembles and C-BERT”. In: *Empirical Software Engineering* 29.2 (2024), p. 48.
  - [115] Gaspard Monge. “Mémoire sur la théorie des déblais et des remblais”. In: *Mem. Math. Phys. Acad. Royale Sci.* (1781), pp. 666–704.
  - [116] Matthew Thorpe. “Introduction to optimal transport”. In: *Lecture Notes* 3 (2019).
  - [117] Cédric Villani. “Optimal Transport: Old and New”. In: 2008.
  - [118] V. Nguyen et al. “Deep Domain Adaptation for Vulnerable Code Function Identification”. In: *The International Joint Conference on Neural Networks (IJCNN)*. 2019.
  - [119] Van Nguyen et al. “Dual-Component Deep Domain Adaptation: A New Approach for Cross Project Software Vulnerability Detection”. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2020).
  - [120] Van-Anh Nguyen et al. “ReGVD: Revisiting graph neural networks for vulnerability detection”. In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 2022, pp. 178–182.
  - [121] Daya Guo et al. “GraphCodeBERT: Pre-training Code Representations with Data Flow”. In: *International Conference on Learning Representations*. 2021.
  - [122] Van Nguyen et al. “Information-theoretic source code vulnerability highlighting”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–8.
  - [123] Benjamin Steenhoek, Hongyang Gao, and Wei Le. “Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection”. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society. 2023, pp. 166–178.
  - [124] Daya Guo et al. “Unixcoder: Unified cross-modal pre-training for code representation”. In: *arXiv preprint arXiv:2203.03850* (2022).
  - [125] OpenAI. *ChatGPT*. <https://openai.com/blog/chatgpt>. 2022.
  - [126] Google. *BARD*. <https://bard.google.com/>. 2023.
  - [127] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).
  - [128] Marco Cuturi. “Sinkhorn Distances: Lightspeed Computation of Optimal Transport”. In: *Advances in Neural Information Processing Systems*. Ed. by C.J. Burges et al. Vol. 26. Curran Associates, Inc., 2013.
  - [129] Tuan Nguyen et al. “TIDOT: A Teacher Imitation Learning Approach for Domain Adaptation with Optimal Transport”. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*. Ed. by Zhi-Hua Zhou. Main Track. International Joint Conferences on Artificial Intelligence Organization, Aug. 2021, pp. 2862–2868. doi: [10.24963/ijcai.2021/394](https://doi.org/10.24963/ijcai.2021/394). url: <https://doi.org/10.24963/ijcai.2021/394>.
  - [130] Nhat Ho et al. “Multilevel clustering via Wasserstein means”. In: *International conference on machine learning*. PMLR. 2017, pp. 1501–1509.

- 
- [131] Yue Wang et al. “Codet5+: Open code large language models for code understanding and generation”. In: *arXiv preprint arXiv:2305.07922* (2023).
- [132] Yue Wang et al. “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation”. In: *in Proceedings of the International Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2021).
- [133] Shuai Lu et al. “Codexglue: A machine learning benchmark dataset for code understanding and generation”. In: *arXiv preprint arXiv:2102.04664* (2021).
- [134] Fabian Yamaguchi et al. “Modeling and discovering vulnerabilities with code property graphs”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 590–604.
- [135] Yahui Chen. “Convolutional neural network for sentence classification”. MA thesis. University of Waterloo, 2015.
- [136] Yizheng Chen et al. “Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection”. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 2023, pp. 654–668.
- [137] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 3982–3992.
- [138] Mihael Ankerst et al. “OPTICS: Ordering points to identify the clustering structure”. In: *ACM Sigmod record 28.2* (1999), pp. 49–60.
- [139] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. “Data quality for software vulnerability datasets”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 121–133.
- [140] Google. *Key Statistics of the Google Bug Bounty Program*. <https://bughunters.google.com/about/key-stats>. 2022.
- [141] MSRC. *Microsoft Bug Bounty Programs Year in Review: \$13.6M in Rewards*. <https://msrc-blog.microsoft.com/2021/07/08/microsoft-bug-bounty-programs-year-in-review-13-6m-in-rewards/>. 2021.
- [142] Dan Gurfinkel. *Charting the future of our bug bounty program*. <https://engineering.fb.com/2021/12/15/security/bug-bounty-scraping/>. 2021.
- [143] Serkan Özkan. *Log4j: Security Vulnerabilities*. [https://www.cvedetails.com/vulnerability-list/vendor\\_id-45/product\\_id-37215/Apache-Log4j.html](https://www.cvedetails.com/vulnerability-list/vendor_id-45/product_id-37215/Apache-Log4j.html). 2021.
- [144] Tianyi Wang, Shengzhi Qin, and Kam Pui Chow. “Towards Vulnerability Types Classification Using Pure Self-Attention: A Common Weakness Enumeration Based Approach”. In: *2021 IEEE 24th International Conference on Computational Science and Engineering (CSE)*. IEEE. 2021, pp. 146–153.
- [145] Tsung-Yi Lin et al. “Focal loss for dense object detection”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988.
- [146] Yin Cui et al. “Class-balanced loss based on effective number of samples”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 9268–9277.
- [147] Kaidi Cao et al. “Learning imbalanced datasets with label-distribution-aware margin loss”. In: *Advances in neural information processing systems* 32 (2019).
- [148] Aditya Krishna Menon et al. “Long-tail learning via logit adjustment”. In: *International Conference on Learning Representations*. 2020.

- 
- [149] Yu Li et al. “Overcoming classifier imbalance for long-tail object detection with balanced group softmax”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 10991–11000.
- [150] Liuyu Xiang, Guiguang Ding, and Jungong Han. “Learning from multiple experts: Self-paced knowledge distillation for long-tailed classification”. In: *European Conference on Computer Vision*. Springer. 2020, pp. 247–263.
- [151] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. “Distilling the Knowledge in a Neural Network”. In: *stat* 1050 (2015), p. 9.
- [152] CWE Community. *CWE Abstract Type (CWE Glossary)*. <https://cwe.mitre.org/documents/glossary/>. 2021.
- [153] Yoon Kim. “Convolutional Neural Networks for Sentence Classification”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751. doi: 10.3115/v1/D14-1181. url: <https://aclanthology.org/D14-1181>.
- [154] Hugo Touvron et al. “Training data-efficient image transformers & distillation through attention”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 10347–10357.
- [155] Daya Guo et al. “Graphcodebert: Pre-training code representations with data flow”. In: *arXiv preprint arXiv:2009.08366* (2020).
- [156] Shuai Lu et al. “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation”. In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. 2021.
- [157] Van Nguyen et al. “Information-theoretic Source Code Vulnerability Highlighting”. In: *International Joint Conference on Neural Networks (IJCNN)*. 2021.
- [158] Chakkrit Tantithamthavorn et al. “Explainable ai for se: Challenges and future directions”. In: *IEEE Software* 40.3 (2023), pp. 29–33.
- [159] Jurgen Cito et al. “Expert perspectives on explainability”. In: *IEEE Software* 40.03 (2023), pp. 84–88.
- [160] Yue Liu et al. “Explainable ai for android malware detection: Towards understanding why the models perform so well?” In: *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2022, pp. 169–180.
- [161] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John Grundy. “Practitioners’ Perceptions of the Goals and Visual Explanations of Defect Prediction Models”. In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2021, To Appear.
- [162] Cristian Buciluă, Rich Caruana, and Alexandru Niculescu-Mizil. “Model compression”. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 535–541.
- [163] A real-world vulnerability example of CWE-787. <https://github.com/chromium/chromium/commit/d59a4441697f6253e7dc3f7ae5caad6e5fd2c778>. 2016.
- [164] Sarang Na, Taeeun Kim, and Hwankuk Kim. “A study on the classification of common vulnerabilities and exposures using naïve bayes”. In: *International Conference on Broadband and Wireless Computing, Communication and Applications*. Springer. 2016, pp. 657–662.
- [165] Masaki Aota et al. “Automation of vulnerability classification from its description using machine learning”. In: *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2020, pp. 1–7.
- [166] Bo Shuai et al. “Automatic classification for vulnerability based on machine learning”. In: *2013 IEEE International Conference on Information and Automation (ICIA)*. IEEE. 2013, pp. 312–318.

- 
- [167] CWE Community. 2022 *CWE Top 25 Most Dangerous Software Weaknesses*. [https://cwe.mitre.org/top25/archive/2022/2022\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html). 2022.
- [168] Longhui Wei et al. “Circumventing outliers of autoaugment with knowledge distillation”. In: *European Conference on Computer Vision*. Springer. 2020, pp. 608–625.
- [169] Youngkyu Hong et al. “Disentangling label distribution for long-tailed visual recognition”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 6626–6636.
- [170] Bingyi Kang et al. “Decoupling Representation and Classifier for Long-Tailed Recognition”. In: *International Conference on Learning Representations*. 2019.
- [171] Kevin Clark et al. “Electra: Pre-training text encoders as discriminators rather than generators”. In: *arXiv preprint arXiv:2003.10555* (2020).
- [172] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.
- [173] Yujia Li et al. “Gated Graph Sequence Neural Networks”. In: *Proceedings of ICLR’16*. 2016.
- [174] Foster Provost. “Machine learning from imbalanced data sets 101”. In: *Proceedings of the AAAI’2000 workshop on imbalanced data sets*. Vol. 68. AAAI Press. 2000, pp. 1–3.
- [175] Zhi-Hua Zhou and Xu-Ying Liu. “Training cost-sensitive neural networks with methods addressing the class imbalance problem”. In: *IEEE Transactions on knowledge and data engineering* 18.1 (2005), pp. 63–77.
- [176] Guillem Collell, Drazen Prelec, and Kaustubh Patil. “Reviving threshold-moving: a simple plug-in bagging ensemble for binary and multiclass imbalanced data”. In: *arXiv preprint arXiv:1606.08698* (2016).
- [177] Chandra Thapa et al. “Transformer-Based Language Models for Software Vulnerability Detection: Performance, Model’s Security and Platforms”. In: *arXiv preprint arXiv:2204.03214* (2022).
- [178] Yongshun Zhang et al. “Bag of tricks for long-tailed visual recognition with deep convolutional neural networks”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 35. 2021, pp. 3447–3455.
- [179] Ajinkya More. “Survey of resampling techniques for improving classification performance in unbalanced datasets”. In: *arXiv preprint arXiv:1608.06048* (2016).
- [180] Mateusz Buda, Atsuto Maki, and Maciej A Mazurowski. “A systematic study of the class imbalance problem in convolutional neural networks”. In: *Neural networks* 106 (2018), pp. 249–259.
- [181] Hongyi Zhang et al. “mixup: Beyond Empirical Risk Minimization”. In: *International Conference on Learning Representations*. 2017.
- [182] Vikas Verma et al. “Manifold mixup: Better representations by interpolating hidden states”. In: *International conference on machine learning*. PMLR. 2019, pp. 6438–6447.
- [183] Boyan Zhou et al. “Bbn: Bilateral-branch network with cumulative learning for long-tailed visual recognition”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 9719–9728.
- [184] Michael Fu. *Replication Package of VulExplainer*. <https://github.com/awsm-research/VulExplainer>. 2022.
- [185] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006. isbn: 0321444426.

- 
- [186] 25+ cyber security vulnerability statistics and facts of 2021. <https://www.comparitech.com/blog/information-security/cybersecurity-vulnerability-statistics/>. 2021.
  - [187] Gary Wassermann and Zhendong Su. "Sound and precise analysis of web applications for injection vulnerabilities". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 32–41.
  - [188] Gary Wassermann and Zhendong Su. "Static detection of cross-site scripting vulnerabilities". In: *2008 ACM/IEEE 30th International Conference on Software Engineering (ICSE)*. IEEE. 2008, pp. 171–180.
  - [189] Hua Yan et al. "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 327–337.
  - [190] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "Phasar: An inter-procedural static analysis framework for c/c++". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2019, pp. 393–410.
  - [191] Huanting Wang et al. "Combining graph-based learning with automated data collection for code vulnerability detection". In: *IEEE Transactions on Information Forensics and Security* 16 (2020), pp. 1943–1958.
  - [192] Patrick J Morrison et al. "Are vulnerabilities discovered and resolved like other defects?" In: *Empirical Software Engineering* 23.3 (2018), pp. 1383–1421.
  - [193] Zhe Yu et al. "Improving vulnerability inspection efficiency using active learning". In: *IEEE Transactions on Software Engineering* (2019).
  - [194] Chakkrit Tantithamthavorn et al. "An Empirical Comparison of Model Validation Techniques for Defect Prediction Models". In: *IEEE Transactions on Software Engineering (TSE)* (2017).
  - [195] Guru Bhandari, Amara Naseer, and Leon Moonen. "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software". In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2021, pp. 30–39.
  - [196] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).
  - [197] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems (NeurIPS) 27* (2014).
  - [198] Michele Tufano et al. "On learning meaningful code changes via neural machine translation". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 25–36.
  - [199] Abram Hindle et al. "On the naturalness of software". In: *Communications of the ACM* 59.5 (2016), pp. 122–131.
  - [200] Wenyuan Zeng et al. "Efficient summarization with read-again and copy mechanism". In: *arXiv preprint arXiv:1611.03382* (2016).
  - [201] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. "Self-attention with relative position representations". In: *in Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)* (2018).
  - [202] Colin Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *Journal of Machine Learning Research (JMLR)* (2019).
  - [203] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. "Auto-Transform: Automated Code Transformation to Support Modern Code Review Process".

- 
- In: *Proceedings of the ACM/IEEE 44nd International Conference on Software Engineering (ICSE)*. 2022.
- [204] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).
- [205] Ehsan Mashhadi and Hadi Hemmati. “Applying codebert for automated program repair of java simple bugs”. In: *in Proceedings of the International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 505–509.
- [206] Alec Radford et al. “Improving language understanding by generative pre-training”. In: (2018).
- [207] Yi Li, Shaohua Wang, and Tien N Nguyen. “Dlfix: Context-based code transformation learning for automated program repair”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 602–614.
- [208] Kui Liu et al. “A critical review on the evaluation of automated program repair systems”. In: *Journal of Systems and Software* 171 (2021), p. 110817.
- [209] Edgescan. 2022 Vulnerability Statistic Report. <https://www.edgescan.com/2022-vulnerability-statistics-report-lp/>. 2022.
- [210] Berkay Berabi et al. “Tfix: Learning to fix coding errors with a text-to-text transformer”. In: *the Proceedings of the International Conference on Machine Learning (ICML)*. PMLR. 2021, pp. 780–791.
- [211] Nicolas Carion et al. “End-to-end object detection with transformers”. In: *the Proceedings of the European Conference on Computer Vision (ECCV)*. Springer. 2020, pp. 213–229.
- [212] Xizhou Zhu et al. “Deformable DETR: Deformable Transformers for End-to-End Object Detection”. In: *the Proceedings of the International Conference on Learning Representations (ICLR)*. 2020.
- [213] Yingming Wang et al. “Anchor DETR: Query Design for Transformer-Based Object Detection”. In: *arXiv preprint arXiv:2109.07107*. 2021.
- [214] GoPro. An example software vulnerability from GoPro systems. <https://github.com/gopro/gpmf-parser/commit/341f12cd5b97ab419e53853ca00176457c9f1681>. 2019.
- [215] ImageMagick. ImageMagick. <https://github.com/ADVAN-ELAA-8QM-PRC1/platform-external-ImageMagick/commit/d8ab7f046587f2e9f734b687ba7e6e10147c294b>. 2016.
- [216] Colin Raffel et al. “Exploring the limits of transfer learning with a unified text-to-text transformer.” In: *J. Mach. Learn. Res.* 21.140 (2020), pp. 1–67.
- [217] Tomas Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *Advances in neural information processing systems* 26 (2013).
- [218] 2020 CWE Top 25 Most Dangerous Software Weaknesses. [https://cwe.mitre.org/top25/archive/2020/2020\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html). 2020.
- [219] Barbara A Kitchenham and Shari L Pfleeger. “Personal opinion surveys”. In: *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 63–92.
- [220] Yue Liu et al. “Autoupdate: Automatically recommend code updates for android apps”. In: *arXiv preprint arXiv:2209.07048* (2022).
- [221] Yang Hong et al. “Commentfinder: a simpler, faster, more accurate code review comments recommendation”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 507–519.
- [222] Yang Hong, Chakkrit Kla Tantithamthavorn, and Patanamon Pick Thongtanunam. “Where should I look at? Recommending lines that reviewers should pay attention to”. In: 2022

- 
- IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 1034–1045.
- [223] Chanathip Pornprasit et al. “D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation”. In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2023, pp. 296–307.
- [224] Wannita Takerngsaksiri, Chakkrit Tantithamthavorn, and Yuan-Fang Li. “Syntax-Aware On-the-Fly Code Completion”. In: *arXiv preprint arXiv:2211.04673* (2022).
- [225] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. “A3Test: Assertion-Augmented Automated Test Case Generation”. In: *arXiv preprint arXiv:2302.10352* (2023).
- [226] Van Nguyen et al. “An Information-Theoretic and Contrastive Learning-based Approach for Identifying Code Statements Causing Software Vulnerability”. In: *arXiv preprint arXiv:2209.10411* (2022).
- [227] Van Nguyen et al. “Cross Project Software Vulnerability Detection via Domain Adaptation and Max-Margin Principle”. In: *arXiv preprint arXiv:2209.10406* (2022).
- [228] Michael Fu et al. “Learning to Quantize Vulnerability Patterns and Match to Locate Statement-Level Vulnerabilities”. In: *arXiv preprint arXiv:2306.06109* (2023).
- [229] Elizabeth Dinella et al. “Hoppity: Learning graph transformations to detect and fix bugs in programs”. In: *the Proceedings of the International Conference on Learning Representations (ICLR)*. 2020.
- [230] WhiteSource. *What are the most secure programming languages?* <https://www.mend.io/most-secure-programming-languages/>. 2019.
- [231] Kev Zettler. *The DevSecOp tools that secure DevOps workflows*. <https://www.redhat.com/en/topics/devops/what-is-devsecops>. 2022.
- [232] Zhao Chen et al. “Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks”. In: *International conference on machine learning*. PMLR. 2018, pp. 794–803.
- [233] Alex Kendall, Yarin Gal, and Roberto Cipolla. “Multi-task learning using uncertainty to weigh losses for scene geometry and semantics”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7482–7491.
- [234] Ozan Sener and Vladlen Koltun. “Multi-task learning as multi-objective optimization”. In: *Advances in neural information processing systems* 31 (2018).
- [235] NVD. CVSS Version 3.1. <https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator>. 2019.
- [236] Xue Yuan et al. “Deep Neural Embedding for Software Vulnerability Discovery: Comparison and Optimization”. In: *Security and Communication Networks* 2022 (2022).
- [237] Chenguang Zhu et al. “BERT-Based Vulnerability Type Identification with Effective Program Representation”. In: *Wireless Algorithms, Systems, and Applications: 17th International Conference, WASA 2022, Dalian, China, November 24–26, 2022, Proceedings, Part I*. Springer. 2022, pp. 271–282.
- [238] CWE. *CWE Abstract Type - Class Weakness*. <https://cwe.mitre.org/documents/glossary/#Class%20Weakness>. 2021.
- [239] CWE. *CWE Abstract Type - Variant Weakness*. <https://cwe.mitre.org/documents/glossary/#Variant%20Weakness>. 2021.
- [240] Andre Renaud. *A Vulnerable C Function*. <https://github.com/AndreRenaud/PDFGen/commit/8f9b3202f67feb386c9974520d9bcc4531350fff>. 2018.

- 
- [241] Georgios Spanos and Lefteris Angelis. "A multi-target approach to estimate software vulnerability characteristics and severity scores". In: *Journal of Systems and Software* 146 (2018), pp. 152–166.
- [242] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. "Evaluation of ChatGPT Model for Vulnerability Detection". In: *arXiv preprint arXiv:2304.07232* (2023).
- [243] Microsoft. *What is DevOps?* <https://learn.microsoft.com/en-us/devops/what-is-devops>. 2023.
- [244] Andi Mann et al. *State of DevOps Report, 2019*. 2019.
- [245] Floris MA Erich, Chintan Amrit, and Maya Daneva. "A qualitative study of DevOps usage in practice". In: *Journal of software: Evolution and Process* 29.6 (2017), e1885.
- [246] Håvard Myrbakken and Ricardo Colomo-Palacios. "DevSecOps: a multivocal literature review". In: *Software Process Improvement and Capability Determination: 17th International Conference, SPICE 2017, Palma de Mallorca, Spain, October 4–5, 2017, Proceedings*. Springer. 2017, pp. 17–29.
- [247] M Mehdi Kholoosi, M Ali Babar, and Cemal Yilmaz. "Empirical Analysis of Software Vulnerabilities Causing Timing Side Channels". In: *2023 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2023, pp. 1–9.
- [248] Michael A Howard. "A process for performing security code reviews". In: *IEEE Security & privacy* 4.4 (2006), pp. 74–79.
- [249] Wachiraphan Charoenwet et al. "An Empirical Study of Static Analysis Tools for Secure Code Review". In: *arXiv preprint arXiv:2407.12241* (2024).
- [250] Roshan N Rajapakse et al. "Challenges and solutions when adopting DevSecOps: A systematic review". In: *Information and software technology* 141 (2022), p. 106700.
- [251] GuardRails. *From Security Last to DevSecOps: The Driving Forces Behind the Transformation*. <https://www.guardrails.io/blog/from-security-last-to-devsecops-the-driving-forces-behind-the-transformation/>. 2023.
- [252] Google. *2023 State of DevOps Report*. <https://cloud.google.com/devops/state-of-devops>. 2023.
- [253] Ahmed Bahaa et al. "Monitoring real time security attacks for IoT systems using DevSecOps: a systematic literature review". In: *Information* 12.4 (2021), p. 154.
- [254] Luís Prates et al. "Devsecops metrics". In: *Information Systems: Research, Development, Applications, Education: 12th SIGSAND/PLAIS EuroSymposium 2019, Gdańsk, Poland, September 19, 2019, Proceedings* 12. Springer. 2019, pp. 77–90.
- [255] Runfeng Mao et al. "Preliminary findings about devsecops from grey literature". In: *2020 IEEE 20th international conference on software quality, reliability and security (QRS)*. IEEE. 2020, pp. 450–457.
- [256] Tiina Leppänen, Anne Honkaranta, and Andrei Costin. "Trends for the DevOps security. A systematic literature review". In: *International Symposium on Business Modeling and Software Design*. Springer. 2022, pp. 200–217.
- [257] Muhammad Azeem Akbar et al. "Toward successful DevSecOps in software development organizations: A decision-making framework". In: *Information and Software Technology* 147 (2022), p. 106894.
- [258] Rennie Naidoo and Nicolaas Möller. "Building Software Applications Securely With DevSecOps: A Socio-Technical Perspective". In: *ECCWS 2022 21st European Conference on Cyber Warfare and Security*. Academic Conferences and publishing limited. 2022.

- 
- [259] Yolanda Valdés-Rodríguez et al. “Towards the Integration of Security Practices in Agile Software Development: A Systematic Mapping Review”. In: *Applied Sciences* 13.7 (2023), p. 4578.
- [260] Microsoft. *DevSecOps controls*. <https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/secure/devsecops-controls>. 2022.
- [261] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.
- [262] Robert S Arnold and Shawn A Bohner. “Impact analysis-towards a framework for comparison”. In: *1993 conference on software maintenance*. IEEE. 1993, pp. 292–301.
- [263] Anna Khrupa. *What Is Software Impact Analysis*. <https://qarea.com/blog/what-is-software-impact-analysis>. 2022.
- [264] Snyk. Snyk. <https://snyk.io>. 2015.
- [265] Daniel Stahl, Torvald Martensson, and Jan Bosch. “Continuous practices and devops: beyond the buzz, what does it all mean?” In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2017, pp. 440–448.
- [266] Mojtaba Shahin et al. “An empirical study of architecting for continuous delivery and deployment”. In: *Empirical Software Engineering* 24 (2019), pp. 1061–1108.
- [267] Jingyue Li et al. “Development with off-the-shelf components: 10 facts”. In: *IEEE software* 26.2 (2009), pp. 80–87.
- [268] Gede Artha Azriadi Prana et al. “Out of sight, out of mind? How vulnerable dependencies affect open-source projects”. In: *Empirical Software Engineering* 26 (2021), pp. 1–34.
- [269] GitHub. *Dependabot*. <https://github.com/dependabot>. 2024.
- [270] Peng Huang et al. “Confvalley: A systematic configuration validation framework for cloud services”. In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–16.
- [271] RedHat. *What is Infrastructure as Code (IaC)?* <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac>. 2022.
- [272] Michele Guerriero et al. “Adoption, support, and challenges of infrastructure-as-code: Insights from industry”. In: *2019 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2019, pp. 580–589.
- [273] Antonio Carzaniga et al. *A characterization framework for software deployment technologies*. Tech. rep. Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, 1998.
- [274] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
- [275] Jeanderson Cândido, Maurício Aniche, and Arie van Deursen. “Log-based software monitoring: a systematic mapping study”. In: *PeerJ Computer Science* 7 (2021), e489.
- [276] Ane Blázquez-García et al. “A review on outlier/anomaly detection in time series data”. In: *ACM Computing Surveys (CSUR)* 54.3 (2021), pp. 1–33.
- [277] Tanja Hagemann and Katerina Katsarou. “A systematic review on anomaly detection for cloud computing environments”. In: *Proceedings of the 2020 3rd Artificial Intelligence and Cloud Computing Conference*. 2020, pp. 83–96.
- [278] E Chickowski. “Seven Winning DevSecOps Metrics Security Should Track”. In: *Bitdefender) Retrieved March 25* (2018), p. 2019.
- [279] Staffs Keele et al. *Guidelines for performing systematic literature reviews in software engineering*. 2007.

- 
- [280] Barbara Kitchenham, Lech Madeyski, and David Budgen. "SEGRESS: Software engineering guidelines for reporting secondary studies". In: *IEEE Transactions on Software Engineering* 49.3 (2022), pp. 1273–1298.
- [281] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. "Security as culture: a systematic literature review of DevSecOps". In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 2020, pp. 266–269.
- [282] Saima Rafi et al. "Prioritization based taxonomy of DevOps security challenges using PROMETHEE". In: *IEEE Access* 8 (2020), pp. 105426–105446.
- [283] A.W. Harzing. *Publish or Perish*. Available online at <https://harzing.com/resources/publish-or-perish>. 2007.
- [284] CORE. *International CORE Conference Rankings: ICORE*. <https://www.core.edu.au/icore-portal>. 2023.
- [285] Cody Watson et al. "A systematic literature review on the use of deep learning in software engineering research". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.2 (2022), pp. 1–58.
- [286] Lanxin Yang et al. "Quality assessment in systematic literature reviews: A software engineering perspective". In: *Information and Software Technology* 130 (2021), p. 106397.
- [287] Virginia Braun and Victoria Clarke. "Using thematic analysis in psychology". In: *Qualitative research in psychology* 3.2 (2006), pp. 77–101.
- [288] Alexandra Sbaraini et al. "How to do a grounded theory study: a worked example of a study of dental practices". In: *BMC medical research methodology* 11 (2011), pp. 1–10.
- [289] Maria Rosala. "How to analyze qualitative data from UX research: Thematic analysis". In: *NN-Nielsen Norman Group* (2019).
- [290] Microsoft. *Threat Modeling*. <https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling>. 2024.
- [291] Michael Howard and Steve Lipner. *The security development lifecycle*. Vol. 8. Microsoft Press Redmond, 2006.
- [292] David LeBlanc and Michael Howard. *Writing secure code*. Pearson Education, 2002.
- [293] Testsigma. *Impact Analysis In Software Testing-A Complete Overview*. <https://testsigma.com/blog/impact-analysis-in-testing/>. 2023.
- [294] Richard J Turver and Malcolm Munro. "An early impact analysis technique for software maintenance". In: *Journal of Software Maintenance: Research and Practice* 6.1 (1994), pp. 35–52.
- [295] Malcom Gethers et al. "Integrated impact analysis for managing software changes". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 430–440.
- [296] Hoa Khanh Dam et al. "Automatic feature learning for predicting vulnerable software components". In: *IEEE Transactions on Software Engineering* 47.1 (2018), pp. 67–85.
- [297] Laura Wartschinski et al. "VUDENC: vulnerability detection with deep learning on a natural codebase for Python". In: *Information and Software Technology* 144 (2022), p. 106809.
- [298] Wenjing Cai et al. "A software vulnerability detection method based on deep learning with complex network analysis and subgraph partition". In: *Information and Software Technology* 164 (2023), p. 107328.
- [299] Sicong Cao et al. "Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection". In: *Information and Software Technology* 136 (2021), p. 106576.
- [300] Yisroel Mirsky et al. "VulChecker: Graph-based Vulnerability Localization in Source Code". In: *31st USENIX Security Symposium, Security 2022*. 2023.

- 
- [301] Chunyong Zhang et al. “CPVD: Cross Project Vulnerability Detection Based On Graph Attention Network And Domain Adaptation”. In: *IEEE Transactions on Software Engineering* (2023).
- [302] Wenbo Wang et al. “DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 2249–2261.
- [303] Bozhi Wu et al. “Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-procedural Slicing”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 1371–1383.
- [304] Deqing Zou et al. “mVulPreter: A Multi-Granularity Vulnerability Detection System With Interpretations”. In: *IEEE Transactions on Dependable and Secure Computing* (2022).
- [305] Yukun Dong et al. “SedSVD: Statement-level software vulnerability detection based on Relational Graph Convolutional Network with subgraph embedding”. In: *Information and Software Technology* 158 (2023), p. 107168.
- [306] Bin Yuan et al. “Enhancing Deep Learning-based Vulnerability Detection by Building Behavior Graph Model”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 2262–2274.
- [307] Junwei Zhang et al. “Vulnerability Detection by Learning from Syntax-Based Execution Paths of Code”. In: *IEEE Transactions on Software Engineering* (2023).
- [308] Deqing Zou et al. “ $\mu$ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection”. In: *IEEE Transactions on Dependable and Secure Computing* 18.5 (2019), pp. 2224–2236.
- [309] Yukun Dong et al. “DeKeDVer: A deep learning-based multi-type software vulnerability classification framework using vulnerability description and source code”. In: *Information and Software Technology* (2023), p. 107290.
- [310] Xiangwei Li et al. “Prediction of Vulnerability Characteristics Based on Vulnerability Description and Prompt Learning”. In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2023, pp. 604–615.
- [311] Shengyi Pan et al. “Fine-grained commit-level vulnerability type prediction by CWE tree structure”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 957–969.
- [312] Mohammed Latif Siddiq et al. “Sqlifix: Learning based approach to fix sql injection vulnerabilities in source code”. In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2021, pp. 354–364.
- [313] Marjane Namavar, Noor Nashid, and Ali Mesbah. “A controlled experiment of different code representations for learning-based program repair”. In: *Empirical Software Engineering* 27.7 (2022), p. 190.
- [314] Zimin Chen, Steve Kommrusch, and Martin Monperrus. “Neural transfer learning for repairing security vulnerabilities in c code”. In: *IEEE Transactions on Software Engineering* 49.1 (2022), pp. 147–165.
- [315] Qihao Zhu et al. “A syntax-guided edit decoder for neural program repair”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 341–353.
- [316] Qihao Zhu et al. “Tare: Type-aware neural program repair”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 1443–1455.

- 
- [317] Sichong Hao et al. “Enhancing Code Language Models for Program Repair by Curricular Fine-tuning Framework”. In: *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2023, pp. 136–146.
- [318] Quanjun Zhang et al. “Pre-trained model-based automated software vulnerability repair: How far are we?” In: *IEEE Transactions on Dependable and Secure Computing* (2023).
- [319] Armin Zirak and Hadi Hemmati. “Improving automated program repair with domain adaptation”. In: *ACM Transactions on Software Engineering and Methodology* (2022).
- [320] Matthew Jin et al. “Inferfix: End-to-end program repair with llms”. In: *arXiv preprint arXiv:2303.07263* (2023).
- [321] Chao Ni et al. “Unifying Defect Prediction, Categorization, and Repair by Multi-Task Deep Learning”. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 1980–1992.
- [322] Yonghee Shin and Laurie Williams. “Can traditional fault prediction models be used for vulnerability prediction?” In: *Empirical Software Engineering* 18 (2013), pp. 25–59.
- [323] Riccardo Scandariato et al. “Predicting vulnerable software components via text mining”. In: *IEEE Transactions on Software Engineering* 40.10 (2014), pp. 993–1006.
- [324] Xiaoning Du et al. “Leopard: Identifying vulnerable code for vulnerability assessment through program metrics”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 60–71.
- [325] Xiao Cheng et al. “Deepwukong: Statically detecting software vulnerabilities using deep graph neural network”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30.3 (2021), pp. 1–33.
- [326] Yueming Wu et al. “Vulcnn: An image-inspired scalable vulnerability detection system”. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 2365–2376.
- [327] Nitesh V Chawla et al. “SMOTE: synthetic minority over-sampling technique”. In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.
- [328] Chengzhi Mao et al. “Metric learning for adversarial robustness”. In: *Advances in neural information processing systems* 32 (2019).
- [329] René Just, Dariush Jalali, and Michael 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, pp. 437–440.
- [330] Claire Le Goues et al. “The ManyBugs and IntroClass benchmarks for automated repair of C programs”. In: *IEEE Transactions on Software Engineering* 41.12 (2015), pp. 1236–1256.
- [331] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. “Discovering bug patterns in JavaScript”. In: *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 2016, pp. 144–156.
- [332] Derrick Lin et al. “QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge”. In: *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 2017, pp. 55–56.
- [333] Shin Hwei Tan et al. “Codeflaws: a programming competition benchmark for evaluating automated program repair tools”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 180–182.

- 
- [334] Ripon K Saha et al. “Bugs. jar: A large-scale, diverse dataset of real-world java bugs”. In: *Proceedings of the 15th international conference on mining software repositories*. 2018, pp. 10–13.
- [335] Michele Tufano et al. “An empirical study on learning bug-fixing patches in the wild via neural machine translation”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28.4 (2019), pp. 1–29.
- [336] Rafael-Michael Karampatsis and Charles Sutton. “How often do single-statement bugs occur? the manysstubs4j dataset”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 573–577.
- [337] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. “Angelix: Scalable multiline program patch synthesis via symbolic analysis”. In: *Proceedings of the 38th international conference on software engineering*. 2016, pp. 691–701.
- [338] Ming Wen et al. “Context-aware patch generation for better automated program repair”. In: *Proceedings of the 40th international conference on software engineering*. 2018, pp. 1–11.
- [339] Abigail See, Peter J Liu, and Christopher D Manning. “Get to the point: Summarization with pointer-generator networks”. In: *arXiv preprint arXiv:1704.04368* (2017).
- [340] Zimin Chen and Martin Monperrus. “The codrep machine learning on source code competition”. In: *arXiv preprint arXiv:1807.03200* (2018).
- [341] Serena Elisa Ponta et al. “A manually-curated dataset of fixes to vulnerabilities of open-source software”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 383–387.
- [342] Akshay Utture and Jens Palsberg. “From Leaks to Fixes: Automated Repairs for Resource Leak Warnings”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 159–171.
- [343] Diego Marcilio et al. “SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings”. In: *Journal of Systems and Software* 168 (2020), p. 110671.
- [344] Ibrahim Mesecan et al. “HyperGI: Automated detection and repair of information flow leakage”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 1358–1362.
- [345] Stuart Larsen. *MongoDB security team enables secure development with Snyk*. <https://snyk.io/case-studies/mongodb/>. 2024.
- [346] Taylor McCaslin. *How to put generative AI to work in your DevSecOps environment*. <https://about.gitlab.com/blog/2024/03/07/how-to-put-generative-ai-to-work-in-your-devsecops-environment/>. 2024.
- [347] GitLab. *European tech company Cube drives secure software with AI in GitLab Duo*. <https://about.gitlab.com/customers/cube/>. 2024.
- [348] Xiang Chen et al. “MULTI: Multi-objective effort-aware just-in-time software defect prediction”. In: *Information and Software Technology* 93 (2018), pp. 1–13.
- [349] Weiwei Li et al. “Effort-aware semi-supervised just-in-time defect prediction”. In: *Information and Software Technology* 126 (2020), p. 106364.
- [350] George G Cabral et al. “Class imbalance evolution and verification latency in just-in-time software defect prediction”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 666–676.

- 
- [351] Hailemehlekot Demtse Tessema and Surafel Lemma Abebe. "Enhancing just-in-time defect prediction using change request-based metrics". In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2021, pp. 511–515.
- [352] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. "Fine-grained just-in-time defect prediction". In: *Journal of Systems and Software* 150 (2019), pp. 22–36.
- [353] Meng Yan et al. "Just-in-time defect identification and localization: A two-phase framework". In: *IEEE Transactions on Software Engineering* 48.1 (2020), pp. 82–101.
- [354] Yan Sun et al. "Improving missing issue-commit link recovery using positive and unlabeled data". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 147–152.
- [355] Alexander Suh. "Adapting bug prediction models to predict reverted commits at Wayfair". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1251–1262.
- [356] Hang Ruan et al. "DeepLink: Recovering issue-commit links based on deep learning". In: *Journal of Systems and Software* 158 (2019), p. 110406.
- [357] Hoa Khanh Dam et al. "Lessons learned from using a deep tree-based model for software defect prediction in practice". In: *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*. IEEE. 2019, pp. 46–57.
- [358] Fangcheng Qiu et al. "Deep just-in-time defect localization". In: *IEEE Transactions on Software Engineering* 48.12 (2021), pp. 5068–5086.
- [359] Fangcheng Qiu et al. "JITO: a tool for just-in-time defect identification and localization". In: *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2020, pp. 1586–1590.
- [360] Sonu Mehta et al. "Rex: Preventing bugs and misconfiguration in large services using correlated change analysis". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 435–448.
- [361] Jinpeng Lan et al. "BTLink: automatic link recovery between issues and commits based on pre-trained BERT model". In: *Empirical Software Engineering* 28.4 (2023), p. 103.
- [362] Chenyuan Zhang et al. "EALink: An Efficient and Accurate Pre-trained Framework for Issue-Commit Link Recovery". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 217–229.
- [363] Ivan Pashchenko et al. "Vulnerable open source dependencies: Counting those that matter". In: *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*. 2018, pp. 1–10.
- [364] Sonatype. Sonatype. <https://www.sonatype.com/>. 2024.
- [365] Synopsys. Black Duck Software Composition Analysis. <https://www.synopsys.com/>. 2024.
- [366] Snyk. Software dependencies: How to manage dependencies at scale. <https://snyk.io/series/open-source-security/software-dependencies/>. 2022.
- [367] G Boetticher. "The PROMISE repository of empirical software engineering data". In: <http://promised.org/repository> (2007).
- [368] Yasutaka Kamei et al. "A large-scale empirical study of just-in-time quality assurance". In: *IEEE Transactions on Software Engineering* 39.6 (2012), pp. 757–773.
- [369] Shane McIntosh and Yasutaka Kamei. "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction". In: *Proceedings of the 40th international conference on software engineering*. 2018, pp. 560–560.

- 
- [370] RedHat. *What is CI/CD security?* <https://www.redhat.com/en/topics/security/what-is-cicd-security>. 2023.
- [371] Suraj Yatish et al. “Mining software defects: Should we consider affected releases?” In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 654–665.
- [372] Yunhua Zhao, Kostadin Damevski, and Hui Chen. “A systematic survey of just-in-time software defect prediction”. In: *ACM Computing Surveys* 55.10 (2023), pp. 1–35.
- [373] Jerome H Friedman and Bogdan E Popescu. “Predictive learning via rule ensembles”. In: (2008).
- [374] Rongxin Wu et al. “Relink: recovering links between bugs and changes”. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 15–25.
- [375] Guoli Cheng, Shi Ying, and Bingming Wang. “Tuning configuration of apache spark on public clouds by combining multi-objective optimization and performance prediction model”. In: *Journal of Systems and Software* 180 (2021), p. 111028.
- [376] Yuanjie Xia, Zishuo Ding, and Weiyi Shang. “CoMSA: A Modeling-Driven Sampling Approach for Configuration Performance Testing”. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 1352–1363.
- [377] Mubin Ul Haque, M Mehdi Kholoosi, and M Ali Babar. “KGSecConfig: A Knowledge Graph Based Approach for Secured Container Orchestrator Configuration”. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 420–431.
- [378] Huong Ha and Hongyu Zhang. “DeepPerf: Performance prediction for configurable software with deep sparse neural network”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1095–1106.
- [379] Yangyang Shu et al. “Perf-AL: Performance prediction for configurable software through adversarial learning”. In: *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2020, pp. 1–11.
- [380] Liang Bao et al. “ACTGAN: automatic configuration tuning for software systems with generative adversarial networks”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 465–476.
- [381] Akond Rahman and Laurie Williams. “Characterizing defective configuration scripts used for continuous deployment”. In: *2018 IEEE 11th International conference on software testing, verification and validation (ICST)*. IEEE. 2018, pp. 34–45.
- [382] Akond Rahman and Laurie Williams. “Source code properties of defective infrastructure as code scripts”. In: *Information and Software Technology* 112 (2019), pp. 148–163.
- [383] Stefano Dalla Palma et al. “Within-project defect prediction of infrastructure-as-code using product and process metrics”. In: *IEEE Transactions on Software Engineering* 48.6 (2021), pp. 2086–2104.
- [384] Nemanja Borovits et al. “FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code”. In: *Empirical Software Engineering* 27.7 (2022), p. 178.
- [385] Norbert Siegmund et al. “Performance-influence models for highly configurable systems”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 284–294.
- [386] Tianyin Xu and Yuanyuan Zhou. “Systems approaches to tackling configuration errors: A survey”. In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–41.

- 
- [387] Microsoft. *What is infrastructure as code (IaC)?* <https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code>. 2022.
- [388] Akond Rahman, Chris Parnin, and Laurie Williams. “The seven sins: Security smells in infrastructure as code scripts”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 164–175.
- [389] Wael Khreich et al. “An anomaly detection system based on variable N-gram features and one-class SVM”. In: *Information and Software Technology* 91 (2017), pp. 186–197.
- [390] Javier Alvarez Cid-Fuentes, Claudia Szabo, and Katrina Falkner. “Adaptive performance anomaly detection in distributed systems using online svms”. In: *IEEE Transactions on Dependable and Secure Computing* 17.5 (2018), pp. 928–941.
- [391] Shangbin Han et al. “Log-based anomaly detection with robust feature extraction and online learning”. In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 2300–2311.
- [392] Lin Yang et al. “Try with Simpler—An Evaluation of Improved Principal Component Analysis in Log-based Anomaly Detection”. In: *ACM Transactions on Software Engineering and Methodology* (2023).
- [393] Xingfang Wu, Heng Li, and Foutse Khomh. “On the effectiveness of log representation for log-based anomaly detection”. In: *Empirical Software Engineering* 28.6 (2023), p. 137.
- [394] Carla Sauvanaud et al. “Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned”. In: *Journal of Systems and Software* 139 (2018), pp. 84–106.
- [395] Min Du et al. “Deeplog: Anomaly detection and diagnosis from system logs through deep learning”. In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 1285–1298.
- [396] Xu Zhang et al. “Robust log-based anomaly detection on unstable log data”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 807–817.
- [397] Weibin Meng et al. “Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.” In: *IJCAI*. Vol. 19. 7. 2019, pp. 4739–4745.
- [398] Hudan Studiawan, Ferdous Sohel, and Christian Payne. “Anomaly detection in operating system logs with deep learning-based sentiment analysis”. In: *IEEE Transactions on Dependable and Secure Computing* 18.5 (2020), pp. 2136–2148.
- [399] Junwei Zhou et al. “DeepSyslog: Deep Anomaly Detection on Syslog Using Sentence Embedding and Metadata”. In: *IEEE Transactions on Information Forensics and Security* 17 (2022), pp. 3051–3061.
- [400] Xuheng Wang et al. “LogOnline: A Semi-Supervised Log-Based Anomaly Detector Aided with Online Learning Mechanism”. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 141–152.
- [401] Min Du et al. “Lifelong anomaly detection through unlearning”. In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 1283–1297.
- [402] Xiaoyun Li et al. “SwissLog: Robust anomaly detection and localization for interleaved unstructured logs”. In: *IEEE Transactions on Dependable and Secure Computing* (2022).
- [403] Lun-Pin Yuan, Peng Liu, and Sencun Zhu. “Recompose event sequences vs. predict next events: A novel anomaly detection approach for discrete event logs”. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 2021, pp. 336–348.

- 
- [404] Jiangming Li et al. "LogGraph: Log Event Graph Learning Aided Robust Fine-Grained Anomaly Diagnosis". In: *IEEE Transactions on Dependable and Secure Computing* (2023).
- [405] Van-Hoang Le and Hongyu Zhang. "Log-based anomaly detection without log parsing". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 492–504.
- [406] Dongqi Han et al. "Deepaid: Interpreting and improving deep learning-based anomaly detection in security applications". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 3197–3217.
- [407] Diana Laura Aguilar et al. "Towards an interpretable autoencoder: A decision-tree-based autoencoder and its application in anomaly detection". In: *IEEE transactions on dependable and secure computing* 20.2 (2022), pp. 1048–1059.
- [408] Cheryl Lee et al. "Maat: Performance Metric Anomaly Anticipation for Cloud Services with Conditional Diffusion". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 116–128.
- [409] Liang Xi et al. "Adaptive-Correlation-aware Unsupervised Deep Learning for Anomaly Detection in Cyber-physical Systems". In: *IEEE Transactions on Dependable and Secure Computing* (2023).
- [410] Qin Lin et al. "TABOR: A graphical model-based approach for anomaly detection in industrial control systems". In: *Proceedings of the 2018 on asia conference on computer and communications security*. 2018, pp. 525–536.
- [411] Qinghua Xu, Shaukat Ali, and Tao Yue. "Digital Twin-based Anomaly Detection with Curriculum Learning in Cyber-physical Systems". In: *ACM Transactions on Software Engineering and Methodology* (2023).
- [412] Qinghua Xu, Shaukat Ali, and Tao Yue. "Digital twin-based anomaly detection in cyber-physical systems". In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2021, pp. 205–216.
- [413] Zhe Xie et al. "From Point-wise to Group-wise: A Fast and Accurate Microservice Trace Anomaly Detection Approach". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 1739–1749.
- [414] Jun Huang et al. "Twin Graph-Based Anomaly Detection via Attentive Multi-Modal Learning for Microservice System". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 66–78.
- [415] Qinghua Xu et al. "KDDT: Knowledge Distillation-Empowered Digital Twin for Anomaly Detection". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 1867–1878.
- [416] Yahoo. *Yahoo! Webscope Dataset*. <https://webscope.sandbox.yahoo.com/>. 2024.
- [417] Adam Oliner and Jon Stearley. "What supercomputers say: A study of five system logs". In: *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*. IEEE. 2007, pp. 575–584.
- [418] Wei Xu et al. "Detecting large-scale system problems by mining console logs". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 117–132.
- [419] Gideon Creech and Jiankun Hu. "Generation of a new IDS test dataset: Time to retire the KDD collection". In: *2013 IEEE wireless communications and networking conference (WCNC)*. IEEE. 2013, pp. 4487–4492.

- 
- [420] Javier Alvarez Cid-Fuentes, Claudia Szabo, and Katrina Falkner. “Online behavior identification in distributed systems”. In: *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2015, pp. 202–211.
- [421] Nour Moustafa and Jill Slay. “UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)”. In: *2015 military communications and information systems conference (MilCIS)*. IEEE. 2015, pp. 1–6.
- [422] Shilin He et al. “Loghub: A large collection of system log datasets towards automated log analytics”. In: *arXiv e-prints* (2020), arXiv–2008.
- [423] Jon Stearley and Adam J Oliner. “Bad words: Finding faults in spirit’s syslogs”. In: *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE. 2008, pp. 765–770.
- [424] Yahoo Research. *A Benchmark Dataset for Time Series Anomaly Detection*. <https://yahooresearch.tumblr.com/post/114590420346/a-benchmark-dataset-for-time-series-anomaly>. 2015.
- [425] Jonathan Ho, Ajay Jain, and Pieter Abbeel. “Denoising diffusion probabilistic models”. In: *Advances in neural information processing systems* 33 (2020), pp. 6840–6851.
- [426] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. “Isolation-based anomaly detection”. In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6.1 (2012), pp. 1–39.
- [427] Thomas H Morris, Zach Thornton, and Ian Turnipseed. “Industrial control system simulation and data logging for intrusion detection system research”. In: *7th annual southeastern cyber security summit* (2015), pp. 3–4.
- [428] Aditya P Mathur and Nils Ole Tippenhauer. “SWaT: A water treatment testbed for research and training on ICS security”. In: *2016 international workshop on cyber-physical systems for smart water networks (CySWater)*. IEEE. 2016, pp. 31–36.
- [429] Chuadhry Mujeeb Ahmed, Venkata Reddy Palleti, and Aditya P Mathur. “WADI: a water distribution testbed for research in the design of secure cyber physical systems”. In: *Proceedings of the 3rd international workshop on cyber-physical systems for smart water networks*. 2017, pp. 25–28.
- [430] Riccardo Taormina et al. “Battle of the attack detection algorithms: Disclosing cyber attacks on water distribution networks”. In: *Journal of Water Resources Planning and Management* 144.8 (2018), p. 04018048.
- [431] Tianzhu Zhang et al. “System log parsing: A survey”. In: *IEEE Transactions on Knowledge and Data Engineering* 35.8 (2023), pp. 8596–8614.
- [432] NSF. *Cyber-physical systems*. [https://www.nsf.gov/news/special\\_reports/cyber-physical/](https://www.nsf.gov/news/special_reports/cyber-physical/). 2024.
- [433] Abdulmotaleb El Saddik. “Digital twins: The convergence of multimedia technologies”. In: *IEEE multimedia* 25.2 (2018), pp. 87–92.
- [434] Yuequan Bao et al. “Computer vision and deep learning-based data anomaly detection method for structural health monitoring”. In: *Structural Health Monitoring* 18.2 (2019), pp. 401–421.
- [435] Thomas Schlegl et al. “Unsupervised anomaly detection with generative adversarial networks to guide marker discovery”. In: *International conference on information processing in medical imaging*. Springer. 2017, pp. 146–157.
- [436] Xu Yang et al. “Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!” In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 2287–2298.

- 
- [437] Emily Rowan Winter et al. “Towards developer-centered automatic program repair: findings from Bloomberg”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 1578–1588.
- [438] Agnar Aamodt and Enric Plaza. “Case-based reasoning: Foundational issues, methodological variations, and system approaches”. In: *AI communications* 7.1 (1994), pp. 39–59.
- [439] Alicja Gosiewska and Przemyslaw Biecek. “IBreakDown: Uncertainty of model explanations for non-additive predictive models”. In: *arXiv preprint arXiv:1903.11420* (2019).
- [440] Jirayus Jiarpakdee et al. “An empirical study of model-agnostic techniques for defect prediction models”. In: *IEEE Transactions on Software Engineering* 48.1 (2020), pp. 166–185.
- [441] Vy Vo et al. “An additive instance-wise approach to multi-class model interpretation”. In: *arXiv preprint arXiv:2207.03113* (2022).
- [442] Kun Kuang et al. “Causal inference”. In: *Engineering* 6.3 (2020), pp. 253–263.
- [443] Pierre Tempel and Eric Tooley. *Found means fixed: Introducing code scanning autofix, powered by GitHub Copilot and CodeQL*. <https://github.blog/2024-03-20-found-means-fixed-introducing-code-scanning-autofix-powered-by-github-copilot-and-codeql/>. 2024.
- [444] GitHub. *About autofix for CodeQL code scanning*. <https://docs.github.com/en/code-security/code-scanning/managing-code-scanning-alerts/about-autofix-for-codeql-code-scanning#supported-languages>. 2024.
- [445] Aristiun. *Automated Threat Modeling using AI*. <https://www.aristiun.com/automated-threat-modeling-using-ai>. 2024.
- [446] Validata. *AI-powered Live Impact Analysis*. <https://www.validata-software.com/products/validata-sense-ai/ai-powered-live-impact-analysis>. 2024.
- [447] Microsoft. *Microsoft Copilot for Security is generally available on April 1, 2024, with new capabilities*. <https://www.microsoft.com/en-us/security/blog/2024/03/13/microsoft-copilot-for-security-is-generally-available-on-april-1-2024-with-new-capabilities/>. 2024.
- [448] Benjamin G Edelman et al. “Randomized Controlled Trial for Microsoft Security Copilot”. In: *Available at SSRN* 4648700 (2023).
- [449] Benjamin Steenhoek et al. “An empirical study of deep learning models for vulnerability detection”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 2237–2248.
- [450] Shigang Liu et al. “CD-VULD: Cross-domain vulnerability discovery based on deep domain adaptation”. In: *IEEE Transactions on Dependable and Secure Computing* 19.1 (2020), pp. 438–451.
- [451] Adriana Sejfia et al. “Toward Improved Deep Learning-based Vulnerability Detection”. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society. 2023, pp. 730–741.
- [452] Kai Huang et al. “An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair”. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 1162–1174.
- [453] Iz Beltagy, Matthew E Peters, and Arman Cohan. “Longformer: The long-document transformer”. In: *arXiv preprint arXiv:2004.05150* (2020).

- 
- [454] Yutao Sun et al. *Retentive Network: A Successor to Transformer for Large Language Models*. 2023. arXiv: [2307.08621 \[cs.CL\]](https://arxiv.org/abs/2307.08621).
- [455] Chenze Shao and Yang Feng. “Overcoming catastrophic forgetting beyond continual learning: Balanced training for neural machine translation”. In: *arXiv preprint arXiv:2203.03910* (2022).
- [456] Chunqiu Steven Xia and Lingming Zhang. “Less training, more repairing please: revisiting automated program repair via zero-shot learning”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 959–971.
- [457] Susan Hall. *AI-Assisted Dependency Updates without Breaking Things*. <https://thenewstack.io/ai-assisted-dependency-updates-without-breaking-things/>. 2024.
- [458] Josh Achiam et al. “Gpt-4 technical report”. In: *arXiv preprint arXiv:2303.08774* (2023).
- [459] Baptiste Roziere et al. “Code llama: Open foundation models for code”. In: *arXiv preprint arXiv:2308.12950* (2023).
- [460] Yassine Ouali, Céline Hudelot, and Myriam Tami. “An overview of deep semi-supervised learning”. In: *arXiv preprint arXiv:2006.05278* (2020).
- [461] Dongqi Han et al. “Anomaly Detection in the Open World: Normality Shift Detection, Explanation, and Adaptation”. In: *30th Annual Network and Distributed System Security Symposium (NDSS)*. 2023.
- [462] Limin Yang et al. “{CADE}: Detecting and explaining concept drift samples for security applications”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2327–2344.
- [463] Guolin Lyu and Robert W Brennan. “Multi-agent modelling of cyber-physical systems for IEC 61499-based distributed intelligent automation”. In: *International Journal of Computer Integrated Manufacturing* (2023), pp. 1–27.
- [464] Mariam Ibrahim and Ruba Elhafiz. “Security analysis of cyber-physical systems using reinforcement learning”. In: *Sensors* 23.3 (2023), p. 1634.