THESIS

THE IMPACT OF FEW-SHOT EXAMPLES FOR LLM-BASED CODE VULNERABILITY DETECTION

Submitted by

Md Abdul Hannan

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2026

Master's Committee:

Advisor: Ravi Mangal

Sudipto Ghosh
Leo Vijayasarathy

ABSTRACT

THE IMPACT OF FEW-SHOT EXAMPLES FOR LLM-BASED CODE VULNERABILITY DETECTION

Large language models (LLMs) have shown promise for detecting software vulnerabilities through in-context learning (ICL), where a small number of labeled examples are provided in the prompt to guide model predictions. However, the effectiveness of ICL depends critically on how these few-shot examples are selected, and little is studied about what these models actually attend to when making their decisions. This work addresses both gaps directly. First, how should those few-shot examples be selected to maximize detection performance? Second, when the model makes correct predictions, is it actually focusing on the right tokens of the vulnerable code?

The first part studies two selection criteria: Learn-from-Mistakes (LFM), which prioritizes examples on which the model consistently makes errors, and Learn-from-Nearest-Neighbors (LFNN), which retrieves semantically similar examples using a code embedding model. We evaluate these methods and three combined strategies on five datasets spanning C/C++, Python, and JavaScript, using Qwen2.5-Coder-7B-Instruct, Gemma-3-4b-it, and GPT-5-mini. As a single method, LFNN consistently improves performance on Python and JavaScript, while C/C++ datasets remain resistant to few-shot adaptation. Combined methods showed the best performance overall.

The second part takes the LFNN pipeline and Qwen model, and audits its internal attribution behavior using Layer Integrated Gradients (LIG). Our experiments reveal that among the top-50 tokens identified by the attribution method, 40 to 52 percent belong to the injected few-shot examples. More strikingly, fewer than 2 percent of those top-50 tokens correspond to actual vulnerable tokens, even on samples the model predicted correctly. These findings show that even when LFNN improves detection accuracy, gradient-based attributions are not always grounded in the actual

vulnerable code tokens, revealing a gap between performance and reliability in LLM-based code vulnerability detection.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Ravi Mangal, for his guidance, patience, invaluable feedback, and all the interactive meetings throughout this research. I am especially thankful for the opportunity he provided to collaborate with a research group at Carnegie Mellon University, an experience that greatly strengthened my technical skills and shaped my understanding of academic writing. I am also grateful to my committee members, Dr. Sudipto Ghosh and Dr. Leo R. Vijayasarathy, for their time, insights, and serving as my committee.

I would also like to thank Dr. Ravi Mangal for providing access to his research server, Poppy, which was instrumental in running the experiments for this work and significantly reduced the time required for computation.

I would like to thank the Department of Computer Science at Colorado State University for providing the resources and environment that made this work possible.

Finally, I am deeply grateful to my family and friends for their unwavering support and encouragement throughout my graduate studies.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iv
LIST OF TABLES vi

Chapter 1 Introduction 1
1.1 Algorithms for Selecting Few-Shot Examples 1
1.2 Auditing Token-Level Attributions 2
1.3 Thesis Organization 3

Chapter 2 Algorithms for Selecting Few-Shot Examples 4
2.1 Introduction 4
2.2 Related Work 6
2.3 Two Algorithms: LFM and LFNN 8
2.3.1 Learn-from-Mistakes (LFM) 8
2.3.2 Learn-from-Nearest-Neighbors (LFNN) 11
2.4 Combining the Two Algorithms 12
2.5 Experiments 14
2.6 Conclusion 26

Chapter 3 Auditing Token-Level Attributions 27
3.1 Introduction 27
3.2 Related Work 28
3.3 Methodology 29
3.3.1 Task Formulation 29
3.3.2 Integrated Gradients and Captum 29
3.3.3 Ground Truth 30
3.3.4 Metrics 30
3.4 Experiments and Results 31
3.4.1 Datasets 31
3.4.2 Results 31
3.5 Discussion 34
3.5.1 What the Results Mean 34
3.5.2 Comparison with Prior Work 35
3.5.3 Limitations 35
3.6 Conclusion 35

Bibliography 37

LIST OF TABLES

2.1 Results for all approaches across three models, with standard deviation. 20
2.2 Performance comparison of different LFM variations. Each cell shows the mean and standard deviation over five runs. For the LFM variation names, the following abbreviations are used: S = stacked, U = unstacked, 1× = one iteration, m× = more than one iteration, inc. = incorrect, corr. = correct, and gray = gray cases. '–' indicates metrics that are undefined due to division by zero. 24

3.1 Token count statistics by prompt region and dataset. 32
3.2 Attribution analysis: correct predictions ($K = 50$). 32
3.3 Attribution analysis: incorrect predictions ($K = 50$). 33

Chapter 1

Introduction

Software vulnerabilities are a persistent and serious problem in modern software development. A single undetected vulnerability can lead to data breaches, system failures, or arbitrary code execution [1]. Detecting these vulnerabilities reliably and at scale remains one of the core open challenges in software security [2]. Traditional approaches such as static analysis tools and rule-based scanners have been widely used [3], but they struggle to generalize across diverse codebases and often produce high rates of false positives [4].

Large language models (LLMs) have recently emerged as a promising alternative. Trained on vast corpora of source code and natural language, these models can reason about code structure and semantics in ways that earlier machine learning approaches could not [5]. Several studies have shown that LLMs can achieve competitive performance on vulnerability detection benchmarks [4, 6], particularly when given a small number of labeled examples in their prompt through in-context learning (ICL) [4, 7]. However, two fundamental questions remain open. First, how should those few-shot examples be selected to maximize the model's detection performance? Second, even when the model makes correct predictions, is it actually reasoning about the right parts/tokens of the code?

This thesis addresses both questions in sequence. The two chapters are closely connected: Chapter 2 builds the prompting pipeline and studies how to make it perform better, while Chapter 3 audits the internal behavior of that same pipeline to ask whether its attributions are faithful to the actual vulnerable code.

1.1 Algorithms for Selecting Few-Shot Examples

Chapter 2 investigates the problem of few-shot example selection for LLM-based vulnerability detection.

We explore two selection criteria. The first, Learn-from-Mistakes (LFM), selects examples based on the model's own past behavior, prioritizing programs on which the model consistently makes errors. The intuition is that these hard examples expose systematic weaknesses and, when placed in the prompt context, help the model correct them. The second, Learn-from-Nearest-Neighbors (LFNN), selects examples based on semantic similarity to the program under analysis, using a code embedding model to retrieve the closest training samples. We also explore three combined methods that blend both criteria in different ways.

We evaluate these methods on five datasets spanning C/C++, Python, and JavaScript, using three models: Qwen2.5-Coder-7B-Instruct, Gemma-3-4b-it, and GPT-5-mini. The results show that careful example selection, particularly LFNN and combined methods, can lead to measurable improvements for Python and JavaScript datasets, while C/C++ datasets remain more resistant to few-shot adaptation. The combined methods generally provide more stable performance across diverse settings.

1.2 Auditing Token-Level Attributions

Chapter 3 takes the LFNN pipeline from Chapter 2 and asks a different kind of question. When the model makes a correct vulnerability detection decision using LFNN-retrieved few-shot examples, is it actually attending to the tokens that correspond to the known vulnerability?

To answer this, we apply Layer Integrated Gradients (LIG) to the Qwen2.5-Coder-7B-Instruct model running the LFNN setup, and measure how well the resulting attribution scores align with ground-truth patch token indices derived from code diffs and NodeTaint location fields. We introduce a Few-Shot Contribution metric that directly measures how much of the model's top-attributed tokens belong to the injected few-shot examples rather than the target code.

The results are consistent across four datasets: PyVul Python, PyVul Javascript/Typescript, NodeMedic, and SVEN Python. Roughly 40 to 52 percent of the model's top-50 attributed tokens belong to the few-shot retrieval context, while patch tokens rank on average between position 584 and 1604 in the full attribution list. These findings show that even when the LFNN pipeline

improves detection accuracy, the model's internal attribution sometimes is not grounded in the actual vulnerable code regions.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents the few-shot example selection study, including the LFM and LFNN algorithms, their combinations, and a full experimental evaluation across five datasets and three models. Chapter 3 presents the token-level attribution analysis study, including the evaluation framework, metrics, and results. Each chapter contains its own related work discussion relevant to its specific contribution.

Chapter 2

Algorithms for Selecting Few-Shot Examples

2.1 Introduction

Large language models (LLMs) have demonstrated impressive capabilities for coding tasks including detection of software defects and vulnerabilities (such as resource leaks, use-after-free of dynamic memory during program operation, and denial-of-service attacks), variable misuse detection (i.e., programmers used wrong variables), code summarization (i.e., represent a code segment with a single word), and code generation and code completion [5, 8–11]. In this paper we explore the applicability of LLMs for vulnerability detection. This is a particularly important application, as software vulnerabilities are prevalent in many software systems, posing serious risks such as compromising sensitive data and system failures. There are limited studies in previous work that show that state-of-the-art LLMs with few-shot-learning capabilities can achieve competitive results in detecting software vulnerabilities compared to previous machine learning techniques [4, 7]. However, the performance (measured in terms of precision, recall, F1 score) of these models remains low [12, 13] preventing the deployment of these models in realistic settings.

In this paper, we explore black-box, prompt-based methods for improving the performance of LLMs on vulnerability detection. We seek to evaluate techniques for improving the efficacy of in-context learning (ICL)—the well-known phenomenon of LLM's exhibiting improved performance when the provided prompt includes a set of examples (referred to as the *few-shot set*) that demonstrate the task to be performed—for the purpose of vulnerability detection. Previous works indicate that the impact of ICL on model performance is highly sensitive to the specific examples chosen [14–16]. For vulnerability detection, the few-shot set comprises programs along with the ground-truth label indicating whether they have a vulnerability or not.

We explore methods for choosing the example programs that comprise the few-shot set used in the code vulnerability detection task. In particular, we explore two natural algorithms for choosing

the few-shot examples. The first algorithm, **Learn-from-Mistakes** (LFM), is based on the idea that LLM performance on a sample is informative about its usefulness as a few-shot example. Therefore, LFM first queries the model (multiple times) on a potential example and records if the model is consistently mistaken (or correct) on the example. An example is added to the few-shot set only if the model consistently *makes mistakes on it*. The intuition is that the model does not know well how to reason about that example therefore adding it in the few-shot set will likely help rectify the model behavior, while a randomly chosen example may not be as useful. We also explore alternate versions of LFM where an example is added only if the model is consistently correct on it. The intuition here is that adding examples where the model is consistently correct can help reinforce the desired behavior and improve performance. The second algorithm, **Learn-from-Nearest-Neighbors** (LFNN), is based on the intuition that the semantic similarity between an example and the program under test can be a helpful guide in deciding if the example should be added to the few-shot set. In particular, LFNN adds the nearest neighbors to the set. To compute semantic similarity, LFNN relies on code embedding models that map programs to embedding vectors. Similarity of vectors can be computed using metrics such as cosine similarity.

We further explore different ways for combining LFM and LFNN to yield few-shot sets that are both semantically similar to the program under test and provide a context summarizing the model's past mistakes. Each combination yields a unique algorithm to construct the few-shot set for the vulnerability detection task and represents a specific trade-off between the two criteria—model performance and semantic similarity—for selecting examples.

We evaluate these methods using two popular open-source models that are known to perform well on coding tasks, namely **Qwen2.5-Coder-7B-Instruct** and **Gemma-3-4B-it**, as well as a closed-source model, **GPT-5-mini**. We use four challenging datasets for the evaluation that include programs from a variety of languages such as C/C++, Python, and JavaScript.

Our experimental results show that while certain standalone strategies, particularly LFNN, can improve baseline vulnerability detection performance, their effectiveness varies across models and datasets. LFM introduces a strong inductive bias that tends to skew predictions toward the

positive class, which often limits its applicability. In contrast, the combined methods provide more stable and generally more balanced performance, although their benefits are not uniform. Depending on the underlying model and dataset, these combinations may yield meaningful gains in both accuracy and F1-score, or only modest or no improvements, reflecting the heterogeneous nature of the vulnerability detection task.

2.2 Related Work

Vulnerability Detection Vulnerability detection is a long-standing challenge in software security. Over the years, a wide range of techniques have been developed to detect vulnerabilities, ranging from static program analysis [17, 18] to dynamic testing and fuzzing [19–21], and from these traditional methods to more recent machine-learning-based approaches [22–25].

Machine-learning-based approaches aim to detect vulnerabilities by learning patterns directly from data. Early work extracted hand-crafted features such as token frequencies, code complexity metrics, or dependency patterns, and trained classifiers to separate vulnerable from non-vulnerable code [26, 27]. More recent methods use representation learning to automatically encode code or abstract representations of code into vector spaces that possibly capture both syntax and semantics, enabling models to recognize patterns that are difficult to design manually. These approaches have shown promising results [24, 28–31], but their effectiveness is often constrained by dataset quality and the challenge of generalizing across diverse projects [32–35].

LLM-based Vulnerability Detection Recent attempts to apply large models (LM) to vulnerability detection generally fall into two broad approaches. The first line of work treats LMs as embedding models, extracting vector representations of code and then either training lightweight classifiers on top of these embeddings or fine tuning the LM itself for the downstream task [24, 36, 37]. Many of these methods also incorporate a program's abstract representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), or data flow features to complement the raw code input [28, 30, 31, 38]. Early research in this category has often focused on models like Code-

BERT [39] and GraphCodeBERT [40], with more recent work exploring larger foundation models such as Qwen [41–44] and LLaMA [45–47].

The second line of work leverages the generative capabilities of modern large language models (LLM) more directly. Instead of relying only on embeddings, these approaches test the ability of LLMs to reason about and classify vulnerabilities through code understanding and natural language generation. Some studies evaluate models in their pretrained form, while others fine tune the models to improve task performance [9]. Several recent efforts reflect this trend. [48] introduce VulLLM, a multi-task LLM framework that integrates vulnerability interpretation and data augmentation to significantly improve code vulnerability detection. [49] leverage few-shot prompting to enhance out-of-the-box LLMs for vulnerability detection. [50] proposes knowledge-level retrieval-augmented generation (RAG) for code vulnerability detection and reports sizable gains over baselines.

Our work falls into this second category. Specifically, we focus on evaluating LLMs out of the box in a few-shot classification setting.

In-Context Learning and Example Selection In-context learning (ICL) [51] enables a model to adapt to a new task by conditioning on a small number of input output examples provided in the prompt. These examples serve as implicit supervision, enabling the model to infer the task format and desired output style without expensive fine tuning. ICL has been widely studied in natural language and code related tasks, where it has been shown to substantially improve model performance over zero-shot prompting [51–54].

A central question in ICL is how to select the few examples that are most useful for the model. Selection strategies can range from simple random sampling to more sophisticated methods that consider similarity between the test input and candidate examples, prior evidence about LLM behavior, or task specific heuristics [14–16, 55, 56]. [14] retrieve examples that are semantically-similar to a test sample to formulate its corresponding prompt. [56] train an efficient dense retriever to select training examples as prompts at test time. [16] employ in-context learning to create expert profiles that condition LLM responses. In our setting, we investigate new strategies for select-

ing few-shot examples and study how they influence the performance of LLMs on vulnerability detection.

Prompt Optimization Prompt optimization broadly refers to techniques that improve how tasks are presented to large language models so that they yield more reliable outputs [57, 58]. Beyond manual design, recent work has begun to explore automated prompt optimization. DSPy [59] introduces a declarative framework that compiles language model pipelines into self-improving programs. GEPA [60] uses reflective natural-language feedback and evolutionary search to iteratively refine prompts. Maestro [61] extends this line of work to multi-agent settings by jointly optimizing node configurations and the structure of agent graphs to better mitigate structural failure modes. These advances connect closely to the problem of example selection in ICL, as the choice and arrangement of input–output demonstrations influences model performance [62–65]. In our work, we treat the construction of the few-shot example set as a form of prompt optimization and study how different selection strategies affect the performance of LLMs for vulnerability detection.

2.3 Two Algorithms: LFM and LFNN

We explore two algorithms for choosing the examples presented in the LLM's context when performing vulnerability detection. The output of both these algorithms is a set of few-shot examples where each example is a program with a yes/no label indicating if a vulnerability is present or not.

2.3.1 Learn-from-Mistakes (LFM)

Algorithm 1 chooses few-shot examples based on the intuition that the correctness of the LLM response on an example is informative about its usefulness as a few-shot example. The algorithm makes a linear scan over a labeled dataset—for each sample in the dataset, it queries an LLM asking it to predict if the sample program has a vulnerability or not. This information is used to construct a few-shot set from this dataset (which we refer to as the *training* dataset). In its default version, the algorithm, which we call **Learn-from-Mistakes** (LFM), operates under the assumption that

Algorithm 1: Learn from Mistakes (LFM)

Input: (1) Vulnerability detection model f_{VD}; (2) Training dataset D with m labeled samples; (3) Initial few-shot set $\mathcal{S}_{init}$ with r labeled samples; (4) Output few-shot set size n; (5) Boolean st indicating stacked or unstacked version; (6) Number of iterations k; (7) Option opt choosing between incorrect (I), correct (C), and gray (G)

Output: Few-shot set $\mathcal{S}$

// Default values of parameters are $st = \texttt{TRUE}, k = 1, opt = I$

$\mathcal{S}_C \leftarrow D$, $\mathcal{S}_I \leftarrow D$;
foreach $idx \in \{1, \ldots, k\}$ **do**
  $\mathcal{S}_{ctxt} \leftarrow \mathcal{S}_{init}$, $\mathcal{S}_{Ci} \leftarrow \emptyset$, $\mathcal{S}_{Ii} \leftarrow \emptyset$;
  foreach $(x, y) \in D \setminus \mathcal{S}_{init}$ **do**
    Compute prediction $\hat{y} = f_{VD}(\mathcal{S}_{ctxt}; x)$;
    if $\hat{y} \neq y$ **then**
      $\mathcal{S}_{Ii} \leftarrow \mathcal{S}_{Ii} \cup \{(x, y)\}$;
      if st **then** $\mathcal{S}_{ctxt} \leftarrow \mathcal{S}_{ctxt} \cup \{(x, y)\}$;
    else
      $\mathcal{S}_{Ci} \leftarrow \mathcal{S}_{Ci} \cup \{(x, y)\}$;
  end
  $\mathcal{S}_C \leftarrow \mathcal{S}_C \cap \mathcal{S}_{Ci}$;
  $\mathcal{S}_I \leftarrow \mathcal{S}_I \cap \mathcal{S}_{Ii}$;
end
$\mathcal{S}_G \leftarrow D \setminus (\mathcal{S}_C \cup \mathcal{S}_I)$;
$\mathcal{S}_{opt} \leftarrow \mathcal{S}_I$;
if $opt = C$ **then**
  $\mathcal{S}_{opt} \leftarrow \mathcal{S}_C$
else if $opt = G$ **then**
  $\mathcal{S}_{opt} \leftarrow \mathcal{S}_G$
if st **then**
  $\mathcal{S} \leftarrow \mathcal{S}_{init}$;
  $\mathcal{S}_{rand} \leftarrow$ Uniformly draw $(n - |\mathcal{S}|)$ samples from $S_{opt} \setminus \mathcal{S}_{init}$;
  $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}_{rand}$;
else
  $\mathcal{S} \leftarrow$ Uniformly draw n samples from S_{opt};
return $\mathcal{S}$;

the examples on which the LLM makes mistakes are more informative than random examples and should be added to the few-shot set. It iteratively updates the few-shot set based on incorrect predictions. However, the algorithm can be configured to run under various other settings that we describe next. Although some variants may also select correct or gray examples, we hold onto

the name Learn-from-Mistakes because the selection process is defined through mistake-based scanning. Mistakes serve as the primary signal for identifying informative examples.

The primary inputs to the algorithm are the LLM f_{VD}, a labeled dataset D with m pairs of programs and their corresponding yes/no label indicating the presence or absence of a vulnerability, an initial set $\mathcal{S}_{init}$ of few-shot examples, and the desired number of examples n in the few-shot set returned by the algorithm. Note that the set $\mathcal{S}_{init}$ can be empty. The remaining inputs to the algorithm are used to configure it. The boolean input st, which stands for *stacked*, indicates whether the context used while querying the LLM during a run of the LFM algorithm should be iteratively refined or not. The input k dictates the number of linear scans over the dataset. Multiple scans help deal with the non-determinism of LLM responses. Finally, the input opt controls if the examples that are added to the few-shot set are the ones where the model makes a mistake or the ones where it is correct.

The algorithm begins by initializing sets $\mathcal{S}_C$ and $\mathcal{S}_I$ with the entire training dataset D (line 1). The $\mathcal{S}_C$ tracks the examples from D where the LLM is correct while $\mathcal{S}_I$ tracks the examples where it is incorrect. The algorithm then enters a loop (lines 2-14) and makes a linear scan over the dataset in each loop iteration. Before starting a scan, the algorithm initializes the few-shot set $\mathcal{S}_{ctxt}$ that will be used when querying the LLM during the linear scan. It also initializes sets $\mathcal{S}_{Ci}$ and $\mathcal{S}_{Ii}$ that track the correctly and incorrectly labeled examples during each iteration. In each linear scan (lines 4-11), the LLM f_{VD} is queried on each of the examples in the dataset (except the ones in the initial set $\mathcal{S}_{init}$). Depending on whether the prediction is correct or not, the example is added to the set $\mathcal{S}_{Ci}$ (line 10) or $\mathcal{S}_{Ii}$ (line 7), respectively. Moreover, if the boolean input st is set to TRUE, the few-shot set $\mathcal{S}_{ctxt}$ used to query the LLM during the current linear scan is updated whenever the model makes a mistake (line 8). At the end of each iteration, the sets $\mathcal{S}_C$ and $\mathcal{S}_I$ of correctly and incorrectly labeled examples are updated (lines 12-13). Note that these sets track the examples that are *consistently* labeled correctly of incorrectly by the LLM across the different iterations. This is enforced by the set intersection operations in lines 12 and 13. After all the linear scans are over

Algorithm 2: Learn from Nearest Neighbors (LFNN)

Input: (1) Training dataset D with m labeled samples; (2) Query instance x; (3) Number of nearest neighbors n; (4) Encoder model enc

Output: Nearest neighbor set $\mathcal{NN}_x$ for the given query instance x

```
// Part 1: General pre-computation
```

$K \leftarrow \emptyset$;
foreach $(x_i, y_i) \in D$ **do**
  $K \leftarrow K \cup (i, enc(x_i))$;
end

```
// Part 2: Instance-specific computation
```

$q \leftarrow enc(x)$, $C \leftarrow \emptyset$;
foreach $(i, k) \in K$ **do**
  $C \leftarrow C \cup (i, cosine(k, q))$;
end

```
// Top_n^2(C) returns the n pairs from C with the largest
   second components
```

$\mathcal{NN}_x \leftarrow \{D[i] \mid (i, k) \in Top_n^2(C)\}$;
return $\mathcal{NN}_x$;

and a final version of the sets $\mathcal{S}_C$ and $\mathcal{S}_I$ have been constructed, the algorithm also constructs a set $\mathcal{S}_G$ of examples where the LLM is not consistently correct or incorrect (G stands for gray).

Finally, the few-shot set $\mathcal{S}$ to be returned by the algorithm is computed (lines 21-26). If the flag st is set to TRUE, the examples in the initial set $\mathcal{S}_{init}$ are included in the set $\mathcal{S}$. The remaining samples in the set $\mathcal{S}$ are chosen from the appropriate sets $\mathcal{S}_C$, $\mathcal{S}_I$, and $\mathcal{S}_G$ (denoted by $\mathcal{S}_{opt}$) as dictated by the value of the opt input (lines 23 and 26).

2.3.2 Learn-from-Nearest-Neighbors (LFNN)

Algorithm 2, which we call the **Learn-from-Nearest-Neighbors** (LFNN) algorithm, chooses few-shot examples based on the intuition that the program samples most *similar* to the program under query are the most helpful in improving LLM performance. Although similar ideas have been proposed in the context of other applications [66], in this work, we use this idea to improve the performance of LLMs for the vulnerability detection task.

The inputs to the LFNN algorithm are a labeled dataset D with m pairs of programs and their corresponding yes/no label indicating the presence or absence of a vulnerability, the program x

under query, the number n of nearest neighbors of x (i.e., the size of the few-shot set $\mathcal{S}$) returned by the algorithm, and the encoder model enc to be used. enc is used to compute embedding vectors for programs which are then used for the nearest neighbor computation.

The algorithm begins with a query-agnostic phase. First, the set of embedding vectors K is initialized to be an empty set (line 1). The algorithm then makes a pass over the dataset D, computes the embedding vector corresponding to each sample program in the dataset, and stores the pair of program index i and the corresponding embedding vector in K (lines 2-4). This part is independent of the query x and can be computed in advance. The key vectors are stored for later use with any query instance x.

The next phase of the algorithm is query-specific. Given a query instance x, the algorithm first computes the embedding vector q for x (line 5). It also initializes a set C to record the similarity between x and the programs in the dataset D. For each vector k in the set of embedding vectors K, the algorithm calculates the cosine similarity between the embedding of x and k (lines 6-8), which measures the similarity between the query embedding and each program embedding from the dataset. The set C records the indices of k and the corresponding cosine similarity with q.

Finally, the algorithm selects the n programs from D that have the highest cosine similarities, identifying the most relevant neighbors to the query (Line 9). These selected programs form the nearest neighbor set $\mathcal{NN}_x$, which serves as the output of this algorithm.

Note that the general pre-computation phase of the algorithm need not be run for each query. Instead, the embedding vectors can be stored and then reused for each new query.

2.4 Combining the Two Algorithms

We explore three different strategies for combining LFM and LFNN to enhance the model's performance in detecting code vulnerabilities (Algorithm 3). The output of each of these combinations is a query-specific few-shot set of examples that is then used to predict the label for the query. Note that while there can be other ways of combining LFM and LFNN, we believe the three combinations we explore in this work represent the most natural starting point.

Algorithm 3: Combined Methods

Input: (1) Vulnerability detection model f_{VD}; (2) Training dataset D with m labeled samples; (3) Initial few-shot set $\mathcal{S}_{init}$ with r labeled samples; (4) LFM output few-shot set sizes n_1 and n_2; (5) Boolean st indicating stacked or unstacked version; (6) Number of iterations k; (7) Option opt choosing between incorrect (I), correct (C), and gray (G); (8) Query instance x; (9) Number of nearest neighbors n_3; (10) Encoder model enc

Output: Few-shot set $\mathcal{S}$

1 **def** $method_1()$**:**
2 $\quad \mathcal{S}_{LFM} \leftarrow LFM(f_{VD}, D, \mathcal{S}_{init}, n_1, st, k, opt)$;
3 $\quad \mathcal{S}_{LFNN} \leftarrow LFNN(D, x, n_3, enc)$;
4 $\quad$ **return** $\mathcal{S}_{LFM} \cup \mathcal{S}_{LFNN}$;
5 **def** $method_2()$**:**
6 $\quad \mathcal{S}_{LFNN} \leftarrow LFNN(D, x, n_3, enc)$;
7 $\quad \mathcal{S}_{LFM} \leftarrow LFM(f_{VD}, D, \mathcal{S}_{LFNN}, n_1, st, k, opt)$;
8 $\quad$ **return** $\mathcal{S}_{LFM}$;
9 **def** $method_3()$**:**
10 $\quad \mathcal{S}_{LFM} \leftarrow LFM(f_{VD}, D, \mathcal{S}_{init}, n_2, st, k, opt)$;
11 $\quad \mathcal{S}_{LFNN} \leftarrow LFNN(D, x, n_3, enc)$;
12 $\quad$ **return** $LFM(f_{VD}, \mathcal{S}_{LFNN}, \mathcal{S}_{LFM}, n_1, st, k, opt)$

Method 1 In this method (lines 1-4), we combine the few-shot set from the LFM with the nearest neighbors of the query instance x computed by LFNN. This method is the most straightforward and cost-effective compared to the subsequent two approaches. We begin by constructing a few-shot set $\mathcal{S}_{LFM}$ with a total of n_1 samples. Note that this set is agnostic of the query x, so it just needs to be computed once for all the queries. Next, for each query instance x, we generate a unique few-shot set $\mathcal{S}_{LFNN}$ with the n_3 nearest neighbors of x. The final few-shot set is obtained by taking the union of the general few-shot set $\mathcal{S}_{LFM}$ with the nearest neighbors $\mathcal{S}_{LFNN}$ of x. Typically, in practice, the final few-shot set is composed of an equal number of samples from both the sources.

Method 2 In this method (lines 5-8), we use the nearest neighbors of the query instance x as initial few-shot examples $\mathcal{S}_{init}$ for LFM. Our intuition is that compared to using no initial few-shot examples or using random examples with LFM, the use of nearest neighbors provides the model with starting knowledge that is closely related to the query instance x. As a result, the few-shot

set constructed by LFM is specifically tailored to the program x under query and therefore, can be more effective at improving the vulnerability detection capabilities of the model. Note that, in contrast to Method 1, we alter the order of applying the two algorithms such that both the calls (lines 6 and 7) generate distinct few-shot sets tailored for each query instance x. In other words, we are not able to reuse any computation across the different queries. Although this approach is more resource-intensive, we hypothesize that this customized few-shot set could enhance model performance.

Method 3 In contrast to methods 1 and 2 that both invoke LFM and LFNN just once, method 3 (lines 9-12) invokes LFM twice. This method first invokes LFM (line 10) in a manner similar to method 1, generating a few-shot set $\mathcal{S}_{LFM}$ of size n_2. This first call to LFM is query-agnostic and therefore, only needs to be made once. Next, the method invokes LFNN (line 11), again in a manner similar to method 1 and generates a set of size n_3. As usual, the call to LFNN is query-specific and needs to be repeated for each query. Next, and unlike the other methods, LFM is invoked a second time. For this invocation, instead of using D as the dataset, the few-shot set $\mathcal{S}_{LFNN}$ computed by LFNN is used as the dataset. This enables inclusion of only those examples in the final few-shot set that are most similar to the query while also accounting for the model correctness on these examples. Moreover, the second call to LFM uses the few-shot set $\mathcal{S}_{LFM}$ computed on line 10 as the initial set. The intuition here is that initializing LFM with these examples can make LFM aware of the examples on which the model makes a mistake (or is correct, depending on the opt parameter) and thus, enable LFM to pick more effective examples for the final few-shot set. Note that the second call to LFM is also query-specific.

2.5 Experiments

In this section, we report on our experiments with the proposed methods, using open source models. We aim to answer the following research questions.

Research Questions

1. How do the proposed algorithms compare individually with baselines (zero-shot and few-shot settings) in helping large language models find vulnerabilities in code?

2. How do different strategies for combining LFM and LFNN influence the overall performance of the model and how do they compare to using either strategy in isolation?

3. Are the performance improvements introduced by LFM and LFNN consistent across different large language models, or are they model-specific?

4. Does the programming language or other linguistic characteristics of the dataset influence the effectiveness of LFM, LFNN, and their combinations?

Datasets and Models For datasets, we consider established benchmarks such as PrimeVul, DiverseVul, SVEN [12, 67, 68]. These are well-curated datasets, including pairs of code samples (vulnerable vs. non-vulnerable). We experiment with adding both vulnerable and non-vulnerable few-shot examples to better gauge the performance of the LLMs on the vulnerability detection task. We also leverage recent work on vulnerabilities in JavaScript programs [20, 21] and obtained a copy of the dataset generated by their tools directly from the authors. We refer to this dataset as NodeMedic.

PrimeVul consists of a training set with 7578 samples, comprising 3789 pairs, and a test set with 870 samples. It has 112 unique CWEs. To mitigate computational overhead, the test set was downsampled to a representative subset of 200 examples. This sampling process was conducted based on the intersection of Common Weakness Enumerations (CWEs) found in both the training and test data. The final sampled test dataset is balanced, containing 100 vulnerable and non-vulnerable pairs, and 58 unique CWEs.

DiverseVul comprises 330492 unpaired samples, and it has 150 unique CWEs. To ensure computational tractability, the dataset was first partitioned into a primary training and a primary test set, following an 80:20 ratio which were unbalanced. Subsequently, based on the intersection of CWEs present in both splits, a final balanced sample was created. This resulted in a training set

of 200 examples and a test set of 300 examples. Both of these sampled sets have 114 unique and common CWEs.

SVEN comprises 1440 training and 166 validation samples. This dataset was partitioned into two distinct subsets based on the programming language of the functions. The C/C++ subset, designated SVENC, consists of 756 training and 90 validation samples. The Python subset, SVENP, is composed of 684 training and 76 validation samples. SVENC and SVENP have 7 and 4 unique CWEs, respectively (both train and validation set).

NodeMedic dataset was provided by the authors of NodeMedic-FINE [21], a dynamic analysis tool that detects taint flows from package APIs to dangerous sinks that may enable arbitrary command injection or code execution. The dataset is divided into 1,506 training and 189 test samples, each corresponding to a Node.js package with potentially vulnerable dataflows reported by the tool. All reports are either automatically confirmed by NodeMedic-FINE [21] or manually verified by its authors.

To facilitate semantic code retrieval, we employed a specialized encoder model from the Salesforce SFR family: "Salesforce/SFR-Embedding-Code-400M_R" [69]. These vector embeddings are used in the nearest neighbor search, which utilizes cosine distance to identify the closest matches for any given query.

We conducted experiments on two open-source models, "Qwen-2.5-Coder-7B-Instruct" [70] and "gemma-3-4b-it" [71], as well as the closed-source GPT-5-mini (the *gpt-5-mini-2025-08-07* snapshot), to assess the generalizability of our proposed techniques. The Qwen model was selected for its strong, well-documented proficiency on coding tasks and very large context window, while the Gemma model was chosen for its computational efficiency and competitive performance given its parameter count. GPT-5-mini serves as a high-quality closed-source baseline to contextualize the behavior of the open models. The combination of these models allows evaluation across differing trade-offs of capability and resource requirements, while staying within our available computational budget.

Experiment Setup For our baselines, we consider a *zero-shot* setting, i.e., no examples in context, and a *random few-shot* setting where twenty vulnerable and non-vulnerable examples are drawn at random, from our train datasets. To address RQ1, we create several variants of the LFM algorithm with different parameter settings. However, since the performance differences are minimal, we only report the results for the following configurations: $n = 20, st =$ TRUE, $k = 1, opt = I$ and an initial few-shot set $\mathcal{S}_{init}$ with an empty set. We run the LFNN algorithm with parameter settings $n = 20$. To answer RQ2, we run the combined methods with the following parameter settings (for all three combinations, the initial few-shot set $\mathcal{S}_{init}$ has 5 examples that are randomly drawn from the train dataset):

- Combined Method 1: $n_1 = 10, st =$ TRUE, $k = 1, opt = I, n_3 = 10$
- Combined Method 2: $n_1 = 20, st =$ TRUE, $k = 1, opt = I, n_3 = 5$
- Combined Method 3: $n_1 = 10, n_2 = 15, st =$ TRUE, $k = 1, opt = I, n_3 = 20$

Prompt Used for Vulnerability Detection The prompt shown here is used when running LFM as well as for evaluating the vulnerability detection capabilities of the LLM for the experiments reported in Section 2.5. To instruct the model clearly, we provided a concise and explicit system instruction, guiding the model to behave strictly as a security expert and to output responses in a standardized format. The exact system-level instruction used to prime the LLM is defined as follows:

```
You are a security expert that is good at static program
analysis.
First, you will be given some examples of vulnerable and
non-vulnerable codes indicated through Yes and No. There can be
no examples too.You will be given a piece of code. Your task is
to analyze whether
it contains a security vulnerability.
```

```
Please only reply with one of the following options:
(1) YES: A security vulnerability detected.
(2) NO: No security vulnerability.

Only reply with one of the options above. Do not include any
further information.
```

For each code snippet x, we first construct a few-shot set $\mathcal{S}$ containing representative examples of code snippets, each annotated as either *vulnerable* or *non-vulnerable*. These labeled examples serve as context to guide the model's decision by explicitly illustrating the desired behavior. The few-shot prompt we use is as follows (`System, User, Assistant` refer to the roles used for prompting the model):

```
System:
  (System Instruction from above)

User:
  Code: <Example 1: Vulnerable code snippet>
  Answer:

Assistant:
  YES

User:
  Code: <Example 2: Non-vulnerable code snippet>
  Answer:

Assistant:
  NO
```

```
... (remaining few-shot examples)

User:
  Code: {code}
  Answer:
```

In our experiments, all dataset sampling is performed with a fixed random seed to ensure reproducibility. For the Qwen and Gemma models, deterministic algorithms are enabled, and all random number generators in Python and PyTorch are explicitly seeded to ensure deterministic behavior under our hardware and software setup. For GPT-5-Mini, we set the API seed parameter to a fixed integer to encourage deterministic generation. However, as documented by OpenAI[1], setting this seed does not guarantee consistent outputs because changes to the backend system, such as model version updates, can still affect the generations.

Results We present a detailed analysis of the experimental results from our evaluation of three LLMs from the Gemma, GPT, and Qwen families across the four datasets (Results are presented separately for SVENC and SVENP). All findings are summarized in Table 2.1, which reports the mean performance metrics over five independent runs for the Gemma and Qwen models using five different random seeds, along with the corresponding standard deviations. For the GPT model, we report the results from a single run, and standard deviation is therefore not applicable. Our analysis is structured on a per-dataset basis to highlight the varying effectiveness of each few-shot selection approach under different data distributions and model capabilities.

On the **NodeMedic** dataset, LFM and LFNN demonstrate substantial performance gains over the zero-shot (ZS) and random few-shot (R-FS) baselines across all models. For the Gemma model, LFNN achieves the highest F1-score of 0.854, a significant improvement from the ZS baseline of 0.574. The LFM approach pushed the model to a perfect recall of 1.000 and achieved an F1-score

[1] https://cookbook.openai.com/examples/reproducible_outputs_with_the_seed_parameter

Table 2.1: Results for all approaches (ZS = zero-shot, R-FS = random few-shot, LFM = Learn-from-Mistakes, LFNN = Learn-from-Nearest-Neighbors, CM = Combined Method) across three models. Each cell reports the mean and standard deviation over five runs for Gemma and Qwen; GPT results are based on a single run and therefore have no standard deviation. A dash ('–') indicates that the metric is undefined in at least one of the runs due to division by zero.

Dataset	Approach	Gemma-3-4b-it				GPT-5-mini				Qwen-2.5-Coder			
		Acc	Prec	Recall	F1	Acc	Prec	Recall	F1	Acc	Prec	Recall	F1
DiverseVul	ZS	0.619 (±0.003)	0.770 (±0.009)	0.340 (±0.000)	0.472 (±0.002)	0.593	0.588	0.627	0.607	0.497 (±0.000)	0.000 (±0.000)	0.000 (±0.000)	0.000 (±0.000)
	R-FS	0.545 (±0.053)	0.571 (±0.081)	0.467 (±0.210)	0.480 (±0.132)	0.560	0.598	0.367	0.455	0.599 (±0.045)	0.741 (±0.027)	0.303 (±0.139)	0.411 (±0.150)
	LFM	0.525 (±0.003)	0.840 (±0.056)	0.061 (±0.005)	0.114 (±0.009)	0.617	0.647	0.513	0.573	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)
	LFNN	0.512 (±0.004)	0.511 (±0.004)	0.537 (±0.009)	0.524 (±0.006)	0.590	0.624	0.453	0.525	0.659 (±0.009)	0.632 (±0.007)	0.759 (±0.014)	0.690 (±0.009)
	CM 1	0.548 (±0.020)	0.530 (±0.015)	0.868 (±0.035)	0.658 (±0.007)	0.587	0.610	0.480	0.537	0.638 (±0.008)	0.606 (±0.011)	0.792 (±0.039)	0.686 (±0.011)
	CM 2	0.515 (±0.002)	0.508 (±0.001)	0.973 (±0.000)	0.667 (±0.001)	0.560	0.573	0.473	0.518	0.597 (±0.006)	0.557 (±0.003)	0.949 (±0.008)	0.702 (±0.004)
	CM 3	0.499 (±0.002)	0.499 (±0.001)	0.985 (±0.007)	0.663 (±0.002)	0.570	0.585	0.480	0.527	0.531 (±0.021)	0.517 (±0.012)	0.988 (±0.015)	0.678 (±0.007)
NodeMedic	ZS	0.506 (±0.003)	0.765 (±0.005)	0.460 (±0.000)	0.574 (±0.001)	0.757	0.786	0.912	0.845	0.375 (±0.009)	0.870 (±0.049)	0.162 (±0.005)	0.273 (±0.008)
	R-FS	0.687 (±0.060)	0.726 (±0.003)	0.914 (±0.136)	0.804 (±0.060)	0.794	0.845	0.876	0.860	0.632 (±0.060)	0.748 (±0.020)	0.750 (±0.163)	0.739 (±0.071)
	LFM	0.725 (±0.000)	0.725 (±0.000)	1.000 (±0.000)	0.840 (±0.000)	0.767	0.789	0.927	0.852	0.720 (±0.000)	0.723 (±0.000)	0.993 (±0.000)	0.837 (±0.000)
	LFNN	0.758 (±0.004)	0.759 (±0.003)	0.975 (±0.004)	0.854 (±0.002)	0.788	0.849	0.861	0.855	0.713 (±0.011)	0.768 (±0.004)	0.866 (±0.014)	0.814 (±0.008)
	CM 1	0.701 (±0.005)	0.755 (±0.009)	0.870 (±0.018)	0.808 (±0.004)	0.788	0.839	0.876	0.857	0.751 (±0.008)	0.776 (±0.004)	0.923 (±0.020)	0.843 (±0.007)
	CM 2	0.565 (±0.008)	0.719 (±0.005)	0.657 (±0.008)	0.686 (±0.006)	0.794	0.840	0.883	0.861	0.735 (±0.006)	0.748 (±0.004)	0.958 (±0.003)	0.840 (±0.003)
	CM 3	0.722 (±0.007)	0.728 (±0.003)	0.984 (±0.014)	0.837 (±0.005)	0.794	0.846	0.883	0.864	0.720 (±0.000)	0.724 (±0.001)	0.991 (±0.003)	0.837 (±0.000)
PrimeVul	ZS	0.599 (±0.044)	0.723 (±0.106)	0.358 (±0.036)	0.472 (±0.003)	0.535	0.524	0.750	0.617	0.500 (±0.000)	–	0.002 (±0.004)	0.004 (±0.008)
	R-FS	0.522 (±0.041)	0.553 (±0.086)	0.373 (±0.148)	0.422 (±0.101)	0.560	0.556	0.600	0.577	0.503 (±0.010)	0.528 (±0.058)	0.232 (±0.105)	0.304 (±0.094)
	LFM	0.516 (±0.013)	0.726 (±0.186)	0.437 (±0.459)	0.336 (±0.270)	0.550	0.557	0.490	0.521	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)
	LFNN	0.527 (±0.030)	0.520 (±0.017)	0.648 (±0.147)	0.571 (±0.064)	0.555	0.549	0.620	0.582	0.483 (±0.009)	0.390 (±0.056)	0.060 (±0.009)	0.104 (±0.015)
	CM 1	0.543 (±0.032)	0.527 (±0.022)	0.885 (±0.040)	0.659 (±0.010)	0.575	0.577	0.600	0.588	0.473 (±0.007)	0.462 (±0.010)	0.336 (±0.092)	0.383 (±0.057)
	CM 2	0.516 (±0.001)	0.508 (±0.001)	0.964 (±0.022)	0.666 (±0.005)	0.555	0.549	0.610	0.578	0.455 (±0.012)	0.445 (±0.015)	0.370 (±0.026)	0.404 (±0.021)
	CM 3	0.500 (±0.002)	0.500 (±0.001)	0.982 (±0.017)	0.663 (±0.004)	0.555	0.546	0.650	0.594	0.499 (±0.006)	0.499 (±0.003)	0.986 (±0.015)	0.663 (±0.005)
SVENC	ZS	0.478 (±0.000)	0.474 (±0.000)	0.400 (±0.000)	0.434 (±0.000)	0.533	0.523	0.756	0.618	0.500 (±0.000)	–	0.000 (±0.000)	0.000 (±0.000)
	R-FS	0.506 (±0.010)	0.503 (±0.005)	0.778 (±0.347)	0.564 (±0.174)	0.600	0.579	0.733	0.647	0.491 (±0.013)	–	0.422 (±0.373)	0.345 (±0.274)
	LFM	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)	0.544	0.543	0.556	0.549	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)
	LFNN	0.471 (±0.009)	0.428 (±0.034)	0.187 (±0.044)	0.258 (±0.049)	0.567	0.554	0.689	0.614	0.504 (±0.009)	0.567 (±0.133)	0.040 (±0.009)	0.075 (±0.017)
	CM 1	0.520 (±0.008)	0.513 (±0.005)	0.813 (±0.027)	0.629 (±0.010)	0.578	0.566	0.667	0.612	0.520 (±0.019)	0.527 (±0.024)	0.387 (±0.074)	0.442 (±0.055)
	CM 2	0.536 (±0.004)	0.520 (±0.003)	0.947 (±0.011)	0.671 (±0.002)	0.589	0.574	0.689	0.626	0.493 (±0.005)	0.495 (±0.004)	0.636 (±0.011)	0.556 (±0.005)
	CM 3	0.500 (±0.000)	0.500 (±0.000)	0.938 (±0.029)	0.652 (±0.007)	0.611	0.596	0.689	0.639	0.496 (±0.005)	0.498 (±0.003)	0.982 (±0.009)	0.661 (±0.004)
SVENP	ZS	0.587 (±0.006)	0.597 (±0.007)	0.537 (±0.013)	0.565 (±0.009)	0.763	0.727	0.842	0.780	0.705 (±0.011)	0.965 (±0.028)	0.426 (±0.020)	0.591 (±0.019)
	R-FS	0.558 (±0.038)	0.561 (±0.044)	0.632 (±0.117)	0.584 (±0.035)	0.789	0.806	0.763	0.784	0.616 (±0.027)	0.707 (±0.071)	0.432 (±0.144)	0.514 (±0.102)
	LFM	0.500 (±0.000)	0.500 (±0.000)	0.905 (±0.013)	0.644 (±0.003)	0.776	0.744	0.842	0.790	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)
	LFNN	0.634 (±0.005)	0.641 (±0.004)	0.611 (±0.011)	0.625 (±0.007)	0.895	0.941	0.842	0.889	0.792 (±0.005)	0.845 (±0.002)	0.716 (±0.011)	0.775 (±0.007)
	CM 1	0.553 (±0.012)	0.546 (±0.009)	0.616 (±0.043)	0.579 (±0.023)	0.882	0.892	0.868	0.880	0.787 (±0.023)	0.793 (±0.034)	0.779 (±0.021)	0.785 (±0.019)
	CM 2	0.632 (±0.000)	0.639 (±0.000)	0.605 (±0.000)	0.622 (±0.000)	0.895	0.917	0.868	0.892	0.697 (±0.019)	0.660 (±0.016)	0.816 (±0.029)	0.729 (±0.018)
	CM 3	0.521 (±0.013)	0.512 (±0.008)	0.942 (±0.045)	0.663 (±0.009)	0.803	0.795	0.816	0.805	0.600 (±0.018)	0.558 (±0.012)	0.963 (±0.027)	0.707 (±0.010)

of 0.840. The Qwen model shows a similar trend, with LFM and LFNN improving the F1-score to 0.837 and 0.814, respectively. GPT-5-mini, a more powerful model, exhibits strong performance even with the R-FS baseline (F1-score of 0.860), but our adaptive methods still provide a slight edge. Overall, there was a good balance between accuracy and F1-score with combined method 1 and 3 , and LFNN (accuracy: 0.701, 0.722, 0.758 respectively).

The **SVENP** dataset also yields strong results, but with a key difference: the ZS baseline is notably more effective here, especially for Qwen (0.965 Precision) and GPT-5-mini (0.780 F1-score). This indicates that the patterns in SVENP align well with the models' pre-trained knowledge or maybe the models are in general better at analyzing python source code. Despite the strong baseline, LFNN improves the performance by achieving top-tier F1-scores of 0.889 (GPT) and 0.775 (Qwen). For the Gemma model, LFNN boosts the F1-score from 0.565 (ZS) to 0.625, and accuracy from 0.587 (ZS) to 0.634. A critical observation is the behavior of LFM with the Qwen model; it again defaults to predicting the positive class for all instances (1.000 Recall). This highlights LFM's tendency to act as a powerful bias amplifier, which is effective when correcting false negatives but can be overly simplistic and increase false positives. The Combined Methods (CM), particularly on GPT-5-mini, achieve the highest overall F1-scores (e.g., 0.892 for CM 2), demonstrating that integrating both mistake-based and similarity-based signals is optimal when the baseline performance is already high.

The **DiverseVul** dataset presents a more complex challenge. Here, the ZS baseline for Qwen completely fails, predicting the negative class for all samples and resulting in an F1-score of 0.000. In contrast, the Gemma ZS baseline is more reasonable, with an F1-score of 0.472. For the Qwen model, LFNN is the most effective individual strategy, raising the F1-score to 0.690. However, the most compelling finding on this dataset comes from the Gemma model. While LFM performs poorly (0.114 F1-score) and LFNN offers only a modest improvement (0.524 F1-score), the Combined Methods deliver the best performance. CM 1, CM 2, and CM 3 achieve F1-scores of 0.658, 0.667, and 0.663, with corresponding accuracies of 0.548, 0.515, and 0.499, respectively. The ro-

bustness of the Combined Methods indicates that a blended approach is necessary to navigate the diverse patterns present in the data.

The **PrimeVul** and **SVENC** datasets underscore the importance of F1-score over accuracy. On both datasets, the Qwen model's ZS baseline fails by uniformly predicting the negative class (0.000 F1-score), while the LFM approach predictably does the opposite, classifying all samples as positive (0.667 F1-score). In both cases, the accuracy is a misleading 0.500, masking these divergent failure modes. A crucial observation from these two datasets is the significant degradation of the LFNN method. For Qwen, LFNN yields very low F1-scores of 0.104 on PrimeVul and 0.075 on SVENC. This is a stark contrast to its success on NodeMedic and SVENP. This failure implies that for these datasets, the nearest neighbors examples are not helpful. Once again, the Combined Methods demonstrate greater resilience, particularly for the Gemma model. On SVENC, CM 2 elevates the F1-score to 0.671, the highest for that model. This pattern reinforces the conclusion that when simpler adaptive heuristics like LFNN fail, a more robust, multi-faceted example selection strategy is required to achieve better performance.

Finally, one interesting insight was hard examples (LFM) in context try to bias the model towards greater recall and lower precision. As CM3 applies LFM on LFNN examples keeping first LFM's result as context, it also keeps that trend mostly.

Different Variants of Learn-from-Mistakes (LFM) To explore the impact of different LFM variants, we run LFM with the following different parameter settings:

- S-1×: $n = 20, st = \text{TRUE}, k = 1, opt = I$
- U-1×: $n = 20, st = \text{FALSE}, k = 1, opt = I$
- U-m× (inc.): $n = 20, st = \text{FALSE}, k = 5, opt = I$
- U-m× (corr.):$n = 20, st = \text{FALSE}, k = 5, opt = C$
- U-m× (gray):$n = 20, st = \text{FALSE}, k = 5, opt = G$

For all the unstacked variations, the initial few-shot set $\mathcal{S}_{init}$ has 20 examples that are randomly drawn from the train dataset while $\mathcal{S}_{init}$ is the empty set for the stacked version.

Table 2.2 shows a comprehensive breakdown of performance across different LFM configurations. It is evident that the method's effectiveness is highly sensitive to its parameterization and the specific characteristics of the dataset. First, we have already stated that the default LFM method used in our main experiments, S-1x (inc.), consistently induces a strong bias towards positive predictions. This is most evident with the Qwen model, where it achieves a perfect 1.000 recall four times. However, this perfect recall on balanced datasets results in an uninformative accuracy of 0.500 and a misleadingly high F1-score of 0.667. For Gemma, this method shows highly divergent performance. It is effective on NodeMedic, achieving both high accuracy (0.725) and a high F1-score (0.840). Yet, on SVENC, it shows the biased trend: 0.667 F1-score with 0.500 accuracy. On DiverseVul and PrimeVul, it fails on both metrics, with F1-scores of 0.114 and 0.336, respectively. This confirms that it often trades all precision for recall.

Second, the comparison between stacked (S-1x) and unstacked (U-1x) methods highlights the critical impact of the initial prompt $\mathcal{S}_{init}$. For the NodeMedic and SVENC datasets, the U-1x (inc.) variant fails catastrophically. On NodeMedic, accuracy plummets to 0.273 for Gemma, and the F1-score becomes near-zero (0.003). On SVENC, it results in 0.000 recall, indicating a complete shift to negative-class bias. Conversely, on DiverseVul and PrimeVul with the Gemma model, where the stacked method failed, the unstacked version provides a better balance. On DiverseVul, it improves the F1-score from 0.114 to 0.654, though accuracy remains low at 0.502. This indicates that while the random examples help, they do not solve the accuracy/F1 trade-off, instead achieving a high F1 (0.654) through high recall (0.940) and low precision (0.501).

Third, a comparison of the unstacked, multi-iteration (U-mx) variants reveals dataset-dependent findings. For NodeMedic and SVENC, where learning from incorrect examples (opt=I) failed, learning from correct examples (opt=C) is highly effective. On NodeMedic, this U-mx (corr.) variant restores both high accuracy (0.725) and a high F1-score (0.840) for Gemma, mirroring the performance of the original stacked LFM. This demonstrates an ideal balance. On SVENC, how-

Table 2.2: Performance comparison of different LFM variations. Each cell shows the mean and standard deviation over five runs. For the LFM variation names, the following abbreviations are used: S = stacked, U = unstacked, 1× = one iteration, m× = more than one iteration, inc. = incorrect, corr. = correct, and gray = gray cases. '–' indicates metrics that are undefined due to division by zero.

Dataset	LFM Variations	Gemma-3-4b-it				Qwen-2.5-Coder			
		Accuracy	Precision	Recall	F1 Score	Accuracy	Precision	Recall	F1 Score
DiverseVul	S-1× (inc.)	0.525 (±0.003)	0.840 (±0.056)	0.061 (±0.005)	0.114 (±0.009)	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)
	U-1× (inc.)	0.502 (±0.013)	0.501 (±0.007)	0.940 (±0.025)	0.654 (±0.010)	0.502 (±0.016)	0.501 (±0.010)	0.844 (±0.063)	0.628 (±0.022)
	U-m× (inc.)	0.507 (±0.010)	0.504 (±0.005)	0.995 (±0.008)	0.669 (±0.003)	0.527 (±0.039)	0.516 (±0.025)	0.976 (±0.045)	0.674 (±0.009)
	U-m× (corr.)	0.597 (±0.053)	0.778 (±0.056)	0.295 (±0.189)	0.383 (±0.204)	0.621 (±0.065)	0.752 (±0.027)	0.365 (±0.199)	0.455 (±0.199)
	U-m× (gray)	0.513 (±0.000)	0.667 (±0.000)	0.053 (±0.000)	0.099 (±0.000)	0.598 (±0.061)	0.713 (±0.059)	0.341 (±0.226)	0.415 (±0.199)
NodeMedic	S-1× (inc.)	0.725 (±0.000)	0.725 (±0.000)	1.000 (±0.000)	0.840 (±0.000)	0.720 (±0.000)	0.723 (±0.000)	0.993 (±0.000)	0.837 (±0.000)
	U-1× (inc.)	0.273 (±0.004)	0.500 (±0.707)	0.001 (±0.003)	0.003 (±0.006)	0.274 (±0.005)	0.333 (±0.422)	0.004 (±0.006)	0.009 (±0.011)
	U-m× (inc.)	0.277 (±0.000)	-	0.000 (±0.000)	0.000 (±0.000)	0.272 (±0.003)	–	0.000 (±0.000)	0.000 (±0.000)
	U-m× (corr.)	0.725 (±0.000)	0.725 (±0.000)	1.000 (±0.000)	0.840 (±0.000)	0.724 (±0.002)	0.727 (±0.001)	0.991 (±0.003)	0.839 (±0.001)
	U-m× (gray)	0.506 (±0.003)	0.765 (±0.005)	0.460 (±0.000)	0.574 (±0.001)	0.467 (±0.100)	0.748 (±0.022)	0.412 (±0.245)	0.493 (±0.158)
PrimeVul	S-1× (inc.)	0.516 (±0.013)	0.726 (±0.186)	0.437 (±0.459)	0.336 (±0.270)	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)
	U-1× (inc.)	0.507 (±0.010)	0.529 (±0.054)	0.773 (±0.352)	0.551 (±0.212)	0.497 (±0.027)	0.489 (±0.042)	0.466 (±0.137)	0.470 (±0.092)
	U-m× (inc.)	0.496 (±0.009)	0.397 (±0.221)	0.034 (±0.022)	0.061 (±0.038)	0.501 (±0.005)	0.502 (±0.005)	0.864 (±0.219)	0.621 (±0.077)
	U-m× (corr.)	0.499 (±0.002)	0.499 (±0.001)	0.886 (±0.223)	0.626 (±0.079)	0.489 (±0.010)	0.386 (±0.090)	0.034 (±0.010)	0.062 (±0.018)
	U-m× (gray)	0.499 (±0.002)	0.496 (±0.009)	0.366 (±0.284)	0.357 (±0.202)	0.498 (±0.017)	0.529 (±0.141)	0.246 (±0.241)	0.267 (±0.198)
SVENC	S-1× (inc.)	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)
	U-1× (inc.)	0.500 (±0.000)	-	0.000 (±0.000)	0.000 (±0.000)	0.500 (±0.000)	–	0.000 (±0.000)	0.000 (±0.000)
	U-m× (inc.)	0.500 (±0.000)	-	0.000 (±0.000)	0.000 (±0.000)	0.500 (±0.000)	–	0.000 (±0.000)	0.000 (±0.000)
	U-m× (corr.)	0.504 (±0.009)	0.503 (±0.006)	0.960 (±0.069)	0.659 (±0.013)	0.493 (±0.009)	0.496 (±0.005)	0.911 (±0.056)	0.642 (±0.016)
	U-m× (gray)	0.478 (±0.000)	0.474 (±0.000)	0.400 (±0.000)	0.434 (±0.000)	0.502 (±0.011)	–	0.080 (±0.109)	0.114 (±0.142)
SVENP	S-1× (inc.)	0.500 (±0.000)	0.500 (±0.000)	0.905 (±0.013)	0.644 (±0.003)	0.500 (±0.000)	0.500 (±0.000)	1.000 (±0.000)	0.667 (±0.000)
	U-1× (inc.)	0.558 (±0.067)	0.571 (±0.083)	0.600 (±0.127)	0.573 (±0.050)	0.590 (±0.051)	0.607 (±0.074)	0.547 (±0.111)	0.567 (±0.062)
	U-m× (inc.)	0.540 (±0.033)	0.567 (±0.079)	0.447 (±0.130)	0.483 (±0.074)	0.571 (±0.054)	0.569 (±0.065)	0.721 (±0.132)	0.624 (±0.041)
	U-m× (corr.)	0.524 (±0.023)	0.522 (±0.018)	0.621 (±0.154)	0.557 (±0.063)	0.629 (±0.038)	0.712 (±0.045)	0.437 (±0.133)	0.528 (±0.110)
	U-m× (gray)	0.547 (±0.062)	0.549 (±0.064)	0.742 (±0.145)	0.619 (±0.034)	0.626 (±0.038)	0.767 (±0.120)	0.437 (±0.175)	0.519 (±0.106)

ever, the same method only restores the high F1-score (0.659) while accuracy remains low (0.504), indicating it learned to trade precision for high recall (0.960). For these same two datasets, the U-m$\times$ (gray) variant was largely ineffective. Gemma's performance regressed to F1-scores of 0.574 (NodeMedic) and 0.434 (SVENC), close to the ZS baseline.

This pattern is completely reversed on the DiverseVul dataset. Here, learning from incorrect examples (opt=I) yields a high F1-score (0.669 for Gemma) but poor accuracy (0.507). In contrast, learning from correct examples (opt=C) achieves a much better accuracy (0.597) at the cost of a poor F1-score (0.383), presenting a clear trade-off. The U-m$\times$ (gray) variant was similarly ineffective on this dataset, achieving a very low F1-score of 0.099 for Gemma. The PrimeVul dataset presents the most significant finding: a direct contradiction between the models. Gemma achieves its best unstacked F1-score (0.626) with U-mx (corr.), though its accuracy remains low (0.499). It fails with U-mx (inc.) (0.061 F1). Qwen's performance is the exact inverse, performing best with U-mx (inc.) (F1 0.621, Acc 0.501) and failing with U-mx (corr.) (F1 0.062, Acc 0.489). Here, the U-m$\times$ (gray) variant for Gemma on PrimeVul landed in the middle, with a modest F1-score of 0.357.

In summary, the LFM's behavior is not monolithic. The stacked, single-iteration approach is a predictable bias-inducer, sacrificing accuracy for recall. The unstacked, multi-iteration approaches are highly contingent on the type of examples used for learning (i.e., whether $opt = I$, $opt = C$, or $opt = G$), and no single strategy has been proven universally superior. Furthermore, our analysis highlights a persistent tension between optimizing for F1-score and accuracy. On balanced datasets, in some cases, LFM variants achieve better F1-score by inducing a strong recall bias, which simultaneously results in an accuracy score close to 0.500. To summarize, the optimal approach is highly dependent on the specific model and dataset, as demonstrated by the contradictory results on PrimeVul.

Quality of Vulnerability Detection Datasets [12] recently highlighted key challenges in existing vulnerability detection datasets, including label noise, data duplication, and data leakage. To mitigate these concerns, we incorporate the PrimeVul dataset in our experiments, as it was

carefully curated to address such issues, along with several other commonly used datasets. We also include an additional unpublished dataset (NodeMedic) to demonstrate the generality of our approach. Beyond the noisiness of current vulnerability datasets, we acknowledge that function-level vulnerability detection has inherent limitations compared to repository-level detection [2]. However, we believe that function-level datasets and detectors provide a valuable first step toward addressing vulnerability detection in broader contexts.

2.6 Conclusion

This work studies the effectiveness of different few-shot selection methods for large language models in code vulnerability detection. We evaluated several common techniques and introduced combined methods built upon them, showing that while open-source models perform worse under baseline few-shot settings, they achieve substantially greater improvements with our combined methods and can, in some cases, approach the performance of the closed-source model GPT-5-Mini.

Chapter 3

Auditing Token-Level Attributions

3.1 Introduction

Chapter 2 showed that the choice of few-shot examples meaningfully affects LLM vulnerability detection performance, and that retrieval-based selection using nearest neighbors (LFNN) consistently provides performance gains on Python and JavaScript datasets. The LFNN pipeline retrieves semantically similar training examples and injects them into the prompt before the target code. This raises a natural follow-up question: when the model makes a correct decision using this setup, which tokens it is actually attending to?

LLMs operate as black boxes. When a model says "YES, this code is vulnerable," there is no immediate way to know whether it arrived at that answer because it recognized the vulnerable logic or because of something else in the prompt. In security settings, a developer needs to trust that a tool is looking at the right place. An explanation that points to irrelevant code creates false confidence.

Explainability methods for neural networks aim to surface the reasoning behind predictions. Gradient-based attribution methods, in particular Layer Integrated Gradients (LIG), assign each input token a score. These scores tell which tokens have the most impact on the model's output.

This chapter asks a direct question: in the LFNN-based vulnerability detection pipeline, does the model prediction depend most strongly on actually vulnerable tokens? Our evaluation pipeline tries to answer this using known patch locations from real-world vulnerability datasets as ground truth. All metrics are computed only on the vulnerable code samples as we are trying to find if the model can find vulnerability reliably, and in a fixed code, there is nothing technically be called patch.

Across four datasets, roughly 40 to 52 percent of the model's top-50 attributed tokens belong to the few-shot retrieval context, and patch tokens rank on average between position 584 and 1604

in the full attribution list. These findings raise a serious concern: even when the LFNN pipeline improves detection accuracy, the model's internal attribution is not reliably grounded in the actual vulnerable code regions.

3.2 Related Work

This chapter builds on two bodies of literature: explainability methods for code and LLM-specific attribution. Vulnerability detection background and the role of few-shot learning in this context are covered in Chapter 2.

Wang et al. introduced WheaCha, which defines the "wheat" of a code model prediction as the minimal set of statements that are both sufficient (alone preserve the prediction) and necessary (their removal changes it) [72]. They tested Integrated Gradients and SHAP on smaller code classifiers such as code2vec and CodeBERT.

Ganz et al. proposed using directed fuzzing to create local ground truth around code regions highlighted by explanation methods [73]. A directed fuzzer generates malformed inputs steered toward the lines an explanation method highlights. If those lines are genuinely close to the real vulnerability, the fuzzer finds a crash relatively quickly. Vitale et al. proposed FeatureSHAP, which groups tokens into semantically meaningful features such as code blocks, docstrings, and method signatures before computing Shapley values [74]. FeatureSHAP operates on code generation tasks and requires only black-box access.

Huang and Chan proposed Grad-ELLM, a gradient and attention combined attribution method for decoder-only LLMs, and introduced the π-Soft-NC/NS faithfulness metrics to address the problem of comparing attribution methods whose score distributions have different means [75]. Their work is representative of recent interest in making attribution more principled for generative models.

3.3 Methodology

3.3.1 Task Formulation

The model is prompted with the same system message, user message, LFNN algorithm from Chapter 2. After the model generates its answer, we apply LIG to score every input token in the prompt according to its contribution to that generated answer token.

3.3.2 Integrated Gradients and Captum

The attribution method at the core is Integrated Gradients, introduced by Sundararajan et al. The method computes feature attributions by accumulating gradients along a straight-line path from a neutral baseline to the actual input in the embedding space. A baseline is a zero vector representing an input that carries no information. The method samples gradients at many points along the path from that zero vector to the actual token embedding and averages them. This average gradient, multiplied by the difference between the input and the baseline, gives the attribution score for each token [76].

We use the Layer Integrated Gradients (LIG) variant. The key difference between IG and LIG is where the integration takes place. Standard IG operates in the raw input space, which works for continuous inputs such as image pixels. For text, the raw input is a discrete token identifier, and there is no meaningful way to interpolate between two different token identifiers. LIG moves the integration to the embedding layer, where each token has been converted to a continuous vector, making interpolation mathematically valid [77, 78]. The implementation is provided by Captum [78], an open-source model interpretability library for PyTorch. Captum's `LLMGradientAttribution` wrapper, introduced in Captum v0.7 to support attribution over autoregressive generation loops [79], manages the generation loop and produces one attribution score per input token.

3.3.3 Ground Truth

For PyVul (details can be found in Section 3.4.1) and SVEN, which use paired vulnerable/fixed code, we derive ground truth from character-level diffs between the vulnerable and fixed versions using Python's `difflib` SequenceMatcher [80]. SequenceMatcher compares two strings and identifies the exact character positions where they differ. The changed spans, each a (start, end) character index pair, define the patch. These character spans are then mapped to token indices using the tokenizer's offset mapping, which records the exact start and end character position of each token in the prompt string.

For NodeMedic, vulnerability locations are provided as line and column coordinates by NodeTaint. NodeTaint is a taint analysis tool for Node.js packages that tracks how untrusted user input flows through a program to dangerous function calls such as `child_process.exec()` [20]. The location fields in the dataset record precisely where tainted data reaches a dangerous operation.

3.3.4 Metrics

We compute six metrics per sample, described below.

Unfiltered Precision@K and Recall@K measure what fraction of the top-K attributed tokens are patch tokens (precision) and what fraction of all patch tokens appear in the top-K (recall). **Few-Shot Contribution@K** measures what fraction of the top-K attributed tokens belong to the injected few-shot examples. This metric directly quantifies how much of the model's attention is being absorbed by the LFNN retrieval context rather than the target code.

Local Precision@K and Local Recall@K restrict the candidate pool to only the target code tokens, excluding few-shot and system tokens, before selecting the top-K. This gives a view of localization within the code being analyzed, independent of the few-shot distraction.

Dynamic Precision (R-Precision) sets K equal to the number of patch tokens, measuring precision at the scale of the ground truth.

Mean Rank of Patch (MRP) reports the average position of patch tokens in the full attribution ranking. A lower number is better. A value of 500 means on average a patch token ranks 500th out of all tokens.

3.4 Experiments and Results

3.4.1 Datasets

We evaluate on three datasets spanning two programming languages, Python and JavaScript. PyVul, a large-scale dataset covering C/C++, Python, and JavaScript/TypeScript [81], provides our two PyVul-derived sets. From its Python subset, a paired collection of 2,948 vulnerable/fixed function samples (1,474 pairs), we reserve 20% (590 samples) for evaluation, with the remaining 80% serving as the retrieval pool from which few-shot examples are drawn. The JavaScript/TypeScript subset (PyVul JS/TS) is considerably smaller, comprising 61 pairs (122 samples); we split it 40:60 into training and test sets, yielding a 37 pairs (74 samples) test set used for evaluation, with the 40% training split (24 pairs, 48 samples) serving as the retrieval pool for few-shot example selection; consistent with the PyVul Python setup. SVEN (evaluated samples=78) and NodeMedic (evaluated samples=261) are covered already in Chapter 2.

3.4.2 Results

The Qwen2.5-Coder-7B-Instruct model achieved an accuracy of 61.5% on PyVul Python (Precision: 64.9%, Recall: 50.2%, F1: 56.6%) and 58.1% on PyVul JS/TS (Precision: 68.8%, Recall: 29.7%, F1: 41.5%) with the LFNN strategy. For NodeMedic, three samples could not be processed due to CUDA out-of-memory errors caused by exceptionally long input sequences; all remaining 258 samples were evaluated successfully.

Table 3.1 contextualizes the Mean Rank of Patch (MRP) values in Table 3.2 and Table 3.3: MRP is a rank within the full token sequence, so the size of each region determines the scale against which patch token ranks should be read.

We have chosen the value of K to be 50. As a fraction of the total sequence, K=50 corresponds to roughly the top 2–4% of all tokens across datasets (2.8% for PyVul Python, 4.2% for SVEN Python, 2.3% for PyVul JS/TS, 1.9% for NodeMedic). For a reliable LLM, we expect to find most of the patches within at least the top 5% in the ranked list of tokens. Additionally, it approximates the mean patch token count across the majority of datasets.

Table 3.1: Token count statistics by prompt region and dataset for vulnerable samples only, reported as mean ± standard deviation.

Region	PyVul Python ($N = 295$)	SVEN Python ($N = 38$)	PyVul JS/TS ($N = 37$)	NodeMedic ($N = 132$)
Patch Tokens	18.25 ± 45.09	12.39 ± 15.27	31.38 ± 84.18	62.48 ± 79.23
Target Code Tokens	291.77 ± 310.06	176.13 ± 134.93	274.65 ± 294.51	660.84 ± 717.68
Few-Shot Tokens	1373.08 ± 896.69	894.39 ± 656.63	1810.95 ± 1179.87	1908.30 ± 990.94
Total Tokens	1787.26 ± 1030.22	1192.79 ± 739.71	2207.59 ± 1294.64	2691.31 ± 1137.87

Table 3.2 and Table 3.3 show attribution analysis results for correct and incorrect predictions respectively, with $K = 50$ across all four datasets.

Table 3.2: Attribution analysis for correct predictions where the model answered YES on vulnerable code ($K = 50$).

Metric	PyVul Python ($n = 148$)	SVEN Python ($n = 26$)	PyVul JS/TS ($n = 11$)	NodeMedic ($n = 89$)
Unfiltered Prec@K	0.88 ± 1.80	0.46 ± 1.15	0.55 ± 1.23	1.23 ± 2.01
Unfiltered Rec@K	5.38 ± 15.63	1.39 ± 3.45	1.51 ± 4.40	1.87 ± 4.04
FS Contribution@K	47.55 ± 8.73	48.31 ± 7.08	52.55 ± 6.16	43.78 ± 12.33
Local Prec@K	10.56 ± 17.09	10.80 ± 10.00	8.14 ± 9.59	16.34 ± 14.96
Local Rec@K	45.01 ± 41.14	51.74 ± 36.02	64.29 ± 36.32	21.56 ± 22.52
Dynamic Prec	0.54 ± 1.93	0.16 ± 0.58	0.93 ± 2.26	1.27 ± 2.54
MRP	787.4 ± 654.4	584.1 ± 434.3	765.0 ± 636.2	1299.5 ± 689.9

All percentage metrics reported as %; MRP reported as token rank.

Table 3.3: Attribution analysis for incorrect predictions where the model answered NO on vulnerable code ($K = 50$).

Metric	PyVul Python ($n = 147$)	SVEN Python ($n = 12$)	PyVul JS/TS ($n = 26$)	NodeMedic ($n = 43$)
Unfiltered Prec@K	1.50 ± 3.51	1.64 ± 2.23	1.46 ± 2.58	1.44 ± 2.49
Unfiltered Rec@K	6.26 ± 19.50	10.98 ± 21.49	10.29 ± 22.00	1.69 ± 3.96
FS Contribution@K	46.33 ± 11.73	52.55 ± 6.16	48.00 ± 10.23	38.00 ± 17.40
Local Prec@K	10.37 ± 18.06	8.91 ± 10.77	18.04 ± 29.96	12.03 ± 14.94
Local Rec@K	35.20 ± 38.55	35.87 ± 28.15	43.16 ± 40.75	11.88 ± 19.01
Dynamic Prec	1.33 ± 3.75	0.19 ± 0.59	1.20 ± 3.80	1.61 ± 2.78
MRP	1003.7 ± 705.7	713.5 ± 431.6	1219.8 ± 818.9	1604.7 ± 538.5

All percentage metrics reported as %; MRP reported as token rank.

The first and most striking pattern is the Few-Shot Contribution@K. In every dataset and every condition, roughly 40 to 52 percent of the model's top-50 attributed tokens belong to the injected LFNN examples rather than the target code. This is the same retrieval context that Chapter 2 showed improves detection performance. The model is attending heavily to those examples when forming its answer, which is plausible from a gradient perspective but unhelpful for a security practitioner who needs to know which tokens in the target code are vulnerable.

Unfiltered Precision@K is extremely low across all settings, ranging from 0.46% to 1.64%. Dynamic Precision, which adapts K to the actual number of patch tokens, confirms this with values between 0.16% and 1.61%.

Local Precision@K, which removes the few-shot distraction by restricting the ranked list to only target code tokens, shows values of around 8 to 18 percent. This means that within the target code itself, the model does show some localization signal. However, the high standard deviations, often matching or exceeding the mean, indicate this signal is highly inconsistent across samples.

The Mean Rank of Patch is the most intuitive metric. As Table 3.1 shows, the mean total token count for vulnerable samples ranges from approximately 1193 (SVEN Python) to 2691 (NodeMedic). Patch tokens are a small fraction of this: on average, they make up just 1.0% of the total input on both SVEN Python and PyVul Python, 1.4% on PyVul JS/TS, and 2.3% on NodeMedic. A reliable LLM should rank most of these patch tokens in the top 1–2% of the

ranked list, that is, within the first 12–54 positions. Instead, on PyVul Python, the average patch token ranks 787th on correct predictions and 1004th on incorrect ones, out of a mean sequence of roughly 1787 tokens — placing patch tokens near the 44th and 56th percentile of the ranked list respectively. On NodeMedic, the figures are rank 1299 and 1605 out of a mean sequence of 2691, corresponding to the 48th and 60th percentile. Patch tokens are not near the top of the attribution list; they are buried near the middle.

One consistent pattern is that MRP is lower for correct predictions than for incorrect ones across all datasets. This suggests a weak but real signal: the model does attribute slightly more to the patch when it answers correctly. But the signal is too weak to be actionable.

3.5 Discussion

3.5.1 What the Results Mean

The core finding is clear: LIG attribution for the LFNN vulnerability detection setup is revealing that the models are not always reliable. This has several possible explanations that are not mutually exclusive.

First, the LFNN few-shot examples may genuinely be doing a lot of the work. Chapter 2 showed that LFNN improves detection performance specifically because the retrieved examples provide relevant context. It is plausible that the gradient signal is large on those tokens precisely because they are crucial for the model knowing how to answer.

Second, LIG may not be the right tool for this kind of long, structured prompt. WheaCha [72] showed that both IG and SHAP fail to identify the defining features of code model predictions with high accuracy. The decoder-only architecture of Qwen2.5-Coder further complicates attribution because each generated token attends to all previous tokens via the causal mask, and gradients flow back through all of them.

Third, the vulnerability detection task may be asymmetric in a way that makes attribution difficult. Recognizing a vulnerability often requires detecting the absence of a safety check or the presence of a very specific pattern. The model may answer correctly by recognizing global context

cues from the few-shot examples rather than attending to the specific token that makes the code vulnerable.

3.5.2 Comparison with Prior Work

Pintore et al. [82] showed that vulnerability detection models can make correct predictions based on non-vulnerable lines, suggesting that correct classification does not imply correct reasoning. Our findings are consistent with this conclusion: the model answers YES on vulnerable code, yet gradient-based attributions rank the actual patch tokens near the middle of the sequence.

The LLMVulExp paper [83] takes the position that LLMs can explain vulnerabilities through generated natural language. Our results suggest a caution for that approach: if the model's internal attention is not on the vulnerable tokens, a generated explanation may be fluent and confident but not grounded.

The survery on explainable AI (XAI) for software engineering by Cao et al. [84] identifies the lack of hard ground truth as one of the main weaknesses of XAI for software engineering evaluation. Our use of diff-derived patch spans and NodeTaint location fields as token-level ground truth is a concrete step toward addressing this.

3.5.3 Limitations

The results are specific to one model (Qwen2.5-Coder-7B-Instruct), one attribution method (LIG via Captum), and four datasets. A different attribution method such as Grad-ELLM [75], or a model fine-tuned on security data as in GoodVibe [85] or LLMVulExp [83], may show better result. The LIG computation is also expensive: each sample requires a full forward and backward pass.

3.6 Conclusion

This chapter evaluated whether LLM based vulnerability detection is reliable using the LFNN few-shot pipeline from Chapter 2. Across four datasets covering Python and JavaScript, the answer

is mostly no. The top-50 attributed tokens are dominated by the injected LFNN retrieval context at roughly 40 to 52 percent. Patch tokens rank hundreds to over a thousand positions below the top of the attribution list. There is a weak signal that correct predictions have slightly better attribution than incorrect ones, but it is too small to be practically useful.

These findings reveal a gap between detection performance and reliability in the LFNN pipeline. The same retrieval context that makes the model more accurate also absorbs the bulk of its attributed attention. Future work should explore whether other detection improvement methods not just improve accuracy but also the reliability of LLMs.

Bibliography

[1] Pengyue Chen et al. Llms in software security: A survey of vulnerability detection techniques and insights. *arXiv preprint arXiv:2502.07049*, 2025.

[2] Niklas Risse, Jing Liu, and Marcel Böhme. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):388–410, 2025.

[3] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[4] Jie Lin and David Mohaisen. From large to mammoth: A comparative evaluation of large language models in vulnerability detection. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*, 2025.

[5] Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Computing Surveys*, 56(12):1–41, 2024.

[6] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we?, 2024.

[7] Tianming Zheng, Haojun Liu, Hang Xu, Xiang Chen, Ping Yi, and Yue Wu. Few-vuld: A few-shot learning framework for software vulnerability detection. *Computers & Security*, 144:103992, 2024.

[8] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[9] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–31, 2025.

[10] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1882–1894, 2025.

[11] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92:103917, 2025.

[12] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 469–481. IEEE Computer Society, 2024.

[13] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 103–114. IEEE, 2025.

[14] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

[15] Sewon Min, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Noisy channel language model prompting for few-shot text classification. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5316–5330, 2022.

[16] Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. Expertprompting: Instructing large language models to be distinguished experts. *arXiv preprint arXiv:2305.14688*, 2023.

[17] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 544–555, 2022.

[18] Filipe Marques, Mafalda Ferreira, André Nascimento, Miguel E Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. Automated exploit generation for node. js packages. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1341–1366, 2025.

[19] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[20] Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node. js vulnerabilities with provenance graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1101–1127. IEEE, 2023.

[21] Darion Cassel, Nuno Sabino, Ruben Martins, and Limin Jia. Nodemedic-fine: Automatic detection and exploit synthesis for node. js vulnerabilities.

[22] Penghui Li, Songchen Yao, Josef Sarfati Korich, Changhua Luo, Jianjia Yu, Yinzhi Cao, and Junfeng Yang. Automated static vulnerability detection via a holistic neuro-symbolic approach. *arXiv preprint arXiv:2504.16057*, 2025.

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[24] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.

[25] William Melicher, Clement Fung, Lujo Bauer, and Limin Jia. Towards a lightweight, hybrid approach for detecting dom xss vulnerabilities with machine learning. In *Proceedings of the Web Conference 2021*, WWW '21, 2021.

[26] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463. IEEE, 2015.

[27] Nadia Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. Vulnerable code detection using software metrics and machine learning. *IEEE Access*, 8:219174–219198, 2020.

[28] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[29] Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021.

[30] Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[31] Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024.

[32] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 121–133. IEEE, 2023.

[33] Anurag Swarnim Yadav and Joseph N Wilson. R+ r: Security vulnerability dataset quality is critical. In *2024 Annual Computer Security Applications Conference (ACSAC)*, pages 1047–1061. IEEE, 2024.

[34] Yikun Li, Ngoc Tan Bui, Ting Zhang, Martin Weyssow, Chengran Yang, Xin Zhou, Jinfeng Jiang, Junkai Chen, Huihui Huang, Huu Hung Nguyen, et al. Out of distribution, out of luck: How well can llms trained on vulnerability datasets detect top 25 cwe weaknesses? *arXiv preprint arXiv:2507.21817*, 2025.

[35] Rijha Safdar, Danyail Mateen, Syed Taha Ali, M Umer Ashfaq, and Wajahat Hussain. Data and context matter: Towards generalizing ai-based software vulnerability detection. *arXiv preprint arXiv:2508.16625*, 2025.

[36] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2022.

[37] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2237–2248. IEEE, 2023.

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

[39] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[40] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[41] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[42] Qwen Team. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[43] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[44] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[45] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[46] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[48] Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. *arXiv preprint arXiv:2406.03718*, 2024.

[49] David Farr, Kevin Talty, Alexandra Farr, John Stockdale, Iain Cruickshank, and Jevin West. Expert-in-the-loop systems with cross-domain and in-domain few-shot learning for software vulnerability detection. *arXiv preprint arXiv:2506.10104*, 2025.

[50] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, et al. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147*, 2024.

[51] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[52] G Ramesh, Mahammad Sahil, Shashank A Palan, Darshan Bhandary, Teli Abhishek Ashok, J Shreyas, and N Sowjanya. A review on nlp zero-shot and few-shot learning: methods and applications. *Discover Applied Sciences*, 7(9):1–22, 2025.

[53] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[54] Trapit Bansal, Rishikesh Jha, and Andrew McCallum. Learning to few-shot learn across diverse natural language classification tasks. *arXiv preprint arXiv:1911.03863*, 2019.

[55] Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:423–438, 2020.

[56] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671, 2022.

[57] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023.

[58] Antonio Sabbatella, Andrea Ponti, Ilaria Giordani, Antonio Candelieri, and Francesco Archetti. Prompt optimization in large language models. *Mathematics*, 12(6):929, 2024.

[59] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023.

[60] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2025.

[61] Wenxiao Wang, Priyatham Kattakinda, and Soheil Feizi. Maestro: Joint graph & config optimization for reliable ai agents, 2025.

[62] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In

Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 8086–8098, 2022.

[63] Qi Guo, Leiyu Wang, Yidong Wang, Wei Ye, and Shikun Zhang. What makes a good order of examples in in-context learning. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 14892–14904, 2024.

[64] Rishabh Agarwal, Avi Singh, Lei Zhang, Bernd Bohnet, Luis Rosias, Stephanie Chan, Biao Zhang, Ankesh Anand, Zaheer Abbas, Azade Nova, et al. Many-shot in-context learning. *Advances in Neural Information Processing Systems*, 37:76930–76966, 2024.

[65] Rahul Atul Bhope, Praveen Venkateswaran, KR Jayaram, Vatche Isahagian, Vinod Muthusamy, and Nalini Venkatasubramanian. Optiseq: Ordering examples on-the-fly for in-context learning. *arXiv preprint arXiv:2501.15030*, 2025.

[66] Benfeng Xu, Quan Wang, Zhendong Mao, Yajuan Lyu, Qiaoqiao She, and Yongdong Zhang. k nn prompting: Beyond-context learning with calibration-free nearest neighbor inference. *arXiv preprint arXiv:2303.13824*, 2023.

[67] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection, 2023.

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

[69] Ye Liu, Rui Meng, Shafiq Jot, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644*, 2024.

[70] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[71] Gemma Team. Gemma 3. 2025.

[72] Yu Wang, Ke Wang, and Linzhang Wang. An explanation method for models of code. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2), 2023.

[73] Tom Ganz, Philipp Rall, Martin Härterich, and Konrad Rieck. Hunting for truth: Analyzing explanation methods in learning-based vulnerability discovery. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023.

[74] Antonio Vitale et al. Toward explaining large language models in software engineering tasks. *arXiv preprint arXiv:2512.20328*, 2025.

[75] Xingyi Huang and Antoni B. Chan. Grad-ellm: Gradient-based explanations for decoder-only llms. *arXiv preprint arXiv:2601.03089*, 2026.

[76] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2017.

[77] Pramod Kaushik Mudrakarta, Ankur Taly, Mukund Sundararajan, and Kedar Dhamdhere. Did the model understand the question? In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1896–1906, 2018.

[78] Narine Kokhlikyan, Vivek Miglani, Miguel Martin, Edward Wang, Bilal Alsallakh, Jonathan Reynolds, Alexander Melnikov, Natalia Kliushkina, Carlos Araya, Siqi Yan, and Orion Reblitz-Richardson. Captum: A unified and generic model interpretability library for PyTorch. *arXiv preprint arXiv:2009.07896*, 2020.

[79] Vivek Miglani, Aobo Yang, Aram H. Markosyan, Diego Garcia-Olano, and Narine Kokhlikyan. Using captum to explain generative language models. *arXiv preprint arXiv:2312.05491*, 2023.

[80] Python Software Foundation. difflib — helpers for computing deltas. Python Standard Library Documentation, 2024. Accessed: 2025.

[81] Haowei Quan, Junjie Wang, Xinzhe Li, Terry Yue Zhuo, Xiao Chen, and Xiaoning Du. An empirical study of vulnerabilities in python packages and their detection. *arXiv preprint arXiv:2509.04260*, 2025.

[82] Marco Pintore et al. Evaluating line-level localization ability of learning-based code vulnerability detection models. *arXiv preprint arXiv:2510.11202*, 2025.

[83] Qing Mao et al. Towards explainable vulnerability detection with large language models. *arXiv preprint arXiv:2406.09701*, 2024.

[84] Sicong Cao et al. A systematic literature review on explainability for machine/deep learning-based software engineering research. *arXiv preprint arXiv:2401.14617*, 2025.

[85] Minh Thang et al. Goodvibe: Security-by-vibe for llm-based code generation. *arXiv preprint arXiv:2602.10778*, 2026.