Fine-Tuning Large Language Models for Software Vulnerability Detection

Dan Schrage
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
dschrag2@nd.edu

I. Introduction

Security breaches pose significant threats to individuals, corporations, and society at large. For individuals, breaches often result in the loss of sensitive personal data such as credit card information, social security numbers, or medical records, which can be exploited for identity theft and financial fraud. For corporations, the consequences can be even more severe: data breaches can compromise intellectual property, damage customer trust, and lead to direct financial losses through theft or costly incident response efforts.

A leading cause of security breaches is unsafe or vulnerable code that traditional static analysis tools often fail to detect. With the rise of advanced machine learning, models such as VulDeePecker [5] and Devign [12] have demonstrated the feasibility of detecting vulnerabilities from source code using learned representations. These models illustrate the benefits of deep learning and graph-based approaches for capturing semantic and structural information in code, improving vulnerability detection beyond rule-based tools.

While these methods represent important advances, large language models (LLMs) offer unique advantages for software vulnerability detection. Pre-trained on massive corpora of code, LLMs can learn rich syntactic and semantic patterns across programming languages and project types. They are capable of zero- or few-shot reasoning, enabling vulnerability detection even in codebases that were unseen during training. Additionally, LLMs can potentially generalize to new vulnerability types and provide contextual insights to developers, complementing traditional static analysis and specialized ML models. These properties make LLMs a promising tool for building more robust, scalable, and adaptable vulnerability detection systems.

Recent research has begun to explore the use of LLMs for software security, investigating both their capabilities and limitations. Studies have applied LLMs to tasks such as vulnerability detection, patch generation, code summarization, and secure coding suggestions. Early results indicate that fine-tuned LLMs can achieve performance comparable to or exceeding specialized models for certain types of vulnerabilities, while prompted LLMs show potential in zero- and few-shot scenarios. However, challenges remain, including sensitivity to prompt design, varying performance across programming languages, and limited interpretability of the model's reasoning. Additionally, systematic benchmarks comparing LLM-based approaches to traditional static analyzers and graph-based models are still scarce, leaving important questions about their generalizability, reliability, and practical deployment in real-world development workflows.

In this paper I focus on a critical, unanswered question regarding the generalization of fine-tuned LLMs for security vulnerability detection. While previous work has shown that LLMs can find vulnerabilities, it is unclear whether these models are understanding the underlying concepts of these vulnerabilities or simply overfitting to the patterns of a given dataset. To address this gap, I will conduct a cross-dataset generalization study. I will fine-tune a model on "real-world" vulnerabilities (found in the Big-Vul dataset) and test its ability to find "textbook" vulnerability examples (found in the Juliet Test Suite) and vice-versa. This analysis will assess the robustness of these models and their suitability for real-world applications.

This project aims to address these gaps through the following research question:

- **RQ:** To what extent do fine-tuned Large Language Models for vulnerability detection generalize across different data domains (i.e., from "textbook" code to "real-world" code and vice-versa)?

II. Related Work

The task of automatically detecting software vulnerabilities has been addressed through a variety of approaches, ranging from traditional static analysis tools to recent machine learning and deep learning models. In this section, I will review existing work within these three domains: static analysis techniques, machine learning-based detection, and large language model approaches.

A. Static Analysis Tools

Static analysis tools are widely used in practice to detect coding errors that could lead to vulnerabilities. These tools examine source code without executing it, following a set of predefined rules to map control and logic flows. They can be highly effective at catching security flaws such as buffer overflows, SQL injection risks, or improper input validation. However, their effectiveness is often limited by the scope of their rule sets and their sensitivity to different language-specific and framework-specific constructs.

Tools such as CodeQL [4], semgrep [7], ITS4 [10], SPLINT [1], and FindBugs [8] perform rule-based analysis of source code to identify potential weaknesses. While effective at catching certain low-level issues, these tools often suffer from high false positive and false negative rates and are limited by the coverage and specificity of their rule sets. Moreover, they generally do not capture higher-level design or architectural flaws, which can propagate vulnerabilities in a system-wide manner.

B. Machine Learning Approaches for Vulnerability Detection

Recent research has explored machine learning models to automatically detect vulnerabilities from code. VulDeePecker [5] was one of the first deep learning approaches to do so, introducing the concept of *code gadgets*—semantically related lines of code extracted from a program. These gadgets are tokenized and fed into a Bidirectional Long Short-Term Memory (BiLSTM) network, which learns to classify each *gadget* as vulnerable or not. VulDeePecker was evaluated on both benchmark datasets and real-world projects, demonstrating reduced false negatives compared to traditional static analysis tools. The model was able to detect previously unknown vulnerabilities that were later confirmed in 3 upstream software projects, illustrating its practical effectiveness. This work highlighted the importance of capturing semantic information in code for automated vulnerability detection.

Devign [12] extended the deep learning paradigm by modeling program semantics as graphs and applying Graph Neural Networks (GNNs) for function-level vulnerability detection. Using program dependence graphs (PDGs), Devign represents both the control-flow and data-flow dependencies in a function, capturing structural relationships that sequential models cannot easily learn. A graph convolutional module aggregates node-level information into function-level embeddings, which are then classified as vulnerable or not. Devign was trained on large-scale real-world C projects and demonstrated higher accuracy and F1 scores than VulDeePecker and prior approaches. This showed that leveraging structural and semantic information in code significantly improves vulnerability detection performance. Furthermore, Devign's use of graphs enables it to generalize better across diverse code patterns and vulnerability types.

Together, VulDeePecker and Devign illustrate the benefits of using learned representations of code for automated vulnerability detection. They show that deep learning and graph-based models can capture semantic and structural information that static analysis tools often miss, reducing false negatives and improving the detection of both known and previously unseen vulnerabilities.

C. Large Language Models (LLMs) for Code Analysis

Large language models pre-trained on code, such as Codex, CodeT5, and CodeBERT, have recently been explored for vulnerability detection. These models leverage vast amounts of source code data to learn syntactic and semantic patterns, enabling zero- or few-shot detection of vulnerabilities in novel codebases. Fine-tuned LLMs have been shown to perform comparably or better than specialized models on certain tasks, but systematic comparisons with GNN-based methods and traditional static analyzers remain limited. Moreover, the relationship between the scale of code pre-training and detection performance has not been thoroughly investigated, leaving open questions about how much pre-training data is necessary for robust vulnerability detection.

Fine-tuning LLMs for vulnerability detection is important for several reasons. First, it enables models to specialize in identifying security vulnerabilities by training them on labeled datasets containing examples. This specialization can lead to improved performance over general-purpose models. For instance, Shestov et al. [9] demonstrated that fine-tuning the WizardCoder model on vulnerability datasets resulted in significant improvements in detection accuracy, outperforming CodeBERT-like models. They also explored techniques to handle class imbalance and accelerate training, further enhancing model performance.

Second, fine-tuning enables models to adapt to specific programming languages and coding styles prevalent in different development environments. This adaptability is essential for detecting vulnerabilities in diverse codebases. Yang et al. [11] introduced the MSIVD framework, which combines self-instructed fine-tuning over multiple tasks with program control flow graphs encoded as GNNs. Their approach achieved superior performance on the BigVul dataset, which highlighted the importance of tailoring models to the nuances of specific programming languages and frameworks.

In summary, fine-tuning LLMs for vulnerability detection is a promising direction that leverages the strengths of large-scale pre-trained models while adapting them to the specific challenges of software security. By specializing models through fine-tuning, researchers and practitioners can develop more effective tools for identifying and mitigating vulnerabilities in software systems.

D. Work Novelty

While prior work has established that fine-tuned LLMs cna be effective at detecting software vulnerabilities, the depth of their understanding is not well understood. The novelty of this project lies in its specific, empricial focus on cross-domain generalization.

I am not merely benchmarking a model's performance on a single dataset. Instead, I am investigating whether these models are learning the underlying concept of a security vulnerability. I will train on a ”real-world” dataset (Big-Vul) and test on a ”synthetic” one (Juliet Test Suite), and then I will reverse the process. Through this testing, I aim to be the first to provide a clear analysis of this generalization gap. This addresses a major open question, as a model that performs well only on data similar to its training set cannot be used effectively in other settings.

III. Methodology

This study proposes fine-tuning a large language model (LLM) for software vulnerability detection and systematically comparing its performance against prior machine learning methods, few-shot prompted LLMs, and conventional static analysis tools.

A. Research Question

As stated previously, here is the proposed research question:

RQ: To what extent do fine-tuned Large Language Models for vulnerability detection generalize across different data domains (i.e., from “textbook” code to “real-world” code and vice-versa)?

B. Datasets

I will use two publicly available datasets for evaluation:

- **Big-Vul**: Real-world C/C++ functions linked to CVEs [2].
- **Juliet Test Suite**: Synthetic CWE-labeled programs for controlled testing [6].

Both datasets are freely accessible for research purposes. They provide a combination of real-world and synthetic vulnerabilities, enabling robust evaluation and cross-dataset generalization experiments.

C. Large Language Model

For this study, I will use a single, pre-trained model: CodeBERT-Base [3]. This is a RoBERTa-style model pre-trained on a large corpus of code data.

I will fine-tune the model for binary sequence classification (0 for non-vulnerable, 1 for vulnerable) with a learning rate of 2×10^{-4}, a batch size of 24, and for one epoch. These hyper-parameters were optimal for a RTX 3060 Laptop GPU. I use the `RobertaForSequenceClassification` class from the Hugging Face `transformers` library.

D. Evaluation Metrics

Performance will be measured using standard metrics:

- Accuracy
- Precision
- Recall
- F1-score

These metrics are standard for binary classification tasks. They will show me an interpretable depiction of the model’s ability to detect vulnerabilities.

E. Experimental Procedure

The experiments will proceed as follows:

1) Experiment A: Real-to-Synthetic
    - **Training:** The pre-trained `CodeBERT-Base` model will be fine-tuned exclusively on the `Big-Vul` training split. This is the ”real world” dataset.
    - **Evaluation:** The resulting fine-tuned model will be tested on the `Julet Test Suite` test split.
    - **Purpose:** This measures the model’s ability to apply knowledge learned from ”real-world” examples to ”textbook” vulnerability examples.
2) Experiment B: Synthetic-to-Real
    - **Training:** The pre-trained `CodeBERT-Base` model will be fine-tuned exclusively on the `Juliet Test Suite` training split. This is the ”synthetic/textbook” dataset.
    - **Evaluation:** The resulting fine-tuned model will be tested on the `Big-Vul` test split.
    - **Purpose:** This measures the model’s ability to apply knowledge learned from ”textbook” examples to ”real-world” vulnerability examples.

The precision, recall, and F1-scores from these two cross-domain evaluations will be compared to directly answer the research question.

IV. Results

In this section, we present the experimental results of our cross-domain generalization study. We evaluated the performance of the fine-tuned CodeBERT model in two distinct scenarios: training on real-world data to test on synthetic data (Experiment A), and training on synthetic data to test on real-world data (Experiment B). Table I summarizes the performance metrics for both experiments.

TABLE I: Cross-Domain Generalization Results. **Exp A** trains on real-world data and tests on synthetic. **Exp B** trains on synthetic data and tests on real-world. Note the ”Accuracy Paradox” in Experiment B, where high accuracy masks extremely low precision and recall.

Experiment	**Accuracy**	**Precision**	**Recall**	**F1-Score**
A: Real $\rightarrow$ Synthetic	59.11%	74.33%	27.83%	40.50%
B: Synthetic $\rightarrow$ Real	92.24%	7.70%	13.65%	9.84%

V. Discussion

Our findings demonstrate that generalization in vulnerability detection is not bidirectional. While models trained on real-world data exhibit a moderate ability to transfer their knowledge to synthetic domains, the reverse is not true. This section analyzes the causes of this asymmetry and the implications for future dataset curation.

A. The Asymmetry of Generalization

The most significant finding of this study is that real-world training promotes robustness, while synthetic training promotes overfitting. In Experiment A, the model trained on *Big-Vul* achieved a precision of 74.33% on the unseen *Juliet* dataset. This suggests that by learning from the ”messy” noise of real-world code, the model acquired abstract concepts of vulnerability that held true even when tested on rigid, textbook examples.

Conversely, the model trained on *Juliet* failed to generalize to the real world (Experiment B), achieving a precision of only 7.70%. This indicates that the model likely overfitted to the specific artifacts of the synthetic generation process (such as specific variable names or rigid structures) rather than learning the underlying semantic logic of vulnerabilities. This theory can be verified by further inspection of the Juliet Test Suite dataset. The "vulnerable" code samples often include comments or variable names that directly indicate that a function or code block is vulnerable. For example, one data point has a comment that says "bad function declaration" directly before the vulnerable code segment. In turn, this likely trained the model to identify obvious comments and variables as opposed to underlying vulnerabilities.

B. *The Accuracy Paradox*

Experiment B presents a paradox in the accuracy score. The model achieved a high accuracy of 92.24%, which suggests strong performance. However, this figure is a direct result of class imbalance in the real-world testing set. Since 94.2% of real-world data points are labeled "not vulnerable," a model that simply predicts "Safe" for every input will achieve high accuracy while failing its primary objective.

The critically low Recall (13.65%) and F1-score (9.84%) reveal the true failure: the model was unable to distinguish actual threats from safe code, defaulting to a safe prediction in most cases. This highlights why accuracy is a poor metric for security tasks, where the cost of a false negative (missing a vulnerability) is extremely high.

As previously discussed, the Juliet Test Suite dataset contains explicit vulnerability labels within the code. However, when a model trained on this dataset attempts to generalize to a different dataset without the labels, it may assume that everything is safe. This is because there is no label explicitly telling the model what is vulnerable and what is not. This would explain the defaulted "Safe" choice from this model. It is not confirmed that this is why this model acted this way, but it is a possible theory.

C. *Implications for Synthetic Data*

These results challenge the prevailing reliance on synthetic datasets for training security models. While synthetic data (like *Juliet*) provides balanced and clean labels, it lacks the entropic variety of real software. Our study suggests that synthetic data is insufficient for training deployable models on its own. Instead, it may be better suited for benchmarking or as a pre-training step in a curriculum learning framework, rather than as the primary source of truth.

Our study shows that the messy complexity of real world code boosts Large Language Model performance on vulnerability detection. This suggests that real world examples force the model to learn the underlying concept of a vulnerability as opposed to searching for patterns and labels. Furthermore, synthetic testing data has easier-to-detect patterns, which may allow for an easier testing environment and not promote rigorous models.

D. *Overall Poor Performance*

While Experiment A provided superior performance to Experiment B, both models scored relatively low precision, recall, and F1 scores. This could be due to several reasons.

First of all, cross generalization is much more difficult than training and testing on the same model. There could be vulnerability types covered in one dataset that are not mentioned in another. Coding style and patterns will differ between datasets. This unfamiliar environment can lower a model's accuracy because it has never similar code before. This could suggest that both models were overfit to the patterns and labels of their respective datasets. Albeit, Experiment A, was still much more effective at generalizing, and was thus less overfit.

Another plausible reason for overall low scores is the choice of model and training hyperparameters. CodeBERT-Base is a relatively small model at 125M parameters, which is a much less powerful model that other code-based models. This lack of power could be the reason for poor scores. Perhaps a larger model could perform better on these tasks. Additionally, training was capped at 1 epoch. This likely limited the performance of the model as it had insufficient time to train.

E. *Future Work*

There are several future steps this research could take.

The first of which would be to properly benchmark the CodeBERT-Base model in a same-domain scenario. By training and testing on the same dataset, we would gain an understanding of the baseline of the model's performance when not in a cross-domain scenario. This would quantify the performance difference when cross generalizing.

As mentioned in the previous subsection, model and training hyperparamters likely limited the performance of this model. With a higher time and compute budget, a more powerful model could be chosen. A stronger model combined with longer training time would likely produce stronger results. These results could give a clearer picture of how generalizable large language models are when trained on real world or synthetic datasets respectively.

VI. Conclusion

In this study, we investigated the extent to which fine-tuned Large Language Models for vulnerability detection generalize across conflicting data domains. Our experiments demonstrate that generalization is fundamentally asymmetric: models trained on real-world code exhibit a moderate ability to transfer knowledge to synthetic benchmarks, whereas models trained on synthetic data fail to generalize to the real world.

These findings highlight a critical limitation in current data-centric security research. While synthetic datasets provide clean, balanced labels, they lack the messy complexity required to teach models robust vulnerability concepts. The failure of the *Juliet*-trained model in Experiment B suggests that without the "noise" of real-world software, LLMs tend to overfit to superficial patterns and labels rather than learning semantic logic.

We conclude that while synthetic data remains valuable for controlled benchmarking, it is insufficient as a primary training source for deployable security tools. Achieving robust, generalizable vulnerability detection will likely require hybrid training strategies and sufficient compute resources.

References

[1] The Splint Developers. *Splint: A Tool for Statically Checking C Programs for Security Vulnerabilities and Coding Mistakes*. 2025. URL: https://www.splint.org/.

[2] Lingming Fan et al. "A C/C++ code vulnerability dataset with code changes and CVE labels". In: *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. 2020, pp. 508–512.

[3] Zhangyin Feng et al. "CodeBERT: A pre-trained model for programming and natural languages". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 1536–1547.

[4] GitHub. *CodeQL*. 2025. URL: https://codeql.github.com/.

[5] Zhen Li et al. "VulDeePecker: A deep learning-based system for vulnerability detection". In: *25th Annual Network and Distributed System Security Symposium (NDSS)*. 2018.

[6] NIST Software Assurance Reference Dataset (SARD). *Juliet Test Suite*. https://samate.nist.gov/SRD/testsuite.php. Accessed: 2025-10-03. 2014.

[7] Return Path. *Semgrep*. 2025. URL: https://semgrep.dev/.

[8] Bill Pugh and David Hovemeyer. *FindBugs*. 2025. URL: https://findbugs.sourceforge.io/.

[9] Aleksei Shestov et al. "Finetuning large language models for vulnerability detection". In: *IEEE Access* (2025).

[10] ITS4 Team. *ITS4: A Tool for Detecting Vulnerabilities in Web Applications*. 2001. URL: https://www.owasp.org/index.php/ITS4.

[11] Aidan ZH Yang et al. "Security vulnerability detection with multitask self-instructed fine-tuning of large language models". In: *arXiv preprint arXiv:2406.05892* (2024).

[12] Yaqin Zhou et al. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.