Understanding and improving deep learning models for vulnerability detection

by

Benjamin Jeremiah Steenhoek

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Wei Le, Major Professor
Hongyang Gao
Myra Beth Cohen
Qi Li
Samik Basu

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2024

DEDICATION

This dissertation is dedicated to my family, who have supported me in every way possible.

To my wife and best friend, Cierrah, whose unwavering love, encouragement, and belief in me have become my foundation. You have been my constant support and companion. Thank you for your patience and sacrifices, and for standing with me through every challenge and blessing.

To my dad, Loren Steenhoek, for inspiring me to pursue higher education and always being proud of me; to my mom, Younghee Steenhoek, for giving me strength; to my brother, AJ Steenhoek, for going side-by-side with me throughout my schooling, from the crib until now; and to my late 할머니, 주 진찬, for cheering me on through all of it.

I love you, thank you, and would never have completed this journey without you.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

I would like to take this opportunity to express my heartfelt gratitude to those who helped me throughout the research and writing of this dissertation.

First, I extend my deepest thanks to my advisor, Dr. Wei Le, for her advice, patience, encouragement, and critiques. I am truly grateful to have an advisor who treats me as a peer and looks out for my best interests. Thank you, Dr. Le, for the time and effort you have spent for me. I also thank our faculty collaborators, Drs. Baishakhi Ray and Earl Barr, and my committee members, Drs. Hongyang Gao, Myra Cohen, Qi Li, and Samik Basu, for their thought-provoking questions and research discussions.

My sincere thanks to my internship mentors at Microsoft, Drs. Roshanak Zilouchian Moghaddam, Michele Tufano, and Alexey Svyatkovskiy, for being supportive, engaged, and inspiring leaders; for setting an example of scientific excellence; and for treating me with kindness while encouraging me to push my limits.

Thank you to all of my colleagues, especially Md Mahbubur Rahman and Yangruibo (Robin) Ding, for being dependable and pleasant co-authors; I've enjoyed every moment of working together with you. A special thank-you to Yaojie (Jason) Hu for his encouragement to scientific rigor, advice on writing and experiments, steadfast support, and many enjoyable evenings spent workshopping research ideas.

Thank you to the many people who helped me both directly and indirectly. I have learned from and been helped by many researchers, colleagues, and friends in the course of graduate school. I also want to also offer my appreciation to those who were willing to participate in my surveys and observations, without whom my work would not have been possible.

Your collective support has been essential to the completion of this achievement. For that, I am deeply grateful.

This research was partially supported by the U.S. National Science Foundation (NSF) under Awards #1816352 and #2313054, and partially performed during an internship at Microsoft. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of the U.S. Government, NSF, or Microsoft.

ABSTRACT

Vulnerability detection tools are essential for ensuring security while maintaining software development velocity. Deep Learning (DL) has shown potential in this domain, often surpassing static analyzers on certain open-source datasets. However, current DL-based vulnerability detection systems are limited, resulting in models that are poorly understood, inefficient, and struggle to generalize, and their applicability in practical applications is not well understood. In this dissertation, we comprehensively evaluate state-of-the-art (SOTA) DL vulnerability detection models, including Graph Neural Networks (GNNs), fine-tuned transformer models, and Large Language Models (LLMs), yielding a deeper understanding of their benefits and limitations and a body of approaches for improving DL for vulnerability detection using static and dynamic analysis.

First, we empirically study the model capabilities, training data, and model interpretation of fine-tuned graph neural networks and transformer models and provide guidance on understanding model results, preparing training data, and improving the robustness of the models. We found that state-of-the-art models were limited in their ability to leverage vulnerability semantics, which are critical aspects of vulnerability detection.

Building on these findings, we developed DeepDFA and TRACED, which integrate static and dynamic analysis into DL model architecture and training. DeepDFA is a GNN architecture and learning approach inspired by dataflow analysis. DeepDFA outperforms several other state-of-the-art models, is efficient in terms of computational resources and training data, and generalizes to novel software applications better than other SOTA approaches. TRACED is a transformer model which is pre-trained on a combination of source code, executable inputs, and execution traces. TRACED improves upon statically pre-trained code models on predicting

program coverage and variable values, and outperforms statically pre-trained models in two downstream tasks: code clone retrieval and vulnerability detection.

Additionally, we evaluate large language models (LLMs) for vulnerability detection using SOTA prompting techniques. We find their performance is hindered by failures to localize and understand individual statements, and logically arrive at a conclusion, and suggest directions for improvement in these areas.

Finally, we introduce DeepVulGuard, an IDE-integrated tool based on DL models for vulnerability detection and fixing. Through a real-world user study with professional developers, we identify promising aspects of in-IDE DL integration, along with critical issues such as high false-positive rates and non-applicable fixes that must be addressed for practical deployment.

CHAPTER 1. GENERAL INTRODUCTION

Static analysis has become a critical component of software developer workflows, intended to facilitate rapid growth and development while preventing bugs and increasing security [1, 9, 11, 4]. Deep Learning models have demonstrated substantial improvements in performance on the task of vulnerability detection, even outperforming static analyzers on open-source vulnerability datasets [10, 8, 2]. This advent of high-performing models enables new applications for developers to use these models as static analysis tools to detect vulnerabilities during development.

However, there is more to the usage of these models than performance metrics measured on singular benchmarks. We must understand in which situations these models work, how they perform in realistic scenarios, what their limitations are, and how to overcome these limitations. In order to enable practical deployment, it's also important to maintain their efficiency so that they can scale to widespread use, including use on consumer hardware. The key problem which we study in this dissertation is: How can we utilize deep learning models to effectively detect security vulnerabilities in the real world?

To this end, we empirically studied state-of-the-art deep learning models, including graph neural networks, fine-tuned transformers, and large language models, on a wide variety of datasets. We found that the models often failed because they lacked knowledge of *vulnerability semantics*, as a result of training on purely textual data. Based on the findings of our empirical study, we designed approaches for integrating static and dynamic analysis into deep learning models: DeepDFA and TRACED. We show that both approaches successfully improved model performance, with DeepDFA showing greater generalization and efficiency.

Beyond evaluations on offline vulnerability datasets, deep learning models have not been widely applied in software development. We deployed state-of-the-art detection and fixing models

in an IDE-integrated application and studied its usefulness with professional software developers in real-world usage scenarios.

1.1 Contributions

This dissertation represents a step forward in understanding and improving the capabilities and applicability of deep learning-based vulnerability detection models. We make the following contributions:

Empirical study of deep learning models: At the time of writing, several papers have proposed new deep learning models based on technical improvements to model architectures, most notably fine-tuned Graph Neural Networks (GNNs) and fine-tuned transformer-based "Small" Language Models (SLMs) and Large Language Models (LLMs). However, beyond comparing with baseline models on benchmarks (which have several limitations, demonstrated in preceding and contemporary studies [5, 3, 6]), little was understood about how the models would perform in more realistic scenarios. We comprehensively evaluated fine-tuned SLMs in Chapter 2 and LLMs in Chapter 5 in order to understand in which settings the state-of-the-art models performed best, where they failed, and what could be done to improve them. Based on our results, we provided concrete recommendations which directly drive model improvements in Chapters 3 and 4.

Integration of static and dynamic analysis with deep learning models: DeepDFA (Chapter 3) represents the first integration of static dataflow analysis with deep learning models for vulnerability detection. We show that integrating dataflow analysis allowed DeepDFA to perform more effectively and efficiently than other state-of-the-art models, as well as generalize to real-world vulnerabilities in a novel dataset. TRACED (Chapter 4) represents the first application of dynamic execution-aware fine-tuning to deep learning models for vulnerability detection. We show that execution-awareness allowed TRACED to outperform other vulnerability detection and code clone detection models, as well as identify execution-based information about source code more accurately than prior pre-trained models which were trained only on static source code. The

integration of static and dynamic analysis techniques introduces a new paradigm for model architectures, which allows models to make more precise and nuanced predictions.

User-facing implementation and user study: Beyond benchmark evaluations, the instantiation and deployment of a deep learning-based user-facing tool presents many practical challenges, such as delivering model predictions with low latency, presenting fixes in an actionable way, and dealing with false positives. We implemented DeepVulGuard, which combined state-of-the-art SLMs and LLMs to surface vulnerability detection alerts in an IDE, and leveraged LLMs to suggest fixes for the vulnerabilities. We conducted an empirical user study, providing the first view of vulnerability detection + fixing models in a real-world setting, and report novel findings which reflect on the models' benefits, effective performance, and pain points. We show that, although current deep learning models are not yet ready for deployment, they bear great promise, and lay out concrete recommendations for further development of this technology for real-world applications.

The majority of this dissertation is adapted from peer-reviewed work published in top-tier software engineering conferences. Chapter 2 is based on a paper [13] published at the International Conference on Software Engineering (ICSE 2023). Chapters 3 and 4 are based on papers [12, 7] both published at the International Conference on Software Engineering (ICSE 2024). Chapter 5 is based on a paper submitted to the International Conference on Learning Representations (ICLR 2025). Chapter 6 is based on a paper published at the International Conference on Software Engineering (ICSE 2025).

1.2 Outline

The dissertation is organized as follows: Chapter 2 presents our empirical study, studying and providing recommendations for DL model capabilities, training data, and model interpretation. Chapter 3 introduces our proposed vulnerability detection model, DeepDFA, a GNN inspired by dataflow analysis. Chapter 4 introduces TRACED, our proposed method for execution-aware

language model pretraining. Chapter 5 presents our comprehensive empirical study of LLM performance and reasoning errors for vulnerability detection. Chapter 6 presents our user study on DeepVulGuard, an IDE-integrated deployment of DL-based vulnerability detection and fix models. Chapter 7 concludes the dissertation.

1.3 Bibliography

[1] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM 53*, 2 (2010), 66–75.

[2] Cao, S., Sun, X., Bo, L., Wu, R., Li, B., and Tao, C. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. *arXiv:2203.02660* (Mar 2022). arXiv: 2203.02660.

[3] Chen, Y., Ding, Z., Alowain, L., Chen, X., and Wagner, D. 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* (New York, NY, USA, 2023), RAID '23, Association for Computing Machinery, p. 654–668.

[4] Cook, B. Formal reasoning about the security of amazon web services. In *Computer Aided Verification: 30th International Conference* (2018), CAV '18, Springer, pp. 38–47.

[5] Croft, R., Babar, M. A., and Kholoosi, M. M. Data quality for software vulnerability datasets. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 121–133.

[6] Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., David Wagner, B. R., and Chen, Y. Vulnerability detection with code language models: How far are we? In *Proceedings of the 47th International Conference on Software Engineering* (2025), ICSE '25.

[7] Ding, Y., Steenhoek, B., Pei, K., Kaiser, G., Le, W., and Ray, B. TRACED: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (2024), ICSE '24, pp. 1–12.

[8] Ding, Y., Suneja, S., Zheng, Y., Laredo, J., Morari, A., Kaiser, G., and Ray, B. VELVET: a novel ensemble learning approach to automatically locate vulnerable statements. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering* (Los Alamitos, CA, USA, Mar 2022), SANER '22, IEEE Computer Society, pp. 959–970.

[9] Distefano, D., Fähndrich, M., Logozzo, F., and O'Hearn, P. W. Scaling static analyses at facebook. *Communications of the ACM 62*, 8 (2019), 62–70.

[10] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the Network and Distributed System Security Symposium* (2018), NDSS '18, Internet Society.

[11] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at Google. *Communications of the ACM 61*, 4 (2018), 58–66.

[12] Steenhoek, B., Gao, H., and Le, W. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[13] Steenhoek, B., Rahman, M. M., Jiles, R., and Le, W. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 2237–2248.

CHAPTER 2. AN EMPIRICAL STUDY OF DEEP LEARNING MODELS FOR VULNERABILITY DETECTION

Benjamin Steenhoek[1], Md Mahbubur Rahman[2], Richard Jiles[3], and Wei Le[4]

[1-4] Department of Computer Science, Iowa State University, Ames, IA, 50011

Modified from a manuscript published in the *45th International Conference on Software Engineering (ICSE 2023)*

Abstract

Deep learning (DL) models of code have recently reported great progress for vulnerability detection. In some cases, DL-based models have outperformed static analysis tools. Although many great models have been proposed, we do not yet have a good understanding of these models. This limits the further advancement of model robustness, debugging, and deployment for the vulnerability detection. In this chapter, we surveyed and reproduced 9 state-of-the-art (SOTA) deep learning models on 2 widely used vulnerability detection datasets: Devign and MSR. We investigated 6 research questions in three areas, namely *model capabilities*, *training data*, and *model interpretation*. We experimentally demonstrated the variability between different runs of a model and the low agreement among different models' outputs. We investigated models trained for specific types of vulnerabilities compared to a model that is trained on all the vulnerabilities at once. We explored the types of programs DL may consider "hard" to handle. We investigated the relations of training data sizes and training data composition with model performance. Finally, we studied model interpretations and analyzed important features that the models used to make predictions. We believe that our findings can help better understand model results, provide guidance on preparing training data, and improve the robustness of the models. All of our datasets, code, and results are available at https://doi.org/10.6084/m9.figshare.20791240.

2.1 Introduction

Deep learning vulnerability detection tools have achieved promising results in recent years. The state-of-the-art (SOTA) models reported 0.9 F1 score [14, 34] and outperformed static analyzers [11, 5]. The results are exciting in that deep learning may bring in transformative changes for software assurance. Thus, industry companies such as IBM, Google and Amazon are very interested and have invested heavily to develop such tools and datasets [26, 31, 19, 45].

Although promising, deep learning vulnerability detection has not yet reached the level of computer vision and natural language processing. Most of our research focuses on trying a new emerging deep learning model and making it work for a dataset like the Devign or MSR dataset [12, 46, 26]. However, we know little about the model itself, e.g., what type of programs the model can/cannot handle well, whether we should build models for each vulnerability type or we should build one model for all vulnerability types, what is a good training dataset, and what information the model has used to make the decisions. Knowing the answers to these questions can help us better develop, debug, and apply the models in practice. But considering the black-box nature of deep learning, these questions are very hard to answer. This chapter does not mean to provide a complete solution for these questions but is an exploration towards these goals.

In this chapter, we surveyed and reproduced a collection of SOTA deep learning vulnerability detection models, and constructed research questions and studies to understand these models, with the goal of distilling lessons and guidelines for better designing and debugging future models. To the best of our knowledge, this is the first paper that systematically investigated and compared a variety of SOTA deep learning models. In the past, Chakraborty et al. [6] have explored four existing models such as VulDeePecker [23], SySeVR [22] and Devign [46] and pointed out that the models trained with synthetic data reported low accuracies on real-world test set, and the models used spurious features like variable names to make the predictions.

We constructed our research questions and classified them into three areas, namely *model capabilities*, *training data*, and *model interpretation*. Specifically, our first goal is to understand

the capabilities of deep learning for handling vulnerability detection problems, especially regarding the following research questions:

- **RQ1** Do models agree on the vulnerability detection results? What are the variabilities across different runs of a model and across different models?
- **RQ2** Are certain types of vulnerabilities easier to detect? Should we build models for each type of vulnerabilities or should we build one model that can detect all the vulnerabilities?
- **RQ3** Are programs with certain code features harder to be predicted correctly by current models, and if so, what are those code features?

Our second study focuses on training data. We aim to understand whether and how the training data size and project composition can affect the model performance. Specifically, we constructed the following research questions:

- **RQ4** Can increasing the dataset size help improve the model performance for vulnerability detection?
- **RQ5** How does the project composition in the training dataset affect the performance of the models?

Finally, our third investigation area is model intepretation. We used SOTA model explanation tools to investigate:

- **RQ6** What source code information the models used for prediction? Do the models agree on the important features?

To answer the research questions, we surveyed the SOTA deep learning models and successfully reproduced 11 models on their original datasets (see Section 2.2). These models used different deep learning architectures such as GNN, RNN, LSTM, CNN, and Transformers. To compare the models, we managed to make 9 models work with the Devign and MSR, two popular

datasets. We selected the two datasets because (1) both of the datasets contain real-world projects and vulnerabilities; (2) the majority of models are evaluated and tuned with the Devign dataset in their papers; and (3) the MSR dataset contains 310 projects and its data have annotations on vulnerability types, which are needed to study our RQs. We discovered the findings for our 6 RQs with carefully designed experiments (Section 2.3) and considerations of the threats (Section 2.4). In summary, our research contributions include:

1. We conducted a comprehensive survey for the deep learning vulnerability detection models.
2. We delivered a reproduction package, consisting of the trained models and datasets for 11 SOTA deep learning frameworks with various study settings;
3. We designed 6 RQs to understand model capabilities, training data and model interpretation;
4. We constructed the studies and experimentally obtained the results for the RQs; and
5. We prepared interesting examples and data for further studying model interpretability.

2.2 A Survey of Models and their Reproduction

To collect the SOTA deep learning models, we studied the papers from 2018 to 2022 and also used Microsoft's CodeXGLUE leaderboard [1] and IBM's Defect detection D2A leaderboard [2]. We worked with all the open-source models we can find, and successfully reproduced 11 models. The complete list of models and the reasons we failed to reproduce some models are given in our data replication package.

As shown in Table 2.1, the reproduced models cover a variety of deep learning architectures. Devign [46] and ReVeal [6] used GNN on *property graphs* [46] that integrate control flow, data dependencies and AST. ReGVD[29] used GNN on tokens. Code2Vec used *multilayer perceptron (MLP)* on AST. VulDeeLocator [21] and SySeVR [22] are based the sequence models of RNN and

[1] https://microsoft.github.io/CodeXGLUE
[2] https://ibm.github.io/D2A

Table 2.1: 11 Reproduced Models.

Model	**Year**	**Architecture**	**Dataset**
Devign [46]	2019	GNN, property graph	Devign
ReVeal [6]	2021	GNN, property graph	Devign, ReVeal
ReGVD [29]	2022	GNN, token	Devign
CodeBERT [13]	2020	Transformer	Devign
VulBERTa-CNN [15]	2021	Transformer, CNN	VulDeePecker, Draper,
VulBERTa-MLP [15]	2021	Transformer, MLP	ReVeal μVulDeePecker,
PLBART [2]	2021	Transformer	Devign, D2A
LineVul [14]	2022	Transformer	MSR
Code2Vec [8]	2021	MLP, AST	Devign
SeSyVR [22]	2018	RNN	SARD, NVD
VulDeeLocator [21]	2020	Bi-LSTM	SARD, NVD

Bi-LSTMs. Recent deep learning detection used pre-trained transformers, including CodeBERT [13], VulBERTa-CNN [15], VulBERTa-MLP, PLBART [2] and LineVul [14]

For our RQs, we used the Devign [46] and MSR [12] datasets. We studied the datasets used in these 11 models in their original papers, shown under *Dataset* in Table 2.1. We found that the Devign dataset has been evaluated and tuned in 8 out of 11 models. It is a *balanced* dataset consisting of roughly the same number of vulnerable and non-vulnerable examples, a total of 27,318 data points (each example is also called an data point). LineVul worked with the MSR dataset, which is a more recently available dataset. It is an *imbalanced dataset*, consisting of 10,900 vulnerable examples and 177,736 non-vulnerable examples. These examples are labeled with their source projects and their Common Weakness Enumeration entries (CWE), which indicates the types of the vulnerabilities. We leverage such traits of the datasets for some of our RQs.

Table 2.2: Model reproduction on their original datasets. Reproduction results are reported as the mean of 3 random seeds. '-' indicates that the results for a metric were not reported in their papers. The numbers are in percentage.

	Paper Results				Our Reproduction			
Model	**Acc**	**Precision**	**Recall**	**F1**	**Acc'**	**Precision'**	**Recall'**	**F1'**
Devign	59	54	63	57	56	50	71	59
ReVeal	63	57	75	64	53	48	71	56
ReGVD	63	-	-	-	62	62	46	52
CodeBERT	62	-	-	-	64	59	54	55
VulBERTa-CNN	64	-	-	-	64	60	59	59
VulBERTa-MLP	65	-	-	-	63	60	58	59
PLBART	63	-	-	-	62	58	59	59
LineVul	-	97	86	91	99	96	88	92
Code2Vec	62	-	-	-	59	55	58	57
SeSyVR	98	90	92	90	94	88	84	86
VulDeeLocator	99	98	-	97	98	99	96	98

We reproduced the results for the models using their original datasets and settings, shown in Table 2.2. The columns of A, P, R and F represents the commonly used metrics in deep learning vulnerability detection, including *accuracy*, *precision*, *recall* and *F1*. Our reproduction results are able to compute within 2% difference compared with the original papers in general. The exceptions are ReVeal, for which the authors confirmed that our results fixed a data leakage error in the original paper, and Devign, for which we used the third-party[3] reproduction released by Chakraborthy et al. [6], as the original Devign code is not open-sourced.

To enable the comparisons of the models, we improved the models' implementations to support both Devign and MSR datasets. When running experiments for the RQs, we excluded *VulDeeLocator* and *SeSyVR* as they cannot be easily modified for the Devign and MSR datasets. As a result, we used the rest 9 models for our studies of the RQs.

[3] https://github.com/saikat107/Devign

2.3 Research Questions and Findings

We organized the research questions into three areas, namely *model capabilities*, *training data*, and *model interpretation*. See Sections 2.3.1 to 2.3.3 respectively. For each RQ, we present the motivation, study setup and our findings.

2.3.1 Capabilities of Deep Learning Models

RQ1 Do models agree on the vulnerability detection results? What are the variabilities across different runs of a model and across different models?

Motivation: It is known that deep learning model performance can vary across training runs when using different random seeds. In this RQ, we aim to measure for vulnerability detection, how much such variability actually exists. Additionally, we want to discover how much agreement exists across different deep learning models and across the models with similar architectures. We hope our findings can inform developers and researchers of the uncertainty which potentially exists behind the numbers reported by such tools.

Study Setup: We trained the models using 3 different random seeds on the same train/valid/test partitions of the Devign dataset. We used this dataset because almost all the models tuned their hyperparameters on it. We measured the percentage of *stable* inputs—-an input that has the same binary label for all 3 random seeds. We then compared the stable inputs across the models to measure their agreement.

Findings: In Table 2.3, we reported the percentage of stable inputs for the entire dataset, under *Total stability*, and for the test dataset, under *Test stability*. We also reported the variations of the F1 score across 3 seeds (on test dataset) under σ *Test F1*.

Our results show that on average 34.9% test data (30.6% total data) reported different predictions dependent on the seeds used in training. The GNN models that work on property graph ranked the top 2 variability; especially for ReVeal, for 50% of the test data, its outputs changed between runs. Code2Vec reported the least variability compared to the GNN and transformer models. Interestingly, we found that unstable inputs are associated with more

Table 2.3: Variability over 3 random seeds on Devign dataset.

Model	**Total stability**	**Test stability**	**σ Test F1**
ReVeal	55%	50%	2.73
Devign	57%	55%	2.24
VulBERTa-MLP	60%	58%	3.13
PLBART	72%	67%	1.03
LineVul	72%	67%	3.46
CodeBERT	72%	69%	2.78
VulBERTa-CNN	74%	71%	2.60
ReGVD	74%	72%	7.33
Code2Vec	89%	77%	0.78

incorrect predictions — stable inputs had a total of 19% incorrect predictions across all seeds and unstable had 47%.

Although many examples reported different predictions between runs, we found that F1 test scores did not change as much, and had a standard deviation of 2.9 on average. That said, for most models, we expect 95% of performance measurements to be within a range of 5.8% above or below the mean performance when measured on multiple random seeds.

Table 2.4: Agreement across different models.

Model	**Total agreement**	**Test agreement**
All 9 models	7%	7%
All 3 GNN models	25%	20%
Top 3 transformer models	44%	34%
All 5 transformer models	29%	22%

Table 2.4 shows that the deep learning models learned diverse classifiers in that only 7% of the test data (and 7% total data) are agreed by all the models. The 3 GNN models agreed on 20% of test examples (and 25% total), whereas the 3 top performing transformers (LineVul, PLBART, and VulBERTa-CNN) agreed on 34% test data (and 44% total). But when we compared all 5 transformer models, only 22% of test examples (and 29% total) are agreed. The low agreement

among different models implies that when there are no ground truth labels, a *differential testing* approach that compares across models as an oracle may have limited uses.

RQ2 Are certain types of vulnerabilities easier to detect? Should we build models for each type of vulnerabilities or should we build one model that can detect all vulnerabilities?

Motivation: In traditional software assurance techniques such as program analysis, we use different algorithms to detect different vulnerabilities. Certain types, e.g., infinite loops, are more difficult to detect than other types, e.g., memory leaks, because one requires to track symbolic values and reasoning about the loops, and the other only needs to check if the memory free is invoked after its allocation. In this RQ, we are interested to learn whether for deep learning vulnerability detectors, it is also true that certain types of vulnerabilities are easier to detect than the others. Considering different types of vulnerabilities have different semantics and root causes, we also want to gain some insights as to whether we should build a model for each type of vulnerability or for vulnerability in general (without separating them into the types), like most current work has done.

Table 2.5: Five types of vulnerabilities.

Vulnerability Type	**Total**	**CWE examples**
Buffer overflow	37,291	CWE-125, CWE-787
Value error	15,126	CWE-190, CWE-369
Resource error	33,748	CWE-415, CWE-404
Input validation error	25,514	CWE-134, CWE-89
Privilege escalation	32,749	CWE-264, CWE-255

Study Setup: Here, we study models based on the vulnerability types, and thus we used the MSR dataset. The examples in the MSR dataset are annotated with CWE [4]. Using these CWE types, we group the vulnerabilities into 5 categories, namely *buffer overflow*, *value error*, *resource error*, *input validation error*, and *privilege escalation*, shown under *Vulnerability Types* in Table 2.5. Our criteria are that (1) each group contains the bugs of similar root causes and

[4] https://cwe.mitre.org

semantics, and (2) each group has a sufficiently large dataset for effectively training the models. Column *Total* lists the number of the examples collected from the MSR datasets, including all vulnerable and their patched examples that have a CWE annotation.

Specifically, buffer overflow is caused by reading or writing to the memory outside the bounds of the buffer, e.g, CWE-125 "Out-of-bounds Read" and CWE-787 "Out-of-bounds Write". The mapping of the complete CWE list to the 5 groups is given in our dataset. *Value error* includes the examples of CWE-190 "Integer Overflow", CWE-369 "Divide By Zero" and CWE-682 "Incorrect calculation". Such errors are caused by propagating incorrect values through data processing or arithmetic operations. *Resource error* is caused by incorrectly freeing or using a resource such as memory or a file pointer, and are typically detected using typestate analysis [35]. The CWE examples include CWE-415 "Double Free" and CWE-404 "Improper Resource Shutdown". *Input validation error* is caused by using an external input without validating whether it is correct/benign, e.g., CWE-134 "Use of Externally-Controlled Format String" and CWE-89 "Improper Neutralization of Special Elements used in an SQL Command" ('SQL Injection'). They are often detected using taint analysis [39]. Finally, *Privilege escalation* is caused by missing proper permission checks and allowing an unauthorized entity to execute privileged commands or view privileged data, such as CWE-264: "Permissions, Privileges, and Access Controls", and CWE-255: "Credentials Management Errors".

We partitioned the dataset for each bug type into train, valid, and test datasets with 80%/10%/10% ratios. We trained 5 models using the 5 groups of bugs respectively, as well as a *Combined Model* trained with all the bug types for comparison. This reflects the real-world scenarios in that a model trained with specific vulnerability type may be more focused, but the combined model can train with more data. We report the *same-bugtype* performance and *cross-bugtype* performances for each model. The same-bugtype performance reports the test F1 score when the training and test data have the same bug type. The cross-bugtype performances report the test F1 scores when the training and test data have different bug types.

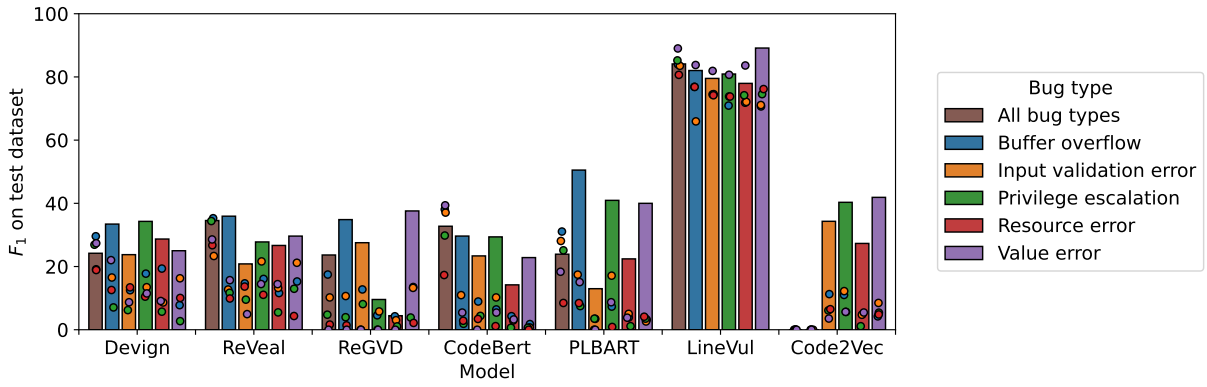


Figure 2.1: Same-bugtype and cross-bugtype performance. Bars indicates same-bugtype performance; circles indicate cross-bugtype performance for each other bugtype.

In this experiment, VulBERTa-CNN, VulBERTa-MLP and some models for Code2Vec did not report valid results because they always predicted the same class on the test data.

Findings: We present out results in Figure 2.1. The bars report the F1 score for the same-bugtype setting, and the circles report the cross-bugtype performances. Each bug type is associated with 4 cross-bugtype test sets, and thus we have four circles for each bar, except that the combined model has five circles, each of which represents running tests for a bug type on the combined models.

Analyzing same-bugtype performance, we found that the models did not always agree on which type of vulnerability is the easiest, and different types of vulnerabilities have achieved the best F1 score in different models. Interestingly, Input validation and Resource errors (the orange and red bars) often reported lower performance than the other types. On the contrary, Buffer overflows and Value errors (the blue and purple bars) often reported better performance compared to other types. In traditional program analysis, these types are harder to detect because they require tracking variable values, and sometimes reasoning about loops.

Devign and ReVeal used GNN architectures on the property graphs. Their bars show the similarities. For the rest of the models, resource errors (the red bars) manifest the lowest performance among the 5 vulnerability types. One possibility is that resource allocation and free

can be located far apart in the code, and the transformer models cannot capture such long range dependencies. Another possibility is that such errors cover a variety of resources, and the training dataset may not contain sufficient data for each resource for the model to extract the patterns.

The combined model (the brown bar) is generally less performant compared to the models trained with a specific type of vulnerability, but for some vulnerability types like Input validation and Resource errors, the combined model can perform better. For example, for CodeBERT, the combined model achieved higher F1 compared to all of the other 5 models. The circles inside the brown bar show that Privilege escalation and Resource errors reported relatively low accuracy, but for all of the vulnerability types, the combined model reported better performance compared to the type specific models.

Analyzing cross-bugtype performance, we found that for the most of time, cross-bugtype detection reports much lower performance except LineVul, implying that different vulnerability types represent different data distributions. LineVul seems to handle value errors very well. When applying models trained with other vulnerabilities, value errors reported higher performance than the same-bug performance.

RQ3 Are programs with certain code features harder to be predicted correctly by the current vulnerability detection models, and if so, what are those code features?

Motivation: Here, we investigated whether we can characterize the programs that cannot be predicted well and are "hard" to deep learning models, and whether different models agree on such difficulties. Knowing what programs we cannot handle gives a good target for our future work to improve upon. In program analysis, we know that certain features are hard to handle, such as loops and pointers. We want to know whether these features are also hard for deep learning.

Study Setup: In the first step, we prepared a list of code features for investigation. We think it is interesting to compare with program analysis tools regarding what types of programs are hard to handle. So our approach is to list code features that are important to program analysis, and then check if they also made a difference for deep learning tools.

We obtained a total of 12 code features. Some are control flow related, e.g., the structures of *while*, *for*, *if*, *goto*, *call* and *switch* as well as *unconditional jumps* of break, continue, return; some are data structure and pointer related, e.g., *arrays* and *pointers* (including the field accesses); and finally some are auxiliary structures such as *comment* and *macro*. Based on this feature list, we applied the *tree-sitter* [5] parser to count the frequency of code features present in each function.

To understand whether certain code features make deep learning models harder to predict, we used a *multivariate logistic regression (LR)* model (see Eq. 1) to associate the code features with the likeliness of a function being predicted correctly. If a function with certain code features is more likely predicted correctly, we consider it as easier to deep learning, and vice versa. Given a function with a particular feature composition, Y in Eq. 1 reports the predicted probability that the deep learning model will predict it correctly. x_i is the count of each code feature in the function. β_is are the coefficients learned from the data. Each β_i is associated with one code feature x_i. When β_i is negative, the value $\beta_i * x_i$ decreases the predicted probability of correctness, so we term these features to be *difficult* for the model. Likewise, code features with positive coefficients are termed to be *easy*. Meanwhile, a large β_i implies that the increase in the count of code feature x_i greatly increases the predicted probability of correct prediction, and vice versa.

$$Y = \sigma(\sum_i \beta_i * x_i + \beta_0) \tag{2.1}$$

We trained the LR model on the predictions made on the validation set, and then used the trained LR model to find difficult/easy examples from the test set. To quantify the difficulty of an example in test set, we used the logit input to the sigmoid function in the LR model: $\ell(x) = \sum_i \beta_i * x_i$. We denote the negation of this quantity as the *difficulty score*. A function with a higher difficulty score is expected to be more likely predicted incorrectly than a function with a lower difficulty score. To evaluate the effectiveness of this LR model, we selected the top and bottom 10% of examples in the test set, sorted by their difficulty scores. We then evaluate the

[5] https://tree-sitter.github.io/tree-sitter/

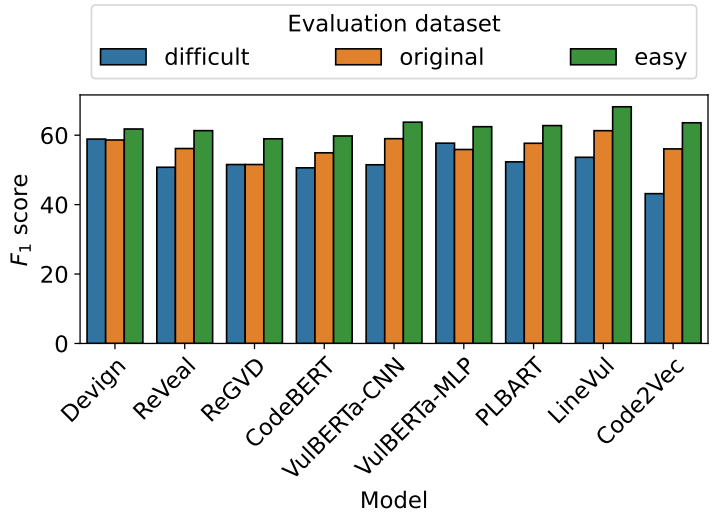


Figure 2.2: Comparative performance on evaluation sets selected according to LR model difficulty score, averaged over 3 random seeds on the Devign dataset. "Original" is the performance on the original test set, reported in Table 2.2.

predictions of our LR model by comparing the model performance on the easy and difficult datasets we selected.

It should be noted that initially we have tried statistical significance tests and correlation-based methods to link code features with the model accuracy. Then, we realized that these approaches consider only one feature a time. For example, the functions with 3 loops, 400 pointers, and 500 macros perform well, while those with 300 loops, 300 pointers and 100 macros perform bad. We cannot conclude whether the performance difference is due to the loops, pointers or macros. On the other hand, the LR model incorporates multiple code features at once and considers the effect of combinations of the features.

Findings: Figure 2.2 shows that all 9 models performed better on the easy dataset than on the difficult set. The average difference between easy/difficult performance was 10.3% for all models. For the majority of models (7 out of 9), the original test set performance lies between the performances of difficult and easy sets. These results demonstrate that the LR model and difficulty score are effective for choosing difficult and easy examples for the deep learning models.

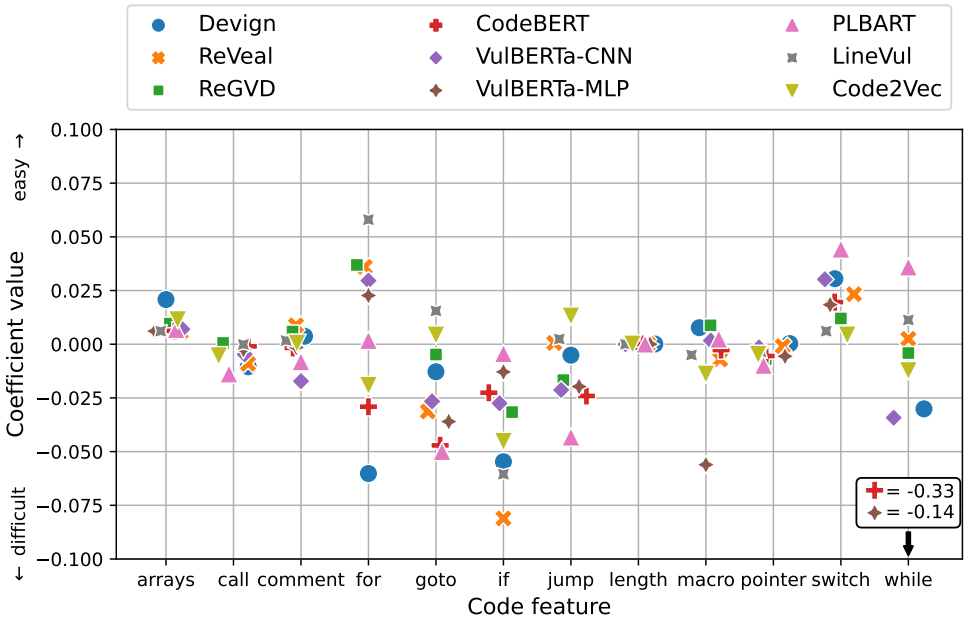


Figure 2.3: Coefficients of LR models trained on the stable examples from the Devign dataset.

Figure 2.3 plots the coefficients of each code feature in the LR model trained for each deep learning model. We found that the dots for the features `call`, `length`, and `pointers` are grouped together for all the models, implying that all the models agreed on the importance of these features. Interestingly, all of these dots are located near 0, which indicates that the features did not have a large effect. On the other hand, the features `for`, `goto`, `if`, `jump`, `switch`, and `while` varied the across models. These are all control flow related structures.

We also observed that for all the models, `arrays` and `switch` are associated with the positive coefficients, and that for the majority of models, `if`, `goto`, and `while` fall into the negative ranges. In particular, `if` had a negative coefficient for all the models. The high number of `if`, `goto`, and `while` in a program indicates its high cyclomatic complexity [28], and this type of program is also challenging for program analysis. Especially, property graph-based models like Devign use control flow information, so it makes sense that features `for`, `goto`, `jump` and `while` were all negative (hard).

2.3.2 The Training Data

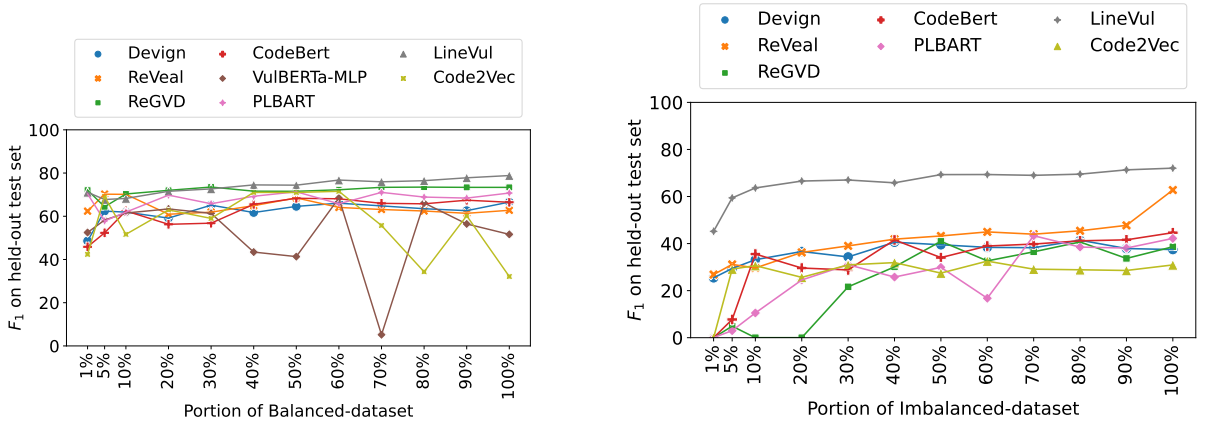
RQ4 Can increasing the dataset size help improve the model performance for vulnerability detection?

Motivation: High-quality vulnerability detection data are hard to obtain. In the past, we used automatic labeling methods such as static analysis and mining change commits. These approaches can introduce incorrect labels [6]. We also used manual labels [46], which are slow to produce [24]. This research question helps us understand whether currently available datasets are large enough to train the models, and whether increasing the dataset size can significantly improve the model performance.

Study Setup: To investigate this RQ, we combined the Devign and MSR datasets, namely, *Imbalanced-dataset*. Since the projects used in Devign overlap with the projects used in MSR, we excluded 82 duplicated examples where the commit IDs are matched. This generates a dataset of 194,285 examples in total. Some of the published models are originally tuned on the balanced model such as Devign, so we also constructed a *Balanced-dataset* of 45,363 examples by taking all the vulnerable examples in MSR and then randomly undersampling an equal number of non-vulnerable examples. For each dataset, we held out 10% data as the test set for all the models. Then we prepared 10%, 20% ... 90%, 100% of the rest of the data to train 10 models to observe how the F1 score for test set changes when the dataset size increases. In addition, we prepared two small datasets of 1% and 5% of the total data to experiment what is the minimum amount of the data needed for the model being able to learning something.

Findings: We summarized our results in Figure 2.4. Figures 2.4a and 2.4b show a similar trend. Generally, all the models increased in performance when we added more data. However, the improvement was not significant. Comparing 100% data with 10%, the F1 on test set reported no difference when we take an average over all the models for the Balanced dataset. For the Imbalanced dataset, the value of F1 score improved 0.16 on average.

In Figure 2.4a, among the models, only LineVul showed consistent improvement when we added 10% more data each time. All other models fluctuated when we increased the training



(a) Results on Balanced-dataset. 100% dataset size = 45,363.

(b) Results on Imbalanced-dataset. 100% dataset size = 194,285.

Figure 2.4: F1 score on a held-out test set when models are trained with increased portions of the training dataset.

dataset, which indicates that the increasing data does not always bring benefit. For example, for VulBERTa-MLP, the F1 value drops an average 0.1 over the course of increasing the dataset sizes. The performance of the model trained with 100% of the dataset is 0.08 less than the performance of the model trained with 10% of the dataset. It seems that other factors rather than the dataset size has played a much significant role in terms of performance. In Figure 2.4b, ReVeal is the model that improves the most with the increased dataset. At 100% dataset, ReVeal is catching up the best model LineVul. However, Devign, which has a similar architecture of using GNN on the property graph, does not show this benefit of additional data.

The experiments on models with the small datasets (consisting of 1% and 5% total data) showed that surprisingly, we can bring the models up to good performance only using 5% (about 2268 data points and 1134 vulnerable examples) for most of the models except CodeBERT, when learning with the balanced data. When learning with the imbalanced data, the turning point comes a little later. For example, ReGVD and CodeBERT require 50% (96.4 k) and 30% (57.8 k) of the total data. The other models need about 5–10% of the data (9.7–19.4 k) to reach a high point. Interestingly, this dataset has 10.8% vulnerable examples; that said, the models needed

about 1048-2095 vulnerable examples to achieve good performance—a comparable amount of vulnerable examples to the setting of the Balanced dataset.

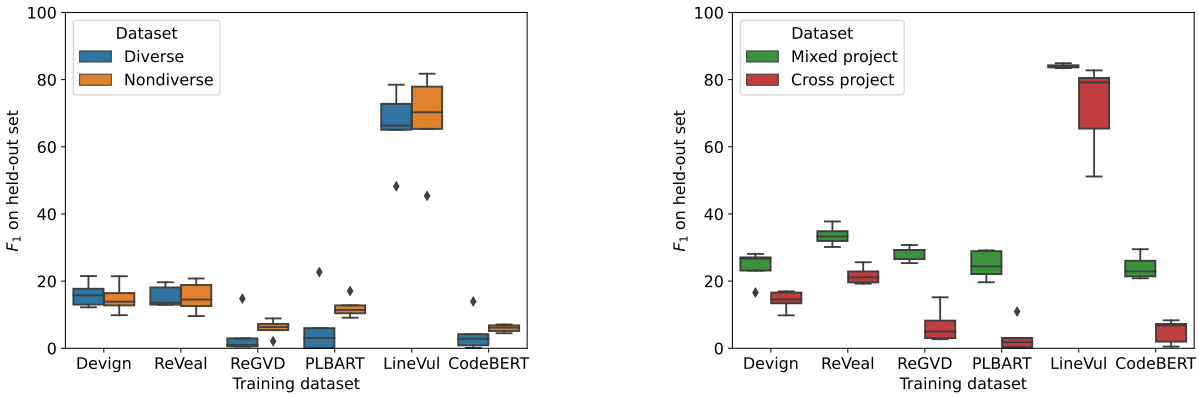
RQ5 How does the project composition in the training dataset affect the performance of vulnerability detection models?

Motivation: In this RQ, we aim to further understand how to compose a good training dataset for vulnerability detection. Specifically, we are interested to know whether the diversity of the projects in the training dataset helps. We are also interested to learn whether different projects indeed represent different distributions such that when the test and training data come from the same projects, the model can significantly perform better, and when the training and test data are from different projects, whether the models can generalize over unseen projects.

Study Setup: We designed two experiments for this study. In the first experiment, we prepared a *non-diverse* training dataset and a *diverse* training dataset, and compared the models trained with the two datasets on the same test set. In the MSR dataset, we found that *Chrome* contains 76k examples and is the largest among all the 310 projects. We used it as the non-diverse dataset. We performed this experiment in a 5-fold cross validation setting to eliminate the potential biases that may exist when selecting projects. For each fold, we randomly sampled 10 k examples from the MSR dataset as a test set. We then excluded Chrome and the projects used in the test set, and randomly sampled a total of 76 k examples (the same number as Chrome has) from the remaining projects. The average number of projects in the diverse dataset is 50.6 across 5 folds.

In the second experiment, we prepared a *mixed-projects* setting where the test set is separated from the training set without considering the source project, and some examples in the training and test sets may originate from the same projects. This is the setting where most of our deep learning papers are evaluated with. We also constructed a *cross-projects* setting where the test set examples must originate from different projects than the projects represented the training set. This setting helps us understand whether we will have a significant performance degradation when using an off-the-shelf trained deep learning vulnerability detection models that have not seen the test projects.

We also used the MSR dataset in a 5-fold cross validation setting. For each fold, we first constructed a test set for the cross-project setting by including all the examples from randomly chosen projects, until the set contained at least 10k examples. Because each project had a different number of examples, the resulting set was slightly larger than 10k examples. We then constructed a test set for the mixed-project setting by randomly partitioning the remaining examples into test (10 k), validation (10 k), and training (the remaining examples, about 158 k) sets. We trained the model, using the 158 k training examples and 10 k validation set, then ran it on the test sets for both the cross-project setting and the mixed-project setting.



(a) Diverse vs. non-diverse performance

(b) Mixed-project vs. cross-project performance

Figure 2.5: Studies on project composition in training data. The bar shows mean F1 and the interval shows standard deviation.

Findings: The results for the first experiment are presented in Figure 2.5. We used the boxplot to summarize the results of 5-fold cross-validation. To our surprise, we found that for all the models, the diverse training set does not provide any benefits compared to the training set that only consists of *Chrome*. In fact, 5 out of 6 models reported a higher median performance when trained on non-diverse data than on diverse data.

For the second experiment, Figure 2.5b shows that mixed projects perform significantly better than cross projects for all the models (the average and the largest differences in F1 score were 0.11 and 0.32 respectively). This implies that seeing the data from one project can indeed help

predict other data from the same project. Vulnerability detection can greatly benefit from customized trained models compared to directly used already trained off-the-shelf models. For LineVul, the 5 folds reported very different performances in the cross-project setting, indicating that given a target test set, some projects are more useful for training than others. The results also imply that the models may learn to detect vulnerabilities from project-specific attributes of the dataset, such as style, language features or naming conventions. We believe that this motivates further research into causal detection of bugs for generalization.

2.3.3 Internals of Deep Learning

RQ6 What code information do the models use for prediction? Do the models agree on the important features?

Motivation: Recent deep learning vulnerability detectors have achieved high performance, e.g., LineVul reported 91% F1 score. We want to know why these tools can perform well, and whether the model has used any aspects of semantics of the vulnerability to make decisions. For example, to detect buffer overflows, a semantic-based program analysis tool identifies the dependent statements and reasons about the string lengths and buffer sizes. We also want to investigate whether different models agree on what are the important features.

Study Setup: We surveyed a set of SOTA deep learning interpretation tools, especially for GNN and transformer architectures. We used GNNExplainer [43] for Devign and ReGVD, and LIT [38, 36, 32] for LineVul, VulBERTa-CNN, VulBERTa-MLP, CodeBert and PLBART, as among all the tools we investigated, these two tools can work with the most for our models. In our data reproduction package, we documented the reasons why the rest of the models cannot work with GNNExplainer and LIT, as well as the other intepretability tools we have tried.

To explain the models, both of GNNExplainer and LIT provide scores to measure the importance of the code features. GNNExplainer gives a score for each edge in the graph, and LIT gives a score for each token in the program. To compare the results reported by GNNExplainer and LIT, we performed the following *normalization* on the output of the tools. For

GNNExplainer, we calculated the score of each node by taking the average scores of all of its incident edges, as what has been done in [20]. Each token in the node will use this score. For both GNNExplainer and LIT, we calculated the score for each source code line by summing up the scores for all the tokens of that line, following the literature [14]. For each example in the test dataset, we selected the top 10 highest scored lines, as done in [20, 14], to form the *important feature set*, denoted as I. We considered these 10 lines as the most important code features the model used to make a decision.

To measure the similarity of the two important feature sets I_A and I_B reported by the models A and B, we calculated the intersection $I_{AB} = I_A \cap I_B$. We also used *Jaccard index* [18] as another metric, defined as:

$$J(I_A, I_B) = \frac{|I_A \cap I_B|}{|I_A \cup I_B|} \tag{2.2}$$

To report the similarity, we first compute I_{AB} and $J(I_A, I_B)$ for each program in the test set, and then we took the average for I_{AB} and $J(I_A, I_B)$ respectively.

We sampled examples from the following groups for manual inspection: 1) the example is vulnerable, and the model correctly detected it (correct); 2) the example is non-vulnerable, and the model predicted as vulnerable (false positive); and 3) the example is vulnerable, and the model predicted as non-vulnerable (false negative).

Table 2.6: The similarity of important feature sets between every two models, measured by I_{AB} (reported in blue) and $J(I_A, I_B)$ (in black). Max and min values are **bold**.

Model	**LineVul**	**CodeBERT**	**PLBART**	**Devign**	**ReGVD**	**VulCNN**	**VulMLP**
LineVul	-	0.58	0.31	0.38	**0.60**	0.32	0.40
CodeBERT	6.84	-	0.37	0.35	0.50	0.31	0.40
PLBART	4.08	4.89	-	**0.24**	0.27	0.30	0.28
Devign	4.90	4.54	**3.38**	-	0.42	0.27	0.30
ReGVD	**6.88**	5.86	3.58	5.36	-	0.33	0.40
VulCNN	4.20	4.06	3.88	3.87	4.35	-	0.30
VulMLP	4.83	4.74	3.71	3.95	4.83	3.97	-

Findings: In Table 2.6, we reported the similarity of the important feature sets from every model pair. Our results show that Linevul and ReGVD have a maximum overlap among all the model pairs. Among the 10 lines ranked as the important features, the two models shared an average of 6.88 lines. We found it interesting that although the models can have a lot of disagreement on individual predictions (See Table IV), the code information they used overlap. All the model pairs have at least 3 lines in common for the important features. Devign, as the only GNN model based on the property graph, has a low overlap with the other models and the lowest is with PLBART, on average 3.38 lines. PLBART used a different transformer architecture compared to the other transformer model, and also reported low overlaps with other transformer models.

We have also analyzed the corrected predicted examples using the same approach as Table 2.6. We found that the important feature sets have more overlaps for the corrected examples, e.g., Linevul and ReGVD still have the most in common, sharing 7.29 lines in the important feature sets.

Table 2.7: The frequently highlighted code features.

Model	`error`	`print`	`alloc`	`for`	`memset`	`memcpy`	`while`	`if`
LineVul	.035	.021	.017	.042	.003	.003	.005	.115
CodeBERT	.027	.015	.015	.032	.004	.002	.004	.111
PLBART	.025	.012	.011	.034	.003	.001	.005	.114
Devign	.023	.012	.014	.069	.002	.002	.007	.147
ReGVD	.034	.027	.022	.038	.004	.005	.005	.132
VulCNN	.024	.014	.011	.028	.002	.002	.004	.116
VulMLP	.031	.018	.015	.020	.003	.003	.004	.095
Func	.010	.010	.010	.026	.002	.002	.004	.108
I/F	2.79	1.78	1.52	1.47	1.39	1.21	1.12	1.10

From our manual inspection, we observed that the models commonly highlighted code lines of `for`, `if`, and `while`, as well as the function signatures as important features. The models also often highlighted memory operations of `alloc`, `memset`, and `memcpy`, as well as the lines which print error messages containing `error` or `printf`. To confirm this observation, we performed

profiling on the code using these keywords and reported the results in Table 2.7. Using `error` as an example, without loss of generality, the first 7 rows report the probability of `error` occurring in the important feature set for each model (total number of `error` in the important feature set/total number of lines in the important feature set). In Row *Func*, we show the probability of `error` occurring in a function (total number of `error`/total number of lines in a function). Comparing the two, we show that the probability of `error` occurring in the important feature sets is 2.79 times of the probability of `error` occurring in the program on average, shown in Row *I/F*. This implies that `error` is preferred to be selected into the important feature sets. Among all the features, `error`, `print`, and `alloc` ranked the highest ratios.

Our second observation is that the transformer models sometimes made predictions without seeing the root cause. This is because the transformer models take a fixed-size input, and some code, sometimes including the root cause, is truncated. Interestingly, those models are still able to correctly predict whether a function is vulnerable with high F1 score.

Third, we inspected the vulnerabilities that all the models missed and studied the important feature sets used to detect such vulnerabilities. We found that these vulnerabilities are very application specific. The bugs are missed may because there are not sufficient training data for such types of bugs.

In Listing 1, we show an example where the prediction is correct, but the features used are not causal. The example contains a memory leak vulnerability, and LineVul predicted the example as vulnerable. The memory allocated to `name` at line 14 is never released. The patch at line 23 showed a fix. The important feature set (top-10 lines) reported by LIT are highlighted in yellow. We can see that it includes the "patterns" we have discussed, including function signature at line 1, the lines that contain `ERROR`, e.g., lines 13 and 16, as well as the `if` statement at line 21. We also see that for this project, variable `name_len` is important and included multiple times. However, none of the 10 lines cover the memory allocation at line 14, which is important to understand this bug.

This example indicates that the models try to capture patterns of a vulnerability, instead of reasoning about the values, and have difficulty capturing long range semantic dependencies in the

```
1  static int asf_read_ext_content(AVFormatContext *s, const GUIDParseTable *g) // (3)
2  {
3      ASFContext *asf  = s->priv_data;
4      AVIOContext *pb  = s->pb;
5      uint64_t size    = avio_rl64(pb); //(7)
6      uint16_t nb_desc = avio_rl16(pb); //(8)
7      int i, ret;
8      for (i = 0; i < nb_desc; i++) {
9          uint16_t name_len, type, val_len; //(5)
10         uint8_t *name = NULL;
11         name_len = avio_rl16(pb);
12         if (!name_len)
13             return AVERROR_INVALIDDATA; //(6)
14         name = av_malloc(name_len);
15         if (!name)
16             return AVERROR(ENOMEM); //(9)
17         avio_get_str16le(pb, name_len, name, //(4)
18                          name_len); //(2)
19         type    = avio_rl16(pb); //(10)
20         val_len = avio_rl16(pb);
21         if ((ret = process_metadata(s, name, name_len, val_len, type, &s->metadata)) < 0) //(1)
22         ret = process_metadata(s, name, name_len, val_len, type, &s->metadata);
23         av_freep(&name);
24         if (ret < 0)
25             return ret;
26     }
27 }
```

Listing 1: Memory leak successfully detected by LineVul. The top-10 lines reported by LIT are highlighted in yellow.

```
1  static int decode_frame(AVCodecContext *avctx,
2                          void *data, int *got_frame, AVPacket *avpkt) // (1)
3  {
4      //...10 lines
5      bytestream2_init(&s->gb, avpkt->data, avpkt->size); // (5)
6      if ((ret = ff_tdecode_header(&s->gb, &le, &off))) { // (6)
7          av_log(avctx, AV_LOG_ERROR, "Invalid TIFF header\n"); // (3)
8          return ret;
9      } else if (off >= UINT_MAX - 14 || avpkt->size < off + 14) { // (4)
10
11         return AVERROR_INVALIDDATA;
12     }
13     s->le          = le;
14 // TIFF_BPP is not a required tag and defaults to 1 // (10)
15     s->bppcount    = s->bpp = 1; // (9)
16     s->photometric = TIFF_PHOTOMETRIC_NONE; // (7)
17     s->compr       = TIFF_RAW; // (8)
18     // ...140 lines
```

Listing 2: Non-vulnerable code is predicted as vulnerable because of spurious features.

code. But we also observed in other examples that sometimes, the control structures and memory statements highlighted as important (see Table VII) can be a part of the dependent statements of the vulnerability, and thus they are useful for inspecting the root cause of the bugs.

Listing 2 shows an example of a non-vulnerable function which LineVul erroneously predicted as vulnerable. The model highlighted the function signature (line 2), lines with "ERROR" (lines 7 and 10), initialization routines (line 5), `if` (lines 6 and 9), and field assignments (lines 15-17), and predicted the function as vulnerable. It showed that making decisions based on the patterns of these structures can lead to mistakes.

2.4 Threats to Validity

Our observations are drawn from the models and data sets available and may not be generalized for deep learning vulnerability detection in general. We used both a balanced dataset (Devign) and an imbalanced dataset (MSR) to mitigate this threat. The two datasets both included real-world bugs. Devign is used by most of the models in their evaluation, so we need it to reproduce the models (see Table 2.1). However, our datasets may still not be representative of

the real-world vulnerability distribution. We included all the models that we could find and reproduce.

The grouping in RQ2 is subject to bias in that different researchers may divide vulnerability types differently. Here, two of the authors who have the domain knowledge inspected the CWE list individually and discussed and agreed on the grouping. To mitigate the bias that may be brought in by the specific project compositions, RQ5 performed 5-fold cross-validation. For RQ6, we selected the SOTA model interpretation tools; however, such techniques may not be perfect to identify the important features that the models use. The experiments for RQ2, RQ4 and RQ5 require the models to work with our customized data that are not shipped with the models. We tried different random seeds for any suspicious data we have observed, e.g., when a model reported all 0s or 1s. We excluded such models in our results when tuning could not resolve the issues.

2.5 Related Work

Several works have done empirical studies of machine learning based vulnerability detection models. Chakraborthy et al. [6] studied 4 DL models, and investigated the issues of synthetic datasets, data duplication and data imbalance, and pointed out the use of spurious features, then used these to improve their model design. Tang et al. [37] aim to determine which neural network architectures, vector representation methods, symbolization methods are the best. They surveyed 2 models. Mazuera-Rozo et al. [27] evaluated 1 shallow and 2 deep models on binary classification and bug type (non-binary) classification. After we completed our study, we found two related empirical studies. Lin et al. [25] evaluated 6 DL models' generalization for 9 software projects. Ban et al. [3] evaluated 6 machine learning models (1 of which is a neural network) in a cross-project setting with 3 software projects, and also studied training on 2 bug types vs. a single bug type.

Recently, many vulnerability detection models are proposed with a variety of architectures, such as MLP [8], RNN [23, 22, 21, 44], CNN [33, 41], Transformer [13, 30, 40, 10, 14, 2, 11, 15], and GNN [7, 9, 4, 20, 34, 5, 42, 17, 46, 6, 29, 16]. For example, Devign used gated graph neural network on property graphs [1]. LineVul [14] used a transformer model pretrained over a large

body of diverse open-source projects. ReVeal [6] applied SMOTE to address the data imbalance issue and triplet loss to learn to maximally separate vulnerable and non-vulnerable code.

In these papers, most models were evaluated on in-distribution data, where the training set can contain projects and bug types which overlap with the test set. Russell et al. [33], Li et al. [23], and Xu et al. [42] trained their models to detect specific kinds of vulnerabilities, and all found that some vulnerabilities were more difficult than others. Hin et al. [17] evaluated their model in a cross-project setting by holding out one project at a time and found that the performance was slightly degraded. Most model evaluations compared different baselines on metrics such as F1, but did not quantify the agreement on the predictions. To the best of our knowledge, our work is the first attempt to characterize the programs and code features which the model cannot predict well.

2.6 Conclusions and Future Work

To understand deep learning vulnerability detection models, we performed an empirical study with 6 research questions. We experimentally show that on average, 34.9% test data have different predictions between runs, and only 7% of predictions are agreed across 9 models. Vulnerability detection based on a specific type generally performs better than a model built for all vulnerabilities. The model performance does not increase significantly with an increased dataset, and for both balanced and imbalanced datasets, the models start performing well using around 1k vulnerable examples. We developed a logistic regression model that can find programs that are difficult for the model to predict correctly. The explanation tools showed that the models used common features to make predictions, ranging from 3.38-6.88 lines in common per top 10 important lines. We report the code patterns that the models frequently highlighted as important features. In the future work, we plan to further investigate these patterns.

2.7 Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We thank Hongyang Gao for providing computing resources for our experiments. We thank Qi Li for discussing an experiment

metric. This research is partially supported by the U.S. National Science Foundation (NSF) under Award #1816352.

2.8 Bibliography

[1] Joern. https://github.com/octopus-platform/joern.

[2] Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Online, June 2021), K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds., Association for Computational Linguistics, pp. 2655–2668.

[3] Ban, X., Liu, S., Chen, C., and Chua, C. A performance evaluation of deep-learnt features for software vulnerability detection. *Concurrency and Computation: Practice and Experience 31*, 19 (2019), e5103. e5103 cpe.5103.

[4] Cao, S., Sun, X., Bo, L., Wei, Y., and Li, B. BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology 136*, C (Aug. 2021).

[5] Cao, S., Sun, X., Bo, L., Wu, R., Li, B., and Tao, C. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. *arXiv:2203.02660* (Mar 2022). arXiv: 2203.02660.

[6] Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering 48*, 09 (Sept. 2022), 3280–3296.

[7] Cheng, X., Wang, H., Hua, J., Xu, G., and Sui, Y. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering Methodology 30*, 3 (Apr 2021), 38:1–38:33.

[8] Coimbra, D., Reis, S., Abreu, R., Păsăreanu, C., and Erdogmus, H. On using distributed representations of source code for the detection of c security vulnerabilities, 2021.

[9] Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *Proceedings of the International Conference on Learning Representations* (2020).

[10] Ding, Y., Buratti, L., Pujar, S., Morari, A., Ray, B., and Chakraborty, S. Towards learning (dis)-similarity of source code from program contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Dublin, Ireland, May 2022), S. Muresan, P. Nakov, and A. Villavicencio, Eds., Association for Computational Linguistics, pp. 6300–6312.

[11] Ding, Y., Suneja, S., Zheng, Y., Laredo, J., Morari, A., Kaiser, G., and Ray, B. VELVET: a novel ensemble learning approach to automatically locate vulnerable statements. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering* (Los Alamitos, CA, USA, Mar 2022), SANER '22, IEEE Computer Society, pp. 959–970.

[12] Fan, J., Li, Y., Wang, S., and Nguyen, T. N. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (New York, NY, USA, 2020), MSR '20, Association for Computing Machinery, p. 508–512.

[13] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (Online, Nov. 2020), T. Cohn, Y. He, and Y. Liu, Eds., Association for Computational Linguistics, pp. 1536–1547.

[14] Fu, M., and Tantithamthavorn, C. LineVul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories* (New York, NY, USA, 2022), MSR '22, Association for Computing Machinery, p. 608–620.

[15] Hanif, H., and Maffeis, S. VulBERTa: Simplified source code pre-training for vulnerability detection. In *Proceedings of the 2022 International Joint Conference on Neural Networks* (2022), IJCNN '22, pp. 1–8.

[16] Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *International Conference on Learning Representations* (2020).

[17] Hin, D., Kan, A., Chen, H., and Babar, M. A. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories* (New York, NY, USA, 2022), MSR '22, Association for Computing Machinery, p. 596–607.

[18] Jaccard, P. The distribution of the flora in the alpine zone.1. *New Phytologist 11*, 2 (1912), 37–50.

[19] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science 378*, 6624 (Dec. 2022), 1092–1097.

[20] Li, Y., Wang, S., and Nguyen, T. N. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2021), ESEC/FSE 2021, Association for Computing Machinery, pp. 292–303.

[21] Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., and Jin, H. VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector. *IEEE Transactions on Dependable and Secure Computing 19*, 4 (Jul 2021), 1–1. arXiv: 2001.02350.

[22] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities . *IEEE Transactions on Dependable and Secure Computing 19*, 04 (July 2022), 2244–2258.

[23] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the Network and Distributed System Security Symposium* (2018), NDSS '18, Internet Society.

[24] Lin, G., Wen, S., Han, Q.-L., Zhang, J., and Xiang, Y. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE 108*, 10 (2020), 1825–1848.

[25] Lin, G., Xiao, W., Zhang, L. Y., Gao, S., Tai, Y., and Zhang, J. Deep neural-based vulnerability discovery demystified: data, model and performance. *Neural Computing and Applications 33*, 20 (Oct 2021), 13287–13300.

[26] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *CoRR abs/2102.04664* (2021).

[27] Mazuera-Rozo, A., Mojica-Hanke, A., Linares-Vásquez, M., and Bavota, G. Shallow or Deep? An Empirical Study on Detecting Vulnerabilities using Deep Learning. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension* (May 2021), ICPC '21, pp. 276–287. ISSN: 2643-7171.

[28] McCabe, T. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 4 (1976), 308–320.

[29] Nguyen, V.-A., Nguyen, D. Q., Nguyen, V., Le, T., Tran, Q. H., and Phung, D. ReGVD: Revisiting graph neural networks for vulnerability detection. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (2022), pp. 178–182.

[30] Phan, L., Tran, H., Le, D., Nguyen, H., Annibal, J., Peltekian, A., and Ye, Y. CoTexT: Multi-task learning with code-text transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming* (Online, Aug. 2021), R. Lachmy, Z. Yao, G. Durrett, M. Gligoric, J. J. Li, R. Mooney, G. Neubig, Y. Su, H. Sun, and R. Tsarfaty, Eds., NLP4Prog '21, Association for Computational Linguistics, pp. 40–47.

[31] Puri, R., Kung, D., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J. T., Chen, J., Choudhury, M., Decker, L., Thost, V., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., and Reiss, F. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (2021), J. Vanschoren and S. Yeung, Eds., vol. 1.

[32] Ribeiro, M. T., Singh, S., and Guestrin, C. "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), KDD '16, pp. 1135–1144.

[33] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. Automated vulnerability detection in source code using deep representation learning. *17th IEEE International Conference on Machine Learning and Applications* (2018), 757–762. Publisher: IEEE.

[34] Song, Z., Wang, J., Liu, S., Fang, Z., and Yang, K. HGVul: A code vulnerability detection method based on heterogeneous source-level intermediate representation. *Security and Communication Networks 2022*, 1 (2022), 1919907.

[35] Strom, R. E., and Yemini, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering SE-12*, 1 (1986), 157–171.

[36] Sundararajan, M., Taly, A., and Yan, Q. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (2017), ICML '17, JMLR.org, p. 3319–3328.

[37] Tang, G., Meng, L., Wang, H., Ren, S., Wang, Q., Yang, L., and Cao, W. A Comparative Study of Neural Network Techniques for Automatic Software Vulnerability Detection. In *2020 International Symposium on Theoretical Aspects of Software Engineering* (Dec 2020), TASE '20, pp. 1–8.

[38] Tenney, I., Wexler, J., Bastings, J., Bolukbasi, T., Coenen, A., Gehrmann, S., Jiang, E., Pushkarna, M., Radebaugh, C., Reif, E., and Yuan, A. The language interpretability tool: Extensible, interactive visualizations and analysis for NLP models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Online, Oct 2020), Association for Computational Linguistics, pp. 107–118.

[39] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. TAJ: effective taint analysis of web applications. *ACM SIGPLAN Notices 44*, 6 (Jun 2009), 87–97.

[40] Wang, X., Wang, Y., Mi, F., Zhou, P., Wan, Y., Liu, X., Li, L., Wu, H., Liu, J., and Jiang, X. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation, Sep 2021. arXiv:2108.04556.

[41] Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., and Jin, H. VulCNN: an image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA, May 2022), ICSE '22, Association for Computing Machinery, pp. 2365–2376.

[42] Xu, J., Ai, J., Liu, J., and Shi, T. ACGDP: An Augmented Code Graph-Based System for Software Defect Prediction. *IEEE Transactions on Reliability* (2022), 1–10. Conference Name: IEEE Transactions on Reliability.

[43] Ying, R., Bourgeois, D., You, J., Zitnik, M., and Leskovec, J. GNNExplainer: A tool for post-hoc explanation of graph neural networks. *CoRR abs/1903.03894* (2019).

[44] Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering* (Montreal, QC, Canada, May 2019), ICSE '19, IEEE, pp. 783–794.

[45] Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., and Su, Z. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA, 2021), ICSE-SEIP '21, Association for Computing Machinery.

[46] Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems 32* (2019).

CHAPTER 3. DEEPDFA: DATAFLOW ANALYSIS-INSPIRED DEEP LEARNING FOR EFFICIENT VULNERABILITY DETECTION

Benjamin Steenhoek[1], Hongyang Gao[2], and Wei Le[3]

[1-3] Department of Computer Science, Iowa State University, Ames, IA, 50011

Modified from a manuscript published in the *46th International Conference on Software Engineering (ICSE 2024)*

Abstract

Deep learning-based vulnerability detection has shown great performance and, in some studies, outperformed static analysis tools. However, the highest-performing approaches use token-based transformer models, which are not the most efficient to capture code semantics required for vulnerability detection. Classical program analysis techniques such as dataflow analysis can detect many types of bugs based on their root causes. In this chapter, we propose to combine such causal-based vulnerability detection algorithms with deep learning, aiming to achieve more efficient and effective vulnerability detection. Specifically, we designed DeepDFA, a dataflow analysis-inspired graph learning framework and an embedding technique that enables graph learning to simulate dataflow computation. We show that DeepDFA is both performant and efficient. DeepDFA outperformed all non-transformer baselines. It was trained in 9 minutes, 75x faster than the highest-performing baseline model. When using only 50+ vulnerable and several hundreds of total examples as training data, the model retained the same performance as 100% of the dataset. DeepDFA also generalized to real-world vulnerabilities in DbgBench; it detected 8.7 out of 17 vulnerabilities on average across folds and was able to distinguish between patched and buggy versions, while the highest-performing baseline models did not detect any vulnerabilities. By combining DeepDFA with a large language model, we surpassed the

state-of-the-art vulnerability detection performance on the Big-Vul dataset with 96.46 F1 score, 97.82 precision, and 95.14 recall. Our replication package is located at https://doi.org/10.6084/m9.figshare.21225413.

3.1 Introduction

Software vulnerabilities cause great harm to people and corporations. Many Internet users have had their personal information breached because of security vulnerabilities, with common reports of breaches exposing millions of records [52]. The average data breach costs the target company $4.24 million, according to IBM's 2021 report [2]. The number of vulnerabilities is growing every year, as reported by the Common Vulnerability Enumeration (CVE) from 2016-2021 [1]. Due to its importance, we urgently need to develop effective and automatic vulnerability detection tools.

The rapid advance of AI technologies has motivated software companies to invest heavily in deep learning-based vulnerability detection tools [39, 55]. These tools have outperformed traditional static analysis [38, 20, 11]. Recently, large language models (LLMs) have reported state-of-the-art results; LineVul [24], a recent model based on CodeBERT, reported 91 F1 score on a commonly used real-world vulnerability dataset [22].

However, LLMs require large amounts of training data and computational resources for training and inference (see Section 3.5.4), but a large volume of high-quality vulnerability detection data is hard to get. They also can fail to detect vulnerabilities beyond the training dataset (see Section 3.5.5); for example, the top-performing transformer models LineVul and UniXcoder were not able to detect any of the real-world vulnerabilities in DbgBench [9]. Furthermore, by using solely text tokens, these models may not effectively learn program semantics, such as program values along paths, propagation of taint values, and security-sensitive API calls along the control flow paths. The performance of these models can be further improved when we consider such information (see Section 3.5.3).

In this chapter, we explore the idea of combining *dataflow analysis (DFA)* algorithms with deep learning to develop small, efficient, yet effective models for vulnerability detection. In prior literature [53, 16], deep learning integrated with domain-specific knowledge and algorithms has reported improved performance and better generalization to unseen data, while using less data and computational resources.

Dataflow Analysis (DFA) computes the data usage patterns and relations in the control flow graph (CFG) of a program and reports a vulnerability based on its root cause, i.e., whether the values and data relations collected from the program indicate the occurrence of the vulnerable conditions. *Graph learning* (*learning based on graph neural networks (GNN)*) can aggregate and propagate information in the graph in a similar fashion to DFA. In this chapter, we explore the analogy between DFA and the GNN message-passing mechanism and design an embedding technique that encodes dataflow information at each node of the CFG. Specifically, we leverage the efficient *bit-vector* representation of dataflow facts to encode the definitions and uses of the variables. Graph learning on such an embedding propagates and aggregates dataflow information and thus simulates the dataflow computation as done in DFA. Using this approach, we hope that the learned graph representation can better encode program semantic information, e.g., *reaching definitions*, which will be very useful for accurate vulnerability detection.

Based on this rationale, we developed an *abstract dataflow embedding* that can map variable definitions of individual programs to a common space so that the model can compare and generalize data usage patterns (dataflow) related to vulnerabilities across programs. We selected a graph learning architecture whose aggregate and update functions worked most effectively for the dataflow propagation.

Our evaluation shows that DeepDFA is substantially faster than our baseline models in terms of both training and inference time. It only took 9 minutes to train, and inference on a CPU took 5.8 ms/example. This remarkable efficiency permits applications for personalized training and inference in non-GPU environments. It is also efficient in its use of training data, achieving its best F1 score using only 50+ vulnerable examples and several hundred total examples

(Section 3.5.4). This frugality allows applications within a single development team, where it may be impractical to collect thousands of vulnerable examples. Yet, DeepDFA still outperformed all non-transformer baselines (Section 3.5.3) and retained its performance on unseen projects better than all baseline models (Section 3.5.5). Additionally, when applied to a real-world benchmark of unseen projects, DbgBench [9], DeepDFA detected 8.7 out of 17 of bugs (averaged over 3 runs) and correctly reported 3 out of 5 patched programs as non-vulnerable (Section 3.5.5). In comparison, the highest-performing baselines, LineVul [24] and UniXcoder [26], did not detect any vulnerabilities. We also show that DeepDFA's learned representation can be used with other models to further improve their performance. By combining UniXcoder with DeepDFA, we surpassed state-of-the-art performance with 96.46 F1 score, 97.82 precision, and 95.14 recall.

In summary, we made the following contributions:

1. We designed an abstract dataflow embedding to enable deep learning to generalize semantics/dataflow patterns of vulnerabilities across programs (Section 3.4.1);
2. We applied graph learning on the control flow graph (CFG) of the program and abstract dataflow embedding to simulate reaching definition dataflow analysis (Section 3.4.2);
3. We implemented DeepDFA and experimentally demonstrated that DeepDFA outperforms baselines in vulnerability detection for effectiveness, efficiency, and generalization over unseen projects (Section 3.5);
4. We provided rationale to help understand why DeepDFA performs well and is efficient (Section 3.3); and
5. We surpassed the state-of-the-art vulnerability detection performance by combining DeepDFA and UniXcoder (Section 3.5).

3.2 Overview

We propose DeepDFA, a deep learning framework guided by dataflow analysis algorithms, shown in Figure 3.1. Given the source code of a potentially vulnerable program (left), we convert

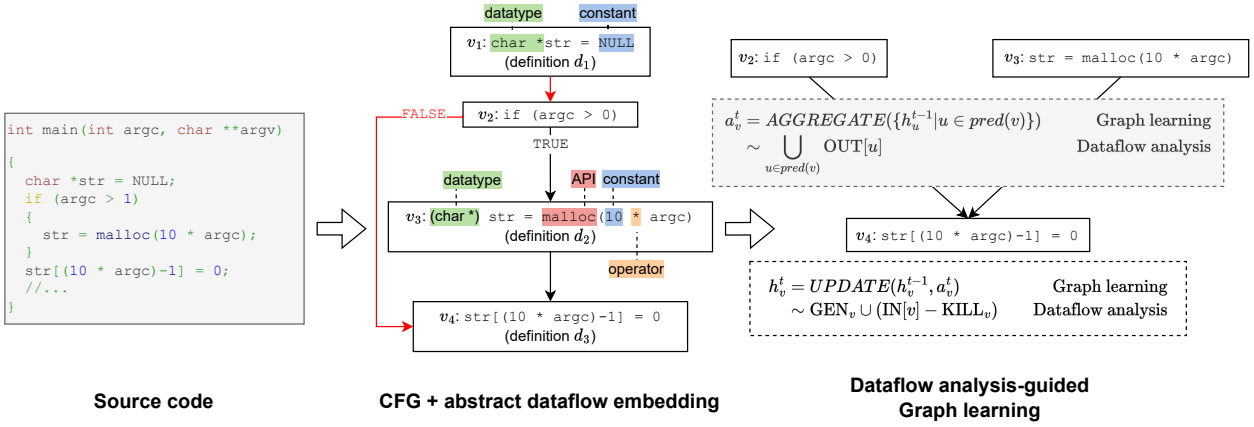


Figure 3.1: Overview of DeepDFA.

it to a CFG and encode the nodes using an *abstract dataflow embedding* which we designed. The CFG specifies the execution order of statements, and is the data structure on which dataflow analysis operates.

In the middle of the figure, we show our approach of computing abstract dataflow embeddings. In dataflow analysis, definitions of variables, e.g., `a=3`, are program specific. Applying to deep learning, we *abstract* these concrete definitions from different programs, and hypothesize that the usage patterns of the *abstract definitions* can be compared and summarized across programs during learning. To construct the abstract definitions, we used the properties of definitions that are important for vulnerability detection, based on domain knowledge from program analysis. Specifically, we considered the data types of the defined variable, the API calls, constants, and operators used to define the variables. Inspired by the bit-vector representation used in dataflow analysis, we encode the abstract definitions in a compact and very efficient fashion. We will provide more detailed design of this embedding in Section 3.4.1.

We used a bit-vector style of representing a *set* of abstraction definitions. This numerical representation can be directly used as the initial node representations for graph learning. In the right of the figure, we apply graph learning which *aggregates* the information from nodes like the "merge" operation performed in dataflow analysis, and also *updates* using the information at each

nodes like the "update" operation performed in dataflow analysis. We provide more background on the analogy in Section 3.

Finally, we use the learned graph representation to classify whether the function is vulnerable or not. By directly propagating dataflow information through graph learning, we hope to present to the classifier a representation of the program which encodes useful information directly related to vulnerability, achieving efficient and effective vulnerability detection. The advantage of deep learning is that the mapping from the encodings of programs to the decisions are learned from the data, but in dataflow analysis, we need to manually craft rules to map from the dataflow analysis results to vulnerability decisions.

3.3 Rationale

In this section, we provide the relevant background of dataflow analysis for vulnerability detection and graph learning. It provides understanding on why our approach is efficient and effective. Then, we compared the closely related work that also considers dataflow in deep learning to clarify the novelty of our work.

3.3.1 Dataflow Analysis for Vulnerability Detection

Dataflow analysis (DFA) is a method for computing data usage patterns in a program. In addition to compiler optimization, dataflow analysis is an important method for vulnerability detection. One instance of dataflow analysis, called *reaching definition analysis*, reports at which program points a particular variable definition can *reach*. A definition *reaches* a node when there is a path in the CFG that connects the definition and the node, and the variable is not redefined along the path. The reaching definition analysis can detect a null-pointer dereference vulnerability based on its root cause when it identifies that a definition of an NULL pointer reaches a dereference of the pointer. Similarly, it is a causal step to detect many other vulnerabilities such as buffer overflows, integer overflow, uninitialized variables, double-free and use-after-free [12].

DFA uses two equations to propagate the dataflow information through the neighboring nodes in the CFG, namely *meet operator* and *transfer function* [4]. The meet operator aggregates the dataflow sets from its neighbors. The transfer function updates the dataflow set using the information available in the node v. In the reaching definition analysis, the dataflow set is a set of definitions that reach a program point. A simple approach of performing a DFA is the *Kildall method* [33]. It iteratively propagates the dataflow information to the neighbors of v in the CFG, one step at a time. The algorithm terminates when the dataflow information of all nodes stops changing, denoted a *fixpoint*. At termination, all nodes will incorporate the dataflow information from all other relevant nodes. When used for vulnerability detection, this information is compared to a user-specified vulnerability condition to determine whether a vulnerability has occurred in the program.

3.3.2 Analogy of Graph Learning and Dataflow Analysis

Graph learning starts with an initial node representation, and then it performs a fixed number of iterations of the message-passing algorithm [25] to propagate information through the graph. The initial node representation is generally a fixed-size continuous vector which represents the content of the node. At each iteration, each node aggregates information from its neighbors, and then updates its state to integrate the information. The two steps are done through the *AGGREGATE* and *UPDATE* functions, similar to the two dataflow equations of meet operator and transfer function. These functions can be simple numerical equations or neural networks. After iteration is done, all node representations are combined to produce a graph-level representation, which is passed to a classifier layer to make a prediction.

In Figure 3.4, we visualize the analogy between graph learning and dataflow analysis on a snippet of CFG. In the CFG, each node is a statement, and each edge indicates the order of execution between two statements. In Figure 3.2, we show the two dataflow equations [4] that define a reaching definition dataflow analysis.

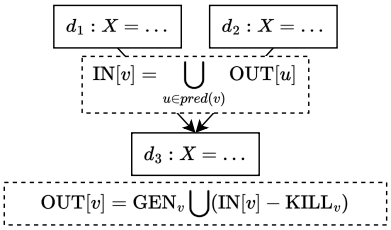


Figure 3.2: Dataflow Analysis.

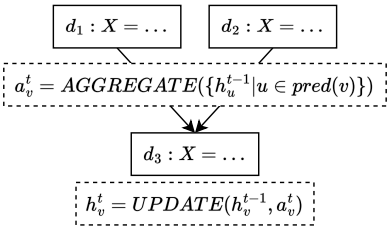


Figure 3.3: Graph Learning.

Figure 3.4: Analogy of information propagation in Dataflow Analysis and Graph Learning.

$$\text{meet operator:} \quad IN[v] = \bigcup_{u \in pred(v)} OUT[u] \tag{3.1}$$

$$\text{transfer function:} \quad OUT[v] = GEN_v \bigcup (IN[v] - KILL_v) \tag{3.2}$$

where $IN[v]$ and $OUT[v]$ are the sets of dataflow located at the beginning and end of a statement. GEN_v and $KILL_v$ represent the dataflow *generated* (new definitions) and *killed* (overwritten definition) in node v. Reaching definition is a *may* dataflow problem and thus the meet operator used *union* to merge the dataflow information from its predecessors. Meanwhile, reaching definition is a *forward* dataflow problem, and thus we used $IN[v]$, GEN_v, and $KILL_v$ to compute the dataflow at the exit of the statement.

In Figure 3.3, we show an analogous behavior of graph learning.

$$AGGREGATE: \quad a_v^t = AGGREGATE(\{h_u^{t-1} | u \in pred(v)\}) \tag{3.3}$$

$$UPDATE: \quad h_v^t = UPDATE(h_v^{t-1}, a_v^t) \tag{3.4}$$

where a_v^t denotes the aggregated information from the neighboring nodes and h_v^t denotes the state of node v after t iterations of message-passing (analogous to $OUT[v]$). We set t as a hyperparameter.

3.3.3 The Novelty of Our Work

Previously, researchers have proposed to integrate dataflow information with deep learning for program analysis tasks. A category of approaches similar to Devign [56] used data dependency graphs as a part of the program representation on which deep learning is performed. However, Devign used word embeddings to encode statements into vector representations based on their unstructured text content. Such an encoding, even propagated through data dependency edges, cannot directly capture the dataflow patterns.

ProGraML [17] has developed graph learning on LLVM IR code and applied it for compiler optimization tasks. It is another work that pointed out the analogy between DFA and graph learning. Their solution is to modify CFGs by creating instruction nodes and data nodes separately. ProGraML adds control-flow edges between instruction nodes and data-flow edges between the data nodes. However, this work encoded nodes using an embedding which only represents LLVM IR operators and variable types. This approach is very coarse-grained in that many statements can have the same operators and variable types, but they will lead to different dataflow. Therefore, similar to Devign, the propagation of such an encoding even along dataflow edges does not directly capture dataflow patterns.

Our abstract dataflow embedding attempts to directly represent the variable definitions which are propagated in DFA and is modeled after the bit-vector representation used in DFA, which allows the network to learn the operations of the dataflow analysis algorithm. We also target a specific problem (reaching definitions, which was not targeted by ProGraML), for which the results of DFA are directly useful and pertinent to vulnerability detection (e.g. § 4.2).

3.4 Approach

Based on the analogous behaviors of DFA and GNN, we designed a node embedding that can represent the dataflow set at each node. We developed DeepDFA, a deep learning framework which conducts graph learning on the CFG of a program and propagates dataflow information for vulnerability detection.

3.4.1 Abstract Dataflow Embedding

In dataflow analysis, we use a *bit vector* to represent the dataflow set at each node. A bit vector consists of n bits of 0s and 1s. Its length is the size of the domain. A bit is set to 1 if its corresponding element is present in the set. In reaching definition analysis, the domain consists of all the definitions in the program, and the bits are set to "1" if the corresponding definitions reach the node. For example, in Figure 3.1, the program contains three definitions at nodes v_1, v_3, and v_4 so the reaching definition analysis uses a bit vector $[0\ 0\ 0]$ to initialize each node at the beginning of the analysis. This bit vector represents $OUT[v]$ in the dataflow equations (See Section 3.3.2). It is updated at each step of propagation, and when the analysis terminates, the bit vectors for each node represent all possible definitions that can reach *that* node.

The bit-vector representation of reaching definition analysis efficiently encodes program semantic features related to vulnerability detection. The definitions of programs can be quickly obtained at the node via lightweight analysis locally at the statements. However, in graph learning, we cannot directly use the bit vector of definitions as the node embedding. This is because in dataflow analysis and the domain of definitions are both specific to a program. In other words, different programs have different variable definitions; the bit vectors of each program thus have different lengths and the elements (each definition) are not comparable either. Whereas, in graph learning, we want to extract dataflow patterns of vulnerabilities from all the programs in the training dataset. Thus, we need to have a "global" definition set that can be used to specify definitions for different programs, so that graph learning can compare them and generalize from them.

To address this challenge, we map all the concrete definitions in the programs in a training dataset to *abstract definitions* by identifying important properties of the definitions. Following a list of attack surfaces identified by Moshtari et al. [41], we designed the following four properties that can encompass the attack surfaces of a vulnerability and used them to represent a definition:

1. API call: the call to library or system functions used to define a variable, e.g. `malloc` and `strlen`.

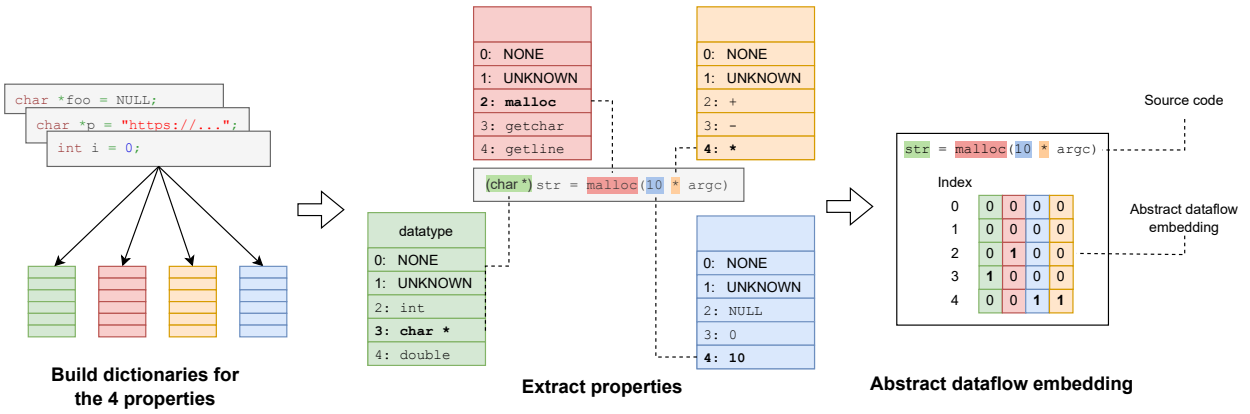


Figure 3.5: Abstract dataflow embedding generation.

2. Data type: the data type of the variable being assigned, e.g. `int`, `char*` and `float`.

3. Constant: the constant values assigned in the definition, e.g. `NULL`, `-1` and the hard-coded string `"foo"`.

4. Operator: the operators used to define a variable, e.g. `+`, `-` and `*`.

We analyze a large corpus of programs, e.g., the training set, and collect the top-k frequently used API calls, data types, constants and operators to construct a dictionary. k is a hyperparameter of DeepDFA. We select only the top-k keys because the representations of user-defined names of APIs and data types cannot be generalized across programs unless they are represented frequently in the dataset.

In Figure 3.5, we show an example of abstract dataflow embedding for an example d_2 in Figure 1: `str = malloc(10 * argc)`. This definition used an API call, `malloc`, with the constant `10`, operator `*`, and data type `char*`. Contrasted with the 3-bit bit vector (the example in Figure 1 includes three variable definitions) that represents a concrete definition in dataflow analysis for this program, the abstract embedding is larger but a fixed size, consisting of 5x4 elements for this example. Here, 4 is the four properties we considered and 5 is the hyper-parameter k we mentioned above, which defines the size of the pre-defined dictionary, and the length 5 hot-vector encoding represents the value of the property. Because the vector that encodes the abstract

dataflow embedding has a fixed size, our embedding approach can scale to any program size in the dataset without impacting the model's efficiency. The vectors in different programs encode common properties of definitions, so the model can capture the dataflow patterns across programs.

Abstraction potentially brings in approximation. Using the abstract dataflow embedding, two different definitions may lead to the same encoding. The embedding is designed to be sparse enough that within a program, unique definitions are often represented by unique embedding keys, which allows the model to distinguish definitions within the same function, similar to the bit-vector used in dataflow analysis.

3.4.2 Using Graph Learning to Propagate Dataflow Information

Our goal in utilizing graph learning is to learn a node embedding that contains dataflow information. Without loss of generality, we use *reaching definition* as an instance of dataflow analysis for our explanations. Our approach takes the following steps. First, we construct the CFG for a program. Second, we perform static analysis to identify all the definitions in the CFG. We then initialize each node of the CFG using the abstract dataflow embedding, based on whether the node is a definition or not. The abstract dataflow embedding is computed from all the programs in the training dataset (see Section 3.4.1 for details).

Once the nodes are initialized, we apply the message-passing algorithm [25] from graph learning to propagate the dataflow information throughout the CFG, similar to *Kildall's method* [33]. The main differences are that (1) we propagate the abstract dataflow embeddings of the CFG nodes, and (2) instead of using the dataflow equations of transfer function and meet operator, we alternatively apply the $AGGREGATE$ and $UPDATE$ functions defined in Equations (3.3) and (3.4) (See Section 3.3). Although the analogy applies for all GNN architectures trained with message-passing, we implemented our approach using a *Gated Graph Sequence Neural Network (GGNN)* [36], where $AGGREGATE$ is an *Multi-Layer Perceptron*

(MLP) and *UPDATE* is a *Gated Recurrent Unit (GRU)*; we will use this architecture as an example to compare the two algorithms.

When dataflow information arrives at the merge point of a branch in CFG, graph learning applies the *AGGREGATE* function. Specifically, in GGNN, the MLP calculates a weighted sum of the representations of multiple neighboring predecessors, resulting in a single vector; this fulfills the same function as the meet operator. When dataflow information arrives at a new node, the *UPDATE* function in graph learning computes the next state by combining the information in the current node with the output of *AGGREGATE* from its predecessors. Specifically, in GGNN, the GRU selectively forgets portions of the previous state and integrates new information from the current node and from the neighboring states, similar to the set union/difference with GEN/KILL performed in the transfer function. Through applying *AGGREGATE* and *UPDATE*, the initial embedding will be updated with the dataflow information from the neighboring nodes, similar to the effect of dataflow analysis.

As Cummins et al. [17] noted, DFA iterates to a fixpoint and thus propagates information throughout the entire graph, while graph learning performs a fixed number of iterations t and thus propagates to neighbors in a distance t. We set t to the setting which maximized validation-set performance.

Finally, we combine the learned abstract node embeddings to produce graph level representation using *Global Attention Pooling* [36], and pass it to a classifier to predict the function as vulnerable or non-vulnerable.

The *AGGREGATE* and *UPDATE* functions are learned from labeled data during training, rather than using a fixed formula as in dataflow analysis. By learning from data, we provide an alternative solution to the challenges that often block dataflow analysis such as tracking pointers and handling library calls. Importantly, we no longer need to explicitly specify vulnerability conditions, as required in static analysis. Through learning from training examples, the classifier can capture patterns of dataflow information that represent various types of vulnerabilities and also select the relevant dataflow information for vulnerability detection.

Table 3.1: $OUT[v]$ at each iteration of DFA.

Iteration	v_1	v_2	v_3	v_4
0	[0 0 0]	[0 0 0]	[0 0 0]	[0 0 0]
1	[1 0 0]	[0 0 0]	[0 1 0]	[0 0 1]
2	[1 0 0]	[1 0 0]	[0 1 0]	[0 1 1]
3	[1 0 0]	[1 0 0]	[0 1 0]	[1 1 1]

In Table 3.1, we step through a reaching definition analysis for the CFG example in Figure 4.2 to demonstrate how dataflow information propagates through the graph and how our approach uses dataflow information for vulnerability detection.

The row *Iteration 0* shows the initialization of each node in the reaching definition analysis. At iteration 1, the DFA updates $OUT[v_1]$, $OUT[v_3]$ and $OUT[v_4]$ using the transfer function to indicate that the new definitions are introduced at the nodes. At iteration 2, $OUT[v_1]$ (including d_1) propagates to v_2 and $OUT[v_3]$ (including d_2) propagates to v_4, through the CFG edges. At iteration 3, the meet operator is used to combine $OUT[v_2]$ and $OUT[v_3]$. Specifically, $IN[v_4] = \bigcup\{OUT[v_2], OUT[v_3]\}$, computed as $[1\ 0\ 0] \vee [0\ 1\ 0] = [1\ 1\ 0]$; then the transfer function combines $IN[v_4]$ with GEN_{v_4}, resulting in $OUT[v_4] = [1\ 1\ 1]$.

After the DFA algorithm terminates, the final states of the nodes are used to detect vulnerabilities. The state of v_4 is $[1\ 1\ 1]$, which indicates that both d_1 and d_2 may reach v_4 depending on the program values. Because the definition d_1 : `str = NULL` can reach the dereference at v_4, we can conclude that this program has a null-pointer dereference vulnerability. Similarly, in graph learning, after a fixed number of iterations, all the node representations are combined using a graph readout operation to produce a graph-level representation, which is used to predict for vulnerability detection. Programs with the null-pointer dereference bugs will have the same abstract definitions characterized by the `char*` type and the constant `NULL` to reach the pointer dereference statements. We believe that the dataflow information represented by DeepDFA will allow a relatively simple classifier to recognize this pattern among the training dataset.

3.5 Evaluation

In the evaluation, we studied 3 research questions:

1. Is DeepDFA **effective** for finding vulnerabilities?
2. Is DeepDFA **efficient**, both in terms of training data and computational resources?
3. Can DeepDFA **generalize** to unseen projects?

We also performed ablations on DeepDFA to understand the effects of each feature on its performance.

3.5.1 Implementation

To explore whether DeepDFA can advance the state-of-the-art, we created two settings, **DeepDFA** and **DeepDFA+LLM**. We implemented DeepDFA using the GGNN architecture [36] and based on LineVD's implementation[1], using PyTorch and DGL[2]. We used Joern[3] to parse the CFGs because it does not require compilation; this allows our approach to be utilized out-of-the-box given only the source code, without extra configuration. To implement DeepDFA+LLM, during training and inference, we combine the graph embedding generated by DeepDFA's graph readout stage with the sentence embedding produced by the final self-attention layer of LLM. The embeddings are concatenated and fed into a feed-forward classifier layer; both embeddings and the classifier are trained jointly. We believe that providing dataflow information can improve the LLM embedding, as LLM is trained exclusively from text and it is hard to learn dataflow relations among all the dependencies of tokens.

To avoid data leakage, we extracted the initial abstract dataflow embedding from the training set only. We set the hyperparameters k and t based on the best validation performance through our experimentation. When $k = 1000$, the model covered most (79.38%) of the definitions in the test dataset. That means, the dictionary still misses some APIs, constants, data types or

[1] https://github.com/davidhin/linevd
[2] All of our code and data are available at https://doi.org/10.6084/m9.figshare.21225413
[3] Joern version 1.1.1072, available at https://joern.io

Table 3.2: Hyperparameters used for training DeepDFA.

Hyperparameter	**Value**
λ (learning rate)	$1e^{-3}$
L_2 weight	$1e^{-2}$
k (threshold)	1000
t (number of GNN steps)	5
Hidden size	32
# output layers	3
Batch size	256

operators that occur in the test data set but are not frequent in the training dataset. To learn a more general representation and improve the coverage for test dataset, in future work, we can train the abstract dataflow embedding using a very large dataset of code, e.g., using self-supervised learning without the need of vulnerability labels.

For reproducibility, Table 3.2 documents the hyperparameters we used for training DeepDFA.

3.5.2 Experimental setup

In the recent literature [24, 35, 13], vulnerability detection models are typically evaluated with the Devign [56] or Big-Vul [22] datasets, both of which contain real-world open-source C/C++ projects. In our evaluation, we used the Big-Vul dataset because (1) it is bigger than Devign, consisting of 188,636 functions with 10,900 (6%) vulnerable labels and 177,736 (94%) non-vulnerable labels, and (2) it reflects the imbalanced distribution of real-world code (Devign is a balanced dataset), with the minority of code being labeled as vulnerable [13]. To corroborate our results, we also evaluated the models on DbgBench [9], explained further in RQ3.

Scope: Currently, DeepDFA reports whether a function is vulnerable or not. We can further apply deep learning explanation tools to report line level vulnerabilities, as done in the work of Li et al. [35]. We leave this evaluation for future work. We evaluated DeepDFA on C/C++ programs, as done by the most deep learning-based vulnerability detection tools. However, we believe that DeepDFA can also be applied to other popular programming languages, such as

Python and Java. This is because we extract the abstract dataflow embedding (API, datatype, literal, operator) from the training dataset, independent of the programming language.

RQ1: To evaluate the models' performance, we trained the models on the train/validation/test splits of 80/10/10% published by the LineVul paper [24]. To address class imbalance while training DeepDFA, we undersampled the majority class (non-vulnerable) following Japkowicz [31]; our initial studies found that this improved our performance on the validation set. We kept the original ratio of vulnerable/non-vulnerable labels for the validation and test sets. We used Joern[4] [54] to parse the code into its CFG representation. Joern could not parse some programs in the dataset (0.8%; see Section 3.B for details), so we used the remaining data in our experiments. The performance of the baseline models was similar to the full dataset (see Sections 3.C and 3.F for details).

We report the following performance metrics:

- **Precision** reports the portion of positive predictions which were correct: $P = \frac{TP}{TP+FP}$.
- **Recall** calculates the portion of positive examples which were recalled correctly: $R = \frac{TP}{TP+FN}$.
- **F1** is the harmonic mean between Precision and Recall: $F1 = 2 * \frac{P*R}{P+R}$. We used $F1$ to decide the highest performing model because it balances precision and recall, which are both important in an imbalanced dataset.

Since the model performance can vary with different random seeds [48], we trained the models 3 times with different random seeds and reported the mean score and standard deviation for each metric. We used McNemar's statistical test, following best practices [18], to confirm that our improvement is statistically significant, using the implementation in *statsmodels* v0.14.0 [47].

RQ2: To evaluate the models' efficiency in terms of computational resources, we measured the runtime and memory usage. These are often contested resources in deep learning workloads [30]. We report the following metrics for runtime and memory usage:

[4] https://joern.io

- **Training time**: the wall-clock time to execute one training run with one validation run per epoch
- **Inference time**: the average wall-clock time to predict for one example.
- **MACs**: the average number of Multiply-Accumulate operations[5] to predict for one example; this measures the performance independently of the computing platform [49, 30].
- **Parameter count**: the number of trainable parameters in the neural network model.

We evaluated the runtime on an AMD Ryzen 5 1600 3.2 GHz processor with 48GB of RAM and an Nvidia 3090 GPU with 24GB of GPU memory.

To evaluate the models' efficiency in terms of training data, we trained the models on progressively smaller subsets which we randomly sampled from Big-Vul (100%, 10%, 1%, 0.5%, 0.1%, shown in the columns of Table 3.8). Each subset includes the smaller subsets (e.g. 10% subset includes the 1% subset and 1% includes 0.5%). We generated 3 versions of the subsets using different random seeds and reported the mean and standard deviation F1 score. The goal of this study is to discover what are the minimum training data needed for these models to perform well on the test dataset.

RQ3: We prepared two experiments for this RQ. In the first experiment, we created a dataset from Big-Vul to evaluate how well the models generalize to unseen projects. This dataset consists of the *mixed-project* and *cross-project* two settings. To set up the mixed-project setting, we held out 10k randomly selected examples for the validation and test sets and used the rest for training, similar to the original method of partitioning the dataset. The training set and test set can and often do contain examples from the same project, though individual examples will not be duplicated between the two sets. To set up the cross-project setting, we held out 10k examples from randomly selected projects in Big-Vul for the validation and test sets, and used the rest of the projects for training. The projects in the test set are distinct from the projects in the training set. To mitigate the potential bias caused by the selection of projects, we repeated this process 5

[5]We used DeepSpeed profiler to measure MACs [44]. https://www.deepspeed.ai

times with different selections of the cross-project data and report the results of 5-fold cross validation.

In order to further evaluate generalization to unseen projects, we applied DeepDFA on buggy and patched programs from DbgBench [9]. DbgBench consists of a set of real-world C programs with bugs, which were analyzed and fixed by professional software engineers. The DbgBench programs are distinct from the programs in the Big-Vul dataset. We labeled the buggy functions using the fault locations documented in DbgBench; these were labeled by the consensus of multiple developers and were manually checked for correctness, and thus are more reliable than Big-Vul's labeling process based on bug-fixing commits. We included all functions which had a bug location marked as "buggy" and their corresponding patched versions in our study. We excluded the bugs marked as "Functional" because these bugs cannot be detected without program-specific bug constraints. We only included the patched versions which were modified by the developers' fixes, taking the first correct[6] developer patch which could be applied to the program. We included only one patch to reduce the effects of code duplication, which can unfairly bias test performance [5]. Since the models only view the function-level context, they will not produce a different prediction on the functions which were not modified by the patch. We also excluded 8 examples which could not be processed by Joern in order to fairly compare the models' performance scores. This resulted in a dataset of 22 programs: 17 buggy + 5 patched. We evaluated the checkpoints trained from 3 random seeds in Section 3.5.3 and report the mean performance scores.

Ablation study: We ran two ablation settings for each of the four abstract dataflow embedding features, resulting in eight settings: (1) using one feature at a time and (2) using three features at a time (leaving one out). In each setting, we trained DeepDFA on the Big-Vul [22] training dataset, and then evaluated on the Big-Vul test dataset and DbgBench [9].

Baselines: We compared against 7 non-transformer models: VulDeePecker, SySeVR, Draper, Devign, ReVeal, ReGVD, IVDetect, and 4 large language models: CodeBERT, LineVul,

[6]Marked in DbgBench as "Developer fix" or "Different but Correct Fix", e.g. https://github.com/dbgbench/dbgbench.github.io/blob/master/patches/find.dbcb10e9/README.md

Table 3.3: DeepDFA outperformed the baselines and can be used to further improve the existing model performance. All scores are reported as *Mean (Standard deviation)*. Note that VulDeePecker, SySeVR, Draper, and IVDetect performance were directly taken from the IVDetect paper [35], so we do not report the variance.

Table 3.4: Comparison with non-transformer models.

Model	**F1**	**Precision**	**Recall**
VulDeePecker	12.00	49.00	19.00
SySeVR	15.00	**74.00**	27.00
Draper	16.00	48.00	24.00
ReGVD	19.15 (2.65)	63.67 (4.43)	11.33 (1.94)
IVDetect	23.00	72.00	35.00
Devign	26.85 (0.97)	29.00 (0.38)	25.03 (1.67)
ReVeal	32.94 (0.75)	34.27 (1.58)	31.73 (0.65)
DeepDFA	**68.26** (0.16)	53.98 (0.06)	**92.81** (0.40)

Table 3.5: Comparison with transformer models.

Model	**F1**	**Precision**	**Recall**
CodeBERT	21.04 (6.72)	68.48 (11.76)	12.91 (5.51)
CodeT5	45.61 (0.71)	56.47 (6.22)	38.56 (2.45)
LineVul	93.23 (0.31)	97.32 (0.66)	89.48 (0.42)
UniXcoder	95.11 (0.21)	96.96 (1.14)	93.34 (1.23)
DeepDFA	68.26 (0.16)	53.98 (0.06)	92.81 (0.40)
DeepDFA +CodeT5	81.39 (0.96)	94.23 (2.98)	71.67 (1.09)
DeepDFA +LineVul	96.40 (0.13)	**98.69** (0.28)	94.22 (0.46)
DeepDFA +UniXcoder	**96.46** (0.09)	97.82 (0.99)	**95.14** (1.09)

UniXcoder, and CodeT5 [7]. These models were developed recently with diverse architectures, and they represent the state-of-the-art of vulnerability detection models [48]. See Section 3.7 for an overview of the models and Section 3.A in the supplementary materials for the details of our reproductions.

3.5.3 Effectiveness

Comparison with non-transformer models: In Table 3.4, we show that DeepDFA performed much better than the baseline models on F1 score and recall. DeepDFA's score was 47.51 higher than the average F1 score computed over all the baselines. In addition, compared to the other 6 models, DeepDFA reported lower variances for all the three metrics. This indicates that DeepDFA was more robust to random noise throughout training, and thus more likely to perform as expected after training.

The results show that our abstract dataflow embedding indeed encodes useful information for vulnerability detection, despite the fact that the node representation is small and the graph is simple. It is more effective than *property graphs* (a combination of AST, CFG, and PDG) used in Devign and Reveal. These baseline models represented nodes using unsupervised word embeddings [40, 34, 43], which do not have a direct relationship with vulnerabilities. In contrast, DeepDFA's node representation encodes the dataflow sets of reaching definitions, related to the root causes of vulnerabilities.

Comparison with transformer models: We compared DeepDFA with CodeBERT, LineVul, UniXcoder, and CodeT5 – the state-of-the-art among the transformer language models we evaluated. (see Section 3.C for the performances of all the baseline models). Table 3.5 shows that DeepDFA performed considerably better than CodeBERT and CodeT5 in F1 score and had the smallest variance (among all the models) between runs.

Although UniXcoder and LineVul performed better than DeepDFA in terms of F1 score, DeepDFA's embedding can be combined with UniXcoder and LineVul to further improve their performance. We achieved state-of-the-art performance on all three metrics by adding DeepDFA's embedding to UniXcoder, with an F1 score of 96.46 (1.35 improvement), a Precision score of 97.82 (0.86 improvement), and a Recall score of 95.14 (1.80 improvement). Adding DeepDFA's embedding improved CodeT5 considerably – by 35.78 F1 score – and improved LineVul by 3.17

[7] We could not reproduce LineVD and ContraFlow on Big-Vul for function-level vulnerability detection.

Table 3.6: Results of statistical tests for model comparison.

Models compared	χ^2 **statistic**	p**-value**
LineVul vs. DeepDFA+LineVul	20.0	2.77×10^{-7}
UniXcoder vs. DeepDFA+UniXcoder	25.0	5.35×10^{-4}

F1 score. We used McNemar's significance test, as recommended by Dietterich [18], to confirm that the differences in performance were statistically significant ($p < 0.05$; see Table 3.6).

DeepDFA does not use any text-/token-level information such as variable and function names, yet it has achieved excellent performance. We believe that leveraging the domain-specific algorithm of reaching definition analysis to guide graph learning indeed plays an important role and that the embedding indeed encodes semantic features (e.g., data relations) that are important for vulnerability detection. The fact that DeepDFA can further improve the top-performing LLMs indicates that LLMs, which exclusively leverage text information, may not sufficiently learn the dataflow of code; DeepDFA thus provides the complementary information for vulnerability detection. We further believe that the examples which DeepDFA predicted incorrectly could be attributed to the fact that reaching definition analysis cannot handle all types of vulnerabilities. Thus, by adding other dataflow analyses such as live variable analysis, DeepDFA could further improve its performance. We will leave such an investigation to our future work.

RQ1 result: DeepDFA performed much better than all non-transformer baselines. When combined with transformer models, it achieved the highest SOTA score on all metrics.

3.5.4 Efficiency

Efficiency of computational resources: In Table 3.7, we present the runtime comparison of DeepDFA, LineVul, and UniXcoder. Here, we did not list other models because their performances are much worse (shown in Table 3.3), and they took hours to train (see Section 3.D in the supplementary material), compared to DeepDFA which finished training in 9 minutes

Table 3.7: Training and inference time of DeepDFA and baselines. DeepDFA's training/inference time was faster than the baselines.

		Inference cost per example		
Model	**Train time (ms)**	**GPU (ms)**	**CPU (ms)**	**MACs**
LineVul	10h19m	11.1	1068.2	48.32 B
DDFA+LV	10h40m	15.4	1571.5	48.32 B
UniXcoder	11h16m	9.5	486.0	48.32 B
DDFA+UXC	13h22m	13.3	922.1	48.32 B
DeepDFA	**9m**	**4.6**	**5.8**	**40.27 M**

(excluding data preprocessing time). In Section 3.E, we also listed the sizes of the models in terms of the number of parameters.

Compared to UniXcoder, DeepDFA took 75x less time to train, 2x faster inference on GPU, and 84x faster inference on CPU. DeepDFA had the least parameters of all models, equal to 67% of the smallest model (ReVeal) and 0.3% of the highest-performing baseline model (UniXcoder). These results consistently indicate that DeepDFA excels in its efficiency compared to other models. This is possible because DeepDFA is based on the dataflow analysis's compact representation – bitvector, which captures the relevant semantic information in bits and thus is more efficient compared to tokenized strings. DeepDFA propagated information along only the domain-specific CFG edges, rather than associating every pair of tokens in an exhaustive fashion.

DeepDFA's short inference time due to a low number of MAC operations enables its use in non-GPU environments (which are common for software development) where large language models may not be easily deployed. DeepDFA's short training time enables techniques like per-project fine-tuning and hyperparameter tuning, which would be much more costly with the LLMs' training times of over 10 hours. Because of DeepDFA's small parameter count, it is ideal for resource-limited computing platforms such as mobile devices, where large models cannot be used [30].

Efficiency on training data: In Table 3.8, we report the performance of DeepDFA over reduced training dataset sizes, compared to the SOTA models, LineVul and UniXcoder. The columns "#

Table 3.8: Performance of DeepDFA and baselines on limited data. DeepDFA retained its performance on limited data.

			F1		
Portion	**# data**	**# vul**	**LineVul**	**UniXcoder**	**DeepDFA**
0.1%	151	11	29.75	4.36	**55.24**
0.5%	755	56	77.69	**79.37**	68.17
1.0%	1,510	90	**84.62**	79.60	68.40
10.0%	15,091	885	86.67	**92.53**	68.44
100.0%	150,908	8,736	93.23	**95.11**	68.26

data" and "# vul" list the number of training examples in each subset and, of these, the number of vulnerable examples. The results show that DeepDFA maintained a stable performance across small dataset sizes, even with only 0.1% of the training dataset, using only 151 training examples. In contrast, LineVul and UniXcoder steadily dropped in performance as the size of the training dataset decreased. At 0.1% data, LineVul's mean performance was only 29.75 in F1 and UniXcoder's was 4.36.

For project-specific training in applications within a single development team, a model which can learn efficiently from a small dataset is useful.

We believe that our model's stable performance over the reduced dataset and good performance with very small training datasets demonstrate the advantage of the small models and the effectiveness of domain-specific algorithms to guide model learning. DeepDFA is less prone to overfitting to datasets of limited size since it has fewer parameters than LineVul [8]. On the other hand, the transformer models require a large corpus of programs to learn the patterns among the unstructured token data.

RQ2 result: DeepDFA was considerably faster than the baselines; it took 9 minutes to train, 4.64 milliseconds for inference on GPU, and 5.8 milliseconds for inference on CPU. DeepDFA retained stable performance as the training dataset size was reduced. In a low-data scenario, DeepDFA outperformed LineVul and UniXcoder by 25.49 and 50.88 points F1 score.

Table 3.9: How do the models handle unseen projects? Note the performance drop (ΔF1) from the cross-project to mixed-project setting.

Model	Mixed F1	Cross F1	ΔF1
LineVul	84.03	71.37	-12.66
UniXcoder	86.30	76.72	-9.58
DeepDFA	70.49	68.58	**-1.91**
DeepDFA+LineVul	87.89	71.88	-16.02
DeepDFA+UniXcoder	**89.85**	**78.07**	-11.77

3.5.5 Generalization

Cross-project evaluation on Big-Vul: We compared the models' F1 scores on the *cross-project* (shown as Cross F1) and *mixed-project* (shown as Mixed F1) settings to evaluate the models' capabilities of generalizing over unseen projects. Table 3.9 presents the highest-performing baseline models, LineVul and UniXcoder, compared to DeepDFA (the results of the other baseline models are available in the supplementary material, Section 3.F). Among the most important metrics, ΔF1 shows how performance changes when the model is applied to unseen projects. Shown under Column ΔF1, DeepDFA only dropped 1.91 (2.7%) F1 score, compared to 12.66 (15.1%) drop for LineVul and 9.58 (11.1%) drop for UniXcoder. DeepDFA+UniXcoder reported the best performance for both the mixed-project and cross-project settings, improving on UniXcoder's mean F1 score by 3.55 and 1.35 points respectively; DeepDFA+LineVul also improved LineVul's F1 score in both settings.

Applying to DbgBench: In Table 3.10, we report our experience of applying deep learning tools to real-world bug benchmarks. DeepDFA detected 8.7 out of 17 total bugs on average across 3 runs. DeepDFA also correctly predicted non-vulnerable for 3 out of 5 patched programs. On the other hand, neither of the competing LLMs, LineVul and UniXcoder, detected any bugs and in fact both models reported all programs as non-vulnerable with high confidence. This implies that these models were heavily biased to predict all examples in DbgBench as non-vulnerable. With the addition of DeepDFA, DeepDFA+LineVul and DeepDFA+UniXcoder's generalization greatly improved, yet they did not perform as well overall as DeepDFA alone. It should be noted that the

Table 3.10: Generalization performance of DeepDFA and baselines. DeepDFA generalized to real-world bugs in DbgBench. Results are averaged over checkpoints from 3 random seeds. Buggy/-Patched columns show the number of correct predictions on buggy/patched programs respectively.

Model	Buggy	Patched	Accuracy	F1
LineVul	0.0	**5.0**	22.73	0.00
DeepDFA+LineVul	**9.0**	1.3	46.97	60.67
UniXcoder	0.0	5.0	22.73	0.00
DeepDFA+UniXcoder	**9.0**	1.3	46.97	60.67
DeepDFA	8.7	3.0	**53.03**	**64.29**
Total	17.0	5.0	-	-

bugs in DbgBench are very complex and took human experts hours to diagnose [9]. In the past, we have tried a variety of static analysis tools, such as Cppcheck[8] and Polyspace[9], to detect bugs in DbgBench, but we have not detected any of these bugs.

We believe that DeepDFA generalizes better because it does not rely on spurious features that may exist at token and text level, such as variable names and function names, as reported by previous research [13]. These spurious features are no longer correlated with vulnerabilities in unseen projects, as their input tokens will likely change. Our abstract dataflow embedding encodes the usage patterns of commonly used API calls, operators, constants, and data types. Such patterns can be extracted from unseen text and are directly related to the cause of the vulnerabilities, and thus might help DeepDFA generalize better over unseen projects.

RQ3 result: DeepDFA had the smallest drop in F1 score (Δ F1) when applying to the vulnerabilities in the projects that are not seen in training datasets. DeepDFA was able to detect complex bugs in DbgBench and was able to distinguish the buggy and patched versions. The SOTA models, LineVul and UniXcoder, did not detect any bugs in DbgBench.

Table 3.11: Ablation study evaluated on DbgBench.

Feature set	Buggy	Patched	Acc	F1
DeepDFA	**8.7**	**3.0**	**53.03**	**64.29**
API only	7.7	3.0	48.48	57.50
Datatype only	8.0	3.0	50.00	59.26
Literal only	7.7	3.0	48.48	57.50
Operator only	7.7	3.0	48.48	57.50
Api+datatype+literal	8.0	3.0	50.00	59.26
Api+datatype+operator	8.0	3.0	50.00	59.26
Api+literal+operator	8.0	3.0	50.00	59.26
Datatype+literal+operator	8.0	3.0	50.00	59.26
Total	17.0	5.0	-	-

3.5.6 Ablation studies

Table 3.11 shows the model's performance on DbgBench. The model detected the most bugs when using all four features compared to other ablation settings. When using only one feature at a time, the model consistently missed 1-2 bugs which were detected by DeepDFA. When using three features at a time (leaving one out), the model still consistently failed to detect 1 bug which was detected by DeepDFA.

Table 3.12 shows the model's performance on the Big-Vul test dataset. DeepDFA (integrating all the four features) performed the best out of all configurations. When testing one feature at a time, datatype by itself performed better than the other 3 features alone. When we used the combined feature sets, the model performed better than using only one feature.

3.6 Threats to Validity and Discussions

Threats: We evaluated performance primarily on the Big-Vul dataset because this dataset was supported by all the baseline models. Compared to the Devign dataset, Big-Vul is imbalanced and can better reflect a real-world vulnerability detection scenario. However, Big-Vul's data collection

[8] https://cppcheck.sourceforge.io/
[9] https://www.mathworks.com/products/polyspace.html

Table 3.12: Ablation study evaluated on the Big-Vul test dataset.

Feature set	**F1**	**Precision**	**Recall**
DeepDFA	**68.26** (0.16)	53.98 (0.06)	**92.81** (0.40)
Datatype only	68.04 (0.19)	53.83 (0.31)	92.46 (0.30)
Literal only	62.50 (0.81)	50.52 (1.83)	82.46 (7.26)
Operator only	64.47 (0.33)	52.82 (1.59)	82.98 (4.90)
API only	63.67 (0.45)	50.66 (2.36)	86.14 (5.58)
API + datatype + literal	68.18 (0.10)	**54.06** (0.13)	92.28 (0.15)
API + datatype + operator	68.16 (0.13)	54.03 (0.18)	92.28 (0.15)
API + literal + operator	68.11 (0.14)	53.98 (0.20)	92.28 (0.15)
Datatype + literal + operator	68.12 (0.20)	54.04 (0.11)	92.11 (0.46)

process based on bug-fixing commits can introduce label noise and selection bias and as a result, the evaluation could fail to represent real-world performance. To address the selection bias, we studied settings which reflect more realistic scenarios with reduced training datasets and cross-project generalization; to address the label noise, we evaluated the models on additional real-world bugs collected in DbgBench, which were labeled by developers and manually checked.

The performances reported in RQ1, RQ2, and RQ3 will be affected by the random noise in the model training and, for RQ2, dataset selection. To mitigate this effect, we generated 3 versions of the subsets using different random seeds and reported the mean performance.

The mixed-project and cross-project performance reported in RQ3 will be affected by the random selection of projects in the training/held-out datasets. To mitigate this effect, we performed 5-fold cross-validation and reported the mean performance.

Discussions: We believe our approach can be extended to *bit-vector* dataflow problems [29, 45]. All problems in this category contain a finite set of dataflow facts and have the same form of transfer functions and meet operators (see Equations (3.1) and (3.2)). For example, live variables and available expressions [45] are bit-vector problems that are important for vulnerability detection [12]. We believe a new dataflow analysis can be integrated by: (1) defining an abstract dataflow embedding which can capture the dataflow set of the analysis, (2) configuring the neural network used as the aggregate function in GGNN to better simulate the meet operator (based on whether it is a union or intersection operation), and (3) reversing the CFG edges for backward dataflow problems (as reaching definition is a forward dataflow problem).

3.7 Related Work

Many works have used GNN for vulnerability detection [6, 51, 19, 21, 14, 10, 42]. In several recent approaches, Devign [56], ReVeal [13], IVDetect [35], and LineVD [28] used GNN on program graph representations such as AST, CFG, and PDG, and annotated the nodes with unsupervised or pretrained word embeddings. The novelty of our work is a bit-vector inspired abstract dataflow embedding based on the analogy of graph learning and DFA algorithms.

Transformer models such as CodeBERT [23], LineVul [24], and UniXcoder [26] used a token-based program representation pretrained on a large body of NL-PL pairs, and then fine-tuned for vulnerability detection. Using CFGs, our graph learning only propagates the information along semantically important edges instead of trying to learn the relations of each pair of tokens. Thus, our approach is substantially more efficient. Since we have used a semantic-based embedding, we show that we can improve the performance of token based models. The most recent work, ContraFlow [15], learns embeddings of def-use paths (an output of

dataflow analysis), then predicts vulnerability detection using a transformer model. Our work directly emulates dataflow analysis and does not require an expensive pretraining phase.

There were also models that used sequence and CNN architectures. VulDeePecker [38] used BiLSTM on slices considering data dependencies. SySeVR [37] used BiGRU on slices and adds data dependencies. Draper [46] used CNN and Random Forest. However, none of these models integrates dataflow analysis in its algorithm.

Cummins et al. [17] formulated dataflow analyses as supervised learning tasks and applied it for device mapping and algorithm classification; we discuss the differences from our work in-depth in Section 3.3.3. Other relevant work that explores dataflow analysis and deep learning include: (1) VenkataKeerthy et al. [50] used the output of dataflow analysis, reaching definitions and live variables, to learn flow-aware embeddings; and (2) Bielik et al. [7] and Jeon et al. [32] learned static analysis formulas from a dataset based on a fixed language. None of these works aims to develop a model for vulnerability detection.

3.8 Conclusions and Future Work

We propose DeepDFA, an efficient graph learning framework and embedding technique for vulnerability detection. Our *abstract dataflow embedding* leverages the idea of *bit-vector* in dataflow analysis and integrates data usage patterns from semantic features: commonly used API calls, operations, constants, and data types that potentially capture the causes of the vulnerabilities. DeepDFA emulates the Kildall method of dataflow analysis using the analogous message-passing algorithm. Our experimental results show that DeepDFA is very efficient. It is trained in 9 minutes and used only 50 vulnerable examples to achieve its top performance. Yet, it still outperformed all non-transformer baselines and generalized the best among all the models. DeepDFA found bugs in real-world programs from DbgBench while neither of the highest-performing baselines, LineVul and UniXcoder, detected any bugs. Importantly, DeepDFA can be used to improve other models. By combining DeepDFA with the top performing models, we surpassed the state-of-the-art performance for vulnerability detection.

In the future, we plan to incorporate other dataflow analyses, e.g., live variable analysis, that have been used for or vulnerability detection [12]. We also plan to explore the application of explanation tools to precisely pinpoint the vulnerability location at specific lines in the code, and evaluate our framework on detecting vulnerabilities in other programming languages.

3.9 Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This research is partially supported by the U.S. National Science Foundation (NSF) under Awards #1816352 and #2313054.

3.10 Bibliography

[1] Browse vulnerabilities by date. https://web.archive.org/web/20211014235218/https://www.cvedetails.com/browse-by-date.php, 2021. Accessed November 5 2021.

[2] IBM Cost of a Data Breach Report 2021, 2021. Accessed October 29 2021.

[3] Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Online, June 2021), K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds., Association for Computational Linguistics, pp. 2655–2668.

[4] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[5] Allamanis, M. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2019), Onward! 2019, Association for Computing Machinery, p. 143–153.

[6] Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *International Conference on Learning Representations* (2018).

[7] Bielik, P., Raychev, V., and Vechev, M. Learning a static analyzer from data. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30* (2017), Springer, pp. 233–253.

[8] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[9] Böhme, M., Soremekun, E. O., Chattopadhyay, S., Ugherughe, E., and Zeller, A. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2017), ESEC/FSE 2017, pp. 1–11.

[10] Cao, S., Sun, X., Bo, L., Wei, Y., and Li, B. BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology 136*, C (Aug. 2021).

[11] Cao, S., Sun, X., Bo, L., Wu, R., Li, B., and Tao, C. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. *arXiv:2203.02660* (Mar 2022). arXiv: 2203.02660.

[12] Cesare, S. Bugalyze.com - Detecting Bugs Using Decompilation and Data Flow Analysis. In *BlackHat USA* (2013), p. 9.

[13] Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering 48*, 09 (Sept. 2022), 3280–3296.

[14] Cheng, X., Wang, H., Hua, J., Xu, G., and Sui, Y. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering Methodology 30*, 3 (Apr 2021), 38:1–38:33.

[15] Cheng, X., Zhang, G., Wang, H., and Sui, Y. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2022), ISSTA 2022, Association for Computing Machinery, p. 519–531.

[16] Cranmer, M., Sanchez Gonzalez, A., Battaglia, P., Xu, R., Cranmer, K., Spergel, D., and Ho, S. Discovering symbolic models from deep learning with inductive biases. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 17429–17442.

[17] Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., O'Boyle, M. F. P., and Leather, H. ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. In *Proceedings of the 38th International Conference on Machine Learning* (18–24 Jul 2021), vol. 139 of *PMLR '21*, pp. 2244–2253.

[18] Dietterich, T. G. Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. *Neural Computation 10*, 7 (Oct 1998), 1895–1923.

[19] Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *Proceedings of the International Conference on Learning Representations* (2020).

[20] Ding, Y., Suneja, S., Zheng, Y., Laredo, J., Morari, A., Kaiser, G., and Ray, B. VELVET: a novel ensemble learning approach to automatically locate vulnerable statements. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering* (Los Alamitos, CA, USA, Mar 2022), SANER '22, IEEE Computer Society, pp. 959–970.

[21] Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., and Jiang, Y. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering* (2019), ICSE '19, IEEE Press, p. 60–71.

[22] Fan, J., Li, Y., Wang, S., and Nguyen, T. N. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (New York, NY, USA, 2020), MSR '20, Association for Computing Machinery, p. 508–512.

[23] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (Online, Nov. 2020), T. Cohn, Y. He, and Y. Liu, Eds., Association for Computational Linguistics, pp. 1536–1547.

[24] Fu, M., and Tantithamthavorn, C. LineVul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories* (New York, NY, USA, 2022), MSR '22, Association for Computing Machinery, p. 608–620.

[25] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (2017), ICML '17, JMLR.org, p. 1263–1272.

[26] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Dublin, Ireland, May 2022), S. Muresan, P. Nakov, and A. Villavicencio, Eds., Association for Computational Linguistics, pp. 7212–7225.

[27] Hanif, H., and Maffeis, S. VulBERTa: Simplified source code pre-training for vulnerability detection. In *Proceedings of the 2022 International Joint Conference on Neural Networks* (2022), IJCNN '22, pp. 1–8.

[28] Hin, D., Kan, A., Chen, H., and Babar, M. A. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories* (New York, NY, USA, 2022), MSR '22, Association for Computing Machinery, p. 596–607.

[29] Horwitz, S. Dataflow analysis. CS704 Lecture Notes [Accessed 19-09-2023].

[30] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. MobileNets: Efficient convolutional neural networks for mobile vision applications, 2017.

[31] Japkowicz, N. The class imbalance problem: Significance and strategies. In *Proceedings of the International Conference on Artificial Intelligence* (2000), vol. 56, pp. 111–117.

[32] Jeon, M., Jeong, S., Cha, S., and Oh, H. A machine-learning algorithm with disjunctive model for data-driven program analysis. *ACM Transactions Programming Language Systems 41*, 2 (Jun 2019).

[33] Kildall, G. A. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1973), POPL '73, Association for Computing Machinery, p. 194–206.

[34] Le, Q., and Mikolov, T. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning* (Bejing, China, 22–24 Jun 2014), E. P. Xing and T. Jebara, Eds., vol. 32 of *Proceedings of Machine Learning Research*, PMLR, pp. 1188–1196.

[35] Li, Y., Wang, S., and Nguyen, T. N. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2021), ESEC/FSE 2021, Association for Computing Machinery, pp. 292–303.

[36] Li, Y., Zemel, R., Brockschmidt, M., and Tarlow, D. Gated graph sequence neural networks. *Proceedings of the 4th International Conference on Learning Representations*, 1 (2016), 1–20.

[37] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities . *IEEE Transactions on Dependable and Secure Computing 19*, 04 (July 2022), 2244–2258.

[38] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the Network and Distributed System Security Symposium* (2018), NDSS '18, Internet Society.

[39] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *CoRR abs/2102.04664* (2021).

[40] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. arXiv: 1301.3781.

[41] Moshtari, S., Okutan, A., and Mirakhorli, M. A grounded theory based approach to characterize software attack surfaces. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA, 2022), ICSE '22, Association for Computing Machinery, p. 13–24.

[42] Nguyen, V.-A., Nguyen, D. Q., Nguyen, V., Le, T., Tran, Q. H., and Phung, D. ReGVD: Revisiting graph neural networks for vulnerability detection. In *Deep Learning for Code Workshop* (2022).

[43] Pennington, J., Socher, R., and Manning, C. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Doha, Qatar, 2014), Association for Computational Linguistics, pp. 1532–1543.

[44] Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (New York, NY, USA, 2020), KDD '20, Association for Computing Machinery, p. 3505–3506.

[45] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, Jan. 1995), POPL '95, Association for Computing Machinery, pp. 49–61.

[46] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. Automated vulnerability detection in source code using deep representation learning. *17th IEEE International Conference on Machine Learning and Applications* (2018), 757–762. Publisher: IEEE.

[47] Seabold, S., and Perktold, J. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference* (2010).

[48] Steenhoek, B., Rahman, M. M., Jiles, R., and Le, W. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 2237–2248.

[49] Tan, M., and Le, Q. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning* (09–15 Jun 2019), vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 6105–6114.

[50] VenkataKeerthy, S., Aggarwal, R., Jain, S., Desarkar, M. S., Upadrasta, R., and Srikant, Y. N. IR2VEC: LLVM IR based scalable program embeddings. *ACM Transactions on Architectural Code Optimization 17*, 4 (Dec 2020).

[51] Wang, Y., Gao, F., Wang, L., and Wang, K. Learning a Static Bug Finder from Data. *arXiv:1907.05579* (Mar 2020). arXiv: 1907.05579.

[52] Wikipedia. List of data breaches. https://web.archive.org/web/20211011144237/https://en.wikipedia.org/wiki/List_of_data_breaches, Oct 2021. Accessed October 29 2021.

[53] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations* (2019).

[54] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and discovering vulnerabilities with code property graphs. *Proceedings of the IEEE Symposium on Security and Privacy* (2014), 590–604.

[55] Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., and Su, Z. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA, 2021), ICSE-SEIP '21, Association for Computing Machinery.

[56] Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems 32* (2019).

3.A Appendix: Baseline reproductions

- We could not reproduce VulDeePecker, SySeVR, Draper, or IVDetect, so we repeated the performances reported in Li et al. [35]. Their measurements may vary slightly from our reproduction.
- We confirmed with Chakraborthy et al. [13] that our results fixed a data leakage bug in the original implementation, so the results we report may differ from the original paper.
- Zhou et al. [56] did not release their model code, so we reproduced Devign from the third-party implementation released by Chakraborthy et al. [13] (https://github.com/saikat107/Devign).
- Hin et al. [28] did not report function-level metrics for LineVD, and we could not reproduce their statement-level performance from their model code, so we did not compare with their approach.
- Cheng et al. [15] did not release their model code and evaluated a different dataset than ours, so we did not compare with their approach.

3.B Appendix: Programs that are removed

We excluded a total of 1,564 programs (0.8%) of the Big-Vul dataset, following the implementation of LineVD[10]. Specifically, the following programs are removed:

- Incomplete functions, i.e., ending with '`);`', or not ending in '`}`' or no '`;`'. Joern can fail to parse such functions.
- Programs where no lines were added or removed, but the function was labeled vulnerable.
- Vulnerable programs where more than 70% of the lines are modified from the vulnerable to the fixed version, indicating that a substantial change has been done and the vulnerability may be changed.

[10] https://github.com/davidhin/linevd

- Programs that are fewer than 5 lines long.

3.C Appendix: Additional effectiveness results

Of the transformer models, we evaluated CodeBERT and LineVul in our experiment for Section 3.5.3. Table 3.13 reports the performances of the other models.

Table 3.13: Initial trial run of performance on 100% of the Big-Vul dataset.

Model	Model type	F1	Precision	Recall
Devign	GNN	29.33 (6.58)	32.83 (5.55)	26.59 (7.25)
ReVeal	GNN	33.60 (0.69)	33.08 (3.49)	34.67 (3.80)
ReGVD	GNN	24.49 (3.16)	63.76 (3.54)	15.26 (2.71)
CodeBERT	Transformer	22.68 (8.12)	67.79 (4.90)	19.11 (3.39)
LineVul	Transformer	**91.58** (0.49)	**95.99** (0.85)	**87.55** (0.49)
VulBERTaMLP [27]	Transformer	1.75 (3.03)	19.33 (33.49)	0.92 (1.59)
VulBERTaCNN [27]	Transformer	10.59 (0.00)	5.59 (0.00)	100.00 (0.00)
PLBART [3]	Transformer	25.35 (3.74)	61.84 (6.54)	16.18 (3.52)

3.D Appendix: Training times of all models

Table 3.14 lists the training times of the lower-performing models which we did not report in Section 3.5.4. These training runs were not performed in the same environment as those in Section 3.5.4, but they provide an approximate measurement of the training time. We were not able to reproduce IVDetect, so we do not report its training time.

Table 3.14: Approximate training times of all models.

Model	**Training time**
Devign	2h58m
ReVeal	13h20m
ReGVD	5h33m
CodeBERT	7h33m
LineVul	10h19m
DeepDFA+LineVul	10h40m
UniXcoder	11h16m
DeepDFA+UniXcoder	13h22m
CodeT5	27h40m
DeepDFA+CodeT5	27h57m
DeepDFA	**9m**

3.E Appendix: Model sizes

Table 3.15 lists the size of each model.

Table 3.15: DeepDFA was smallest in terms of parameter count.

Model	**# parameters**
IVDetect	924,165
Devign	1,148,553
ReVeal	560,291
ReGVD	124,794,500
CodeBERT	124,646,401
LineVul	125,238,531
DeepDFA+LineVul	125,679,236
UniXcoder	126,522,627
DeepDFA+UniXcoder	126,963,332
CodeT5	222,883,586
DeepDFA+CodeT5	223,128,195
DeepDFA	**375,938**

3.F Appendix: Additional cross-project evaluation results

We only evaluated LineVul in our experiment for Section 3.5.5 because its absolute performance was substantially better than the other baseline models in our initial trial. Table 3.16 reports the results of our initial trial.

Table 3.16: Initial trial run of cross-project evaluation with 100% of the dataset.

Model	**Mixed-project F1**	**Cross-project F1**	**ΔF1**
LineVul	**84.07**	**71.81**	-12.26
Devign	24.32	14.26	**-10.06**
ReVeal	28.24	6.83	-21.41
ReGVD	33.61	21.70	-11.91
CodeBERT	24.14	4.98	-19.16

CHAPTER 4. TRACED: EXECUTION-AWARE PRE-TRAINING FOR SOURCE CODE

Yangruibo Ding[1], Benjamin Steenhoek[2], Kexin Pei[3], Gail Kaiser[4], Wei Le[5], and Baishakhi Ray[6]

[1,3,4,6] Department of Computer Science, Columbia University, New York, NY, 10027

[2,5] Department of Computer Science, Iowa State University, Ames, IA, 50011

Modified from a manuscript published in the *46th International Conference on Software Engineering (ICSE 2024)*

Abstract

Most existing pre-trained language models for source code focus on learning the static code text, typically augmented with static code structures (abstract syntax tree, dependency graphs, *etc.*). However, program semantics will not be fully exposed before the real execution. Without an understanding of the program execution, statically pre-trained models fail to comprehensively capture the dynamic code properties, such as the branch coverage and the runtime variable values, and they are consequently less effective at code understanding tasks, such as retrieving semantic clones and detecting software vulnerabilities.

To close the gap between the static nature of language models and the dynamic characteristics of programs, we introduce TRACED, an execution-aware pre-training strategy for source code. Specifically, we pre-train code language models with a combination of source code, executable inputs, and corresponding execution traces. Our goal is to teach code models the complicated execution logic during the pre-training, enabling the model to statically *estimate* the dynamic code properties without repeatedly executing code during task-specific fine-tuning.

*Benjamin led data collection of over 100k execution traces; Yangruibo led model design/evaluation.

To illustrate the effectiveness of our proposed approach, we fine-tune and evaluate TRACED on three downstream tasks: static execution estimation, clone retrieval, and vulnerability detection. The empirical results show that TRACED relatively improves the statically pre-trained code models by 12.4% for complete execution path prediction and by 25.2% for runtime variable value predictions. TRACED also significantly outperforms statically pre-trained models in clone retrieval and vulnerability detection across four public benchmarks. Our data, code, and pre-trained models are available at https://doi.org/10.6084/m9.figshare.27367989.

4.1 Introduction

Machine Learning (ML) for source code has enabled many software engineering tasks, such as automated program repair [12, 20, 18, 19], bug finding [8, 56], and refactoring [7]. Recently, the common practice of training ML models for source code understanding is based on pre-training a Transformer-based language model on source code. These approaches treat source code programs as *static text* [13, 6, 1, 48], sometimes augmented with program-specific structures such as abstract syntax trees and dependency graphs [15, 14, 11, 33], and adapt pre-training strategies for natural language to learn program representations.

However, many source code understanding tasks require a more comprehensive understanding of *program behavior*. For instance, detecting semantic clones [30] involves determining if two pieces of code behave similarly under similar inputs, even if their structures are apparently different. Likewise, detecting vulnerabilities often requires developers to analyze whether a potentially problematic location can be executed and what kinds of value flows can expose any vulnerability. While existing code models are primarily trained to capture static code properties, they are not effective at reasoning about program behavior. In fact, many of the deeper program semantics only manifest when the code is executed. As a result, they tend to underperform when it comes to tasks that require deeper semantic understanding.

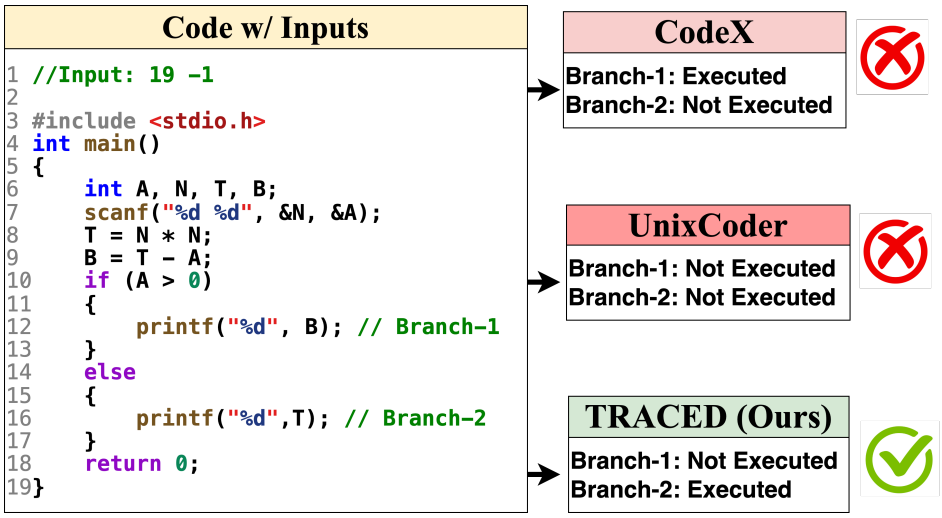


Figure 4.1: A motivating example from CodeNet's coding challenge No.3597 [40] reveals that statically pre-trained code language models, regardless of their size, could not reason about the branch coverage given a specific input, while TRACED, enhanced with program execution features, correctly identify the execution path.

Motivating Examples. Figure 4.1 presents an example with simple execution logic to illustrate the failure of statically pre-trained code models on the branch coverage prediction. We query three pre-trained code models, CodeX [9] (code-davinci-002), UnixCoder [14], and TRACED (ours), to predict the branch coverage, according to the given program inputs. For CodeX, we prompt the model with carefully designed questions, similar to [34], to ask for the branch coverage prediction in the zero-shot setting. Specifically, we augment the prompts by adding comments at the end of lines 12 and 16: `// Will this line be executed? Yes or no?`. To give more hints regarding the data flow, we further add a comment at the end of line 10: `// A is -1, since it accepts the second value of the input`. Unfortunately, even if provided with additional hints of the required data flow for branch prediction, CodeX still failed to predict the correct coverage labels, suggesting it cannot interpret this simple execution.

Besides the zero-shot prompting, we also study whether fine-tuning pre-trained code models to predict execution can lead to better branch prediction. Specifically, we fine-tune another popular pre-trained code model, UnixCoder [14], to predict branch execution while ensuring the motivation example is not seen during training. From the inference results in Figure 4.1, we notice

that UnixCoder cannot predict covered branches even after being fine-tuned. It predicts neither of the branches will be covered, indicating that it does not have the basic understanding that, for this specific example, at least one branch will always be taken on a valid input.

Our approach. To address the limitation of the statically pre-trained code models, we propose TRACED, an execution-aware pre-training strategy to capture the static and dynamic perspectives of the source code. Specifically, we pre-train the Transformer-based language model with multi-task objectives on predicting source code, program states, and execution coverage, forcing the model to reason about both program's runtime behavior and the naturalness of the source code [42] at the same time. We address several technical challenges, such as representing program execution states, encoding the runtime variable values, and representing code coverage, to implement the pre-training strategy.

Representing Program States. During program execution, variables are used to store data that is used by the program. These variables can have different types, such as integers, floating-point numbers, pointers, and arrays. As the program executes, the values of these variables change, reflecting the changes in the program's state. Consequently, software developers typically monitor the variable values, via debugging tools, to observe the execution facts [54] and understand the dynamic behaviors of the program.

In this work, we define the *program state* at a specific time step of the execution as the set of values of every defined variable in the current scope. In other words, the program state is equivalent to the value mapping table of the debugger, which is monitored by the developer when the program is paused by a specific breakpoint.

Value Quantization. While the runtime variable values are traced as concrete values, directly learning them brought challenges to machine learning models. Concrete values span over a wide range of possible values, especially when considering different data types (integers, floating-point numbers, arrays, pointers, etc.), leading to a high-dimensional, complex, but sparse

data distribution. This increased data complexity and sparsity challenges the model to learn patterns and relationships between the variable values, as it must deal with many unique inputs, which causes the model to overfit and memorize specific instances rather than generalize to broader patterns. Additionally, noise, outliers, and irregularities of concrete values also mislead the model's learning process. We will empirically demonstrate these limitations in Section 4.6.3.

To decrease the data complexity and increase the density, we define thirty value categories, covering a wide range of variable types, to map the continuous but sparse variable values into discrete bins. We call this process as *value quantization*, which is similar in design to the quantization in signal processing[1]. This simplification potentially helps the model to be more resilient to noise and outliers, allowing it to focus on learning the underlying execution patterns and relationships between variables, rather than being sensitive to specific instances or irregularities.

Representing Execution Coverage. While program state labels provide important information about the current state of the program, they do not capture information about how the program arrived at that state. To boost the training with more comprehensive execution features, besides the variable values, we also log the execution coverage during the execution, in terms of which lines are executed and which are not, and construct execution coverage features for the model to learn.

Results. We fine-tune and evaluate TRACED's performance using three tasks: static execution estimation, clone retrieval, and vulnerability detection. On statically predicting the program executions, TRACED substantially improves the statically pre-trained code models by 12.4% for execution path prediction and by 25.2% for runtime variable value predictions. TRACED also obtains state-of-the-art results in code understanding tasks: TRACED reports 91.2% MAP@R on CodeXGLUE-POJ104 [30], 50.4% F1 on ReVeal [8], and 65.9% accuracy on CodeXGLUE-defect-detection [30].

[1] https://en.wikipedia.org/wiki/Quantization_(signal_processing)

Contributions. We make the following contributions:

- We present a simplified and compact representation of program executions, including the program states and the execution coverage, to effectively guide code models to learn program semantics and reason about program behavior.
- We propose a novel multi-task pre-training strategy to jointly learn the static and dynamic code properties. As a result, the pre-trained model with our approach will be empowered with a decent execution awareness.
- We pre-train TRACED with the proposed trace representation and the execution-aware strategy and evaluate its performance on several downstream tasks. The experiment results demonstrate that TRACED significantly outperforms the statically pre-trained code models in these tasks.
- We publicly released our data, code, and pre-trained models at https://doi.org/10.6084/m9.figshare.27367989.

4.2 Overview

Figure 4.2 shows the overview of TRACED, consisting of three main stages: (1) tracing the source code and engineering the features, (2) execution-aware pre-training using the program traces, and (3) loading the pre-trained weights and performing task-specific fine-tuning.

Stage-1: Tracing & Feature Engineering. The goal of this stage is to prepare the data for pre-training. The process begins with providing a source program and its executable inputs. The first step is to execute the program with each input to generate corresponding traces. The traces record the runtime variable values, together with the execution coverage, logging the full execution history of the program and revealing the changes to program states throughout execution. To reduce the complexity and sparsity of the data, and make it easier for the model to learn patterns and relationships between the variable values, we quantize the concrete runtime

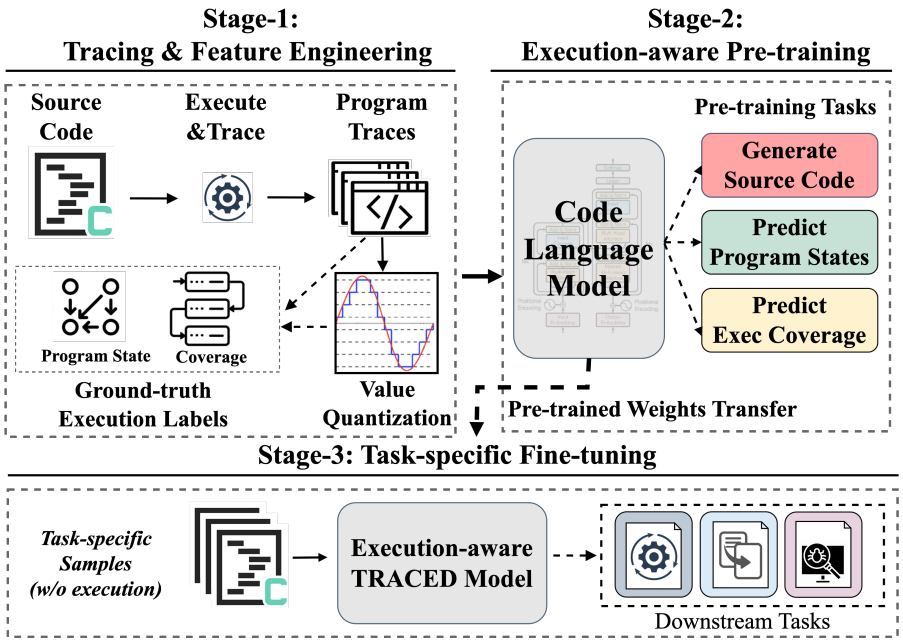


Figure 4.2: Overview of the workflow of TRACED.

values recorded in the traces into pre-defined value ranges. The quantization process maps continuous values to a fixed set of discrete or bins. By quantizing the values, we create a finite set of possible outputs that can be used as ground-truth labels during training. After quantization, we create program state labels and execution coverage labels that will help the model to capture the program executions. The dataset finally ends up with a set of samples and labels, where each sample includes the source code with its program input and the labels represent the execution trace of this sample.

Stage-2: Execution-aware Pre-training with Traces. We utilize the pre-processed samples and labels obtained from Stage-1 to perform supervised pre-training. Specifically, we use a Transformer-encoder-based model [29] to learn the program traces and improve the model's understanding of program execution. The model could be either trained from scratch or loaded by the pre-trained weights of existing code language models. To achieve the goal of producing execution-aware code representation, we propose three pre-training objectives. The first objective is learning to generate the source code. We believe that understanding the naturalness of code

text [17, 42] is fundamental for the model to capture more sophisticated signals such as program execution. This objective is implemented with masked language modeling (MLM), which masks a certain percentage of tokens in the source code and trains the model to reconstruct the masked tokens based on the surrounding context. The second objective is learning to predict the program states. By predicting program state labels that were generated in Stage-1, the model learns to capture the data flows and the side effects of code execution. The third objective is to predict the execution coverage. By predicting the execution coverage labels generated by Stage-1, the model learns to capture the dynamic control flow and helps the model understand how the program state is reached and evolving.

Stage-3: Task-specific Fine-tuning. Finally, we apply TRACED to several downstream tasks. We load the pre-trained weights of TRACED, fine-tune the model for a specific task, and keep updating the model weights. Fine-tuning does not require the program to be executed; rather, TRACED will reason about the execution statically with its learned execution signals during the pre-training, and learn to accomplish the task accordingly. In many useful applications, we would not have program traces available. We consider three downstream tasks for TRACED: static execution estimation, which includes execution coverage and runtime variable value predictions, clone retrieval, and vulnerability detection.

4.3 Tracing & Feature Engineering

In this section, we introduce how TRACED builds the learnable features from program traces for models to learn the program executions.

4.3.1 Representing Program States

To imitate the way that human developers monitor variable values to understand program behavior, we propose to train neural models with the log of runtime variable values to recognize execution patterns and infer dynamic program behaviors in a way that is similar to human

```
1  // INPUT: 4
2  int factorial() {
3    int x, y; // {'x': 32767, 'y': 32767}
4    x = atoi(argv[1]); // {'x': 4, 'y': 32767}
5    if (x < 0) { // {'x': 4, 'y': 32767}
6      y = −1;
7      return y;
8    }
9    y = 1; // {'x': 4, 'y': 1}
10   for (int i = 1; i <= x; i++) // {'x': 4, 'y': 24, 'i': 5}
11   {
12     y *= i; // {'x': 4, 'y': 24, 'i': 5}
13   }
14   return y; // {'x': 4, 'y': 24, 'i': 5}
15 }
```

Figure 4.3: Program states with concrete runtime values.

intuition. By taking the log of variable values during the execution, we can represent the program states in a more compact and interpretable form that is manageable for deep neural nets to process.

We build the program state by taking snapshots of variable values at program points during execution. When we take a snapshot at a specific time step, similar to the moment that the program is paused by a debugging breakpoint set right after line l, we maintain a value mapping, M, to map the variable to its current value, similar to the value mapping table of the debugger. To record the program state, we take the value snapshot after each line of execution and log the variables' current values.

Definition: *Program State.* Formally, we define the program state after the execution of a specific line, l, as $s(l)$, represented as a set of variable values at this moment:

$$s(l) = \{M(v, l) \mid v \in V,\ l \in L\}$$

V represents the set of all traced variables, and L is the set of lines with source code. Figure 4.3 shows an illustrative example of a simple factorial program and the comments after the source code indicate the program state after the execution of that line. Also, we do not log the program state for lines without executable code, such as line-8 of Figure 4.3.

Note that a source code line could be executed multiple times due to a loop or recursion. While a more detailed representation of program execution might provide additional insights, it also increases the complexity and computational requirements of the model. As a trade-off between the complexity and performance, we use the last occurring execution of each line to finalize the program states, so that $s(l)$ keeps getting updated until the execution terminates.

We apply such a trade-off based on the observations of real executions. Specifically, the last occurring values are typically sufficient to capture the results of loops and recursions. For example, when calling a recursive function, only the last occurring value(s) of returned variable(s) will be taken to fulfill the following execution of the caller. Similarly, the final values when loops finish will take part in the future execution. As shown in line-12 of Figure 4.3, variable `y` gets multiplied inside a loop to calculate the factorial. Its value changes in each iteration, but it is less informative to reason about the program's overall behavior, as only the final value is used as the return value (line-14). Thus, we would represent `y` using the value from the last occurring execution of the loop.

4.3.2 Quantized Variable Values

As we introduced in Section 4.1, the distribution of concrete values is sparse and complex, consequently difficult for a statistical model to fit. In addition, concrete values are not always necessary. Some common program behaviors are accompanied by extremely large or small variable values – for example, in C, uninitialized variables are often set to zero or uncommonly large variables, but the concrete values are not meaningful because they depend only on the data remaining on the stack, which could be randomly large or small. The model could represent such behaviors by estimating the value ranges of variables without accurately predicting their concrete values which are not informative or meaningful. Figure 4.3 displays some of these cases: after the execution of line-3, `x` and `y` are uninitialized and randomly initiated as 32,767, which has no concrete meaning but only makes the training data noisy and sparse.

Table 4.1: TRACED's design of quantized variable values.

Data Type	Value Types	Concrete Value	Quantized Value
Basic	Integer	$0 < v \leq 10,000$	Positive Regular
		$10,000 < v$	Positive Large
		0	Zero
		$-10,000 \leq v < 0$	Negative Regular
		$v < -10,000$	Negative Large
	Float/Double	$0.0 < v \leq 1.0$	Positive Small
		$1.0 < v \leq 10,000.0$	Positive Regular
		$10,000.0 < v$	Positive Large
		0.0	Zero
		$-1.0 < v < 0$	Negative Small
		$-10,000.0 \leq v < -1.0$	Negative Regular
		$v < -10,000.0$	Negative Large
	Character	'\0'	Null
		$v \in$ {a-zA-Z}	Alphabetic
		$v \neq$ '\0'; $v \notin$ {a-zA-Z}	Non-alphabetic
	Boolean	0	False
		1	True
	Void	-	Void
Array	Integer	$[v_1, v_2, \ldots, v_n]$; $quantize(v_i) \in$ Integer	Initialized
			Not Initialized
	Float/Double	$[v_1, v_2, \ldots, v_n]$; $quantize(v_i) \in$ Float/Double	Initialized
			Not Initialized
	Character	"⟨string⟩"	Initialized
			Not Initialized
Pointer	Integer	`0x0`	Null
		Not `0x0`	Not Null
	Float/Double	`0x0`	Null
		Not `0x0`	Not Null
	Character	`0x0`	Null
		Not `0x0`	Not Null

To reduce the data complexity and increase the density, we define 30 categories for quantized values in Table 4.1. To comprehensively represent the variable values, the proposed quantized categories consider both types, *i.e.,* the data types and value types, that are statically defined, and the dynamic runtime values. Our quantized categories cover the most common variable types and value types, which we have found sufficient to capture important program execution behaviors and relationships. By focusing on the most frequent value types, we can capture the essential features of program execution. This makes our approach effective at capturing the generalized program execution behaviors and patterns. We empirically illustrate our quantization strategy's effectiveness in Section 4.6.3.

4.3.3 Building Learnable Labels for Code Models

We used supervised pre-training with traces. We construct labels for code models to learn two main perspectives of execution: program states and execution coverage.

Program State Labels. As we discussed in previous sections, we first trace the program variables during execution and log their runtime values. We then quantize these values into pre-defined categories. This process results in a sequence of program states, each represented by a set of quantized variable values (as shown in Figure 4.3), and we build the learnable features for the code model on top of these program states. Specifically, we build labels for variables that can be quantized into Table 4.1's categories and train the model to predict these labels given their source code representations (Section 4.4.1.2). The label for each variable is represented as a tuple: *(data type, value type, quantized value)*. For example, in Figure 4.3, the label of variable `X` occurring at line-3 is *(Basic, Integer, Positive Large)*, as the current value of `X` is 32,767. We build such labels for all occurrences of valid variables that can be quantized, and the set combining all labels is considered as the program state labels of the code sample.

Execution Coverage Labels. To unify our design and reduce the complexity of the model's learning process, we also build execution coverage labels for each occurrence of variables,

aligning with the program state labels. Concretely, we specify whether a variable is covered or not. The variables within the executed lines will be regarded as covered, and those within the unexecuted lines will be labeled as not covered by execution. For example, in Figure 4.3, Line-6 is not executed, so `y` at this line has the program state label of (Basic, Integer, Not Covered), while `y` at line-9 is executed and has the program state label with concrete quantized value of (Basic, Integer, Positive Regular).

4.4 Model

In this section, we explain the details of TRACED's components and learning objectives during pre-training and fine-tuning.

Model Architecture. Figure 4.4 shows the high-level architecture of TRACED's pre-training. The backbone of TRACED is a 12-layer Transformer encoder, similar to BERT [10] and RoBERTa [29], which learns the generic code representations. On top of the backbone Transformer layers, TRACED stacks multiple multi-layer-perceptron (MLP) layers as prediction heads for different tasks. During the pre-training, as shown in Figure 4.4, TRACED applies a language model prediction head, *i.e.,* LM layer, to predict the masked token given its contextualized representation, a program state prediction head to predict the program states labels that we defined in Section 4.3.3, and an execution coverage head to prediction the execution coverage labels. For the task-specific fine-tuning, the backbone Transformer layers are loaded with the pre-trained weights, while the prediction heads are replaced by a newly initialized head customized for the specific downstream task.

4.4.1 Execution-aware Pre-training

4.4.1.1 Model Input of Pre-training

Each pre-training sample includes the source code of an executable program and a valid executable input. As shown in Figure 4.4, the executable input and the source code are flattened

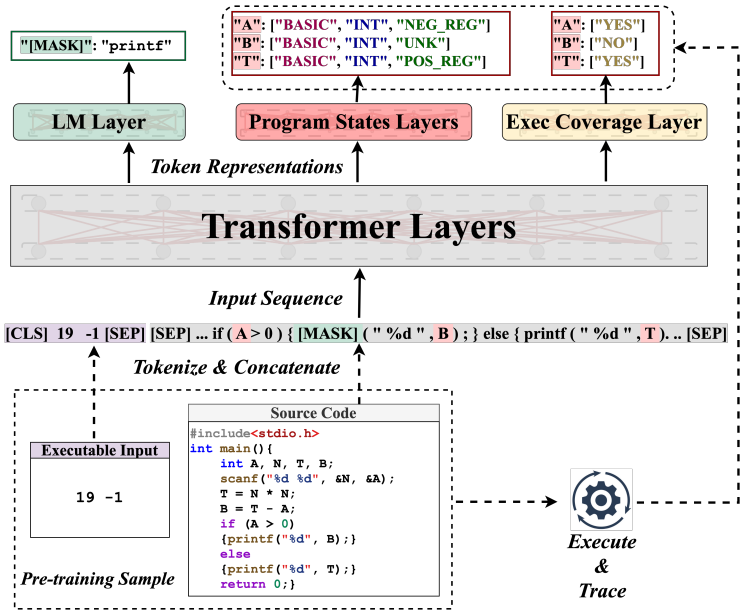


Figure 4.4: High-level model architecture of TRACED. In the labels for program state layers, NEG_REG means "Negative Regular", UNK means "Unknown", and POS_REG means "Positive Regular", which we have defined in Table 4.1.

and concatenated as one sequence. To distinguish the input from the source code, as they are different modalities, TRACED uses special `[SEP]` tokens to separate them and indicate individual positions. To alleviate the out-of-vocabulary concern of programming languages [22], TRACED takes a pre-trained SentencePiece [25] subword tokenizer with vocabulary size of 50,000. It uses this tokenizer to divide the concatenated sequence into a new sequence of sub-tokens.

Formally, we define the executable inputs as $E = \{e_1, ..., e_i\}$ and flattened source code as $C = \{c_1, ..., c_j\}$, then the final model input will be $\mathcal{I} = \{\texttt{[CLS]}, e_1, ..., e_i, \texttt{[SEP]}, \texttt{[SEP]}, c_1, ..., c_j, \texttt{[SEP]}\}$. TRACED truncates the executable inputs and the source code separately if they are too long. TRACED sets the maximum length of the executable input sequence to 64 tokens, and the source code to 960 tokens. These numbers are selected based on the statistics of executable inputs' length of our pre-training dataset (Section 4.5.2.1), and fit the rest of the model input with source code.

Note that the execution traces are not part of the model input, but are used as ground truth labels for the model to predict during pre-training.

4.4.1.2 Learning Execution-aware Code Representations with Traces

TRACED is pre-trained with multiple objectives to jointly capture the static and dynamic perspectives of the source code.

Learning Code Text. Learning code text is the essential first step toward understanding the execution of a program, as code text is the primary source of capturing the code naturalness [17] and other static properties. We implement the code text learning objective by adapting the masked language model objective [29, 10, 13]. Specifically, given the model input sequence, $\mathcal{I}$, TRACED randomly chooses 15% of tokens [29, 10] only from the source code sequence C part and replaces with the special `[MASK]` token (*e.g.,* `printf` in Figure 4.4 is masked). It leaves the executable input sequence E as is. The model is trained to encode the context of `[MASK]` into its code representation, r_{masked}, and reconstruct the concrete masked tokens conditioned on the representation. We represent the loss of learning code text as:

$$\mathcal{L}_{code-text} = \sum_{masked} -logP(c_{masked} \mid r_{masked}) \tag{4.1}$$

In Figure 4.4, the LM (Language Model) layer receives the masked token representation generated by the last Transformer layer. The LM layer then predicts the concrete tokens by mapping the token representation to the probability of each token in the vocabulary, using an MLP (Multi-Layer Perceptron) layer. This process can be thought of as a classification task, where the number of classes is equal to the size of the vocabulary. The goal is to learn a mapping from the masked token representation to the most probable token in the vocabulary, given its context.

Learning Program States. The second pre-training objective, program state prediction (PSP), is designed to enable the model to learn program execution behavior by predicting the program state labels of the traced variables. These program state labels, as defined in Section 4.3.3, contain information about the data types, value types, and quantized values of the variables at the end of the program execution. Specifically, TRACED first identifies the variable

tokens in the source code sequence, denoted as $\{c_{var} \mid c_{var} \in V\} \subseteq C$, where V is the set of all traced variables and C is the source code sequence. It then extracts the representation, r_{var}, of each variable token and feeds it into the program state layer. The program state layer predicts the variable's joint likelihood of being the ground-truth data type, d_{var}, value type, t_{var}, and quantized value, q_{var}. Note that if a variable is tokenized as multiple sub-tokens, all belonging sub-tokens share the same program state label. Finally, TRACED computes the loss of PSP as the sum of the losses of all variable tokens used for predicting their program states. Mathematically, the loss is expressed as follows:

$$\mathcal{L}_{program-state} = \sum_{var} -logP(\ d_{var}, t_{var}, q_{var} \mid r_{var}) \tag{4.2}$$

Learning Variable Coverage. The third pre-training objective, variable coverage prediction (VCP), aims to learn the execution coverage, which is crucial for understanding the control flow of the code given a specific input. Similar to the PSP objective, VCP targets making predictions for variable tokens. Also, sub-tokens belonging to the same variable will be assigned the same coverage label. The loss of VCP is as follows:

$$\mathcal{L}_{var-cov} = \sum_{var} -logP(cov_{var} \mid r_{var}) \tag{4.3}$$

For the efficiency of joint optimization, we share the weights between the program state layers and the execution coverage layers, as they are instinctively both classifiers optimized by cross-entropy loss. Concretely, the coverage label will be learned jointly with the quantized value: if a variable is covered, it will be assigned a specific quantized value label, and otherwise, it will be assigned as "Not Covered".

Finally, TRACED combines the losses of all three objectives and computes their sum as the final loss of a pre-training sample. It back-propagates the gradients through both the prediction layers and the backbone Transformer layers to update their weights. We denote the full set of TRACED's learnable parameters as θ and represent the loss as follows:

$$\mathcal{L}(\theta) = \mathcal{L}_{code-text}(\theta) + \mathcal{L}_{program-state}(\theta) + \mathcal{L}_{var-cov}(\theta) \tag{4.4}$$

4.4.2 Task-specific Fine-tuning

TRACED loads the model weights of Transformers layers, which are pre-trained to produce execution-aware code representations, and further fine-tunes the model for downstream tasks. We consider three downstream tasks as the main applications for TRACED: (1) Static estimation of program execution which includes both execution coverage prediction and runtime variable value prediction; (2) Semantic Clone Retrieval; (3) Vulnerability Detection.

Static Execution Estimation. Our goal of pre-training is to encode the execution patterns into the code representation, so the model could estimate the program execution statically. As a direct application, TRACED fine-tunes the model to predict (1) the execution coverage and (2) runtime variable values using source code and program input. TRACED evaluates the fine-tuned model to estimate the execution of unseen programs in the same way.

Specifically, for execution coverage prediction, TRACED identifies all the branching statements to locate the branches, $B = \{b_1, b_2, ..., b_m\}$, within the source code. It trains the model to predict a binary label, 0 means the branch is not covered by the current execution and 1 means covered, for each $b_i \in B$. For the model's convenience to make predictions, the special token `[MASK]` is inserted at the beginning of each branch. For example, the following `if-else` has two branches that are pre-processed for branch prediction: `if (condition) {[MASK] ...} else {[MASK] ... }`. During the fine-tuning, the Transformer layers learn to encode the branch information into the corresponding `[MASK]` token representation with the built-in bi-directional attention and positional encoding. Then the classification head takes `[MASK]` representations to predict whether a branch is covered by the current execution. For variable value prediction, TRACED identifies variables, $V = \{v_1, v_2, ..., v_n\}$ and trains the model to predict their quantized values (Section 4.3.2) during the execution.

Semantic Clone Retrieval. Detecting semantic clones is significant for software maintenance [23, 28], yet very challenging in practice since the token and syntactic structures overlap among semantic clones may be quite limited. This task requires the model to estimate the program behaviors without executing the programs and capture the similarity among them. It evaluates the model's semantic reasoning capacity to identify the code similarity and retrieve clones: given a program as a query, and an arbitrary collection of programs as candidates, the model needs to identify the query's semantic clones from possibly thousands of candidates.

Vulnerability Detection. Vulnerability detection is a crucial task in software security, aiming to identify potential security vulnerabilities in software code that could be exploited by attackers. The vulnerabilities may exist due to various reasons, including programming errors, design flaws, or configuration issues. Detecting these vulnerabilities early in the software development lifecycle can prevent potential attacks, mitigate risks, and save resources. We fine-tune TRACED's pre-trained model on datasets consisting of vulnerable and non-vulnerable code samples, so the model learns to classify code functions as vulnerable or non-vulnerable by estimating their execution behavior.

4.5 Experimental Setup

4.5.1 Trace Collection

In this section, we explain how we traced the dynamic information in programs to produce concrete traces, given the source code and program input.

First, we compile the program using `gcc` with the options `-g -O0`. Option -g preserves debug information, which is necessary in order to read variables and source code locations using the debugger, and option `-O0` disables compiler optimizations, which could optimize out some variables thus preventing them from being read at runtime. We use this option because we seek to model the semantics of the *source code* in terms of variable values rather than the optimized machine code.

Second, we load the program with the given standard input redirected to `stdin` and attach the `gdb`[2] debugger, using the Python API to implement the tracing command. Starting from the entry point (`main`), we execute the program one line at a time using the `step` command. At each line, we print out the concrete values of all variables in scope. We also set breakpoints at the entry of each user-defined function, where we log the values of each parameter. For numeric types, we simply log their string representation. For `char` and `char *` (string) types, we log the human-readable values of the chars/strings. We use gdb's pretty-printer to print `struct` types and statically allocated array types, such as `int[<size>]`. For pointer types, we print the memory address of the pointer as a hex code. We only traced the functions that were defined in the source code and skipped over all standard library functions.

4.5.2 Dataset

4.5.2.1 Pre-training Dataset

IBM's CodeNet Dataset [40] includes 4,053 programming challenges for several programming languages from the AIZU Online Judge and AtCoder platforms, and each problem has up to thousands of implementations submitted by distinct programmers. In this work, we focus on the C language as the main resource for the pre-training and downstream tasks, so we build our pre-training dataset with programming challenges that have C solutions. Besides the large number of samples and the complexity of programming challenges, we choose CodeNet to build our datasets as it maintains at least one and at most twenty executable inputs for each challenge, so we could execute and trace the implementations of the challenge, and consequently build our execution labels for the model to learn.

Out of 1,900 programming challenges with C solutions, we select 1,805 of them to build the pre-training dataset and leave the other 95 problems as held-out problems for evaluating the model's capacity for the downstream static execution estimation task. Splitting samples strictly by challenge effectively avoids the issue of data leakage from the training set to the held-out set.

[2] https://www.sourceware.org/gdb

Table 4.2: Details of downstream task datasets.

Task	Dataset	Train	Valid	Test
Execution Estimation	CodeNet	121,319	13,116	13,116
Clone Detection	CXG-POJ104	32,000	8,000	12,000
Vulnerability Detection	REVEAL	15,867	2,268	4,535
	D2A	4,644	597	619
	CXG-Devign	21,854	2,732	2,732

We randomly sample up to 200 execution traces for each challenge, and this ends up with 121,319 training traces.

4.5.2.2 Downstream tasks

In this section, we introduce the datasets we use for each downstream task and explain the corresponding evaluation metrics. The statistics of these datasets are in Table 4.2.

Static Execution Estimation. We build the dataset for this task using CodeNet. We build the training samples from the 1,805 challenges that have been selected by the pre-training, and build evaluation samples from the held-out 95 challenges to avoid model memorization and data leakage.

Metrics. For the execution coverage prediction, we consider evaluation metrics in two granularities: full execution path and branch coverage. Concretely, for a sample with m branches, we denote the full set of their labels as $LB = \{lb_1, lb_2, ..., lb_m\}$, and the model prediction set as $\hat{LB} = \{\hat{lb_1}, \hat{lb_2}, ..., \hat{lb_m}\}$. If $LB == \hat{LB}$, we regard the prediction as matching the full execution path. For the branch coverage, we compute the occurrence of $lb_i == \hat{lb_i}$, where $1 \leq i \leq m$, and report the accuracy, precision, recall, and F1. Similarly, for the n quantized variable values within the program, $QV = \{qv_1, qv_2, ..., qv_m\}$, our model makes predictions as $\hat{QV} = \{\hat{qv_1}, \hat{qv_2}, ..., \hat{qv_m}\}$. If $QV == \hat{QV}$, we say the model accurately predicts the full execution. For the individual value match, we compute the occurrence of $qv_i == \hat{qv_i}$ and report the accuracy.

Semantic Clone Retrieval. We use CodeXGLUE-POJ104 [31, 30] as the dataset for this task. CodeXGLUE-POJ104 contains 104 programming challenges, and each has 500 C/C++ solutions submitted by different programmers. CodeXGLUE [30] reconstructs it as a public benchmark by splitting the dataset into Train (64 challenges), Dev (16 challenges), and Test (24 challenges) sets, with no overlapped challenge between any two sets.

Metrics. MAP@R (Mean Average Precision @ R)[3] is the main metric of this task, where we follow the design of the CodeXGLUE benchmark. Average precision at R is a common metric to evaluate the quality of information retrieval; it measures the average precision scores of a set of the top-R clone candidates presented in response to a query program. The "R" for CodeXGLUE is 499 as it has 500 solutions for each challenge.

Vulnerability Detection. We utilized three publicly available datasets: REVEAL (RV) [8], D2A [55], and CodeXGLUE-Devign (CXG) [30, 56]. The REVEAL dataset was curated by Chakraborty et al. [8] to simulate a real-world scenario where bugs are relatively rare, resulting in a ratio of approximately 1:10 between buggy and benign samples. The D2A dataset is a balanced dataset focusing on bug-fixing commits. It labels the previous version of modified functions as buggy and the fixed version as benign. Finally, the CodeXGLUE-Devign dataset, introduced by Zhou et al. [56], is also a balanced dataset that has been reconstructed as a public benchmark by CodeXGLUE, ensuring that all models can be evaluated using the same train/valid/test splits.

Metrics. REVEAL is an imbalanced dataset, so we use F1 as the evaluation metric. D2A and Devign are balanced datasets, so we follow the original benchmark to report the classification accuracy.

4.5.3 Model Configuration

TRACED's backbone is a standard $\text{RoBERTa}_{\text{BASE}}$ architecture [29] with 12 layers of Transformer-encoder, and each layer has 12 attention heads and the hidden dimension is 768.

[3] https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)#Mean_average_precision

TRACED is initialized with the pre-trained weights from UnixCoder [14][4], and we use its BPE tokenizer to split the rare tokens into BPE sub-tokens. The maximum sequence length is 1024 BPE tokens, and the longer sequence will be truncated. When the code sample is paired with executable inputs, the maximum length for the executable input is 64, and the source code is 960. Our experiments are conducted on 2 × 24GB NVIDIA GeForce RTX-3090 GPUs. We further pre-train the model for 10 epochs to learn the program execution with two learning rates, 5e-5 and 2e-5, and report the best-performing models for downstream tasks. For all the fine-tuning tasks, we use the learning rate of 8e-6. Learning rates typically decrease for later phases [13, 15, 11], so TRACED follows the same design. We use Adam optimizer [24] with the linear learning rate decay. Our model is implemented mainly with Pytorch [35] and Huggingface [49].

4.6 Evaluation

In this section, we ask the following four RQs:

- **RQ1:** How effective is TRACED in statically estimating the program execution?
- **RQ2:** How does our proposed training strategy contribute to learning the program execution?
- **RQ3:** Is our proposed quantized values for programs effective in guiding the model to learn program executions?
- **RQ4:** How does TRACED perform *w.r.t.* statically pre-trained baselines for code understanding tasks?

4.6.1 RQ1. Effectiveness of TRACED in Static Estimation of Execution

In this section, we demonstrate the effectiveness of TRACED in statically estimating program execution. The evaluation is more challenging and realistic than TRACED's

[4]Specifically, we load unixcoder-base-nine, as its pre-training considers C language code samples: https://huggingface.co/microsoft/unixcoder-base-nine. Note that this checkpoint is pre-trained only with the MLM objective, while the original paper [14] reports other better-performing variants that are not released publicly.

pre-training as it requires the model to predict not only for individual variables but also branches and the full execution path.

Baseline. In this RQ, we mainly compare the execution-aware TRACED with UnixCoder [14]. Now we explain the reasons for this choice. First, TRACED is initialized with the pre-trained UnixCoder weights, so comparing TRACED with the UnixCoder performance is a direct assessment of the impact of our proposed pre-training. Second, UnixCoder reports the state-of-the-art performance in many tasks, including clone detection, code search and summarization, and code generation and completion, significantly outperforming other pre-trained code models, such as CodeBERT [13] and GraphCodeBERT [15]. Third, it consumes up to 1,024 tokens, while most pre-trained code models [13, 6, 15, 1, 48] take at maximum 512 tokens. By consuming longer sequences, UnixCoder is able to handle longer programs and make complete predictions without truncating code in many cases. As TRACED is also designed to consume 1,024 tokens, it is not fair to compare it in this task with baselines with a maximum length of 512, as the baselines will necessarily consider fewer branches for prediction.

Result. The comparison is shown in Table 4.3, Row-1 *vs.* Row-2. TRACED significantly outperforms UnixCoder in the static estimation of execution coverage and dynamic values of variables, especially when the evaluation granularity is coarse, *i.e.,* full execution path (Full Path column in Table 4.3) and the runtime values of the full execution (Full Exec column in Table 4.3). TRACED correctly predicts the complete execution paths for 71.6% held-out samples and

Table 4.3: Performance on static execution estimation.

Task	**Coverage**					**Runtime Value**	
	Full Path	**Branch**				**Full Exec**	**Var**
Metric	**Acc**	**Acc**	**Prec**	**Rec**	**F1**	**Acc**	**Acc**
UnixCoder	63.7	79.7	81.7	85.4	83.5	39.3	87.8
TRACED	**71.6**	**83.1**	84.6	**88.1**	**86.3**	**49.2**	**89.2**
-w/o MLM	70.4	82.6	**85.3**	86.0	85.6	49.0	**89.2**
-w/o PSP	69.0	81.4	83.0	86.9	84.9	44.0	87.4
-w/o VCP	66.1	80.3	82.4	85.6	84.0	46.7	89.0
-MLM-only	65.6	81.0	83.1	86.0	84.6	43.0	87.5

```
//Input: 19 100
#include <stdio.h>
int main(){
    int A, N, T, B;
    scanf("%d %d", &N, &A);
    T = N * N;
    B = T - A;
    if (A > 0) {printf("%d", B);} // Branch-1
    else {printf("%d", T);} // Branch-2
    return 0;
}
```

UnixCoder Predictions (Wrong)

Branch-1: Not executed

Branch-2: Not executed

TRACED Predictions (Correct)

Branch-1: Executed

Branch-2: Not Executed

Figure 4.5: A qualitative example of execution coverage prediction. The source code is the same as Figure 4.1, but the input triggers a different execution path. TRACED correctly flips the prediction while UnixCoder remains the same prediction.

accurately predicts all variable values for 49.2% executions, revealing the execution-aware pre-training improves over UnixCoder's performance by 12.4% and 25.2%, respectively.

Case Study with Qualitative Examples. We present two qualitative examples in Figures 4.5 and 4.6 to concretely compare TRACED with UnixCoder in execution coverage and runtime value predictions, respectively. Both samples have simple execution logic from the human perspective, but the statically pre-trained UnixCoder still fails to correctly estimate them. Figure 4.5 illustrates that UnixCoder is not sensitive to distinct inputs that trigger different execution coverage, while TRACED is able to determine the numerical relations among varied values. Figure 4.6 illustrates TRACED's capacity in exposing abnormal program behaviors.

Result-1: With a similar number of learnable parameters, TRACED outperforms the state-of-the-art pre-trained code model in the static estimation of program execution task. Our proposed pre-training successfully encodes the execution awareness into TRACED's code representations.

```
//Input: 4 4320 4320 4320
#include <stdio.h>
int main (void) {
    int n, a, max = 0, sum = 0, i;
    for (i = 0; i < n; i++){ // Quantized value of n?
        scanf("\%d", &a);
        if (a > max) max = a;
        sum += a;
    }
    printf("\%d\\n" , sum - max / 2);
    return 0;
}
```

UnixCoder Prediction (Wrong)

n: Zero

TRACED Prediction (Correct)

n: Negative Large

Figure 4.6: A qualitative example of runtime value prediction. The sample contains a vulnerability of type CWE-457 "Use of Uninitialized Variable". The uninitialized `n`, which is randomly assigned as -32767, is used in the `for-loop`. TRACED successfully exposes this abnormal behavior statically by identifying n as a "Negative Large" value while UnixCoder fails. Predictions of other variables are hidden for better illustration.

4.6.2 RQ2. Effectiveness of TRACED's Pre-training Objectives

One of the main contributions of this chapter is proposing multi-task pre-training to effectively learn the execution-aware code representations. In this RQ, we study the effectiveness and contribution of each of TRACED's objectives, and consequently illustrate the importance of the multiple tasks.

To conduct these experiments, we remove one pre-training objective at a time and pre-train the variant with exactly the same setup as the main model. Then we fine-tune the variant on the static execution estimation task and compare the performance with the main model. We also consider a variant that is pre-trained on our dataset but only with MLM objectives. The results are shown in Row 3-6 of Table 4.3. Removing any objective hurts TRACED's performance, suggesting that comprehensively learning both static and dynamic code properties is more effective than learning one perspective alone.

Result-2: TRACED's multi-task pre-training helps the model comprehensively learn both static and dynamic aspects of source code. Removing any one of TRACED's three pre-training objectives noticeably hurts the model's performance in statically estimating program executions.

4.6.3 RQ3. Effectiveness of TRACED's Quantized Variable Values

Another contribution of this chapter is that the simplified and compact representation of program executions helps code models to capture dynamic code properties. In this RQ, we empirically reveal that the design of quantized variable values especially contributes to the effective learning of the code models, as it reduces the data sparsity of variable values but still defines sufficiently detailed value categories to distinguish dissimilar values.

To isolate the evaluation of TRACED's quantized values, we pre-train several variants by only recreating quantized value labels, *i.e.,* q_{var} in Equation (4.2), using different value abstraction strategies. For example, when we pre-train a variant studying the impact of concrete values, we replace TRACED's defined q_{var} with the concrete traced values. As different strategies abstract values at different granularities, it is not feasible to compare them for the value prediction task, since the coarse-grained strategy will benefit. Therefore, we only fine-tune the studied variants for the execution coverage prediction.

Baseline. First, we consider comparing with concrete values, as it is the most intuitive strategy to represent variable values. Then, we consider two data abstractions from LExecutor [44]: coarse and fine-grained. They share similar high-level intuition with us, mapping concrete values to pre-defined bins to reduce data complexity and consequently help the model's learning. Note that LExecutor's data abstraction serves a different goal than TRACED, and focuses on Python while TRACED focuses on C, so we could not directly reuse their pre-defined bins. As their definition of data abstraction is clear and straightforward, we re-implement their data abstraction for the C language and integrate it into our framework for comparison. We discuss and compare LExecutor with TRACED in more detail in the Related Work section (Section 4.7).

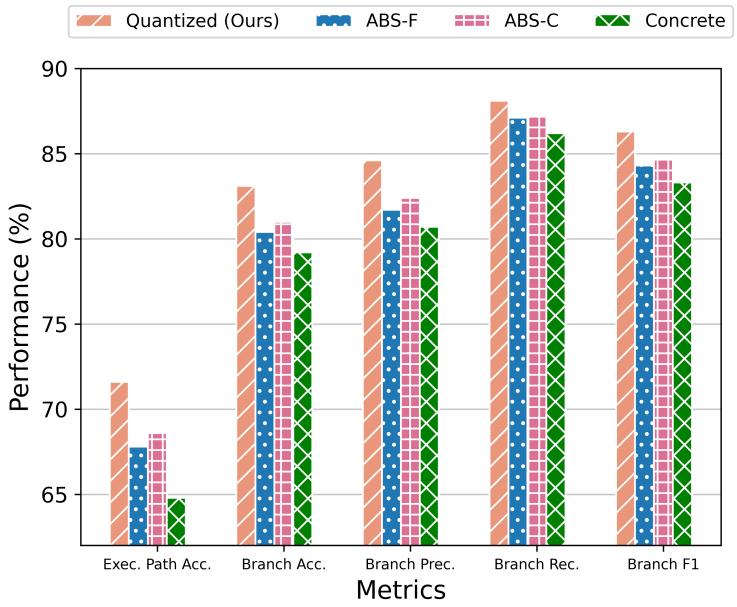


Figure 4.7: Comparing TRACED's design of quantized variable values with other value abstraction strategies.

Results. The comparison of value abstractions are shown in Figure 4.7. Unsurprisingly, concrete values report poor performance compared to other data abstractions, empirically revealing the difficulties for code models to fit sparse and complex data distributions. Interestingly, we notice both of LExecutor's abstractions perform slightly worse than TRACED. We speculate that LExecutor is not as sensitive as TRACED to numeric relations in the conditional statements, as they do not distinguish among small, regular, and large values. Note that execution coverage is not the main focus of LExecutor, so more fine-grained categories are not required to serve its goal, while they are empirically proven to be necessary for TRACED's scope.

Result-3: TRACED's quantized variable values directly contribute to the effectiveness of its execution-aware pre-training. It reduces the data sparsity of concrete values but defines sufficiently detailed value categories to distinguish dissimilar values for reasoning about execution paths.

Table 4.4: Comparison of Clone Retrieval and bug detection.

Task	**Clone Retrieval**	**Vulnerability Detection**		
Metric	**MAP@R**	**F1**	**Accuracy**	
Dataset	**POJ-104**	**RV**	**D2A**	**CXG**
CodeBERT	85.2	45.5	61.0	63.2
GraphCodeBERT	86.7	46.6	58.3	62.9
PLBART-base	75.9	46.9	61.7	63.3
CodeT5-base*	65.9	46.5	**62.1**	64.4
UnixCoder	89.5	47.4	61.2	65.3
TRACED	**91.2**	**50.4**	**62.1**	**65.9**

*CodeT5-base has 223M parameters, roughly twice as large as other baselines and TRACED. We report its performance as CodeT5-small has only 60M parameters and performs poorly, and CodeT5 does not provide a ∼110M model.

4.6.4 RQ4. TRACED's Performance in Code Understanding Tasks

In this RQ, we study TRACED's performance on two code understanding tasks: semantic clone retrieval and function-level vulnerability detection. Note that samples for these tasks are not paired with executable inputs, so the model needs to reason about the general code semantics to make predictions.

Baselines. We consider five pre-trained code models with similar parameter sizes to TRACED. CodeBERT [13] pre-trains a RoBERTa model with MLM and replaced token detection (RTD) tasks. GraphCodeBERT [15] is initialized with CodeBERT and continues pre-training with augmented data flow graphs to learn the static data dependencies. PLBART [1] and CodeT5 [48] both apply the sequence-to-sequence neural architecture, where PLBART adapts the BART [26] model to learn code translation and summarization, and CodeT5 adapts [41] to predict the missing code tokens and locate the identifiers. We also, again, consider UnixCoder as a baseline.

Results. We show the results in Table 4.4. Even though the samples in these benchmarks do not have executable inputs, TRACED still outperforms the statically pre-trained models by a clear margin. We speculate the reason is that TRACED could estimate the general execution behaviors without specific inputs, and the program semantics regarding these two code

understanding tasks could be better captured with such a general sense. Specifically, clone retrieval requires the model to identify the behavioral similarities of code as semantic clones mostly differ in code text and syntax. Also, vulnerable code with potential anomalies could be directly identified by TRACED in some cases like Figure 4.6.

Result-4: TRACED outperforms statically pre-trained models in clone retrieval and vulnerability detection tasks, suggesting TRACED's general estimation of execution helps it capture the code semantics more effectively.

4.7 Related Work

Pre-trained Models for Source Code The research community has shown a growing interest in developing pre-trained Transformer models for source code. These models can be broadly categorized into three primary architectures: Encoder-only [13, 15, 47, 5, 21, 6, 11], Decoder-only [9, 50, 2], and Encoder-decoder [33, 1, 14, 27, 7]. Encoder-only models predominantly employ MLM objective and sequence understanding tasks (*e.g.,* predicting next statement [21] and contrasting semantics [11]). This architecture excels at understanding the static code features. Decoder-only models, on the other hand, are typically trained by predicting code tokens in a left-to-right manner. This architecture focuses on generating code text based on learned patterns. The Encoder-decoder models combine the strengths of both Encoder-only and Decoder-only models and are pre-trained using various tasks, including denoising autoencoding for reconstructing wrongly permuted tokens [1], predicting missing identifiers in the code [48], and recovering method names from the source code [33].

These models primarily focus on learning the static aspects of source code but often miss out on capturing the dynamic properties of code execution. This limitation restricts these models from accurately inferring runtime behaviors, debugging issues, and understanding complex program states.

Modeling Program Execution Pei et al. [39, 37, 38] proposed a series of pioneering works to learn the executions of *binary* programs with Transformer-based models. They used concrete values from registers, which are feasible in their scope because binary programs have a smaller space of possible values and effects compared to source code. On the other hand, our work focuses on encoding execution at the source code level by imitating the developers' code practice. Variables in source code have more complicated data and value types than machine registers. We introduce quantized values in order to decrease the data complexity and sparsity.

Several works [53, 43, 4, 34, 3, 51] have attempted learning to execute programs as a direct goal. Souza and Pradel [44] also proposed LExecutor to predict missing values during execution. While it shares similar intuition of mapping concrete values to discrete categories, LExecutor is distinct from TRACED in several perspectives. First, LExecutor focuses only on predicting the values, while TRACED proposes a general pre-training strategy to encode the comprehensive execution awareness, not only values but also execution coverage, into the code representation. Besides, to yield code representations at a better quality, TRACED jointly learns both code text and dynamic executions rather than sticking to a single perspective. Due to the distinct aims and designs, we empirically illustrate in RQ3 (Section 4.6.3) that LExecutor's value abstractions are not perfectly aligned with our scope.

Nie et al. [32] annotated programs with information about the program's possible executions without executing the code but provided only statically available information. Conversely, several works [16, 45, 46, 36] require dynamic traces as input. We show that TRACED's pre-training is able to encode the execution awareness into code representation and estimate the dynamic semantics with static information alone. We expect such an implicit execution encoding could also help with broader software engineering tasks, such as bug localization and program repair [51, 52].

4.8 Threats to Validity

Internal Validity First, the current design of quantized value is not covering all variables within the program due to the complexity of their data structures, value ranges, and/or memory

allocations. Second, currently, we only trace the program by feeding it valid and executable inputs which will not terminate the program or throw errors. This might make the model less capable of capturing program termination and error-throwing behaviors.

External Validity At present, TRACED supports only the C programming language. This limitation is due to the reliance on the capabilities of the tracer used to log the execution history, which may not be readily available or equally effective for other programming languages. In order to extend TRACED's applicability, it is necessary to ensure that the tracer employed can accurately and consistently capture the required information across different languages. Adapting TRACED to multiple languages would require the development or adaptation of tracers that can effectively handle the intricacies of each language and produce comparable results, enabling a consistent analysis of code behavior across a broader range of programming languages.

4.9 Conclusion

We propose TRACED, an execution-aware pre-trained model that jointly learns the static and dynamic code properties, to address the limitation of existing, statically pre-trained code models. The evaluation empirically reveals that TRACED is more effective in estimating code execution statically than statically pre-trained models. TRACED also successfully transfers execution awareness to code understanding tasks.

4.10 Acknowledgments

We appreciate all the anonymous reviewers for their thoughtful feedback and suggestions to improve this work.

This work was supported in part by NSF grants CCF-2313054, CCF-2313055, CCF-1815494, CCF-210740, CCF-1845893, IIS-2221943, and DARPA/NIWC Pacific N66001-21-C-4018. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect those of the US Government, NSF, or DARPA.

4.11 Bibliography

[1] Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Online, June 2021), K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds., Association for Computational Linguistics, pp. 2655–2668.

[2] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C. J., Terry, M., Le, Q. V., and Sutton, C. Program synthesis with large language models. *CoRR abs/2108.07732* (2021).

[3] Bieber, D., Goel, R., Zheng, D., Larochelle, H., and Tarlow, D. Static prediction of runtime errors by learning to execute programs with external resource descriptions. In *The Eleventh International Conference on Learning Representations* (2023).

[4] Bieber, D., Sutton, C., Larochelle, H., and Tarlow, D. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks. In *Advances in Neural Information Processing Systems* (2020), vol. 33, Curran Associates, Inc., pp. 8626–8637.

[5] Bui, N. D. Q., Yu, Y., and Jiang, L. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *SIGIR '21* (Jul 2021), pp. 511–521. arXiv: 2009.02731.

[6] Buratti, L., Pujar, S., Bornea, M., McCarley, S., Zheng, Y., Rossiello, G., Morari, A., Laredo, J., Thost, V., Zhuang, Y., and Domeniconi, G. Exploring software naturalness through neural language models, 2020.

[7] Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P. T., and Ray, B. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2022), ESEC/FSE 2022, Association for Computing Machinery, p. 18–30.

[8] Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering 48*, 09 (Sept. 2022), 3280–3296.

[9] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. arXiv: 2107.03374.

[10] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. 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* (Minneapolis, Minnesota, Jun 2019), Association for Computational Linguistics, pp. 4171–4186.

[11] Ding, Y., Buratti, L., Pujar, S., Morari, A., Ray, B., and Chakraborty, S. Towards learning (dis)-similarity of source code from program contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Dublin, Ireland, May 2022), S. Muresan, P. Nakov, and A. Villavicencio, Eds., Association for Computational Linguistics, pp. 6300–6312.

[12] Ding, Y., Ray, B., Premkumar, D., and Hellendoorn, V. J. Patching as translation: the data and the metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering* (2020), ASE '20.

[13] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (Online, Nov 2020), Association for Computational Linguistics, pp. 1536–1547.

[14] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Dublin, Ireland, May 2022), S. Muresan, P. Nakov, and A. Villavicencio, Eds., Association for Computational Linguistics, pp. 7212–7225.

[15] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., LIU, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., and Zhou, M. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations* (2021).

[16] Henkel, J., Lahiri, S. K., Liblit, B., and Reps, T. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, Oct 2018), ESEC/FSE 2018, Association for Computing Machinery, pp. 163–174.

[17] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering* (2012), ICSE '12, IEEE Press, p. 837–847.

[18] Jiang, N., Liu, K., Lutellier, T., and Tan, L. Impact of code language models on automated program repair. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 1430–1442.

[19] Jiang, N., Lutellier, T., Lou, Y., Tan, L., Goldwasser, D., and Zhang, X. Knod: Domain knowledge distilled tree decoder for automated program repair. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 1251–1263.

[20] Jiang, N., Lutellier, T., and Tan, L. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering* (2021), ICSE '21, pp. 1161–1173.

[21] Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning* (2020), ICML'20, JMLR.org.

[22] Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C., and Janes, A. Big code != big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering* (2020), ICSE '20, pp. 1073–1085.

[23] Kim, S., Woo, S., Lee, H., and Oh, H. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 595–614. Publisher: IEEE.

[24] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2015).

[25] Kudo, T., and Richardson, J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Brussels, Belgium, Nov 2018), EMNLP '18, Association for Computational Linguistics, pp. 66–71.

[26] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (Online, Jul 2020), Association for Computational Linguistics, pp. 7871–7880.

[27] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science 378*, 6624 (Dec. 2022), 1092–1097.

[28] Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., and Hu, J. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (New York, NY, USA, Dec 2016), ACSAC '16, Association for Computing Machinery, pp. 201–213.

[29] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. RoBERTa: A robustly optimized BERT pretraining approach. *CoRR abs/1907.11692* (2019).

[30] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *CoRR abs/2102.04664* (2021).

[31] Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (2016), pp. 1287–1293.

[32] Nie, P., Banerjee, R., Li, J. J., Mooney, R. J., and Gligoric, M. Learning Deep Semantics for Test Completion. arXiv. arXiv:2302.10166.

[33] Niu, C., Li, C., Ng, V., Ge, J., Huang, L., and Luo, B. SPT-Code: Sequence-to-sequence pre-training for learning source code representations. *CoRR abs/2201.01549* (2022).

[34] Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. Show Your Work: Scratchpads for Intermediate Computation with Language Models, Nov 2021. arXiv:2112.00114.

[35] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. *PyTorch: an imperative style, high-performance deep learning library.* Curran Associates Inc., Red Hook, NY, USA, 2019.

[36] Patra, J., and Pradel, M. Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh Pennsylvania, May 2022), ACM, pp. 1469–1481.

[37] Pei, K., Guan, J., Broughton, M., Chen, Z., Yao, S., Williams-King, D., Ummadisetty, V., Yang, J., Ray, B., and Jana, S. StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, Aug 2021), ESEC/FSE 2021, Association for Computing Machinery, pp. 690–702.

[38] Pei, K., She, D., Wang, M., Geng, S., Xuan, Z., David, Y., Yang, J., Jana, S., and Ray, B. NeuDep: neural binary memory dependence analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, Nov 2022), ESEC/FSE 2022, Association for Computing Machinery, pp. 747–759.

[39] Pei, K., Xuan, Z., Yang, J., Jana, S., and Ray, B. Trex: Learning execution semantics from micro-traces for binary similarity. *CoRR abs/2012.08680* (2020).

[40] Puri, R., Kung, D. S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M. R., Decker, L., Thost, V., Buratti, L., Pujar, S., and Finkler, U. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *CoRR abs/2105.12655* (2021).

[41] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research 21*, 140 (2020), 1–67.

[42] Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., and Devanbu, P. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), vol. 14-22-May- of *ICSE '16*, ACM, pp. 428–439.

[43] Reed, S., and de Freitas, N. Neural Programmer-Interpreters, Feb 2016. arXiv: 1511.06279.

[44] Souza, B., and Pradel, M. LExecutor: Learning-Guided Execution, Feb 2023. arXiv: 2302.02343.

[45] Wang, K., Singh, R., and Su, Z. Dynamic neural program embeddings for program repair. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* (Feb 2018), pp. 1–12.

[46] Wang, K., and Su, Z. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, Jun 2020), PLDI 2020, Association for Computing Machinery, pp. 121–134.

[47] Wang, X., Wang, Y., Mi, F., Zhou, P., Wan, Y., Liu, X., Li, L., Wu, H., Liu, J., and Jiang, X. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation, Sep 2021. arXiv:2108.04556.

[48] Wang, Y., Wang, W., Joty, S., and Hoi, S. C. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021* (2021).

[49] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[50] Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169* (2022).

[51] Ye, H., Martinez, M., Luo, X., Zhang, T., and Monperrus, M. SelfAPR: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2023), ASE '22, Association for Computing Machinery.

[52] Ye, H., and Monperrus, M. ITER: Iterative neural repair for multi-location patches. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[53] Zaremba, W., and Sutskever, I. Learning to execute, 2015. arXiv: 1410.4615.

[54] Zeller, A. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[55] Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., and Su, Z. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA, 2021), ICSE-SEIP '21, Association for Computing Machinery.

[56] Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems 32* (2019).

CHAPTER 5. TO ERR IS MACHINE: VULNERABILITY DETECTION CHALLENGES LLM REASONING

Benjamin Steenhoek[1], Md Mahbubur Rahman[2], Monoshi Kumar Roy[3], Mirza Sanjida Alam[4], Hengbo Tong[5], Swarna Das[6], Earl Barr[7], and Wei Le[8]

[1-6,8] Department of Computer Science, Iowa State University, Ames, IA, 50011

[7] Department of Computer Science, University College London, London, United Kingdom

Modified from a manuscript under review for the *13th International Conference on Learning Representations (ICLR 2025)*

Abstract

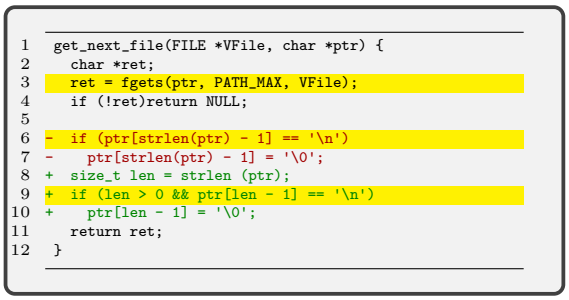
In this chapter, we present a challenging code reasoning task: vulnerability detection. Large Language Models (LLMs) have shown promising results in natural-language and math reasoning, but state-of-the-art (SOTA) models reported only 54.5% Balanced Accuracy in our vulnerability detection evaluation, even those models pre-trained on large amounts of source code. Our error analysis on LLM responses shows that the models struggle to reason about the code semantics relevant to identifying vulnerabilities, especially subtle semantic differences caused by small textual changes. We explored prominent models and training settings to understand their effects on vulnerability detection performance — including better prompts, larger models, more pre-training data, and fine-tuning — but none led to significant improvements. This raises the question of whether simply scaling training data and model size will allow us to "solve" complex code reasoning tasks like vulnerability detection, or if a fundamental shift in modeling and training techniques is required. We also explored adding domain knowledge to prompts; although it helped certain models understand some code semantics, vulnerability detection requires multi-step reasoning, and these models still failed in steps, such as reasoning about variable

relations. Our results suggest that new models, new training methods, or more execution-specific pretraining data may be needed to conquer vulnerability detection. We speculate that auto-regressive pre-training on source code may not effectively extract code semantics, especially on the current pretraining mixtures, in which execution data is scarce. Success on vulnerability detection as a code reasoning task can benefit many areas of software engineering such as debugging, test input generation, and program repair. Our code and data are available at https://doi.org/10.6084/m9.figshare.27368025.

5.1 Introduction

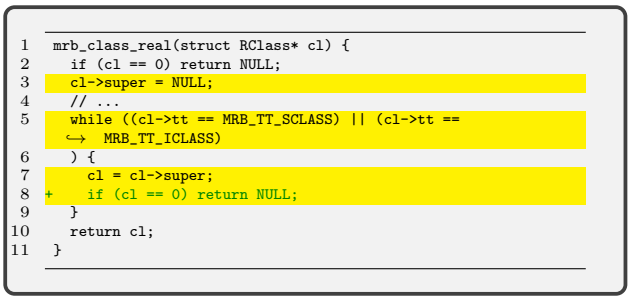
Thousands of new software vulnerabilities are discovered each year, costing users and companies millions of dollars [37]. This makes vulnerability detection critically important for software security. Since Devign in 2019 [64], many deep learning approaches have been proposed to predict the presence of vulnerabilities, but model performance has not breached 70% F1 score on realistic datasets [7]. In this chapter, we show that though existing LLMs achieve impressive results on math, natural language, code reasoning and code generation tasks [48, 9, 19, 6] they struggle to detect vulnerabilities (Section 5.2). We show that vulnerability detection is a complex code reasoning challenge, requiring both multi-step analysis and an accurate understanding of code semantics. This chapter makes the case that vulnerability detection presents a compelling new target task for the ML community; solving it could significantly impact related software engineering tasks, such as debugging, test input generation, and program repair, thereby enhancing developer productivity. Furthermore, improving LLM's ability to reason about code and identify vulnerabilities could potentially drive progress in broader reasoning tasks.

As shown in Figure 5.3, to detect a vulnerability, a developer first needs to accurately locate the statements relevant to a potential vulnerability. Second, a developer must understand the semantics of those relevant statements, which requires domain knowledge, such as recognizing bounds/NULL checks and understanding the effects of string, pointer, and arithmetic operations. Sometimes only a single operator distinguishes vulnerable and non-vulnerable versions of code,



```
1   get_next_file(FILE *VFile, char *ptr) {
2     char *ret;
3     ret = fgets(ptr, PATH_MAX, VFile);
4     if (!ret)return NULL;
5
6   -  if (ptr[strlen(ptr) - 1] == '\n')
7   -    ptr[strlen(ptr) - 1] = '\0';
8   +  size_t len = strlen (ptr);
9   +  if (len > 0 && ptr[len - 1] == '\n')
10  +    ptr[len - 1] = '\0';
11    return ret;
12  }
```

FIGURE 5.1. Buffer Overflow (BOF). To detect this vulnerability in the vulnerable version, the model/developer takes several reasoning steps: (step 1) identify the BOF-relevant statements, e.g., buffer allocation in line 3 and access in line 6; (step 2) understand that the allocated buffer may be empty depending on user input and that `strlen(ptr)` returns 0 in line 6 if the buffer is empty; (step 3) connect the facts, recognizing that if the buffer is empty, then line 6 will access index `-1`, causing a BOF. In the patched version, the model/developer should recognize in step (2) that line 9 checks the length of the buffer before accessing it, and therefore, step 3 concludes that this vulnerability is not exploitable.



```
1   mrb_class_real(struct RClass* cl) {
2     if (cl == 0) return NULL;
3     cl->super = NULL;
4     // ...
5     while ((cl->tt == MRB_TT_SCLASS) || (cl->tt ==
      ↪  MRB_TT_ICLASS)
6     ) {
7       cl = cl->super;
8   +   if (cl == 0) return NULL;
9     }
10    return cl;
11  }
```

FIGURE 5.2. Null-Pointer Dereference (NPD). To detect this vulnerability in the vulnerable version, the model/developer takes several reasoning steps: (step 1) identify the relevant statements, e.g. the assignments `cl->super = NULL` in line 3 and `cl = cl->super` in line 7, and dereference of `cl` in line 5; (step 2) understand that in line 3, `cl->super` is set to NULL; (step 3) connect the facts, recognizing that after assigning `cl` to NULL in line 7, it will be dereferenced when the loop condition is evaluated in line 5, causing a NPD. In the patched version, the model/developer should recognize in step (2) that line 8 checks if `cl` is NULL and returns safely, thus there is no vulnerability.

FIGURE 5.3. Examples of vulnerability detection as a complex code reasoning task. Diffed lines (+/-) show the lines changed to patch the vulnerability.

and effective vulnerability detection requires understanding these nuances of program semantics. Finally, the developer must logically connect the individual facts about the statements to infer whether a vulnerability exists. This last step requires reasoning about the ranges of values and the temporal relations of symbolic variables, and then comparing them to the application's security policy, which is often implicit.

These steps are challenging for LLMs, both individually and in combination. We studied 14 SOTA LLMs and 7 prompting methods on SVEN [21], a high-quality, real-world dataset

consisting of 386 vulnerable functions and their corresponding fixed versions, covering 772 programs. We found that *all models and prompts performed close to random guessing (50-55% Balanced Accuracy)* (Section 5.2). Even GPT-4, a SOTA model, couldn't distinguish vulnerable code from its fixed version for 67.4% cases.

After manually analyzing 300 of the LLM responses (Section 5.3), we found errors occurring at all three steps of the reasoning process. For instance, in step 1, localization, the models frequently (50% of inspected functions) failed to recognize bounds or NULL checks, resulting in false positives. Explicit marking of bounds checks is easily done by humans but seems to be difficult for LLMs to recognize. In step 2, LLMs misinterpreted string, pointer, and integer operations in 10%, 6%, and 8% of functions, respectively. Understanding bounds/NULL checks and the operations requires a precise understanding of code execution semantics, which LLMs generally struggle with [19]; our results confirm this finding and further indicate which structures were most challenging. We attribute the models' lack of understanding of code semantics, even after using various prompting methods, to two key factors: (1) the models may have limited exposure to execution data during pre-training, which restricts their ability to learn semantics directly – although LLMs might acquire some semantic understanding indirectly from simple executions aligned with code text, or developer's discussions about semantics; and (2) the current autoregressive pre-training methods face inherent difficulty of learning execution semantics from code text alone. This is likely why we observe that scaling up model size or dataset volume, and performing fine-tuning, did not significantly improve performance (Sections 5.3.1 and 5.3.2); since the necessary data are not in the dataset, it is unlikely that LLMs can learn this complex reasoning via scaling alone. Annotating code semantics in prompts reduced some of these errors in certain models (Section 5.3.3), but determining vulnerabilities require a multi-step analysis. There are errors in other reasoning steps can further prevent the detection. We show that in step 3, LLMs frequently failed at multi-step logical reasoning, leading to inconsistent or non-sequential inferences in 9% of responses.

To the best of our knowledge, this chapter is the first to utilize vulnerability detection to systematically explore the capabilities of existing LLMs to reason about complex code properties. Ullah et al. [51] compared GPT-4's responses with human-written vulnerability summaries using metrics like *BLEU*, *ROUGE*, and *GPT-4 evaluations*, but did not delve into the specific failure modes which occurred in responses. Yu et al. [58] and Nong et al. [38] examined GPT-4 and GPT-3.5's responses about vulnerabilities but did not perform systematic studies on a set of models, and on the impact of model sizes, training data, training methods, and adding domain knowledge. We classified the errors based on the challenges of reasoning steps, resulting in categories which are more fine-grained and actionable; we explored mitigating a specific type of error using a prototype with CoT-Annotations, as discussed in Section 5.3.3.

In summary, we make the following contributions:

(1) We clarify vulnerability detection as a complex reasoning challenge;

(2) We demonstrate that current SOTA LLMs severely underperform in vulnerability detection, achieving only 50-55% balanced accuracy at best;

(3) Through manual analysis of hundreds of LLM responses, we reveal that LLMs struggle with all stages of reasoning, particularly in understanding semantics of statements involving bounds/NULL checks, string operations, and pointer handling, which contributes significantly to their poor performance;

(4) We show that these reasoning failures and low performance cannot be easily mitigated by increasing model size, improving training data, and applying fine-tuning, even when the model is provided with domain-specific knowledge.

The fact that vulnerability detection exposes the limitations in current models' abilities to reason about vulnerabilities, coupled with the availability of well-defined vulnerability data, makes vulnerability detection an ideal benchmark for evaluating and challenging LLM reasoning capabilities.

5.2 Can LLMs Effectively Detect Vulnerabilities?

Prompts: We used the baseline prompting methods including *Basic (zero-shot)* [16] and *In-context (n-shot)* [30, 63] prompts. We used a system prompt to set the context and instructions including the vulnerability definition [35] and the program source code (see Section 5.A for details).

We designed three additional prompting methods that leverage the metadata available in vulnerability datasets to encourage reasoning and provide the domain knowledge to the model, namely: (1) *In-context examples from contrastive pairs (Contrastive)*, which uses pairs before and after a bug-fix as in-context examples, with the goal of instructing the model the fine-grained differences which caused the bug, (2) *Chain-of-Thought from CVE descriptions (CoT-CVE)*, which uses CVE bug reports [10] from the Big-Vul dataset [13], prompting the model to respond with the explanations of vulnerability, and (3) *Chain-of-Thought from static analysis (CoT-StaticAnalysis)*, which adapts vulnerability proofs output by a static analyzer [3] as reasoning steps for the example response, conditioning the model to reason step-by-step. We obtained the proofs from the D2A dataset [62] (see Section 5.A for details).

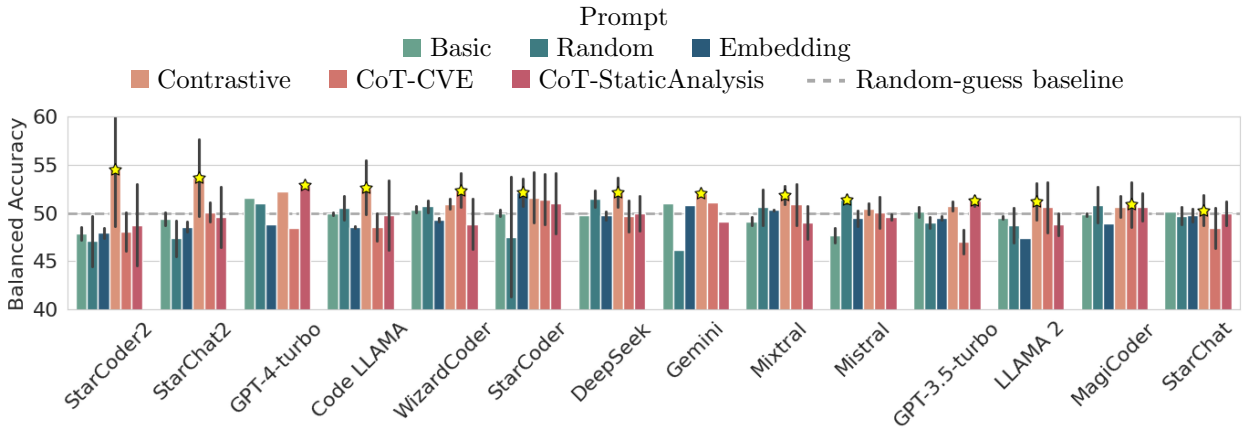


FIGURE 5.4. Vulnerability detection performance. Bar height shows the average performance of three random seeds and error bars show standard deviations; stars (☆) mark the best-performing prompt for each model.

	SVEN	HumanEval	CruxEval	GSM8k	CSQA
Model (params)	**Vuln. detection**	**Code gen.**	**Code execution**	**Math**	**NL reasoning**
StarChat	50.9	-	-	-	-
LLAMA 2 (34b)	51.2	22.6	-	42.2	-
StarCoder (15.5b)	52.2	45.8	34.2	-	-
Mistral (7b)	51.4	30.5	34.3	36.5	62.5
Mixtral (8x7b)	51.9	40.2	40.5	58.4	86.7
Code LLAMA (34b)	52.6	48.8	42.4	-	-
WizardCoder (33b)	52.4	59.8	43.4	-	-
MagiCoder (7b)	50.9	70.7	44.4	-	-
StarCoder2 (16b)	54.5	46.3	47.1	-	-
GPT-3.5-turbo	51.8	64.9	49.4	57.1	85.5
DeepSeek (33b)	52.1	69.2	49.9	-	-
GPT-4-turbo	52.9	87.1	68.7	87.1	95.3
Gemini 1.0 Pro	52.1	67.7	-	86.5	84.7
StarChat2	53.6	71.3	-	-	-

TABLE 5.1. Performance on vulnerability detection vs. NL/math reasoning, code generation, and code execution. Sources for code, math, and NL reasoning performance are cited in Section 5.C.

Models: We evaluated 14 LLMs which are the SOTA in code generation, based on several surveys [61, 29] and the HumanEval leaderboard [42] (as of March 2024). The models include LLAMA 2 [49], Code LLAMA [44], StarCoder [28], StarChat [50], StarCoder2 [31], StarChat2 [22], Mistral [24], Mixtral [25], MagiCoder [54], Wizardcoder [32], DeepSeek-Coder[20], GPT-3.5 [39], GPT-4 [40], and Gemini 1.0 Pro [18]. See Section 5.B for details.

Benchmark and Metrics: We used the SVEN dataset [21], which contains 772 vulnerable and fixed functions from real-world C/C++ projects (average length = 168 lines). Existing vulnerability datasets contain a lot of noise; SVEN contains vulnerabilities selected from multiple benchmarks, and are reported to be 93% accurate [11]. Considering that the commonly used F1 score can bias towards models which predict vulnerable more often [63], we used *Balanced Accuracy* [1] (defined as $(\frac{correct_{vul}}{examples_{vul}} + \frac{correct_{nvul}}{examples_{nvul}})/2$) to evaluate the models.

Results: Figure 5.4 shows the performance of the baseline methods and our proposed prompts. While our new prompts slightly improved the best-case performance for 11 out of 14 models, with Contrastive prompts enhancing 8 out of 14, none of the models or prompts exceeded the random-guessing baseline (Balanced Accuracy = 50) by more than 5% Balanced Accuracy. In

doubt of whether the complexity of the real-world code is the main challenging factor, we studied simple code examples (25 lines per function on average) from the CWE and SARD databases [35, 36] and found that the models still did not predict simple functions correctly, reporting 42-67% Balanced Accuracy across all the models (see Section 5.D). Table 5.1 compares the models' vulnerability detection performance with their performance in other domains. While new models have made steady advances in code generation [6], code execution [19], NL reasoning [48], and math reasoning [9], their vulnerability detection performance has not increased in step, remaining close to the random-guess baseline. This result implied that the until-now successful strategies of scaling model size and training data have not yet proven to be sufficient to solve vulnerability detection; to further confirm this, we investigated further in Sections 5.3.1 and 5.3.2.

TABLE 5.2. Models' abilities to distinguish pairs of vulnerable and non-vulnerable examples. Cell values display the number and percentage of pairs in each category.

		Distinguished	
Model	**Can't Distinguish**	**Both Correct**	**Both Wrong**
StarChat	86.1%	7.9%	6.1%
DeepSeek	82.5%	6.3%	11.2%
StarCoder	82.1%	12.5%	5.4%
GPT-3.5-turbo	80.9%	11.3%	7.8%
LLAMA 2	76.5%	15.6%	8.0%
MagiCoder	75.2%	11.9%	12.9%
Mixtral	67.8%	18.3%	13.9%
GPT-4-turbo	67.4%	18.9%	13.7%
Gemini	64.4%	19.1%	16.5%
Mistral	61.8%	20.6%	17.6%
StarChat2	61.4%	21.0%	17.6%
StarCoder2	57.5%	19.0%	23.5%
Code LLAMA	57.3%	22.3%	20.4%
WizardCoder	55.0%	23.8%	21.1%
Average	69.7%	16.3%	14.0%

Table 5.2 presents our results on the models' capabilities of distinguishing pairs of vulnerable code and its fixed version. In the table, under Column *Can't Distinguish*, we show that, on average across all the models, 69.7% of pairs could not be distinguished, indicating that the models do not understand the nuanced semantics of the vulnerability. Some models/prompts were

better than average, but at best, 55.0% of pairs could not be distinguished. The *Both Correct* and *Both Wrong* columns indicate that the models predicted both versions correctly in some instances (16.3% of pairs), but there were also cases (14.0% of pairs) where the models predicted both versions incorrectly.

5.3 Why do LLMs Fail to Reason About Vulnerabilities?

We manually inspected 300 vulnerable predictions (covering 100 programs) from 14 models, regarding the vulnerability reasoning steps, including locating and understanding the semantics of statements related to vulnerability decisions, as well as the logical reasoning that can integrate variable values and relations, and compare them with the security policy. To reduce subjectivity, our manual inspection used independent ratings from three authors, following Islam et al. [23], and adopted best practices for inductive coding [45]. See Section 5.E.3 for the details of our protocol.

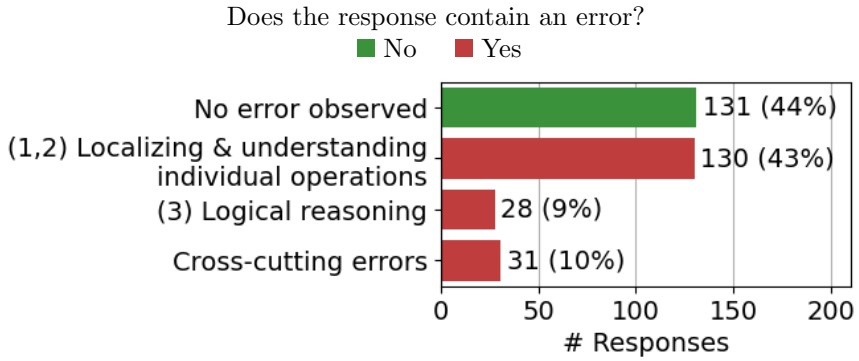


FIGURE 5.5. Error categories observed in responses from all LLMs. Bar width shows the number of responses that contained the category of error. One response can contain more than one type of error.

Figure 5.5 summarizes the errors. The results show that LLMs had some successes in reasoning, with 44% of responses containing no observed errors; however, still more than half of the responses contained an error in at least one step. LLMs made errors on localizing and understanding individual statements for 43% examples; this causes them to make faulty inferences about the effects of the code and flag potential vulnerabilities in safe code. LLMs also made

TABLE 5.3. Error analysis from 300 responses covering 100 programs. We analyzed the errors manually using the rubric and inter-rater agreement procedure detailed in Section 5.E.

Reasoning step	**Error**	**Count**
(1,2) Localizing and understanding statements related to vulnerability	Misunderstood Bounds/NULL check	80/159 (50%)
	Misunderstood string operation	3/29 (10%)
	Misunderstood arithmetic operation	8/96 (8%)
	Misunderstood pointer operation	9/147 (6%)
	Misunderstood alloc/free operation	4/81 (5%)
	Misunderstood index operation	1/60 (2%)
	Misunderstood execution order	11
	Improper assumption	8
	Misunderstood syntax	6
	Total	125
(3) Logical reasoning	Faulty implication ($\Rightarrow$)	14
	Inconsistent ($\perp$)	14
	Total	28
Cross-cutting errors	Hallucination	15
	Memorization	11
	Repetition	5
	Total	31

cross-cutting errors such as hallucination and repetition 10% of the time and made incorrect logical inferences 9% of the time.

The LLMs frequently made errors related to several specific code structures, shown in Table 5.3. For example, out of 159 responses explaining bounds/NULL checks, 80 (50%) were incorrect. The semantics of bounds/NULL checks are critically important for determining whether several pertinent vulnerabilities exist, including buffer overflow, null-pointer dereference, and use after free. Such checks often follow predictable code patterns and thus are relatively simple for developers and static analysis tools to identify — we used static analysis to recognize them in Section 5.3.3 — however, the LLMs often failed to recognize these patterns. In addition, the models face challenges of understanding the semantics of operations; for example, the models incorrectly interpreted 10% string operations and 8% of arithmetic operations, which are necessary for reasoning about buffer overflow and integer overflow vulnerabilities.

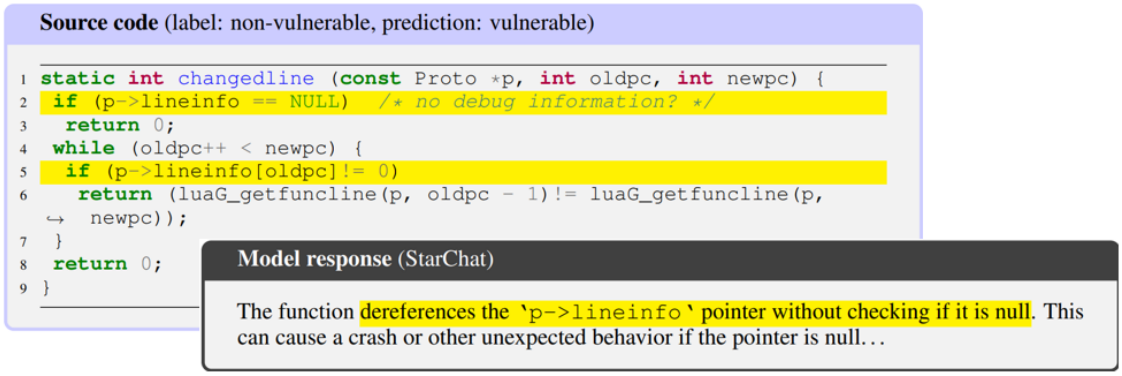


FIGURE 5.6. Missed Bounds/NULL check.

In Figure 5.6, StarChat reported that there is an unchecked null-pointer dereference at line 5 (`p->lineinfo[oldpc]`). However, it overlooked the safety check at line 2, where null values for `p->lineinfo` are handled safely. Figure 5.7 provides an example of a *Misunderstood arithmetic operation* error. GPT-4 correctly identified the bounds-check at line 6, which had been added by developers to prevent overflows [34]. However, the LLM failed to reason about the calculation of the argument value to `AllocChunk` at line 9. Given the upper bound of `0x7fff` for `nSamples+1` and `nPatches+1`, even the maximum values would not cause an overflow in an unsigned integer (`0x7fff` $*$ `0x7fff` $*$ `8` $=$ `0xfff80008`), so the LLM's alert is a false positive.

As a follow-up to the error analysis, we conducted a form of *natural experiment* [55] to compare various LLMs and assess whether prominent training strategies improved vulnerability performance. This study design enabled us to evaluate each training strategy independently while controlling other variables. We compared models of different *sizes* (Section 5.3.1) and models trained with varying data and training methods, including *increased training data volume*, *code vs. NL training data*, *instruction fine-tuning*, and *adapter fine-tuning* (Section 5.3.2). We also investigated the use of external tools (Section 5.3.3) to add domain knowledge targeting the types of reasoning errors we found in Table 5.3.

Source code (label: non-vulnerable, prediction: vulnerable)

```
void AllocateDataSet(cmsIT8* it8) {
  TABLE* t = GetTable(it8);
  if (t -> Data) return;  // Already allocated
  t-> nSamples = atoi(cmsIT8GetProperty(it8, "NUMBER_OF_FIELDS"));
  t-> nPatches = atoi(cmsIT8GetProperty(it8, "NUMBER_OF_SETS"));
  if (t -> nSamples < 0 || t->nSamples > 0x7ffe || t->nPatches < 0
  ↪ || t->nPatches > 0x7ffe)
    SynError(it8, "AllocateDataSet: too much data");
  else {
    t->Data = (char**)AllocChunk(it8, ((cmsUInt32Number)t->nSamples
  ↪ + 1) * ((cmsUInt32Number)t->nPatches + 1) * sizeof(char*));
    if (t->Data == NULL) {
      SynError(it8, "AllocateDataSet: Unable to allocate data
  ↪ array");
    }
  }
}
```

Model response (GPT-4)

Both `t->nSamples` and `t->nPatches` are user-controlled and can be large. When these values are incremented by 1 and then multiplied by each other and further multiplied by `sizeof(char*)`, the result can overflow a 32-bit integer if the values are sufficiently large...
Before performing the allocation, the function checks if `nSamples` and `nSamples` are within a certain range (`0` to `0x7ffe`). However, this check does not prevent overflow. For example, values close to the upper limit can still cause the multiplication result to wrap around, leading to a smaller allocation than required.

FIGURE 5.7. Misunderstood arithmetic operation.

5.3.1 Does Model Size Matter?

We evaluated several models which released different sizes: LLAMA 2 (7b, 13b), Code LLAMA (7b, 13b, 34b), Mistral 7b vs. Mixtral 8x7b, and DeepSeek-Coder (1.3b, 6.7b, and 33b). Figure 5.8 shows that model performance did not significaly improve by scaling up the model size, and we found that there was no statistical correlation between model size and performance ($R^2 = 0.02$, $p = 0.72$). We manually analyzed the responses using the methodology in Section 5.3 and found that all models had error rates similar to those shown in Figure 5.5, although larger models were better at following in-context prompts. For example, Code LLAMA 7b, often analyzed the in-context examples instead of the queried example; this error happened less frequently with Code LLAMA 13b, and not at all with Code LLAMA 34b. This aligns with previous results [52] showing that in-context learning is an emergent property of larger models.

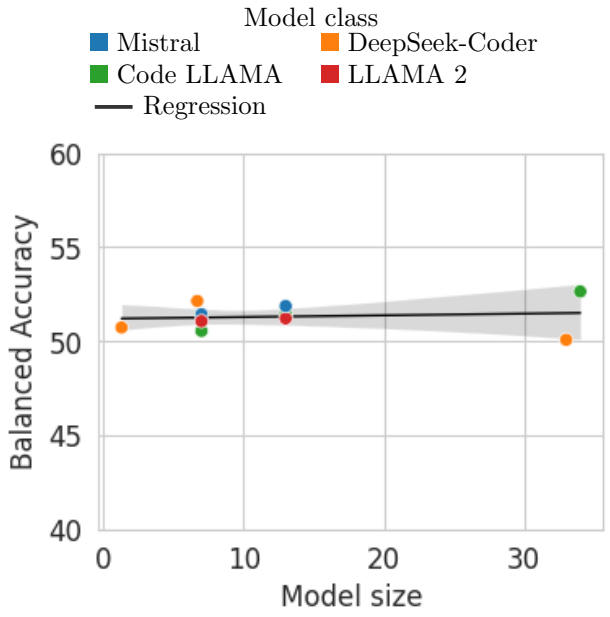


FIGURE 5.8. Larger models did not improve on vulnerability detection.

5.3.2 Do Model Training Data & Methods Matter?

Figure 5.9a (top), General-purpose vs. code training data: Models trained mainly on natural language may lack the knowledge of code seen in models which have been fine-tuned on code. This raises the question: do models specialized for code outperform general-purpose models? To explore this, we compared LLAMA 2, designed as a general-purpose chat assistant [49], against Code LLAMA, which was initialized from the base weights of LLAMA 2 and further fine-tuned on code [44]. Figure 5.9a (top) shows that code-specialized training did not substantially improve Code LLAMA's vulnerability detection capability.

Figure 5.9a (bottom), Training on more data: HuggingFace's Bigcode team released StarCoder and its updated version, StarCoder2, with the primary difference being that StarCoder2 trained on more than twice as much code [31]. This setup provides a relatively controlled environment to assess the impact of this additional training data. Figure 5.9a (bottom) indicates that scaling up the dataset resulted slightly improved StarCoder2's vulnerability detection capability, but yielded only a 4% improvement.

(A) Data volume & modality (B) Instruction fine-tuning (C) Adapter fine-tuning

FIGURE 5.9. Expanding the training dataset and incorporating fine-tuning had minimal impact on vulnerability detection capability.

Figure 5.9b, Instruction fine-tuning : Instruction fine-tuning can improve the truthfulness and relevance of responses [41], as well as performance and generalization [8]. This leads us to ask: do instructed models perform better than their base counterparts? We compared the base versions of DeepSeek Coder, StarCoder2, and Code LLAMA against their instruction fine-tuned counterparts, namely WizardCoder, StarChat2, and Code LLAMA-Instruct/MagiCoder respectively, and found no substantial difference in performance (Figure 5.9b), indicating that instruction fine-tuning did not improve vulnerability detection performance, even though our vulnerability detection prompts are tailored for instruction-tuned models.

Figure 5.9c, Adapter fine-tuning: We fine-tuned the StarCoder2 7b model using the vulnerability dataset PrimeVul [11] (the cleanest supervised dataset large enough for fine-tuning) and compared it with the base 7b model using the Basic prompt on SVEN. We evaluated the models on balanced and imbalanced label distributions, following Chakraborthy et al. [4]. Figure 5.9c shows that finetuning did not substantially improve the model's performance in either

case; the fine-tuned model performed worse on balanced data and only 1.2% better on imbalanced data.

5.3.3 Does Additional Domain Knowledge Help?

CoT-Annotations prompt for the source code in Figure 5.2

Code Explanation: The pointer `cl` is passed as parameters in this function, which could be null. In line `cl->super = NULL`, the pointer `cl->super` is assigned to NULL. In line `while ((cl->tt == MRB_TT_SCLASS)...`, the pointer `cl` is dereferenced. The function has a NULL check in line `if (cl == 0) return NULL;` for the pointer `cl`.
Instructions: For the Query Example only, think step-by-step using these steps, then give an answer based on the Code Explanation. Assume pointers which are passed in will not cause a null-pointer dereference.
1. identify which locations pointers are likely to be null.
2. identify which of these are dereferenced.
3. identify which of these dereferences of potentially-null pointers are checked and handled. Filter these out.
4. if any unchecked dereferences remain, then the function is vulnerable. Otherwise, the function is not vulnerable.

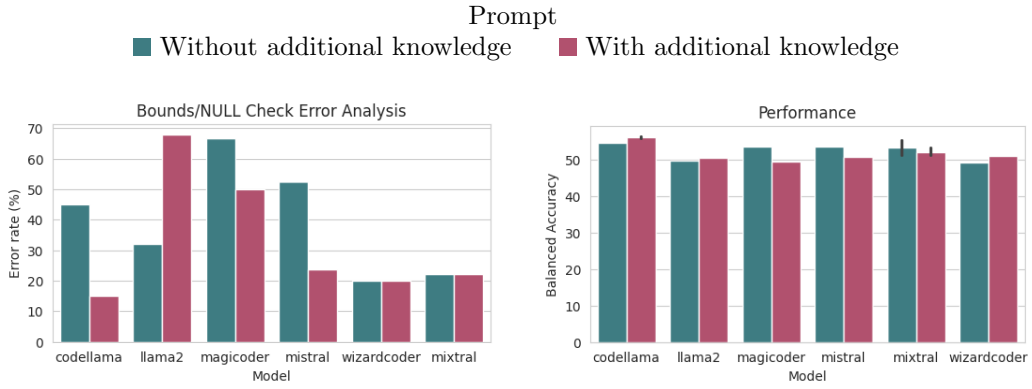
Example response

Reasoning: The function dereferences the pointer `cl` without checking if it is NULL. `cl` may be assigned to NULL in the line `cl = cl->super`. If `cl` is NULL, the function will crash when it tries to access the field `cl->tt` in the loop condition.
Answer: Yes, the function is buggy.

FIGURE 5.10. Example of our CoT-Annotations prompt using the NPD vulnerability from Figure 5.2. CoT-Annotations uses static analysis to provide information to the model (top) and provides reasoning example responses to encourage reasoning (bottom). We annotate null assignments, parameters, and dereferences to help the model to detect potential null-pointer dereferences and annotate null-checks to prevent the model from making false positive predictions.

Table 5.3 indicates that one of the important challenges that prevented LLMs from detecting vulnerabilities is their incapability of understanding *bounds/NULL checks* and *pointer operations*. Thus, we developed *Chain-of-Thought with Annotations (CoT-Annotations)*, shown in Figure 5.10. We introduced an external static analysis tool which annotates the code to highlight possible NULL assignments to pointers, NULL checks, on pointers, and dereferences of pointers. These annotations provide the exact information that defines the vulnerability and that a domain

expert would use to identify NULL-pointer dereference vulnerabilities. We integrated such knowledge into the prompt and evaluated performance on detecting Null-Pointer Dereference (NPD) vulnerabilities for the models we studied above, as a case study. As a quality measure, we manually verified the static analysis output and excluded incorrect annotations caused by heuristic errors.



(A) Reasoning errors related to bounds/NULL checks.

(B) Vulnerability detection performance.

FIGURE 5.11. Domain knowledge is somewhat helpful for one step but not much for overall performance. Some models, e.g., CodeLLAMA, respond to the annotated knowledge better: from a case study on NPD vulnerabilities.

By analyzing a sample of 198 responses[1] with/without annotations, we observed that CoT-Annotations reduced the errors of bounds/NULL checks recognition by 15-70% for Code LLAMA, MagiCoder, and Mistral, shown in Figure 5.11a; however, these models still missed 23-67% of bounds checks. We also observed that the improvement of understanding bounds/NULL checks did not significantly improve the models' performance (see Figure 5.11b). We speculate that this is because there are other blocking issues such as logical reasoning about relations of variables.

[1]This sample represents the intersection of vulnerable responses across all six models, with a maximum of 25 responses per model. We used the error categories established in Section 5.3, with each rater analyzing one-third of the responses (each response was reviewed by a single rater due to time constraints).

5.4 Related Work

Recent studies have initiated investigation into the usage of LLMs for vulnerability detection, using zero-shot prompting [43, 16], in-context learning [17, 30, 5], and fine-tuning [46, 59, 57]. Several papers have utilized chain-of-thoughts (CoT), such as "Let's think step-by-step" [27, 14, 47], multi-step prompts [51, 58], and generic information such as CFG, DFG, PDG, and API calls [60, 38, 26, 51]. In this work, we studied the most common prompting methods and proposed four novel prompt approaches tailored for vulnerability detection, integrating information from bug-fix commits (contrastive pairs), CVE descriptions (CoT-CVE), static analysis reports (CoT-StaticAnalysis), and domain knowledge annotations (CoT-Annotations). We further studied the LLMs' capabilities to distinguish buggy and patched versions of code and studied the reasoning errors in their responses.

Several recent papers have analyzed errors in LLM-generated vulnerability detection responses. Ullah et al. [51] used BLEU, ROUGE, and GPT-4 to automatically compare GPT-4's reasoning summaries with human-generated ones. Yu et al. [58] and Nong et al. [38] examined 82-100 responses from GPT-4 and GPT-3.5, supporting our findings that the models struggled with correctness, logic and consistency in general. However, existing studies do not match the depth and breadth of ours. Our error classifications provide more actionable and detailed categories, enabling us to identify specific code structures and LLM weaknesses (see Table 5.3). Additionally, we analyzed factors such as model size, training data, and training strategies, providing cause for concern about future improvements from model scaling. To our knowledge, our study is the most comprehensive manual analysis of LLMs for vulnerability detection, including 14 models and manually analyzing 300 LLM responses with a rigorous multi-rater agreement protocol.

5.5 Conclusion

In this chapter, we have show that vulnerability detection is complex, multistage reasoning task that current LLMs struggle to solve. We conducted a thorough study to show that the SOTA

models and prompts performed only slightly better than random guessing. None of the model advancements we explored led to significant improvements, including increasing model size, expanding training data, and instruction/adapter fine-tuning. The models particularly struggled to distinguish between vulnerable and fixed versions of code, where small textual differences cause large changes in semantics. We demonstrated that external tools and domain knowledge helped somewhat with single-step reasoning, but did not significantly improve the models' performance, which depends on accurate multi-step reasoning. Our findings bring concerns about further research in this area, raising the question of whether auto-regressively pre-trained LLMs are a good fit for tasks which require deep understanding of code semantics. We suggest that a fundamental shift in modeling and training methods may be necessary in order to overcome the reasoning failures of current LLMs. We believe that solving code reasoning in vulnerability detection could help address many other challenging tasks in software engineering, such as debugging, code execution prediction, test input generation, and program repair. We hope that this chapter laid out some key insights and motivation for the machine learning community to solve this important challenge.

5.6 Bibliography

[1] Brodersen, K. H., Ong, C. S., Stephan, K. E., and Buhmann, J. M. The balanced accuracy and its posterior distribution. In *20th International Conference on Pattern Recognition* (2010), ICPR '10, pp. 3121–3124.

[2] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 1877–1901.

[3] Calcagno, C., and Distefano, D. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium* (2011), Springer, pp. 459–465.

[4] Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering 48*, 09 (Sept. 2022), 3280–3296.

[5] Chan, A., Kharkar, A., Moghaddam, R. Z., Mohylevskyy, Y., Helyar, A., Kamal, E., Elkamhawy, M., and Sundaresan, N. Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning?, 2023. arXiv: 2306.01754.

[6] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. arXiv: 2107.03374.

[7] Chen, Y., Ding, Z., Alowain, L., Chen, X., and Wagner, D. 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* (New York, NY, USA, 2023), RAID '23, Association for Computing Machinery, p. 654–668.

[8] Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Castro-Ros, A., Pellat, M., Robinson, K., Valter, D., Narang, S., Mishra, G., Yu, A., Zhao, V., Huang, Y., Dai, A., Yu, H., Petrov, S., Chi, E. H., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q. V., and Wei, J. Scaling instruction-finetuned language models. *Journal of Machine Learning Research 25*, 70 (2024), 1–53.

[9] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021. arXiv: 2110.14168.

[10] CVE. CVE Website. https://www.cve.org/, 2024.

[11] Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., David Wagner, B. R., and Chen, Y. Vulnerability detection with code language models: How far are we? In *Proceedings of the 47th International Conference on Software Engineering* (2025), ICSE '25.

[12] Facebook. Infer Static Analyzer, 2024.

[13] Fan, J., Li, Y., Wang, S., and Nguyen, T. N. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (New York, NY, USA, 2020), MSR '20, Association for Computing Machinery, p. 508–512.

[14] Feng, S., and Chen, C. Prompting is all you need: Automated android bug replay with large language models. In *ICSE* (2024).

[15] Fleiss, J. L. Measuring nominal scale agreement among many raters. *Psychological Bulletin* (1971).

[16] Fu, M., Tantithamthavorn, C., Nguyen, V., and Le, T. ChatGPT for vulnerability detection, classification, and repair: How far are we? *arXiv:2310.09810* (2023).

[17] Gao, Z., Wang, H., Zhou, Y., Zhu, W., and Zhang, C. How far have we gone in vulnerability detection using large language models, 2023. arXiv: 2311.12420.

[18] Gemini Team. Gemini: A family of highly capable multimodal models. *arXiv:2312.11805* (2023).

[19] Gu, A., Rozière, B., Leather, H., Solar-Lezama, A., Synnaeve, G., and Wang, S. I. CRUXEval: A benchmark for code reasoning, understanding and execution, 2024.

[20] Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., and Liang, W. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence, 2024.

[21] He, J., and Vechev, M. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2023), CCS '23, Association for Computing Machinery, p. 1865–1879.

[22] HuggingFaceH4 Team. HuggingFaceH4/starchat2-15b-v0.1. https://huggingface.co/HuggingFaceH4/starchat2-15b-v0.1, 2024.

[23] Islam, M. J., Nguyen, G., Pan, R., and Rajan, H. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2019), ESEC/FSE '19, Association for Computing Machinery, p. 510–520.

[24] Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7b, 2023. arXiv: 2310.06825.

[25] Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts, 2024. arXiv: 2401.04088.

[26] Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R., and Naik, M. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv:2311.16169* (2023).

[27] Li, H., Hao, Y., Zhai, Y., and Qian, Z. The hitchhiker's guide to program analysis: A journey with large language models. *arXiv:2308.00245* (2023).

[28] Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. StarCoder: may the source be with you!, 2023. arXiv: 2305.06161.

[29] Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Huang, M., Dong, Y., and Tang, J. AgentBench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations* (2024).

[30] Liu, Z., Liao, Q., Gu, W., and Gao, C. Software vulnerability detection with GPT and in-context learning. In *2023 8th International Conference on Data Science in Cyberspace (DSC)* (2023), pp. 229–236.

[31] Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauss, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. StarCoder 2 and the Stack v2: The next generation, 2024. arXiv: 2402.19173.

[32] Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. WizardCoder: Empowering code large language models with evol-instruct. *arXiv:2306.08568* (2023).

[33] MITRE. CVE-2017-9211, 2024.

[34] MITRE. CVE-2018-16435, 2024.

[35] MITRE. CWE - Common Weakness Enumeration. https://cwe.mitre.org/index.html, 2024.

[36] NIST. NIST Software Assurance Reference Dataset. https://samate.nist.gov/SARD/, 2024.

[37] NIST. NVD - NVD Dashboard. https://nvd.nist.gov/general/nvd-dashboard, 2024.

[38] Nong, Y., Aldeen, M., Cheng, L., Hu, H., Chen, F., and Cai, H. Chain-of-Thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv:2402.17230* (2024).

[39] OpenAI. gpt-3.5-turbo-0613 announcement, June 2023.

[40] OpenAI. GPT-4 technical report, 2024. arXiv: 2303.08774.

[41] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems* (2022), S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., pp. 27730–27744.

[42] PapersWithCode. HumanEval Benchmark, 2023. [Accessed 27-10-2023].

[43] Purba, M. D., Ghosh, A., Radford, B. J., and Chu, B. Software vulnerability detection using large language models. In *IEEE 34th International Symposium on Software Reliability Engineering Workshops* (2023), pp. 112–119.

[44] Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code Llama: Open foundation models for code, 2024.

[45] Saldaña, J. *The coding manual for qualitative researchers*. SAGE publications Ltd, 2021.

[46] Shestov, A., Levichev, R., Mussabayev, R., Maslov, E., Cheshkov, A., and Zadorozhny, P. Finetuning large language models for vulnerability detection, 2024. arXiv: 2401.17010.

[47] Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., and Liu, Y. When GPT Meets Program Analysis: Towards Intelligent Detection of Smart Contract Logic Vulnerabilities in GPTScan, Aug. 2023. arXiv: 2308.03314.

[48] Talmor, A., Herzig, J., Lourie, N., and Berant, J. CommonsenseQA: A question answering challenge targeting commonsense knowledge. 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)* (Minneapolis, Minnesota, June 2019), J. Burstein, C. Doran, and T. Solorio, Eds., Association for Computational Linguistics, pp. 4149–4158.

[49] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023. arXiv: 2307.09288.

[50] Tunstall, L., Lambert, N., Rajani, N., Beeching, E., Le Scao, T., Han, S., Schmid, P., Von Werra, L., and Rush, S. Creating a coding assistant with starcoder. *Hugging Face Blog* (2023).

[51] Ullah, S., Han, M., Pujar, S., Pearce, H., Coskun, A., and Stringhini, G. Can large language models identify and reason about security vulnerabilities? not yet. *arXiv:2312.12575* (2023).

[52] Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., and Fedus, W. Emergent abilities of large language models, 2022. arXiv: 2206.07682.

[53] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2024), NIPS '22, Curran Associates Inc.

[54] Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).

[55] Wikipedia. Natural experiment. http://en.wikipedia.org/w/index.php?title=Natural%20experiment&oldid=1235833118, 2024. [Online; accessed 20-September-2024].

[56] Xie, S. M., and Min, S. How does in-context learning work? A framework for understanding the differences from traditional supervised learning. https://ai.stanford.edu/blog/understanding-incontext/, 2022.

[57] Yang, A. Z. H., Le Goues, C., Martins, R., and Hellendoorn, V. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[58] Yu, J., Liang, P., Fu, Y., Tahir, A., Shahin, M., Wang, C., and Cai, Y. An insight into security code review with llms: Capabilities, obstacles and influential factors, 2024. arXiv: 2401.16310.

[59] Yusuf, I. N. B., and Jiang, L. Your instructions are not always helpful: Assessing the efficacy of instruction fine-tuning for software vulnerability detection. *arXiv:2401.07466* (2024).

[60] Zhang, C., Liu, H., Zeng, J., Yang, K., Li, Y., and Li, H. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (New York, NY, USA, 2024), ICSE-Companion '24, Association for Computing Machinery, p. 276–277.

[61] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. A survey of large language models, 2024. arXiv: 2303.18223.

[62] Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., and Su, Z. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA, 2021), ICSE-SEIP '21, Association for Computing Machinery.

[63] Zhou, X., Zhang, T., and Lo, D. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (2024), pp. 47–51.

[64] Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems 32* (2019).

5.A Appendix: Vulnerability detection prompts

Basic (zero-shot) prompting [16]: We first designed a system prompt to set the context: "I want you to act as a vulnerability detection system", along with natural-language instructions: (1) *Basic query*: "Is the following function buggy? Please answer Yes or No." (We also tried "Is the following function vulnerable?"; however, our pilot study shows that it did not perform as well.) (2) *CWE list*: This prompt starts with "Does the following function contain one of the following bug types?", followed by a fixed list of bug types, e.g., "CWE-190: Integer Overflow"; (3) *Q/A*: Begin the query with "Question:" and begin the model's response with "Answer:". This conditions the model to respond in a question-answering mode.

In-context (n-shot) prompting[30, 63]: In this prompt, we provide examples of inputs and responses for in-context learning [2]. The in-context examples condition the model to reply in the same format as the example responses [56]. The selection of in-context examples can impact the performance. We studied three settings: (1) randomly selected examples, (2) the examples that had similar embeddings to the query example, and (3) the examples from *contrastive pairs* (see below for details).

In-context prompting based on contrastive pairs: We formed *contrasting pairs* of in-context examples by providing the vulnerable version of the code (before the bug-fixing commit) and the fixed version (after the commit) as in-context examples in the same prompt. Since these two versions of the source code differ primarily in the portion related to the bug-fix, our intention is that this prompt template would highlight the cause of the bug and instruct the model to learn that the small differences in code can lead to different labels.

In-context prompting based on CoT from CVE descriptions: We designed "chain-of-thought" prompts by providing intermediate reasoning steps which lead to the answer, inspired by Wei et al. [53]. We use in-context examples from the Big-Vul dataset [13], which includes the CVE bug reports. For vulnerable examples, we used the default in-context query and provide the chain-of-thought response. To produce such response, we adapt the descriptions in

these bug reports to describe how the bug manifests. For example, CVE-2017-9211 [33] describes the vulnerability, including the symptoms, attack surface, and variable involved:

```
The crypto_skcipher_init_tfm function in crypto/skcipher.c in the Linux kernel through
4.11.2 relies on a setkey function that lacks a key-size check, which allows local users
to cause a denial of service (NULL pointer dereference) via a crafted application.
```

We use this description as the CoT response and append "Therefore, the example is buggy" to complete the example response. For non-vulnerable examples, we provide the default in-context example query/response.

In-context prompting based on CoT from static analysis: We also used the output buggy paths reported by static analysis tools to prepare the chains of thought prompt. The buggy path consists of a list of statements that can lead to the bug. We use in-context examples from the D2A dataset [62], which lists buggy paths from the Infer static analyzer [12] for several open-source C++ projects. We convert the buggy paths to natural language descriptions and use them as the response. This is an example CoT response for a buffer overflow vulnerability:

```
1. A buffer buf of size 10 is allocated at line 1.
2. An index i is initialized to a value in the range [0, 100] at line 2.
3. The index i is used to access buf at line 3. This may exceed the bounds of buf.
```

We append "Therefore, the example is buggy" to complete the example response. For non-vulnerable examples, we provide the default response.

5.B Appendix: Models

We used the model sizes shown in Table 5.4 and the text generation parameters shown in Table 5.5 for our experiments. The model IDs are documented in our data package.

TABLE 5.4. 14 models we studied.

Model	Parameters	Context Length
GPT-4 [40]	-	128k
Gemini 1.0 Pro [18]	-	32k
GPT-3.5 [39]	-	4k
Mixtral-MoE [25]	45B	8k∼128k
Code LLAMA [44]	7B, 13B, 34B	16k∼100k
LLAMA 2 [49]	7B, 13B	4k
WizardCoder [32]	33B	2k
DeepSeek-Coder [49]	1.3B, 6.7B, 33B	4k
StarChat2 [22]	15.5B	16k
StarCoder2 [22]	15.5B	16k
StarChat [50]	15.5B	8k
StarCoder [28]	15.5B	8k
MagiCoder [54]	7B	16k∼100k
Mistral [24]	7B	8k∼128k

TABLE 5.5. Text generation parameters we used.

Parameter	HuggingFace	OpenAI	Google
Top-p	0.9	1.0	1.0
Temperature	0.1	0.1	0.1
Max. tokens generated	512	512	512

5.C Appendix: Benchmarks for other domains

We gathered the benchmark performance results for Table 5.1 from public benchmarks and from the papers associated with each model:

- CruXeval: https://crux-eval.github.io/leaderboard.html
- HumanEval: https://paperswithcode.com/sota/code-generation-on-humaneval
- GSM8k: reported in the models' papers [49, 24, 25, 39, 40, 18].
- CSQA: reported in the models' papers [24, 25, 39, 40, 18].

Integer Overflow

```
int determineFirstQuarterRevenue()
{

    // Variable for sales revenue for the quarter
    float quarterRevenue = 0.0f;

    short JanSold = getMonthlySales(JAN); /* Get sales in January */
    short FebSold = getMonthlySales(FEB); /* Get sales in February */
    short MarSold = getMonthlySales(MAR); /* Get sales in March */

    // Calculate quarterly total
    short quarterSold = JanSold + FebSold + MarSold;

    // Calculate the total revenue for the quarter
    quarterRevenue = calculateRevenueForQuarter(quarterSold);

    saveFirstQuarterRevenue(quarterRevenue);

    return 0;
}
```

FIGURE 5.12. A simple integer overflow example collected from CWE database. The summation of three short int numbers can be more than the value of the maximum short int number, which will introduce integer overflow vulnerability.

5.D Appendix: Simple CWE examples

Figure 5.12 shows a simple integer overflow vulnerability example. The CodeLLAMA model could detect this vulnerability. The response of the CodeLLAMA model is shown in Figure 5.13. The performances of all models on these simple examples are shown in Table 5.6

Example response

No, the function does not contain any of the following vulnerability types:
* CWE-190: Integer Overflow
* CWE-476: Null Pointer Dereference
* CWE-125: Out-of-bound Read
* CWE-787: Out-of-bound Write
* CWE-416: Use After Free

FIGURE 5.13. Code LLAMA's response to the simple example in Figure 5.12.

TABLE 5.6. The performance of the studied models on simple CWE examples.

Model	Performance
GPT-4	65.78
Gemini 1.0 Pro	50.87
GPT-3.5	56.14
Mixtral-MoE	61.40
Code LLAMA	61.40
LLAMA 2	46.49
WizardCoder	51.75
DeepSeek-Coder	66.67
StarChat2	55.26
StarCoder2	50.87
StarChat	50.00
StarCoder	41.52
MagiCoder	62.28
Mistral	57.01

5.E Appendix: Error analysis methodology

5.E.1 Inter-rater agreement

We first analyzed 50 examples to create detailed error categories for each reasoning step. All three raters independently identified errors in the LLM responses, refining the protocol after processing ⅓, ½, and all of the data. We added new error categories when needed, and merged similar categories after analysis concluded. We measured inter-rater agreement using Fleiss' kappa (κ) [15], achieving 0.78 with 86% agreement. We resolved disagreements by majority vote, followed by discussion for the final categorization. After the categories were set in Section 5.3, we used one rater to analyze the responses reported in Sections 5.3.1 to 5.3.3.

5.E.2 Error analysis UI

Figure 5.14 is a screenshot of the user interface (UI) used by the raters for error analysis. The interface features a display of the source code (center), the model's response and explanation (bottom left), and metadata about the vulnerability (top left). Additionally, it provides a

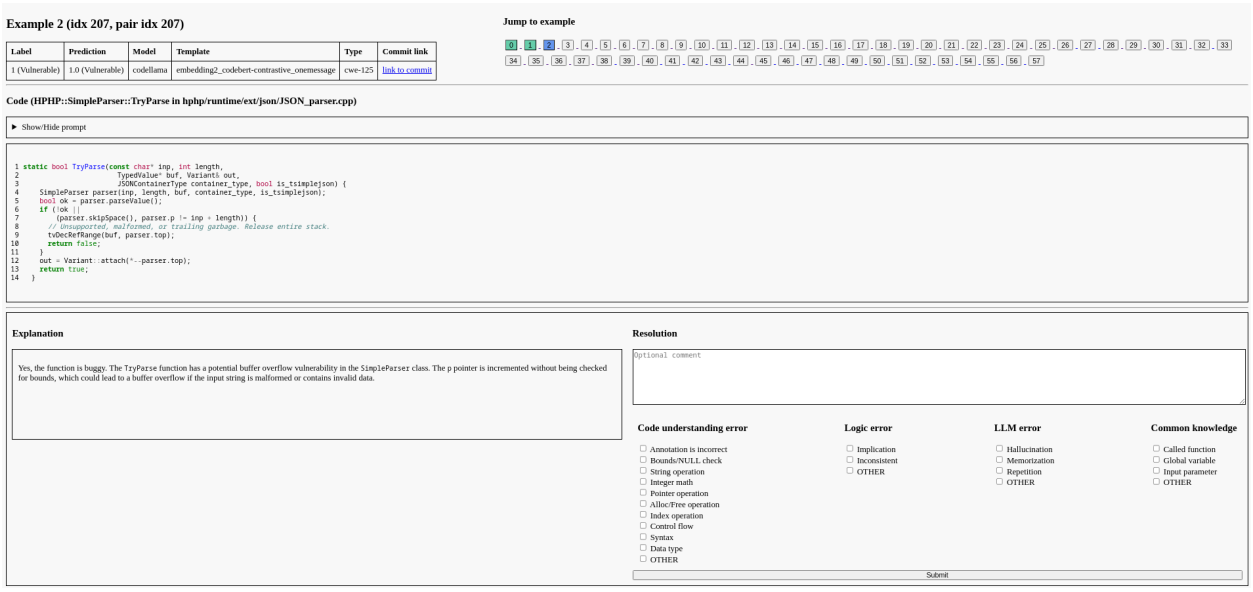


FIGURE 5.14. Error analysis user interface.

configurable set of checkboxes to select one or more error categories, along with a section for free-form text notes (bottom right). We believe that this tool could be valuable for future large-scale manual analyses of LLM responses, which is why we have included it in our data package.

5.E.3 Error categories

Table 5.7 shows the definitions for error categories, which we developed in our manual analysis.

TABLE 5.7. Definitions of Model Reasoning Errors.

Reasoning Step	**Category**	**Description**
(1,2) Localizing and understanding statements related to vulnerability	Misunderstood Bounds/NULL check	Does the model state a false proposition about a bounds check or null check, such as "if (ptr) *ptr" or "if (i ¡ len) buf[i]"?
	Misunderstood string operation	Does the model state a false proposition about allocation, copy, reading, or writing of strings?
	Misunderstood arithmetic operation	Does the model state a false proposition about an arithmetic operation, such as +, -, /, *?
	Misunderstood pointer operation	Does the model state a false proposition about a statement involving a pointer dereference operation?
	Misunderstood alloc/free operation	Does the model state a false proposition about a memory allocation such as malloc or new?
	Misunderstood index operation	Does the model state a false proposition about a statement involving an array index operation?
	Misunderstood execution order	Does the model state a false proposition about a conditional, such as if or switch, or the order of execution between two statements?
	Improper assumption	Does the model make an unreasonable assumption about a function, variable, or parameter in the example?
	Misunderstood syntax	Does the model misinterpret the syntax of visible code, e.g. interpreting the declaration `int *x = NULL;` as a dereference of x?
(3) Logical reasoning	Faulty implication	Does the model make a logical implication where the conclusion does not follow from the premise(s)?
	Inconsistent	Does the model make any statements within its response which are contradictory?
Cross-cutting errors	Hallucination	Does the model reason about code that isn't there?
	Memorization	Does the model reason about code that is potentially memorized from the training data, such as talking about the calling context?
	Repetition	Does the model output repeated sentences which don't make sense in sequence?

CHAPTER 6. CLOSING THE GAP: A USER STUDY ON THE REAL-WORLD USEFULNESS OF AI-POWERED VULNERABILITY DETECTION & REPAIR IN THE IDE

Benjamin Steenhoek[1*], Siva Sivaraman[2], Renata Saldívar Gonzalez[3], Yevhen Mohylevskyy[4], Roshanak Zilouchian Moghaddam[5], and Wei Le[6]

[1,6] Department of Computer Science, Iowa State University, Ames, IA, 50011

[2-5] Data & AI team, Microsoft, Redmond, WA, 98052

Modified from a manuscript published in the *47th International Conference on Software Engineering (ICSE 2025)*

Abstract

Security vulnerabilities impose significant costs on users and organizations. Detecting and addressing these vulnerabilities early is crucial to avoid exploits and reduce development costs. Recent studies have shown that deep learning models can effectively detect security vulnerabilities. Yet, little research explores how to adapt these models from benchmark tests to practical applications, and whether they can be useful in practice.

This chapter presents the first empirical study of a vulnerability detection and fix tool with professional software developers on real projects that they own. We implemented DeepVulGuard, an IDE-integrated tool based on state-of-the-art detection and fix models, and show that it has promising performance on benchmarks of historic vulnerability data. DeepVulGuard scans code for vulnerabilities (including identifying the vulnerability type and vulnerable region of code), suggests fixes, provides natural-language explanations for alerts and fixes, leveraging chat

[*] Work primarily done during an internship at Microsoft.

interfaces. We recruited 17 professional software developers, observed their usage of the tool on their code, and conducted interviews to assess the tool's usefulness, speed, trust, relevance, and workflow integration. We also gathered detailed qualitative feedback on users' perceptions and their desired features. Study participants scanned a total of 24 projects, 6.9k files, and over 1.7 million lines of source code, and generated 170 alerts and 50 fix suggestions. We find that although state-of-the-art AI-powered detection and fix tools show promise, they are not yet practical for real-world use due to a high rate of false positives and non-applicable fixes. User feedback reveals several actionable pain points, ranging from incomplete context to lack of customization for the user's codebase. Additionally, we explore how AI features, including confidence scores, explanations, and chat interaction, can apply to vulnerability detection and fixing. Based on these insights, we offer practical recommendations for evaluating and deploying AI detection and fix models. Our code and data are available at this link: https://doi.org/10.6084/m9.figshare.26367139.

6.1 Introduction

Security vulnerabilities impact users' safety, security, and privacy and cost organizations millions of dollars per year [29, 24], with reports of breaches exposing millions of records becoming commonplace [3]. Early detection of vulnerabilities during the development phase can greatly reduce costs and mitigate potential impacts [7, 4, 23]. In recent years, deep learning (DL) vulnerability detection models have emerged as a promising approach for scanning code during software development [11, 46, 20]. These models can identify vulnerability patterns in code snippets and offer the advantage of analyzing code during editing [12] with less configuration than traditional static analysis tools [21].

Despite promising benchmark performance [11, 46, 20], it remains unclear whether these models are actually useful in real-world development settings. In the past, Major organizations such as Microsoft [15], Google [42], Facebook [17], and Coverity [5] have reported a gap between benchmarking success and practical application with static analyzers. Recently, Fu et al. [21]

conducted a preliminary controlled study with 6 developers, showing that AI tool support reduced the time to diagnose and fix a vulnerability from 10-15 minutes to 3-4 minutes and motivating further user studies of AI detection and fix tools. However, their study used a single bug from their dataset rather covering real-world code-bases and they only studied vulnerability detection and fixing.

In our work, we recruited 17 professional developers from a major software company in a *real-world development setting* with their own projects; beyond detection and fixing, we also built and studied AI-powered *explanation and chat interfaces*, which have recently become prominent in the integrated development environments (IDEs) [1]. Our study provides a deeper understanding of the real-world usefulness and nuances of deploying these models.

To carry out our study, we developed DeepVulGuard, an extension integrated with Visual Studio Code (VSCode) [34], a popular IDE with over 14 million active users. We used state-of-the-art models, CodeBERT [19, 12] and the GPT-4 large language model (LLM) [41], for detection and fix tasks. Participants scanned 24 projects, 6.9k files, and over 1.7 million lines of source code, generating 170 alerts and 50 fix suggestions. To the best of our knowledge, ours is the first study to evaluate a detection and fix tool with professional developers on their own projects.

We initially evaluated DeepVulGuard's potential for deployment by testing its detection and fix models on established vulnerability datasets. Our models achieved 80% precision, 32% recall, and a 46% F1 score on SVEN [22] for vulnerability detection and fixed 13% of vulnerabilities on the Vul4J [9] dataset. DeepVulGuard performs comparably or better than state-of-the-art models [16, 52, 8] and meets the threshold for acceptable false positives [15]. These results indicate that our models are promising for detecting and fixing security vulnerabilities and can generate meaningful results for the user study.

Our results show that 59% of participants expressed interest in future use of DeepVulGuard, although there are several issues that limit its usefulness. For example, one problem was an high rate of false positives in practice, caused by incorrect vulnerability pattern recognition and lack of context about code snippets (e.g., inter-procedural vulnerabilities). This highlights the need for

more precise pattern recognition and better integration of environment and program context. Additionally, the requirement to trigger a manual scan significantly disrupted the users' workflow; developers prefer tools that run in the background and alert them whenever potential vulnerabilities are detected. Regarding fixes, 75% of proposed security fixes were unsuitable to apply "as-is" due to lack of customization and incorrect integration into the code. Although some fixes were functionally correct, they were not tailored to the user's codebase and could not be applied without significant modifications. An interactive chat method shows promise to allow developers to guide the generation towards more applicable fixes. Our findings offer concrete recommendations for improving these pain points found in these tools.

We make the following research contributions:

1. We developed DeepVulGuard a VSCode extension for detecting, explaining and fixing vulnerabilities, incorporating insights from static analysis and AI tool research. Our tool allows customization of backend models, and we provide its code in our data package to support further user studies. DeepVulGuard uses a multi-task training approach for jointly predicting vulnerability classification, localization, and bug type. We also introduced a new vulnerability filtering method with LLMs which improved precision by over 20%.

2. We conducted a user study with 17 professional software engineers at a major software company. Through interviews and surveys as they ran our tool on their own code, we quantitatively assessed multiple dimensions of usefulness for detection and fix tools and provided practical recommendations for improving deep learning-based vulnerability detection and fix tools.

6.2 User Study Interface

To study whether deep learning-based vulnerability tools can be useful in practice, we built DeepVulGuard, a Visual Studio Code extension that brings state-of-the-art detection + fix techniques to an IDE interface. DeepVulGuard allows users to (1) scan source code with CodeBERT and LLM models, (2) view the reported vulnerabilities and LLM-generated

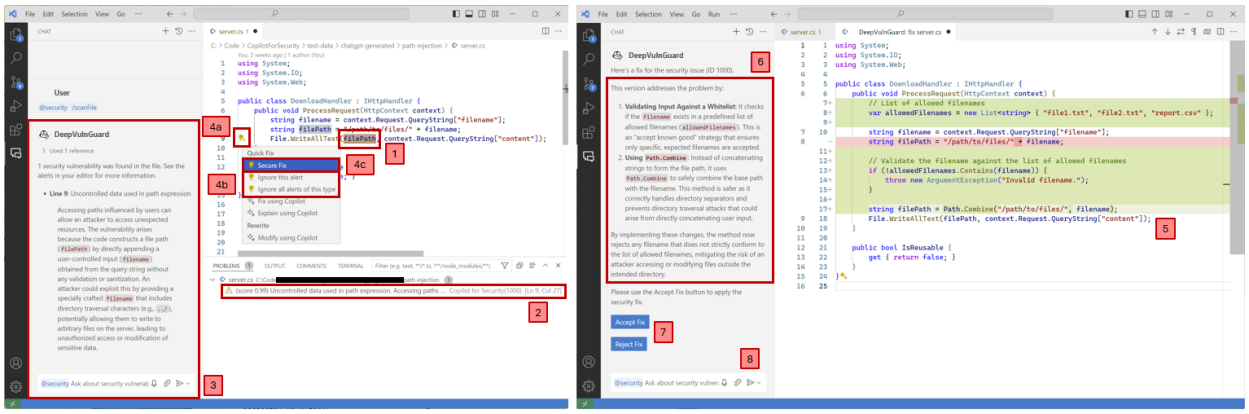


Figure 6.1: An overview of DeepVulGuard's user interface on an example program. (1) An editor alert; (2) Problems menu entry; (3) The explanation of the alert; (4a) Quick fix interaction; (4b) Ignore options; (4c) Fix trigger; (5) Suggested fix; (6) Explanation of the fix suggestion; (7) Accept/Reject buttons.

explanations directly inside the editor, and (3) generate suggestions for mitigating the vulnerability. We also implemented a telemetry module to collect user data, enabling longitudinal studies of AI-based vulnerability detection tools. As our study is the first of its kind in this area, we believe our tool will be a beneficial contribution which facilitates future user studies of vulnerability tools. We released the extension code in our data package. The code can be easily adjusted to call alternative detection and fixing solutions to be studied with developers in a real-world setting.

6.2.1 IDE Integration

Figure 6.1 shows an overview of DeepVulGuard's user interface. Users begin by requesting to scan a file or directory. If any potential vulnerabilities arise, they are shown as highlights in the editor (1) and actionable entries in the Problems window (2), and a natural-language explanation of the vulnerability is shown in the chat panel (3). The user can use this information to assess the vulnerability and decide if a fix is required. They can also ask questions or make suggestions to the chatbot by sending follow-up messages. By clicking on the *quick fix* lightbulb (4a), the user

can ignore the specific alert or alert types (4b), or generate a *quick fix* (4c). On requesting the quick fix, the suggested code modifications will be presented in a *diff* view (5), showing the lines to be removed and added. As well, an explanation of the fix is shown in the chat panel (6). The user can modify the fix in the editor or suggest improvements with natural-language chat messages if desired, then Accept it to apply it to their files or Reject it to revert to the original code (7). Users can enter chat messages (8), e.g. asking for clarification, information, or inputs which trigger the vulnerability, and our tool will generate a conversational response.

We drew inspiration for our tool's design from several foundational research studies on static analyzers and AI-assisted developer tools. Johnson et al. [27] showed that developers requested static analysis tools to be available in the IDE, along with quick fixes, and the ability to modify rule sets. Similarly, Christakis et al. [15] identified bad warning messages, lack of suggested fixes, and poor visualization as pain points. Smith et al. [45] presented design guidelines, such as presenting alerts in actionable locations, integrating with their workflow by tracking progress, batch processing, allowing code editing during scans, and scalability of the interface. We incorporated all of these features into DeepVulGuard.

A recent study on AI-powered code completion Wang et al. [50] found that users in focus groups valued the ability to view a measure of the model's confidence. To study this in a practical implementation, we integrated confidence scores into our tool's alerts, shown in Figure 6.1 (2). Fu et al. [21] conducted a survey study and found that most participants valued localizations, CWE type prediction, and quick fixes, so we integrated these features into our tool and evaluate them in our study, shown in Figure 6.1 (1, 2, and 4a).

6.2.2 Model Architecture & Training

Our tool can be easily configured to leverage a wide variety of deep learning models or static analyzers. Figure 6.2 shows the workflow of the current design; specifically, we implemented the following techniques (please refer to our data package [2] for the implementation details, including our model training procedure, dataset statistics, and hyper-parameters).

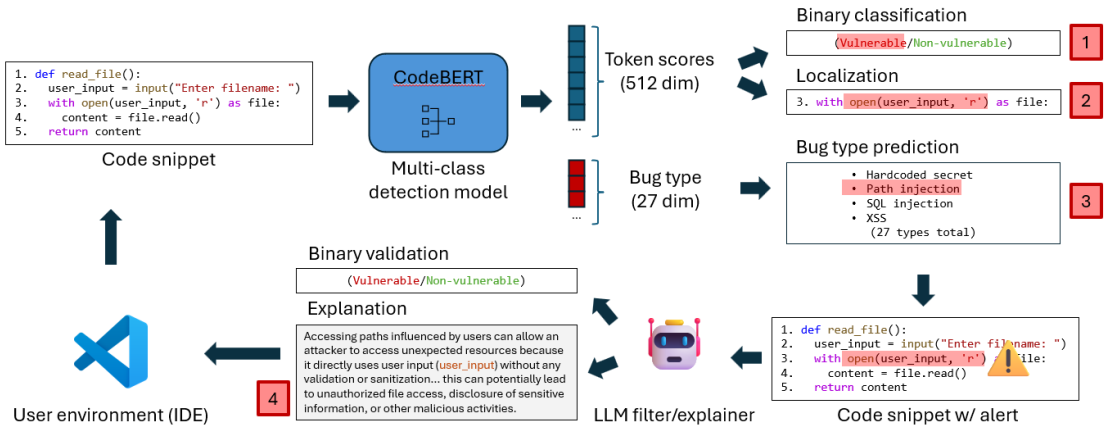


Figure 6.2: An overview of DeepVulGuard's detection workflow. (1) Binary classification into vulnerable/not-vulnerable; (2) Localization; (3) Multi-class classification into one of 27 vulnerability types; (4) Alert and explanation shown to the user.

Fine-tuning CodeBERT for multi-task vulnerability detection: CodeBERT [19] and similar models consistently perform well on various vulnerability datasets [20, 21, 46] with relatively low latency which is suitable for detection in the editor [12]. We fine-tuned CodeBERT using *multi-task learning* to (1) predict whether a code snippet contains a vulnerability, (2) localize the tokens causing it, and (3) identify the vulnerability type. We trained on a dataset of over 1.3 million alerts labeled by CodeQL in GitHub projects following Chan et al. [12]'s methodology, focusing on 27 vulnerability types related to Web security, e.g. Path Injection, SQL Injection, Hard-coded Credentials, Unvalidated URL Redirect, Cross-Site Scripting (details in data package). We generate alerts in the extension based on the predicted vulnerability type, confidence score, and localization.

```
'''
You are a vulnerability detector. Only respond with "Yes" or "No" and an explanation. Does the
↪ following code snippet contain a SQL Injection vulnerability at line marked by ALERT?
'''
```

Figure 6.3: DeepVulGuard's LLM filter prompt.

Filtering and explaining alerts with GPT-4: To further filter the false positives produced by fine-tuned CodeBERT, we used GPT-4 [41] to filter the alerts and generate explanations. We annotated the code snippet with a comment describing the alert type at the localized line, e.g., for SQL injection: `// ALERT: This SQL query depends on a user-provided value` (see our data package for all types of annotations). Then we instructed GPT-4 using the prompt shown in Figure 6.3. If the answer is *Yes*, the alert and explanation are shown to the user; otherwise, the alert is not shown ((4) in Figure 6.2).

```
'''
A static analyzer has identified a {rule_id} security vulnerability in the {language} method
↪  below:

```
{method}
```

The SARIF result message is as follows: {message}

{description}

Write a fixed version of the method above and wrap it in triple backticks, then explain why your
↪ version addresses the problem.
'''
```

Figure 6.4: DeepVulGuard's fix model prompt.

Prompting GPT-4 for repair and explanation: We used GPT-4 with custom prompts to generate and explain code fixes. The prompt, shown in Figure 6.4, includes the source code, vulnerability report, and an instruction to provide a fixed version of the code and an explanation. We displayed the explanation in the chat panel and inserted the code suggestion to show a diff with the original content, shown in Figure 6.1 on the right.

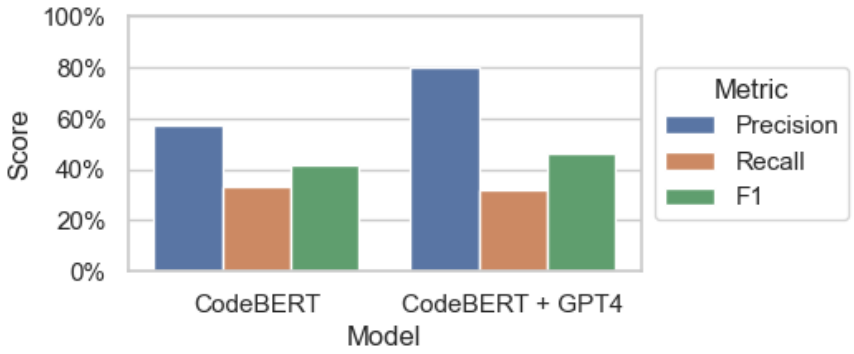


Figure 6.5: Performance of DeepVulGuard's detection component on SVEN.

6.2.3 Evaluating Detection and Fix Capabilities

To ensure that DeepVulGuard is both effective and representative of the state-of-the-art, we tested its performance on benchmarks that resemble the real-world deployment scenario as closely as possible. To evaluate DeepVulGuard's detection capability, we used: the SVEN dataset [22] with 380 high-quality vulnerability examples from open-source Python projects (93% label accuracy [16])). Our detection model supports all the security vulnerability types present in the dataset. We used a strict definition for true positives: the predicted bug type must match, and localized line number must match the lines changed in the patch. Figure 6.5 shows on the SVEN dataset our model achieved 80% Precision and 32% Recall, with an F1 score of 46%.

Overall, our results are better than or on par with the prediction quality of SOTA models on vulnerability detection. For example, most recently, Ding et al. [16] reported that SOTA models, including CodeBERT, attained 18-21 F1 score on their dataset of C/C++ vulnerabilities. We cannot directly compare our model with other SOTA models on our dataset as most are trained on C/C++-specific memory or pointer bugs [32, 20, 46, 11, 47, 16].

Christakis et al. [15] found that most developers tolerate up to a 20% false-positive rate; with the LLM filter, our model meets this threshold on the SVEN dataset, with 80% precision. These results highlight DeepVulGuard's practical effectiveness and potential for deployment in real-world applications.

To evaluate DeepVulGuard's fix component, we used the Vul4J [9] dataset, which includes executable tests to reproduce security vulnerabilities. We assessed the test results, supplemented by manual validation, to verify that the suggested fixes mitigated issues without breaking other functionality. Among the 24 single-hunk bugs with vulnerability types that our tool handles, our model produced 3 (13%) correct fixes and 2 (8%) partial fixes, which resolved the issue but broke 1-3 other tests; 10 (42%) fixes had errors inserting the generated code into the file and 9 (37%) fixes could not compile. For efficiency needed for using in IDE, we chose to not run LLM multiple times. These results show that our model performs similarly to SOTA evolution-based automated program repair (APR) tools [8] (13% correct fixes, taking up to 7 minutes in the 75th percentile, intersection $n = 24$) and LLMs such as Codex [52] (15.4% plausible fixes on the first try, intersection $n = 13$).

We conducted the above performance probe to confirm that DeepVulGuard can be used to conduct a meaningful study; that it is practical for handling real-world vulnerabilities and offers performance comparable to state-of-the-art techniques. We did not aim for a comprehensive controlled evaluation to claim that DeepVulGuard outperforms the current state of the art.

6.3 User Study Design

We developed three research questions to guide our study.

RQ1: Is DeepVulGuard useful in practice?

RQ2: Which aspects of vulnerability detection + fix tools are most useful?

RQ3: What features do developers want from vulnerability detection + fix tools?

6.3.1 Study Design

We carried out an *exploratory case study* [18] with a group of 17 professional developers from a major software company. We asked users to run DeepVulGuard on projects they were actively developing or were familiar with, and answer survey questions about their perception of the tool. To the best of our knowledge, our tool is the first vulnerability detection + fix tool to be studied

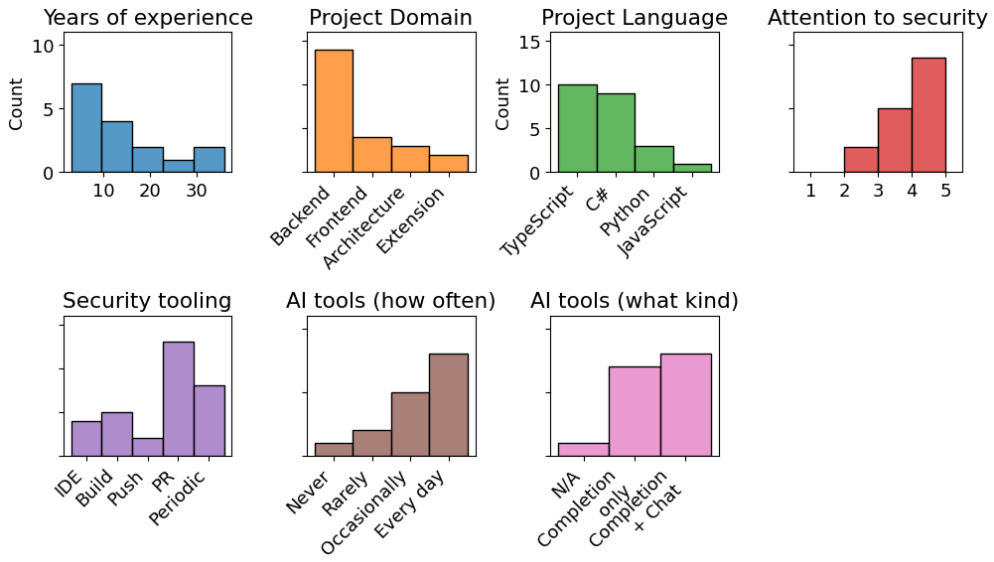


Figure 6.6: Participant demographics and tool adoption. Where applicable, participants listed multiple items for the project domain, project language, and security tooling.

in a real-world setting with professional developers on projects which they own. This study enabled us to explore many open questions such as the role of explanations, the developers' tolerance for false-positives or delayed results, and what constitutes an effective fix in a secure development context. We chose an exploratory study over, e.g. a controlled study, because it elicits rich feedback from developers in a real-world setting. Our approach takes advantage of the developers' deep understanding of their own projects, leading to a more accurate assessment of potential vulnerabilities and providing more valuable insights.

Recruitment: We carried out our study with a group of 17 professional developers recruited from a major software development company. We recruited developers primarily using snowball sampling, with a 53% participation rate. In total, participants scanned a total of 24 projects, 6.9k files, and over 1.7 million lines of source code, and generated 170 alerts and 50 fix suggestions.

Figure 6.6 shows the participants' demographic information (with the exception of one participant who declined the demographic survey), indicating that we studied a diverse set of developers from various levels of experience and backgrounds. Participants had a median of 11 years of experience; participants worked on both front-end and back-end domains and represented

a diverse set of applications such as web applications, back-end services, IDE extensions. Most projects were written in C# or TypeScript. Most participants considered themselves security-conscious (median 4/5), and all had some form of a static analysis tool running in continuous integration or periodically, though more than half of the projects used tools for reasons of organizational compliance rather than individual initiative; most participants did not use security tools in the IDE. The majority of participants used AI-powered tools occasionally (a few times a week) or every day, such as code completion tools or chatbots, though three participants used AI tools rarely or not at all, stating that they did not find them useful.
Interviews: Each participant ran our tool on a code-base associated with a production application which they actively develop and shared their perspective guided by both structured and free-form questions. We first asked the participants to run our tool on a simple web server containing a known vulnerability to introduce the features of our tool: detection, fix, and chat. Then, we directed the participants to run the tool on security-critical areas of their application, such as web interfaces, API endpoints, database code, and file processing code.

We ran the study as a *think-aloud empirical study* [43, 30], meaning we asked developers to verbalized their thoughts while running the tool and processing the results. When participants explicitly asked questions, we provided help and answered questions to facilitate a smooth interview process and clarify the participant's statements, e.g., about the meaning of different UI elements, bug type descriptions, or behavior of the tool, but we refrained from explaining the results of the tool or interpreting the meaning of its outputs to avoid biasing the study. Each interview lasted approximately 50 minutes, consisting of a 10-minute setup and demographic survey, average 28 minutes usage of the tool and 12 minutes post-usage survey and discussion. We interviewed all subjects over video calls, and with their full consent, recorded field notes and demographic, audio, screen-capture, survey, and tool usage data.

After they used the tool, we asked participants about various aspects of the tool: (1) their overall perception of the usefulness of the detection alerts and suggested fixes and their satisfaction with the speed (Q1-Q3, reported on a Likert scale from 1 to 5 from "not

useful/satisfied at all" (1) to "very useful/satisfied" (5)); (2) whether the tool fits their workflow, whether they trust in the tool, whether the reported alert types were relevant, and whether they would keep using the tool (Q4-Q7, reported as Yes/No). We also asked the participants what features they found especially useful and what features they would like to see in the tool. We asked the questions verbally during the interview, immediately after trying the tool, in order to collect free-form feedback on each question and ensure that the participant could recollect their experiences with the tool.

We designed the initial set of interview questions, guided by our research questions and informed/inspired by findings and open questions from previous studies of static analysis and AI tools [27, 15, 21, 50, 45, 6, 35]. Three authors tried the tool and all authors reviewed the survey questions, and we gathered feedback from outside researchers within our organization to improve the design of the planned questions.

Data Analysis Process: We analyzed the data quantitatively and qualitatively, reporting the results in Section 6.4.

To quantify users' perceptions of our tool, we tallied the responses to the post-interview survey, shown in Figure 6.7. Regarding questions Q1-Q3, we report the mean and distribution of Likert scores, and regarding Q4-Q7, we report the proportion of "Yes" responses. We also categorized each alert or fix that the participants examined during the interviews into "Useful" or one of 8 problem categories, based on the participant's explanation. We discuss the results in Section 6.4.1.

We conducted a *grounded-theory analysis* to analyze the study participants' rich free-form feedback [13], following the literature [27, 15, 26]. Grounded-theory analysis is a method used to analyze data by identifying recurring concepts, grouping these concepts into salient categories, and developing themes that provide an overall understanding of participants' perceptions of the tool. These concepts are derived from participants' quotations, reflecting their thoughts while using the tool and their responses to survey questions. All the resulting concepts and groups are referred to as a *codebook*.

We analyzed over 11 hours of usage and survey transcripts and identified a total of 161 codes in 12 distinct groups. Relevant codes are shown in Figure 6.8 and Figure 6.9. To create the initial codebook, the first and second authors independently analyzed two randomly selected interviews and generated lists of recurring concepts. They then met to create a unified list of concepts, create higher-level groups, and develop overall themes. Each author independently analyzed half of the remaining interviews, periodically syncing and jointly analyzing the same interviews to update the codebook and compare notes. Both raters agreed on all the classifications for alert responses. This was an iterative process [13], where we created the initial codebook after conducting the first 6 interviews and refactored/added groupings periodically as we conducted the remaining 11 interviews. We present our qualitative analysis in Section 6.4.2.

During the interviews, study participants suggested several features they felt would be useful, which provide useful recommendations for tool builders and directions for further research; we identify these as concepts in our grounded-theory analysis and discuss these feature requests in Section 6.4.3. The anonymized demographic data, interview and survey script, and codebook are in our data package [2].

6.4 User Study results

6.4.1 RQ1: Is DeepVulGuard useful in practice?

Detection: Figure 6.7 reports the results of our post-interview survey. On average, participants rated DeepVulGuard's alerts at 2.5 out of 5 for usefulness (Q1), with 2 participants giving it a rating of 4.5 or above and 3 participants giving it a rating of 1. Only 53% of participants felt that they trusted the tool's warnings about vulnerability alerts (Q5). **The biggest barrier to usefulness and trust in the tool's alerts was the amount of false positives**, with 30% of users explicitly reporting losing trust in the tool after frequently encountering false positives. The false positive rate in real-world settings was higher than in our SVEN dataset measurements (Section 6.2.3). We attribute this difference to varying languages and vulnerability types: SVEN contains Python code, whereas most participants worked on Typescript or C# which comprise

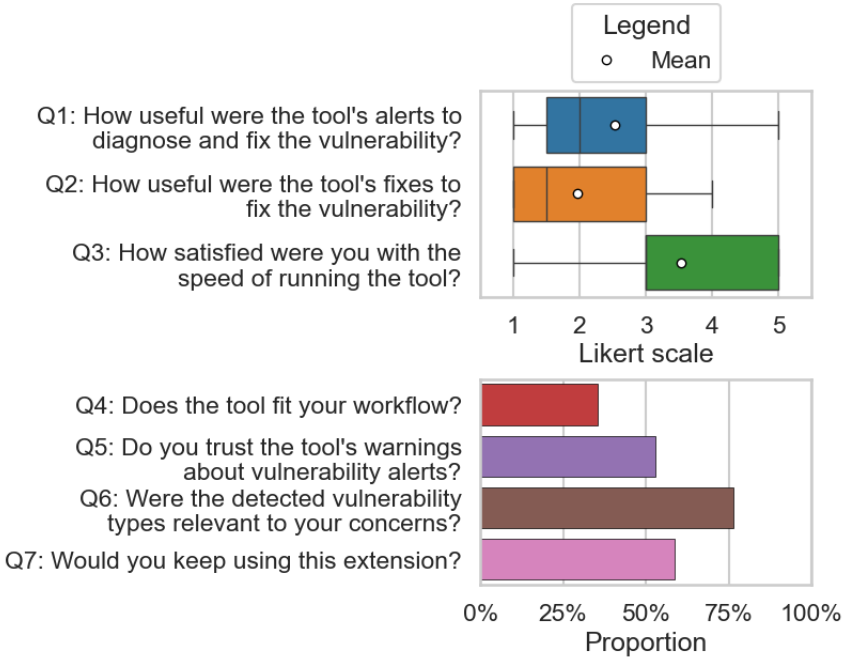


Figure 6.7: Summary of participants' overall perceptions of DeepVulGuard, from our post-interview survey.

only 6% of our training data. Additionally, SVEN examples are intra-procedural, lacking information about the calling context and runtime environment, which may widen the gap between benchmark data and real-world testing.

76% of participants felt that the vulnerability types detected by the tool were relevant (Q6), with some participants expressing strong approval, for example: *"Definitely all of the all the categories of the vulnerabilities that were found here were good. They're all ones that hit these kinds of code all the time."*.

Fix suggestions: On average, participants rated DeepVulGuard's fix suggestions at 2 out of 5 for usefulness (Q1). 2 participants gave it a rating of 4, and 7 participants gave it a rating of 1. **For fixes, one of the most common issues was that the fix was not customized to the developer's codebase**, for example, creating a function to sanitize user inputs when the developer wants to reuse their existing sanitization library; this often prevented the users from directly applying the fix, requiring an overhaul to produce a fix with their intended approach.

This highlights a limitation of common exact match or execution-based metrics for evaluating AI-based fixes, as these metrics do not capture the practical nuances of generating fixes for real-world codebases.

Speed: The average response time for the tool was 3.9 seconds per file. More than half of the participants were "very satisfied" with the speed of the tool, rating it at 5/5 (Q3). When asked about the tool's speed, one participant stated *"Totally satisfied. I can wait for this kind of stuff"* (referring to security alerts + fixes).

Workflow integration: More than half (65%) of participants felt that the tool in its current state would not fit into their workflow (Q4). 14 out of 17 users expressed that **the tool would be more useful if it was running in the background and scanning their code while they were editing or ran along with their build or commit commands**; manually triggering the scan was a barrier to usage, since it required a stopping point in development.

Summary: Although not fully satisfactory, the tool shows promise — **59% of participants expressed that they would keep using the extension.**

Figure 6.8 reports the participants' responses to the 51 alerts and 24 fixes for which they provided direct feedback during the interviews, based on the categories assigned in our grounded-theory analysis. Here we display alerts from the combined CodeBERT + LLM model, excluding four participants who used the CodeBERT model. Participants considered 18% of alerts, and 25% of fixes, to be useful and without significant problems. The primary causes of false positive alerts were missing context (totaling 51% of alerts) and incorrect pattern recognition (31%). Missing context involved misidentifying variables as user-controlled or overlooking vulnerabilities handled by the calling context or runtime environment. Incorrect pattern recognition involved misidentifying harmless patterns as vulnerabilities, such as constant strings mistaken for hard-coded credentials. We hypothesize that incorporating references to the calling context and runtime environment [31], along with in-context examples [53] of commonly misidentified patterns, into the LLM filter prompt shown in Figure 6.3 may help to address these limitations.

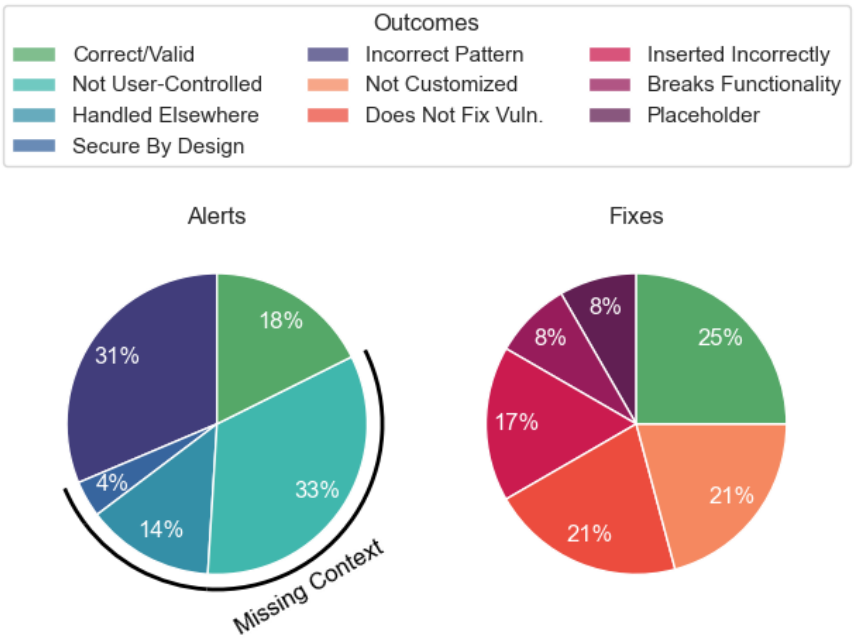


Figure 6.8: Participant responses to LLM-filtered alerts and LLM-generated fixes while using DeepVulGuard.

21% of fixes were rejected because they were not customized to the user's codebase – they did not incorporate existing functions in the project (e.g. sanitization) or didn't comply with project style and linting rules, and thus could not be directly applied. Additionally, 21% did not address the underlying vulnerability, another 17% incorrectly inserted code generated by the LLM, resulting in syntax or indenting issues, and 8% of fixes mitigated the issue but broke existing functionality. 8% were placeholders containing instructional comments rather than functional fixes.

6.4.2 RQ2: Which aspects of vulnerability detection + fix tools are most useful?

Figure 6.9 summarizes the participants' comments about different components of detection and fix tool components, based on the participants' think-aloud feedback while using DeepVulGuard, which we categorized in our grounded-theory analysis. Based on their responses,

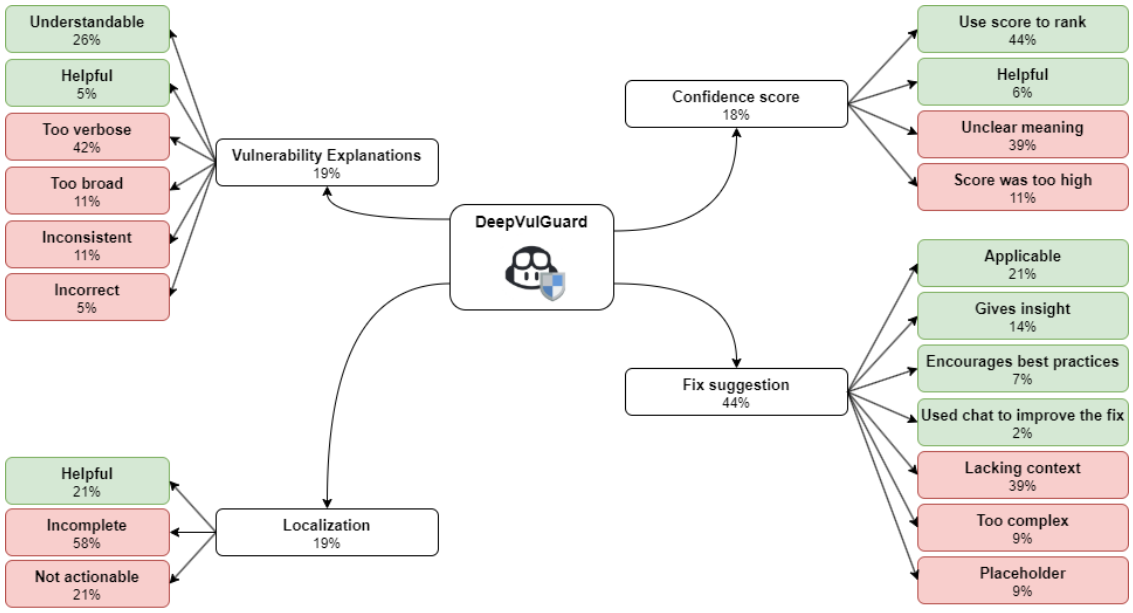


Figure 6.9: Participants' in-use feedback on the aspects of DeepVulGuard. The aspects of the tool are organized into trees, where the leaf nodes are categories of comments from participants and the intermediate nodes are groupings of each category. Percentages show what portion of the comments on their respective aspects that each category constitutes; positive comments are green and negative comments are red.

fix suggestions and confidence scores seem to be the most useful aspects of the tool, with 44% and 50% positive feedback respectively.

Fix suggestions: Fixes that have the correct initial approach allow the user to apply them with minimal changes. Out of the comments on fixes, 21% noted that the fixes could be applicable, with 7% of these noting that the fixes required minor changes such as changing variable names or error messages.

Fixes gave developers insight on vulnerabilities and best security practices. Beyond mitigating the vulnerability, 14% of comments noted that seeing the diff between 'bad' and 'good' code helped them understand the root cause of the issue. Fixes can also provide guidance on secure best practices when developers are working on unfamiliar code (7%). One developer said, *"If I knew I was starting in an area I wasn't very familiar on the most secure practices, it would be very helpful."*

However, some fixes lacked context (39%) or were more complex than the user's ideal solution (9%). Many security issues require multi-site edits, either when changing the semantics of a shared function or when encoding or decoding data; our tool is currently limited to fixing one function at a time, so the fix can lack context, which can result in breaking functionality, expressed by one participant as follows: *"This is how [the fix] should be defined, but then I would have to go find all the places where it's used and fix them all. And that might be a lot of places."* Incorporating additional context, such as the list of functions which call the function to be changed, could help provide more, contextualized fixes.

Placeholder fix suggestions were less preferred for users who were expecting functional fixes. LLMs occasionally generate placeholder code by default, including containing instructional comments rather than functional fixes. Some users did not find these useful, constituting 9% of comments on fixes, such as *"Well, that's not helpful"* and *"It's not really adding anything to the output"*. Since placeholder fixes can negatively impact users who prefer functional fixes, it's important to set expectations; a potential improvement could be to re-generate the fix when placeholders are initially generated, and if a functional fix cannot be provided, the placeholder fix should be accompanied by an explanatory message.

Chat interactions added value by allowing developers to iterate on fixes. In one case where the fix used the wrong approach at first, the developer iterated on the fix by first suggesting a different approach and then specifying their style guidelines, and arrived at a fix which they would apply without having to write the code themselves, saying afterwards, *"I like that this is conversational and I could do a few more rounds of interaction to understand what could be alternative solutions or better ways to approach this issue besides the initial suggestion"*. Before the chat feature was implemented, 50% of participants expressed the desire to ask the chatbot for more information about alerts or to suggest modifications to fixes. Recent research supports the potential usefulness of chat interactions. Nam et al. [35] found that developers completed more coding tasks within a given time when using a chatbot for code explanations compared to using a search engine.

Confidence score: **Users overwhelmingly used the confidence score to rank issues by importance.** This interaction constituted 44% of user feedback on this feature; an additional 6% noted that the confidence score was helpful for understanding the model's prediction. The confidence score can be useful to rank issues, but should be displayed to the user with full understanding of its meaning; 39% of feedback indicated that participants were unclear about the meaning of the score. We hypothesize that integrating the severity score [39] with the model confidence score will make it more useful for prioritizing vulnerabilities. Finally, a high-confidence result which is a false-positive can degrade the trust in the tool, as seen in 11% of feedback; therefore, expectations should be managed.

Vulnerability Explanations: Explaining the vulnerability and security best practices can be useful for providing comprehensive understanding of a vulnerability; 35% of feedback was positive, indicating that the explanations were understandable and helpful. One developer compared with existing static analysis tools: *"Normally with a static analysis tool, if I get an error that I'm a little unsure on, I would have to go out to a website of track down [an explanation], so providing me a diff and some text here explained it a bit."*

With explanations, brevity and adding visual annotations are important. 42% of feedback mentioned that the alert descriptions were too verbose; 11% mentioned that the verbiage was too broad to be useful. To quote one developer, *"If I see the code, it says sanitized input then OK, so I need to sanitize it... I would be more comfortable looking at the fix to know what the issue it is detecting, than read the verbose text."* Later they stated, *"I usually read one or two lines and then I stopped there. If it is too verbose, I probably don't pay too much attention."*. Another developer noted, *"Rather than giving me a wall of text, it would be great if it gave me bad and good examples."* One suggestion from this participant is to visually annotate the explanation by presenting labels with short, recognizable names, such as *Path Injection*, and allow the user to read the full explanation if they are interested.

Users expect the tool's outputs to be consistent, which introduces challenges when integrating LLM explanations and chat. 11% of feedback on explanations noted

inconsistencies between the explanation and subsequent fixes. For example, one user noted that an alert's explanation specified not to use an insecure hashing function `btoa`, while the suggested fix used this function. While this specific issue could be solved by simply including the LLM-generated explanation in the fix prompt, the general issue is important for tool builders to be aware of.

Localization: **Users preferred highlights on a complete line or variable/string/function call.** DeepVulGuard highlights the tokens which were localized by the model, which may not necessarily align to semantic boundaries. 21% of participant feedback noted that localizations was helpful, especially the ability to zoom to a vulnerability' location. However, 58% of feedback noted that the localization seemed incomplete because it only highlighted part of the structure it was referencing. This behavior is by-design since the underlying model was specifically trained to only flag parts of the code that contributed to the vulnerability. However, this was one reason that users lost trust in our tool; to quote one user, *"The fact that the squiggle starts part way through a word made me wonder – Oh, is it just on the wrong line, or maybe got some wires crossed somewhere?"*. Our model detects vulnerable code patterns at the fault location. **In 21% of feedback, users noted that they would prefer to see an alert in a more actionable location – the root cause, or source for input validation issues, rather than at the fault location**; one user stated, *"Preferably, I'd actually do that check way before this, either where we download or where we extract, and that way we know that when we get here, this path is already sanitized."*.

6.4.3 RQ3: What features do developers want from vulnerability detection + fix tools?

Figure 6.10 displays a word cloud of feature suggestions from the study, identified through our grounded-theory analysis. We implemented basic versions of some highly requested features from early interviews, specifically alert suppression and basic chat interaction. We measured the frequency of these requests from the participants who lacked these features.

Figure 6.10: The relative frequency of features suggested by the study participants.

Chat interaction was a frequently requested feature. Before implementing the chat feature, 3 out of 6 participants wanted a chat interface to better understand alerts, view examples of inputs that triggered them, and refine suggested fixes. Users who had chat available found it useful; for example, one user asked for alternative suggestions to address an alert and expressed, *"Yeah, these are good ideas. This would seed some ideas for me"*.

Starting the scan manually was a major disruption to developers' workflow. 12 out of 17 participants expressed that they would prefer if the tool scanned their code in the background and reported problems while they were editing, and 3 participants said it would be useful to trigger scans through a build or commit hook. One developer explained as follows: *"From a workflow perspective, I don't typically reach a point where I say, 'Oh, I'm just gonna stop now and go check for security issues'."*

Before we implemented the suppression feature, 5 out of 6 participants wanted the ability to suppress irrelevant alerts. One user specifically mentioned that the tool's outputs would be more manageable if they could filter out alerts identified as false positives or those frequently incorrect or irrelevant to their use case. Participants suggested several other enhancements, such as options for long-running overnight scans, click-to-scan UI interactions, the ability to scan entire directories or projects and their dependencies, and team-shared configurations sensitivity and rulesets.

6.5 Discussions

Based on our study, we find that current SOTA AI vulnerability detection and fix tools are not yet satisfactory, but developers remain eager to continue using them. This motivates us to continue researching and improving deep learning based methods and tools of vulnerability detection and fix. We have highlighted key lessons from our study in bold in Section 6.4. Here, we expand on several aspects of evaluating and deploying AI detection and fix tools.

Our detection results show a substantially higher false positive rate in real-world deployment compared to benchmark tests. Our study indicates it is impractical to (re)train a model for every new code base due to the lack of labeled data. While current test data for AI models often come from the same projects as the training data, in real-world scenarios, AI models are typically applied to unseen projects. Additionally, current deep learning models handle one function at a time [44], including DeepVulGuard; however, we see that many vulnerabilities in real-world code are related to multiple functions and the runtime environment. Lacking such program and environment context led to 51% of our tool's alerts being identified as false positives, as shown in Figure 6.8. This result highlights the need for vulnerability benchmarks that incorporate more realistic contextual information.

We also see that developers usually have specific definitions of "false positive" tailored to their own codebase and deployment scenario. For instance, one user dismissed a warning about sensitive data in an SSH key file, citing feasibility issues despite acknowledging the vulnerability: *"This is a problem, but I don't think there's anything they can do about it... I mean, it is a vulnerability – but if somebody can obtain access to the file system, then they have access to all kinds of password files."* These user-specific assumptions are difficult to incorporate into dataset labels, emphasizing the need for holistic evaluations in realistic development scenarios. To improve AI to predict likely feasible bugs, we may need to construct datasets using bugs with reproducible exploits rather than potential (but possibly infeasible) vulnerabilities from CVEs.

We found that 21% of suggested fixes, though functionally-correct (i.e. they would pass unit tests), were rejected because they were not customized to the user's code-base. Current test execution-based benchmarks [9, 28] do not capture this critical issue, highlighting the need for more realistic evaluations that consider this aspect of fix suggestions.

Based on our study, we make several recommendations for deploying AI detection and fix tools. First, when we deployed multiple models for detection, explanations, and fix, we need to ensure that the outputs of these models are consistent, so we do not confuse users. Second, LLM-generated explanations were often too verbose, suggesting a need to guide the LLM to generate concise output and code examples, and to add visual annotations. Third, users prioritize issues based on the displayed score, and this score should reflect important aspects such as severity, not just the confidence score.

6.6 Threats to Validity

Since our work is an empirical study, there may be limits to the generalizability of its findings [25].

External and internal validity: Our sample of 17 developers from a large company may not fully represent all software developers' opinions. Research estimates that 16 users are typically sufficient to fully understand the challenges users face when using a tool [37]. This aligned with our findings, as the final two rounds of five interviews each added only 5 and 2 new codes respectively, indicating saturation. We recruited 11 more developers than a similar user study [21]. We included developers with experience ranging from 3 to over 30 years, covering projects in four programming languages and including backend, frontend, and IDE extension code.

Following the literature's recommendation to test iteratively with small user groups [36], we first tested with three participants, added key features to our tool like *directory scan*, *click-to-scan*, and *LLM filter*, then tested with three more users before adding *chat functionality* and *alert suppression*, and finally included the remaining eleven participants. We account for the developing feature set in Figure 6.10 and report only the LLM filter model responses in Figure 6.8.

We studied one set of models and 27 vulnerability types, which may limit generalization to other models and vulnerability types. The focus of our study was on understanding the practical usefulness of DL models in the IDE, so we chose to study one set of SOTA models (validated in Section 6.2.3). Our tool supports the top 25 CWEs [49], plus the most frequent vulnerability types CodeQL detected in our dataset. Future work could study more models and vulnerability types.
Construct validity: We used think-aloud interviews, discussed in Section 6.4.2, which may result in users providing personal preferences rather than real system issues [40]. We addressed this by using grounded-theory analysis to assess the users' objective verdicts on root causes of alerts and fixes (Figure 6.8) and quantify the support for each feedback category (Figure 6.9).

6.7 Related Work

Deep learning for vulnerability detection and fixing: Recent research has explored various DL methods for vulnerability detection, including graph neural networks (GNNs) [11, 32, 46] and transformer models [54, 20, 12, 21]. GNNs typically require complete source code to generate the necessary abstract syntax trees (ASTs) and control flow graphs (CFGs), which limits their effectiveness on incomplete code snippets. Our model is based on the state-of-the-art approach from Chan et al. [12], which optimizes for both in-IDE latency and incomplete code snippets. Compared to Fu et al. [21], which uses three separate models for localization, type, and severity prediction, we fine-tune our model to predict the presence, location, and type of vulnerabilities in a single forward pass, enhancing both simplicity and efficiency. Additionally, we introduce a novel LLM-based filtering technique that improved our model's precision by 20%; this is compatible with Fu et al. [21]'s approach. We also integrate SOTA LLMs for fix suggestions and show that they perform on par with existing DL and APR tools in Section 6.2.3.
Benchmark studies of AI detection + fix models: Several empirical studies of DL models corroborate our results in underscoring the need for user studies in realistic scenarios. Chakraborthy et al. [11] found that DL models often face issues with data duplication and unrealistic distributions of vulnerable classes. Chen et al. [14] showed that existing models have

difficulty generalizing to unseen projects, but increasing the volume of training data can improve their generalization. Steenhoek et al. [47] demonstrated that while some models perform well on benchmarks matching their training data, they may struggle to generalize to new projects and bug types. Recently, Ding et al. [16] indicated that current benchmarks may overestimate the performance of deep learning models.

User studies of static analysis and AI tools: We developed our tool based on findings and recommendations from several user studies of traditional static analysis tools [27, 15, 45] (see Section 6.2.1 for more details). A controlled study by Fu et al. [21] involving six software practitioners demonstrated that DL detection & fix tools can be beneficial. They also surveyed 21 practitioners about the usefulness of features like localization, type and severity prediction, and fix suggestions, which informed our tool's design. To our knowledge, we are the first to conduct a study with professional developers on projects they own in a real-world deployment setting. There has been work on investigating whether APR tools are useful in practice. Surveys showed that most software practitioners prefer manual bug fixes over current APR tools due to unreliable and slow patch production [33, 51, 38]. Campos et al. [10] deployed APR in the IDE in a controlled study with 16 developers on a given project; APR increased developers' speed but may impact maintainability. We studied deep learning tools in a real-world development scenario where professional developers run the tool on their own projects. The tool is fast and can improve fixes via conversations with developers.

6.8 Conclusions

Recent research has introduced various deep learning vulnerability tools with promising benchmark performance. However, there has been no extensive user study on their real-world utility. To address this, we conducted a comprehensive user study with 17 professional developers, analyzing 24 projects, 6.9k files, and over 1.7 million lines of code, generating 170 alerts and 50 fix suggestions. Our study revealed that while current models show promise, they are not yet practical for everyday use due to challenges with (1) false positives caused by missing code

context and incorrect pattern recognition, and (2) fixes which were not customized to the codebase. Based on user feedback, we make several recommendations for aligning model evaluations with real-world development scenarios, and for deploying models in practice.

Through our user study, we identified several areas for further research. One direction is to support automatic code scanning and address questions such as when to scan and how much code context to include. Another direction is to further develop AI powered chat interaction. Our preliminary chatbot implementation showed useful for explaining vulnerabilities and generating fixes. Future research should also resolve consistency issues among AI models' outputs.

6.9 Bibliography

[1] Github copilot. https://github.com/features/copilot.

[2] The data package for our study. https://doi.org/10.6084/m9.figshare.26367139, 2024.

[3] List of data breaches - Wikipedia. https://en.wikipedia.org/wiki/List_of_data_breaches, 2024.

[4] Baziuk, W. BNR/NORTEL: path to improve product quality, reliability and customer satisfaction. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95* (1995), pp. 256–262.

[5] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM* (2010).

[6] Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., and Gazit, I. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* (2023).

[7] Boehm, B. W. *Software Engineering Economics*. Springer Berlin Heidelberg, 2002.

[8] Bui, Q.-C., Paramitha, R., Vu, D.-L., Massacci, F., and Scandariato, R. APR4Vul: an empirical study of automatic program repair techniques on real-world java vulnerabilities. *Empirical Software Engineering*.

[9] Bui, Q.-C., Scandariato, R., and Ferreyra, N. E. D. Vul4J: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories* (New York, NY, USA, 2022), MSR '22, Association for Computing Machinery, p. 464–468.

[10] Campos, D., Restivo, A., Sereno Ferreira, H., and Ramos, A. Automatic program repair as semantic suggestions: An empirical study. In *14th IEEE Conference on Software Testing, Verification and Validation* (2021), ICST '21, pp. 217–228.

[11] Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering 48*, 09 (Sept. 2022), 3280–3296.

[12] Chan, A., Kharkar, A., Moghaddam, R. Z., Mohylevskyy, Y., Helyar, A., Kamal, E., Elkamhawy, M., and Sundaresan, N. Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning?, 2023. arXiv: 2306.01754.

[13] Charmaz, K. *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006.

[14] Chen, Y., Ding, Z., Alowain, L., Chen, X., and Wagner, D. 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* (New York, NY, USA, 2023), RAID '23, Association for Computing Machinery, p. 654–668.

[15] Christakis, M., and Bird, C. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE '16, Association for Computing Machinery, p. 332–343.

[16] Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., David Wagner, B. R., and Chen, Y. Vulnerability detection with code language models: How far are we? In *Proceedings of the 47th International Conference on Software Engineering* (2025), ICSE '25.

[17] Distefano, D., Fähndrich, M., Logozzo, F., and O'Hearn, P. W. Scaling static analyses at facebook. *Communications of the ACM* (2019).

[18] Easterbrook, S., Singer, J., Storey, M.-A., and Damian, D. *Selecting Empirical Methods for Software Engineering Research*. Springer London, 2008.

[19] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (Online, Nov. 2020), T. Cohn, Y. He, and Y. Liu, Eds., Association for Computational Linguistics, pp. 1536–1547.

[20] Fu, M., and Tantithamthavorn, C. LineVul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories* (New York, NY, USA, 2022), MSR '22, Association for Computing Machinery, p. 608–620.

[21] Fu, M., Tantithamthavorn, C., Le, T., Kume, Y., Nguyen, V., Phung, D., and Grundy, J. AIBugHunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering 29*, 1 (Nov. 2023).

[22] He, J., and Vechev, M. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2023), CCS '23, Association for Computing Machinery, p. 1865–1879.

[23] Humphrey, W. S. *A Discipline for Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[24] IBM. Cost of a data breach 2024. https://www.ibm.com/reports/data-breach, 2024.

[25] Jedlitschka, A., Ciolkowski, M., and Pfahl, D. *Reporting Experiments in Software Engineering*. Springer London, 2008.

[26] Johnson, B., Bird, C., Ford, D., Forsgren, N., and Zimmermann, T. Make your tools sparkle with trust: The picse framework for trust in software tools. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice* (2023), ICSE-SEIP '23, IEEE Press, p. 409–419.

[27] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering* (2013), ICSE '13, pp. 672–681.

[28] Just, R., Jalali, D., and Ernst, M. D. 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* (New York, NY, USA, 2014), ISSTA '14, Association for Computing Machinery, p. 437–440.

[29] Keman, H., Wang, X., Wei, W., and Madnick, S. The Devastating Business Impacts of a Cyber Breach. https://hbr.org/2023/05/the-devastating-business-impacts-of-a-cyber-breach, 2023.

[30] Lewis, C. *Using the "thinking-aloud" method in cognitive interface design.* IBM TJ Watson Research Center Yorktown Heights, NY, 1982.

[31] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2020), NIPS '20, Curran Associates Inc.

[32] Li, Y., Wang, S., and Nguyen, T. N. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2021), ESEC/FSE 2021, Association for Computing Machinery, pp. 292–303.

[33] Meem, F. N., Smith, J., and Johnson, B. Exploring experiences with automated program repair in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[34] Microsoft. Visual Studio Code - Code Editing. Redefined. https://code.visualstudio.com/, 2024.

[35] Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., and Myers, B. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[36] Nielsen, J. Why You Only Need to Test with 5 Users. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/, 2000.

[37] Nielsen, J., and Landauer, T. K. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (New York, NY, USA, 1993), CHI '93, Association for Computing Machinery, p. 206–213.

[38] Noller, Y., Shariffdeen, R., Gao, X., and Roychoudhury, A. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA, 2022), ICSE '22, Association for Computing Machinery, p. 2228–2240.

[39] NVD. NVD - Vulnerability Metrics. https://nvd.nist.gov/vuln-metrics/cvss.

[40] O'Brien, L., and Wilson, S. Talking about thinking aloud: Perspectives from interactive think-aloud practitioners. *Journal of User Experience* (2023).

[41] OpenAI. GPT-4 technical report, 2024. arXiv: 2303.08774.

[42] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at google. *Communications of the ACM (CACM) 61 Issue 4* (2018), 58–66.

[43] Seaman, C. B. *Qualitative Methods*. Springer London, 2008.

[44] Sejfia, A., Das, S., Shafiq, S., and Medvidović, N. Toward improved deep learning-based vulnerability detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[45] Smith, J., Do, L. N. Q., and Murphy-Hill, E. Why can't Johnny fix vulnerabilities: a usability evaluation of static analysis tools for security. In *Proceedings of the Sixteenth USENIX Conference on Usable Privacy and Security* (USA, 2020), SOUPS'20, USENIX Association.

[46] Steenhoek, B., Gao, H., and Le, W. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[47] Steenhoek, B., Rahman, M. M., Jiles, R., and Le, W. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 2237–2248.

[48] The MITRE Corporation. CWE - Common Weakness Enumeration database. https://cwe.mitre.org/.

[49] The MITRE Corporation. CWE - CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/.

[50] Wang, R., Cheng, R., Ford, D., and Zimmermann, T. Investigating and Designing for Trust in AI-powered Code Generation Tools, May 2023. arXiv:2305.11248.

[51] Winter, E., Bowes, D., Counsell, S., Hall, T., Haraldsson, S., Nowack, V., and Woodward, J. How do developers really feel about bug fixing? directions for automatic program repair. *IEEE Transactions on Software Engineering* (2023).

[52] Wu, Y., Jiang, N., Pham, H. V., Lutellier, T., Davis, J., Tan, L., Babkin, P., and Shah, S. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2023), ISSTA '23, Association for Computing Machinery, p. 1282–1294.

[53] Xie, S. M., and Min, S. How does in-context learning work? A framework for understanding the differences from traditional supervised learning. https://ai.stanford.edu/blog/understanding-incontext/, 2022.

[54] Yang, A. Z. H., Le Goues, C., Martins, R., and Hellendoorn, V. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

6.A Appendix: Detection model

This appendix lists the details of our detection model and training procedure.

6.A.1 Training procedure

We implemented our detection model by fine-tuning CodeBERT [19], following the procedure and using the dataset introduced in Chan et al. [12]. We chose this model because it is suited to Edit-Time vulnerability detection, localization, and bug-type prediction, while other state-of-the-art models we considered [20, 46] are trained on and primarily intended for complete code snippets without the ability to both localize lines and predict a specific bug type. We fine-tuned the CodeBERT model for *multi-task prediction*. For each given code snippet, it predicts: (1) whether the snippet contains a vulnerability, (2) whether any specific tokens in the snippet contain a vulnerability, and (3) the type of the vulnerability. See Figure 2 in the paper for an overview of our detection model approach. We replaced the final classifier layer (normally performing binary classification) with a layer of size $(d + k)$, where d is the number of tokens in the context window (512 for CodeBERT) and k is the number of bug types in our training dataset (27 in our case), followed by a softmax layer. During training, we labeled the tokens indicated in the CodeQL alert with 1 when the code contained a vulnerability related to that token and 0 otherwise, did likewise for the bug types, and trained the model to jointly predict these $512 + 27 = 539$ classes for each example. To train the model, we used a dataset of over 1.3 million alerts from CodeQL, collected using the same methodology as Chan et al. [12].

We trained the CodeBERT model on a dataset containing over 1 million code snippets in seven languages (C/C++, C#, Go, Java, JavaScript/TypeScript, Python, and Ruby), including bug types corresponding to 27 CodeQL rules marked as impactful in the CWE database [48], including Path Injection, SQL injection, Cross-Site Scripting (XSS), URL Redirection, Hard-coded Credentials, and Plain-Text Logging of Sensitive Data. We found that this multi-task fine-tuning was effective for providing localization and bug type classification, as shown in Section II-C in the paper.

6.A.2 Dataset statistics

Table 6.1: Proportion of alerts in each language.

Language	Percentage
js	40.1%
py	30.3%
java	9.2%
c	5.4%
ts	5.1%
go	4.8%
cpp	1.6%
html	1.1%
cs	0.8%
tsx	0.5%
cc	0.4%
jsx	0.3%
rb	0.1%
mjs	0.1%
h	0.1%
hpp	0.0%
es6	0.0%
xhtml	0.0%

Type	Percentage
path-injection	18.1%
sql-injection	16.8%
incomplete-sanitization	13.1%
unvalidated-url-redirection	7.7%
reflected-xss	7.6%
stack-trace-exposure	7.2%
clear-text-logging	6.3%
hardcoded-credentials	6.0%
weak-cryptographic-algorithm	3.2%
insecure-randomness	2.1%
overly-permissive-file	1.9%
command-line-injection	1.5%
clear-text-storage-sensitive-data	1.4%
incomplete-hostname-regexp	1.3%
ssrf	1.3%
flask-debug	1.3%
regex-injection	0.9%
insufficient-password-hash	0.9%
bind-socket-all-network-interfaces	0.6%
code-injection	0.6%
tarslip	0.1%
weak-crypto-key	0.1%
cleartext-storage-file	0.1%

6.A.3 Full list of languages in the training dataset, by file extension

- .js
- .py
- .h
- .rb
- .ts
- .cs
- .hpp
- .jsx
- .cpp
- .java
- .tsx
- .c
- .go

6.A.4 Hyperparameters

- n_epochs: 200
- batch_size: 24
- learning_rate: 2e-5

CHAPTER 7. GENERAL CONCLUSION

Now that deep learning models have demonstrated promising capabilities for vulnerability detection, it has become essentially important to understand these models (especially their limitations) and develop techniques to apply them to real-world code. This dissertation provides a deep and comprehensive study of the state-of-the-art models, including diverse evaluation scenarios and deployment in a real-world user study, and shows our successful efforts to improve these models by integrating static and dynamic analysis.

7.1 Summary of Contributions

First, we performed comprehensive empirical studies of a wide range of deep learning models. In Chapter 2, we evaluated fine-tuned transformers and graph neural networks, which were the state-of-the-art at the time. Through this, we showed that these models have high variability in their predictions; that training on a single type of vulnerability generally performed better than multi-type training; that performance did not increase significantly by simply increasing the dataset size; and that models tended to focus on textual code patterns apart from the root cause of vulnerabilities. In Chapter 5, as large language models (LLMs) began to show new capabilities, we evaluated LLMs using state-of-the-art prompting techniques. We found that LLMs could not perform substantially better than random guessing and could not differentiate buggy and fixed versions of code; that they often made errors in understanding code, hallucinating, and logically reasoning to a conclusion; and that they performed far worse than humans in localizing bugs. Our findings from these empirical studies continue to apply to new approaches for applying deep learning models, as many of the problems we found are yet unsolved.

Based on the results of our empirical studies, we saw that models trained for textual pattern matching were limited in their understanding of vulnerability semantics and failed to generalize to

unseen data. We developed two approaches to integrate program analysis into these models: static analysis with DeepDFA (Chapter 3) and dynamic analysis with TRACED (Chapter 4). We found that integrating static and dynamic analysis allowed these models to improve on the state-of-the-art performance while improving other areas of efficiency and generalization. DeepDFA yielded better generalization, including generalization to unseen datasets, and greater efficiency in terms of training data and runtime resources, than text-based models. TRACED demonstrated the ability to predict function execution better than statically pre-trained models, and yielded greater performance on two downstream tasks: code clone detection and vulnerability detection. Both of these works represent an initial step towards integrating program analysis with deep learning, but more open questions remain about how to broadly and effectively utilize these new techniques.

Finally, in Chapter 6, we ran a user study on an IDE-integrated instantiation of state-of-the-art deep learning models, reporting several findings which were unseen because they were difficult to measure in offline benchmark evaluations. We found that, when deployed in practice, the models tend to generate a high rate of false positives and produce fixes which were not immediately applicable to the codebase. We also surfaced several feature requests which were highly desired by users, such as chat integration and automatic scanning. Our findings motivate investigation into how to mitigate the pain points expressed by developers and longitudinal studies of how these tools are best used by developers.

7.2 Future Work

As future work from Chapter 3, we envision that DeepDFA can be extended to integrate more dataflow analyses [2] and abstract interpretation [1] techniques. As our work showed marked improvement even when tightly scoped to reaching definitions analysis with the Kildall method, we believe that it only scratches the surface of possibilities of this technique.

Building on Chapter 4, we would like to scale up the integration of execution information. TRACED was pre-trained on contest programs from Project CodeNet [3], but the next step will

be to pre-train on a larger dataset of execution traces from realistic programs, which contain much more complexity and diversity than contest programs. We believe that this would extend its applications to be more effective for real-world programs.

Chapter 5 showed that LLMs often experienced Code Understanding errors. We plan to integrate static analysis to extract a reliable set of facts about the code under analysis, and include these facts in the prompts given to the model; given that Logic errors also appeared frequently, we also want to explore ways to make LLM reasoning more faithful, such as self-consistency [4].

Finally, following up on Chapter 6, we plan to continue studying the deployment of deep learning models in the IDE scenario, building on the DeepVulGuard prototype and initial user study. User feedback motivated the addition of several specific features, such as automatic scanning, in-depth chat interaction, and false-positive mitigation.

7.3 Bibliography

[1] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, Jan 1977), POPL '77, Association for Computing Machinery, pp. 238–252.

[2] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. Springer, 2015.

[3] Puri, R., Kung, D., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J. T., Chen, J., Choudhury, M., Decker, L., Thost, V., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., and Reiss, F. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (2021), J. Vanschoren and S. Yeung, Eds., vol. 1.

[4] Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations* (2023).