Exploring Representation-level Augmentation and RAG-based Vulnerability Augmentation with LLMs for Vulnerability Detection

by

S. Shayan Daneshvar

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
January 2025

Thesis advisor

Shaowei Wang

Author

S. Shayan Daneshvar

Exploring Representation-level Augmentation and RAG-based Vulnerability Augmentation with LLMs for Vulnerability Detection

Abstract

Using deep learning (DL) for detecting software vulnerabilities has become commonplace. However, data shortage remains a significant challenge due to the scarce nature of vulnerabilities. A few papers have attempted to address the data scarcity issue through oversampling, creating specific types of vulnerabilities, or generating code with single-statement vulnerabilities. In this thesis, we aim to find a general-purpose methodology that covers various types of vulnerabilities and multiple-statement ones while beating previous methods. Specifically, we first explore traditional mixup-inspired augmentation methods that work at the representation level and show that these methods can be useful, although they cannot beat random oversampling. One possible reason is that mixing samples heavily degrades the integrity of the code. Hence, we introduce VulScribeR, a RAG-based vulnerability augmentation pipeline that leverages LLMs and maintains code integrity, unlike mixup-based methods. We show that VulScribeR outperforms the state-of-the-art (SOTA), oversampling, and representation-level augmentation methods.

Contents

List of Figures

List of Tables

Acknowledgments

I want to express my heartfelt gratitude to my advisor for granting me the freedom to pursue a challenging topic of my choosing and encouraging creativity in my research. I am also deeply thankful to my parents, committee members, friends, and lab members for their support throughout this journey. Lastly, I thank my initial advisor, Dr. Lorenzo Livi, with whom I did not have the privilege to work, for his help and support.

To my sanity (what's left of it!), resilience, for enduring not just the stress and late nights but also the streak of bad luck and bouts of bad health that somehow shaped this journey into something meaningful.

Chapter 1

Introduction

In Software Engineering and Development, software vulnerabilities are quite widespread and cause many security risks in the industry. To help with this problem, researchers have tried to come up with deep learning-based vulnerability detection models (DLVD)[15; 62; 5; 25; 26] that can predict the vulnerability of code pieces. However, all these works face the reality of being unable to perform well in real-life scenarios due to a shortage of data. Specifically, vulnerabilities are not always disclosed, and collecting vulnerable code from open-source projects is not easy as they are sometimes fixed silently. To tackle this issue, data augmentation and generation methods can be utilized. Figure1.1 shows a famous vulnerability known as Heartbleed (Buffer Over-Read in OpenSSL) that allows attackers to read the system memory.

Representation-level augmentation refers to an augmentation method in which text data are augmented after they are embedded into a vector space to be used for the training of deep learning models, and is used for many NLP tasks [21]. Following recent works on augmenting data for more common code tasks by Dong et al.[10]

and on code search by Li et al.[23] we explore and show that such augmentation techniques, explored by these papers which are all influenced by the idea of Sentence MixUp [21], can help the training of token-based vulnerability detection models, such as LineVul [15] by a tiny amount due to breaking the code integrity and structure. Hence, we propose a prompt learning-based approach where we leverage LLMs and a Retrieval Augmented Generation (RAG) mechanism, which is a prompt engineering technique, to augment data more efficiently without breaking code integrity, such that it can compete with the state-of-the-art (SOTA) models. Such a model can generate meaningful vulnerable code that is mostly correct in terms of the language's syntax, rather than augmenting their representation.

LLMs have been used for various software engineering tasks (including bug and vulnerability fixing, program repair, unit test generation, and many more) [29; 56; 36; 48; 51], and their usefulness has been demonstrated. However, to our knowledge, none of the current vulnerability generation methods utilize LLMs in any way. While LLMs have already been used for fixing vulnerable code [36], which is the opposite of our task. This is because the creation of a new sample requires two code samples instead of one, namely the retrieved vulnerable sample and the clean code sample, in which the vulnerable segments of the vulnerable sample are injected. We show that while representation-level augmentation can benefit the models somewhat, it still cannot beat randomly oversampling (ROS) vulnerable samples. We also show that all SOTA models fail to compete with ROS. At the same time, our proposed LLM-based RAG-enhanced approach, named **VulScribeR**, beats ROS as well as SOTA methods, making it the first practical vulnerability augmentation approach. In this

approach, we cover 3 different strategies for augmenting data, namely **Mutation**, **Injectin**, and **Extension**.

In this thesis, we show the results of our research which comes in the form of the following research questions (RQs):

- ***RQ 1: How effective is VulScribeR compared to SOTA?***

  Results: Both Injection and Extension outperform the baselines and the Mutation strategy by a large margin, while Injection performs slightly better compared to the Extension strategy. For instance, Injection outperforms NoAug, Vulgen, VGX, and ROS by 30.80%, 27.48%, 27.93%, and 15.41% on average.

- ***RQ 2: How does RAG contribute to VulScribeR?***

  Results: RAG contributes significantly to Injection and Extension strategies.

- ***RQ 3: 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 oversampling 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.

- ***RQ 4: Can Representation-Level Augmentation outperform Random Oversampling (ROS)? Is it effective at all?***

  Results: Representation-level Augmentation using Mixup-inspired methods, cannot beat ROS, but they can help improve performance.

In summary, our contributions include the following:

- To our knowledge, we conducted the first systematic and thorough study to understand how representation-level augmentation affects DLVD. We examined the impact of five data augmentation methods on a leading token-based DLVD approach, utilizing one of the largest datasets available that provide line-level information. Line-level information is crucial, particularly for the conditioned approach.

- We present a novel application of these augmentation methods designed for datasets that incorporate line-level information about the location of vulnerable statements. In this approach, we specifically locate and keep the representation of the corresponding vulnerable line(s) fixed during the augmentation process. This ensures that the augmented vector is more likely to reflect vulnerable data.

- To 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 latest techniques.

- We have made the code for the experiments available. The code for representation-level augmentation and VulScribeR can be found here and here.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFFER_SIZE 64
void send_heartbeat(char *user_input, int length) {
    char buffer[BUFFER_SIZE];
    // Unsafe: Copies `length` bytes from `user_input` to `
       buffer`
    memcpy(buffer, user_input, length);
    printf("Heartbeat response: %.*s\n", length, buffer);
}
int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <data> <length>\n", argv[0]);
        return 1;
    }
    int length = atoi(argv[2]);  // User-controlled length
    send_heartbeat(argv[1], length);
    return 0;
}
```

Figure 1.1: **Heartbleed Vulnerability (CVE-2014-0160)** – A buffer over-read occurs due to a lack of proper bounds checking on user-supplied input. Data is copied from user input into a fixed-size buffer without verifying that the requested length is within bounds, allowing the attacker to read beyond the allocated memory and leak sensitive information.

1.1 Thesis Structure

This thesis is organized as follows: In Chapter 2, we cover an overview of relevant background information and related work on Vulnerability Detection, Augmentation, and Generation. We get more specific in Chapter 3 and define the task of this thesis more formally. We provide the details of our study on representation-level augmentation in 4 and our RAG-based vulnerability augmentation with LLMs approach (VulScribeR) in Chapter 5. Finally, we conclude the thesis and propose future works in Chapter 6. Some parts of this thesis, including figures and tables have appeared in the following publications:

- Daneshvar, S., Nong, Y., Yang, X., Wang, S., and Cai, H. (2025). Exploring RAG-based Vulnerability Augmentation with LLMs. ***Under Review***
- Daneshvar, S., Tan, D., Wang, S., and Leung, C. (2025). Representation-level Augmentation for Vulnerability Detection? Not Quite, Use Random Oversampling! Under Review. ***Under Review***

Chapter 2

Background and Related Work

As noted, representation-level augmentation has been explored briefly for related software engineering tasks, but none has been done for vulnerability detection. In this section, we cover the three general methods for vulnerability detection, vulnerability generation, sampling techniques used for vulnerability detection, the representation-level augmentation methods used by recent works, plus the related and required works and prerequisites.

2.1 LLM4SE and In-context learning

Given the success of LLMs for Natural Language Processing (NLP) tasks, researchers began to use them for software engineering problems. This led to the emergence of Code-aware LLMs which are pre-trained on code, namely CodeBERT [13], CodeT5[52], and many more. These LLMs could then be used for downstream tasks via transfer learning, which would involve fine-tuning these models for a specific

task, such as Vulnerability Detection [15].

On the other hand, LLMs can be utilized differently via In-context Learning also known as few-shot prompting, where zero or a few examples are given as a prompt in the form of natural language to an LLM plus a query that describes the task. Gao et al. [17] have shown that the order and similarity of such selected examples are important when we are using LLMs for software engineering tasks in a few-shot setting. Nong et al. [36] have leveraged LLMs to fix vulnerabilities with the help of in-context learning and Chain of Thought (CoT) prompting, which adds intermediate reasoning steps to the given prompt to increase the LLM's reasoning capability. Retrieval Augmented Generation (RAG) is another prompt engineering technique, where an information retrieval component is used alongside the LLM so that the LLM can do better when the given task is very knowledge-intensive. In a recent study, Wang et al. [51] used RAG for the task of program repair.

We believe that LLMs have shown extraordinary performance in previous related studies. We can use them to generate vulnerabilities to tackle the data shortage problem in vulnerability detection.

2.2 Vulnerability Detection

Based on the types of features that can be extracted from code, and how these features are transformed into a representation vector, Deep learning-based vulnerability detection methods can be classified into three categories of Token-based and Graph-based models, as well as LLM-based.

2.2.1 Token-based Vulnerability Detection

In this class of models [15; 26], code is treated as a sequence of tokens and is represented as a vector with the use of text embedding techniques, such as Word2Vec [32]. LineVul [15], for instance, is a recent token-based model that uses CodeBERT [13] for the embedding of functions into the representation vector. CodeBERT is an encoder-only transformer model that was pre-trained on a huge dataset of various programming languages.

2.2.2 Graph-based Vulnerability Detection

Graph-based methods [5; 62; 50; 25] still use the represented semantic information in the sequential tokens, but they also use the graph representation of code and use Graph Neural Networks(GNNs) [55] to use the data for training. These graph representations can be created by using the abstract syntax tree (AST), code property graph (CPG), or Program Dependency Graph (PDG). Two of the most recent but frequently studied graph-based models are Devign [62] and Reveal [5], where the latter is based on the former.

2.2.3 LLM-based Vulnerability Detection

With the rise of LLMs in recent years, researchers have explored the effectiveness of prompt-based vulnerability detection with LLMs [61; 28; 11; 16; 8; 47]. These works show that LLMs are capable of outperforming SOTA methods when RAG and in-context learning are used in synergy.

2.3 Vulnerability Generation

Vulnerability generation refers to the generation of vulnerable samples that resemble real-life code with vulnerabilities and is different from vulnerability synthesis where the generated vulnerable code may not be close to real-life code or even compile. Vulnerability Generation is a hard task as there's no metric for labeling a piece of code as vulnerable, and hence it requires manual checking of the generated code by experts, hence no one has attempted to generate vulnerabilities until very recently. Nong, et al. [35; 37] for the first time, proposed a solution for generating single-statement vulnerabilities using clustering techniques to mine the data and extract the vulnerable samples, and using CodeT5 [52] for finding the most suitable line for the injection of the mined vulnerability. Their approach is impractical as they only generate vulnerable code with single statement vulnerabilities, and the fact that checking whether the generated sample is vulnerable is done manually by security experts. Nong et al. [34] investigated vulnerability injection through a deep learning-based code editing model [58], which is trained to transform clean code into vulnerable code via a set of changes to the AST of the program. This approach needs high-quality datasets and hence suffers from a chicken-egg dilemma and has limited use.

2.4 Sampling Techniques

Sampling techniques are the most common way to deal with data imbalance. Yang, et al. [57] leveraged different sampling techniques to balance the data, and compared the effect of different sampling techniques for different vulnerability detection models

and datasets. They found that oversampling in general beats undersampling, and the naive random oversampling method performs best. They also explored the effect of such methods both at the latent level and on raw data and found that sampling raw data gives better results. Random oversampling is useful, but it does not add any new information to the model as it creates duplicates from the minority class; therefore, we need advanced augmentation techniques to deal with the data imbalance problem more effectively and generate useful data that can be used by the model.

2.5 Representation Level Augmentation

Representation level augmentation hasn't been explored in great detail in the field of software engineering, and to the best of our knowledge, the most recent work that incorporates all representation-level augmentation methods is Li et al.[23] where they explored 2 already known methods plus 3 new methods for augmenting data for the task of code search. All of these methods can be shown with the general format of Equation 2.1, where H is the augmented data, α and β are coefficients determined by the method, h and h' are two pieces of data in their representation form.

$$H = \alpha \odot h + \beta \odot h' \tag{2.1}$$

- Linear Interpolation: In this method $\alpha \in (0,1)$, sampled from a uniform distribution and $\beta = 1 - \alpha$, and $h \neq h'$. This method is sometimes called Mixup as well.

- Stochastic Perturbation: In this method $h' = 0$, α is a mask sampled from

the Bernoulli distribution and β is the complement of α. In simple terms, this method sets random tokens as 0. This method is usually implemented using Dropout, which multiplies unchanged items by a ratio so that the expected value of the vector wouldn't change.

- Linear Extrapolation: Similar to Linear Interpolation, but $\alpha \in (0,\ 1 + \epsilon)$ and $\epsilon > 0$

- Binary Interpolation: Similar to Stochastic Perturbation, but $h \neq h'$. This method is also similar to Linear Interpolation but α can only take either values of 0 or 1.

- Gaussian Scaling: In this method $h = h'$, $\alpha = 1$, and β is sampled from a Gaussian distribution centered at 0 with a small value of σ

2.6 Source Code Augmentation with RAG and LLMs

Using LLMs for augmenting source code is getting more popular and it has been used for augmenting data for Semantic Code Search [53] and Code Generation [7], both of which heavily use RAG. Wang et al. [53] utilizes RAG to retrieve similar $query - code$ pairs, then for each pair ChatGPT [38] is prompted to change the query and code in separate prompts using static rules written inside prompts. Then, the augmented pairs are filtered using an encoder model that calculates the similarity score between them to remove the low-scoring pairs. Chen et al. [7] used Code Search to augment data for Code Generation. Particularly, they retrieve pairs of $context - function$ snippets, where context can be a function header or a comment

to populate prompt templates that ask the LLM to generate code based on the given context and retrieved function.

Chapter 3

Problem Statement

As mentioned earlier, DLVD suffers from data shortage, especially in vulnerable samples. This also leads to heavily imbalanced datasets that may affect the training of such models. BigVul [12] is one of those datasets that is more imbalanced compared to datasets provided by Devign [62] and Reveal [5] papers. Hence, more vulnerable samples are needed to deal with these problems. Random oversampling (ROS) as evaluated by Yang et al. [57] is a good method for balancing the datasets, yet it does not add anything to the knowledge of the model and improves performance by a certain amount which interestingly all SOTA methods cannot beat! Hence, the need for an effective vulnerability augmentation technique that can beat ROS is deeply felt.

The previously mentioned augmentation methods were used for the task of code search by Li et al.[23], which by nature is very different from vulnerability detection as code search is an information retrieval task and data is labeled as relevant or irrelevant to other pieces of data. These augmentation methods can easily be leveraged to

augment data for information retrieval tasks as we can label all the generated data as irrelevant. In vulnerability detection, however, generating data blindly and using that data for training can heavily confuse the model as the augmentation process is not as straightforward and we will have difficulty labeling a newly generated data point when obtained via mixing a vulnerable and a non-vulnerable piece of code. Aside from the labeling complexity, we show that even if we augment data using the same labels, we still wouldn't get better results compared to random oversampling (ROS) since blind augmentation may remove tokens in the data that might have been useful for the correct prediction of the model.

In our work, we explore the effectiveness of the aforementioned representation-level augmentation methods for the task of vulnerability detection and show their limited capability for this task. Also, since such augmentation methods do not generate considerable improvement, we proposed a different method, specially designed for augmenting vulnerable code, which leverages RAG and LLMs to generate vulnerable code snippets. We show that generating vulnerabilities with LLMs can easily beat representation-level augmentation techniques, plus the previous vulnerability generation techniques.

Chapter 4

Representation-level Augmentation

Representation-level augmentation has been studied widely in other SE tasks but not for vulnerabilities, even though such methods are very easy to use and implement and are not resource-consuming. The only related study [57] to that, is on the use of sampling techniques; therefore, this easily motivates us to explore such methods and evaluate their effectiveness. Hence, this chapter targets finding the answer to the following research question: "Can Representation-Level Augmentation outperform Random Oversampling (ROS)? Is it effective at all?".

4.1 Experiment Design

We made a few key decisions in applying representation-level augmentation for vulnerability detection. Firstly, we chose to only generate vulnerable samples, given the substantial number of clean samples in datasets. Secondly, we opted to perform the augmentations before training. This approach gives us access to the entire dataset,

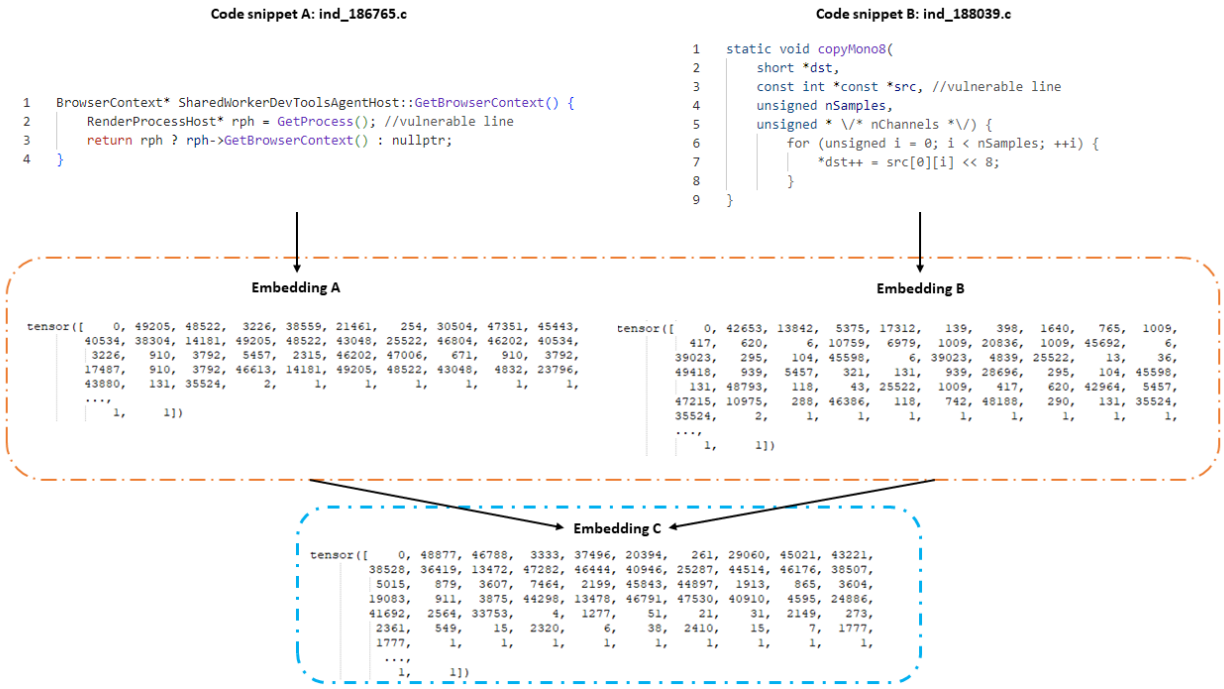

Figure 4.1: Blind augmentation of vulnerabilities using Linear Interpolation.

allowing the generation of more diverse samples. In contrast to Li et al. [23], we did not use in-batch augmentation during training. In-batch augmentation is commonly used in contrastive settings, where negative samples are created via augmentation.

Blind augmentation: We refer to applying the 5 augmentation methods, introduced in Chapter 2, to vulnerable samples of the BigVul dataset as Blind Augmentation. This is to emphasize that the vulnerable section in each vulnerable section may be overwritten or removed.

Conditioned augmentation: Aside from blindly applying 5 augmentation methods, we investigate the effect of fixing the vulnerable sections (i.e. conditioned augmentation) while using these augmentation methods to see if preventing the vulnerable section from being altered can increase the effectiveness of such methods.

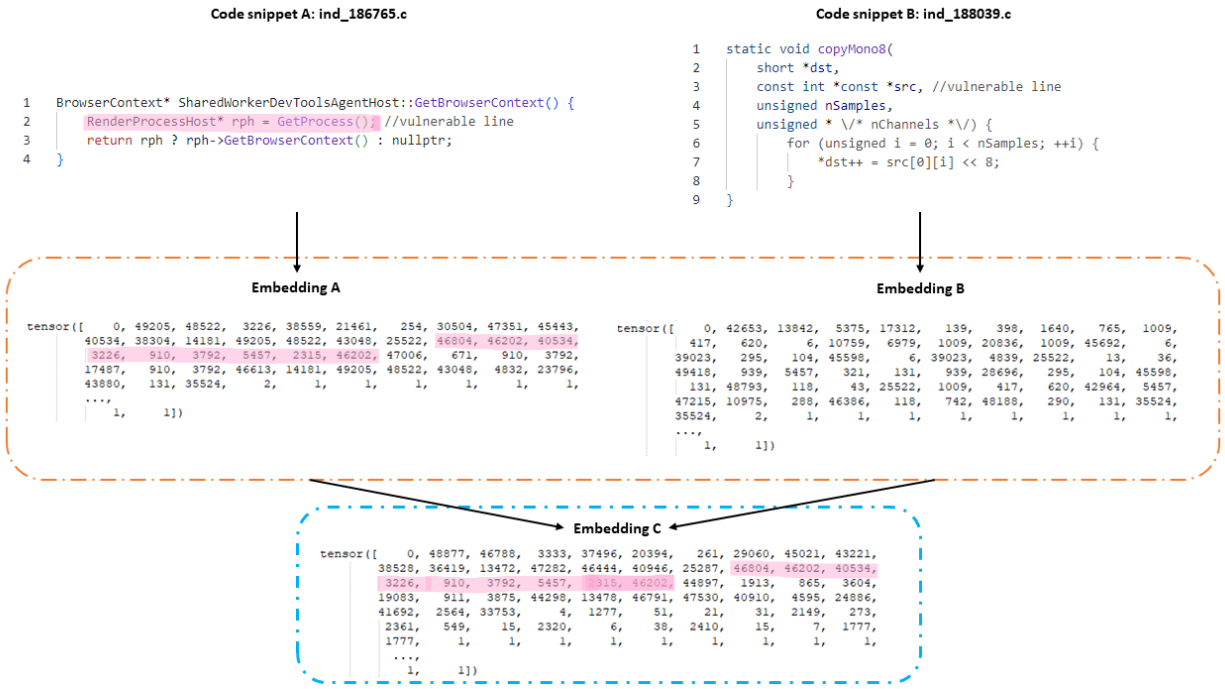

Figure 4.2: Conditioned augmentation of vulnerabilities using Linear Interpolation. The vulnerable line and the corresponding tokens are highlighted.

This involves excluding the tokens in the embedding corresponding to the specified vulnerable statements within the dataset from the augmentation process.

For **Linera Interpolation (LI)** we used the uniform distribution $\alpha \sim U(0.9, 1.0)$ for sampling. Figure 4.1 shows an example of this method with the sampled α set to 0.95. For **Linear Extrapolation (LE)** we sample from $\alpha \sim U(1, 1.1)$. For **Stochastic Perturbation (SP)** we set $p = 0.1$. For **Binary Interpolation (BI)** we change 25% of cells, replacing them with those of another vulnerable sample. For **Gaussian Scaling (GS)**, we sample from the normal distribution $\beta \sim N(1, 0.1)$. We used the chosen augmentation method with a factor of 23 in all settings to get a balanced dataset. We also used random oversampling (ROS) as a baseline as it can be useful in the training of DLVD and is not easy to beat [57].

To evaluate different data augmentation methods, we used the Bigvul [12] dataset for training and testing Linevul [15]. Specifically, we used the original training set of Bigvul and augmented the vulnerable samples of the training set that contained the vulnerable line information to be a fair comparison between the naive and conditioned augmentation methods. We trained Linevul with mostly the default settings but in batches of 32 and a learning rate of 2e-5. We also used the validation set to choose the best-performing checkpoint during training for testing. To compare these methods we employed the four common evaluation metrics [41], namely recall, precision, F1-score, and Area Under The Curve (AUC), in line with previous studies [57; 26; 50; 44; 43].

4.2 Results

Finding 1: **Stochastic Perturbation performs better than other representation-level methods in both settings.** Table 4.1 and Table 4.2 illustrate the results of the 5 representation-level augmentation methods when applied blindly (without conditioning to the vulnerable lines) and conditioned, respectively. We see that the majority of these augmentation methods lead to performance improvements as they are applied blindly. Binary Interpolation is the only method that degrades the overall F1-score, and Stochastic Perturbation generates the highest gain (8.92%). We observe a similar trend in conditioned augmentation variants, where all methods achieved a performance gain compared with no augmentation, and Stochastic Perturbation performs the best among those representation-level augmentation approaches and achieves an improvement (9.96%) over No Augmentation in terms of F1-score.

Finding 2: **None of the representation augmentation methods can beat the ROS.** Specifically, when compared with the representation augmentation methods with both blind and conditioned settings, ROS outperforms all of them, with an improvement of 10.82% in terms of F1-score, compared with No Augmentation.

Table 4.1: Blind Augmentation Results. The ones outperforming No Augmentation are shown in blue, otherwise, shown in red.

Augmentation Strategy	**AUC**	**Recall**	**Precision**	**F1-score**
No Augmentation	83.97	28.17	45.31	34.74
Random Oversampling	84.71	33.92	44.69	38.5
Linear Interpolation	83.53	29.38	50.32	37.1
Linear Extrapolation	83.08	31.14	44.80	36.74
Stochastic Perturbation	83.26	33.09	44.91	38.1
Binary Interpolation	83.69	25.95	50.18	34.21
Gaussian Scaling	82.33	28.92	48.15	36.13

4.3 Discussion

4.3.1 Why does the conditioned Stochastic Perturbation reduce the performance?

As we investigate the mechanism of **Stochastic Perturbation**, we find that using this method to augment data is equivalent to using a DropOut layer in the network, which gets omitted one out of 23 times (the original data) it sees a vulnerable sample. However, in the conditioned setting, since the vulnerable tokens will be

Table 4.2: Conditioned Augmentation Results. The ones outperforming No Augmentation are shown in blue, otherwise, shown in red.

Augmentation Strategy	**AUC**	**Recall**	**Precision**	**F1-score**
No Augmentation	83.97	28.17	45.31	34.74
Random Oversampling	84.71	33.92	44.69	38.5
Linear Interpolation	83.47	30.95	46.39	37.13
Linear Extrapolation	83.72	30.77	48.19	37.56
Stochastic Perturbation	83.70	32.81	44.70	37.84
Binary Interpolation	83.49	31.33	44.36	36.72
Gaussian Scaling	83.43	30.58	47.08	37.07

reinstated after the DropOut, they will behave differently from the original DropOut's regularization effect as the expected value will no longer match that of the input. This is while other methods do not have such an effect in the naive setting (keeping the expected value unchanged), and keeping the vulnerable section proves to be useful for them.

4.3.2 Why none of the methods can beat ROS?

We know that minimal noise in the data can act as a regularizer for machine learning models [3]. Hence, if the data is very noisy, the learning process will be heavily limited. In the case of studied methods, all of these methods generate samples that contain numbers that do not necessarily represent anything meaningful, and so the generated sample is very noisy such that repeating the minority class without introducing any change will have a better effect on the learning process.

4.3.3 Summary

In summary, representation-level augmentation can be used to improve the performance of DLVD, yet none of the study methods seems to beat Random Oversampling. Hence, since clean code is abundant and can be found easily, a better alternative to using representation-level augmentation is to simply oversample vulnerable samples and add extra clean items if one desires to keep the ratio.

Chapter 5

RAG-based Vulnerability Augmentation with LLMs

LLMs have powerful code comprehension and generation ability, which could overcome the limitation of mixup-inspired methods, and generate code with higher integrity and more correct structure. Therefore, we designed an LLM-based RAG-enhanced vulnerability detection model and in this chapter, we explain how this works. We organize this chapter into four sections: methodology, experimental setting, results, and discussion.

5.1 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,

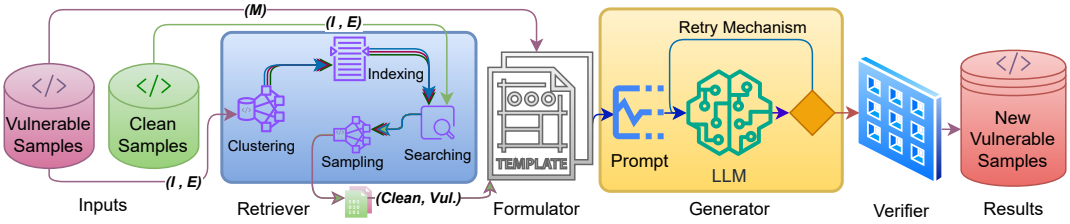

Figure 5.1: The components of VulScribeR. All of the proposed strategies share the same generation and verifier components. The Formulator component of the mutation strategy only requires the vulnerable samples and receives input directly. *(M, I, E)* tags distinguish the difference between the input data flows for **M**utation, **I**njection, and **E**xtension strategies, respectively.

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 5.1.1.

Figure 5.1 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 3 times to receive a response that contains a code snippet, otherwise, we move on to the next prompt.

- ***Verifier*** The point of the Verifier is to have control over the quality of the generated code as there's no guarantee that the LLM produces acceptable code. We use Joern's [42], a fuzzy C language parser, which is based on Antlr's [40] 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 [9; 60; 21; 24; 10]. 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. Completely correct code is not necessary for training models [54] and data with subtle noise helps with the generalization of models [3; 14; 22]. 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 results, 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 5.1.1.

5.1.1 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.

Mutation

The majority of previous works on program augmentation [59; 39; 31; 4; 10; 6], 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 [59; 39; 31; 4; 10; 6]. 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 5.2.

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)

Figure 5.2: The mutation prompt template.

We instruct the LLM to use one of the 18 program transformation rules by following a recent study [59] 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 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 5.1, 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.

Injection

Similar to recent works [37; 35; 34], 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 5.3.

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)

Figure 5.3: The injection prompt template.

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 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 [46; 19]. 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 [13] 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 [45], 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 [7; 18; 33]. 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 [30]. We set $G = 5$ in our study.

After the clean-vul pairs are retrieved, those pairs will be fed into Formulator to

Algorithm 1: (Clean, Vulnerable) Pair Retrieval

Input: V, C, N: Vulnerable samples, Clean Samples, 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

```
vul_embeddings ← compute_codebert_embedding(V), clustered_pairs ← [[]...] // G arrays
clustered_vuls ← KMeans(G, V, vul_embeddings) , indices ← []
D ← [] // Index vulnerable samples of each cluster separately
for cluster_id = 0 to G − 1 do
    search_engine_index ← index(clustered_vuls[cluster_id])
    indices[cluster_id].append(search_engine_index)
end for
// each clean item is paired with a similar vulnerable item of each cluster
foreach clean ∈ C do
    for cluster_id = 0 to G − 1 do
        clustered_pairs[cluster_id].append((clean, search(indices[cluster_id], clean)))
    end for
end foreach
// Sort each cluster's pairs based on their similarity score
for cluster_id = 0 to G − 1 do
    sort_by_score(clustered_pairs[cluster_id], "desc")
end for
// sort the arrays based on their size in descending order
clustered_pairs ← sort_arrays_by_size(clustered_pairs, "desc")
i ← 0 // Iterate through clustered_pairs and select the best pair each time
while size(D) <N do
    pairs ← clustered_pairs[i mod G] , index ← i//G, i ← i + 1
    if index < size(pairs) then
        D.append(pairs[index])
    end if
end while
return D
```

instantiate the **Injection** template and go through the rest of the components in the workflow to generate N vulnerable new samples similar to **Mutation**.

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 5.4.

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

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)

Figure 5.4: The extension prompt template.

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.

5.2 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.

5.2.1 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?*

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 affect the performance of DLVD models.

Table 5.1: The Studied Datasets

	Vul. Samples	**Clean Samples**	**Total**	**Ratio**
Devign	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

5.2.2 Datasets

In this study, we evaluate our approach on three different widely used vulnerability detection datasets [35; 37; 57; 15; 62; 5], namely Devign [62], Reveal [5] and BigVul [12]. Table 5.1 shows the details of these datasets. All of these datasets include C/C++ functions gathered from real-world projects.

Devign

Following previous studies [37; 35] we use Devign as our primary training dataset as it has a better balance between the two categories, namely the vulnerable and clean samples (i.e. non-vulnerable samples) by following a previous study, VulGen [35]. We also use the clean samples of this dataset as the clean inputs for the **Retriever** component.

Bigvul

We use all three sets (i.e., training, validation, and testing set) of BigVul [12] 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.

Reveal

Similar to VulGen [35], 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.

5.2.3 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 [15], Devign [62], and Reveal [5] models. For Devign and Reveal models, we use the same setting as VulGen [35] 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 [37], 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 [35] and VGX [37], to ensure there is no information leakage and the results can be trusted to apply to real-world unseen data.

5.2.4 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 [41], in line with previous related studies [57; 41; 26; 50; 44; 43; 35; 37], namely Precision, Recall, and F1-Score.

5.2.5 Base LLMs

In our study, we use two different LLMs to generate vulnerable code, namely ChatGPT3.5 [38], a commercial general-purpose LLM, and CodeQwen1.5 [49], an open-source LLM specialized for code tasks.

For ChatGPT, We use the "gpt-3.5-turbo", variant of OpenAI's ChatGPT [38] which has shown great potential in previous studies [53; 18; 7**?**]. 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 [49], "CodeQwen1.5-7B-Chat", which

is a variant of the general purpose Qwen [2]. 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).

5.2.6 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. [57] 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 [35] 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 [37] 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.

5.2.7 Approach for RQs

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 [37] 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 [27] to maintain the original ratio of the dataset, by following previous studies [35; 37]. 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).

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 cluster 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.

Approach of RQ3

Lastly, 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 5.3, 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 [37], VulGen [35], and ROS.

5.2.8 Implementation details

We use four 24GB Nvidia RTX 3090s for loading and prompting CodeQwen1.5, training, and testing LineVul, Devign, and Reveal models. We conducted a total of 616 experiments for all our RQs which cost more than 1500 GPU hours solely for training and testing DLVD models (excluding the time spent generating the samples).

Table 5.2: Comparison of VulScribeR's strategies with baselines using ChatGPT and CodeQwen when augmenting 5K Samples. The cells with larger values (better performance) compared to NoAug are highlighted darker.

		ChatGPT 3.5 Turbo									*CodeQwen1.5-7B-Chat*								
	Strategy	**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**

5.3 Results

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

Extension and Injection outperforms Mutation. Injection slightly outperforms Extension. Table 5.2 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.

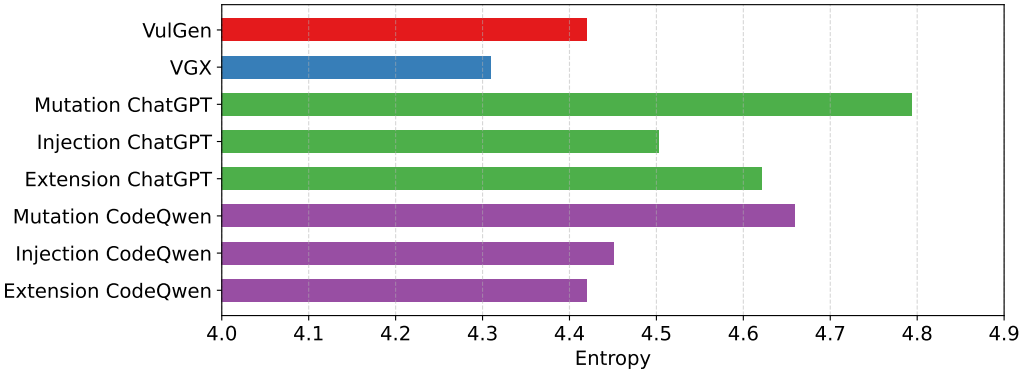

Figure 5.5: Entropy of the augmented vulnerable datasets by different approaches. Higher indicates more diversity.

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 5.2, **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 5.5 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 [1] and DeepSeek-Coder [20]) 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.

5.3.2 *RQ2: How does RAG contribute to VulScribeR?*

Table 5.3: Results of Ablation Studies on the Retriever and Clustering for Injection.

		ChatGPT 3.5 Turbo									*CodeQwen1.5-7B-Chat*								
	Strategy	**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 5.4: Results of Ablation Studies on the Retriever and Clustering for Extension.

		ChatGPT 3.5 Turbo									*CodeQwen1.5-7B-Chat*								
	Strategy	**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

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 5.3, 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 5.4, 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.

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

Augmenting more vulnerable data by using Injection helps improve the effectiveness of DLVD models. Table 5.5 and Figure 5.6 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.

Table 5.5: Impact of Generated Samples on Improving DVLD models' Performance at 5K, 10K, 15K, and 20K

		ChatGPT 3.5 Turbo									*CodeQwen1.5-7B-Chat*								
	Strategy	**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	54.28	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
	Injection 20K	16.74	49.28	24.99	16.91	57.90	26.18	16.97	17.80	17.38	15.11	55.97	23.79	15.31	58.69	24.28	13.92	17.26	15.41
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
	Injection 20K	13.41	29.89	18.51	12.54	38.95	18.98	11.91	25.49	16.23	9.55	34.02	14.92	11.60	39.59	17.94	7.87	38.00	13.05

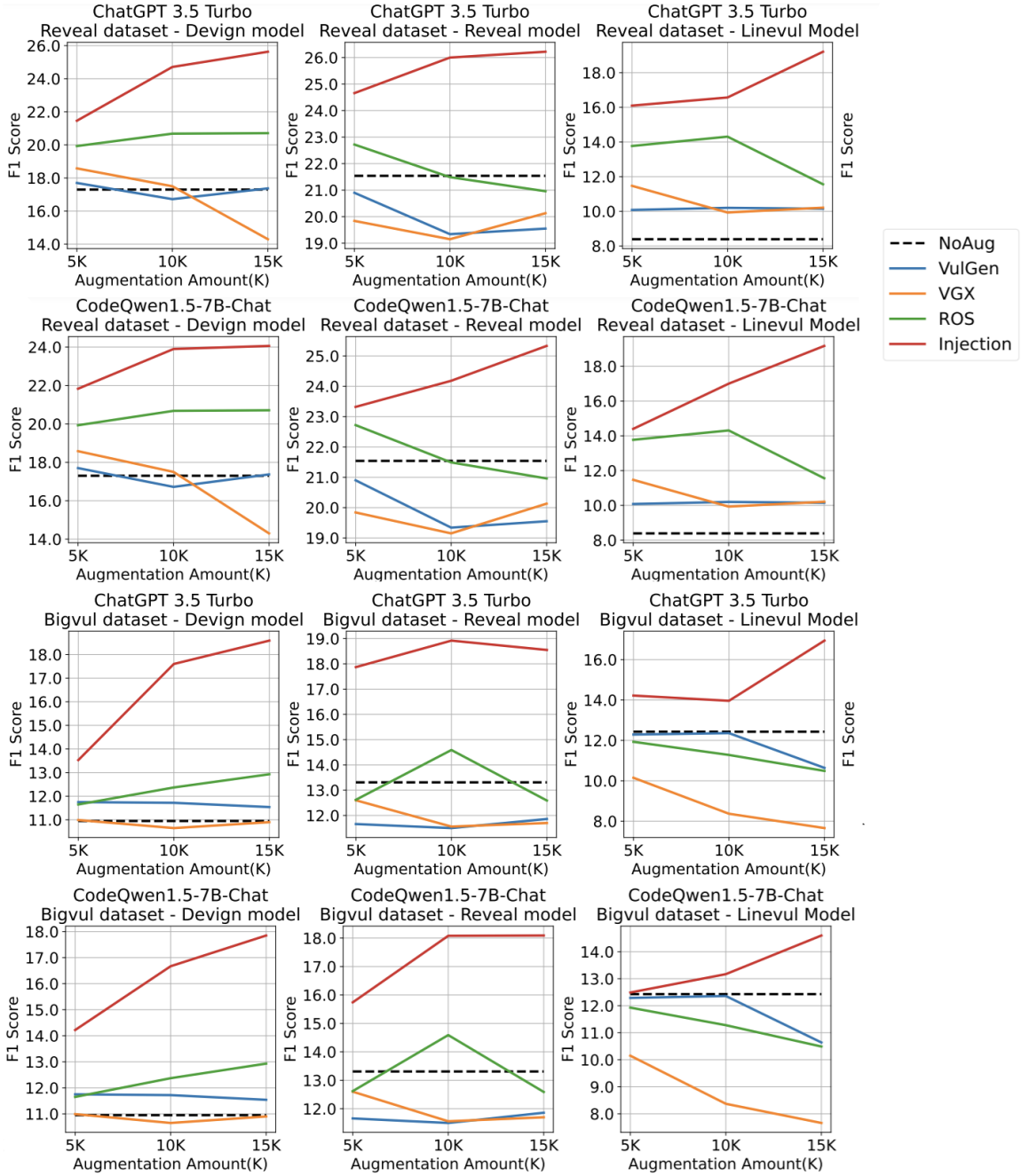

Figure 5.6: The impact of the number of augmented samples on the effectiveness of DLVD models across the studied datasets and LLMs. [1]

1

[1]Injection stops increasing the performance at 20K samples.

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 5.6, 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. [57], indicating that excessive random over-sampling does not improve 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 Discussion

5.4.1 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%.

5.4.2 Threats to Validity

Internal Validity A common threat to the validity of our study is our hyper-parameter settings for DLVD models and LLMs. For DLVD, hyper-parameter tuning is extremely expensive given the scale of our study, hence we followed the setting of previous studies [37; 35]. 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 hyper-parameters. 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). 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. Completely correct code is not necessary for training models [54] and data with subtle noise helps with the generalization of models [3; 14; 22]. 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.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

In this thesis, we looked at various representation-level techniques for vulnerability augmentation as well as our proposed model, VulScribeR. We showed that representation-level augmentation techniques can be useful for augmenting vulnerabilities, yet they cannot outperform random oversampling. We also showed that finding the vulnerable lines inside the representation and fixing those sections during augmentation can improve most methods, yet the best representation-level augmentation technique turned out to be the naive Stochastic Perturbation.

Moreover, we proposed our own method, VulScribeR, which uses a RAG mechanism to match similar clean and vulnerable samples and then asks an LLM to create a new vulnerable sample using 3 different strategies: Mutation, Injection, and Extension. We showed that the new strategy of Extension, which was never used in previous works, has potential while Injection appears to be the best strategy of VulScribeR. In

VulScribeR, since we introduced the first methodology ever to use an LLM for data augmentation of vulnerability augmentation, we decided to use the most simple yet effective tools. Particularly, we used KMeans for clustering, BM25 algorithm for our RAG module, and a simple parser for our verifier.

6.2 Future Work

It is worth noting that for both of the works presented in this thesis, further experiments can be done to evaluate the effectiveness of these methods from different aspects. For representation-level augmentation techniques, more models can be used for evaluation to make sure of generalizability. For VulScribeR, much more advanced RAG modules can be designed to retrieve closer samples, as ours treated code like text, while a more advanced RAG mechanism can take other factors into account, such as Code Property Graph, Abstract Syntax Tree, etc. Also, since VulScribeR is a good large-scale augmentation technique, better and more strict verifiers can be designed to make sure the generated items are of the highest quality.

Chapter 7

Data Availability

To encourage future research on data augmentation for vulnerability detection, we have made our replication package for representation-level mixup-inspired vulnerability augmentation as well as RAG-based vulnerability augmentation with LLMs, available here and here.

Bibliography

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[3] C. M. Bishop. Training with Noise is Equivalent to Tikhonov Regularization. *Neural Computation*, 7(1):108–116, 01 1995. ISSN 0899-7667. doi: 10.1162/neco.1995.7.1.108. URL `https://doi.org/10.1162/neco.1995.7.1.108`.

[4] N. D. Q. Bui, Y. Yu, and L. Jiang. 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, SIGIR '21, page 511–521, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380379. doi: 10.1145/3404835.3462840. URL https://doi.org/10.1145/3404835.3462840.

[5] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48:3280–3296, 2020. URL https://api.semanticscholar.org/CorpusID:221703797.

[6] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray. 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*, ESEC/FSE 2022, page 18–30, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549162. URL https://doi.org/10.1145/3540250.3549162.

[7] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639085. URL https://doi.org/10.1145/3597503.3639085.

[8] A. Cheshkov, P. Zadorozhny, and R. Levichev. Evaluation of chatgpt model for vulnerability detection, 2023. URL https://arxiv.org/abs/2304.07232.

[9] T. DeVries and G. W. Taylor. Improved regularization of convolutional neural networks with cutout, 2017. URL https://arxiv.org/abs/1708.04552.

[10] Z. Dong, Q. Hu, Y. Guo, Z. Zhang, M. Cordy, M. Papadakis, Y. L. Traon, and J. Zhao. Boosting source code learning with data augmentation: An empirical study, 2023.

[11] X. Du, G. Zheng, K. Wang, J. Feng, W. Deng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag, 2024. URL `https://arxiv.org/abs/2406.11147`.

[12] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375177. doi: 10.1145/3379597.3387501. URL `https://doi.org/10.1145/3379597.3387501`.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[14] B. Frenay and M. Verleysen. Classification in the presence of label noise: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 25(5): 845–869, 2014. doi: 10.1109/TNNLS.2013.2292894.

[15] M. Fu and C. Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620, 2022. doi: 10.1145/3524842.3528452.

[16] M. Fu, C. K. Tantithamthavorn, V.-A. Nguyen, and T. Le. Chatgpt for vulnerability detection, classification, and repair: How far are we? *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 632–636, 2023. URL `https://api.semanticscholar.org/CorpusID:264146048`.

[17] S. Gao, X. Wen, C. Gao, W. Wang, and M. R. Lyu. What makes good in-context demonstrations for code intelligence tasks with llms? *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 761–773, 2023. URL `https://api.semanticscholar.org/CorpusID:258180059`.

[18] S. Gao, X. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu. What makes good in-context demonstrations for code intelligence tasks with llms? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 761–773, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society. doi: 10.1109/ASE56229.2023.00109. URL `https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00109`.

[19] Z. Gong, P. Zhong, and W. Hu. Diversity in machine learning. *IEEE Access*, 7: 64323–64350, 2019. doi: 10.1109/ACCESS.2019.2917620.

[20] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL `https://arxiv.org/abs/2401.14196`.

[21] H. Guo, Y. Mao, and R. Zhang. Augmenting data with mixup for sentence

classification: An empirical study. *ArXiv*, abs/1905.08941, 2019. URL https://api.semanticscholar.org/CorpusID:162168620.

[22] R. J. Hickey. Noise modelling and evaluating learning from examples. *Artificial Intelligence*, 82(1):157–179, 1996. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(94)00094-8. URL https://www.sciencedirect.com/science/article/pii/0004370294000948.

[23] H. Li, C. Miao, C. Leung, Y. Huang, Y. Huang, H. Zhang, and Y. Wang. Exploring representation-level augmentation for code search. In Y. Goldberg, Z. Kozareva, and Y. Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4924–4936, Abu Dhabi, United Arab Emirates, Dec. 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.327. URL https://aclanthology.org/2022.emnlp-main.327.

[24] H. Li, C. Miao, C. Leung, Y. Huang, Y. Huang, H. Zhang, and Y. Wang. Exploring representation-level augmentation for code search. In Y. Goldberg, Z. Kozareva, and Y. Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4924–4936, Abu Dhabi, United Arab Emirates, Dec. 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.327. URL https://aclanthology.org/2022.emnlp-main.327.

[25] Y. Li, S. Wang, and T. N. Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, page 292–303, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468597. URL https://doi.org/10.1145/3468264.3468597.

[26] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. 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, 2018. URL https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf.

[27] G. Lin, W. Xiao, J. Zhang, and Y. Xiang. 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*, page 219–232, Berlin, Heidelberg, 2019. Springer-Verlag. ISBN 978-3-030-41578-5. doi: 10.1007/978-3-030-41579-2_13. URL https://doi.org/10.1007/978-3-030-41579-2_13.

[28] G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031, 2024. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2024.112031. URL https://www.sciencedirect.com/science/article/pii/S0164121224000748.

[29] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, and Y. Liu.

Chatgpt: Understanding code syntax and semantics. URL https://api.semanticscholar.org/CorpusID:258832612.

[30] Z. Ma, A. R. Chen, D. J. Kim, T.-H. Chen, and S. Wang. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[31] R. Mao, L. Zhang, and X. Zhang. Mutation-based data augmentation for software defect prediction. *Journal of Software: Evolution and Process*, 36(6):e2634, 2024. doi: https://doi.org/10.1002/smr.2634. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2634.

[32] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space, 2013.

[33] N. Nashid, M. Sintaha, and A. Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2450–2462. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00205. URL https://doi.org/10.1109/ICSE48619.2023.00205.

[34] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai. 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, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130.

[35] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai. Vulgen: Realistic vulnerability generation via pattern mining and deep learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2527–2539, 2023. doi: 10.1109/ICSE48619.2023.00211.

[36] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, and H. Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *ArXiv*, abs/2402.17230, 2024. URL `https://api.semanticscholar.org/CorpusID:268032799`.

[37] Y. Nong, R. Fang, G. Yi, K. Zhao, X. Luo, F. Chen, and H. Cai. Vgx: Large-scale sample generation for boosting learning-based software vulnerability analyses. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639116. URL `https://doi-org.uml.idm.oclc.org/10.1145/3597503.3639116`.

[38] OpenAI. Introducing chatgpt. `https://openai.com/blog/chatgpt`, 2022. Accessed: 2023-12-28.

[39] P. Orvalho, M. Janota, and V. Manquinho. 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*, ESEC/FSE 2022, page 1657–1661, New York, NY, USA, 2022. Association for Comput-

ing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3558931. URL https://doi.org/10.1145/3540250.3558931.

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

[41] D. Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011. ISSN 2229-3981.

[42] QwietAI. Joern: Code analysis tool. https://github.com/joernio/joern.

[43] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan. Impact of discretization noise of the dependent variable on machine learning classifiers in software engineering. *IEEE Transactions on Software Engineering*, 47(7):1414–1430, 2021. doi: 10.1109/TSE.2019.2924371.

[44] G. K. Rajbahadur, S. Wang, G. A. Oliva, Y. Kamei, and A. E. Hassan. The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering*, 48(7):2245–2261, 2022. doi: 10.1109/TSE.2021.3056941.

[45] S. Robertson and H. Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, apr 2009. ISSN 1554-0669. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

[46] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li. 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*, DAC '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342360. doi: 10.1145/2897937.2897995. URL https://doi.org/10.1145/2897937.2897995.

[47] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, and W. Le. A comprehensive study of the capabilities of large language models for vulnerability detection, 2024. URL https://arxiv.org/abs/2403.17218.

[48] N. M. S. Surameery and M. Y. Shakor. Use chat gpt to solve programming bugs. *International Journal of Information technology and Computer Engineering*, 2023. URL https://api.semanticscholar.org/CorpusID:257262719.

[49] Q. Team. Code with codeqwen1.5, April 2024. URL https://qwenlm.github.io/blog/codeqwen1.5/.

[50] W. Wang, T. N. Nguyen, S. Wang, Y. Li, J. Zhang, and A. Yadavally. Deepvd: Toward class-separation features for neural network vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2249–2261. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00189. URL https://doi.org/10.1109/ICSE48619.2023.00189.

[51] W. Wang, Y. Wang, S. R. Joty, and S. C. H. Hoi. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. *Pro-*

ceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023. URL `https://api.semanticscholar.org/CorpusID:261697451`.

[52] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL `https://aclanthology.org/2021.emnlp-main.685`.

[53] Y. Wang, L. Guo, E. Shi, W. Chen, J. Chen, W. Zhong, M. Wang, H. Li, H. Zhang, Z. Lyu, and Z. Zheng. You augment me: Exploring chatgpt-based data augmentation for semantic code search. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 14–25, 2023. doi: 10.1109/ICSME58846.2023.00014.

[54] Y. Wang, X. Li, T. N. Nguyen, S. Wang, C. Ni, and L. Ding. Natural is the best: Model-agnostic code simplification for pre-trained large language models. *Proceedings of the ACM on Software Engineering*, 1(FSE):586–608, 2024.

[55] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021. doi: 10.1109/TNNLS.2020.2978386.

[56] Z.-Q. Xie, Y.-L. Chen, C. Zhi, S. Deng, and J. Yin. Chatunitest: a chatgpt-

based automated unit test generation tool. *ArXiv*, abs/2305.04764, 2023. URL https://api.semanticscholar.org/CorpusID:258556880.

[57] X. Yang, S. Wang, Y. Li, and S. Wang. Does data sampling improve deep learning-based vulnerability detection? yeas! and nays! In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2287–2298, 2023. doi: 10.1109/ICSE48619.2023.00192.

[58] Z. Yao, F. F. Xu, P. Yin, H. Sun, and G. Neubig. Learning structural edits via incremental tree transformations. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=v9hAX77--cZ.

[59] S. Yu, T. Wang, and J. Wang. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2022.111304. URL https://www.sciencedirect.com/science/article/pii/S0164121222000541.

[60] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz. mixup: Beyond empirical risk minimization. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=r1Ddp1-Rb.

[61] X. Zhou, T. Zhang, and D. Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 47–51, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705007. doi: 10.1145/3639476.3639762. URL https://doi.org/10.1145/3639476.3639762.

[62] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper_files/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf`.