VulScribeR: Exploring RAG-based Vulnerability Augmentation with LLMs

SEYED SHAYAN DANESHVAR, University of Manitoba, Canada

YU NONG, Washington State University, USA

XU YANG, University of Manitoba, Canada

SHAOWEI WANG, University of Manitoba, Canada

HAIPENG CAI, University at Buffalo, USA

Detecting vulnerabilities is vital for software security, yet deep learning-based vulnerability detectors (DLVD) face a data shortage, which limits their effectiveness. Data augmentation can potentially alleviate the data shortage, but augmenting vulnerable code is challenging and requires a generative solution that maintains vulnerability. Previous works have only focused on generating samples that contain single statements or specific types of vulnerabilities. Recently, large language models (LLMs) have been used to solve various code generation and comprehension tasks with inspiring results, especially when fused with retrieval augmented generation (RAG). Therefore, we propose **VulScribeR**, a novel LLM-based solution that leverages carefully curated prompt templates to augment vulnerable datasets. More specifically, we explore three strategies to augment both single and multi-statement vulnerabilities, with LLMs, namely Mutation, Injection, and Extension. Our extensive evaluation across four vulnerability datasets and DLVD models, using three LLMs, show that our approach beats two SOTA methods Vulgen and VGX, and Random Oversampling (ROS) by 27.48%, 27.93%, and 15.41% in f1-score with 5K generated vulnerable samples on average, and 53.84%, 54.10%, 69.90%, and 40.93% with 15K generated vulnerable samples. Our approach demonstrates its feasibility for large-scale data augmentation by generating 1K samples at as cheap as US$ 1.88.

CCS Concepts: • **Software and its engineering** → Software testing and debugging; *Maintaining software*; **Software reliability**; **Software safety**; Parsers; *Software maintenance tools*; **Source code generation**; • **Security and privacy** → **Vulnerability scanners**; **Software security engineering**.

Additional Key Words and Phrases: Vulnerability Augmentation, Deep Learning, Vulnerability Generation, Program Generation, Vulnerability Injection

1 INTRODUCTION

Vulnerability detection is a crucial task in software engineering. Recent Deep Learning-based Vulnerability Detection (DLVD) models [8, 21, 33, 34, 38, 59, 70] have drawn more attention from research and industry communities due to their promising performance. However, these models suffer greatly from 1) a lack of sizable datasets and 2) discrepancies between the distribution of training and testing datasets. Data Sampling techniques [22, 63] can alleviate the former problem to an extent by upsampling vulnerable samples to balance the dataset and optionally adding more clean samples, which are abundant. However, Data Augmentation presents a more promising solution as it can help with both problems by generating large diverse datasets.

Authors' addresses: Seyed Shayan Daneshvar, University of Manitoba, Winnipeg, Canada, daneshvs@myumanitoba.ca; Yu Nong, Washington State University, Pullman, USA, yu.nong@wsu.edu; Xu Yang, University of Manitoba, Winnipeg, Canada, yangx4@myumanitoba.ca; Shaowei Wang, University of Manitoba, Winnipeg, Canada, shaowei.wang@umanitoba.ca; Haipeng Cai, University at Buffalo, Buffalo, USA, haipengc@buffalo.edu.

For instance, Zhang [69] and Liu et al. [37] focus on generating specific type(s) of bug/vulnerability. Similarly, Nong et al. proposed VulGen [44] and VGX [42] to mine single-statement vulnerability patterns and inject them into clean samples to generate vulnerable samples. Alternatively, Daneshvar et al. [12] evaluated mixup-inspired [68] augmentation techniques for token-based DLVD and proposed conditioned variants for augmenting vulnerabilities at the representation level. Despite their contributions, these methods have limitations. They either generate only specific types of vulnerabilities [42, 44, 69] or require sizable datasets to learn models for generating vulnerable code or pinpointing the location to inject the vulnerability [42–44], or they are limited to a certain class of DLVD models while yielding minimal performance gains [12], making them impractical for adoption in a real-world scenario.

It is worth noting that recent works have also explored vulnerability detection using LLMs [14, 16, 36, 38], which pose as competitors to DLVD models. However, adopting LLM-based vulnerability detection methods poses various challenges and limitations [2, 64] that make DLVD models still highly relevant: 1) Data Privacy Concerns: in the industry, there are privacy concerns regarding the use of LLMs for vulnerability detection and essentially exposing private commercial code to LLMs; 2) Limited Robustness to Domain-specifc Vulnerabilities: LLMs require robust datasets to be used alongside various techniques, thus they usually struggle with project and domain-specifc vulnerabilities not present in the training data; 3) High Computational Costs: LLMs are resource-intensive to run while VulScribeR uses LLMs to augment vulnerable samples once and the trained DLVD models can be deployed while requiring significantly less compute and memory, making them easier to be deployed on-premise or in a constrained environment. Consequently, it is important to augment vulnerable data to improve the performance of DLVD models.

Recently, Large Language Models (LLMs) have demonstrated promising results in code-related tasks, such as code understanding [31, 62], generation [25], and vulnerability understanding [18, 30]. Intuitively, vulnerable data augmentation is a task that requires the ability of code comprehension, typically in vulnerability understanding, and code generation. LLMs' strong ability for code comprehension and code generation fits this task well.

Therefore, to tackle both of those problems and overcome the limitations of previous works, we propose **VulScribeR**, a novel LLM-based solution that leverages carefully curated prompt templates to augment vulnerable datasets. More specifically, we design three strategies to generate vulnerable new code samples, namely **Mutation**, **Injection**, and **Extension**. **Mutation** refers to changing vulnerable code using code transformation that preserves the code's semantics and syntactical correctness by prompting LLMs. **Injection** prompts LLMs to inject vulnerable segments of the vulnerable samples into a clean sample to create a new vulnerable sample. **Extention** aims to extend the vulnerable sample by adding parts of the logic of the clean sample to the vulnerable sample to improve the diversity of the context where the vulnerability could occur.

To generate realistic code, we employ Retrieval-Augmented Generation (RAG) in **Injection** and **Extension**. More specifically, in **Injection**, we retrieve vulnerable samples that are similar to the clean sample into which the vulnerable segment will be injected. Additionally, to enhance the diversity of the generated vulnerable samples, we employ a clustering process to ensure that the data in all clusters have the choice to be retrieved during RAG. We employ RAG and clustering in **Extension** similarly.

Eventually, we employ a fuzzy parser as a filtering mechanism to discard invalid responses such as empty code samples and those with severe syntactical errors, to reduce noise in the generated samples.

To assess VulScribeR, we generated vulnerable samples for each strategy using two different LLMs, and evaluated the effectiveness of each data augmentation approach on three SOTA DLVD models (including both token-based and graph-based models) and three commonly used vulnerable datasets. We conducted various experiments in the form of the following Research Questions (RQs):

- *RQ1: How effective is VulScribeR compared to SOTA approaches?*
  Results: Both **Injection** and **Extension** outperform the baselines and the **Mutation** by a large margin,

while **Injection** provides a slightly higher performance compared to the **Extension**. For instance, **Injection** outperforms NoAug, Vulgen, VGX, and ROS by 30.80%, 27.48%, 27.93%, and 15.41% on average F1-score.

- *RQ2: How does RAG contribute to VulScribeR?*
  Results: RAG component makes a significant contribution to **Injection** and **Extension** strategies, accounting for 4.99% and 10.77% of their respective improvements.
- *RQ3: How does the quantity of the generated samples impact the effectiveness of vulnerability detection models?*
  Results: Augmenting more vulnerable data by using **Injection** helps improve the effectiveness of DLVD models, while VulGen, VGX, and random over-sampling fail to improve the performance of DLVD models by augmenting more than 5K vulnerable samples. Our LLM-based approach is more feasible for large-scale vulnerable data augmentation.
- *RQ4: Can VulScribeR be effective on large, heavily-imbalanced datasets with complex real-world vulnerabilities?*
  Results: on PrimeVul, both **Injection** and **Extension** outperform the baselines and **Mutation** by a considerable margin, while **Extension** provides a slightly higher performance compared to **Injection**. For instance, **Extension** outperforms NoAug, Vulgen, VGX, and ROS by 22.80%, 30.83%, 25.24%, and 3.87% on average F1-score.

In summary, our contributions are as follows:

- To the best of our knowledge, we are the first to explore vulnerability augmentation using LLMs. We carefully designed three novel prompt templates with different strategies and proposed a comprehensive pipeline for vulnerability augmentation that can be used for large-scale vulnerability augmentation (that is as cheap as US$1.88 per 1K samples).
- We performed an extensive evaluation using two different LLMs, three DLVD models, and three datasets, demonstrating the superiority of VulScribeR compared to SOTA baselines, including the best latest techniques.
- We made our source code, experimental results, and the augmented datasets publicly available for future research [1].

2 BACKGROUND & RELATED WORK

2.1 Vulnerability Augmentation

Vulnerable code samples are scarce in practice, and various approaches have been developed [22, 37, 42–44] to enhance the capability of vulnerability detection models by generating such data and expanding the datasets. For instance, Nong et al. [43] explored the feasibility of vulnerability injection through a neural code editing model [65], which is a DL-based model trained to transform clean code into vulnerable code via introducing a set of predicted changes to the AST of the program. This approach requires high-quality datasets and hence suffers from a chicken-egg dilemma and has limited use. Ganz et al. [22] present CodeGraphSMOTE, a method to generate new vulnerable samples by porting SMOTE [9] to the graph domain. They convert code to graphs and use graph autoencoders to encode the graphs into their latent space. Then, it applies SMOTE (i.e. an interpolation-based sampling method) on the latent space to generate new vulnerable items.

Nong et al. later proposed VulGen [44], which mines vulnerabilities to collect single-statement vulnerable patterns. Then uses a modified transformer model to locate where injection should take place. VGX [42] is an improved version of VulGen where a significantly bigger vulnerability dataset was used to cover a wider range of single-statement vulnerabilities in the mining phase and the localization model was replaced with a semantics-aware contextualization transformer to predict the injection contexts better. However, both VulGen
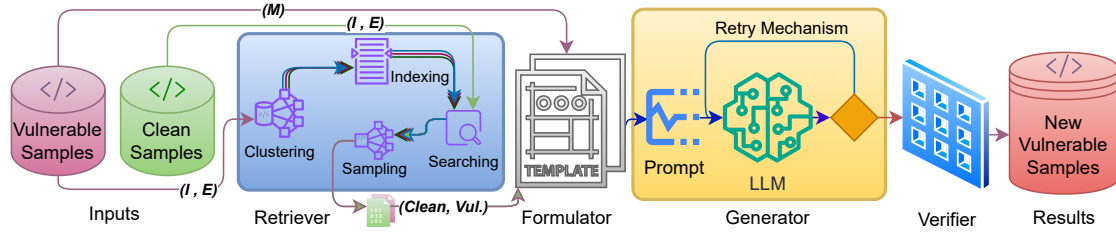
Fig. 1. The components of VulScribeR. All of the proposed strategies share the same generation and verifier components. The Formulator component of the mutation strategy only requires the vulnerable samples and receives input directly. *(M, I, E)* tags distinguish the difference between the input data flows for **M**utation, **I**njection, and **E**xtension strategies, respectively.

and VGX only target single statement vulnerabilities and require a comprehensive pattern mining process. Single statement vulnerabilities only account for less than 40% of all vulnerabilities in the Bigvul [17] dataset.

2.2 Source Code Augmentation with RAG and LLMs

Leveraging LLMs for source code augmentation is getting popular and it has been used for data augmentation for Semantic Code Search [60] and Code Generation [10], both of which heavily rely on RAG. Wang et al. [60] leverages RAG to retrieve similar *query − code* pairs, then for each pair they instruct ChatGPT [46] to change the query and code in separate prompts using predefined rules included in the prompts. Then, they filter the augmented pairs using an encoder model that calculates their similarity score, removing the low-scoring pairs. Chen et al. [10] utilized Code Search to augment data for Code Generation. Specifically, they retrieve *context − function* pairs, where the context can be either a function header or a comment, to populate prompt templates that instruct the LLM to generate code based on the given context and retrieved function. Their architecture uses three components, namely Retriever, Formulator, and Generator. Similar to previous studies, we also employ RAG to retrieve relevant context to enhance the code generation, while different from them, we focus on augmenting vulnerabilities and we employ a clustering process to ensure the diversity of generated data.

3 METHODOLOGY

In this section, we elaborate on the details of our methodology, VulScribeR. We propose a RAG-based solution that leverages carefully curated prompt templates with a lenient filtering mechanism to generate vulnerable code snippets by utilizing LLMs, which allows us to generate diverse and realistic vulnerable code snippets.

We specifically design three prompting strategies to generate vulnerable code samples for data augmentation.

Consequently, We carefully designed 3 prompt templates for each augmentation strategy, namely **Mutation**, **Injection**, and **Extension** templates, on which we will elaborate in Section 3.1.

Figure 1 presents the workflow for the three proposed strategies, and all of the three proposed augmentation strategies can be abstracted into at most four components if applicable as follows:

- *Retriever* Given input code samples (i.e., vulnerable and clean samples), Retriever is responsible for seeking suitable vulnerable and non-vulnerable pairs from the database and attaching them in the prompt to provide context for vulnerable sample generation if applicable. **Injection** and **Extension** strategies employ the Retriever component, while **Mutation** does not require a Retriever.
- *Formulator* For different strategies, we employ their corresponding prompt template. The Formulator instantiates the corresponding prompt template by filling input code samples or the retrieved pairs from Retriever if applicable. For the **Mutation** prompt template, the input comes directly from the vulnerable

samples dataset, while it comes in the form of pairs of clean and vulnerable samples from the Retriever for other templates.

- *Generator* In the Generator, we use the instantiated templates to prompt the LLM to generate vulnerable code samples. If the response does not contain a code snippet or we face an API error, a retry mechanism is activated to feed the prompt to the LLM up to three times to receive a response that contains a code snippet; otherwise, we move on to the next prompt.

- *Verifier* The goal of the Verifier[1] is to have control over the quality of the generated code, as there's no guarantee that the LLM produces compilable code. We use Joern's [51], a fuzzy C language parser, which is based on Antlr's [49] C Parser, to filter out generated codes that contain severe syntax errors. The rationale behind using a fuzzy parser instead of a strict parser is that in data augmentation for various machine learning tasks, the generated data does not need to conform to the original data's strict standards [12, 13, 15, 27, 32, 45, 67, 68]. Note that our goal in this study is to augment existing vulnerable datasets to help models capture the vulnerability patterns and generalize better, rather than generating high-quality vulnerable samples. As such, completely correct code is not necessary for training models [61], and augmented code can improve the performance of the models even when they slightly break the source code syntax [15, 32]; in fact, data with subtle noise helps with the generalization of models [5, 20, 29]. We use a parser instead of a compiler as we wish to keep the post-processing lightweight, as compilation needs significant efforts, such as identifying the dependencies and resolving type errors, which are unavailable in the original datasets, as they only contain functions. Nevertheless, we acknowledge that more sophisticated methods can be used for the verification phase, but starting from a simple module (i.e, a parser) was necessary for evaluating whether using LLMs for vulnerability detection is even feasible. As a result, the fuzzy C Parser filters out 2%-13% of the generated data.

Note that all of our three strategies share the same **Generator** and **Verifier** components, while different strategies have slightly different designs for **Retriever** and **Formulator** components. We discuss the details of the Retriever and Formulator components for each strategy in Section 3.1.

3.1 Augmentation Strategies

In this section, we break down the details of **Mutation**, **Injection**, and **Extension** strategies. Specifically, we introduce the prompt template and Retriever for each strategy.

*3.1.1 **Mutation**.* The majority of previous works on program augmentation [6, 7, 15, 40, 48, 66], rely on code transforms that do not change the flow, semantics, and syntactical correctness of the program using program analysis. Variable name changing, replacing for with while loops and vice versa, and adding dead code are examples of this. However, program analysis is very time-consuming, and selecting the locations for transformation is challenging. Previous studies usually select the types and locations of transformation randomly without any program comprehension [6, 7, 15, 40, 48, 66]. In **Mutation** strategy, we aim to augment vulnerable code samples by utilizing the program comprehension capability of LLM to mutate existing vulnerable code samples and let the LLM choose both the type of transformation and the potential statements to transform. In this way, we get around the program analysis and rely on LLM's creativity to generate more suitable and diverse items. We design the prompt template as shown in Figure 2.

We instruct the LLM to use one of the 18 program transformation rules by following a recent study [66] and mutating vulnerable code samples while using at least one of the rules. When designing the template, we prioritize the following points. First, we aim to generate diverse code. For this, we mention all of the rules in a

---

[1]Verifier is a carefully chosen generic term; in this work, it is only a fuzzy parser that filters code with severe syntax errors.

Prompt Template: Mutation

Here's a code snippet including a function. Except for the important lines mentioned below, mutate the rest of the code snippet so it is different from the original while keeping the original semantics. To do so you can use one or more of the following rules:

Rules: 1- Replace the local variables' identifiers with new non-repeated identifiers
2- Replace the for statement with a semantic-equivalent while statement, and vice versa
3- *{...remaining_rules_are_hidden_to_save_space}*

Code Snippet: *{input_vulnerable_sample}*

The following lines are important and should be included in the final result, but they can still be changed using only the first 5 rules, the rest may be changed using any of the rules or can even be removed if they have no relation to these lines: (Lines are separated via /□/) *{input_vulnerable_lines}*

Put the generated code inside "'C "'.
(Do not include comments)

Fig. 2. The mutation prompt template.

single prompt instead of having a prompt per rule to give the LLM the freedom to choose the more suitable rules and apply them to the vulnerable code snippet. Second, we seek to keep the existing vulnerability unchanged in the code. As some of the rules might change the execution trace of the code (e.g. transforming a "switch-case" to an "if-else" statement) and potentially change the status of the vulnerability, we instruct the LLM to use such rules only on the lines that are not important. Important lines are essentially the vulnerable lines of the code snippet. We refer to vulnerable lines as important lines in the prompt to prevent unwanted changes in the resulting code. However, these lines can still be transformed with the rules (i.e., the first five rules in the template) that do not change the execution trace without changing the vulnerable state of the code snippet.

As presented in Figure 1, the overall workflow for **Mutation** is straightforward. To augment N vulnerable samples, N vulnerable samples are randomly sampled from the input dataset, and then are directly fed into the Formulator to instantiate the template, as a result, N prompts are instantiated by filling the templates. The prompts are then fed to the Generator to generate N vulnerable samples. It's worth noting that on average 3% of the generated samples will be filtered out in the Verifier component and so if one desires to end up with at least N samples, a higher target should be selected in the generator phase, or else they should redo the generation after the verification to create more items to reach the target.

3.1.2 ***Injection***. Similar to recent works [42–44], we also focus on injecting vulnerable code segments into a clean sample as our main strategy, but we aim to cover all types of vulnerabilities and not just single statement vulnerabilities. More specifically, we instruct the LLM to inject the logic of the vulnerable sample into a clean sample by prioritizing the injection of vulnerable segments. Using an LLM for injecting the vulnerable segments into a clean item gives the freedom to LLM to identify the best location. We present our **Injection** template prompt as shown in Figure 3.

The templates contain two or three placeholders to fill, namely *input_vulnerable_sample*, *input_clean_sample*, and *input_vulnerable_lines*. *input_clean_sample* is the input clean samples where we aim to inject vulnerability and is not present in the **Mutation** strategy. *input_vulnerable_sample* is a vulnerable code example to be retrieved that is similar to the clean input sample. We retrieve similar vulnerable code because we believe that the logic from

---

Prompt Template: Injection

Here are two code snippets specified below, modify code snippet 2 in a way that it includes the logic of code snippet 1:

Code Snippet 1: *{input_vulnerable_sample}*
Code Snippet 2: *{input_clean_sample}*

Note that the following lines from the first code snippet have a high priority to be in the final modified code:
Lines separated by /□/: *{input_vulnerable_lines}*

Put the generated code inside "'C "'.
(Do not include comments)

Fig. 3. The injection prompt template.

a similar vulnerable code could be integrated into the clean code more easily and naturally. *input_vulnerable_lines* provides meta-information indicating the vulnerable lines in the retrieved vulnerable example. We instruct the LLM to prioritize including these vulnerable lines since our goal is to generate vulnerable code.

To fill the template, we need to construct a dataset of pairs of clean samples and their corresponding retrieved similar vulnerable code samples (i.e., *clean-vul* pairs). Note that we choose to provide a clean code sample as the input and search for similar vulnerable samples, rather than providing a vulnerable code sample and searching for similar non-vulnerable samples. This strategy is faster and more efficient due to the smaller number of vulnerable samples, which reduces the dataset size for the retrieval process.

Algorithm 1 outlines our method for constructing a dataset of *clean-vul* pairs through a retrieval process. Given a dataset containing both vulnerable samples (V) and clean samples (C), our goal is to retrieve N *clean-vul* pairs. A straightforward approach would be to retrieve the most similar vulnerable samples for each clean sample, sort them in descending order of similarity, and select the top N pairs. However, focusing solely on similarity reduces the diversity of the retrieved samples, which is counterproductive for data augmentation. Previous studies have shown that higher diversity in the training dataset improves the generalizability of deep learning models [24, 57]. Therefore, to enhance diversity, we incorporate a clustering phase to group the vulnerable samples into G clusters, ensuring that samples from all clusters are considered for selection.

First, we cluster the vulnerable samples into G groups (line 2). To do this, we use CodeBERT [19] to embed each code sample into a vector, specifically utilizing the [CLS] token's embedding for each sample. We then apply KMeans for clustering, using cosine similarity to measure the similarity of each pair of samples.

Following clustering, we create an index on each cluster using Lucene to facilitate efficient search (lines 7-10). For each clean sample, we search for the most similar vulnerable sample within each cluster using the cluster index, and store the resulting *clean-vul* pairs for further processing (lines 12-17). To find similar vulnerable samples for each clean sample, we use the BM-25 algorithm [55], as previous studies have shown that there is no significant difference between sparse methods like BM-25 and dense methods like CodeBERT when leveraging RAG [10, 23, 28, 41]. We use Lucene's implementation of BM-25 for this study.

Once the similarity score is calculated for each pair of clean-vul, we sort the results within each cluster based on the similarity score (lines 19-21). Then, we iteratively select the top pair from each cluster, ensuring a diverse yet relevant selection (lines 25-32). We also sort the groups based on size and start from the largest group ensuring a higher coverage for higher values of G (line 23) by following previous study [39]. We set $G = 5$ in our study.

Algorithm 1: (Clean, Vulnerable) Pair Retrieval

Input: V: Vulnerable samples
Input: C: Clean samples
Input: N: Number of pairs to sample
Input: G: Number of groups into which the vulnerable samples will be clustered
Output: D: Dataset with N Clean-Vulnerable pairs

1 $vul\_embeddings \leftarrow$ compute_codebert_embedding(V)
2 $clustered\_vuls \leftarrow$ KMeans($G, V, vul\_embeddings$)
3 $indices \leftarrow [\,]$
4 $clustered\_pairs \leftarrow [[\,], [\,], ..., [\,]]$ // An array of G arrays
5 $D \leftarrow [\,]$
6 // Index vulnerable samples of each cluster separately
7 **for** $cluster\_id = 0$ **to** $G - 1$ **do**
8      $search\_engine\_index \leftarrow$ index($clustered\_vuls[cluster\_id]$)
9      $indices[cluster\_id]$.append($search\_engine\_index$)
10 **end for**
11 // each clean item is matched with the most similar vulnerable item of each cluster
12 **foreach** $clean \in C$ **do**
13   **for** $cluster\_id = 0$ **to** $G - 1$ **do**
14         $vul \leftarrow$ search($indices[cluster\_id], clean$)
15         $clustered\_pairs[cluster\_id]$.append($(clean, vul)$)
16   **end for**
17 **end foreach**
18 // Sort each cluster's pairs based on their similarity score
19 **for** $cluster\_id = 0$ **to** $G - 1$ **do**
20     $sort\_by\_score(clustered\_pairs[cluster\_id], "desc")$
21 **end for**
22 // sort the arrays based on their size in descending order
23 $clustered\_pairs \leftarrow$ sort_arrays_by_size($clustered\_pairs, "desc"$)
24 $i \leftarrow 0$ // Iterate between the clustered_pairs and select the best pair each time
25 **while** $size(D) < N$ **do**
26     $pairs \leftarrow clustered\_pairs[i \mod G]$
27     $index \leftarrow i//G$
28   **if** $index < size(pairs)$ **then**
29   D.append($pairs[index]$)
30   **end if**
31     $i \leftarrow i + 1$
32 **end while**
33 **return** D

After the clean-vul pairs are retrieved, those pairs will be fed into Formulator to instantiate the **Injection** template and go through the rest of the components in the workflow to generate N vulnerable new samples similar to **Mutation**.

3.1.3 ***Extension.*** While injecting vulnerable code segments into clean code is the defacto yet effective way of augmenting vulnerabilities, extending an already vulnerable code snippet by adding additional logic can also generate a new vulnerable sample, which enriches the context where vulnerable code happens. Adding certain parts from a clean sample to an already vulnerable sample has a higher chance of keeping the context of the

original vulnerable code intact compared to injecting vulnerable segments into a clean item. By extending an already vulnerable sample, only a small section will be irrelevant to the vulnerable part of the code which is the important part for the models, while injecting vulnerable segments into a clean sample results in a code snippet where the vulnerable section has little connection to the original context and gives itself away. Therefore, we aim to explore it as an alternative to vulnerability injection. We refer to this strategy as **Extension**. We present the prompt template for **Extension** in Figure 4.

Prompt Template: Extension

Here are two code snippets 1 and 2, each including a function. Add some parts of the logic of 1 to 2 in a way that the result would include important lines of 2 with some parts from 1. You can add variables to 2 to produce a more correct code.

Code Snippet 1: *{input_clean_sample}*
Code Snippet 2: *{input_vulnerable_sample}*

The following lines from 2 are important and should be included in the final result, the rest may be changed or even removed if they have no relation to these lines: (Lines are separated via /_/) *{input_vulnerable_lines}*

Put the generated code inside "'C "' and note that the final result should be a function that takes all the input args of 2 and more if required.
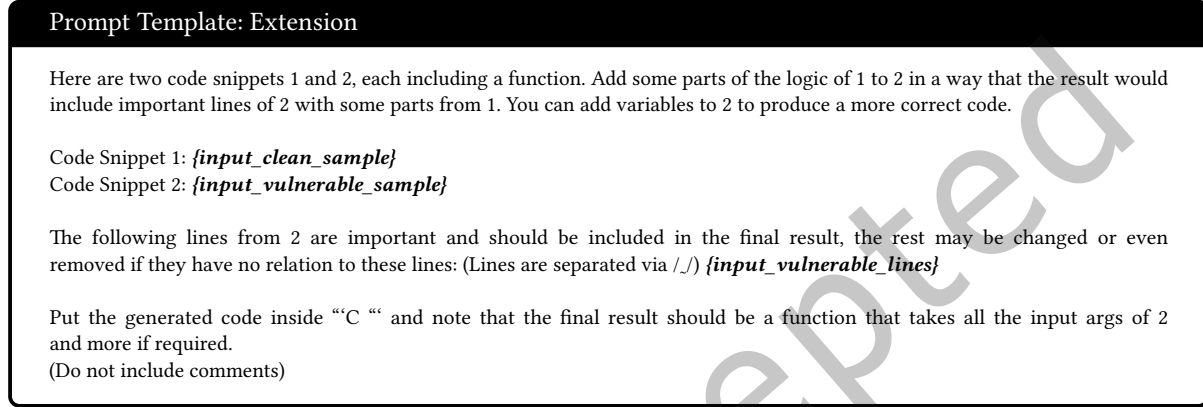(Do not include comments)

Fig. 4. The extension prompt template.

Similar to the **Injection** strategy, we instruct the LLM to add sections of the logic of the clean sample to the vulnerable sample, while enforcing a no-change policy on the vulnerable lines. The only difference is that rather than injecting vulnerable samples into clean samples, the **Extension** strategy does the reverse. Therefore, we can exactly reuse the workflow of the **Injection** strategy by modifying the Retriever, where we aim to retrieve similar clean samples for given vulnerable samples and keep all other components the same. Note that we also tried instructing the LLM to extend vulnerable samples freely without providing clean samples, which offers the LLMs more freedom in extending the vulnerable samples. However, it does not work well, and tends to hallucinate in most cases. Specifically, The LLM usually leaves the vulnerable sample unchanged or behaves like a mutator (e.g., mutating variable names or adding dead code). The LLMs perform poorly at generating code when they are not given clear instructions, hence we designed our strategies in a way that we simply ask the LLM to follow clear instructions (i.e. mix two code pieces) without mentioning the word vulnerability and without asking the LLM anything vague or too general. This prevents the LLM from focusing on the vulnerability aspect of the codes and encourages following the given instructions, rather than hallucinating.

4 EXPERIMENTAL SETTING

In this section, we present our research questions (RQs), the datasets, DLVD models, LLMs, evaluation metrics, our analysis approach for each of the RQs, and implementation details.

4.1 Research Questions

We evaluate VulScribeR from different aspects to answer the following research questions.

- *RQ1: How effective is VulScribeR compared to SOTA approaches?*

- *RQ2: How does RAG contribute to VulScribeR?*
- *RQ3: How does the quantity of the generated samples impact the effectiveness of vulnerability detection models?*
- *RQ4: Can VulScribeR be effective on large, heavily-imbalanced datasets with complex real-world vulnerabilities?*

In RQ1, we evaluate the effectiveness of VulScribeR for improving the performance of DLVD models by augmenting vulnerable data, by comparing it to current SOTA approaches. In RQ2, we aim to investigate the effectiveness of our design for the Retriever component. In RQ3, we explore how the number of generated samples by our approach and SOTA methods affects the performance of DLVD models. In RQ4, we investigate the usefulness of VulScribeR on a new large, high-quality, heavily-imbalanced dataset with real-world vulnerabilities.

Table 1. The Studied Datasets

	Vul. Samples	Clean Samples	Total	Ratio
Design	10768	12024	22792	1:1.1
Reveal	2240	20494	22734	1:9.1
BigVul$_{Train}$	8783	142125	150908	1:16.2
BigVul$_{Validation}$	1038	17826	18864	1:17.2
BigVul$_{Test}$	1079	17785	18864	1:16.5
PrimeVul$_{Train}$	4802	169345	174147	1:35.3
PrimeVul$_{Validation}$	593	23347	23940	1:39.4
PrimeVul$_{Test}$	549	24232	18864	1:44.1

4.2 Datasets

In this study, we use three different widely used vulnerability detection datasets [8, 21, 42, 44, 63, 70], namely Design [70], Reveal [8], and BigVul [17], to evaluate our approach. Additionally, we evaluate our method with PrimeVul [14], which is a new large dataset with less label noise, in RQ4. Table 1 shows the details of these datasets. All of these datasets include C/C++ functions gathered from real-world projects. For Design, Reveal, BigVul, and the extra clean data [35] used for retaining the ratio, we used the cleaned version from VGX [42], which had duplicates removed between the datasets. For PrimeVul, we used the SHA2 hash to remove samples available in Design and the extra cleans [35] from all three sets of the PrimeVul dataset.

4.2.1 Design. Following previous studies [42, 44], we use Design as our primary training dataset as it offers a better balance between the two categories, namely the vulnerable and clean samples (i.e., non-vulnerable samples) by following a previous study, VulGen [44]. We also use the clean samples of this dataset as the clean inputs for the **Retriever** component.

4.2.2 Bigvul. We use all three sets (i.e., training, validation, and testing set) of BigVul [17] for different purposes. We use BigVul's training set as the source for collecting vulnerable samples to feed it to the **Retriever** as the vulnerable samples. However, not all of these items can be used as they lack the vulnerable lines metadata that are required for the **Injection** and **Extension** prompt templates. Hence, we only use the 6,610 vulnerable samples from BigVuls's training set that include the metadata of vulnerable lines. We use the testing set of BigVul as one of the testing sets in our study.

4.2.3 Reveal. Similar to VulGen [44], we use Reveal as another testing set in our study. Reveal is a slightly larger dataset than BigVul's testing set and has twice the number of vulnerable samples.

4.2.4 PrimeVul. Unlike VGX [42] and VulGen [44], which use Devign and BigVul datasets, which are considerably noisy [11, 14, 54] (i.e., some vulnerable samples in these datasets are clean), we also use PrimeVul [14] in RQ4 with the settings of RQ1 with some changes (see more details in Section 5.4). PrimeVul is the largest dataset used in our study, and one of the most imbalanced datasets available. We use all three sets of Primevul in RQ4, with the train set replacing Devign as the main training set as well as BigVul's train set for the retrieved vulnerable samples, and the validation and testing sets replacing those of BigVul.

4.3 Deep learning-based vulnerability detection (DLVD) Models

There are two families of DLVD models, token-based and graph-based. To represent both families, we select a SOTA token-based model and two SOTA graph-based models, namely LineVul [21], Devign [70], and Reveal [8] models. For Devign and Reveal models, we use the same setting as VulGen [44] and train both of these models 5 times with random seed values, test them and report the results with the highest F1-score achieved. For Linevul, we follow the settings of VGX [42], train for 10 epochs, and select the checkpoint that achieved the highest F1 score on Bigvul's validation set. We specifically train and test the models with different datasets, following the settings of VulGen [44] and VGX [42], to ensure there is no information leakage and the results can be trusted to apply to real-world unseen data.

4.4 Evaluation metrics

DLVD is inherently a classification task as a category is assigned to the code snippet. Hence, to evaluate the effectiveness of the studied methods, we use the common classification metrics [50], in line with previous related studies [34, 42, 44, 50, 52, 53, 59, 63], namely Precision, Recall, and F1-Score.

4.5 Base LLMs

In our study, we use two different LLMs to generate vulnerable code, namely ChatGPT3.5 [46], a commercial general-purpose LLM, and CodeQwen1.5 [58], an open-source LLM specialized for code tasks.

For ChatGPT, We use the "gpt-3.5-turbo", variant of OpenAI's ChatGPT [46] which has shown great potential in previous studies [10, 23, 56, 60]. We set the temperature hyper-parameter to 0.5 since we wish to reduce the randomness of the response to a degree that ChatGPT would follow our instructions, but would not limit its freedom of creativity. We left all the other hyper-parameters to their default values. For CodeQwen, we use the 7B variant of CodeQwen1.5 [58], "CodeQwen1.5-7B-Chat", which is a variant of the general purpose Qwen [4]. CodeQwen1.5, similar to ChatGPT, is a decoder-only transformer-based model with the difference that it was pre-trained on code data, and supports a larger context length. We did not change any of the hyper-parameters for this model and used the defaults, except for the number of new tokens that we set to ChatGPT's default (i.e. 4096 tokens).

Additionally, we use GPT4o-mini [47] for our RQ4, which is as capable as ChatGPT3.5, but more affordable. Similar to ChatGPT3.5, we set the temperature parameter to 0.5, but we did not change any other hyperparameters.

4.6 Baselines

To compare our studies with previous work and demonstrate the effectiveness of VulScribeR we consider the following baselines:

- **NoAug:** This is the most basic of our baselines, where we use the original dataset for training without introducing any changes.

- **ROS:** As Yang et al. [63] show that Random Oversampling (ROS) can have a considerable effect on improving the performance of DLVD, thus we also ROS as another baseline.
- **VulGen** Vulgen [44] is a SOTA vulnerability generation method that mines single-statement vulnerabilities and uses a transformer model for locating where a vulnerable pattern should be used for injection. We use this baseline as it can generate a significant amount of data that can be used to improve the performance of DLVD models.
- **VGX:** VGX [42] is the improved version of VulGen that uses a larger dataset for mining single-statement patterns and employs a more sophisticated localization model. Hence, we use VGX as it is potentially a more powerful method and can generate more diverse high-quality data than VulGen.

4.7 Approach for RQs

4.7.1 Approach of RQ1. To demonstrate the effectiveness of VulScribeR and examine which of the proposed strategies is superior to others, we first generate 5K vulnerable samples by employing different approaches (i.e., VulScribeR and baselines). More specifically, to sample 5K items with any of our strategies, we first aim to generate up to 6K samples with the **Generator** which are then filtered by the **Verifier** to yield more than 5K generated samples, from which we sample randomly to get 5K items. For VGX and VulGen, we use the generated results in VGX [42] and sample 5K from their results randomly. For ROS, we randomly up-sample 5K samples from the vulnerable samples in the Devign dataset.

We then train the three DLVD models using the augmented Devign dataset. To augment the Devign dataset, we add the 5K generated vulnerable samples to it and add the proportional number of clean samples from the vulnerability benchmark proposed in [35] to maintain the original ratio of the dataset, by following previous studies [42, 44]. We use BigVul's testing set and Reveal dataset to evaluate the performance of all of these models. We assess the performance of a vulnerable data augmentation approach by assessing the performance of a specific DLVD with the augmented dataset of the specified strategy. Hence, each strategy will be tested in 12 instances (i.e. 3 DLVD Models * 2 Testing Datasets * 2 LLMs).

4.7.2 Approach of RQ2. In this RQ, we investigate the usefulness of our proposed RAG for the **Injection** and **Extension** strategies. Specifically, we eliminate the **Retriever** component, and instead, we match the clean and vulnerable samples randomly to examine the effect of RAG (i.e. **Injection w/o Retriever** and **Extension w/o Retriever**). Moreover, we also conduct an ablation study on the clustering phase of the **Retriever** component to examine the effect of diversity on the quality of the generated samples. To do so, we remove the clustering phase from the Retriever component and search for similar samples for each clean input item within all the vulnerable items instead of within each cluster. Then we take the top 5 retrieved items to end up with a similar number of $clean - vul$ pairs. In the sampling phase, we sort all the pairs based on their relevance score and start from the top. We denote the **Injection** strategy without clustering phase as **Injection w/o Clustering** and the **Extension** strategy without clustering as **Extension w/o Clustering**.

Similar to RQ1, we generate 5k vulnerable samples using the studied approaches and compare their effectiveness by assessing the DLVD models that are trained on the augmented datasets.

4.7.3 Approach of RQ3. In this RQ, we investigate how the number of augmented samples impacts the performance of DLVD models, and how much performance can be gained by providing more data.

To see the impact and find out whether VulScribeR can be used for large-scale vulnerability augmentation, we use the **Injection** strategy, our best strategy as depicted in Section 5, and generate up to 16.5K samples so that after the filtering we end up with a set of at least 15K vulnerable samples which is about 40% higher than the number of vulnerable samples in the original dataset. Then we repeat the experiments for the **Injection** using

both LLMs. We sample 10K and 15K vulnerable items while adding the proportional number of clean items to maintain the ratio. We do the same for VGX [42], VulGen [44], and ROS.

4.7.4 Approach of RQ4. Lastly, we propose a slightly different setting than RQ1 for our approach to work with heavily imbalanced large datasets, and evaluate the effectiveness of our strategies.
For this RQ, we utilized GPT4-omini and used PrimeVul's trainset as the primary training set as well as the source for collecting vulnerable samples that are fed to the **Retriever**, and used Devign's cleans as the source of cleans. Since the PrimeVul dataset does not contain line-level information of vulnerabilities, we collected the vulnerable lines by computing the diff between paired clean and vulnerable samples and extracting the vulnerable lines.

It is worth mentioning that PrimeVul is a challenging dataset with a highly imbalanced ratio between vulnerable and clean samples (1:35) for training DLVD models [14]. To make training possible, we balanced the dataset by randomly removing clean samples to achieve a ratio of 1:15 (larger ratios made training less effective on some models). Then we followed the settings of RQ1 for each strategy, and collected 5K new samples. To conduct a fair comparison between the methods, we used ROS to randomly oversample the original vulnerable data to get to a balanced dataset (i.e., 1:1 ratio) for training DLVD. Note that we do not change the ratio of the validation and testing sets.

4.8 Implementation details

We used a machine with four 24GB Nvidia RTX 3090s for our experiments. We conducted a total of 781 experiments for all our RQs, which cost close to 2000 GPU hours solely for training and testing DLVD models (excluding the time spent generating the samples).

5 RESULTS

5.1 *RQ1: How effective is VulScribeR compared to SOTA approaches?*

Table 2. Comparison of VulScribeR's strategies with baselines using ChatGPT and CodeQwen when augmenting 5K samples across the studied DLVD models (i.e., Devign, Reveal, and Linevul). The cells with larger values (better performance) compared to NoAug are highlighted darker.

	Strategy	ChatGPT 3.5 Turbo									CodeQwen1.5-7B-Chat								
		Devign			Reveal			Linevul			Devign			Reveal			Linevul		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Reveal	NoAug	10.59	47.23	17.30	12.90	65.14	21.54	5.57	16.98	8.39	10.59	47.23	17.30	12.90	65.14	21.54	5.57	16.98	8.39
	VulGen	10.45	57.90	17.70	12.46	64.60	20.90	5.31	98.09	10.08	10.45	57.90	17.70	12.46	64.60	20.90	5.31	98.09	10.08
	VGX	11.02	59.35	18.58	11.69	65.62	19.84	9.51	87.24	11.47	11.02	59.35	18.58	11.69	65.62	19.84	9.51	87.24	11.47
	ROS	13.28	39.93	19.93	14.35	54.52	22.72	9.88	22.71	13.77	13.28	39.93	19.93	14.35	54.52	22.72	9.88	22.71	13.77
	Mutation	10.98	53.44	18.22	12.23	68.46	20.75	7.80	22.62	11.60	10.96	61.64	18.60	15.01	57.96	**23.85**	9.48	30.43	14.45
	Injection	13.44	53.26	21.46	15.43	61.46	**24.66**	11.57	26.43	**16.10**	13.67	54.28	**21.83**	14.23	64.48	23.32	9.99	25.79	14.40
	Extension	14.97	37.21	21.35	14.76	50.00	22.80	18.01	12.81	14.97	15.20	38.18	21.75	14.82	52.71	23.13	20.65	12.62	**15.67**
Bigvul	NoAug	6.27	43.24	10.95	7.28	77.27	13.31	7.90	29.19	12.43	6.27	43.24	10.95	7.28	77.27	13.31	7.90	29.19	12.43
	VulGen	6.49	62.32	11.75	6.36	70.59	11.66	8.51	22.150	12.29	6.49	62.32	11.75	6.36	70.59	11.66	8.51	22.15	12.29
	VGX	6.06	59.30	10.99	6.89	73.29	12.60	5.35	98.82	10.15	6.06	59.30	10.99	6.89	73.29	12.60	5.35	98.82	10.15
	ROS	7.54	25.60	11.65	7.96	33.39	12.86	10.91	13.16	11.93	7.54	25.60	11.65	7.96	33.39	12.86	10.91	13.16	11.93
	Mutation	7.80	63.91	**13.90**	7.63	67.73	13.72	6.78	74.14	12.42	7.56	57.23	13.36	9.37	53.26	**15.94**	6.65	78.31	12.25
	Injection	8.79	29.41	13.53	11.64	38.00	**17.82**	11.43	18.81	14.22	10.33	31.80	**15.59**	9.91	38.16	15.74	7.85	30.49	12.49
	Extension	10.60	17.33	13.16	11.13	34.02	16.77	19.03	11.96	**14.68**	9.89	31.48	15.05	10.22	33.86	15.70	15.74	11.86	**13.53**

Extension and Injection outperforms Mutation. Injection slightly outperforms Extension. Table 2 presents the results of VulScribeR and baselines across different experimental instances. As observed, **Extension** and **Injection** beat **Mutation** in most (9 out of 12) instances. **Extension** and **Injection** perform close to each other, while **Injection** appears to have the advantage by a tiny margin in most (9 out of 12) settings. In terms of F1-score, **Injection** achieves an F1-score of 17.60%, outperforming **Extension** (F1-score 17.38%) and **Mutation**

(F1-score 15.75%) by 0.96% and 12.44% on average, respectively. One possible reason is that **Injection** and **Extension** enrich the context where vulnerabilities could occur while **Mutation** does not alter the semantics of vulnerable code. **Mutation** does not enrich the context of the semantics of the original vulnerable code.
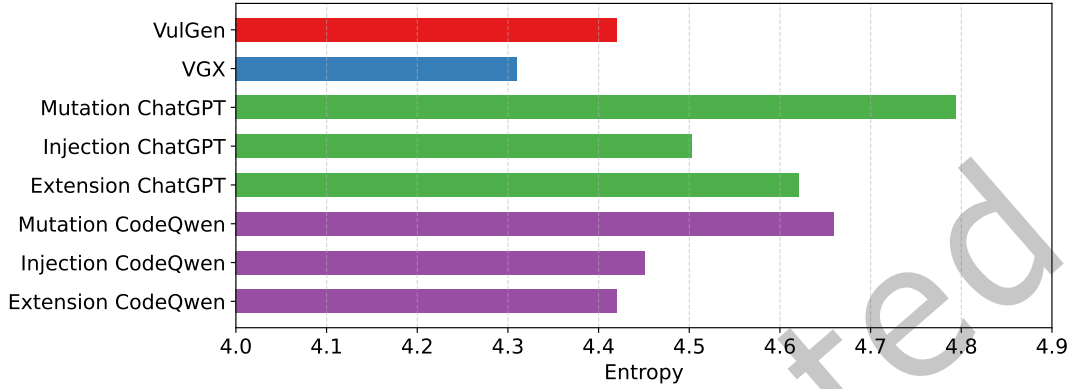

Fig. 5. Entropy of the augmented vulnerable datasets by different approaches. Higher indicates more diversity.

All our proposed LLM-based augmentation strategies outperform the baselines. Typically, Injection and Extension always outperform baselines in all instances. Injection outperforms NoAug, Vulgen, VGX, and ROS by 30.80%, 27.48%, 27.93%, and 15.41% on average F1-score, respectively. As shown in Table 2, **Injection** and **Extension** strategies outperform baselines (i.e., NoAug, VulGen, VGX, ROS) in all of the experimental instances in terms of F1-score. More specifically, **Extension** beats the baselines: NoAug, Vulgen, VGX, and ROS by 29.68%, 26.27%, 26.90%, and 14.35% on average F1-score, while the **Injection** strategy beats the baselines: NoAug, Vulgen, VGX, and ROS by 30.80%, 27.48%, 27.93%, and 15.41% on average F1-score respectively.

As observed, **Mutation** beats most baselines by a large margin. However, **Mutation** does not have a dominating advantage over ROS, and ROS beats **Mutation** in some (4 out of 12) instances. It is worth noting that ROS beats the baseline, Vulgen, and VGX by 13.96%, 10.72%, and 11.21% in terms of F1-score on average.
As discussed in Section 2, SOTA approaches like VGX and VulGen only focus on single-statement that limits the diversity of generated vulnerabilities. However, our approaches do not have this limitation and are able to generate more diverse vulnerable samples. To examine this, we measure the diversity of the augmented vulnerable samples by our approaches and VGX and VulGen. By following previous studies, we first apply principle component analysis (PCA) on CodeBERT's embeddings of the generated vulnerable samples and reduce the dimension to three. Next, we calculate the histogram using 10 bins, from which we calculate the entropy of the vectors. Higher entropy values indicate greater diversity among the vulnerable samples, while lower values suggest a more concentrated distribution with similar samples. As depicted in Figure 5 the entropy for **Mutation**, **Injection**, and **Extension** are 4.79, 4.5, and 4.62 for ChatGPT, and 4.66, 4.45, 4.42 for CodeQwen, while for Vulgen and VGX, it is 4.42 and 4.31 respectively. The results demonstrate that our approaches have a better potential for generating more diverse samples as we beat them while using only one of the datasets they used for vulnerability mining. **VulScribeR produces more diverse vulnerable samples compared to SOTA approaches VGX and VulGen.**

Lastly, it is worth noting that the results on both of the LLMs perform very close to each other. CodeQwen1.5-7B-Chat slightly outperforms ChatGPT3.5 Turbo by a tiny margin (i.e. 1.49%) averaged across all three strategies,

yet one cannot conclude that CodeQwen is a better LLM for vulnerability generation as we did not explore the hyper-parameters for the optimum setting for each of the LLMs. However, we can conclude that our strategies work with similar LLM to ChatGPT and CodeQwen (e.g. GPT4 [3] and DeepSeek-Coder [26]) and do not require an LLM that is trained specifically on code.

> Both **Injection** and **Extension** outperforms the baselines and the **Mutation** by a large margin, while **Injection** provides a slightly higher performance compared to the **Extension**. For instance, **Injection** outperforms NoAug, Vulgen, VGX, and ROS by 30.80%, 27.48%, 27.93%, and 15.41% on average F1-score.

Table 3. Results of Ablation Studies on the Retriever and Clustering for Injection across the studied DLVD models (i.e., Devign, Reveal, and Linevul).

	Strategy	ChatGPT 3.5 Turbo									CodeQwen1.5-7B-Chat								
		Devign			Reveal			Linevul			Devign			Reveal			Linevul		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Reveal	w/o Retriever	13.91	44.33	21.18	14.17	64.29	23.22	8.11	32.15	12.95	11.94	57.12	19.75	12.52	72.74	21.37	13.00	32.06	18.50
	w/o Clustering	14.75	36.37	20.99	15.59	59.89	24.74	9.92	24.61	14.14	13.86	43.37	21.01	14.92	52.17	23.20	12.23	24.61	16.34
	Injection	13.44	53.26	**21.46**	15.43	61.46	**24.66**	11.57	26.43	**16.10**	13.67	54.28	**21.83**	14.23	64.48	**23.32**	9.99	25.79	14.40
Bigvul	w/o Retriever	8.47	36.09	13.72	8.96	65.50	15.76	6.67	66.54	12.12	8.87	53.74	15.23	8.88	64.39	15.60	7.09	61.72	12.72
	w/o Clustering	8.53	33.23	13.57	11.19	32.75	16.68	9.56	16.68	12.16	9.77	32.75	15.05	9.49	48.65	15.88	14.61	21.78	17.49
	Injection	8.79	29.41	13.53	11.64	38.00	**17.82**	11.43	18.81	**14.22**	10.33	31.80	**15.59**	9.91	38.16	15.74	7.85	30.49	12.49

Table 4. Results of Ablation Studies on the Retriever and Clustering for Extension across the studied DLVD models (i.e., Devign, Reveal, and Linevul).

	Strategy	ChatGPT 3.5 Turbo									CodeQwen1.5-7B-Chat								
		Devign			Reveal			Linevul			Devign			Reveal			Linevul		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Reveal	w/o Retriever	13.96	35.59	20.05	14.00	49.94	21.87	8.63	12.20	12.20	11.55	67.73	19.74	13.21	65.26	21.98	11.52	32.15	16.96
	w/o Clustering	15.14	36.01	21.32	14.34	63.63	23.41	7.82	48.38	13.46	15.68	36.85	22.00	14.93	50.84	23.09	9.77	21.34	13.40
	Extention	14.97	37.21	**21.35**	14.76	50.00	22.80	18.01	12.81	**14.97**	15.20	38.18	21.75	14.82	52.71	**23.13**	20.65	12.62	15.67
Bigvul	w/o Retriever	7.88	15.42	10.43	8.62	25.12	12.83	7.13	72.75	12.99	7.66	67.57	13.77	8.91	62.48	15.60	6.89	67.66	12.50
	w/o Clustering	10.78	33.70	16.33	11.13	22.58	14.91	8.20	17.98	11.26	12.52	30.21	17.70	9.83	39.75	15.76	9.38	26.41	13.85
	Extention	10.60	17.33	13.16	11.13	34.02	**16.77**	19.03	11.96	**14.68**	9.89	31.48	15.05	10.22	33.86	**15.70**	15.74	11.86	13.53

5.2 RQ2: How does RAG contribute to VulScribeR?

For Injection, the clustering phase of the Retriever adds 0.49% to the F1-score averaged across all settings compared to w/o clustering, and w/o clustering improves the average f1-score by 6.34% compared to w/o Retriever. As shown in Table 3, both similarity and diversity affect the effectiveness of the models positively with similarity having a more significant impact. When comparing **Injection** with **Injection** w/o Retriever, we observe that the complete Retriever component improves the effectiveness of the **Injection** by 4.99% on the average F1-score and in 8 out of 12 instances. Specifically, when we compare **Injection** with w/o the clustering phase, we see that the clustering phase enhances the effectiveness of RAG by 0.49% on the average F1-score and in 8 out of 12 instances, and by comparing w/o clustering with w/o Retriever, we see that RAG enhances the effectiveness of RAG by 6.34% on the average F1-score and in 8 out of 12 instances.

For Extension strategy, the clustering phase of The Retriever provides 2.54% of improvement to the F1-score averaged across all settings compared to w/o clustering, and w/o clustering improves the average f1-score by 5.61% compared to w/o Retriever. As shown in Table 4, both similarity and diversity affect the effectiveness of the models positively with similarity having a more significant impact. Overall, our complete **Retriever** module improves the effectiveness of the **Extension** strategy by 10.77% on the average

F1-score and beats the other settings in 6 out of 12 settings. Specifically, by comparing **Extension** with w/o clustering we see that the clustering phase enhances the effectiveness of RAG by by 2.54% on the average F1-score and in 7 out of 12 instances, and by comparing w/o clustering with w/o Retriever, we see that RAG enhances the effectiveness of RAG by by 5.61% on the average F1-score and in 10 out of 12 instances.

> Retriever component makes significant contribution for **Injection** and **Extension**. **Extension** gains more than twice the increase from the **Retriever** component, implying that it is more sensitive to both the similarity and diversity of the retrieved pairs.

5.3 *RQ3: How does the quantity of the generated samples impact the effectiveness of vulnerability detection models?*

Table 5. Impact of Generated Samples on Improving DLVD models' Performance at 5K, 10K, and 15K

| | Strategy | ChatGPT 3.5 Turbo | | | | | | | | | CodeQwen1.5-7B-Chat | | | | | | | | |
| | | Devign | | | Reveal | | | Linevul | | | Devign | | | Reveal | | | Linevul | | |
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Reveal	NoAug	10.59	47.23	17.30	12.90	65.14	21.54	5.57	16.98	8.39	10.59	47.23	17.30	12.90	65.14	21.54	5.57	16.98	8.39
	VulGen 5K	10.45	57.90	17.70	12.46	64.60	20.90	5.31	98.09	10.08	10.45	57.90	17.70	12.46	64.60	20.90	5.31	98.09	10.08
	VulGen 10K	10.21	46.20	16.72	11.31	66.77	19.34	5.37	100.00	10.20	10.21	46.20	16.72	11.31	66.77	19.34	5.37	100.00	10.20
	Vulgen 15K	10.30	55.25	17.37	11.23	75.51	19.55	5.35	99.09	10.15	10.30	55.25	17.37	11.23	75.51	19.55	5.35	99.09	10.15
	VGX 5K	11.02	59.35	18.58	11.69	65.62	19.84	9.51	87.24	11.47	11.02	59.35	18.58	11.69	65.62	19.84	9.51	87.24	11.47
	VGX 10K	10.31	57.90	17.50	11.53	56.45	19.15	5.24	95.91	9.93	10.31	57.90	17.50	11.53	56.45	19.15	5.24	95.91	9.93
	VGX 15K	9.01	34.68	14.30	11.95	63.75	20.13	5.38	99.91	10.21	9.01	34.68	14.30	11.95	63.75	20.13	5.38	99.91	10.21
	ROS 5K	13.28	39.93	19.93	14.35	54.52	22.72	9.88	22.71	13.77	13.28	39.93	19.93	14.35	54.52	22.72	9.88	22.71	13.77
	ROS 10K	12.92	51.81	20.68	14.00	46.20	21.49	9.39	30.06	14.31	12.92	51.81	20.68	14.00	46.20	21.49	9.39	30.06	14.31
	ROS 15K	13.25	47.29	20.71	13.77	43.85	20.96	8.18	19.71	11.56	13.25	47.29	20.71	13.77	43.85	20.96	8.18	19.71	11.56
	Injection 5K	13.44	53.26	21.46	15.43	61.46	24.66	11.57	26.43	16.10	13.67	54.26	21.83	14.23	64.48	23.32	9.99	25.79	14.40
	Injection 10K	17.39	42.70	24.71	16.37	63.21	26.00	11.19	31.97	16.57	15.87	48.31	23.90	14.96	63.03	24.18	15.05	19.53	17.00
	Injection 15K	16.94	52.65	25.63	17.00	57.36	26.22	13.02	36.61	19.21	15.60	52.53	24.06	16.27	57.18	25.33	16.01	23.89	19.17
Bigvul	NoAug	6.27	43.24	10.95	7.28	77.27	13.31	7.90	29.19	12.43	6.27	43.24	10.95	7.28	77.27	13.31	7.90	29.19	12.43
	VulGen 5K	6.49	62.32	11.75	6.36	70.59	11.66	8.51	22.150	12.29	6.49	62.32	11.75	6.36	70.59	11.66	8.51	22.150	12.29
	VulGen 10K	6.59	52.62	11.72	6.21	76.79	11.50	12.04	12.70	12.36	6.59	52.62	11.72	6.21	76.79	11.50	12.04	12.70	12.36
	Vulgen 15K	6.35	62.96	11.54	6.48	69.95	11.86	8.32	14.74	10.64	6.35	62.96	11.54	6.48	69.95	11.86	8.32	14.74	10.64
	VGX 5K	6.06	59.30	10.99	6.89	73.29	12.60	5.35	98.82	10.15	6.06	59.30	10.99	6.89	73.29	12.60	5.35	98.82	10.15
	VGX 10K	5.84	61.21	10.65	6.36	63.28	11.56	7.14	10.10	8.37	5.84	61.21	10.65	6.36	63.28	11.56	7.14	10.10	8.37
	VGX 15K	6.11	50.40	10.90	6.42	66.14	11.70	9.36	6.49	7.66	6.11	50.40	10.90	6.42	66.14	11.70	9.36	6.49	7.66
	ROS 5K	7.54	25.60	11.65	7.96	33.39	12.86	10.91	13.16	11.93	7.54	25.60	11.65	7.96	33.39	12.86	10.91	13.16	11.93
	ROS 10K	7.64	32.43	12.37	9.32	33.55	14.59	8.13	18.44	11.28	7.64	32.43	12.37	9.32	33.55	14.59	8.13	18.44	11.28
	ROS 15K	9.31	21.14	12.93	8.22	26.87	12.59	8.77	13.07	10.49	9.31	21.14	12.93	8.22	26.87	12.59	8.77	13.07	10.49
	Injection 5K	8.79	29.41	13.53	11.64	38.00	17.82	11.43	18.81	14.22	10.33	31.80	15.59	9.91	38.16	15.74	7.85	30.49	12.49
	Injection 10K	13.94	17.60	17.60	12.09	43.56	18.92	8.84	33.09	13.96	11.49	30.37	16.67	11.54	41.81	18.08	9.86	19.83	13.17
	Injection 15K	12.70	34.66	18.59	11.92	41.81	18.55	11.78	30.15	16.94	12.12	33.86	17.85	11.54	41.81	18.09	9.74	29.19	14.60

Augmenting more vulnerable data by using Injection helps improve the effectiveness of DLVD models. Table 5 and Figure 6 present the performance (in terms of F-1 score) of DLVD models when being trained on a dataset that is augmented with different vulnerable samples generated by **Injection** and other baselines (e.g., VulGen, VGX, ROS, and NoAug). As we can see, The performance of DLVD models increases as more augmented vulnerable samples are added to the training data in almost all experimental instances (11 out of 12), which indicates that adding more data augmented by **Injection** provides more useful information for the model to capture vulnerabilities.

In general, VulGen, VGX, and ROS fail to improve the performance of DLVD models by augmenting more vulnerable samples. When looking at the baselines in Figures 6, we notice that Vulgen, VGX, and ROS struggle with improving the performance of DLVD models by augmenting more data. For instance, if we compare the performance of DLVD models when augmenting 5K and 15K vulnerable samples, VGX, VulGen, and ROS decrease the performance in 5 out of 6, 4 out of 6, and 4 out of 6 instances, respectively (Note that the baselines are not dependent on the LLMs, but they are shown for easier comparison). This finding aligns with the results reported in Yang et al. [63], indicating that excessive random over-sampling does not improve
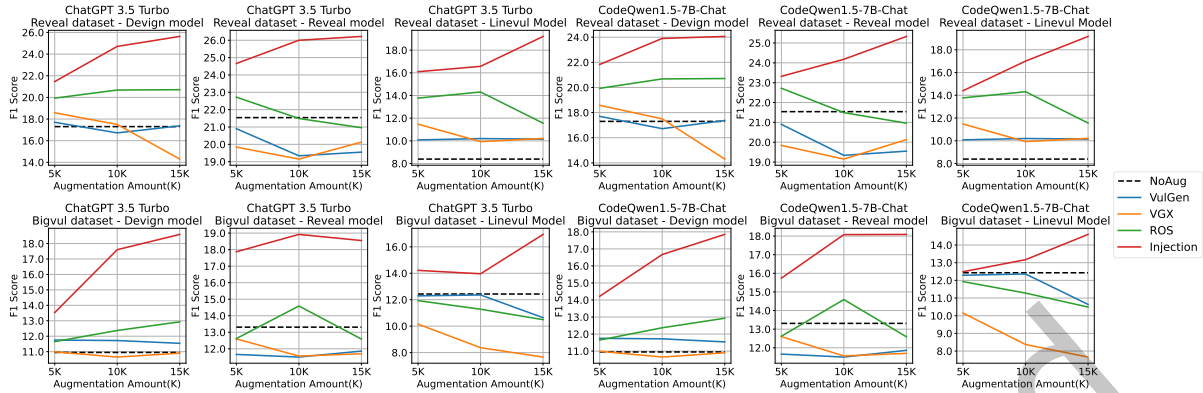
Fig. 6. The impact of the number of augmented samples on the effectiveness of DLVD models across the studied datasets and LLMs across the studied DLVD models (i.e., Devign, Reveal, and Linevul).

the effectiveness of DLVD models and can diminish their performance. One possible explanation is that since random over-sampling does not introduce any new information, and excessive over-sampling probably leads to overfitting of the oversampled data in the models. For VGX and VulGen, this is probably due to the nature of their method, which solely focuses on single-statement vulnerabilities. This heavily limits the usefulness of such methods and makes them unfeasible to be used for large-scale data augmentation.

In summary, **Injection** outperforms all baselines (i.e., VulGen, VGX, and ROS) in all scales, typically when used for large-scale data augmentation. The studied baselines fail to improve the effectiveness of DLVD models by augmenting more than 5K samples. Our LLM-based approach is more feasible for large-scale vulnerable data augmentation.

> Augmenting more vulnerable data by using **Injection** helps improve the effectiveness of DLVD models, while VulGen, VGX, and ROS fail to improve the performance of DLVD models by augmenting more than 5K vulnerable samples. In contrast, our LLM-based approach is more feasible for large-scall vulnerable data augmentation. Explicitly, **Injection** at 15K beats NoAug, Vulgen at 15K, VGX at 15K, and ROS at 15K by 53.84%, 54.10%, 69.90%, and 40.93%.

5.4 *RQ4: Can VulScribeR be effective on large, heavily-imbalanced datasets with complex real-world vulnerabilities?*

Extension and Injection outperform Mutation. Extension slightly outperforms Injection. Mutation continues to struggle against ROS. Table 6 presents the results of VulScribeR and baselines across different DLVDs. As observed, **Extension** and **Injection** beat **Mutation** in all cases. **Extension** and **Injection** perform close to each other, while **Extension** appears to be more effective. In terms of F1-score, **Extension** achieves an F1-score of 16.89%, outperforming **Injection** (F1-score 16.41%) and **Mutation** (F1-score 15.24%) by 2.88% and 10.78% on average, respectively. One possible reason is that the **Extension** strategy creates samples that are closer to the decision boundaries compared to **Injection**, with the reason lying in the source of the vulnerable samples and cleans. In RQ1 5.1, for the **Injection**, most of the context, being the clean sections, comes from the same dataset, while in this RQ, this is true for **Extension**.

Extension appears to be useful and always outperforms baselines in all instances, with Injection

Table 6. Comparison of VulScribeR's strategies on the PrimeVul dataset with baselines using GPT4o-mini when augmenting 5K Samples across the studied DLVD models (i.e., Devign, Reveal, and Linevul). The cells with larger values (better performance) compared to NoAug are highlighted darker.

Strategy	Devign			Reveal			Linevul		
	P	R	F1	P	R	F1	P	R	F1
NoAug	25.93	5.12	8.55	7.35	44.63	12.62	18.31	31.15	23.06
VulGen	4.94	59.76	9.11	5.29	59.27	9.72	19.39	28.23	22.99
VGX	5.61	53.66	10.15	5.44	50.73	9.82	19.77	27.69	23.06
ROS	7.45	52.44	13.04	7.58	43.66	13.07	18.48	30.05	22.89
Mutation	7.17	52.93	12.63	7.81	43.66	13.25	15.71	26.96	19.85
Injection	7.57	55.37	13.31	7.81	49.76	13.51	17.29	31.88	22.42
Extension	7.61	53.41	**13.33**	8.51	41.22	**14.11**	18.51	31.15	**23.22**

performing close to Extension and outperforming all baselines with two of the three DLVDs. Extension outperforms NoAug, Vulgen, VGX, and ROS by 14.54%, 21.14%, 17.73%, and 3.39% on average F1-score, respectively. As shown in Table 6, **Extension** strategy outperforms baselines (i.e., NoAug, VulGen, VGX, ROS) in all of the experimental instances in terms of F1-score. The **Injection** fails to increase the performance for LineVul compared to the baselines. More specifically, **Extension** beats the baselines: NoAug, Vulgen, VGX, and ROS by 14.54%, 21.14%, 17.73%, and 3.39% on average F1-score, while the **Injection** strategy beats the baselines: NoAug, Vulgen, VGX, and ROS by 11.33%, 17.74%, 14.43%, and 0.49% on average F1-score, respectively.

As observed, **Mutation** only beats all the baselines with Reveal. Similar to RQ1 5.1, **Mutation** does not have a dominating advantage over ROS, and ROS beats **Mutation** in the two other settings. ROS beats the baseline, Vulgen, VGX, and Mutation by 10.78%, 17.17%, 13.97%, and 7.15% in terms of F1-score on average.

> **Extension** outperforms the baselines and the **Mutation** by a large margin, while **Injection** provides a slightly lower performance compared to the **Extension**, but beats all the baselines with two DLVDs. **Extension** outperforms NoAug, Vulgen, VGX, and ROS by 14.54%, 21.14%, 17.73%, and 3.39% on average F1-score.

6 DISCUSSION

Table 7. Impact of mixing our strategies' generated samples on Improving DLVD models' performance with a total of 5K augmented samples. Each involved strategy provides an equal number of samples. The cells with larger values (better performance) compared to NoAug are highlighted darker.

	Strategy	ChatGPT 3.5 Turbo									CodeQwen1.5-7B-Chat								
		Devign			Reveal			Linevul			Devign			Reveal			Linevul		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Reveal	Mutation	10.98	53.44	18.22	12.23	68.46	20.75	7.80	22.62	11.60	10.96	61.64	18.60	15.01	57.96	23.85	9.48	30.43	14.45
	Injection	13.44	53.26	**21.46**	15.43	61.46	**24.66**	11.57	26.43	16.10	13.67	54.28	**21.83**	14.23	64.48	23.32	9.99	25.79	14.40
	Extension	14.97	37.21	21.35	14.76	50.00	22.80	18.01	12.81	14.97	15.20	38.18	21.75	14.82	52.71	23.13	20.65	12.62	15.67
	M + I	11.89	66.41	20.17	13.74	70.39	22.99	11.46	47.14	**18.43**	11.78	58.02	19.58	13.14	72.74	22.25	9.35	51.41	**15.82**
	M + E	11.23	58.02	18.82	14.14	59.89	22.87	9.45	43.23	15.50	11.92	55.19	19.60	13.20	65.98	22.00	7.85	42.51	13.26
	I + E	13.39	47.71	20.91	15.26	59.71	24.31	22.07	9.01	12.80	14.15	46.98	21.75	14.92	62.24	**24.06**	10.34	18.80	13.35
	M + I + E	11.12	46.98	17.98	9.67	53.26	16.36	10.55	47.14	17.25	11.07	57.60	18.57	13.34	75.09	22.65	40.33	9.47	15.34
Bigvul	Mutation	7.80	63.91	13.90	7.63	67.73	13.72	6.78	74.14	12.42	7.56	57.23	13.36	9.37	53.26	15.94	6.65	78.31	12.25
	Injection	8.79	29.41	13.53	11.64	38.00	**17.82**	11.43	18.81	14.22	10.33	31.80	15.59	9.91	38.16	15.74	7.85	30.49	12.49
	Extension	10.60	17.33	13.16	11.13	34.02	16.77	19.03	11.96	**14.68**	9.89	31.48	15.05	10.22	33.86	15.70	15.74	11.86	**13.53**
	M + I	7.28	66.93	13.14	8.79	66.45	15.53	7.41	72.75	13.45	7.30	58.82	12.99	8.31	64.71	14.72	7.03	83.60	12.97
	M + E	7.24	61.21	12.94	8.68	57.55	15.09	6.94	81.65	12.78	7.45	61.84	13.30	8.27	58.51	14.49	6.64	82.76	12.30
	I + E	9.41	35.29	**16.46**	10.83	39.90	17.04	8.53	41.80	14.16	11.11	31.96	**16.49**	10.42	38.47	**16.40**	7.46	34.01	12.24
	M + I + E	7.14	42.77	15.20	9.25	53.42	15.77	7.38	79.61	13.50	7.45	54.85	13.12	9.23	52.78	15.71	6.83	78.59	12.57

Table 8. Impact of using random oversampling for clean samples instead of adding additional clean samples on our strategies and the baselines. Starred rows indicate using oversampled cleans. Bold F1-scores show the best performance between the two settings for each augmentation strategy.

	Strategy	ChatGPT 3.5 Turbo									CodeQwen1.5-7B-Chat								
		Devign			Reveal			Linevul			Devign			Reveal			Linevul		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Reveal	ROS	13.28	39.93	19.93	14.35	54.52	22.72	9.88	22.71	13.77	13.28	39.93	19.93	14.35	54.52	22.72	9.88	22.71	13.77
	VulGen	10.45	57.90	**17.70**	12.46	64.60	**20.90**	5.31	98.09	10.08	10.45	57.90	**17.70**	12.46	64.60	**20.90**	5.31	98.09	10.08
	VulGen*	8.56	64.29	15.10	5.64	78.38	10.52	5.37	100.00	**10.20**	8.56	64.29	15.10	5.64	78.38	10.52	5.37	100.00	**10.20**
	VGX	11.02	59.35	**18.58**	11.69	65.62	**19.84**	9.51	87.24	**11.47**	11.02	59.35	**18.58**	11.69	65.62	**19.84**	9.51	87.24	**11.47**
	VGX*	8.86	65.98	15.62	5.74	74.40	10.66	5.38	99.91	10.21	8.86	65.98	15.62	5.74	74.40	10.66	5.38	99.91	10.21
	Mutation	10.98	53.44	**18.22**	12.23	68.46	**20.75**	7.80	22.62	**11.60**	10.96	61.64	**18.60**	15.01	57.96	**23.85**	9.48	30.43	**14.45**
	Mutation*	9.05	59.95	15.72	9.90	73.34	17.45	4.66	44.77	8.44	9.07	66.77	15.98	6.25	86.17	11.65	5.66	70.12	10.47
	Injection	13.44	53.26	**21.46**	15.43	61.46	**24.66**	11.57	26.43	16.10	13.67	54.28	**21.83**	14.23	64.48	**23.32**	9.99	25.79	14.40
	Injection*	11.16	63.21	18.97	13.07	69.24	21.99	11.69	27.25	**16.36**	10.33	48.01	17.00	7.90	65.66	14.10	11.35	22.34	**15.05**
	Extension	14.97	37.21	**21.35**	14.76	50.00	**22.80**	18.01	12.81	14.97	15.20	38.18	**21.75**	14.82	52.71	**23.13**	20.65	12.62	**15.67**
	Extension*	10.51	56.82	17.73	12.52	54.46	20.36	12.33	30.43	**17.55**	10.24	46.62	16.80	7.23	53.74	12.74	6.83	60.58	12.28
Bigvul	ROS	7.54	25.60	11.65	7.96	33.39	12.86	10.91	13.16	11.93	7.54	25.60	11.65	7.96	33.39	12.86	10.91	13.16	11.93
	VulGen	6.49	62.32	**11.75**	6.36	70.59	**11.66**	8.51	22.150	12.29	6.49	62.32	**11.75**	6.36	70.59	**11.66**	8.51	22.15	**12.29**
	VulGen*	5.72	72.02	10.60	5.77	85.21	10.81	6.76	50.51	11.92	5.72	72.02	10.60	5.77	85.21	10.81	6.76	50.51	11.92
	VGX	6.06	59.30	**10.99**	6.89	73.29	**12.60**	5.35	98.82	10.15	6.06	59.30	**10.99**	6.89	73.29	**12.60**	5.35	98.82	10.15
	VGX*	5.82	69.16	10.74	5.75	77.27	10.71	6.98	49.03	**12.22**	5.82	69.16	10.74	5.75	77.27	10.71	6.98	49.03	**12.22**
	Mutation	7.80	63.91	**13.90**	7.63	67.73	**13.72**	6.78	74.14	**12.42**	7.56	57.23	**13.36**	9.37	53.26	**15.94**	6.65	78.31	**12.25**
	Mutation*	6.33	70.75	11.63	6.39	78.22	11.82	5.96	83.50	11.12	6.20	82.51	11.53	6.15	85.69	11.48	6.08	92.22	11.40
	Injection	8.79	29.41	**13.53**	11.64	38.00	**17.82**	11.43	18.81	14.22	10.33	31.80	**15.59**	9.91	38.16	**15.74**	7.85	30.49	**12.49**
	Injection*	7.24	54.21	12.77	8.86	59.46	15.42	12.82	23.45	**16.57**	7.46	40.38	12.59	7.90	71.54	14.23	8.61	24.37	12.72
	Extension	10.60	17.33	13.16	11.13	34.02	**16.77**	19.03	11.96	14.68	9.89	31.48	**15.05**	10.22	33.86	**15.70**	15.74	11.86	**13.53**
	Extension*	7.76	47.85	**13.35**	7.58	51.67	13.22	11.75	24.28	**15.84**	8.48	41.18	14.06	6.86	52.15	12.12	6.67	71.73	12.21

6.1 Ensemble of Strategies

Since our strategies appeared to be useful, we decided to investigate whether a mix of strategies would perform better than using a single strategy. For such a comparison, we used the results of RQ1, and added three settings for using two strategies, with each contributing 2.5K samples, amounting to the same 5K extra samples, and an all-strategy setting where one third of the samples came from each of the strategies. Table 7 shows the result of this experiment. As observed, the combination does not always outperform the single strategy. In half (6 out of 12) of the settings, a mixed strategy works worse than a single strategy. Specifically, a mix of all three (M+I+E) and a combination of Mutation and Extension always fail to beat all of the single strategies. The only promising combination is the mix of Injection and Extension (I+E), which achieves the best performance in 4 out of 12 settings.

> The best ensemble setting appears to be a combination of Injection and Extension settings, yet this setting appears to be less useful than only using Injection. Yet, since Extension appears to beat Injection in some settings (e.g. RQ4), we recommend using Injection and Extension strategies and a mix of these to get to achieve a higher performance.

6.2 Impact Analysis of the Extra Clean Samples

Following VGX and Vulgen, a proportional number of extra clean samples are added to the dataset to keep the ratio of vulnerable and clean samples in RQs 1-3. To examine the effect of the extra clean samples, we did an additional study alongside the results of RQ1, where instead of adding extra clean samples, we randomly oversampled the original clean samples to keep the ratio. Aside from our strategies, we also included VGX and Vulgen to see if we get a different result. As observed in Table 8, VGX and Vulgen get better performance in 10 out of 12 settings, while this is in all cases of Mutation, and 9 out of 12 settings for Injection and Extension. Specifically, Vulgen, VGX, Mutation, Injection, and Extension gain 22.75%, 20.07%, 28.4%, 12.76%, and 18.48% on average F1-score, respectively. Hence, Mutation appears to be the most dependent on the addition of extra clean

samples, and Injection appears to be the least dependent method.

Compared to ROS (of RQ1), where only extra clean samples are added and vulnerable samples are oversampled, Vulgen, VGX, and Mutation drop 21.46%, 20.42%, and 17.21%, on average F1-score, respectively. Injection and Extension gain an improvement of 5.12% and 0.27% in terms of the average F1-score. This shows that the effect of vulnerability augmentation is more important for achieving a higher performance than adding extra clean samples for Injection and Extension, while the extra clean samples play a bigger role for Vulgen, VGX, and Mutation strategies.

6.3 Cost Analysis

Based on our experiments, generating every 1,000 vulnerable samples with one prompt at a time and no concurrency, takes 3.4 hours and costs about $1.88 with GPT3.5-Turbo and about 9 GPU hours with CodeQwen1.5-7B-Chat on two RTX3090s. This shows the feasibility of VulScribeR as it takes less than $19 with ChatGPT to augment a dataset like Devign to twice its size and improves the performance of a DLVD model up to 128.95%.

6.4 Quality Analysis of Augmented Vulnerable Samples

We focus on data augmentation and not generation in this study, and introducing noise is inevitable. This is common in the application of other data augmentation techniques in the fields of natural language processing and computer vision [13, 27, 45, 67, 68]. It is crucial to realize that data augmentation is used as a means to improve the generalizability of the models. A reasonable amount of noise can help with the regularization of the model. Otherwise, performance degradation will be seen. In addition, based on the results of recent studies [15, 32] show that code data augmentation techniques that slightly break the syntax can help the training of the models. However, we recognize the importance of evaluating the quality of augmented vulnerable samples to provide practical insights for practitioners when using our approach.

To this end, we conducted a manual assessment of the generated code. Specifically, we randomly selected 50 samples per augmentation strategy (i.e., Mutation, Injection, and Extension), produced using ChatGPT-3.5 in RQ1 (150 samples in total). We manually inspected each sample to determine whether the LLM followed our instructions and successfully generated a vulnerable example that preserved the intended vulnerability pattern. Our analysis revealed that 72%, 82%, and 90% of the samples generated via Mutation, Injection, and Extension, respectively, were correctly implemented and truly vulnerable. These results indicate an acceptable level of quality. The higher success rates for Injection and Extension may help explain their superior performance compared to Mutation. This also suggests that LLMs may find it easier to perform injection and extension tasks than mutation. We also study the failure cases, and we observe certain patterns. For instance, the common case for the Mutation was changing the code in a way different from the methods mentioned, such that the output is significantly different from the instruction. For Injection, wrong injections or half-complete injections were more common. For Extension, this had to do with the removal of the input code instead of extending it.

6.5 Threats to Validity

Internal Validity A common threat to the validity of our study is our hyperparameter settings for DLVD models and LLMs. For DLVD, hyperparameter tuning is extremely expensive given the scale of our study; as such, we followed the setting of previous studies [42, 44]. In addition, when comparing different data augmentation approaches, we used the same settings for DLVD models, which ensures our comparisons are fair. For LLMs, we mostly used the defaults, and only set the number of new tokens to ChatGPT's maximum, and lowered ChatGPT's temperature to 0.5. Better results might be achievable by fine-tuning these hyperparameters.

Label noise in the datasets can pose another threat to the validity of our experiments. Particularly, Devign and BigVul, even though they are commonly used in previous studies, are known to be noisy datasets [11, 14, 54], meaning that a considerable number of vulnerable samples are incorrectly labeled, which can affect the results of our experiments. To ensure these results can be trusted, we expanded our study by dedicating an additional RQ to using PrimeVul, which is a more recent dataset with considerably less label noise, and showed that our results still hold.

Another threat is that LLMs might not always follow the prompts as instructed and could produce hallucinations. To alleviate this threat, we add a verifier component to filter out low-quality code, and we use a retry mechanism that prompts the LLM up to three times in case of errors (including an empty code response). In addition, we study the quality of augmented vulnerable samples, and found that the quality is at an acceptable level. We encourage future research to integrate more advanced approaches to reduce hallucinations. More importantly, our goal in this study is to augment existing vulnerable datasets to help models capture the vulnerability patterns and generalize better, rather than generating high-quality vulnerable samples. Slightly breaking the syntax and not having completely correct code remains beneficial for training of the models [15, 32, 61], and data with subtle noise helps with the generalization of models [5, 20, 29]. We acknowledge that more sophisticated methods can be used for the verification phase, but starting from a simple module (i.e., a parser) was necessary for evaluating whether using LLMs for vulnerability detection is even feasible.

External Validity relates to the generalizability of our findings. Even though we used two different LLMs (ChatGPT 3.5 Turbo and CodeQwen1.5-7B-Chat), evaluated on three commonly used datasets, and covered three SOTA DLVD models from two different families, our findings may not generalize well to real-world scenarios where different models, LLMs, and datasets are used. To encourage future research to investigate more LLMs (including encoder-decoder-based models), datasets, and DLVD models.

7 CONCLUSION

In this paper, we propose VulScribeR, a novel yet effective framework for augmenting vulnerable samples that uses a customized RAG mechanism to formulate carefully designed prompt templates which are then used for prompting the LLMs. We also use a fuzzy parser as a verification method, to make sure that the generated code doesn't have severe syntax problems. Our evaluation results show that our method significantly outperforms SOTA methods, and is suitable for large-scale vulnerability augmentation with low cost.

8 DATA AVAILABILITY

We made our replication package publicly available [1] to encourage future research on enhancing the performance of LLM-generated vulnerable code and more advanced filtering mechanisms.

REFERENCES

[1] 2024. https://github.com/shayandaneshvar/VulScribeR.

[2] Sara Abdali, Richard Anarfi, CJ Barberan, Jia He, and Erfan Shayegani. 2025. Securing Large Language Models: Threats, Vulnerabilities and Responsible Practices. arXiv:2403.12503 [cs.CR] https://arxiv.org/abs/2403.12503

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[4] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang

Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* (2023).

[5] Chris M. Bishop. 1995. Training with Noise is Equivalent to Tikhonov Regularization. *Neural Computation* 7, 1 (01 1995), 108–116. https://doi.org/10.1162/neco.1995.7.1.108 arXiv:https://direct.mit.edu/neco/article-pdf/7/1/108/812990/neco.1995.7.1.108.pdf

[6] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, Canada) *(SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 511–521. https://doi.org/10.1145/3404835.3462840

[7] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 18–30. https://doi.org/10.1145/3540250.3549162

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48 (2020), 3280–3296. https://api.semanticscholar.org/CorpusID:221703797

[9] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *J. Artif. Int. Res.* 16, 1 (jun 2002), 321–357.

[10] Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code Search is All You Need? Improving Code Suggestions with Code Search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 73, 13 pages. https://doi.org/10.1145/3597503.3639085

[11] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 121–133. https://doi.org/10.1109/ICSE48619.2023.00022

[12] Seyed Shayan Daneshvar, Da Tan, Shaowei Wang, and Carson Leung. 2025. A Study on Mixup-Inspired Augmentation Methods for Software Vulnerability Detection. arXiv:2504.15632 [cs.SE] https://arxiv.org/abs/2504.15632

[13] Terrance DeVries and Graham W. Taylor. 2017. Improved Regularization of Convolutional Neural Networks with Cutout. arXiv:1708.04552 [cs.CV] https://arxiv.org/abs/1708.04552

[14] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We? *arXiv preprint arXiv:2403.18624* (2024).

[15] Zeming Dong, Qiang Hu, Yuejun Guo, Zhenya Zhang, Maxime Cordy, Mike Papadakis, Yves Le Traon, and Jianjun Zhao. 2025. Boosting source code learning with text-oriented data augmentation: an empirical study. *Empirical Software Engineering* 30, 3 (2025), 68.

[16] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. arXiv:2406.11147 [cs.SE] https://arxiv.org/abs/2406.11147

[17] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[18] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. 2024. Llm agents can autonomously exploit one-day vulnerabilities. *arXiv preprint arXiv:2404.08144* (2024).

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[20] Benoit Frenay and Michel Verleysen. 2014. Classification in the Presence of Label Noise: A Survey. *IEEE Transactions on Neural Networks and Learning Systems* 25, 5 (2014), 845–869. https://doi.org/10.1109/TNNLS.2013.2292894

[21] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 608–620. https://doi.org/10.1145/3524842.3528452

[22] Tom Ganz, Erik Imgrund, Martin Härterich, and Konrad Rieck. 2023. CodeGraphSMOTE-data augmentation for vulnerability discovery. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 282–301.

[23] S. Gao, X. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 761–773. https://doi.org/10.1109/ASE56229.2023.00109

[24] Zhiqiang Gong, Ping Zhong, and Weidong Hu. 2019. Diversity in Machine Learning. *IEEE Access* 7 (2019), 64323–64350. https://doi.org/10.1109/ACCESS.2019.2917620

[25] Qiuhan Gu. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2201–2203.

[26] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] https://arxiv.org/abs/2401.14196

[27] Hongyu Guo, Yongyi Mao, and Richong Zhang. 2019. Augmenting Data with Mixup for Sentence Classification: An Empirical Study. *ArXiv* abs/1905.08941 (2019). https://api.semanticscholar.org/CorpusID:162168620

[28] Pengfei He, Shaowei Wang, Shaiful Chowdhury, and Tse-Hsun Chen. 2025. Evaluating the Effectiveness and Efficiency of Demonstration Retrievers in RAG for Coding Tasks. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 500–510. https://doi.org/10.1109/SANER64311.2025.00053

[29] Ray J. Hickey. 1996. Noise modelling and evaluating learning from examples. *Artificial Intelligence* 82, 1 (1996), 157–179. https://doi.org/10.1016/0004-3702(94)00094-8

[30] Nafis Tanveer Islam, Joseph Khoury, Andrew Seong, Gonzalo De La Torre Parra, Elias Bou-Harb, and Peyman Najafirad. 2024. LLM-Powered Code Vulnerability Repair with Reinforcement Learning and Semantic Reward. *arXiv preprint arXiv:2401.03374* (2024).

[31] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 124–130.

[32] Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang, and Yanlin Wang. 2022. Exploring Representation-level Augmentation for Code Search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 4924–4936. https://doi.org/10.18653/v1/2022.emnlp-main.327

[33] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 292–303. https://doi.org/10.1145/3468264.3468597

[34] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf

[35] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. 2019. Deep Learning-Based Vulnerable Function Detection: A Benchmark. In *Information and Communications Security: 21st International Conference, ICICS 2019, Beijing, China, December 15–17, 2019, Revised Selected Papers* (Beijing, China). Springer-Verlag, Berlin, Heidelberg, 219–232. https://doi.org/10.1007/978-3-030-41579-2_13

[36] Jie Lin and David Mohaisen. 2025. 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)*.

[37] Shangqing Liu, Wei Ma, Jian Wang, Xiaofei Xie, Ruitao Feng, and Yang Liu. 2024. Enhancing Code Vulnerability Detection via Vulnerability-Preserving Data Augmentation. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 166–177.

[38] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112031. https://doi.org/10.1016/j.jss.2024.112031

[39] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[40] Rui Mao, Li Zhang, and Xiaofang Zhang. 2024. Mutation-based data augmentation for software defect prediction. *Journal of Software: Evolution and Process* 36, 6 (2024), e2634. https://doi.org/10.1002/smr.2634 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2634

[41] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2450–2462. https://doi.org/10.1109/ICSE48619.2023.00205

[42] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-Scale Sample Generation for Boosting Learning-Based Software Vulnerability Analyses. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 149, 13 pages. https://doi.org/10.1145/3597503.3639116

[43] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA.

[44] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VULGEN: Realistic Vulnerability Generation Via Pattern Mining and Deep Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2527–2539. https://doi.org/10.1109/ICSE48619.2023.00211

[45] Viktor Olsson, Wilhelm Tranheden, Juliano Pinto, and Lennart Svensson. 2021. Classmix: Segmentation-based data augmentation for semi-supervised learning. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*. 1369–1378.

[46] OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt. Accessed: 2023-12-28.

[47] OpenAI. 2024. GPT-4o mini. https://platform.openai.com/docs/models/gpt-4o-mini. Accessed: 2025-05-10.

[48] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. 2022. MultIPAs: applying program transformations to introductory programming assignments for data augmentation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1657–1661. https://doi.org/10.1145/3540250.3558931

[49] T. J. Parr and R. W. Quong. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810. https://doi.org/10.1002/spe.4380250705 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705

[50] David Powers. 2011. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.

[51] QwietAI. [n. d.]. Joern: Code analysis tool. https://github.com/joernio/joern.

[52] Gopi Krishnan Rajbahadur, Shaowei Wang, Yasutaka Kamei, and Ahmed E. Hassan. 2021. Impact of Discretization Noise of the Dependent Variable on Machine Learning Classifiers in Software Engineering. *IEEE Transactions on Software Engineering* 47, 7 (2021), 1414–1430. https://doi.org/10.1109/TSE.2019.2924371

[53] Gopi Krishnan Rajbahadur, Shaowei Wang, Gustavo A. Oliva, Yasutaka Kamei, and Ahmed E. Hassan. 2022. The Impact of Feature Importance Methods on the Interpretation of Defect Classifiers. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2245–2261. https://doi.org/10.1109/TSE.2021.3056941

[54] Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 238, 18 pages.

[55] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (apr 2009), 333–389. https://doi.org/10.1561/1500000019

[56] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 23–30. https://doi.org/10.1109/APR59189.2023.00012

[57] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. 2016. C-brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 123, 6 pages. https://doi.org/10.1145/2897937.2897995

[58] Qwen Team. 2024. Code with CodeQwen1.5. https://qwenlm.github.io/blog/codeqwen1.5/

[59] Wenbo Wang, Tien N. Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2249–2261. https://doi.org/10.1109/ICSE48619.2023.00189

[60] Yanlin Wang, Lianghong Guo, Ensheng Shi, Wenqing Chen, Jiachi Chen, Wanjun Zhong, Menghan Wang, Hui Li, Hongyu Zhang, Ziyu Lyu, and Zibin Zheng. 2023. You Augment Me: Exploring ChatGPT-based Data Augmentation for Semantic Code Search. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 14–25. https://doi.org/10.1109/ICSME58846.2023.00014

[61] Yan Wang, Xiaoning Li, Tien N Nguyen, Shaohua Wang, Chao Ni, and Ling Ding. 2024. Natural Is the Best: Model-Agnostic Code Simplification for Pre-trained Large Language Models. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 586–608.

[62] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2024. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for Effective Software Development*. Springer, 71–108.

[63] Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang. 2023. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2287–2298.

[64] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (LLM) security and privacy: The Good, The Bad, and The Ugly. *High-Confidence Computing* 4, 2 (2024), 100211. https://doi.org/10.1016/j.hcc.2024.100211

[65] Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning Structural Edits via Incremental Tree Transformations. In *International Conference on Learning Representations*. https://openreview.net/forum?id=v9hAX77--cZ

[66] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data Augmentation by Program Transformation. *Journal of Systems and Software* 190 (2022), 111304. https://doi.org/10.1016/j.jss.2022.111304

[67] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. 2019. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*. 6023–6032.

[68] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. 2018. mixup: Beyond Empirical Risk Minimization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1Ddp1-Rb

[69] Shasha Zhang. 2021. A Framework of Vulnerable Code Dataset Generation by Open-Source Injection. In *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. 1099–1103. https://doi.org/10.1109/ICAICA52286.2021.9497888

[70] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf