

On Selecting Few-Shot Examples for LLM-based Code Vulnerability Detection

Md Abdul Hannan*
Colorado State University
ma.hannan@colostate.edu

Ronghao Ni*
Carnegie Mellon University
ronghaon@andrew.cmu.edu

Chi Zhang
Carnegie Mellon University
chiz5@andrew.cmu.edu

Limin Jia
Carnegie Mellon University
liminjia@andrew.cmu.edu

Ravi Mangal
Colorado State University
ravi.mangal@colostate.edu

Corina S. Pasareanu
Carnegie Mellon University
pcorina@andrew.cmu.edu

Abstract

Large language models (LLMs) have demonstrated impressive capabilities for many coding tasks, including summarization, translation, completion, and code generation. However, detecting code vulnerabilities remains a challenging task for LLMs. An effective way to improve LLM performance is in-context learning (ICL) – providing few-shot examples similar to the query, along with correct answers, can improve an LLM’s ability to generate correct solutions. However, choosing the few-shot examples appropriately is crucial to improving model performance. In this paper, we explore two criteria for choosing few-shot examples for ICL used in the code vulnerability detection task. The first criterion considers if the LLM (consistently) makes a mistake or not on a sample with the intuition that LLM performance on a sample is informative about its usefulness as a few-shot example. The other criterion considers *similarity* of the examples with the program under query and chooses few-shot examples based on the k -nearest neighbors to the given sample. We perform evaluations to determine the benefits of these criteria individually as well as under various combinations, using open-source models on multiple datasets.

Keywords

LLMs, Few-shot examples, Code vulnerability detection

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 [23, 24, 53, 57, 67]. 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 [33, 66]. However, the performance (measured in terms of precision, recall, F1 score) of

these models remains low [11, 26] 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. Our idea is based on 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. The impact of ICL on model performance is highly sensitive to the specific examples chosen and therefore selecting the appropriate few-shot examples is crucial to improving model performance [35, 42, 61]. For vulnerability detection, the few-shot set comprises of programs along with the ground-truth label indicating whether they have a vulnerability or not.

We propose new methods for choosing the example programs that comprise the few-shot set used in the code vulnerability detection task. Our first contribution is to present a pair of algorithmic primitives for choosing the few-shot examples. The first primitive, **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 primitive, **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.

Our second contribution is to propose three different methods for combining LFM and LFNN to yield few-shot sets that are both semantically similar to the program under test and provide a context

*Equal contribution.
Preprint. Under review.

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 our 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 our evaluation that include programs from a variety of languages such as C/C++, Python, and JavaScript. Our experimental results indicate that individual strategies, particularly LFNN, can significantly enhance the models’ baseline vulnerability detection capabilities. LFM introduces a strong inductive bias, consistently skewing predictions toward the positive class. However, our proposed combined methods demonstrate a more robust and balanced performance profile, effectively optimizing both accuracy and F1-score across the diverse range of models and datasets evaluated.

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 [34, 39] to dynamic testing and fuzzing [7, 8, 38], and from these traditional methods to more recent machine learning based approaches [21, 28, 31, 41].

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 [40, 43]. 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 [21, 29, 52, 65, 68], but their effectiveness is often constrained by dataset quality and the challenge of generalizing across diverse projects [10, 30, 50, 62].

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 [19, 21, 51]. 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 [32, 52, 65, 68]. Early research in this category has often focused on models like CodeBERT [16] and GraphCodeBERT [17], with more recent work exploring larger foundation models such as Qwen [3, 55, 63, 64] and LLaMA [14, 47, 56].

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 [67]. Several recent efforts reflect this trend. Du et al. [12] introduce VulLLM, a multi-task LLM framework that integrates vulnerability interpretation and data augmentation to significantly improve code vulnerability detection. Farr et al. [15] leverage few-shot prompting to enhance out-of-the-box LLMs for vulnerability detection. Du et al. [13] 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) [6] 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 [4, 6, 45, 59].

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 [25, 35, 42, 48, 61]. Liu et al. [35] retrieve examples that are semantically-similar to a test sample to formulate its corresponding prompt. Rubin et al. [48] train an efficient dense retriever to select training examples as prompts at test time. Xu et al. [61] employ in-context learning to create expert profiles that condition LLM responses. In our setting, we investigate new strategies for selecting 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 [44, 49]. Beyond manual design, recent work has begun to explore automated prompt optimization. DSPy [27] introduces a declarative framework that compiles language model pipelines into self-improving programs. GEPA [2] uses reflective natural-language feedback and evolutionary search to iteratively refine prompts. Maestro [58] 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 [1, 5, 18, 37]. 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.

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

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 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 S

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

```

1  $S_C \leftarrow D, S_I \leftarrow D$ ;
2 foreach  $idx \in \{1, \dots, k\}$  do
3    $S_{ctxt} \leftarrow S_{init}, S_{Ci} \leftarrow \emptyset, S_{Ii} \leftarrow \emptyset$ ;
4   foreach  $(x, y) \in D \setminus S_{init}$  do
5     Compute prediction  $\hat{y} = f_{VD}(S_{ctxt}; x)$ ;
6     if  $\hat{y} \neq y$  then
7        $S_{Ii} \leftarrow S_{Ii} \cup \{(x, y)\}$ ;
8       if  $st$  then  $S_{ctxt} \leftarrow S_{ctxt} \cup \{(x, y)\}$ ;
9     else
10       $S_{Ci} \leftarrow S_{Ci} \cup \{(x, y)\}$ ;
11   end
12    $S_C \leftarrow S_C \cap S_{Ci}$ ;
13    $S_I \leftarrow S_I \cap S_{Ii}$ ;
14 end
15  $S_G \leftarrow D \setminus (S_C \cup S_I)$ ;
16 if  $st$  then
17    $S \leftarrow S_{init}$ ;
18    $S_{rand} \leftarrow$ 
19     Uniformly draw  $(n - |S|)$  samples from  $S_{opt} \setminus S_{init}$ ;
20    $S \leftarrow S \cup S_{rand}$ ;
21 else
22    $S \leftarrow$  Uniformly draw  $n$  samples from  $S_{opt}$ ;
23 return  $S$ ;

```

where each example is a program with a yes/no label indicating if a vulnerability is present or not.

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 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.

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 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 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 S_C and S_I with the entire training dataset D (line 1). The S_C tracks the examples from D where the LLM is correct while 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 S_{ctxt} that will be used when querying the LLM during the linear scan. It also initializes sets S_{Ci} and 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 S_{init}). Depending on whether the prediction is correct or not, the example is added to the set S_{Ci} (line 10) or S_{Ii} (line 7), respectively. Moreover, if the boolean input st is set to TRUE, the few-shot set 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 S_C and 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 or 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 and a final version of the sets S_C and S_I have been constructed, the algorithm also constructs a set S_G of examples where the LLM is not consistently correct or incorrect (G stands for gray).

Finally, the few-shot set S to be returned by the algorithm is computed (lines 16-21). If the flag st is set to TRUE, the examples in the initial set S_{init} are included in the set S . The remaining samples in the set S are chosen from the appropriate sets S_C , S_I , and S_G (denoted by S_{opt}) as dictated by the value of the opt input (lines 18 and 21).

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 [60], 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

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

```

1  $K \leftarrow \emptyset$ ;
2 foreach  $(x_i, y_i) \in D$  do
3    $K \leftarrow K \cup (i, enc(x_i))$ ;
4 end
// Part 2: Instance-specific computation
5  $q \leftarrow enc(x)$ ,  $C \leftarrow \emptyset$ ;
6 foreach  $(i, k) \in K$  do
7    $C \leftarrow C \cup (i, cosine(k, q))$ ;
8 end
//  $Top_n^2(C)$  returns the  $n$  pairs from  $C$  with the
// largest second components
9  $\mathcal{NN}_x \leftarrow \{D[i] \mid (i, k) \in Top_n^2(C)\}$ ;
10 return  $\mathcal{NN}_x$ ;

```

query, the number n of nearest neighbors of x (i.e., the size of the few-shot set 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.

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

Algorithm 3: Combined Methods

Input: (1) Vulnerability detection model f_{VD} ; (2) Training dataset D with m labeled samples; (3) Initial few-shot set 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 S

```

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

```

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.

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 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 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 S_{LFM} with the nearest neighbors 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 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.

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 [9, 11, 20]. 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 [7, 8] 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 has 7 and 4 unique CWEs respectively (both train and validation set).

NodeMedic dataset was provided by the authors of NodeMedic-FINE [8], 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 [8] 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" [36]. 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" [22] and "gemma-3-4b-it" [54], 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: $n = 20$, $st = \text{TRUE}$, $k = 1$, $opt = I$ 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):

Table 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. Gemma and Qwen results are mean over five runs; GPT results are based on one run. 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	0.770	0.340	0.472	0.593	0.588	0.627	0.607	0.497	0.000	0.000	0.000
	R-FS	0.545	0.571	0.467	0.480	0.560	0.598	0.367	0.455	0.599	0.741	0.303	0.411
	LFM	0.525	0.840	0.061	0.114	0.617	0.647	0.513	0.573	0.500	0.500	1.000	0.667
	LFNN	0.512	0.511	0.537	0.524	0.590	0.624	0.453	0.525	0.659	0.632	0.759	0.690
	CM 1	0.548	0.530	0.868	0.658	0.587	0.610	0.480	0.537	0.638	0.606	0.792	0.686
	CM 2	0.515	0.508	0.973	0.667	0.560	0.573	0.473	0.518	0.597	0.557	0.949	0.702
	CM 3	0.499	0.499	0.985	0.663	0.570	0.585	0.480	0.527	0.531	0.517	0.988	0.678
NodeMedic	ZS	0.506	0.765	0.460	0.574	0.757	0.786	0.912	0.845	0.375	0.870	0.162	0.273
	R-FS	0.687	0.726	0.914	0.804	0.794	0.845	0.876	0.860	0.632	0.748	0.750	0.739
	LFM	0.725	0.725	1.000	0.840	0.767	0.789	0.927	0.852	0.720	0.723	0.993	0.837
	LFNN	0.758	0.759	0.975	0.854	0.788	0.849	0.861	0.855	0.713	0.768	0.866	0.814
	CM 1	0.701	0.755	0.870	0.808	0.788	0.839	0.876	0.857	0.751	0.776	0.923	0.843
	CM 2	0.565	0.719	0.657	0.686	0.794	0.840	0.883	0.861	0.735	0.748	0.958	0.840
	CM 3	0.722	0.728	0.984	0.837	0.794	0.846	0.883	0.864	0.720	0.724	0.991	0.837
PrimeVul	ZS	0.599	0.723	0.358	0.472	0.535	0.524	0.750	0.617	0.500	–	0.002	0.004
	R-FS	0.522	0.553	0.373	0.422	0.560	0.556	0.600	0.577	0.503	0.528	0.232	0.304
	LFM	0.516	0.726	0.437	0.336	0.550	0.557	0.490	0.521	0.500	0.500	1.000	0.667
	LFNN	0.527	0.520	0.648	0.571	0.555	0.549	0.620	0.582	0.483	0.390	0.060	0.104
	CM 1	0.543	0.527	0.885	0.659	0.575	0.577	0.600	0.588	0.473	0.462	0.336	0.383
	CM 2	0.516	0.508	0.964	0.666	0.555	0.549	0.610	0.578	0.455	0.445	0.370	0.404
	CM 3	0.500	0.500	0.982	0.663	0.555	0.546	0.650	0.594	0.499	0.499	0.986	0.663
SVENC	ZS	0.478	0.474	0.400	0.434	0.533	0.523	0.756	0.618	0.500	–	0.000	0.000
	R-FS	0.506	0.503	0.778	0.564	0.600	0.579	0.733	0.647	0.491	–	0.422	0.345
	LFM	0.500	0.500	1.000	0.667	0.544	0.543	0.556	0.549	0.500	0.500	1.000	0.667
	LFNN	0.471	0.428	0.187	0.258	0.567	0.554	0.689	0.614	0.504	0.567	0.040	0.075
	CM 1	0.520	0.513	0.813	0.629	0.578	0.566	0.667	0.612	0.520	0.527	0.387	0.442
	CM 2	0.536	0.520	0.947	0.671	0.589	0.574	0.689	0.626	0.493	0.495	0.636	0.556
	CM 3	0.500	0.500	0.938	0.652	0.611	0.596	0.689	0.639	0.496	0.498	0.982	0.661
SVENP	ZS	0.587	0.597	0.537	0.565	0.763	0.727	0.842	0.780	0.705	0.965	0.426	0.591
	R-FS	0.558	0.561	0.632	0.584	0.789	0.806	0.763	0.784	0.616	0.707	0.432	0.514
	LFM	0.500	0.500	0.905	0.644	0.776	0.744	0.842	0.790	0.500	0.500	1.000	0.667
	LFNN	0.634	0.641	0.611	0.625	0.895	0.941	0.842	0.889	0.792	0.845	0.716	0.775
	CM 1	0.553	0.546	0.616	0.579	0.882	0.892	0.868	0.880	0.787	0.793	0.779	0.785
	CM 2	0.632	0.639	0.605	0.622	0.895	0.917	0.868	0.892	0.697	0.660	0.816	0.729
	CM 3	0.521	0.512	0.942	0.663	0.803	0.795	0.816	0.805	0.600	0.558	0.963	0.707

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

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 OpenAI¹, 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

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

separately for SVENC and SVENP). All findings are summarized in Table 1, which reports the mean performance metrics over five independent runs for the Gemma and Qwen models using five different random seeds. The standard deviations between runs are minimal, so they are not included in the table because of space constraints. For the GPT model, we report the results from a single run. 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 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 robustness 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 keep that trend mostly.

Quality of Vulnerability Detection Datasets. Ding et al. [11] 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 [46]. However, we believe that function-level datasets and detectors provide a valuable first step toward addressing vulnerability detection in broader contexts.

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.

References

- [1] R. Agarwal, A. Singh, L. Zhang, B. Bohnet, L. Rosias, S. Chan, B. Zhang, A. Anand, Z. Abbas, A. Nova, et al. Many-shot in-context learning. *NeurIPS*, 37:76930–76966, 2024.
- [2] L. A. Agrawal, S. Tan, D. Soylu, N. Ziemis, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang, C. Potts, K. Sen, A. G. Dimakis, I. Stoica, D. Klein, M. Zaharia, and O. Khattab. Gema: Reflective prompt evolution can outperform reinforcement learning, 2025. URL <https://arxiv.org/abs/2507.19457>.
- [3] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.
- [4] T. Bansal, R. Jha, and A. McCallum. Learning to few-shot learn across diverse natural language classification tasks. *arXiv preprint arXiv:1911.03863*, 2019.
- [5] R. A. Bhope, P. Venkateswaran, K. Jayaram, V. Isahagian, V. Muthusamy, and N. Venkatasubramanian. Optiseq: Ordering examples on-the-fly for in-context learning. *arXiv preprint arXiv:2501.15030*, 2025.
- [6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *NeurIPS*, 33:1877–1901, 2020.

- [7] D. Cassel, W. T. Wong, and L. Jia. Nodemedic: End-to-end analysis of node. js vulnerabilities with provenance graphs. In *IEEE S&P*, 2023.
- [8] D. Cassel, N. Sabino, M.-C. Hsu, R. Martins, and L. Jia. Nodemedic-fine: Automatic detection and exploit synthesis for node. js vulnerabilities. In *Proceedings of NDSS'25*, volume 10, 2025.
- [9] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection, 2023. URL <https://arxiv.org/abs/2304.00409>.
- [10] R. Croft, M. A. Babar, and M. M. Kholoosi. Data quality for software vulnerability datasets. In *ICSE*, 2023.
- [11] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen. Vulnerability detection with code language models: How far are we? In *ICSE*, 2024.
- [12] X. Du, M. Wen, J. Zhu, Z. Xie, B. Ji, H. Liu, X. Shi, and H. Jin. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. *arXiv preprint arXiv:2406.03718*, 2024.
- [13] X. Du, G. Zheng, K. Wang, Y. Zou, Y. Wang, W. Deng, J. Feng, M. Liu, B. Chen, X. Peng, et al. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147*, 2024.
- [14] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [15] D. Farr, K. Talty, A. Farr, J. Stockdale, I. Cruickshank, and J. 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.
- [16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- [17] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.
- [18] Q. Guo, L. Wang, Y. Wang, W. Ye, and S. Zhang. What makes a good order of examples in in-context learning. In *ACL*, pages 14892–14904, 2024.
- [19] H. Hanif and S. Maffei. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2022.
- [20] J. He and M. Vechev. Large language models for code: Security hardening and adversarial testing. In *CCS, CCS '23*, page 1865–1879, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3623175. URL <https://doi.org/10.1145/3576915.3623175>.
- [21] D. Hin, A. Kan, H. Chen, and M. A. 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.
- [22] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.
- [23] R. A. Husein, H. Aburajouh, and C. Catal. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92: 103917, 2025.
- [24] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.
- [25] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8: 423–438, 2020.
- [26] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities. In *ICST 2025*, pages 103–114. IEEE, 2025.
- [27] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023. URL <https://arxiv.org/abs/2310.03714>.
- [28] P. Li, S. Yao, J. S. Korich, C. Luo, J. Yu, Y. Cao, and J. Yang. Automated static vulnerability detection via a holistic neuro-symbolic approach. *arXiv preprint arXiv:2504.16057*, 2025.
- [29] Y. Li, S. Wang, and T. N. Nguyen. Vulnerability detection with fine-grained interpretations. In *FSE*, pages 292–303, 2021.
- [30] Y. Li, N. T. Bui, T. Zhang, M. Weyssow, C. Yang, X. Zhou, J. Jiang, J. Chen, H. Huang, H. H. 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.
- [31] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeeP-ecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.
- [32] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.
- [33] J. Lin and D. Mohaisen. From large to mammoth: A comparative evaluation of large language models in vulnerability detection. In *NDSS*, 2025.
- [34] S. Lipp, S. Banescu, and A. Pretschner. An empirical study on the effectiveness of static code analyzers for vulnerability detection. In *ISSTA*, 2022.
- [35] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.
- [36] Y. Liu, R. Meng, S. Jot, S. Savarese, C. Xiong, Y. Zhou, and S. Yavuz. Codexembed: A generalist embedding model family for multilingual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644*, 2024.
- [37] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In *ACL*, pages 8086–8098, 2022.
- [38] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021. doi: 10.1109/TSE.2019.2946563.
- [39] F. Marques, M. Ferreira, A. Nascimento, M. E. Coimbra, N. Santos, L. Jia, and J. Fragoso Santos. Automated exploit generation for node. js packages. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1341–1366, 2025.
- [40] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira. Vulnerable code detection using software metrics and machine learning. *IEEE Access*, 8:219174–219198, 2020.
- [41] W. Melicher, C. Fung, L. Bauer, and L. Jia. Towards a lightweight, hybrid approach for detecting dom xss vulnerabilities with machine learning. In *Proceedings of the Web Conference 2021*, WWW '21, 2021.
- [42] S. Min, M. Lewis, H. Hajishirzi, and L. Zettlemoyer. Noisy channel language model prompting for few-shot text classification. In *ACL*, pages 5316–5330, 2022.
- [43] J. Nam and S. Kim. Clami: Defect prediction on unlabeled datasets (t). In *ASE*, pages 452–463. IEEE, 2015.
- [44] R. Pryzant, D. Iter, J. Li, Y. T. Lee, C. Zhu, and M. Zeng. Automatic prompt optimization with "gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023.
- [45] G. Ramesh, M. Sahil, S. A. Palan, D. Bhandary, T. A. 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.
- [46] N. Risse, J. Liu, and M. Böhme. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. (ISSTA):388–410, 2025.
- [47] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- [48] O. Rubin, J. Herzig, and J. Berant. Learning to retrieve prompts for in-context learning. In *ACL*, pages 2655–2671, 2022.
- [49] A. Sabbatella, A. Ponti, I. Giordani, A. Candelieri, and F. Archetti. Prompt optimization in large language models. *Mathematics*, 12(6):929, 2024.
- [50] R. Safdar, D. Mateen, S. T. Ali, M. U. Ashfaq, and W. Hussain. Data and context matter: Towards generalizing ai-based software vulnerability detection. *arXiv preprint arXiv:2508.16625*, 2025.
- [51] B. Steenhoeck, M. M. Rahman, R. Jiles, and W. Le. An empirical study of deep learning models for vulnerability detection. In *ICSE*, 2023.
- [52] B. Steenhoeck, H. Gao, and W. Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *ICSE*, 2024.
- [53] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen. Source code summarization in the era of large language models. In *ICSE*, 2025.
- [54] G. Team. Gemma 3. 2025. URL <https://go.gle/Gemma3Report>.
- [55] Q. Team. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.
- [56] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- [57] Y. Wan, Z. Bi, Y. He, J. Zhang, H. Zhang, Y. Sui, G. Xu, H. Jin, and P. Yu. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Computing Surveys*, 56(12):1–41, 2024.
- [58] W. Wang, P. Kattakinda, and S. Feizi. Maestro: Joint graph & config optimization for reliable ai agents, 2025. URL <https://arxiv.org/abs/2509.04642>.
- [59] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 35:24824–24837, 2022.
- [60] B. Xu, Q. Wang, Z. Mao, Y. Lyu, Q. She, and Y. Zhang. k nn prompting: Beyond-context learning with calibration-free nearest neighbor inference. *arXiv preprint arXiv:2303.13824*, 2023.
- [61] B. Xu, A. Yang, J. Lin, Q. Wang, C. Zhou, Y. Zhang, and Z. Mao. Expertprompting: Instructing large language models to be distinguished experts. *arXiv preprint arXiv:2305.14688*, 2023.
- [62] A. S. Yadav and J. N. Wilson. R+ r: Security vulnerability dataset quality is critical. In *ACSAC*, pages 1047–1061. IEEE, 2024.
- [63] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [64] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [65] A. Z. Yang, H. Tian, H. Ye, R. Martins, and C. L. Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024.

- [66] T. Zheng, H. Liu, H. Xu, X. Chen, P. Yi, and Y. Wu. Few-vuld: A few-shot learning framework for software vulnerability detection. *Computers & Security*, 144: 103992, 2024.
- [67] X. Zhou, S. Cao, X. Sun, and D. 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.
- [68] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *NeurIPS*, 32, 2019.

A 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 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:
```

B Full Results with Standard Deviation

An extended version of Table 1 with standard deviation is shown in Table 2. A key observation is the general consistency of the methods—most exhibit low standard deviations. Table 2 reinforces the reliability of the mean results discussed in the main text.

As anticipated, the R-FS approach consistently displayed the highest variance, underscoring its sensitivity to the random selection of examples. This is particularly evident on datasets like SVENC with the Qwen model, where the Recall was 0.422 (± 0.373). In contrast, the failure cases of certain baselines and methods proved to be remarkably deterministic. For instance, the LFM method’s tendency to uniformly predict the positive class on datasets like NodeMedic is confirmed by its perfect recall with zero deviation (1.000 ± 0.000) for both Gemma and Qwen models.

Furthermore, the stability of our more successful methods lends additional credence to their effectiveness. The LFNN approach on NodeMedic not only achieved a high F1-score (0.854 for Gemma) but did so with minimal variance (± 0.002). Similarly, the Combined Methods on the DiverseVul dataset demonstrated lower variance than R-FS, suggesting that their improved performance is not an artifact of random chance.

C 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 \times : $n = 20, st = \text{TRUE}, k = 1, opt = I$
- U-1 \times : $n = 20, st = \text{FALSE}, k = 1, opt = I$
- U-m \times (inc.): $n = 20, st = \text{FALSE}, k = 5, opt = I$
- U-m \times (corr.): $n = 20, st = \text{FALSE}, k = 5, opt = C$
- U-m \times (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 3 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-1 \times (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-1 \times) and unstacked (U-1 \times) methods highlights the critical impact of the initial prompt

Table 2: Full results with standard deviation for all approaches across three models. Each cell reports the mean and standard deviation over five runs for Gemma and Qwen models, while GPT results are based on a single run and do not have standard deviation. For the approach names, the following abbreviations are used: ZS = zero-shot, R-FS = random few-shot, LFM = Learn-from-Mistakes, LFNN = Learn-from-Nearest-Neighbors, CM = Combined Method. 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 (±0.037)
	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)	
SVENEC	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)	

Table 3: 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	Gemini-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)	

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, however, 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-mx (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-mx (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-mx (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.