FINETUNING LARGE LANGUAGE MODELS FOR VULNERABILITY DETECTION

Aleksei Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Anton Cheshkov & Pavel Zadorozhny

Aleksei Shestov *
Sber AI Lab
Moscow 117997, Russia
AMShestov@sberbank.ru,shestovmsu@gmail.com

Rodion Levichev
Huawei Russian Research Institute
Moscow 121099, Russia
rodion.levichev@huawei-partners.com

Ravil Mussabayev †
Satbayev University
Almaty 121099, Kazakhstan
r.mussabayev@satbayev.university

Evgeny Maslov
Huawei Russian Research Institute
Moscow 121099, Russia
evgeny.maslov@huawei.com

Anton Cheshkov
Huawei Russian Research Institute
Moscow 121099, Russia
anton@huawei.com

Pavel Zadorozhny
Huawei Russian Research Institute
Moscow 121099, Russia
pavel.zadorozhny@huawei-partners.com

ABSTRACT

This paper presents the results of finetuning large language models (LLMs) for the task of detecting vulnerabilities in Java source code. We leverage Wizard-Coder, a recent improvement of the state-of-the-art LLM StarCoder, and adapt it for vulnerability detection through further finetuning. To accelerate training, we modify WizardCoder's training procedure, also we investigate optimal training regimes. For the imbalanced dataset with many more negative examples than positive, we also explore different techniques to improve classification performance. The finetuned WizardCoder model achieves improvement in ROC AUC and F1 measures on balanced and imbalanced vulnerability datasets over CodeBERT-like model, demonstrating the effectiveness of adapting pretrained LLMs for vulnerability detection in source code. The key contributions are finetuning the state-of-the-art code LLM, WizardCoder, increasing its training speed without the performance harm, optimizing the training procedure and regimes, handling class imbalance, and improving performance on difficult vulnerability detection datasets. This demonstrates the potential for transfer learning by finetuning large pretrained language models for specialized source code analysis tasks.

1 INTRODUCTION

Detecting vulnerabilities in source code is an important problem in software engineering. This project explores finetuning of LLMs for binary classification of Java functions as vulnerable or not. The goals are the following:

- Investigate the possibility of applying the new class of big LLM models for the task of vulnerability detection;
- Improve over previous results of CodeBERT-style models by using big LLM models;
- Investigate whether the encountered performance limit is due to the limited capacity of CodeBERT-like models.

---

*The work was done while the author was at the Huawei Russian Research Institute
†Corresponding author

Transformer models pretrained on large code corpus, like CodeBERT Feng et al. (2020) and Contra-BERT Liu et al. (2023a), have shown promising results on many tasks of code understanding. Line-Vul Fu & Tantithamthavorn (2022), a model built on CodeBERT, achieved state-of-the-art results on the vulnerability detection task, surpassing graph network approaches, like Reveal Chakraborty et al. (2022). Recent LLMs, which have billions of parameters, such as StarCoder Li et al. (2023) and WizardCoder Luo et al. (2023), are pretrained on trillions of tokens. They are 2 orders of magnitude bigger than CodeBERT (CodeBERT has approximately 125 million parameters) and have orders of magnitude more data used for pretraining. They achieve better results on language and code understanding tasks, providing an opportunity for improved transfer learning. The vulnerability detection task can benefit from the increased expressiveness and knowledge contained in state-of-the-art LLMs.

Vulnerability detection is a complex task with many aspects influencing and potentially limiting the solution quality. The problem of function-level vulnerability detection can be divided into two stages: representing the function in some form (such as execution paths or various graphs) and solving the classification problem for this representation Chakraborty et al. (2022); Zhang et al. (2023). It is noted that each stage can be independently improved, and improving each stage enhances the overall quality of the solution to the problem. Justifications to these observations include: the EPVD model Zhang et al. (2023), where improving the data structure with the same ML algorithm resulted in gains, and the LineVul model Fu & Tantithamthavorn (2022) compared to articles that present the function text to an LSTM. Therefore, it is relevant to improve each of the two stages.

We focus on improving the second part, the ML algorithm for vulnerability detection, without addressing the first stage. Our approach involves two directions: enhancing the model's capacity and improving the handling of imbalanced data, which represents real-world conditions. One of the limiting factors could be the capacity and code understanding ability of CodeBERT-like models, as vulnerability detection requires a deep understanding of many code aspects. Checking this hypothesis is a crucial problem in the field of vulnerability detection. Hence, we compare our classification of functions, represented as text, with the previous most powerful family of models, the CodeBERT-based models.

Furthermore, the study by Cheshkov et al. (2023) revealed that simple prompting techiques for the ChatGPT LLM were unable to achieve noticeably better results than random prediction on the vulnerability detection task. This finding suggests that finetuning of LLMs on this task naturally becomes a promising research direction.

The real-world distribution of vulnerable functions is highly class imbalanced, containing far more negative samples than positive ones. This imbalance makes the task inherently more difficult, as standard training objectives, like cross-entropy loss, may lead the model to focus on the dominant negative class. Specialized training techniques, such as focal loss Lin et al. (2017) and sample weighting, are required to properly emphasize the minority positive examples during finetuning.

This project conducts experiments to find the optimal model architecture, training hyperparameters, and loss formulations for effective LLM finetuning on the vulnerability detection task (both for balanced and imbalanced datasets). The aim is to leverage recent advances in pretrained models to improve over prior benchmark results.

The remainder of the paper is organized as follows:

- We discuss the most notable related work in the field;
- We describe the conducted experiments:
  - Comparison with ContraBERT (state-of-the-art non-LLM transformer model) on balanced and imbalanced datasets;
  - Ablation study of the context size and classification loss;
  - Investigation of the batch packing technique;
  - Improving the usage of a more informative part of the imbalanced dataset.
- We conclude the study, highlighting our contributions and stating promising future research directions.

In the appendices, we provide additional details:

- In Appendix A, we describe our approach for building and adapting the model. Specifically, we discuss how we solved the following problems:

    - Choosing the model;

    - Using limited GPU resources for the model size;

    - Adapting a decoder model for the classification task;

    - Underutilization of the input sequence length, resulting in a slow training speed.

- In Appendix B, we detail our main instruments: datasets, models, metrics, as well as measurement procedures.

2 RELATED WORK

Recent work by Zhou et al. Zhou et al. (2024) provides a comprehensive review of Large Language Models (LLMs) for vulnerability detection and repair. Their analysis reveals rapid growth in this field, with 27.8% of reviewed studies published in just the first two months of 2024. The majority of approaches (82%) employ fine-tuning techniques, while zero-shot (11%) and few-shot (7%) methods are less common. CodeBERT emerges as the dominant model for vulnerability detection, used in 38.5% of studies, with GPT-3.5 and GPT-4 following at 10.3% and 2.6% respectively. The review also highlights persistent challenges in the field, including class imbalance, noisy data, and scarcity of labeled datasets for training LLMs in vulnerability detection. Our work addresses these challenges by focusing on fine-tuning WizardCoder for Java vulnerability detection, contributing to this rapidly evolving research area.

Despite the successes of GPT models in solving various code-related tasks, their ability to detect vulnerabilities in code remains insufficient. The report "Evaluation of ChatGPT Model for Vulnerability Detection" (Cheshkov et al., 2023) notes that ChatGPT and GPT-3 models, without additional tuning and training, show results no better than random guessing in vulnerability classification. Similar conclusions are drawn in the study "ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?" (Michael Fu et al.) (Fu et al., 2023), where comparisons with specialized models such as CodeBERT and GraphCodeBERT reveal a significant lag of ChatGPT in vulnerability-related tasks.

Another notable example is the study by Liu et al. (2023) (Liu et al., 2023b), where the GPT model was enhanced using retrieval-augmented generation (RAG). Techniques such as BM-25 and TF-IDF were employed to retrieve relevant information, which was then used as examples for the few-shot learning technique. Using the Devign dataset, the study demonstrated significant improvements in precision, recall, and F1 score compared to traditional methods.

Recent work by Purba et al. Purba et al. (2023) explores the application of large language models (LLMs) for software vulnerability detection in C and C++ code. They evaluate four LLMs, including fine-tuned and zero-shot models, on SQL injection and buffer overflow vulnerabilities using the Code Gadgets and CVEfixes datasets. Their findings reveal that while LLMs demonstrate high recall in identifying vulnerable code patterns, they suffer from high false positive rates. This aligns with our observations on the challenges of using LLMs for vulnerability detection. However, our study extends this approach to Java, a domain where LLM-based vulnerability detection remains underexplored. We focus specifically on fine-tuning the state-of-the-art WizardCoder model for Java code analysis and introduce novel techniques to improve training efficiency and handle class imbalance. Unlike Purba et al., who primarily use prompt-based approaches, we explore more advanced fine-tuning methods and provide a detailed analysis of the model's performance on Java-specific vulnerability types, contributing to the limited body of work on LLM-based vulnerability detection for Java.

3 RESEARCH QUESTIONS

We refer the reader to Appendices A and B, where all the details about the used or suggested methodology, datasets, models, and metrics are provided.

3.1 COMPARISON TO CODEBERT-BASED MODELS.

RQ1: Is a StarCoder-based model more effective than a CodeBERT-based for the balanced vulnerability detection task?

The models were finetuned on the dataset without easy P_3 negatives, using different hyperparameters like the batch size, learning rate, and number of epochs. Optimal settings for this dataset were determined.

RQ2: Is a StarCoder-based model more effective than CodeBERT-based for the imbalanced vulnerability detection task?

Different training recipes, like the number of epochs and loss functions, were compared between the datasets with and without P_3 in order to analyze the transferability of hyperparameters.

3.2 ABLATION STUDY.

RQ3: Is the standard LLM training approach effective for vulnerability detection?

We formulate the vulnerability detection task as a question answering problem where the model predicts the next token in addition to a binary vulnerability label. The goal is to leverage the standard training approach for LLM decoder models, applying it to our classification task. Training utilizes a cross-entropy loss for both the token prediction and vulnerability classification objectives.

Batch packing properties. Batch packing enables dynamic batch sizes but may cause unstable gradients. The key questions in this direction are:

- **RQ4: Does the dynamic batch size harm the model quality or not?**
- **RQ5: Does the mean or the sum loss reduction perform better?**

RQ6: Does an increased context size provide improvements in quality?

To determine sensitivity to context, the model was trained with reduced input sequence lengths of 512 tokens instead of 2048 on the dataset without the P_3 part. Then, the results were compared in order to analyze impact.

3.3 IMBALANCED CLASSIFICATION IMPROVEMENT.

RQ7: Is the focal loss with sample weighting effective for tackling the vulnerability detection class imbalance problem?

On the full dataset, focal loss and sample weighting techniques were applied to emphasize the minority positive examples from P_1. The gamma parameter and sampling weights of the focal loss were systematically varied to ensure the performance.

For all experiments and analyses, the main evaluation metrics were ROC AUC, F1 score, accuracy, and optimal classification threshold. The goal was to make finetuning possible for classification, as well as to determine the best practices for finetuning on an imbalanced task.

4 EXPERIMENTAL RESULTS

4.1 COMPARISON TO CODEBERT-BASED MODELS

RQ1: Is a StarCoder-based model more effective than a CodeBERT-based for the balanced vulnerability detection task?

We compared the Starcoder-based model with the CodeBERT-based model on the dataset X_1 without P_3. This dataset is a balanced dataset without easy negative examples.

The methodology is the following: we take a pretrained model, manually optimize its learning hyperparameters using the training and validation datasets, and then determine the quality of the

best model on the test dataset. The optimized hyperparameters include the batch size, learning rate and number of epochs.

The results are presented in Table 1. We see that **the finetuned WizardCoder surpasses finetuned ContaBERT both in ROC AUC and F1 metrics on the balanced vulnerability detection task.** This superior performance of WizardCoder can be attributed to its larger model capacity of 13 billion parameters and pretraining on a much larger code corpus.

Base model	ROC AUC	F1
ContraBERT	0.66	0.68
WizardCoder	0.69	0.71

Table 1: ROC AUC and F1 scores for finetuned ContraBERT and WizardCoder models on the X_1 without P_3 dataset.

To finetune the WizardCoder model, several important hyperparameters were identified: a batch size of 1 (the actual batch size becomes approximately 120 by packing functions in the batch), learning rate of 0.0001, and 50 epochs of training with cosine annealing schedule. The best checkpoint occurred at epoch 19, indicating sufficient training time is needed.

The number of epochs for the cosine schedule is crucial, as it controls the learning rate distribution across epochs and greatly impacts training regimes.

RQ2: Is a StarCoder-based model more effective than CodeBERT-based for the imbalanced vulnerability detection task?

We compared the Starcoder-based model with the CodeBERT-based model on the X_1 with P_3 dataset. This dataset is an imbalanced dataset with majority of samples belonging to easy negative examples. This dataset is closer to the real-world vulnerability distribution.

The methodology of comparison is the same as in RQ1. The optimal training hyperparameters appeared to be the same for both datasets (X_1 with and without P_3). The identical optimal settings imply that the models train similarly on both datasets.

The results are presented in Table 2. We see that **finetuned WizardCoder surpasses finetuned ContaBERT both in ROC AUC and F1 metrics on the imbalanced vulnerability detection task. However, achieved gains over ContraBERT in ROC AUC score are smaller compared to the X_1 without P_3 dataset**. A potential reason is underutilization of the more informative P_1 and P_2 partitions on this dataset.

Base model	ROC AUC	F1
ContraBERT	0.85	0.22
WizardCoder	0.86	0.27

Table 2: ROC AUC and F1 scores for finetuned ContraBERT and WizardCoder models on X_1 with P_3 dataset.

4.2 ABLATION STUDY

RQ3: Is the standard LLM training approach effective for vulnerability detection?

The WizardCoder model was finetuned on the vulnerability dataset by formulating the task as a question answering problem. Specifically, the model was trained to predict the next token in addition to the binary vulnerability label on the X_1 with P_3 dataset. The results are presented in Table 3.

The next token prediction approach achieved the ROC AUC score of 0.75 for WizardCoder on the X_1 with P_3 dataset. **This result is inferior to CodeBERT and WizardCoder models trained with classification-only objectives (0.85 and 0.86, respectively), but still surpasses random performance.**

Batch packing properties. The batch packing approach introduces some unique properties:

Base model	Training approach	ROC AUC
WizardCoder	Next token prediction	0.75
CodeBERT	Binary classification	0.85
WizardCoder	Binary classification	0.86

Table 3: ROC AUC Scores for finetuning with the next token prediction and classification objectives on the imbalanced X_1 with P_3 dataset.

- The actual batch size becomes dynamic during training as sequences are packed;
- Using the standard mean loss reduction gives different functions unequal influence on the gradient, which may harm the overall performance;
- Using the sum loss reduction training steps unevenly, which may also be harmful to performance.

This raises two research questions:

- **RQ4: Does the dynamic batch size harm the model quality or not?**
- **RQ5: Does the mean or the sum loss reduction perform better?**

RQ4: Does the dynamic batch size harm the model quality or not?

The batch packing approach actually introduces the dynamic batch size during training. This dynamic batch size allows to decrease the training time by up to 13 times. Dynamic batch size is not a standard approach for training neural networks, so the following question arises: does it sacrifice the model quality for training speed improvement? In order to answer this question we tried a variant of batch packing that limits the maximum number of functions in a batch. This limiting decreases the dispersion of the actual batch sizes distribution.

We tested two different values for the maximum number of functions per batch: 50 and 100 on the validation part of the X_1 without P_3 dataset. Our results are present in Table 4. The results imply that there is no significant influence of limiting the maximum number of functions per batch, indicating that dynamic batching may not have a detrimental effect on the model's performance.

Max. functions per batch	Validation ROC AUC
No limit	0.72
50	0.72
100	0.72

Table 4: WizardCoder's results with limiting the maximum number of functions per batch on the validation part of the X_1 without P_3 dataset.

In summary, dynamic batch size works well empirically and limiting it does not lead to any improvements.

RQ5: Does the mean or the sum loss reduction perform better?

Each kind of loss reduction has arguments against it:

- Using the standard mean loss reduction gives different functions unequal influence on the gradient, which may harm the overall performance;
- Using the sum loss reduction training steps unevenly, which may also be harmful to performance.

Thus, it becomes important to investigate whether the mean or sum loss reduction is better suited for the batch packing approach. In order to answer this question, we performed testing on the validation part of the X_1 without P_3 dataset. The results are present in Table 5. The mean loss reduction performed poorly, highlighting issues with the mean loss in comparison with the sum reduction.

The superior performance of the sum loss reduction indicates that it better handles the uneven sequence lengths arising from the batch packing. By summing over the packed sequences, each function contributes equally to the gradient.

Loss reduction	Validation ROC AUC
Mean	0.57
Sum	0.72

Table 5: WizardCoder's performance by a loss reduction method on the validation part of the X_1 without P_3 dataset.

In summary, the sum loss reduction method outperforms the mean one by better handling uneven sequence lengths.

RQ6: Does an increased context size provide improvements in quality?

An open question was whether the quality improvements obtained by WizardCoder stemmed from the larger 2048 context size versus the 512 baseline, or better code understanding by the model.

We conducted an experiment to isolate the impact of context size. The 2048 context size Wizard-Coder model was compared to the 512 context size version on the test part of the X_1 without P_3 dataset.

Context size	Test ROC AUC
512 Tokens	0.69
2048 Tokens	0.69

Table 6: The impact of the WizardCoder's context size on performance.

Table 6 shows that **reducing the context size from 2048 tokens to 512 tokens resulted in the identical validation ROC AUC score of 0.69 for WizardCoder. This suggests the performance gains are mainly due to the model's learning of improved code representations, rather than the increased context size.**

4.3 IMBALANCED CLASSIFICATION IMPROVEMENT

RQ7: Is the focal loss with sample weighting effective for tackling the vulnerability detection class imbalance problem?

The obtained gains of the StarCoder-based model over the CodeBERT-based on the imbalanced X_1 with P_3 dataset are minor compared to the X_1 without P_3 dataset. A potential reason is underutilization of the more informative P_1 and P_2 parts on this dataset. In order to explore the influence of better utilization of P_1 and P_2 parts on the model's quality, we conducted experiments that incorporate the focal loss and sample weighting techniques:

- Focal loss emphasizes hard and rare examples contained in the P_1 and P_2 parts;
- Sample weighting also focuses model on the minority cases from the P_1 and P_2 parts.

Focal Loss. Focal loss Lin et al. (2017) applies a modulating factor $(1 - p_t)^\gamma$ to the standard cross entropy loss. This factor down-weights well-classified examples and focuses training on hard misclassified cases. The standard cross-entropy loss corresponds to $\gamma = 0$.

We tested γ values from 1 to 5 on the validation part of the X_1 with P_3 dataset. The results are presented in Table 7.

	$\gamma = 0$	$\gamma = 1$	$\gamma = 3$	$\gamma = 5$
ROC AUC	0.858	0.873	0.868	0.852
F1	0.277	0.265	0.272	0.250
Best val. epoch	12	12	17	11
Best threshold	0.087	0.288	0.265	0.4

Table 7: Resulting quality of WizardCoder for different γ values in focal loss on the imbalanced X_1 with P_3 dataset.

In summary, the focal loss with $\gamma = 1$ achieves a small improvement in ROC AUC over the standard cross entropy loss ($\gamma = 0$). However, the gains are small.

The peak quality at $\gamma = 1$ followed by a decrease at higher γ values suggests the importance of balancing the emphasis on hard examples with retaining sufficient training signal from easier instances.

One potential reason for such small gains is that the noisy or imperfect labels limit the ability of the focal loss to accurately identify the most important hard examples to prioritize. If the label of an example does not precisely reflect its difficulty, a heavy focus on hard cases may be less effective.

The inconsistent improvements demonstrate the need for further exploration of methods to leverage hard examples without detriment to learning on easier cases, while taking the label quality into account.

An interesting observation is that the focal loss leads to better calibrated models, with thresholds closer to 0.5. This indicates that the predicted class probabilities are closer to the true values under the focal loss. The better calibration is a useful characteristic.

Adding sample weights. Sample weighting assigns higher importance weights to the P_1 and P_2 examples compared to P_3 during training. Weights from 1.0 to 30.0 were tried. Larger weights place more emphasis on the minority informative data. The similar technique was reported to be effective with the focal loss in the original article Lin et al. (2017).

The applied technique is different from the usual class weighting scheme since it puts more weight on both the samples of the positive (P_1) and negative (P_2) parts. In the previous experiments, we found that the class weighting technique was ineffective, so the sample weighting scheme provides a different perspective.

The numeric results are presented in Table 8.

Weight	Best val. AUC	Best epoch	Test AUC	Test F1	Threshold
30	0.876	6	0.863	0.22	0.42
10.0	0.878	13	0.86	0.24	0.44
3.0	0.88	12	0.875	0.265	0.07
1.0 (base)	0.87	12	0.858	0.277	0.08

Table 8: Resulting quality of WizardCoder for different values of $P_1 + P_2$ weights on the imbalanced X_1 with P_3 dataset.

From Table 8 we conclude that adding sample weights for the P_1 and P_2 partitions can provide small improvements in ROC AUC and F1 score over the baseline. However, large weight values degrade performance.

The best weighting scheme of $3x$ attains marginal gains in both ROC AUC and F1 over unweighted training. However, excessive weighting, like $30x$, hurts the validation and test metrics.

This indicates there is an optimal moderate weighting that slightly emphasizes the hard and rare P_1 and P_2 examples without skewing the distribution too heavily. Further tuning of the weighting factor may yield additional gains.

Combination of sample weights with focal loss Both the focal loss and additional weights do similar work. In the original work Lin et al. (2017), it was stated that adding weights to the focal loss leads to some improvements.

Therefore, we combined the focal loss with the sample weighting technique. The results are presented in Table 9.

Experiments with the focal loss and sample weighting demonstrated minor improvements over the baseline training, with the best model achieving 0.877 ROC AUC versus 0.86 for the baseline. However, neither technique provided substantial gains. More advanced methods that can explicitly account for a severe class imbalance are needed.

Weight	γ	Val. AUC	Test AUC	Test F1
3	1.0	0.88	0.877	0.273
10	3	0.872	0.853	0.226
10	1	0.877	0.863	0.243
3	3	0.874	0.865	0.286
None	1	0.87	0.873	0.265
None	3	0.87	0.868	0.272

Table 9: The results of WizardCoder using the focal loss and the sample weighting on the imbalanced X_1 with P_3 dataset.

5 THREATS TO VALIDITY

There are some potential threats to validity that could limit the objectiveness of our study:

- The dataset is not split by projects, and splitting by projects might have resulted in a significant decrease in quality;

- When finetuning a model in the classification regime with batch packing, the model is able to see the functions that precede the current one in the input. This could limit the training effectiveness of the resulting model. However, the negative impact on training should be mitigated since these functions are placed randomly, so it would be hard for the model to learn irrelevant features. Also, during inference for test data, only one is function was considered in any single input sequence, so there is no bias in the test scores;

- The function-level granularity of our dataset limits the types of vulnerabilities that span across multiple functions;

- The dataset size is relatively small. This might limit its representativeness of the true vulnerability distribution.

Dividing data into training and test sets by project would provide a more objective measure of model performance, minimizing overlap between the sets. Addressing vulnerabilities that span multiple functions is outside this paper's scope, but presents a crucial direction for future research. Notably, these two limitations are common to most existing vulnerability detection studies in the literature, including those discussed in the "Related work" section.

6 CONCLUSION

Our work demonstrates the effectiveness of finetuning large language models for the vulnerability detection problem in source code. The WizardCoder model achieved state-of-the-art results, with the ROC AUC score of 0.69 on the dataset without easy negative examples. This improves over previous CodeBERT-based models, likely due to the WizardCoder's larger model capacity and pretraining corpus.

Several key contributions are made:

- An efficient batch packing strategy is developed to mitigate small sequence lengths, providing over $13x$ speedup in training time. This enables faster iteration and tuning;

- An improvement in ROC AUC from 0.66 to 0.69 was obtained over the state-of-the-art non-LLM model ContraBERT on the X_1 dataset without P_3 part. An improvement of the F1 score was obtained for the dataset with P_3 (0.27 vs 0.22);

- For the highly imbalanced dataset, it was shown that the focal loss with sample weighting gives improvements from 0.86 to 0.878. Despite these improvements, more advanced methods are needed to properly emphasize the minority positive examples;

- A new more precise vulnerability benchmark dataset is collected. It is smaller than most of the available datasets, as well as possesses quality labels that are free from the errors stemming from irrelevant code and cleanup changes Croft et al. (2023).

Opportunities remain for further improvements in this approach, mainly related to training on the dataset with the P_3 part included. Future work should explore techniques like curriculum learning, active sampling, and data augmentation to better leverage the scarce minority data. The insights gained can guide research on broader tasks involving code analysis and understanding.

We believe our work can be effectively generalized to programming languages beyond Java. Transfer learning from code LLMs' pretraining data ensures this generalizability with minimal additional fine-tuning. Additionally, specialized fine-tuning datasets for other target languages can be collected using commit-level vulnerability fixing data, similar to the approach described in this paper.

We show an improvement in quality of the vulnerability detection task by finetuning WizardCoder LLM model for the vulnerability detection task. Considering the quality needed for business tasks, there is still a gap between the achieved and required quality. This leaves us with a big room for improvement, particularly, in choosing the optimal representation of vulnerabilities and usage of a broader project-level context information. These are very exciting prospects for future research.

ACKNOWLEDGEMENTS

REFERENCES

Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, pp. 30–39, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386807. doi: 10.1145/3475960.3475985. URL https://doi.org/10.1145/3475960.3475985.

S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(09):3280–3296, sep 2022. ISSN 1939-3520. doi: 10.1109/TSE.2021.3087402.

Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. Evaluation of chatgpt model for vulnerability detection, 2023.

Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. Data quality for software vulnerability datasets. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pp. 121–133. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00022. URL https://doi.org/10.1109/ICSE48619.2023.00022.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

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

Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. Chatgpt for vulnerability detection, classification, and repair: How far are we? In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 632–636, 2023. doi: 10.1109/APSEC60848.2023.00085.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2999–3007, 2017. doi: 10.1109/ICCV.2017.324.

Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. Contrabert: Enhancing code pre-trained models via contrastive learning. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pp. 2476–2487. IEEE Press, 2023a. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00207. URL https://doi.org/10.1109/ICSE48619.2023.00207.

Zhihong Liu, Qing Liao, Wenchao Gu, and Cuiyun Gao. Software vulnerability detection with gpt and in-context learning. In *2023 8th International Conference on Data Science in Cyberspace (DSC)*, pp. 229–236, 2023b. doi: 10.1109/DSC59305.2023.00041.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct, 2023.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023.

Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, pp. 383–387. IEEE Press, 2019. doi: 10.1109/MSR.2019.00064. URL https://doi.org/10.1109/MSR.2019.00064.

Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 112–119, 2023. doi: 10.1109/ISSREW60843.2023.00058.

Shichao Wang, Yun Zhang, Liagfeng Bao, Xin Xia, and Minghui Wu. Vcmatch: A ranking-based approach for automatic security patches localization for oss vulnerabilities. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 589–600, 2022. doi: 10.1109/SANER53432.2022.00076.

J. Zhang, Z. Liu, X. Hu, X. Xia, and S. Li. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering*, 49(08):4196–4212, aug 2023. ISSN 1939-3520. doi: 10.1109/TSE.2023.3286586.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th*

ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD '23, pp. 5673–5684, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701030. doi: 10.1145/3580305.3599790. URL `https://doi.org/10.1145/3580305.3599790`.

Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ArXiv*, abs/2404.02525, 2024. URL `https://api.semanticscholar.org/CorpusID:268876353`.

A   IMPLEMENTATION DETAILS

A.1   CHALLENGES

We briefly describe the main milestones and challenges that we encountered during the implementation of our approach.

LLM model selection for finetuning   Recently, there has been a rise in the number of open-sourced LLM models, trained on large code corpora Li et al. (2023); Luo et al. (2023); Zheng et al. (2023); Nijkamp et al. (2023). The task is to choose an optimal model for finetuning, both from the perspective of quality and potential technical difficulties encountered. The goal is to select a model possessing the following properties:

- It shows a strong potential for effective transfer learning to the vulnerability detection task;
- It is technically easy to use and integrate into our framework.

Adapting LLMs for limited computational resources   Given our computing capabilities, finetuning complete LLM models is infeasible and would be inefficient even with increased resources. To enable effective finetuning of a 13-billion parameter model on our hardware, we need methods to reduce the number of trained parameters and memory requirements.

Adapting LLMs for the classification task   The standard pretraining task for LLM decoders is the next token prediction, which differs from our end task of vulnerability classification. Including the next token prediction component in the loss may misguide weight updates away from the optimal state for classification. To fully leverage the capabilities of the LLM weights for our target task, a better strategy might be to leave only the classification term in the loss.

Improving a slow training speed   The standard sequence classification approach of padding to a fixed length makes training extremely slow. To mitigate the issue of short sequence lengths, the task is to pack multiple functions into each training sequence. This can decrease unused padded tokens, improving computational efficiency.

A.2   CHOOSING AN LLM TO FINETUNE

Recently, there has been a proliferation of open-sourced LLM models trained on large code corpora Li et al. (2023); Luo et al. (2023); Zheng et al. (2023); Nijkamp et al. (2023). The task is to select an optimal model for finetuning.

First, we attempted to use the CodeGeeX model Zheng et al. (2023). Unfortunately, we found that using this model was very difficult from the technical perspective, as the code provided by the authors does not support standard libraries for transformer training and large model adaptation. We were able to run the model on 8 V100 GPUs for inference, but this memory capacity was insufficient for finetuning. The lack of support for common LLM adaptation libraries made it very challenging to overcome the memory limitations, so we decided to discontinue attempts with this model.

Next, we explored two other models, WizardCoder Luo et al. (2023) and CodeGen Nijkamp et al. (2023). In contrast to CodeGeeX, these models include support for major transformer training frameworks like DeepSpeed, HuggingFace Transformers, Peft Mangrulkar et al. (2022), etc. This compatibility with standard libraries significantly eased the application of low-memory adaptation techniques.

When conducting a simple Question Answering training, we find that WizardCoder is much better than CodeGen at directly answering "YES" and "NO". CodeGen sometimes responded with "TRUE", "FALSE", erratic "YES" and "NO" with additional characters, and has a worse sequence stopping behavior. Given the WizardCoder's stronger capabilities on this task, we focused our subsequent experiments on this model.

Choosing the best StarCoder-family model In our study, we evaluate three models from the StarCoder family for finetuning: StarCoderBase, StarCoder Li et al. (2023) (which is StarCoderBase finetuned on the Python dataset), and WizardCoder Luo et al. (2023) (which is an improved version of StarCoder trained using the eval-instruct technique). We compared the performance of StarCoder and StarCoderBase on the question answering task and found no significant differences between the two models, despite the fact that StarCoder was specifically adapted for Python. Subsequently, we primarily used the WizardCoder model, as it is claimed to be superior to StarCoder. However, our experiments did not reveal any significant differences in performance between these two models. In our future writing, we will use the term "StarCoder" to refer specifically to the WizardCoder variant of the StarCoder family.

Generally, more elaborate experiments are needed to find if there are any differences in the performance of these models.

A.3 ADAPTING LLMS FOR LIMITED COMPUTATIONAL RESOURCES

Finetuning complete multi-billion parameter LLMs is infeasible given our hardware constraints, even if more resources were obtained. To enable effective finetuning, we require techniques to reduce the number of trained parameters and memory needs. The HuggingFace Peft Mangrulkar et al. (2022) library implements several methods tackling this, including the LORA method Hu et al. (2022). LORA enables full-model finetuning by decomposing the weight matrices into low-rank approximations, drastically decreasing parameters and memory.

We successfully applied LORA to the WizardCoder model. We used the following optimal LORA settings: r = 8, alpha = 32, dropout = 0.05. This reduced the 13 billion parameter models to just 25 million parameters, smaller than CodeBERT. The results validate LORA's effectiveness for LLM adaptation under memory limitations.

A.4 ADAPTING LLMS FOR CLASSIFICATION

Typically, code LLMs are trained on a large amount of unlabeled code data in a special way. The principle is to iteratively take code tokens as input, predict the next token, and compare it with the ground truth. This principle drives the next token prediction loss. Specifically, for any input sequence $\{x_1, x_2, \ldots, x_N\}$ of length N, the output of an LLM is a probability distribution of the next token $P(x_{N+1}|x_1, x_2, \ldots, x_N, \Theta) = p_{N+1} \in [0, 1]^v$, where Θ represents all parameters of the model, and v is the vocabulary size. By comparing it with the real distribution, i.e., a one-hot vector $y_{N+1} \in \{0, 1\}^v$ of the ground-truth token, we can optimize the cumulative cross-entropy loss:

$$\mathcal{L}_{\text{ntp}} = -\sum_{n=1}^{N-1} y_{n+1} \log \mathbb{P}(x_{n+1}|x_1, x_2, \ldots, x_n, \Theta) \tag{1}$$

In the context of adapting a language model for classification tasks, the objective function used during training needs to be aligned with the classification goal rather than the next token prediction. For an input sequence $\{x_1, x_2, \ldots, x_N\}$ of length N, the traditional generative pretraining objective (next token prediction loss) would compute the loss across all predicted tokens in the sequence. However, this is suboptimal for a classification task such as vulnerability classification, which is concerned with categorizing the entire input sequence rather than predicting subsequent tokens.

To address this, we propose a classification loss that leverages only the predicted probability of the final token x_N, which is then matched against the ground truth label y using cross-entropy. This loss is expressed mathematically as:

$$\mathcal{L}_{\text{class}} = -\log \mathbb{P}(y|x_1, x_2, \ldots, x_N, \Theta) \tag{2}$$

Here, y represents the correct class label for the sequence, and $\mathbb{P}(y|x_1, x_2, \ldots, x_N, \Theta)$ is the probability that the model, parameterized by Θ, assigns to the correct class. This formulation ensures that weight updates during training are driven exclusively by the classification objective, without any influence from the generative task of next token prediction. The classification loss may be viewed as corresponding to the last term of the generative pretraining objective's (next token prediction) cross-entropy loss, focusing entirely on the classification output for $n = N$.

Eliminating the next token prediction objective enables full utilization of the pretrained weights for vulnerability detection, without interference from unrelated generation tasks.

A.5 Speeding up the classification training

The standard approach of passing code samples to the LLM by writing tokens into the context window and padding unused slots is problematic for our data. Most dataset functions have length under 50 tokens (see Table 10), so this padding wastes computation.

Function length (tokens)	Functions number
0-10	1976
10-20	31627
20-50	51972
50-100	32194
100-300	35769
300-500	8756
500-1000	5515
1000-2000	2236
>2000	767

Table 10: Function length histogram, obtained from parsing several projects

We see that most functions have less than 50 tokens, with many having 10-20 tokens. Using one function per batch element (padded to 2048) is highly redundant.

To mitigate the issue of short sequence lengths, we pack multiple functions into each training sequence. This increases the actual batch size versus using one function per sequence.

Thus, during our study we focus on the following two regimes of using code LLMs:

1. *Next token prediction.* This approach corresponds to using the next token prediction loss equation 1 during training and prediction. During training, an LLM's input sequence is packed with the code of training methods and their ground truth labels one after another in the following format:

   Method's code + YES / NO (binary ground truth label) + ⟨EOS⟩

   Here ⟨EOS⟩ denotes the special "end of sequence" token from the vocabulary. We fit as many as possible full training examples into the input sequence length, which is equal to 2048 for StarCoder-based models and 512 for CodeBERT-based ones. The rest of the positions are filled with the special padding token. For prediction, the binary classification token is generated according to equation 1, and its probability is taken for ROC AUC calculation;

2. *Binary classification.* This regime uses the binary cross entropy classification loss equation 2 for training and prediction. Specifically, the training scheme for binary classification loss is formalized as follows:

   (a) Load a batch with as many complete function codes as it can fit in the following format:

   Method's code + ⟨EOS⟩

   (b) Populate an array of labels, assigning a label to each function in the batch, where the label is either 1 or 0, indicating whether the function is vulnerable or not, respectively;

14

(c) During loss computation, match the predicted probability of the next token for the last token of each function with its corresponding label from the label array;

(d) Calculate the cross-entropy loss for each pair of predicted probability and actual label using equation 2;

(e) Sum up the cross-entropy loss values across all predictions in the batch (**not** the average, which usually works worse, as will be investigated later);

(f) The resulting sum is the loss for the batch, which is used to update the model's weights during training.

The loss $\mathcal{L}_{\text{batch}}$ for a batch is thus given by:

$$\mathcal{L}_{\text{batch}} = \sum_{i=1}^{B} -y_i \cdot \log(p_i) - (1 - y_i) \cdot \log(1 - p_i) \tag{3}$$

where B is the number of functions in the batch, y_i is the true label for the i-th function, and p_i is the model's predicted probability that the i-th function is vulnerable.

Due to the memory limits of our hardware, we limited the batch size to only one input sequence during training.

For prediction, no batch packing is used and only one test function is fitted into the entire input sequence, with the rest of the token positions filled by the padding token.

The baseline classification approach achieved only 1500 training samples per hour. This extremely slow training throughput made iterative experiments and tuning infeasible on our dataset. In contrast, batch packing provided a 13x speedup to 20000 samples per hour. By concatenating multiple short sequences into one input example, we can significantly boost efficiency for tasks like ours with small input functions. The key advantage of batch packing is reducing unused padding tokens. By packing sequences, a much larger proportion of computation goes towards informative function tokens rather than wasted padding.

In our case, batch packing enables practical finetuning by speeding up training by an order of magnitude. This allows reasonable iteration times for experimentation. Also, this approach is generally applicable when finetuning decoder models on tasks with short input sequences.

There is a room for further enhancement of the batch packing method. The dynamic batch size could be stabilized using dynamic gradient accumulation steps. The dynamic accumulation would accumulate gradients until a target batch size is reached before applying an optimizer step. This would stabilize batch size while retaining the efficiency benefits of packing sequences.

However, this poses some technical challenges. It requires non-trivial modifications to the training loop code in HuggingFace Transformers to support dynamic gradient accumulation.

B EXPERIMENTAL SETUP

B.1 DATASETS

Sources   Our vulnerability dataset is constructed from several open-source vulnerability datasets: CVEfixes Bhandari et al. (2021), a Manually-Curated Dataset Ponta et al. (2019) and VCMatch Wang et al. (2022).

CVEfixes dataset includes information about vulnerabilities and their fixes in open-source software, with data collected from the NVD database and other repositories. However, some inconsistencies and errors may be present due to the automatic construction process.

A Manually-Curated Dataset focuses on Java language only and includes data for 624 publicly disclosed vulnerabilities across 205 distinct open-source Java projects. The dataset is considered more reliable as fix-commits are found by experts.

VCMatch provides a ranking-based approach for automatic security patches localization for OSS vulnerabilities. It includes data for only 10 popular repositories and only contains fix-commits.

Dataset labeling   Each of the aforementioned datasets have standalone functions as their elements. In each of the datasets functions are labeled as vulnerable or non-vulnerable based on heuristics derived from analyzing vulnerability-fixing commits:

1. From vulnerability-fixing commits, the changed functions are extracted;

2. The pre-change versions are taken as vulnerable functions, while the post-change versions are taken as non-vulnerable ones;

3. Pairs of changed functions are used to create a balanced dataset. Functions that remained unchanged are labeled as non-vulnerable as well.

Dataset filtering   Vulnerability-fixing commits often contain changes related to cleanups, refactors, irrelevant functionality changes. Therefore, the described labeling heuristics produce some amount of false positive labels. To address this issue, we create a modified dataset: we extract a function as vulnerable from a vulnerable-fixing commit if this function is the only one that has changed in the commit. We call this dataset X_1. This dataset have the following advantages:

- Its labels are more robust, as commits fixing only one function are more likely to fix only vulnerabilities;

- The dataset is small, which makes it a good instrument for investigating the algorithmic performance and finding optimal parameters.

More specifically, extracting functions from the commits fixing only one function mitigates labeling errors that stem from irrelevant code and cleanup changes Croft et al. (2023).

Addition of easy negative functions   As a result of these procedures, we obtain a set of vulnerable functions (we call it P_1) and a set of vulnerable functions with fixed vulnerability (we call it P_2). We augmented these sets with a set of unchanged functions scratched from files associated with patch, which do not have any vulnerabilities. We call this part P_3.

Datasets characteristics   Finally, we obtain the following two datasets:

- X_1 without P_3.
  It has 1334 samples, with 810 samples in the train part, 272 samples in the validation part and 252 samples in the test part. The balance of positive to negative classes is approximately equal to 1:1;

- X_1 with P_3.
  It has 22945 samples, with 13247 samples in the train part, 5131 samples in the validation part and 4567 samples in the test part. The balance of positive to negative classes is approximately equal to 1:34, and the majority of the negative class is drawn from the P_3 part.

Figures 1 and 2 show the distribution of the top 19 CWE types for the resulting test datasets with and without the P_3 part.

B.2   EVALUATION METRICS

We use a standard procedure for model training and evaluation. We train a model for a predefined number of epochs, choose the best checkpoint by ROC AUC on the validation dataset, and evaluate the chosen checkpoint on the test dataset.

The metrics reported include ROC AUC, F1 score for the positive class (vulnerable functions), optimal classification threshold (used for F1 score). The optimal classification threshold is determined on the validation dataset. In most cases, we report both validation and test metrics.

Hyperparameter optimization is performed on the validation dataset using the best validation metrics performance.

B.3 PRETRAINED MODELS

In this work, the primary pretrained model architecture employed is WizardCoder Luo et al. (2023), a transformer-based neural network with a decoder-only architecture. With over 13 billion parameters, WizardCoder was pretrained using a causal language modeling objective on a large collection of GitHub source code, endowing the model with extensive knowledge of natural programming language constructs.

We compare WizardCoder to ContraBERT Liu et al. (2023a), which is the state-of-the-art model in the CodeBERT family. In contrast to vanilla CodeBERT, ContraBERT introduces an enhanced encoder architecture obtained by cooperation of two pretrained CodeBERT encoders. ContraBERT is pretrained using the usual masked language modeling (MLM) along with a number of contrastive pretraining tasks, and can be finetuned for the defect detection task Liu et al. (2023a). ContraBERT can be considered as an enhancement of the CodeBERT model, which showcases more accurate and robust results than CodeBERT for the defect detection task. Hence, ContraBERT is selected for the experiments.

B.4 FULL REPRODUCIBILITY PACKAGE

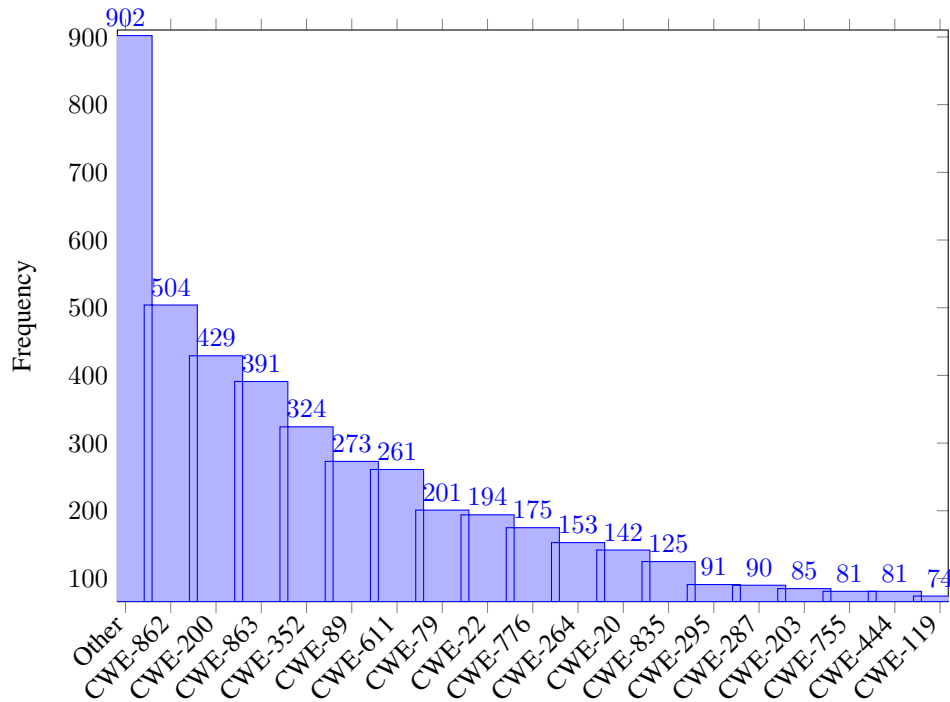All necessary code and data for reproducing our experiments are available in our GitHub repository[1].

Figure 1: Distribution of the top 19 CWE categories and others in the test set with P_3.
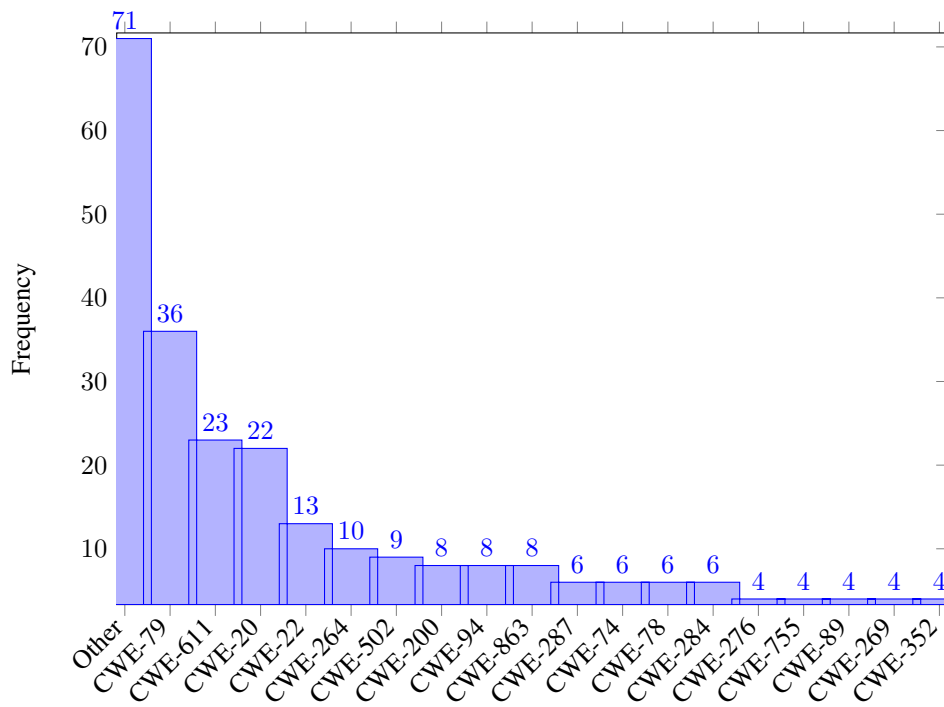

Published as a conference paper at ICOMP 2024

Figure 2: Distribution of the top 19 CWE categories and others in the test set without P_3.