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 vii

LIST OF TABLES vii

LIST OF FIGURES ix

LIST OF FIGURES ix

ACKNOWLEDGMENTS xii

ABSTRACT xiv

CHAPTER 1. GENERAL INTRODUCTION 1
- 1.1 Contributions 2
- 1.2 Outline 3
- 1.3 Bibliography 4

CHAPTER 2. AN EMPIRICAL STUDY OF DEEP LEARNING MODELS FOR VULNERABILITY DETECTION 6
- 2.1 Introduction 7
- 2.2 A Survey of Models and their Reproduction 9
- 2.3 Research Questions and Findings 12
  - 2.3.1 Capabilities of Deep Learning Models 12
  - 2.3.2 The Training Data 21
  - 2.3.3 Internals of Deep Learning 25
- 2.4 Threats to Validity 30
- 2.5 Related Work 31
- 2.6 Conclusions and Future Work 32
- 2.7 Acknowledgements 32
- 2.8 Bibliography 33

CHAPTER 3. DEEPDFA: DATAFLOW ANALYSIS-INSPIRED DEEP LEARNING FOR EFFICIENT VULNERABILITY DETECTION 41
- 3.1 Introduction 42
- 3.2 Overview 44
- 3.3 Rationale 46
  - 3.3.1 Dataflow Analysis for Vulnerability Detection 46
  - 3.3.2 Analogy of Graph Learning and Dataflow Analysis 47

3.3.3 The Novelty of Our Work 49
3.4 Approach 49
3.4.1 Abstract Dataflow Embedding 50
3.4.2 Using Graph Learning to Propagate Dataflow Information 52
3.5 Evaluation 55
3.5.1 Implementation 55
3.5.2 Experimental setup 56
3.5.3 Effectiveness 61
3.5.4 Efficiency 62
3.5.5 Generalization 65
3.5.6 Ablation studies 67
3.6 Threats to Validity and Discussions 67
3.7 Related Work 69
3.8 Conclusions and Future Work 70
3.9 Acknowledgements 71
3.10 Bibliography 71
3.A Appendix: Baseline reproductions 80
3.B Appendix: Programs that are removed 80
3.C Appendix: Additional effectiveness results 81
3.D Appendix: Training times of all models 81
3.E Appendix: Model sizes 82
3.F Appendix: Additional cross-project evaluation results 83

CHAPTER 4. TRACED: EXECUTION-AWARE PRE-TRAINING FOR SOURCE CODE . 84
4.1 Introduction 85
4.2 Overview 89
4.3 Tracing & Feature Engineering 91
4.3.1 Representing Program States 91
4.3.2 Quantized Variable Values 93
4.3.3 Building Learnable Labels for Code Models 95
4.4 Model 96
4.4.1 Execution-aware Pre-training 96
4.4.2 Task-specific Fine-tuning 100
4.5 Experimental Setup 101
4.5.1 Trace Collection 101
4.5.2 Dataset 102
4.5.3 Model Configuration 104
4.6 Evaluation 105
4.6.1 RQ1. Effectiveness of TRACED in Static Estimation of Execution 105
4.6.2 RQ2. Effectiveness of TRACED's Pre-training Objectives 108
4.6.3 RQ3. Effectiveness of TRACED's Quantized Variable Values 109
4.6.4 RQ4. TRACED's Performance in Code Understanding Tasks 111
4.7 Related Work 112
4.8 Threats to Validity 113
4.9 Conclusion 114
4.10 Acknowledgments 114

4.11 Bibliography 115

CHAPTER 5. TO ERR IS MACHINE: VULNERABILITY DETECTION CHALLENGES LLM REASONING 124
5.1 Introduction 125
5.2 Can LLMs Effectively Detect Vulnerabilities? 129
5.3 Why do LLMs Fail to Reason About Vulnerabilities? 132
5.3.1 Does Model Size Matter? 135
5.3.2 Do Model Training Data & Methods Matter? 136
5.3.3 Does Additional Domain Knowledge Help? 138
5.4 Related Work 140
5.5 Conclusion 140
5.6 Bibliography 141
5.A Appendix: Vulnerability detection prompts 152
5.B Appendix: Models 153
5.C Appendix: Benchmarks for other domains 154
5.D Appendix: Simple CWE examples 155
5.E Appendix: Error analysis methodology 156
5.E.1 Inter-rater agreement 156
5.E.2 Error analysis UI 156
5.E.3 Error categories 158

CHAPTER 6. CLOSING THE GAP: A USER STUDY ON THE REAL-WORLD USEFULNESS OF AI-POWERED VULNERABILITY DETECTION & REPAIR IN THE IDE . 160
6.1 Introduction 161
6.2 User Study Interface 163
6.2.1 IDE Integration 164
6.2.2 Model Architecture & Training 165
6.2.3 Evaluating Detection and Fix Capabilities 168
6.3 User Study Design 169
6.3.1 Study Design 169
6.4 User Study results 173
6.4.1 RQ1: Is DeepVulGuard useful in practice? 173
6.4.2 RQ2: Which aspects of vulnerability detection + fix tools are most useful? . 176
6.4.3 RQ3: What features do developers want from vulnerability detection + fix tools? 180
6.5 Discussions 182
6.6 Threats to Validity 183
6.7 Related Work 184
6.8 Conclusions 185
6.9 Bibliography 186
6.A Appendix: Detection model 194
6.A.1 Training procedure 194
6.A.2 Dataset statistics 195
6.A.3 Full list of languages in the training dataset, by file extension 195
6.A.4 Hyperparameters 196

CHAPTER 7. GENERAL CONCLUSION 197
7.1 Summary of Contributions 197
7.2 Future Work 198
7.3 Bibliography 199

LIST OF TABLES

Page

Table 2.1 11 Reproduced Models. 10

Table 2.2 Model reproduction on their original datasets. 11

Table 2.3 Variability over 3 random seeds on Devign dataset. 13

Table 2.4 Agreement across different models. 13

Table 2.5 Five types of vulnerabilities. 14

Table 2.6 The similarity of important feature sets between every two models. 26

Table 2.7 The frequently highlighted code features. 27

Table 3.1 $OUT[v]$ at each iteration of DFA. 54

Table 3.2 Hyperparameters used for training DeepDFA. 56

Table 3.3 Performance comparison between DeepDFA and baselines. 60

Table 3.4 Comparison with non-transformer models. 60

Table 3.5 Comparison with transformer models. 60

Table 3.6 Results of statistical tests for model comparison. 62

Table 3.7 Training and inference time of DeepDFA and baselines. 63

Table 3.8 Performance of DeepDFA and baselines on limited data. 64

Table 3.9 How do the models handle unseen projects? 65

Table 3.10 Generalization performance of DeepDFA and baselines. 66

Table 3.11 Ablation study evaluated on DbgBench. 67

Table 3.12 Ablation study evaluated on the Big-Vul test dataset. 68

Table 3.13 Initial trial run of performance on 100% of the Big-Vul dataset. 81

Table 3.14 Approximate training times of all models. 82

Table 3.15 DeepDFA was smallest in terms of parameter count. 82

Table 3.16 Initial trial run of cross-project evaluation with 100% of the dataset. 83

Table 4.1 TRACED's design of quantized variable values. 94

Table 4.2 Details of downstream task datasets. 103

Table 4.3 Performance on static execution estimation. 106

Table 4.4 Comparison of Clone Retrieval and bug detection. 111

Table 5.1 Performance on vulnerability detection vs. NL/math reasoning, code generation, and code execution. 130

Table 5.2 Models' abilities to distinguish pairs of vulnerable and non-vulnerable examples. 131

Table 5.3 Error analysis from 300 responses covering 100 programs. 133

Table 5.4 14 models we studied. 154

Table 5.5 Text generation parameters we used. 154

Table 5.6 The performance of the studied models on simple CWE examples. 156

Table 5.7 Definitions of Model Reasoning Errors. 159

Table 6.1 Proportion of alerts in each language. 195

LIST OF FIGURES

Page

Figure 2.1 Same-bugtype and cross-bugtype performance. 16

Figure 2.2 Comparative performance on evaluation sets selected according to LR model difficulty score. 19

Figure 2.3 Coefficients of LR models trained on the stable examples from the Devign dataset. 20

Figure 2.4 F1 score on a held-out test set when models are trained with increased portions of the training dataset. 22

Figure 2.5 Studies on project composition in training data. 24

Figure 3.1 Overview of DeepDFA. 45

Figure 3.2 Dataflow Analysis. 48

Figure 3.3 Graph Learning. 48

Figure 3.4 Analogy of information propagation in Dataflow Analysis and Graph Learning. 48

Figure 3.5 Abstract dataflow embedding generation. 51

Figure 4.1 A motivating example for TRACED. 86

Figure 4.2 Overview of the workflow of TRACED. 90

Figure 4.3 Program states with concrete runtime values. 92

Figure 4.4 High-level model architecture of TRACED. 97

Figure 4.5 A qualitative example of execution coverage prediction. 107

Figure 4.6 A qualitative example of runtime value prediction. 108

Figure 4.7 Comparing TRACED's design of quantized variable values with other value abstraction strategies. 110

Figure 5.1 Example of a Buffer Overflow (BOF). 126

Figure 5.2 Example of a Null-Pointer Dereference (NPD). 126

Figure 5.3 Examples of vulnerability detection as a complex code reasoning task. . . . 126

Figure 5.4 Vulnerability detection performance. 129

Figure 5.5 Error categories observed in responses from all LLMs. 132

Figure 5.6 Missed Bounds/NULL check. 134

Figure 5.7 Misunderstood arithmetic operation. 135

Figure 5.8 Larger models did not improve on vulnerability detection. 136

Figure 5.9 Expanding the training dataset and incorporating fine-tuning had minimal impact on vulnerability detection capability. 137

Figure 5.10 Example of our CoT-Annotations prompt. 138

Figure 5.11 Domain knowledge is somewhat helpful for one step but not much for overall performance. 139

Figure 5.12 A simple integer overflow example collected from CWE database. 155

Figure 5.13 Code LLAMA's response to the simple example in Figure 5.12. 155

Figure 5.14 Error analysis user interface. 157

Figure 6.1 An overview of DeepVulGuard's user interface on an example program. . . . 164

Figure 6.2 An overview of DeepVulGuard's detection workflow. 166

Figure 6.3 DeepVulGuard's LLM filter prompt. 166

Figure 6.4 DeepVulGuard's fix model prompt. 167

Figure 6.5 Performance of DeepVulGuard's detection component on SVEN. 168

Figure 6.6 Participant demographics and tool adoption. 170

Figure 6.7 Summary of participants' overall perceptions of DeepVulGuard, from our post-interview survey. 174

Figure 6.8 Participant responses to LLM-filtered alerts and LLM-generated fixes while using DeepVulGuard. 176

Figure 6.9 Participants' in-use feedback on the aspects of DeepVulGuard. 177

Figure 6.10 The relative frequency of features suggested by the study participants. 181

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