Software Vulnerability Prediction in Low-Resource Languages: An Empirical Study of CodeBERT and ChatGPT

Triet Huynh Minh Le
CREST - The Centre for Research on Engineering Software Technologies, The University of Adelaide
Adelaide, Australia
Cyber Security Cooperative Research Centre, Australia
triet.h.le@adelaide.edu.au

M. Ali Babar
CREST - The Centre for Research on Engineering Software Technologies, The University of Adelaide
Adelaide, Australia
Cyber Security Cooperative Research Centre, Australia
ali.babar@adelaide.edu.au

Tung Hoang Thai
CREST - The Centre for Research on Engineering Software Technologies, The University of Adelaide
Adelaide, Australia
hoangtung.thai@adelaide.edu.au

ABSTRACT

Background: Software Vulnerability (SV) prediction in emerging languages is increasingly important to ensure software security in modern systems. However, these languages usually have limited SV data for developing high-performing prediction models. **Aims**: We conduct an empirical study to evaluate the impact of SV data scarcity in emerging languages on the state-of-the-art SV prediction model and investigate potential solutions to enhance the performance. **Method**: We train and test the state-of-the-art model based on CodeBERT with and without data sampling techniques for function-level and line-level SV prediction in three low-resource languages – Kotlin, Swift, and Rust. We also assess the effectiveness of ChatGPT for low-resource SV prediction given its recent success in other domains. **Results**: Compared to the original work in C/C++ with large data, CodeBERT's performance of function-level and line-level SV prediction significantly declines in low-resource languages, signifying the negative impact of data scarcity. Regarding remediation, data sampling techniques fail to improve CodeBERT; whereas, ChatGPT showcases promising results, substantially enhancing predictive performance by up to 34.4% for the function level and up to 53.5% for the line level. **Conclusion**: We have highlighted the challenge and made the first promising step for low-resource SV prediction, paving the way for future research in this direction.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Software vulnerability, Software security, Large language models, ChatGPT, Empirical study

ACM Reference Format:
Triet Huynh Minh Le, M. Ali Babar, and Tung Hoang Thai. 2024. Software Vulnerability Prediction in Low-Resource Languages: An Empirical Study of CodeBERT and ChatGPT. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3661167.3661281

1 INTRODUCTION

Software Vulnerabilities (SVs) present tremendous threats to the security and dependability of software systems. Given the growing scale and complexity of software applications [27], there is a pressing requirement for the automatic detection of SVs [16]. There has been a growing use of Deep Learning models for SV detection, particularly for identifying potentially vulnerable functions and lines [13, 20, 21, 29]. Among these models, CodeBERT has been demonstrated to be the state-of-the-art for SV prediction [13, 32]. The successful development of these SV prediction models heavily depends on the availability of SV datasets [8].

Datasets necessary for constructing models predicting SVs are often lacking for emerging (recent yet widely used) programming languages. We refer to this scenario as "*low-resource SV prediction*." Our investigation of three emerging languages, namely Kotlin, Swift, and Rust, has revealed their SV data is merely 0.2% to 0.8% the size of that for C/C++, the extensively studied language in the literature. The state-of-the-art SV prediction model, utilizing CodeBERT [13], excels at SV prediction for C/C++ with over 90% F1-Score. However, its performance for emerging languages with the demonstrated limited data is likely to be affected, yet the extent of the impact remains unknown. To tackle the data scarcity, besides traditional data sampling techniques, ChatGPT has shown exceptional performance across tasks [10], including low-resource contexts. Nevertheless, to the best of our knowledge, its applicability to SV prediction in low-resource languages has not been explored.

To answer these questions, we conduct an empirical study on the performance of predicting SVs in three emerging yet low-resource languages, namely Kotlin, Swift, and Rust. We first evaluate the performance of the state-of-the-art CodeBERT based model for the tasks. We then investigate whether data sampling techniques such as random over-sampling and random under-sampling, aiming at tackling data scarcity, can improve the performance of CodeBERT. We also explore the potential use of ChatGPT with few-shot learning and fine-tuning for low-resource SV prediction. Our findings are expected to provide evidence-based knowledge about the extent to which we can reuse the state-of-the-art SV model for emerging languages with limited data and whether data sampling or ChatGPT can improve the performance in this practical scenario.

Our key **contributions** can be summarized as follows:

- We are the first to automate function-level and line-level SV prediction in low-resource languages, i.e., Kotlin, Swift, and Rust.
- We empirically demonstrate the performance of function-level and line-level SV prediction in low-resource languages. Compared to C/C++ with abundant data on which the state-of-the-art CodeBERT model was originally trained, the model obtains much lower performance of SV prediction in low-resource languages (e.g., 0.35 vs. 0.9 for the function level). We also show that data sampling techniques cannot improve the performance. On the other hand, ChatGPT enhances the performance by up to 34.4% for the function level and 53.5% for the line level. Overall, the performance gaps between low-resource and abundant-resource languages are still large, motivating further research.
- We share our data and code for future research at [4].

Paper structure. Section 2 introduces the related work and motivation for the study. Section 3 presents the research questions. Section 4 describes the methods used to answer the questions. Section 5 reports the results to each of the questions. Section 6 discusses the threats to validity. Section 7 concludes the study.

2 BACKGROUND AND MOTIVATION

2.1 Software Vulnerability (SV) Prediction

In recent years, data-driven approaches like Machine Learning and Deep Learning models have been widely used to automate the identification/prediction of SVs in source code (e.g., [16, 26, 29]). The predictions have been performed on various levels of granularity, ranging from package/file to function and line. The more fine-grained function and statement levels can reduce inspection effort for developers [13, 17]. Thus, the two aforementioned fine granularities have become the standard for SV prediction, and thus they are also adopted for our investigations.

Fig. 1 gives an example of an SV (CVE-2020-15230) in the *vapor* project written in Swift. This SV originates from the lines "`var path = request.url.path`" in the `respond` function. This line directly assigns user's input to the `path` variable without performing any sanitization, and this variable is then checked for relative paths on the line "`guard !path.contains("../") else {`". However, attackers can bypass this check by replacing the dot (".") with the percent symbol (`%2E`) in the variable, potentially leading to a path traversal SV. This SV was fixed in the commit *cf1651f* in which any percent symbol would be removed from the `path` variable to ensure that all the subsequent checks would be properly performed.

2.2 Challenges of SV Prediction in Low-Resource Languages

The increasing demand of the software industry has given birth to a wide range of new programming languages. Many of these newly introduced languages have later become widely used for software development because of their unique features and advantages, such as Kotlin, Swift, and Rust. For example, Kotlin finds extensive applications in Android mobile development; Swift has become the language of choice for iOS and macOS applications, emphasizing safety and performance; Rust, known for its memory safety and low-level control, is increasingly used for systems programming, particularly in security-critical contexts.

```
  public func respond(to request: Request, chainingTo next: Responder)
                                        -> EventLoopFuture<Response> {
-   var path = request.url.path
+   guard var path = request.url.path.removingPercentEncoding else {
+     return request.eventLoop.makeFailedFuture(Abort(.badRequest))
+   }
     ...
    // protect against relative paths
    guard !path.contains("../") else {
      return request.eventLoop.makeFailedFuture(Abort(.forbidden))
    }
    ...
}
```

Figure 1: Exemplary vulnerable function and lines corresponding to *CVE-2020-15230* extracted from the respective vulnerability-fixing commit in the *vapor* project in Swift.

While emerging languages play pivotal roles in modern software development, the amount of data, especially concerning SVs, is much more limited compared to traditional languages like C/C++. This argument has been strongly supported by our analysis of SV data in these languages (see Table 1). Specifically, we found that the numbers of SVs in Kotlin, Swift, and Rust were 1,598, 389, and 157 times smaller than that of more established languages like C/C++, respectively. The demonstrated scarcity of SV data can significantly hamper the performance of downstream data-driven SV prediction models for these languages given the data hungriness of these models [8]. Thus, our study is the first to evaluate the performance of CodeBERT [13], the state-of-the-art SV prediction model, in such low-resource languages. Additionally, we aim to explore the feasibility of using ChatGPT to address the data scarcity challenge in these emerging languages, given ChatGPT's success in many other low-resource Software Engineering tasks [10].

2.3 Large Language Models for SV Management

The literature witnesses increasing attention and use of large language models, especially ChatGPT, for SV prediction [36]. Cheshkov et al. [6] leveraged ChatGPT for identifying SVs of five different types (CWE-IDs). Zhang et al. [35] improved the performance of the task by leveraging prompt engineering with ChatGPT. Pearce et al. [31] assessed the performance of various large language models including ChatGPT for SV fixing in the zero-shot scenario. In an attempt to automate various tasks for SV management, Fu et al. [15] investigated ChatGPT with prompt engineering for SV classification, severity assessment, and fixing. Overall, these studies have shown promising results of ChatGPT for SV analysis tasks, especially when using few-shot learning with prompt engineering. Fundamentally, our work is different from the current literature as we focus on empirically evaluating ChatGPT for SV detection in *low-resource languages*, which is an important and practical problem in modern software development. We are also the first to investigate fine-tuning ChatGPT besides prompt engineering for function-level and line-level SV prediction in low-resource languages.

3 RESEARCH QUESTIONS

We answer the following Research Questions (RQs) to investigate the performance of SV prediction in low-resource languages.

Table 1: Data statistics in Kotlin, Swift, and Rust languages.

Statistic	Kotlin	Swift	Rust
Distinct Projects	4	7	19
Vulnerable functions	20	36	90
Non-vulnerable functions	98	449	1,109
Vulnerable lines	45	104	350
Non-vulnerable lines	1,208	8,091	26,479

- **RQ1**: How well does the state-of-the-art CodeBERT based model detect SVs in low-resource languages?
- **RQ2**: Can ChatGPT improve the performance for low-resource SV prediction?

4 CASE STUDY SETUP

This section describes the datasets and the models, i.e., CodeBERT and ChatGPT, used for low-resource SV prediction as well as the evaluation procedure for these models.

4.1 Datasets

We leveraged the methods and tools provided by CVEfixes [5] to curate SV data, including vulnerable functions and lines, for low-resource languages. Essentially, the data collection starts with SV-fixing commits. In these commits, the functions encompassing lines changed are considered vulnerable; otherwise, they are non-vulnerable. The deleted lines are labeled as vulnerable lines. This data curation process follows the same practice of Big-Vul [11], the largest SV dataset in C/C++ widely used in the literature.

Regarding low-resource SV prediction, we selected three languages, Kotlin, Swift, and Rust, for two reasons. Firstly, these three languages have an extremely limited number (< 100) of vulnerable functions, making them directly relevant to our focus on low-resource SV prediction. Secondly, these languages, with the first release recently from 2014 to 2016, are being extensively used in practice, as evidenced by the Stack Overflow survey in 2023.[1] The statistics of the data collected for each language are given in Table 1.

4.2 SOTA for SV prediction with CodeBERT

The fine-tuned CodeBERT [12] model by Fu et al. [13] currently stands as the State-Of-The-Art (SOTA) for function-level and line-level SV prediction [32]; thus, it was employed for our investigations. The model derives code representations capturing both syntactic and semantic information. Function-level predictions are crafted using a Transformer-based architecture; the most vulnerable lines within these functions are then pinpointed using attention scores from the trained Transformer model. Following Fu et al. [13], we also fine-tuned CodeBERT for each of the three studied languages to predict vulnerable functions and lines. The hyperparameters of CodeBERT were adapted from previous studies (e.g., [7, 32]) as follows: *epochs*: 10, *learning rate*: 1e-5, and *feature embedding size*: 768. Despite exploring alternative values, no significant performance improvement was observed. To tackle data scarcity, we also applied Random Over-Sampling (ROS) and Random Under-Sampling (RUS) to *only* the training sets before fine-tuning CodeBERT.

[1]https://survey.stackoverflow.co/2023/#most-popular-technologies-language-prof

Here are some examples of **{language}** source code, each followed by an analysis of its vulnerability status.

{examples}

Now, consider the following **{language}** source code:

{target_function}

Please analyze the source code and determine if it is vulnerable. Answering in the format:

{output_format}

Figure 2: Prompt to use ChatGPT with few-shot learning for function-level SV prediction.

4.3 ChatGPT for SV prediction

We utilized the APIs of ChatGPT based on GPT-3.5-Turbo [30] to develop prediction models for identifying SVs in low-resource languages. Our approach, called prompt chaining, decomposed the task into two subtasks: first detecting vulnerable functions and then predicting vulnerable lines based on the result of the first step.

4.3.1 *Function-level prediction.*

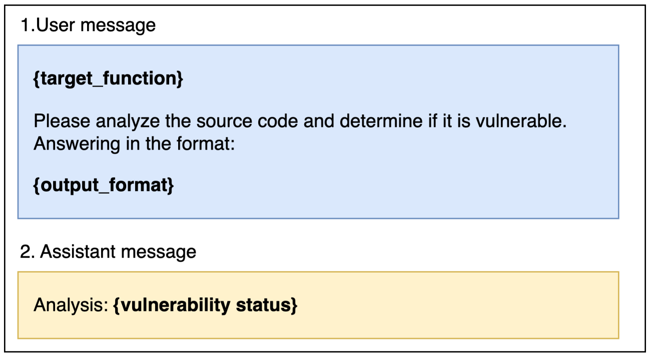
We explored two techniques for function-level SV prediction: few-shot learning and fine-tuning.

Few-shot learning. We crafted a specialized prompt that directed the model to identify SVs within a specified function (see Fig. 2). We incorporated examples showcasing vulnerable and non-vulnerable functions to enhance the model's generalizability across various instances. This method leverages the inherent capability of ChatGPT from vast knowledge to learn from a few SV examples and then draw inferences about new SVs from the contextual clues provided within the given samples.

The prompt of the few-shot learning approach includes:

1. **language:** the programming language of examples and the target function.
2. **examples:** the list of example functions. Each function is formatted as `<input, output>`.
   - input: the code of the function.
   - output: the vulnerability status of the function code.
3. **target_function:** The function we need to predict SV for.
4. **output_format:** The expected format of the model's response. It is set to "vulnerable" or "non-vulnerable".

We carefully designed the few-shot learning prompt, balancing performance with data efficiency and GPT-3.5-turbo model limitation. Our experiments revealed that 10 examples, including nine vulnerable and one non-vulnerable function, achieved this optimal balance. This choice was driven by the limited availability of labeled vulnerable code in training datasets and the maximum token limit of the GPT-3.5-turbo model. Including longer prompts with more examples would exceed this limit, hindering the evaluation stage. Additionally, we found that including too many non-vulnerable examples in the training set also hindered model performance. Conversely, using nine vulnerable examples alongside a single non-vulnerable example demonstrably improved the model's ability to generalize and achieve better performance.


1.User message

{target_function}

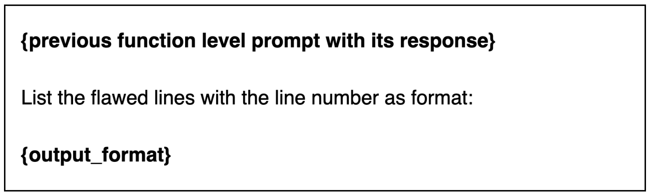
Please analyze the source code and determine if it is vulnerable.
Answering in the format:

{output_format}

2. Assistant message

Analysis: **{vulnerability status}**

Figure 3: Prompt to use ChatGPT with fine-tuning for function-level SV prediction.


{previous function level prompt with its response}

List the flawed lines with the line number as format:

{output_format}

Figure 4: Prompt to use the ChatGPT model trained at the function level for line-level SV prediction.

Fine-tuning. Fine-tuning can address the limitations of the few-shot learning approach, particularly the constraints on the number of examples that can be incorporated within a single prompt illustrated in Fig. 3. Specifically, fine-tuning uses our labeled training data to change the weights of GPT-3.5-Turbo to make the model become more specialized in SV prediction.

Based on the OpenAI's API, the fine-tuning prompt includes:

(1) **User Message:** This element serves as input data for fine-tuning. It includes the *target_function* parameter that specifies the code of each function in a training set and the *output_format* parameter including the expected response, i.e., vulnerable or non-vulnerable.
(2) **Assistant Message:** This element includes the ground truth of the prediction, explicitly indicating the training code snippet's vulnerability status ("vulnerable" or "non-vulnerable"). This clear distinction guides the model in learning the desired output for each training instance.

After fine-tuning, the model can handle zero-shot learning, allowing the model to predict SVs without requiring additional examples within the prompt itself. This streamlines the process and significantly improves response times. Similar to CodeBERT in RQ1, we also investigated fine-tuning ChatGPT with random over-sampling and random under-sampling. We found that over-sampling worked best with ChatGPT, and thus, we would report the ChatGPT's results based on this setting.

4.3.2 Line-level prediction. If a function was predicted as vulnerable by ChatGPT with either few-shot learning or fine-tuning, we would leverage a line-level prediction prompt, given in Fig. 4, to incorporate both the initial prediction and its output as context. We designed this prompt based on the intuition from CodeBERT [13] that the model would know the vulnerable lines, a.k.a the reasons, contributing to its function-level SV prediction. This assumption would be likely valid if the model already correctly predicted the vulnerable functions. This comprehensive view allows the model to focus its attention on specific lines within the function that are most likely vulnerable.

4.4 Model Evaluation

Evaluation technique. We used a 10-round evaluation for the models. Each round split the vulnerable and non-vulnerable functions into training, validation, and testing sets at ratios of 60:20:20, respectively. This ensured sufficient samples for testing function-level and line-level SV prediction. To prevent data leakage, we excluded all duplicate training entries/functions from the validation and testing sets in each round. For CodeBERT, we employed the early stopping strategy [18], i.e., stopping training if the validation performance did not enhance in the last five epochs within a round. We selected the optimal configurations for CodeBERT and ChatGPT based on the highest performance averaging all validation sets.
Evaluation measures. We used established evaluation measures for function-level and line-level SV prediction in our investigations. For function-level SV prediction, we utilized *F1-Score*, *Precision*, and *Recall*, common measures widely applied in prior SV prediction studies (e.g., [13, 23, 32, 37]). F1-Score, being the harmonic mean of Precision and Recall, was chosen for optimal model selection. Reported results reflected the average performance on all the *testing* sets of these optimal models that were determined by the highest validation F1-Score.

For line-level SV prediction, we calculated *Top-3 Accuracy* and *Initial False Alarm* (IFA), standard measures for assessing model interpretability [13, 28].[2] These measures gauge the effectiveness of localizing vulnerable lines, especially the first one, aiding developers in initiating inspection. We also used *Effort@20%Recall* and *Recall@1%LOC* [13] to assess performance while considering the effort developers would need to inspect vulnerable lines.

5 EXPERIMENTAL RESULTS

5.1 RQ1: SOTA Model (CodeBERT) for SV Prediction in Low-Resource Languages

The performance of the fine-tuned CodeBERT for predicting vulnerable functions in low-resource languages was still limited for all three studied languages (see Table 2). The base CodeBERT model attained 0.25 to 0.43 F1-Score for Kotlin, Swift, and Rust, respectively, compared to 0.91 F1-Score originally reported for C/C++ [13]. This finding supports our hypothesis earlier on that low-resource languages with much smaller sizes of data than C/C++ (i.e., 188,636 C/C++ functions, 10,900 of which are vulnerable) are likely to suffer from significant performance degradation. It is also worth noting that CodeBERT's F1-Score rose as the dataset size increased, from Kotlin (the smallest size) to Rust (the largest size). We also observed similar increasing trends for Precision and Recall.

[2]We did not use Top-10 Accuracy, as in CodeBERT, as the average LOC of Kotlin functions was around 10, i.e., vulnerable lines almost always rank in the top list.

Table 2: Function-level SV prediction performance of CodeBERT in low-resource languages. Notes: RUS is Random Under-Sampling; ROS is Random Over-Sampling. The base model of CodeBERT did not use either ROS or RUS.

Lang.	Model	F1-Score	Precision	Recall
Kotlin	CodeBERT (Base)	0.25	**0.30**	0.22
	CodeBERT (RUS)	**0.32**	0.24	**0.47**
	CodeBERT (ROS)	**0.32**	0.29	0.37
Swift	CodeBERT (Base)	**0.37**	**0.44**	0.33
	CodeBERT (RUS)	0.34	0.24	**0.63**
	CodeBERT (ROS)	0.31	0.25	0.42
Rust	CodeBERT (Base)	**0.43**	0.44	0.43
	CodeBERT (RUS)	0.27	0.17	**0.66**
	CodeBERT (ROS)	0.42	**0.46**	0.40
Avg.	CodeBERT (Base)	**0.35**	**0.39**	0.33
	CodeBERT (RUS)	0.31	0.22	**0.59**
	CodeBERT (ROS)	**0.35**	0.33	0.40

Table 3: Line-level SV prediction performance of CodeBERT in low-resource languages. Note: For the metric value, the higher is the better, except for IFA and Effort@20%Recall.

Lang.	Model	Top-3 Accuracy	IFA	Effort@ 20%Recall	Recall@ 1%LOC
Kotlin	CodeBERT (Base)	**100%**	**0.00**	**0.013**	0.111
	CodeBERT (RUS)	87.5%	1.38	0.023	0.134
	CodeBERT (ROS)	**100%**	**0.00**	**0.013**	**0.139**
Swift	CodeBERT (Base)	23.3%	**6.70**	0.032	0.117
	CodeBERT (RUS)	**31.0%**	9.25	**0.024**	**0.250**
	CodeBERT (ROS)	21.7%	9.85	**0.024**	0.163
Rust	CodeBERT (Base)	**58.7%**	**7.69**	0.027	0.216
	CodeBERT (RUS)	43.9%	10.5	0.036	**0.306**
	CodeBERT (ROS)	40.2%	8.47	**0.021**	0.228
Avg.	CodeBERT (Base)	**60.7%**	**4.80**	0.024	0.148
	CodeBERT (RUS)	54.1%	7.03	0.028	**0.230**
	CodeBERT (ROS)	53.9%	6.11	**0.020**	0.176

The performance of the fine-tuned CodeBERT model at line-level prediction varies more than the function level (see Table 3). Unlike the relatively consistent performance of function-level prediction, the line-level prediction exhibited a fluctuating pattern across the Kotlin, Swift, and Rust languages. Generally, we found a significant decrease in performance from Kotlin to Swift, followed by a slight improvement in Rust, suggesting that CodeBERT's performance in pinpointing vulnerable lines may be influenced by other factors beyond the dataset size. For example, a model might have learned code patterns for predicting vulnerable functions that may not align with the actual lines to be fixed by developers [32]. The values of IFA, Effort@20%Recall, and Recall@1%LOC were also lower than those reported for C/C++ [13].

The results in Table 2 show that the data sampling techniques, i.e., ROS and RUS, did not consistently improve performance across all datasets. For the function-level predictions, ROS tended to perform better than RUS and on par with the base model, while for the line-level predictions, ROS and RUS outperformed the base model by only one of four measures. This inconsistency highlights the significant challenge of tackling the data scarcity issue when performing SV prediction in low-resource languages. These results also motivate a need to explore alternative models for the tasks.

Table 4: Comparisons between ChatGPT and CodeBERT for function-level SV prediction in low-resource languages.

Lang.	Model	F1-Score	Precision	Recall
Kotlin	GPT Fine-tuning	0.34	0.37	0.32
	GPT Few-shot	**0.43**	**0.43**	0.44
	CodeBERT (Best)	0.32	0.24	**0.47**
Swift	GPT Fine-tuning	**0.40**	**0.55**	0.32
	GPT Few-shot	0.34	0.36	0.32
	CodeBERT (Best)	0.37	0.44	**0.33**
Rust	GPT Fine-tuning	**0.44**	**0.49**	0.40
	GPT Few-shot	0.09	0.09	0.09
	CodeBERT (Best)	0.43	0.44	**0.43**
Avg.	GPT Fine-tuning	**0.40**	**0.47**	0.35
	GPT Few-shot	0.29	0.29	0.28
	CodeBERT (Best)	0.37	0.38	**0.40**

RQ1 Summary. The performance of CodeBERT for low-resource SV prediction is still limited, as compared to C/C++ with large-sized data. Function-level prediction positively correlates with the data size. Line-level prediction varies more across languages and is not directly affected by the dataset size. Data sampling techniques do not significantly improve function-level and line-level predictions.

5.2 RQ2: ChatGPT for SV Prediction in Low-Resource Languages

Our investigations into ChatGPT with few-shot learning and fine-tuning as alternative models to CodeBERT for SV prediction in low-resource datasets yielded promising results. Note that we reported the results of fine-tuning ChatGPT with random over-sampling because it proved to be the most effective approach for this model type. For CodeBERT, we used the results of the best model obtained from RQ1 for each language.

Function-level prediction. As shown in Table 4, ChatGPT models, on average, produced 2.3% to 34.4% higher F1-Score than CodeBERT across all three datasets (Kotlin, Swift, Rust). Notably, ChatGPT with few-shot learning demonstrated the best performance in the smaller Kotlin dataset. However, its effectiveness tended to diminish with larger and potentially more complex datasets like Rust. In contrast, ChatGPT with fine-tuning exhibited a higher F1-Score than CodeBERT consistently across all languages. Such stability of F1-Score suggests the robustness and scalability of ChatGPT with fine-tuning in handling datasets of varying sizes and complexities. The better overall performance (F1-Score) can be attributed to the higher Precision of ChatGPT than that of CodeBERT, meaning fewer false positives. It is important to note that on average, the higher F1-Score and Precision values of ChatGPT over CodeBERT were statistically significant, based on the Wilcoxon signed-rank tests [34] with p-values < 0.01 and non-negligible effect sizes.[3] Despite the improvements, the best F1 of ChatGPT is still much lower than that (0.91) of C/C++ with abundant SV data.

Line-level SV prediction. For line-level prediction, ChatGPT with few-shot learning and fine-tuning outperformed CodeBERT across

[3] Effect size $(r) = Z/\sqrt{N}$; Z is the Z-score statistic of the test and N is sample size. When $r \geq 0.1$, the effect size is non-negligible [33].

Table 5: Comparisons between ChatGPT and CodeBERT for line-level SV prediction in low-resource languages.

Lang.	Model	Top-3 Accuracy	IFA	Effort@ 20%Recall	Recall@ 1%LOC
Kotlin	GPT Fine-tuning	87.5%	0.12	0.013	0.114
	GPT Few-shot	**100%**	**0.00**	**0.008**	0.114
	CodeBERT (Best)	**100%**	**0.00**	0.013	**0.139**
Swift	GPT Fine-tuning	33.3%	**4.47**	**0.016**	0.090
	GPT Few-shot	**43.3%**	11.5	0.024	0.090
	CodeBERT (Best)	31.0%	9.25	0.024	**0.250**
Rust	GPT Fine-tuning	21.2%	3.29	0.008	0.070
	GPT Few-shot	**64.3%**	**1.00**	**0.001**	0.037
	CodeBERT (Best)	58.7%	7.69	0.027	**0.216**
Avg.	GPT Fine-tuning	47.4%	**2.63**	0.013	0.091
	GPT Few-shot	**69.2%**	4.18	**0.011**	0.081
	CodeBERT (Best)	63.3%	5.65	0.022	**0.202**

all the measures, except Recall@1%LOC, as given in Table 5. On average, the improvements of the ChatGPT variants over CodeBERT were 9.3%, 53.5%, and 50%, for Top-3 Accuracy, IFA, and Effort@20%Recall, respectively. There was no clear winner between the few-shot learning and fine-tuning variants of ChatGPT. The line-level improvements of ChatGPT variants over the CodeBERT models were confirmed statistically significant, based on the Wilcoxon signed-rank tests [34] with p-values < 0.01 and non-negligible effect sizes. Similar to the function level, the line-level performance values of ChatGPT were lower than those of C/C++, except for Top-3 Accuracy as it was not used for C/C++ [13].

RQ2 Summary. ChatGPT performed significantly better than CodeBERT, 2.3% – 34.4%↑ at the function level and 9.3% – 53.5%↑ at the line level, for low-resource SV prediction. ChatGPT with fine-tuning shows the best overall performance for predicting vulnerable functions. There is a performance tie between few-shot learning and fine-tuning for line-level prediction. Despite ChatGPT's improvements, SV predictive performance, especially at the function level, in low-resource languages is still far behind that in abundant-resource languages like C/C++.

6 THREATS TO VALIDITY

The completeness of our datasets is a potential threat. We mitigated this by leveraging the best practice of collecting SV data from the National Vulnerability Database, the largest source of SVs in the wild, based on the methods and tools of CVEfixes [5].

There are possible concerns regarding the choice and optimality of prediction models. Given resource constraints, comprehensively evaluating all available features and models becomes nearly impractical. Thus, we focused on techniques and their associated hyperparameters that have been recommended in the literature. Our pioneering work in low-resource SV prediction, despite imperfect baselines, serves as a catalyst for the evolution of more sophisticated and high-performing techniques in subsequent research.

Regarding the generalizability of our results, we only performed our study in three languages, yet we focused on the languages that are popular among developers with worldwide usage. We also used data from real-world projects of diverse domains and scales.

7 CONCLUSION

Our study addresses the challenge of SV prediction in low-resource languages. Our experiments on Kotlin, Swift, and Rust revealed that the performance of the CodeBERT-based state-of-the-art model was sub-par for function-level and line-level SV prediction in these languages. We explored potential remedies, including data sampling techniques like random over-sampling and under-sampling, yet these approaches failed to enhance CodeBERT's performance. Intriguingly, ChatGPT showed positive results, substantially improving function-level prediction by 2.3–34.4% and line-level prediction by 9.3–53.5%. While our first attempt at low-resource SV prediction for emerging yet low-resource languages is promising, there is still a long way to achieve similar performance as in abundant-resource languages like C/C++. Our findings also underscore the pressing need for continued research in adapting and improving current SV prediction models for low-resource settings.

There are several potential future directions for SV prediction and analysis in low-resource languages. Firstly, ChatGPT can be used in conjunction with latent SVs [22] or semi-supervised learning [25] to enhance the performance of SV prediction in low-resource languages. Secondly, besides SV prediction, future research can investigate other SV management tasks [9, 14, 19, 21, 24, 27] in low-resource languages. Thirdly, the nature of SVs in low-resource languages can change over time, so the predictions should be monitored continuously to mitigate performance degradation [1–3].

ACKNOWLEDGMENTS

The work has been supported by the Cyber Security Research Centre Limited whose activities are partially funded by the Australian Government's Cooperative Research Centres Program.

REFERENCES

[1] Ali Kazemi Arani, Triet Huynh Minh Le, Mansooreh Zahedi, and Muhammad Ali Babar. 2023. Mitigating ML Model Decay in Continuous Integration with Data Drift Detection: An Empirical Study. *arXiv preprint arXiv:2305.12736* (2023).

[2] Ali Kazemi Arani, Triet Huynh Minh Le, Mansooreh Zahedi, and Muhammad Ali Babar. 2023. Systematic Literature Review on Application of Machine Learning in Continuous Integration. *arXiv preprint arXiv:2305.12695* (2023).

[3] Ali Kazemi Arani, Mansooreh Zahedi, Triet Huynh Minh Le, and Muhammad Ali Babar. 2023. Sok: Machine learning for continuous integration. In *2023 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*. IEEE, 8–13.

[4] Authors. [n. d.]. Reproduction package. https://github.com/lhmtriet/LLM4Vul

[5] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. 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*. 30–39.

[6] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT Model for Vulnerability Detection. *arXiv preprint arXiv:2304.07232* (2023).

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

[8] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. 2022. Data preparation for software vulnerability prediction: A systematic literature review. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1044–1063.

[9] Xuanyu Duan, Mengmeng Ge, Triet Huynh Minh Le, Faheem Ullah, Shang Gao, Xuequan Lu, and M Ali Babar. 2021. Automated security assessment for the Internet of Things. In *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 47–56.

[10] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[11] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

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

[13] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[14] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 935–947.

[15] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we? *arXiv preprint arXiv:2310.09810* (2023).

[16] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. 2021. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications* 179 (2021), 103009.

[17] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In *the 19th International Conference on Mining Software Repositories*. 596–607.

[18] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45.

[19] Triet HM Le. 2022. Towards an improved understanding of software vulnerability assessment using data-driven approaches. *arXiv preprint arXiv:2207.11708* (2022).

[20] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–38.

[21] Triet HM Le, Huaming Chen, and M Ali Babar. 2022. A survey on data-driven software vulnerability assessment and prioritization. *Comput. Surveys* 55, 5 (2022), 1–39.

[22] Triet HM Le, Xiaoning Du, and M Ali Babar. 2024. Are Latent Vulnerabilities Hidden Gems for Software Vulnerability Prediction? An Empirical Study. *arXiv preprint arXiv:2401.11105* (2024).

[23] Triet Huynh Minh Le and M Ali Babar. 2022. On the use of fine-grained vulnerable code statements for software vulnerability assessment models. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 621–633.

[24] Triet Huynh Minh Le, Roland Croft, David Hin, and Muhammad Ali Babar. 2021. A large-scale study of security vulnerability support on developer q&a websites. In *Evaluation and assessment in software engineering*. 109–118.

[25] Triet Huynh Minh Le, David Hin, Roland Croft, and M Ali Babar. 2020. PUMiner: Mining security posts from developer question and answer websites with PU learning. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 350–361.

[26] Triet Huynh Minh Le, David Hin, Roland Croft, and M Ali Babar. 2021. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 717–729.

[27] Triet Huynh Minh Le, Bushra Sabir, and Muhammad Ali Babar. 2019. Automated software vulnerability assessment with concept drift. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 371–382.

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

[29] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: A survey. *Proc. IEEE* 108, 10 (2020), 1825–1848.

[30] OpenAI. [n. d.]. OpenAI's GPT 3.5 Turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo

[31] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

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

[33] Maciej Tomczak and Ewa Tomczak. 2014. The need to report effect size estimates revisited. An overview of some recommended measures of effect size. *Trends in Sport Sciences* 1, 21 (2014), 19–25.

[34] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics*. Springer, 196–202.

[35] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2023. Prompt-enhanced software vulnerability detection using chatgpt. *arXiv preprint arXiv:2308.12697* (2023).

[36] Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. *arXiv preprint arXiv:2401.15468* (2024).

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