On the Evaluation of Large Language Models in Multilingual Vulnerability Repair

DONG WANG, College of Intelligence and Computing, Tianjin University, China
JUNJI YU, College of Intelligence and Computing, Tianjin University, China
HONGLIN SHU, Kyushu University, Japan
MICHAEL FU, The University of Melbourne, Australia
CHAKKRIT TANTITHAMTHAVORN, Information Technology, Monash University, Australia
YASUTAKA KAMEI, Kyushu University, Japan
JUNJIE CHEN*, College of Intelligence and Computing, Tianjin University, China

Various Deep Learning-based approaches with pre-trained language models have been proposed for automatically repairing software vulnerabilities. However, these approaches are limited to a specific programming language (C/C++). Recent advances in large language models (LLMs) offer language-agnostic capabilities and strong semantic understanding, exhibiting potential to overcome multilingual vulnerability limitation. Although some work has begun to explore LLM's repair performance, their effectiveness is unsatisfactory. To address these limitations, we conducted a large-scale empirical study to investigate the performance of automated vulnerability repair approaches and state-of-the-art LLMs across seven programming languages. Results show GPT-4o, instruction-tuned with few-shot prompting, performs competitively against the leading approach, VulMaster. Additionally, the LLM-based approach shows superior performance in repairing unique vulnerabilities and is more likely to repair the most dangerous vulnerabilities. Instruction-tuned GPT-4o demonstrates strong generalization on vulnerabilities in previously unseen language, outperforming existing approaches. Analysis shows that Go consistently achieves the highest effectiveness across all model types, while C/C++ performs the worst. Based on findings, we discuss the promising of LLM on multilingual vulnerability repair and reasons behind LLM failed cases. This work takes the first look at repair approaches and LLMs across multiple languages, highlighting the promising future of adopting LLMs to multilingual vulnerability repair.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Multilingual Vulnerability, Vulnerability Repair, Large Language Model

1 INTRODUCTION

Software vulnerabilities are security flaws, glitches, or weaknesses within software code that can be exploited by attackers to compromise the overall security of a system [48]. For example, the *Log4Shell* vulnerability

*Corresponding Author

Authors' Contact Information: Dong Wang, College of Intelligence and Computing, Tianjin University, Tianjin, China, dong_w@tju.edu.cn; Junji Yu, College of Intelligence and Computing, Tianjin University, Tianjin, China, junjiyu@tju.edu.cn; Honglin Shu, Kyushu University, Fukuoka, Japan, shu.honglin.167@s.kyushu-u.ac.jp; Michael Fu, The University of Melbourne, Melbourne, Australia, michaelfu1998@gmail.com; Chakkrit Tantithamthavorn, Information Technology, Monash University, Clayton, Australia, chakkrit@monash.edu; Yasutaka Kamei, Kyushu University, Fukuoka, Japan, kamei@ait.kyushu-u.ac.jp; Junjie Chen, College of Intelligence and Computing, Tianjin University, Tianjin, China, junjiechen@tju.edu.cn.

ACM 1557-7392/2025/10-ART
https://doi.org/10.1145/3771930

(CVE-2021-44228), widely regarded as the most critical vulnerability of the last decade, enables attackers to execute malicious code on any affected system, causing severe financial losses. According to the Common Vulnerabilities and Exposures [68], the number of software vulnerabilities reported in 2023 reached a record high of 28,961, representing a significant 15.57% increase compared to 2022. However, addressing software vulnerabilities frequently necessitates specialized expertise, and the manual resolution of these vulnerabilities is a labor-intensive process [20, 37]. In particular, the time required to fix a vulnerability typically exceeds at least 45 days on average, as indicated by the recent report [16]. Consequently, there is a pressing need for automated approaches to vulnerability repair.

A variety of Deep Learning (DL)-based approaches in recent years have emerged to automate the vulnerability repair process. These methods learn embeddings of vulnerable programs and generate repair patches accordingly. Specifically, the attention-based transformer architecture and pre-trained language models (PLMs) have been widely adopted for automated vulnerability repair, demonstrating effective performance [2, 21, 22, 89]. Their effectiveness primarily stems from the self-attention mechanism, which learns global dependencies among code embeddings, and the pre-trained knowledge that enhances downstream tasks. For instance, Fu et al. [22] introduced VulRepair, a T5-based approach that uses pre-training and Byte-Pair Encoding. Chen et al. [10] created VRepair, which addresses the scarcity of vulnerability-fixing datasets through transfer learning. Most recently, Zhou et al. [89] presented VulMaster, which builds on CodeT5 and incorporates syntax trees and CWE knowledge. VulMaster improves performance by integrating the Fusion-in-Decoder (FiD) framework with multi-task learning, proving to be the most effective approach.

While DL-based vulnerability repair approaches show promise to some extent, their effectiveness is confined to specific programming languages. Table 1 presents the datasets employed in popular DL-based approaches, with CVEfixes [3] and Big-Vul [18] being the most frequently utilized. Notably, both of them focus exclusively on C and C++ languages. *A significant limitation is the lack of knowledge about the performance of automated vulnerability repair approaches in a multilingual context* (**Limitation 1**). This is particularly important for widely-used languages like Python and Java, found to contain numerous vulnerable codes in open-source projects [1, 35]. Li et al. [40] also suggested the importance of assessing and defending against multilingual vulnerabilities. This indicates that modern software development increasingly utilizes multiple programming languages, leading to diverse codebases with unique security challenges. Such complexity makes it difficult for developers to consistently detect and repair vulnerabilities in multilingual programming language environments manually. Therefore, there is an urgent need to expand research on automated vulnerability repairs beyond single programming languages such as C and C++.

Large Language Models (LLMs), trained on the large-scale text and code corpus with billions of parameters, have been extensively explored in the software engineering domain and have shown remarkable performance across various code-related tasks [33, 71, 83]. Moreover, current AVR techniques often rely on detailed information, such as the exact CWE type and precise vulnerability locations, while being constrained by a limited number of

Table 1. Programming languages targeted by existing repair approaches

Approach	Studied Dataset	Languages
TFix [2]	TFix dataset [2]	JavaScript
VRepair [10]	CVEfixes [3] and Big-Vul [18]	C/C++
VulRepair [22]	CVEfixes and Big-Vul	C/C++
VQM [22]	CVEfixes and Big-Vul	C/C++
VulMaster [89]	CVEfixes and Big-Vul	C/C++

context tokens for patch generation. These limitations hinder their applicability in real-world scenarios, where such detailed information is rarely available and a broader contextual understanding is required. In contrast, LLMs could take only the vulnerable code as input and generate the repaired version without requiring additional information or facing similar output constraints. Meanwhile, some researchers have already begun exploring the adoption of LLMs into automated vulnerability repair. For instance, Pearce et al. [56] conducted a large-scale study of five commercially available LLMs, examining their capability for zero-shot vulnerability repair. Likewise, Fu et al. [23] explored the feasibility of two ChatGPT models in prompt-based vulnerability repair. *Despite these attempts, the role of LLMs in automated vulnerability repair remains largely unexplored* (**Limitation 2**). Specifically, existing LLM-based approaches fall short in two key areas: they are inadequately evaluated in complex development environments (i.e., multilingual code) and lack diverse strategies for invoking LLMs, which leads to unsatisfactory repair performance.

To address the above limitations, we conduct an empirical study to comprehensively explore the effectiveness of the existing automated vulnerability approaches and the potential of leveraging LLMs for repairing vulnerabilities in the multilingual context. Given LLMs' strong semantic understanding and language-agnostic capabilities, we assume that LLMs paired with effective learning strategies could achieve promising results. We evaluate the aforementioned approaches using REEF, the latest multi-language vulnerability dataset, which comprises 4,466 CVEs with 30,987 patches across seven popular programming languages (C, C#, C++, Go, JavaScript, Java, and Python). We formulate the following four research questions to guide our study:

RQ1: What is the performance of existing learning-based repair techniques in multilingual vulnerability? We first aim to investigate the effectiveness of the existing repair techniques, encompassing both state-of-the-art DL-based approaches and widely used pre-trained language models. These techniques are typically devised for a specific programming language in the prior work, hence answering this RQ would provide a better understanding of their knowledge transfer capabilities in vulnerability repair across different languages.

RQ2: What is the performance of state-of-the-art LLMs in repairing multilingual vulnerability? This RQ examines the effectiveness of both open-source and closed-source LLMs in repairing multilingual vulnerabilities. Different strategies such as code embedding and prompting, utilized by LLMs may influence repair performance. Thus, we further investigate the extent of their impact. Building on this and drawing inspiration from Mueller et al. [46], we design an instruction-tuning strategy as a multi-task learning paradigm to enhance LLMs' ability to capture vulnerability semantics across different programming languages. Answering this RQ could offer insights into the optimal selection of strategies for enhancing LLM performance on multilingual vulnerability repair.

RQ3: What are the strengths and weaknesses of the studied automated vulnerability techniques? Certain repair techniques and LLM strategies may be prone to repairing unique vulnerabilities based on specific programming languages and vulnerability types. Therefore, RQ3 seeks to gain an understanding of the characteristics of various repair categories and LLM strategies by analyzing the orthogonality between them, revealing the strengths and weaknesses of different techniques and strategies.

RQ4: What is the generalization capability in repairing previously unseen vulnerabilities? This RQ investigates the effectiveness of the best-performing LLM and AVR techniques in repairing previously unseen vulnerabilities. Although current methods demonstrate strong performance on in-domain data, their ability to handle out-of-distribution vulnerabilities remains unclear. Therefore, RQ4 aims to evaluate the generalization capability of these techniques and provide insights into the optimal selection of approaches for addressing unseen vulnerabilities.

Our key empirical findings are: (1) Among state-of-the-art AVR and PLM techniques, VulMaster achieves the best performance across the studied evaluation metrics, with an Exact Match score of 28.94%; (2) Among five advanced LLMs with four compositional strategies, the instruction-tuned ChatGPT-4o with a few-shot prompting strategy significantly outperforms others in Exact Match, BLEU, and ROUGE metrics (with an Exact Match score of 28.71%), which is competitive with the VulMaster; (3) Regardless of model type, performance

remains relatively consistent across the seven studied programming languages, with Go yielding the best results and C/C++ the poorest; (4) The orthogonality analysis reveals that the best-performing LLM-based model demonstrates superiority in unique correct repairs and is more likely to repair the most dangerous vulnerabilities compared to VulMaster. (5) The results on TypeScript vulnerabilities indicate that GPT-4o, when enhanced with instruction tuning and few-shot prompting strategies, exhibits superior generalization to previously unseen programming language vulnerabilities compared to VulMaster. In addition, our manual analysis shows that when instruction-tuned ChatGPT-4 fails to repair vulnerabilities using few-shot prompting, the main challenge lies in error localization.

Contributions. To sum up, the contributions of this study are:

(1) The first empirical study to systematically investigate the effectiveness of existing AVR techniques and various LLMs on multilingual vulnerability repair.
(2) The empirical results confirm the promising role of LLMs in multilingual vulnerability repair, particularly when instruction-tuning LLMs with few-shot prompting strategies.
(3) We provide valuable insights into the capabilities and limitations of LLMs for multilingual vulnerability repair, served as essential guidance for future research aimed at enhancing LLM-based vulnerability repair.
(4) We open source all data, code, and analysis details involved in our study [31].

2 BACKGROUND AND RELATED WORK

2.1 Software Vulnerability and Datasets

A software vulnerability is a flaw or weakness in a software system that attackers can exploit. Its impact can be severe, as unpatched vulnerabilities in widely used software may result in catastrophic consequences, including significant economic losses [4]. To classify different types and instances of vulnerability, Common Weakness Enumeration (CWE) and Common Vulnerability Exposure (CVE) are used to refer to the types of software weakness that can lead to vulnerabilities and a specific instance of a vulnerability in a software system [13, 14]. Software vulnerabilities are frequently found in open-source projects. Alfadel et al. [1] provided insights into common security issues within the Python ecosystem by analyzing 550 vulnerability reports affecting 252 Python packages. Hu et al. [35] examined the prevalence and remediation delays of vulnerabilities in Go modules, revealing that 66.10% of modules are affected and identifying two types of lags that hinder timely vulnerability fixes. Meng et al. [44] investigated developers' challenges in implementing secure Java coding practices, highlighting vulnerabilities like insecure configurations and misused security libraries, while offering recommendations for improving Java application security. Zhang et al. [85] examined 646,716 C/C++ code snippets from Stack Overflow, finding that 2% contained security weaknesses (primarily improper memory operations and null pointer dereferences) and noted that while code revisions often reduced these issues, many weaknesses persisted.

Researchers have developed several datasets to study vulnerability repair. The VulnLoc [64] dataset consists of 43 vulnerable programs in 10 projects that span 6 CWEs. The Vul4J [6] dataset encompasses 79 reproducible vulnerabilities drawn from 51 open-source Java projects, spanning 25 different CWE types. Reis and Abreu [60] developed a dataset of 5,942 security patches from the complete CVE details database, spanning 1,339 projects and 146 vulnerability types across 20 languages. SATE IV [50] is a dataset initially designed to assess static analysis tools for detecting security-relevant defects which consists of 117,738 synthetic C/C++ functions categorized under 116 CWE types. VulinOSS [27] is a dataset derived from open-source projects, assembled using vulnerability reports from the National Vulnerability Database (NVD), encompassing 17,738 vulnerabilities across 153 projects. Big-Vul [18] is a dataset of C/C++ code vulnerabilities sourced from 348 open-source GitHub projects which comprises 3,754 vulnerabilities and represents 91 distinct CWE types. This dataset is designed for multiple purposes, including vulnerability detection, vulnerability repair, and vulnerability analysis. CVEfixes [3]

is a vulnerability dataset based on CVE records up to June 9, 2021 from the NVD, containing 5,365 CVEs from 1,754 projects. Recently, Wang et al. [70] proposed REEF, an automated framework for collecting high-quality vulnerabilities and their fixes across various languages, platforms, and granularity. The dataset details are shown in Section 3.1.

2.2 Automated Vulnerability Repair

Researchers have proposed leveraging various Neural Machine Translation (NMT) approaches for Automated Vulnerability Repair (AVR). Harer et al. [30] introduced using generative adversarial networks (GAN) for AVR task. They utilized a traditional NMT model as the generator to generate the examples to confuse the discriminator whose task is to discern the NMT-generated repairs from the real repairs. Chen et al. [10] proposed VRepair which leverages a word-level tokenizer and a vanilla Transformer model. VRepair starts by pre-training the vanilla Transformer model on a bug-fixing corpus and subsequently uses it to address C/C++ vulnerabilities with fine-tuning. Similar to VRepair, Chi et al. [11] introduced SeqTrans, a Transformer-based NMT model with copy mechanisms designed to repair Java vulnerabilities. Fu et al. [22] introduced VulRepair, a system that utilizes a Byte Pair Encoding (BPE) tokenizer and CodeT5, which is pre-trained on a substantial corpus of code to address C/C++ vulnerabilities. Wu et al. [79] conducted an evaluation involving four program repair models and nine large language models on the Vul4J dataset. Zhou et al. [89] proposed VulMaster. This Transformer-based NMT model addresses the input length constraints inherent in traditional Transformer-based pre-trained models by employing the Fusion-in-Decoder (FiD) framework [36]. By eliminating the length limit, VulMaster can integrate vulnerable code structures and expert knowledge to further enhance its repair capabilities on repairing C/C++ vulnerabilities. Inspired by Vision Transformer (VIT)-based objection detection approaches [8, 90] in the computer vision domain, Fu et al. [21] introduced a T5-based approach named VQM designed to enhance awareness and attention to vulnerable code areas within a function to produce better repairs for C/C++ vulnerabilities. According to the literature, while various AVR techniques exist, they primarily focus on C/C++ as their target language.

With the rapid advancement of LLMs, research has increasingly focused on evaluating their effectiveness in vulnerability repair. Pearce et al. [56] examined the power of zero-shot prompting on vulnerability repair, while Fu et al. [23] instructed ChatGPT for repairing vulnerabilities over extensive real-world C/C++ datasets. Kulsum et al. [39] proposed an LLM-based vulnerability repair technique based on chain-of-thought prompt and patch validation feedback on C vulnerabilities of CVEfixes dataset. Zhang et al. [86] evaluated the capability of advanced LLMs in fixing memory corruption vulnerabilities in real-world C/C++ code. Fakih et al. [17] proposed an LLM-based iterative pipeline designed to effectively and robustly repair vulnerable functions in C code. Zhou et al. [88] conducted a systematic literature review of existing LLM-based approaches for vulnerability repair. Despite these efforts, the effectiveness of LLMs in automated vulnerability repair remains largely unexplored due to the following limitations: (i) the performance is unsatisfactory yet, as demonstrated in Fu et al. [23]; (ii) the diverse learning strategies have not been utilized to unlock the LLMs' potential; and (iii) evaluated datasets are relatively small and do not cover a wide range of programming languages. Our study addresses these limitations by comprehensively evaluating how popular and advanced LLMs perform using different strategies for automated vulnerability repair. To better understand how far we are, we compare their performance against state-of-the-art AVR techniques and PLMs, using fine-grained analysis to investigate their characteristic differences.

3 STUDY DESIGN

Figure 1 illustrates the overview of our study design. Initially, based on the latest multi-language vulnerability dataset REEF [70], we construct the training and test datasets comprising code pairs of the vulnerability function and its repair. We first investigate the effectiveness of the existing learning-based AVR techniques and widely used PLMs on automated vulnerability repair in the multilingual context (RQ1). We also examine how different

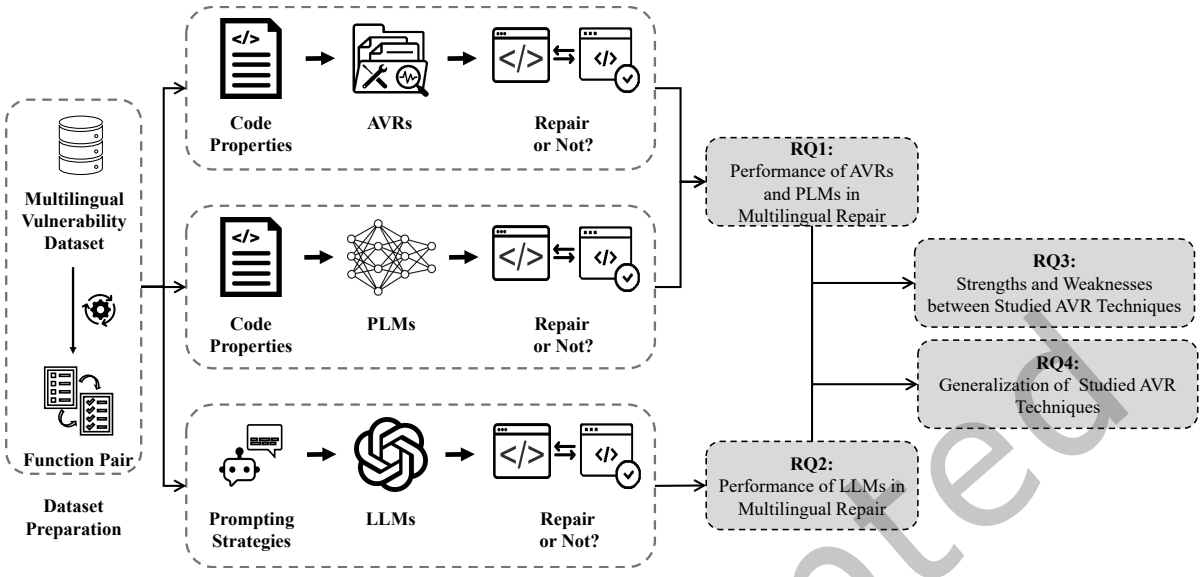


Fig. 1. Overview of study design

LLM strategies affect the performance of automated vulnerability repair (RQ2). Finally, we gain insights into the strengths and limitations of the studied AVR techniques by analyzing their orthogonality (RQ3) and further investigate their generalization capabilities (RQ4).

3.1 Dataset Preparation

Studied dataset. To evaluate the performance of AVR techniques, we select the multi-language vulnerability dataset REEF [70], containing 4,466 CVEs with 30,987 patches (including 236 CWE) across seven programming languages with detailed related information including vulnerability information (e.g., the identifier of CVE and the CVSS scores reflecting the severity) and project information (e.g., the commit message and the URL of the source file changed in the commit). More specifically, REEF gathers real-world vulnerabilities from the NVD database and CVE list maintained by Mend [77] which is an open-source vulnerabilities database and collects CVEs from 2016 to 2023. In this study, we focus on all seven programming languages: C, C++, C#, Go, Java, JavaScript, and Python. At the function level, the dataset comprises 6,957 C, 2,244 C++, 1,529 C#, 3,187 Go, 6,207 Java, 5,066 JavaScript, and 5,797 Python functions.

Data pre-processing. Since the REEF dataset was not originally collected for automated vulnerability repair tasks, we needed to first retrieve the vulnerable functions. The commit data comes in a raw code format with patches that cannot be directly used for extracting vulnerable functions. To obtain the commit data containing both pre-change and post-change files, we applied patches to the raw code using the Linux patch command. [1] After obtaining the required commit data, we removed all code comments to reduce potential bias and collected the function definition code, utilizing the static analysis tool Tree-sitter [5]. Tree-sitter is a parser generator tool and incremental parsing library that can be used to parse any programming language. To extract the vulnerable function definition and corresponding repair code, we iterated the function definition in the pre-change file in order to match the corresponding post-change function definition. Specifically, the function name is used as a

[1] https://www.man7.org/linux/man-pages/man1/patch.1.html

Table 2. Statistic summary of the experimental dataset

Languages	Training	Validation	Test	Total
C	1,063	151	305	1,519
C++	959	137	275	1,371
C#	159	22	47	228
Go	1,171	167	334	1,672
Java	1,193	167	343	1,703
JavaScript	1,668	239	484	2,391
Python	1,235	176	354	1,765
Total	7,448	1,059	2,142	10,649

key to match the same function definition in the post-change file. If multiple function definitions are matched in the post-change file, the edit distance would be calculated between the pre-change function definition and each post-change function definition, and the function definition pair with the minimal edit distance is selected as the vulnerable function and its repair. In other words, we define the pre-change function and the post-change function with different source codes as a function pair. After this step, we obtained a total of 11,808 function pairs.

To minimize bias from potential data leakage, we removed function pairs that appeared in both the Big-Vul and CVEfixes datasets, resulting in 1,159 duplicate pairs. In the end, we were able to collect 10,649 vulnerable function pairs for seven languages, as shown in Table 2. The reduction in vulnerable functions compared to the original dataset could be attributed to three main reasons: (i) the Tree-sitter cannot parse certain cases (such as special C macros), (ii) some functions have multiple patches, and (iii) some functions are completely deleted or added. Moreover, we performed a sanity check to ensure the robustness of the Tree-sitter. To do so, the first author randomly selected 50 samples to examine both the consistency of function definitions between pre-change and post-change versions and their alignment with the provided patch. The manual results suggest that the Tree-sitter is reliable and produces no false positives.

Construction of training and test datasets. Following existing work [21, 89], we adopted a widely-used random sampling strategy to divide the REEF dataset into training, validation, and test sets with a ratio of 7:1:2. To ensure that data across different languages maintain consistent proportions, we first allocate the data for each language based on the predetermined ratio. Subsequently, we collect the corresponding portions to assemble the final dataset. As shown in Table 2, our experimental dataset contains 7,448, 1,059, and 2,142 function pairs for training, validation, and testing over seven languages.

Distribution of vulnerability severity. To analyze the dataset quality, we examined the severity levels of all 10,649 vulnerabilities, as shown in Table 2. We relied on the Common Vulnerability Scoring System (CVSS), a method used to supply a qualitative measure of severity. Specifically, we used the metric CVSS v4.0 Ratings[2] to classify the vulnerabilities into four levels: low, medium, high, and critical. Figure 2 depicts the related severity distribution across the studied seven programming languages. Analysis shows that only 1.2% of vulnerabilities are rated as low severity, while a significant 61.3% are classified as high or critical severity. These results indicate that the real-world vulnerabilities provided by REEF are of high quality and more likely to target severe vulnerabilities.

3.2 Existing Automated Vulnerability Repair Approaches

In our study, we investigated the performance of two types of existing approaches for AVR: learning-based automatic vulnerability repair approaches and pre-trained language models. For learning-based approaches, we included five state-of-the-art approaches.

[2]https://nvd.nist.gov/vuln-metrics/cvss

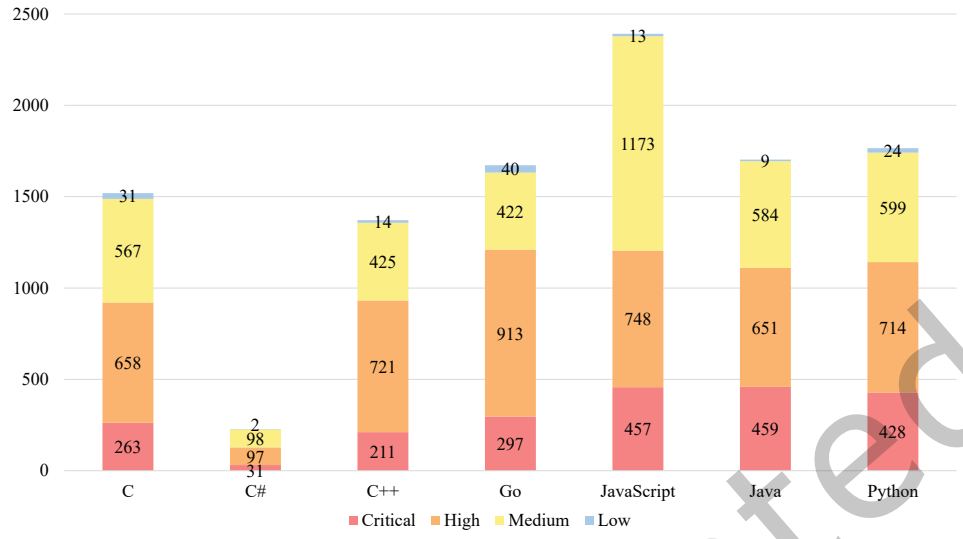


Fig. 2. The severity distribution across seven programming languages on the experimental dataset

- **TFix** [2] is a T5-based approach that utilizes a large language model pre-trained on a natural language corpus [59].
- **VRepair** [10] is a vanilla transformer architecture with word-level tokenization for input, pre-trained on a bug-fixing [10] dataset and fine-tuned for AVR.
- **VulRepair** [22] is a T5-based approach that incorporates the Byte Pair Encoding (BPE) method [63] for subword-level tokenization, using a large language model pre-trained on a source code corpus [74].
- **VulMaster** [89] is a transformer-based neural network model that integrates source code with code syntax trees and specific CWE knowledge in its inputs, employing Fusion-in-Decoder (FiD) framework [36] and multi-task learning framework [73].
- **VQM** [21] is a T5-based approach that utilizes vision transformer methodologies to enhance the encoder and decoder focus on vulnerable areas of code during the repair process.

For pre-trained language models, we selected seven models across three architectures (encoder-only, decoder-only, and encoder-decoder) that are widely used in the literature [21, 22, 89], including:

- **CodeBERT** [19] is a widely recognized pre-trained model that leverages a multilayer Transformer architecture to effectively learn from bimodal data, comprising both source code and natural languages.
- **GraphCodeBERT** [28] is a pre-trained model that utilizes a transformer-based architecture to enhance code representation by incorporating semantic-level information from code.
- **CodeT5** [74] is a unified encoder-decoder model that extends the T5 architecture by incorporating token type information in code and utilizing denoising sequence-to-sequence pre-training to improve model performance.
- **CodeReviewer** [41] is a pre-trained encoder-decoder model specifically designed for analyzing code diffs and reviews, featuring four tailor-made pre-training tasks to enhance its effectiveness in the code review scenario.
- **PolyCoder** [80] is a pre-trained GPT-2-based model [58] specifically designed for multi-lingual code modeling, developed to address the gap in large open-source models exclusively trained on code.

- **CodeGen** [49] is a pre-trained decoder-only model which specifically engineered for program synthesis using natural and programming language data.
- **GPT2-CSRC** [56] is a pre-trained decoder-only model based on GPT-2, equipped with a BPE tokenizer and specifically developed for C/C++ code analysis, having been trained on a substantial dataset of approximately 17 GB of C/C++ code.

3.3 Experimented Large Language Models

We studied five advanced LLMs (i.e., three open-source LLMs and two closed-source LLMs), which have demonstrated excellent performance in the various code-related downstream tasks [67, 71]. The details of these LLMs are as follows:

- **DeepSeek-Coder** [29] trained on 2 trillion tokens across 87 programming languages, utilizing a 16K context window and fill-in-the-blank tasks to enhance code generation and infilling capabilities, achieving state-of-the-art performance among open-source models and surpassing closed-source counterparts like Code Llama and GPT-3.5 in some tasks.
- **Code Llama** [62] is a decoder-only model which is one of the most popular LLMs for code generation and infilling derived from Llama 2 models. It undergoes additional fine-tuning on 500B tokens derived from an extensively code-rich dataset.
- **Llama 3** [15] is a series of large multilingual models based on the Transformer architecture, designed to improve performance across various language understanding tasks. By optimizing data quality, training scale, and model architecture, Llama 3 demonstrates significant potential in the field of natural language processing.
- **ChatGPT** [51] is a groundbreaking LLM capable of transforming various fields through its advanced natural language processing capabilities. It is trained on a vast corpus of natural language texts and code snippets, employing reinforcement learning to enhance its ability to adhere to human directives. Specifically, we studied two LLMs, i.e., GPT-3.5-Turbo [51] and GPT-4o [53].

3.4 Strategies for Large Language Models

We further investigated the impact of LLMs' learning strategies in automated vulnerability repair. Existing studies [69] have demonstrated that fine-tuning strategies can lead to significant enhancement in LLM performance effectively by adapting general LLMs to specific downstream tasks. Meanwhile, prompting strategies have also been proposed to achieve the same objective in a plug-and-play manner [38]. Thus, we devised three LLM strategies as follows:

- **Zero-shot prompting strategy**: we devised a prompt with system role and user role following prior works [87, 89] without any examples, which directly utilized a structured instruction and a vulnerable function to prompt LLMs for vulnerability repair. Figure 3a shows the prompt template for zero-shot prompting. In our prompt design, we first assign a specific role to the LLM (e.g., "You are an expert software developer in <language>"), then define the vulnerability repair task, and finally request the LLM to generate the repaired code.
- **Few-shot prompting strategy**: it enables LLMs to learn the relationship between the vulnerable function and the repaired function based on selected function pair examples. That is, it concatenates these demonstration examples with a zero-shot prompt to form a new few-shot prompt, which is then fed to LLMs for vulnerability repair. Figure 3b shows the prompt template for few-shot prompting. Following prior study [57], for a given vulnerable function, three demonstration examples from the REEF training dataset are selected by BM25 [61] to prompt LLMs for vulnerability repair. BM25 is selected as the sample selection approach since prior work [26, 84] shows that BM25 outperforms other sample selection approaches for

software engineering tasks. We choose three demonstration examples for each function sample based on the findings of Gao et al. [26], which indicate that GPT-3.5 can achieve approximately 90% of its highest Exact Match score with just three examples, compared to using 16 or more.

- **Instruction-tuning strategy**: it enables LLMs to acquire specific knowledge through training on many more instruction-filled function pairs. Research suggests that fine-tuning LLMs on diverse multi-task datasets accompanied by natural language descriptions significantly enhances their performance on previously unseen tasks[54]. Specifically, we used the same structured instruction that is described in the previous zero-shot prompting strategy to construct an instruction-filled fine-tuning set. Then, we fine-tuned the LLMs on the fine-tuning set to repair vulnerable functions by the zero-shot prompt or few-shot prompt.

3.5 Evaluation Metrics

In accordance with previous studies [10, 21, 22, 87, 89], we used the most widely-adopted *Exact Match (EM)* metric with beam search to evaluate our automated vulnerability repair techniques. A repair is considered an EM if any beam search output matches the ground-truth label exactly. We evaluated all methods using beam sizes of 1, 3, and 5.

We also utilized another two widely-used metrics for code generation tasks, i.e. BLEU-4 [55] and ROUGE [42]. BLEU-4 measures how similar the generated code is to the ground truth at the token level. It focuses on precision by comparing n-grams and applying length penalties. In contrast, ROUGE evaluates how well the generated code overlaps with the ground truth, with a particular focus on recall. We adopted ROUGE-1, ROUGE-2, and ROUGE-L as metrics. These metrics evaluate the quality of generated texts from distinct perspectives and are often used in conjunction to provide a more comprehensive assessment.

3.6 Implementation and Environment

We replicated TFix, VRepair, VulRepair, VQM, and VulMaster by following their documented implementation instructions and parameter settings provided in replication packages. All the open-source pre-trained models (i.e., CodeBERT, GraphCodeBERT, CodeT5, CodeReviewer, PolyCoder, CodeGen and GPT2-CSRC) are downloaded from Huggingface [78] and we employed the parameter settings suggested by the work of [21].

Regarding closed-source LLMs, we invoked GPT-3.5-Turbo and GPT-4o through OpenAI's APIs [52]. Concretely, we used `gpt-3.5-turbo-0125` as the specific experimental model version for GPT-3.5-Turbo and used `gpt-4-0613` as the experimental model version for GPT-4o. Regarding open-source LLMs (i.e., DeepSeek-Coder, Code Llama, and Llama 3), we downloaded the pre-trained model from Huggingface [78]. Specifically, the model sizes are 6.7B, 7B, and 8B for the three studied open-source LLMs, respectively. To save computational cost and prevent over-fitting, we adopt Low-Rank Adaption (i.e., LoRA [34]) to fine-tune the open-source LLMs. We configured the LLMs to generate a maximum of 1,024 new tokens, while preserving all other parameters at their default settings. We conducted all the experiments on an Intel Xeon CPU Gold-6342 machine with 512 GB RAM, Ubuntu 20.04.6, and two A800 GPUs. More implementation details can be found in our replication package. To facilitate future research, we have made publicly available all used datasets and model execution scripts.

4 EVALUATION RESULTS

4.1 RQ1: What is the performance of existing learning-based repair techniques in multilingual vulnerability?

Approach. This research question offers a comparative analysis of the performance between various AVRs and state-of-the-art PLMs in repairing multilingual vulnerabilities. Specifically, we investigated the effectiveness of

(Instruction)
You are an expert software developer in **<*language*>**.
You always want to improve your code to have higher quality.
Your task is to repair the given vulnerable C/C++/C#/Java/JavaScript/Go/Python function.
Please only generate the fixed code without your explanation.

(Input)
<*vulnerable code*>

(a) A prompt template for zero-shot prompting

(Instruction and examples)
You are an expert software developer in **<*language*>**.
You always want to prove your code to have higher quality.
Your task is to repair the given vulnerable C/C++/C#/Java/JavaScript/Go/Python function.
Please only generate the fixed code without your explanation You are given 3 examples.
Each example begins with "##Example" and ends with "---".
Each example contains vulnerable code and the fixed code.
The vulnerable code and repaired code is written in **<*language*>**.
Your task is to repair the vulnerable code base on the examples.
Example
Vulnerable code: **<*example1 input*>**
Repaired code: **<*example1 target*>**

Example
Vulnerable code: **<*example2 input*>**
Repaired code: **<*example2 target*>**

Example
Vulnerable code: **<*example3 input*>**
Repaired code: **<*example3 target*>**

(Input)
<*vulnerable code*>

(b) A prompt template for few-shot prompting

Fig. 3. Zero-shot prompt and few-shot prompt for multilingual AVR

five AVRs and seven PLMs. The model information is presented in Section 3.2. We now detail the pre-training process below.

Although PLMs are pre-trained on a large number of natural languages or source code snippets, they have not seen vulnerability repairs with special tokens during their pre-training, resulting in a lack of generating special tokens in vulnerability repairs. Prior studies [10, 21, 22, 87, 89] have demonstrated that the effectiveness of pre-training on a bug fixing task would facilitate the performance of models on vulnerability repair. To incorporate code AST information into input, VulMaster is pre-trained on a bug-fixing dataset provided by Chen et al. [10] which consists of over 500,000 pairs of buggy and fixed functions. Therefore, to mitigate potential bias, we conducted pre-training with AVRs and PLMs on this bug-fixing dataset, aligning them with each other.

Table 3. Performance of existing learning-based techniques on multilingual vulnerability repair

Technique	EM with beam 1	EM with beam 3	EM with beam 5	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
Learning-based AVR technique							
TFix	10.32%	12.79%	12.70%	24.19%	0.5129	0.3124	0.4966
VRepair	3.97%	4.90%	5.32%	14.09%	0.4072	0.1833	0.3909
VulRepair	14.33%	18.58%	19.28%	32.18%	0.5900	0.3943	0.5727
VQM	15.83%	19.42%	20.03%	32.76%	0.5856	0.4029	0.5685
VulMaster	**28.94%**	**33.47%**	**34.59%**	**42.99%**	**0.6753**	**0.5149**	**0.6602**
Encoder-only PLM technique							
CodeBERT	9.24%	11.95%	13.12%	26.14%	0.5137	0.3014	0.4934
GraphCodeBERT	8.82%	11.20%	12.14%	23.71%	0.5125	0.2957	0.4939
Encoder-decoder PLM technique							
CodeT5	14.33%	18.58%	19.28%	32.18%	0.5900	0.3943	0.5727
CodeReviewer	13.77%	17.60%	18.35%	31.49%	0.5777	0.3894	0.5620
Decoder-only PLM technique							
PolyCoder	6.40%	7.89%	7.52%	16.86%	0.4773	0.2435	0.4592
CodeGen	9.20%	11.34%	11.62%	20.98%	0.5142	0.2880	0.4968
GPT2-CSRC	9.80%	10.74%	11.11%	22.22%	0.5289	0.3054	0.5095

Following existing work [10], we reuse their code to process vulnerable function pairs and adapt them as inputs for AVR and PLM models. Particularly, each vulnerable function is marked using the special tokens <StartLoc> and <EndLoc>. The <StartLoc> token indicates the beginning of the vulnerable code lines and <EndLoc> token indicates the end. To put it another way, <StartLoc> and <EndLoc> provide vulnerability localization to vulnerability repair approaches. For the repair function, changed codes with three context tokens are extracted as the ground truth which are surrounded with special tokens <ModStart> and <ModEnd>. In other words, vulnerability repair approaches generate repair code segments that can be mapped to the changed before function rather than the whole repair function. For each vulnerable function, we add CWE-type information at the beginning. For functions without CWE information, we label them as CWE-000.

To evaluate the effectiveness of the studied AVRs and PLMs, we used multiple metrics: Exact Match (EM) with beam sizes of 1, 3, and 5, BLEU score, and ROUGE-1, ROUGE-2, and ROUGE-L metrics, as detailed in Section 3.5.
Results. Table 3 shows the comparison results among AVRs and PLMs in terms of EM, BLEU, and ROUGE scores in the multilingual vulnerability repair context. Within each metric, the values in bold indicate the technique that exhibits the best performance among all AVRs and PLMs. Figure 4 and Figure 5 present the performance of AVRs and PLMs across seven programming languages in terms of EM with beam size 1.

VulMaster significantly outperforms other models across all metrics, demonstrating the highest effectiveness in EM with beams 1, 3, and 5, as well as in BLEU, ROUGE-1, ROUGE-2, and ROUGE-L scores. In particular, VulMaster achieves an impressive 28.94% on the EM with beam 1 metric, surpassing other methods by a substantial margin ranging from 82.82% to 628.97%. Followed by VulMaster, VQM which is a T5-based approach utilizing vision transformer methodologies achieves an EM of 15.83% with beam 1. As for the worst effective technique VRepair in learning-based AVR techniques, it is the only technique that pre-trains a transformer from scratch on the C/C++ only bug-fixing dataset and fine-tuning on the REEF training dataset which comprises seven language programs rather than integrating with powerful PLMs. Moreover, we observe that VulMaster demonstrates relatively balanced performance across different programming languages as shown in Figure 4, highlighting its stability in cross-language scenarios.

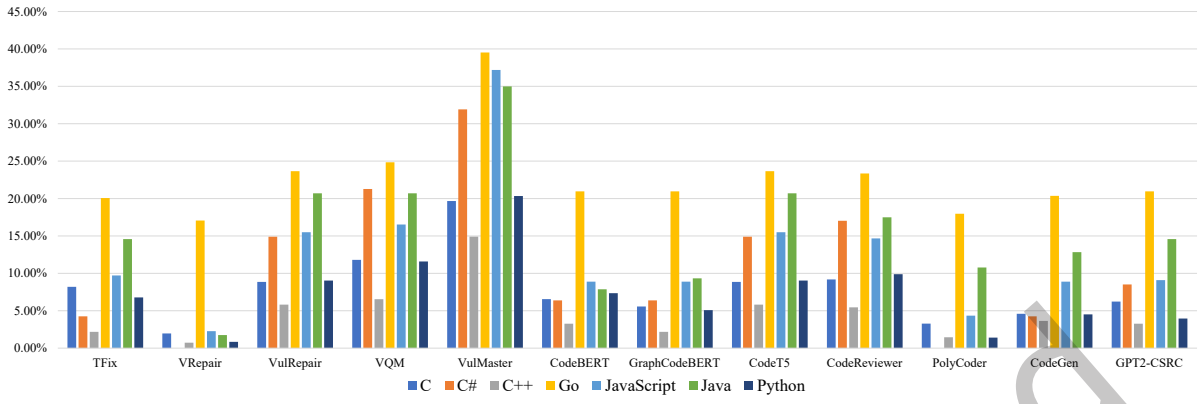


Fig. 4. Performance of AVR and PLM techniques across seven programming languages (y-axis: Exact Match score with beam size 1)

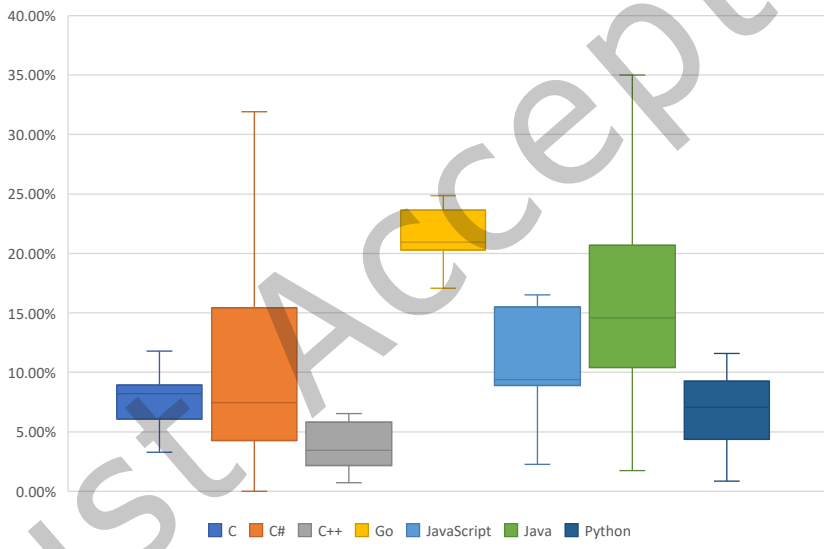


Fig. 5. Overall performance of AVR and PLM techniques across seven programming languages (y-axis: Exact Match score with beam size 1)

Encoder-decoder PLM techniques on average outperform both Encoder-only and Decoder-only techniques across all seven evaluation metrics. Specifically, Encoder-decoder PLM techniques achieve an average improvement of 55.56% over Encoder-only techniques and 65.99% over Decoder-only techniques in terms of the EM with beam 1 metric. Among Encoder-decoder PLM techniques, the best-performing CodeT5 achieves an EM of 14.33% with beam 1. The prevalence of this phenomenon may stem from the fact that the automated vulnerability repair task which gives a vulnerability function and outputs corresponding repair closely aligns with code translation which is the pre-training task for the Encoder-decoder PLM.

At the language level, as shown in Figure 4 and Figure 5, the performance of techniques varies across programming languages. While many models show similar performance patterns, Go consistently achieves the best results, while C/C++ shows the relatively weakest performance. This may be attributed to the relatively recent emergence of the Go programming language and its concise syntax. VulMaster demonstrates superior performance compared to other AVR and PLM techniques across all seven examined programming languages. It shows heterogeneous efficacy depending on the language, achieving its highest accuracy rates in Go (39.52%) and JavaScript (37.19%). In contrast, it encounters more significant challenges in C++ (14.91%) and C (19.67%), while producing moderate outcomes in Python (20.34%), C# (31.91%), and Java (34.99%). Overall, AVR techniques and PLM approaches perform the best performance in Go with a median of 20.96%, followed by Java at 14.58%, JavaScript at 9.71%, C# at 8.51%, C at 8.20%, Python at 7.34% and C++ at 3.64%.

Based on the above data analysis, we observe that Encoder-decoder PLMs perform best among three types of PLM techniques and VulMaster performs best on all seven metrics among all AVRs and PLMs. Given that the backbone model of VulMaster is CodeT5 which belongs to Encoder-decoder PLMs and its input incorporates code AST information and CWE information, researchers would consider utilizing Encoder-decoder architecture and more information about code and CWE knowledge for further improving learning-based automated vulnerability repair in the multilingual vulnerability context. Using Fusion-in-Decoder architecture, the input length of VulMaster which is 5,120 tokens is significantly broadened compared with the input length of the original CodeT5 model which is 512 tokens. Therefore, a model with a larger input length or approach that would broaden the model input length is one direction that researchers could take into consideration.

RQ1 Summary: VulMaster and CodeT5 achieve the best performance among AVR and PLM techniques, with 28.94% and 14.33% in the EM with beam 1 metric, respectively. VulMaster is the best-performing technique across all seven metrics, surpassing all other methods by a margin of 82.82% to 628.97% on the EM with beam 1 metric. All techniques demonstrate optimal performance with Go while showing their relatively poorest results with C/C++.

4.2 RQ2: What is the performance of state-of-the-art LLMs in repairing multilingual vulnerability?

Approach. This research question focuses on comparing language-agnostic LLMs and various learning strategies. We investigated five state-of-the-art LLMs and four prompting strategies to assess their effectiveness. Prior work [23] showed that LLMs perform poorly in automated vulnerability repair when using the special tokens as a part of the prompt. Additionally, closed-source LLMs accessed via API cannot accept special tokens. Thus, to improve the prompt effectiveness and maintain consistency with open-source LLMs, we applied the Sequence-to-Sequence paradigm for all the prompting strategies (including both zero-shot and few-shot) and inference, where the input is a vulnerable function and the expected output is a corresponding repaired function. Moreover, we approach multilingual vulnerability repair as a multi-task learning problem, where repairing vulnerabilities in each programming language represents a distinct task. Drawing from previous research [12, 46], we applied instruction tuning as a multi-task learning strategy to empower LLMs to transfer knowledge across programming languages. On that basis, we can further apply language-specific few-shot prompts to enhance the in-context learning for multilingual vulnerability repair. Due to varying prompting strategies, we restricted the vulnerable function length to 1024 tokens. Section 3.2 and Section 3.4 present detailed information about the LLMs and their strategies. Below, we particularly describe the instruction tuning process.

Our study aims to develop a language-agnostic paradigm for repairing software vulnerabilities using LLMs. To adapt these models to multilingual vulnerability repair, we employed instruction tuning on our complete training dataset of 7,448 function pairs spanning seven programming languages. In the first step, we applied

Table 4. Performance of LLMs and their strategies on multilingual vulnerability repair

Technique	EM with beam 1	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
Zero-shot prompting					
DeepSeek-Coder	0.09%	0.0489	0.2725	0.1354	0.2708
Code Llama	0.37%	0.0606	0.3893	0.2007	0.3864
Llama 3	0.19%	0.0445	0.3189	0.1583	0.3173
GPT-3.5-Turbo	0.23%	0.0588	0.4071	0.2118	0.4055
GPT-4o	0.33%	0.0806	0.4084	0.2188	0.4057
Few-shot prompting					
DeepSeek-Coder	14.15%	0.3857	0.6107	0.5192	0.6079
Code Llama	2.52%	0.0847	0.2859	0.1645	0.2825
Llama 3	11.48%	0.1574	0.4678	0.3491	0.4658
GPT-3.5-Turbo	17.32%	0.3503	0.6595	0.5474	0.6577
GPT-4o	26.89%	0.6193	0.7941	0.7243	0.7920
Instruction-tuning + Zero-shot prompting					
DeepSeek-Coder	0.42%	0.7240	0.8585	0.8143	0.8553
Code Llama	3.08%	0.8361	0.9183	0.8840	0.9158
Llama 3	3.36%	0.4914	0.8274	0.7446	0.8245
GPT-3.5-Turbo	5.79%	0.7067	0.8419	0.7865	0.8373
GPT-4o	7.24%	0.7729	0.8865	0.8408	0.8833
Instruction-tuning + Few-shot prompting					
DeepSeek-Coder	20.96%	0.7579	0.8553	0.8127	0.8502
Code Llama	18.25%	0.7770	0.8557	0.8128	0.8498
Llama 3	18.30%	0.3496	0.6513	0.5819	0.6478
GPT-3.5-Turbo	16.34%	0.7617	0.8701	0.8255	0.8662
GPT-4o	**28.71%**	**0.8448**	**0.9232**	**0.8936**	**0.9202**

zero-shot prompting to convert the multilingual training set into an instruction-filled fine-tuning set, consisting of <*structured instruction*, *vulnerable code*, *repaired code*>. Next, we applied Low-Rank Adaption (LoRA) to a subset of weight matrices in the LLM, adapting only the attention weights for multilingual vulnerability repair while freezing the multilayer perceptron (MLP) modules to reduce trainable parameters. We then trained the LoRA-based LLM on the instruction-filled dataset using supervised learning. After we obtained the instruction-tuned LLMs, we used both zero-shot and few-shot prompting strategies (as described in Section 3.4) to generate the repaired functions. For example, in the few-shot prompting strategy, we used BM25 to retrieve three language-specific pairs of vulnerable and repaired functions as examples, then prompted the instruction-tuned LLM to generate the fixes.

To evaluate the effectiveness of the studied LLMs and their strategies, we used the similar metrics as in RQ1: Exact Match (EM) with beam size 1, BLEU score, and ROUGE-1, ROUGE-2, and ROUGE-L metrics. We limit our analysis to beam size of 1 to maintain consistency between the open-source and closed-source models, as closed-source models do not support beam search. In addition, we performed a McNemar test [43], a non-parametric statistical method, to assess the statistical significance of differences between the best-performing LLM-based approach and VulMaster (the best-performing existing AVR technique) in terms of EM with beam 1.

Results. Table 4 illustrates the performance of five LLMs under four different strategies in terms of EM, BLEU, and ROUGE scores in the multilingual vulnerability repair context. Within each metric, the values in bold indicate the technique that exhibits the best performance among all LLMs with different strategies. Figure 6 and Figure 7 present the performance of LLMs of the few-shot prompting strategy and instruction-tuning with few-shot prompting strategy across seven programming languages in terms of EM with beam size 1.

First, under the zero-shot prompting strategy, both open-source and closed-source LLMs exhibit poor performance on this multilingual vulnerability repair task. Nevertheless, the performance of LLMs is extremely improved by repairing vulnerable code with the few-shot prompting strategy which uses the BM25 algorithm to retrieve three examples from the training dataset with the highest similarity. Specifically, among all LLMs evaluated, Code Llama achieves an EM of 0.37% using a beam size of 1 under zero-shot prompting conditions, which modestly increases to 2.52% with few-shot prompting, marking the lowest relative gain at 581.08%. In contrast, DeepSeek-Coder begins with an EM of 0.09% under zero-shot conditions, which dramatically escalates to 14.15% with few-shot prompting, resulting in the highest relative improvement recorded at 15,622.22% in terms of EM with beam 1. This shows the strong power of prompt engineering and implies that providing more information to LLMs could get more valuable results.

Second, instruction-tuning boosts the performance of the majority of models in terms of EM with beam 1 and also improves some models in terms of BLEU and ROUGE metrics. For example, instruction-tuning Code Llama with few-shot prompting outperforms Code Llama with few-shot prompting with an improvement of 624.21% in terms of EM with beam 1. This suggests that the instruction-tuning strategy can substantially enhance the performance of LLMs in multilingual vulnerability repair. Additionally, upon inspecting their code generation metric results, we observed that despite the small instruction-tuning epoch due to limited resources, few LLMs have learned the repair pattern for multilingual vulnerability repair to some extent.

Additionally, among the five models employing four prompting strategies, the GPT-4o model with instruction-tuning and few-shot prompting achieves the best performance. It records scores of 28.71% in EM with beam 1, 0.8448 in BLEU-4, 0.9232 in ROUGE-1, 0.8936 in ROUGE-2, and 0.9202 in ROUGE-L. Moreover, Table 5 presents the results of the McNemar test conducted to compare the performance of the instruction-tuning GPT-4o model with few-shot prompting strategy and VulMaster. The p-value obtained from the McNemar test is 0.8314, which is well above the conventional significance threshold of 0.05. This indicates that there is no statistically significant difference between the classification results of GPT-4o and VulMaster. Additionally, the odds ratio (OR) is reported as 0.9721. This value, close to 1, suggests that the difference in classification errors between the two models is minimal, with neither model demonstrating a notable advantage over the other. In conclusion, results of the McNemar test suggest that there is no significant difference in performance between GPT-4o and VulMaster in multilingual vulnerability repair, which exhibits the strong power of instruction-tuning with few-shot prompting strategy.

Table 5. McNemar test results between GPT-4o and VulMaster

Approach	McNemar.p	McNemar.OR
GPT-4o vs. VulMaster	0.8314	0.9721

In Figure 6 and Figure 7, owing to the subpar performance observed with zero-shot prompting and instruction-tuning with zero-shot prompting strategies, we present only the results from models utilizing few-shot prompting and instruction-tuning with few-shot prompting strategies. Consistent with the findings from RQ1, the LLMs demonstrate comparable performance across different programming languages, with the highest performance noted in Go and the lowest in C/C++. Instruction-tuning GPT-4o with few-shot prompting shows its best

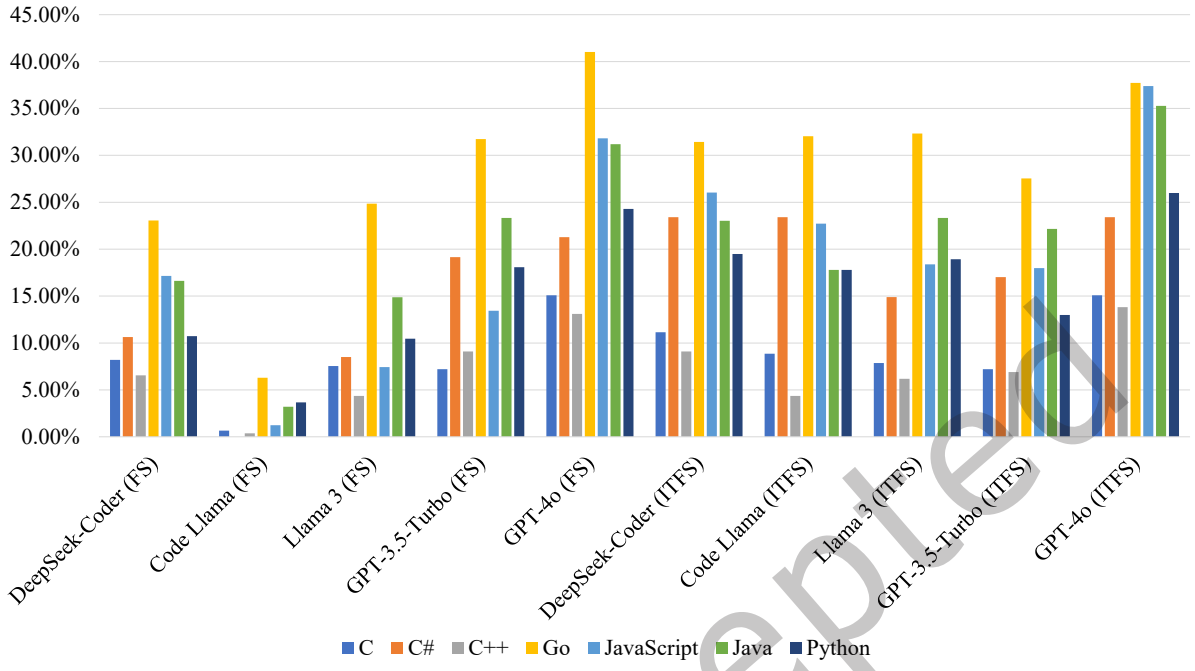


Fig. 6. Performance of LLM techniques across seven programming languages (y-axis: Exact Match score with beam size 1, FS: few-shot prompting, and ITFS: instruction-tuning with few-shot prompting)

performance in C, C#, C++, JavaScript, Java, and Python, achieving EM with beam 1 scores of 15.08%, 23.40%, 13.82%, 37.40%, 35.28%, and 25.99%, respectively. Overall, LLM techniques perform best in Go with a median of 31.59%, followed by Java at 22.59%, JavaScript at 18.18%, C# at 18.09%, Python at 17.94%, C at 8.03% and C++ at 6.73%. Notably, instruction-tuning of both DeepSeek-Coder and Code Llama with few-shot prompting achieved the same results as GPT-4o on C#, indicating the effectiveness of instruction-tuning in enhancing the capabilities of LLMs. Interestingly, GPT-4o with few-shot prompting outperformed its instruction-tuned counterpart in Go, with an EM with beam 1 score of 41.02%. This could be attributed to the limited scale of the existing fine-tuning datasets, which may not fully unleash the potential of GPT-4o.

RQ2 Summary: Instruction-tuning with few-shot prompting strategy is the best technique among all strategies for LLMs on multilingual vulnerability repair and instruction-tuning GPT-4o with few-shot prompting strategy achieves the best performance of 28.71% in terms of EM with beam 1. The performances of LLMs vary from different programming languages and LLMs perform best on Go and worst on C/C++, which is similar to AVR and PLM techniques.

4.3 RQ3: What are the strengths and weaknesses of the existing automated vulnerability techniques?

Approach. This research question aims to gain an understanding of the performance characteristics of various AVR, PLM techniques, and LLM approaches with different strategies. Therefore, following the experimental

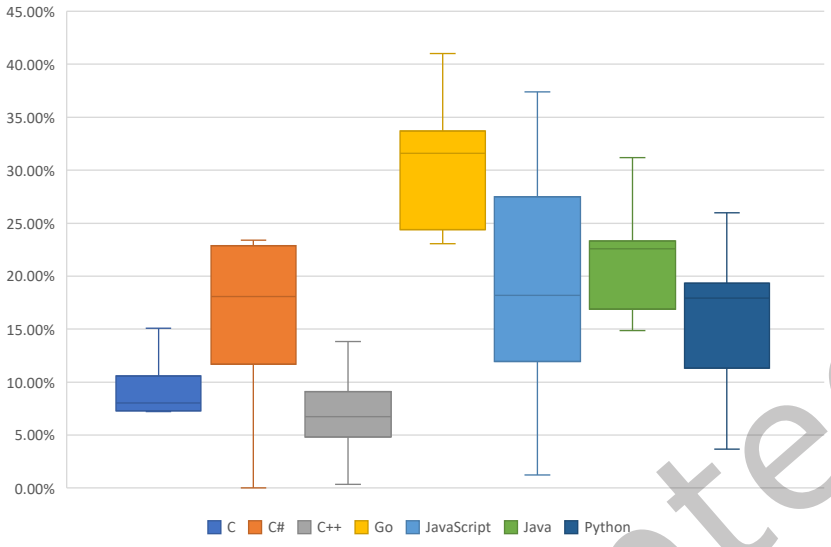


Fig. 7. Overall performance of LLM techniques across seven programming languages (y-axis: Exact Match score with beam size 1)

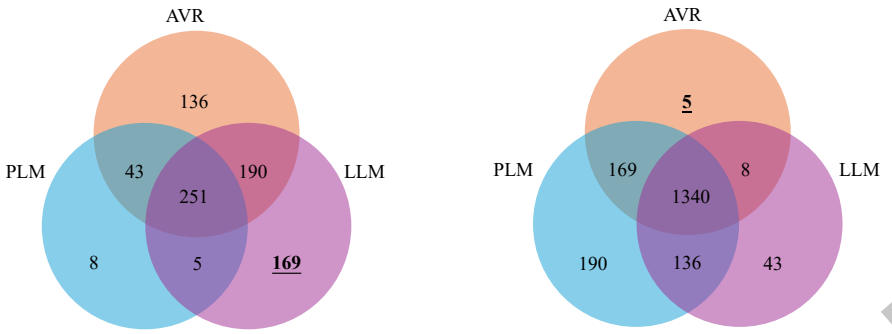
design of RQ1 and RQ2, we conducted a comprehensive analysis to explore the degree of their orthogonality from two different perspectives:

- **Between AVR techniques.** Based on the findings of RQ1 and RQ2, we selected the best-performing AVR technique, PLM technique, and LLM technique. Specifically, the selected techniques are VulMaster for AVR techniques, CodeT5 for three PLM architectures, and instruction-tuning GPT-4o with few-shot prompting for LLM techniques.
- **Between LLM strategies.** Based on the findings of RQ2, we selected the best-performing LLM techniques within four LLM strategies, i.e., zero-shot prompting (ZS), few-shot prompting (FS), instruction-tuning with zero-shot prompting (ITZS), and instruction-tuning with few-shot prompting strategies (ITFS). Specifically, the selected techniques are Code Llama with zero-shot prompting, GPT-4o with few-shot prompting, instruction-tuning GPT-4o with zero-shot prompting, and instruction-tuning GPT-4o with few-shot prompting, each representing their respective LLM strategies.

Based on the above two perspectives, we further conducted a two-level analysis: (1) the unique correct/incorrect repairs, and (2) the repair performance across the dangerous CWE-IDs. Regarding the first level, we employed Venn diagrams to assess the unique correct/incorrect repairs across various studied AVR techniques. Regarding the second level, we investigated the repair performance of studied techniques using testing data based on the 2023 CWE Top 25 Most Dangerous Software Weaknesses, as published by the CWE community.[3] Among the top 25 dangerous CWE-IDs, all of them are involved in our testing data. Finally, we investigate the highest and lowest CWEs in terms of the average EM scores with beam 1, across all AVR, PLM, and LLM techniques for seven programming languages.

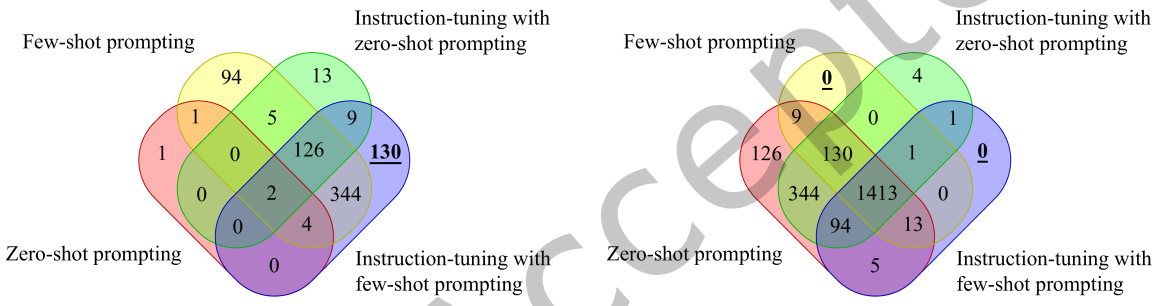
Results. Figure 8 and Figure 9 present the Venn diagrams that demonstrate the intersection of correct/incorrect repairs among the studied AVR techniques based on two analyzed perspectives (i.e., AVR techniques and LLM

[3]https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html



(a) Unique correct repair of three techniques (b) Unique incorrect repair of three techniques

Fig. 8. Unique correct repair and unique incorrect repair across three representative techniques



(a) Unique correct repair of four strategies (b) Unique incorrect repair of four strategies

Fig. 9. Unique correct repair and unique incorrect repair across four LLM strategies

strategies) in terms of EM with beam 1. Overlap areas denote shared correct/incorrect repairs among multiple AVR techniques, while non-overlapping areas signify the unique correct/incorrect repairs of each AVR technique. Table 6 and Table 7 further show the repair performance of studied AVR techniques and the LLM strategies on the top 25 most dangerous CWE-IDs in 2023 in terms of EM with beam 1 and the values highlighted in bold denote the technique or strategy that exhibits the best performance for corresponding CWE-ID. Table 8 further displays the highest and lowest CWEs in terms of the average EM scores with beam 1, across all AVR, PLM, and LLM techniques for seven programming languages. Table 9 provides an analysis of the best and worst performing CWEs in relation to the most effective LLM and AVR techniques.

Between AVR techniques. From Figure 8a, we find that the LLM technique achieves the best performance compared to AVR and PLM in terms of the unique correct repairs. Specifically, the LLM technique demonstrates 169 unique correct repairs, significantly outperforming AVR, which achieves 136 unique correct repairs (a 24.26% improvement), and PLM, which only has 8 unique correct repairs (a 2012.50% improvement). From Figure 8b, we find that the LLM technique also outperforms PLM in terms of unique incorrect repairs. It only has 43 unique incorrect repairs, which is significantly fewer than PLM (190). These results demonstrate the effectiveness of instruction-tuning with few-shot prompting strategy for LLMs in multilingual vulnerability repair.

Table 6. The Exact Match score with beam size 1 of three techniques across the Top 25 Most Dangerous CWE-IDs in 2023

Rank	CWE-ID	PLM	AVR	LLM	Total
1	CWE-787 (Out-of-bounds Write)	11(14.47%)	**12(15.79%)**	**12(15.79%)**	76
2	CWE-79 (Cross-site Scripting)	53(21.03%)	**106(42.06%)**	103(40.87%)	252
3	CWE-89 (SQL Injection)	6(9.38%)	14(21.88%)	**20(31.25%)**	64
4	CWE-416 (Use After Free)	4(13.33%)	**8(26.67%)**	**8(26.67%)**	30
5	CWE-78 (OS Command Injection)	2(7.41%)	**6(22.22%)**	5(18.52%)	27
6	CWE-20 (Improper Input Validation)	11(10.78%)	27(26.47%)	**28(27.45%)**	102
7	CWE-125 (Out-of-bounds Read)	0(0.00%)	**2(4.55%)**	0(0.00%)	44
8	CWE-22 (Path Traversal)	1(1.18%)	11(12.94%)	**17(20.00%)**	85
9	CWE-352 (Cross-Site Request Forgery)	12(23.53%)	**18(35.29%)**	16(31.37%)	51
10	CWE-434 (Unrestricted Upload of File with Dangerous Type)	0(0.00%)	**1(20.00%)**	**1(20.00%)**	5
11	CWE-862 (Missing Authorization)	0(0.00%)	0(0.00%)	0(0.00%)	4
12	CWE-476 (NULL Pointer Dereference)	5(7.69%)	12(18.46%)	**14(21.54%)**	65
13	CWE-287 (Improper Authentication)	82(63.57%)	**93(72.09%)**	**93(72.09%)**	129
14	CWE-190 (Integer Overflow or Wraparound)	4(8.16%)	**10(20.41%)**	6(12.24%)	49
15	CWE-502 (Deserialization of Untrusted Data)	5(18.52%)	**11(40.74%)**	8(29.63%)	27
16	CWE-77 (Command Injection)	3(7.89%)	8(21.05%)	**18(47.37%)**	38
17	CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer)	2(7.14%)	4(14.29%)	**5(17.86%)**	28
18	CWE-798 (Use of Hard-coded Credentials)	0(0.00%)	0(0.00%)	**1(50.00%)**	2
19	CWE-918 (Server-Side Request Forgery)	2(6.25%)	**7(21.88%)**	**7(21.88%)**	32
20	CWE-306 (Missing Authentication for Critical Function)	2(18.18%)	**7(63.64%)**	**7(63.64%)**	11
21	CWE-362 (Race Condition)	0(0.00%)	**5(45.45%)**	**5(45.45%)**	11
22	CWE-269 (Improper Privilege Management)	0(0.00%)	3(23.08%)	**4(30.77%)**	13
23	CWE-94 (Code Injection)	2(8.70%)	3(13.04%)	**6(26.09%)**	23
24	CWE-863 (Incorrect Authorization)	3(9.09%)	**7(21.21%)**	**7(21.21%)**	33
25	CWE-276 (Incorrect Default Permissions)	0(0.00%)	0(0.00%)	0(0.00%)	3
	Average	210(17.44%)	375(31.15%)	**391(32.48%)**	1204

The number in the parentheses represent the corresponding accuracy of techniques and Total column represents the total amount of data for corresponding CWE-ID in our test dataset.

From Table 6, we observe that the LLM technique outperforms the other two categories, AVR and PLM. On average, LLM successfully repairs 32.48% of the vulnerable functions (1204) associated with the top 25 most dangerous CWE-IDs in 2023, surpassing the performance of AVR (31.15%) and PLM (17.44%). Notably, the LLM technique achieves the best performance on 17 out of the 25 CWE-IDs, compared to 14 for AVR and none for PLM. These findings underscore the efficacy of LLM in addressing dangerous vulnerabilities, thereby reinforcing the results presented in RQ2.

Between LLM strategies. From Figure 9a, we observe that the ITFS technique achieves the best performance in terms of unique correct repairs when compared to ZS, FS, and ITZS. The ITFS technique achieves 130 unique correct repairs, which represents a substantial improvement over the other strategies. Specifically, it outperforms ZS, which achieves only 1 unique correct repair, resulting in an improvement of 12,900.00%. It also surpasses ITZS, which yields 13 unique correct repairs, with an improvement of 900.00%, and FS, which produces 94 unique correct repairs, showing an improvement of 38.30%. Moreover, as illustrated in Figure 9b, the LLM technique also excels in avoiding unique incorrect repairs, with no such cases identified—significantly fewer than ZS (126) and ITZS (4). These results emphasize the effectiveness of instruction-tuning with few-shot prompting strategy in enhancing LLM performance for multilingual vulnerability repair.

Table 7 provides additional evidence of the superior performance of ITFS compared to the other three LLM techniques. On average, the ITFS approach successfully repairs vulnerabilities in 32.48% of the functions associated with the top 25 most dangerous CWE-IDs, demonstrating a significant improvement over other methods. Specifically, it outperforms ITZS, which repairs only 10.55% of these vulnerable functions, achieving a 207.87% relative

Table 7. The Exact Match score with beam size 1 of four LLM strategies across the Top 25 Most Dangerous CWE-IDs in 2023

Rank	ID	ZS	FS	ITZS	ITFS	total
1	CWE-787 (Out-of-bounds Write)	0(0.00%)	**13(17.11%)**	5(6.58%)	12(15.79%)	76
2	CWE-79 (Cross-site Scripting)	0(0.00%)	87(34.52%)	32(12.70%)	**103(40.87%)**	252
3	CWE-89 (SQL Injection)	1(1.56%)	17(26.56%)	1(1.56%)	**20(31.25%)**	64
4	CWE-416 (Use After Free)	0(0.00%)	6(20.00%)	0(0.00%)	**8(26.67%)**	30
5	CWE-78 (OS Command Injection)	0(0.00%)	4(14.81%)	0(0.00%)	**5(18.52%)**	27
6	CWE-20 (Improper Input Validation)	0(0.00%)	**28(27.45%)**	4(3.92%)	**28(27.45%)**	102
7	CWE-125 (Out-of-bounds Read)	0(0.00%)	**1(2.27%)**	0(0.00%)	0(0.00%)	44
8	CWE-22 (Path Traversal)	0(0.00%)	12(14.12%)	0(0.00%)	**17(20.00%)**	85
9	CWE-352 (Cross-Site Request Forgery)	0(0.00%)	**17(33.33%)**	3(5.88%)	16(31.37%)	51
10	CWE-434 (Unrestricted Upload of File with Dangerous Type)	0(0.00%)	**1(20.00%)**	0(0.00%)	**1(20.00%)**	5
11	CWE-862 (Missing Authorization)	0(0.00%)	0(0.00%)	0(0.00%)	0(0.00%)	4
12	CWE-476 (NULL Pointer Dereference)	2(3.08%)	**14(21.54%)**	4(6.15%)	**14(21.54%)**	65
13	CWE-287 (Improper Authentication)	1(0.78%)	90(69.77%)	74(57.36%)	**93(72.09%)**	129
14	CWE-190 (Integer Overflow or Wraparound)	0(0.00%)	**10(20.41%)**	0(0.00%)	6(12.24%)	49
15	CWE-502 (Deserialization of Untrusted Data)	0(0.00%)	**8(29.63%)**	0(0.00%)	**8(29.63%)**	27
16	CWE-77 (Command Injection)	1(2.63%)	15(39.47%)	3(7.89%)	**18(47.37%)**	38
17	CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer)	0(0.00%)	3(10.71%)	1(3.57%)	**5(17.86%)**	28
18	CWE-798 (Use of Hard-coded Credentials)	0(0.00%)	0(0.00%)	0(0.00%)	**1(50.00%)**	2
19	CWE-918 (Server-Side Request Forgery)	0(0.00%)	5(15.63%)	0(0.00%)	**7(21.88%)**	32
20	CWE-306 (Missing Authentication for Critical Function)	0(0.00%)	**7(63.64%)**	0(0.00%)	**7(63.64%)**	11
21	CWE-362 (Race Condition)	0(0.00%)	**5(45.45%)**	0(0.00%)	**5(45.45%)**	11
22	CWE-269 (Improper Privilege Management)	0(0.00%)	**5(38.46%)**	0(0.00%)	4(30.77%)	13
23	CWE-94 (Code Injection)	1(4.35%)	4(17.39%)	0(0.00%)	**6(26.09%)**	23
24	CWE-863 (Incorrect Authorization)	0(0.00%)	5(15.15%)	0(0.00%)	**7(21.21%)**	33
25	CWE-276 (Incorrect Default Permissions)	0(0.00%)	0(0.00%)	0(0.00%)	0(0.00%)	3
	Average	6(0.50%)	357(29.65%)	127(10.55%)	**391(32.48%)**	1204

The number in the parentheses represent the corresponding accuracy of techniques and Total column represents the total amount of data for corresponding CWE-ID in our test dataset.

improvement. It also surpasses FS, which repairs 29.65% of the vulnerabilities, reflecting a 9.52% enhancement in effectiveness. In comparison to ZS, which addresses only 0.50% of the vulnerable functions, ITFS achieves a striking improvement of 6416.67%, vastly outperforming ZS. Notably, ITFS achieves the best performance on 18 out of the 25 CWE-IDs, compared to 11 for FS and none for ITZS and ZS. These findings further underscore the efficacy of instruction tuning with few-shot prompting in significantly improving LLM capabilities for addressing dangerous vulnerabilities, reinforcing the results presented in RQ2.

From Table 8, we can see that the performance of the techniques varies across different languages. For instance, the language C exhibits the highest repair rate for CWE-327 (Use of a Broken or Risky Cryptographic Algorithm) with an EM of 31.25%, yet fails to repair CWE-121 (Stack-based Buffer Overflow), scoring 0.00%. Similarly, C# shows a considerable ability to repair CWE-918 (Server-Side Request Forgery) with an EM of 23.13%, but is ineffective at repairing CWE-347 (Improper Verification of Cryptographic Signature), which also scores 0.00%. Notably, Go achieves the highest EM among the languages with a 60.25% repair rate for CWE-287 (Improper Authentication), contrasting with its inability to repair CWE-208 (Observable Timing Discrepancy). This result underscores the varied effectiveness of vulnerability repair techniques across different programming environments and specific vulnerabilities, pointing to potential areas for further refinement in repair capabilities.

Table 9 presents the top 10 best and worst performing CWEs for GPT-4o with instruction-tuning and few-shot prompting strategies and VulMaster in the context of multilingual vulnerability repair. To mitigate bias due to small sample sizes, we only consider CWEs with a count of at least 5. As illustrated in the table, there are 6 CWEs that both GPT-4o and VulMaster perform best on, and 7 CWEs where both perform worst. Notably, there are 2 CWEs - CWE-359 (Exposure of Private Personal Information to an Unauthorized Actor) and CWE-532

Table 8. Highest and lowest average Exact Match score with beam size 1 of all techniques across seven programming languages

Language	Highest EM CWE	EM with beam 1	Lowest EM CWE	EM with beam 1
C	CWE-327: Use of a Broken or Risky Cryptographic Algorithm	31.25%	CWE-121: Stack-based Buffer Overflow	0.00%
C#	CWE-918: Server-Side Request Forgery (SSRF)	23.13%	CWE-347: Improper Verification of Cryptographic Signature	0.00%
C++	CWE-191: Integer Underflow (Wrap or Wraparound)	37.50%	CWE-116: Improper Encoding or Escaping of Output	0.00%
Go	CWE-287: Improper Authentication	60.25%	CWE-208: Observable Timing Discrepancy	0.00%
JavaScript	CWE-285: Improper Authorization	22.57%	CWE-116: Improper Encoding or Escaping of Output	0.00%
Java	CWE-681: Incorrect Conversion between Numeric Types	40.63%	CWE-1236: Improper Neutralization of Formula Elements in a CSV File	0.00%
Python	CWE-522: Insufficiently Protected Credentials	45.31%	CWE-1021: Improper Restriction of Rendered UI Layers or Frames	0.00%

Table 9. Top 10 best and worst performing CWEs for GPT-4o and VulMaster in multilingual vulnerability repair

	GPT-4o				VulMaster			
Top 10	CWE	Name	Accuracy	Count	CWE	Name	Accuracy	Count
Best	287	Improper Authentication	72.09%	129	287	Improper Authentication	72.09%	129
	306	Missing Authentication for Critical Function	63.64%	11	610	Externally Controlled Reference to a Resource in Another Sphere	71.43%	7
	285	Improper Authorization	53.85%	13	306	Missing Authentication for Critical Function	63.64%	11
	77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	47.37%	38	521	Weak Password Requirements	54.55%	11
	362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	45.45%	11	284	Improper Access Control	52.17%	23
	79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	40.87%	252	91	XML Injection (aka Blind XPath Injection)	46.15%	13
	359	Exposure of Private Personal Information to an Unauthorized Actor	40.00%	5	285	Improper Authorization	46.15%	13
	669	Incorrect Resource Transfer Between Spheres	40.00%	5	362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	45.45%	11
	532	Insertion of Sensitive Information into Log File	40.00%	5	754	Improper Check for Unusual or Exceptional Conditions	45.45%	11
	284	Improper Access Control	39.13%	23	79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	42.06%	252
Worst	401	Missing Release of Memory after Effective Lifetime	0.00%	5	674	Uncontrolled Recursion	0.00%	5
	125	Out-of-bounds Read	0.00%	44	359	Exposure of Private Personal Information to an Unauthorized Actor	0.00%	5
	835	Loop with Unreachable Exit Condition ('Infinite Loop')	0.00%	16	824	Access of Uninitialized Pointer	0.00%	5
	134	Use of Externally-Controlled Format String	0.00%	7	350	Reliance on Reverse DNS Resolution for a Security-Critical Action	0.00%	5
	338	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	0.00%	5	532	Insertion of Sensitive Information into Log File	0.00%	5
	824	Access of Uninitialized Pointer	0.00%	5	255	Credentials Management Errors	0.00%	5
	674	Uncontrolled Recursion	0.00%	5	401	Missing Release of Memory after Effective Lifetime	0.00%	5
	369	Divide By Zero	5.88%	17	639	Authorization Bypass Through User-Controlled Key	0.00%	6
	617	Reachable Assertion	5.88%	17	125	Out-of-bounds Read	4.55%	44
	415	Double Free	7.69%	13	835	Loop with Unreachable Exit Condition ('Infinite Loop')	6.25%	16

The CWE highlighted in green represents the best-performing CWE intersection between GPT-4o and VulMaster, while the CWE highlighted in red indicates the worst-performing CWE intersection between GPT-4o and VulMaster.

(Insertion of Sensitive Information into Log File) - that are among the worst performing for VulMaster but are among the best performing for GPT-4o. Moreover, although CWE-125 (Out-of-Bounds Read) and CWE-835 (Loop with Unreachable Exit Condition, i.e., 'Infinite Loop') are among the most frequently occurring CWEs where both GPT-4o and VulMaster perform poorly, GPT-4o failed to repair any instances of these vulnerabilities, whereas VulMaster successfully repaired at least some instances. This suggests that while there is some overlap in performance between the two approaches, each also has unique strengths and weaknesses. To further analyze these results, we draw upon CWE-1000 [45] (Research Concepts), which is intended to support research into software weaknesses and their interdependencies, thereby providing a foundation for systematically identifying theoretical gaps within CWE. This view is primarily organized around behavioral abstractions and is explicitly designed to encompass all weaknesses defined in CWE. By classifying each CWE into the CWE-1000 view, we observe that both GPT-4o and VulMaster perform well on CWE-284 (Improper Access Control) and CWE-707 (Improper Neutralization). Conversely, both approaches perform poorly on CWE-664 (Improper Control of a Resource Through its Lifetime) and CWE-691 (Insufficient Control Flow Management). The complete visualization results can be found in our replicated package.

RQ3 Summary: The instruction-tuned LLM (i.e., GPT-4o) with few-shot prompting, compared to AVR and PLM techniques, not only achieves the highest number of unique correct repairs but also completely avoids incorrect ones. Furthermore, it performs relatively best on the top 25 most dangerous CWE-IDs, showcasing the potential of LLMs in repairing critical multilingual vulnerabilities.

Table 10. Performance of LLMs on TypeScript vulnerability repair

Technique	EM with beam 1	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
VulMaster	5.88%	0.3676	0.6877	0.4847	0.6741
GPT-4o	28.57%	0.7267	0.8453	0.7855	0.8439

4.4 RQ4: What is the generalization capability in repairing previously unseen vulnerability?

Approach. Although current AVR methods and LLMs perform well on the REEF dataset, their ability to generalize to multilingual vulnerability repair remains uncertain. To address this, we evaluate the leading AVR method, VulMaster, alongside the top-performing LLM, GPT-4o, utilizing instruction-tuning and few-shot prompting strategy on vulnerabilities in programming languages that have not been previously encountered. Specifically, we extract vulnerabilities in the TypeScript programming language from the CVEfixes dataset, which provides both the vulnerabilities and their corresponding repairs at the function level. After retrieving the TypeScript data, we adhere to the data processing procedures outlined in Section 3.1 and apply the same dataset division strategy. This results in a collection of 593 TypeScript vulnerability instances, with 119 reserved for evaluation.

Results. Table 10 presents the performance results of VulMaster and GPT-4o, employing instruction-tuning and few-shot prompting strategies for TypeScript vulnerability repair. As illustrated in the table, VulMaster achieves an EM score of 5.88%, while GPT-4o attains a score of 28.57%, which is 385.88% higher than VulMaster. Furthermore, GPT-4o achieves a 28.71% EM score on the REEF dataset, with its performance on TypeScript vulnerabilities surpassing this benchmark. In contrast, VulMaster, which performs at 28.94% on the REEF dataset, experiences a significant decline in performance when applied to TypeScript vulnerabilities. This suggests the robust generalization capabilities of GPT-4o compared to the limited generalization of VulMaster. The superior performance of GPT-4o may be attributed to the strong comprehension abilities inherent in LLMs, whereas domain-specific AVR methods like VulMaster struggle to generalize to unseen data. These findings underscore the potential of LLMs in multilingual vulnerability repair.

RQ4 Summary: On the unseen TypeScript vulnerabilities, GPT-4o with instruction-tuning and few-shot prompting strategies achieves an EM score of 28.57%, outperforming VulMaster by 385.88%. These results demonstrate the strong generalization capability of LLMs in handling vulnerabilities in previously unseen programming languages.

5 DISCUSSION

5.1 Advanced prompting strategies

Although we have investigated zero-shot prompting, few-shot prompting, and instruction-tuning strategies for multilingual vulnerability repair, several advanced prompting techniques, such as Chain-of-Thought (CoT)[76] remain underexplored and may yield different levels of performance. CoT prompting guides LLMs to reason through problems step by step, significantly enhancing performance on reasoning-intensive tasks. In this study, we take an initial step toward evaluating the effectiveness of CoT prompting in the context of multilingual vulnerability repair, with the goal of further enriching the scope and depth of our research. To this end, we construct CoT prompts by extending the zero-shot prompt. Specifically, following prior work [82], we append the sentence "Let's think step by step." to the end of each instruction to form the CoT prompt. We then assess the performance of both CoT prompting and instruction-tuning combined with CoT prompting across all open-source and closed-source LLMs evaluated in the preceding experiments.

Table 11. Performance of LLMs and CoT strategies on multilingual vulnerability repair

Technique	EM with beam 1	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
CoT prompting					
DeepSeek-Coder	0.14%	0.1094	0.2796	0.1765	0.2746
Code Llama	0.14%	0.2109	0.5269	0.4218	0.5200
Llama 3	0.09%	0.0680	0.3200	0.1926	0.3181
GPT-3.5-Turbo	0.05%	0.0576	0.4032	0.2090	0.4013
GPT-4o	0.05%	0.0755	0.3822	0.2008	0.3789
Instruction-tuning + CoT prompting					
DeepSeek-Coder	0.19%	0.2464	0.4930	0.4625	0.4880
Code Llama	1.82%	0.3073	0.7381	0.6748	0.7308
Llama 3	0.00%	0.1854	0.7070	0.6236	0.7039
GPT-3.5-Turbo	5.79%	0.7117	0.8435	0.7876	0.8386
GPT-4o	**7.38%**	**0.7707**	**0.8859**	**0.8397**	**0.8823**

Table 11 presents the performance of various LLMs under different CoT prompting strategies on the multilingual vulnerability repair task. Notably, GPT-4o, when combined with instruction-tuning and CoT prompting, outperforms all other open-source and closed-source LLMs across EM, BLEU, and ROUGE metrics, demonstrating its superior capability for this task—a finding that aligns with the conclusions drawn in RQ2. Moreover, all closed-source LLMs show substantial performance gains when instruction-tuning is added to CoT prompting. Specifically, when employing instruction-tuning and CoT prompting strategies, GPT-3.5-Turbo and GPT-4o achieve EM score of 5.79% and 7.38%, respectively. In contrast, the same models using standard CoT prompting strategy yield significantly lower performance, with both GPT-3.5-Turbo and GPT-4o attaining only 0.05% EM score. In contrast, open-source LLMs exhibit mixed results under the same setting: Code Llama and DeepSeek-Coder benefit from instruction-tuning with CoT prompting, while Llama 3 shows a decline in performance. This suggests that the impact of instruction-tuning on reasoning ability may vary significantly across LLMs. Finally, even the best-performing LLM, GPT-4o with instruction tuning and CoT prompting, which achieves 7.38% EM, falls significantly short of its performance with instruction tuning and few-shot prompting, which achieves 28.71% EM (Table 4). This observation suggests that CoT prompting may not be the most effective strategy for multilingual vulnerability repair.

5.2 The effectiveness of larger LLMs

In this study, we primarily evaluate open-source LLMs with 7B parameters, which demonstrate strong performance on the multilingual vulnerability repair task. Prior research [25, 75] has shown that increasing model size can enhance the effectiveness of instruction tuning. However, other studies [47] have found that smaller, task-specialized models may outperform larger ones depending on the task. To explore whether this holds in our case, we extend our experiments to larger variants of the three open-source LLMs previously evaluated. Specifically, we examine DeepSeek-Coder-33B-Instruct, Code-Llama-13B-Instruct, Code-Llama-34B-Instruct, Code-Llama-70B-Instruct, and LLaMA 3-70B-Instruct. The training and inference settings are consistent with those described in Section 3.6. Based on the findings of RQ2, we adopt the most effective prompting strategy—instruction tuning combined with few-shot prompting—for all evaluations.

Table 12 presents the performance of the larger models on the multilingual vulnerability repair task. Overall, the results indicate substantial variation across different LLMs. Specifically, Llama 3 (70B) achieves the highest

Table 12. Performance of larger LLMs on multilingual vulnerability repair

Technique	EM with beam 1	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
Instruction-tuning + Few-shot prompting					
DeepSeek-Coder (33B)	20.31%	**0.6718**	0.7600	0.7235	0.7556
Code Llama (13B)	0.00%	0.0211	0.0475	0.0215	0.0450
Code Llama (34B)	0.00%	0.0238	0.0507	0.0244	0.0479
Code Llama (70B)	0.47%	0.1836	0.3925	0.2637	0.3781
Llama 3 (70B)	**28.90%**	0.5067	**0.8644**	**0.8048**	**0.8627**

EM score among the larger models at 28.90%, while DeepSeek-Coder (33B) attains a score of 20.31%. In contrast, all models in the Code Llama series (13B, 34B, and 70B) perform poorly, with EM scores close to 0.00%, underperforming their smaller counterparts. Compared to the results shown in Table 4, Llama 3 (70B) outperforms Llama 3 (8B) by 57.92%, with the latter achieving an EM score of 18.30%. However, DeepSeek-Coder (6.7B) achieves a higher EM score of 20.96%, outperforming its 33B counterpart by 3.20%. Similarly, Code Llama (7B) reaches an EM score of 18.25%, while all larger variants in the same series almost entirely fail to repair any vulnerabilities. These results suggest that although larger open-source LLMs have the potential to enhance performance on the multilingual vulnerability repair task, improvements are not guaranteed; in some cases, smaller and more efficient models may yield better results.

5.3 Explanations for the low fix rate on multilingual vulnerability

The instruction-tuning GPT-4o with few-shot prompting strategy showed varying performance across programming languages, with the best results for Go and the worst for C/C++. To understand the reasons for incorrect fixes and improve future LLM-based approaches, we manually analyzed GPT-4o's results. We randomly selected 200 vulnerable-fixed code pairs from the test set and examined the incorrect fixes generated for both Go and C/C++. Specifically, the first author completed the manual coding analysis by referring to the work of Zhou et al. [89]. Five reasons were classified: error localization, multiple chunks errors, logical errors, context format errors, and semantic equivalence.

Table 13 presents the frequency of five main reasons for incorrect fixes. As shown in the table, a significant portion of the errors in the GPT-4o results resulted from localization issues (49% and 43% for Go and C/C++, separately). Additionally, many instances involved multiple or incorrect modifications to the original code. The remaining fixes could be classified as correct vulnerability repairs, with the following observations: GPT-4o did

Table 13. Manual check result for GPT-4o on Go and C/C++

Language	Go	C/C++
Error localization	49	43
Multiple chunk errors	12	9
Logical errors	29	25
Context format errors	5	8
Semantic equivalence	5	15
Total	100	100

An CWE-319 Example of Error localization

1
```
func setUserSession(ctx echo.Context, user *api.User) error {
  sess, _ := session.Get("memos_session", ctx)
  sess.Options = &sessions.Options{
    Path:    "/",
    MaxAge:  3600 * 24 * 30,
    HttpOnly: true,
    Secure : true ,
  }
  sess.Values[userIDContextKey] = user.ID
  err := sess.Save(ctx.Request(), ctx.Response())
  [...]
}
```

2
```
func setUserSession(ctx echo.Context, user *api.User) error {
  sess, _ := session.Get("memos_session", ctx)
  sess.Options = &sessions.Options{
    Path:    "/",
    MaxAge:  3600 * 24 * 30,
    HttpOnly: true,
  }
  sess.Values[userIDContextKey] = user.ID
  sess.Values[userAgentKey] = ctx.Request().UserAgent()
  err := sess.Save(ctx.Request(), ctx.Response())
  [...]
}
```

Fig. 10. A CWE-319 example of error localization in Go language. Code 1 is the repair code and Code 2 is generated by GPT-4o. The code within the brackets highlights the modifications, with additions shown on a green background and deletions on a red background.

not fully learn the specific data format we employed, resulting in multiple token-level differences in the output. Some of the fixes achieved semantic equivalence, which indicates a partial understanding of the task. In terms of semantic equivalence, GPT-4o performed relatively better in C/C++, with 23 out of 100 examples achieving this, while Go performed worse in this aspect. However, the overall accuracy of GPT-4o on Go was already higher, so the slight performance gap is understandable. (Accuracy: Go – 37.72%, C – 15.08%, C++ – 13.82%).

We now selected several representative examples to illustrate the reasons for the failure. Figure 10 exhibits a CWE-319 (i.e., Cleartext Transmission of Sensitive Information) example of localization error in the Go language. In Code 1, the added code ensures that session cookies are transmitted only over HTTPS connections, preventing the transmission of sensitive session information over insecure HTTP. This configuration effectively reduces the risk of session cookies being intercepted and prevents sensitive information from being exposed in unencrypted transmissions. However, in Code 2, GPT-4o failed to identify this vulnerability, making modifications in unrelated areas instead. Figure 11 shows a CWE-120 (i.e., Buffer Copy without Checking Size of Input) example of multiple chunks errors in C language. In Code 1, the changed code addresses this issue by adding an EOF check in the condition statements. This ensures that the pointer is valid and within the bounds of the buffer before accessing its value, preventing potential buffer overflows and fixing CWE-120. In contrast, in Code 2, GPT-4o failed to identify all the vulnerable statements, leaving the vulnerability unresolved. Figure 12 shows a CWE-74 (i.e., Improper

Neutralization of Special Elements in Output Used by a Downstream Component) example of logical error in C language. In Code 1, the vulnerability is potentially mitigated by the addition of the r_str_ansi_strip(s) function call, which removes ANSI control characters from the input string 's' and thereby reduces the risk of injection attacks exploiting special characters. However, in Code 2, GPT-4o failed to address this vulnerability effectively and instead removed all related code, which is not a valid solution. Based on our manual analysis, future research could investigate the use of multi-agent LLMs to tackle localization tasks in vulnerability repair, which could further enhance the performance of LLMs in multilingual vulnerability repair. Furthermore, another promising avenue involves developing a more comprehensive categorization of false positives and their real-world impact. Our current study treats all incorrect repairs uniformly under the Exact Match framework. However, a qualitative breakdown—distinguishing between partial fixes, semantically incorrect patches, superficial/no-op changes, and patches that introduce new flaws—would yield deeper insights. Such an analysis could reveal the characteristic failure modes of LLM-based approaches compared to traditional AVR techniques, providing a more nuanced understanding for practitioners.

5.4 The security of LLMs for vulnerability repair

Although LLMs have demonstrated remarkable capabilities in multilingual vulnerability repair, the use of LLM-generated patches poses significant security risks. First, over-reliance on LLM-generated patches without adequate human oversight may inadvertently introduce new vulnerabilities. Although these models can identify and address known issues, they may also produce patches that only superficially resolve the problem while overlooking deeper security implications. Fu et al. [24] analyzed code snippets generated by GitHub Copilot and found that developers face a high risk of introducing security weaknesses when using Copilot or other AI-based code generation tools, regardless of the programming language. Therefore, implementing appropriate security checks is essential. Second, the internal architectures, training data, and inference mechanisms of closed-source LLMs remain opaque. This lack of transparency raises concerns about the potential presence of backdoors and inherent model biases, which are difficult for users to detect or mitigate. In security-critical contexts, such opacity severely undermines trust, accountability, and verifiability. Moreover, the use of closed-source LLMs raises substantial concerns about data privacy and security, as users have limited control over how their inputs are processed and stored. Third, a recent study [72] has revealed that current LLMs often overlook important security considerations during both code generation and repair. As a result, the output code may contain latent vulnerabilities or subtle logic flaws that compromise system security. In summary, while LLMs hold great promise in automated vulnerability repair, their current limitations—especially in closed-source settings—necessitate caution. Without rigorous validation and human-in-the-loop verification, LLM-generated patches may not only fail to address the original issue but may also introduce new security risks.

5.5 The promise of LLMs on multilingual vulnerability repair

Results from the comparative analysis of RQ1 through RQ4 show that LLMs with appropriate strategies compete effectively with AVR techniques and PLM approaches in multilingual repair tasks. For instance, GPT-4o achieves a 28.71% EM rate, comparable to VulMaster's 28.94% EM. Specifically, by combining instruction-tuning with BM25-based few-shot prompting, LLMs can generate high-quality repairs for inputted vulnerable functions without needing side information (e.g., CWE details) or specialized pre-processing (e.g., special tokens or program analysis). Due to the language-agnostic capabilities of LLMs, they can also achieve competitive results with limited computational resources, even without instruction-tuning. For instance, GPT-4o with sole using few-shot prompting achieves a 26.89% EM, significantly outperforming many conventional AVR techniques and PLM approaches. Furthermore, GPT-4o, when equipped with instruction tuning and few-shot prompting, demonstrates strong generalization to unseen TypeScript vulnerabilities, whereas the leading AVR technique, VulMaster,

An CWE-120 Example of Multiple chunks errors

1

```
PJ_DEF(void) pj_scan_get_newline(pj_scanner *scanner) {
  if (!PJ_SCAN_IS_NEWLINE(*scanner->curptr)) {
  if (pj_scan_is_eof(scanner) || !PJ_SCAN_IS_NEWLINE(*scanner->curptr)) {
    pj_scan_syntax_err(scanner);
    return;
  }
  if (*scanner->curptr == '\r') {
    ++scanner->curptr;
  }
  if (*scanner->curptr == '\n') {
  if (!pj_scan_is_eof(scanner) && *scanner->curptr == '\n') {
    ++scanner->curptr;
  }
  ++scanner->line;
  scanner->start_line = scanner->curptr;
}
```

2

```
PJ_DEF(void) pj_scan_get_newline(pj_scanner *scanner){
  if (!PJ_SCAN_IS_NEWLINE(*scanner->curptr)) {
  if (pj_scan_is_eof(scanner) || !PJ_SCAN_IS_NEWLINE(*scanner->curptr)) {
    pj_scan_syntax_err(scanner);
    return;
  }
  if (*scanner->curptr == '\r') {
    ++scanner->curptr;
  }
  if (*scanner->curptr == '\n') {
    ++scanner->curptr;
  }
  ++scanner->line;
  scanner->start_line = scanner->curptr;
}
```

Fig. 11. A CWE-120 example of multiple chunks errors in C language. Code 1 is the repair code and Code 2 is generated by GPT-4o. The code within the brackets highlights the modifications, with additions shown on a green background and deletions on a red background.

exhibits only limited generalization. These findings show that LLMs have promising capabilities for multilingual vulnerability repair tasks and substantial untapped potential. We identify five future directions for enhancing LLM-based multilingual vulnerability repair.

First, future work could explore additional prompting strategies beyond those examined in this study, which focused solely on zero-shot prompting, few-shot prompting, instruction tuning, and CoT prompting. Advanced techniques such as Tree-of-Thought (ToT) prompting [81] and iterative refinement based on feedback [39] merit further investigation. Specifically, researchers could leverage static analysis security tools to evaluate the code repaired by AVR techniques and incorporate the resulting feedback to guide and enhance LLM outputs. Moreover, the current few-shot prompting approach uses an IR-based BM25 method for selecting examples,

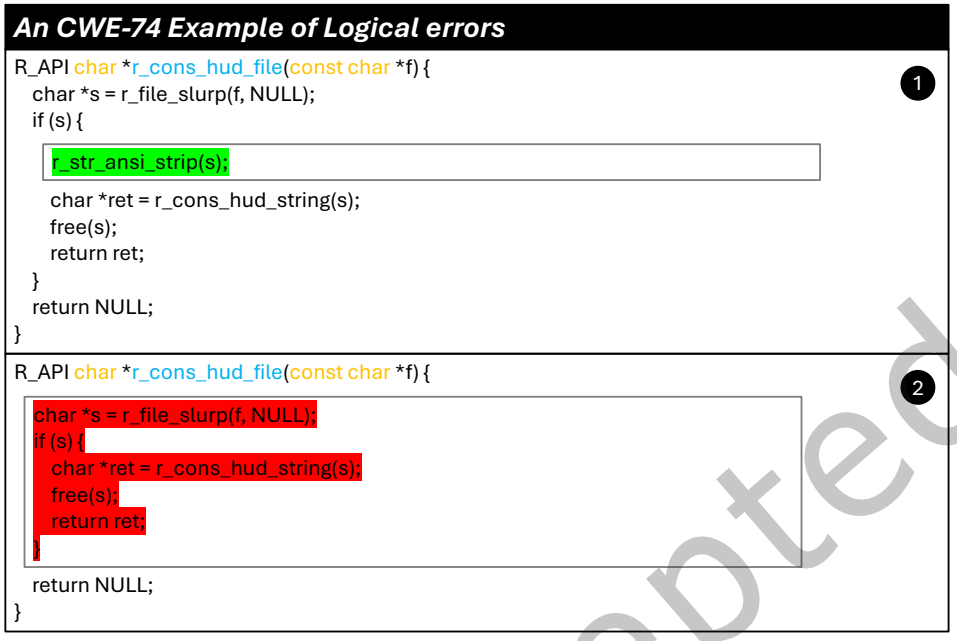


Fig. 12. A CWE-74 example of logical error in Go language. Code 1 is the repair code and Code 2 is generated by GPT-4o. The code within the brackets highlights the modifications, with additions shown on a green background and deletions on a red background.

which relies on syntactic similarity. Future work could explore alternative example selection methods to ensure a more diverse set of examples in few-shot prompts, thereby improving the model's ability to generalize to new inputs. Second, the scale of the dataset we use remains relatively small, which limits the performance of instruction-tuned LLMs to some extent. Future research could expand multilingual vulnerability benchmarks by leveraging the framework of multilingual vulnerability data collection [70]. Third, the most recent study [65] indicates that successful vulnerability repair in production requires that the patched code pass all existing tests, prevent recurrence of the vulnerability, and not introduce new security issues. However, generally existing available datasets are not executable for verification, making it difficult to verify these criteria. Future work should evaluate AVR techniques on dynamically verifiable datasets to better ensure the reliability and applicability of the results in real-world settings. Moreover, current LLMs for multilingual vulnerability repair only use a single modality (i.e., source code), which may limit the effectiveness of LLMs on some specific vulnerability type. For example, recent studies [7, 66] indicate that leveraging control-flow and data-flow graph is beneficial for detecting the memory-related vulnerability. Therefore, we should consider incorporating additional data modality like graph-structural data to improve the performance of LLMs on particular vulnerabilities. Last, recent studies [9, 32]have underscored the importance of robustness testing for LLMs in security-critical tasks. Chen et al. [9] demonstrated that real-world variations in natural language prompts can substantially degrade the performance of code LLMs. Similarly, Honarvar [32] emphasized that LLM-based code generation suffers from generalization gaps across semantically related tasks. These findings highlight the need for further investigation into the robustness of AVR techniques against minor code modifications or adversarial inputs. Given that LLMs are highly sensitive to input variations, where even slight differences in semantically equivalent programs can

lead to repair failures, comprehensive robustness testing is essential before deploying AVR techniques in practical settings.

6 THREATS TO VALIDITY

External Threats mainly lie in the used multilingual vulnerability dataset and the studied automated vulnerability repair approaches. Since this work relied on the REEF dataset, which encompasses seven popular programming languages, our findings may not generalize to other languages. For future work, we plan to expand our collection of vulnerabilities across a broader range of programming languages using the REEF framework. To mitigate the second threat, we conducted a systematic literature review, ensuring that our selected automated vulnerability repair approaches are state-of-the-art and representative. Regarding the LLM selection, we consider both advanced open-source and closed-source LLMs to ensure diversity.

Internal Threat arises from our implementation of the studied automated repair approaches. To mitigate this threat, we implemented all existing approaches using the replication packages provided in their respective papers and two authors carefully reviewed the source code. For the studied LLMs, we used publicly accessible models (i.e., DeepSeek-Coder, Code Llama, and Llama3) from Hugging Face or invoked their APIs (i.e., GPT-3.5-Turbo and GPT-4o) as per official instructions. Another potential threat to validity stems from the non-deterministic nature of LLMs, which may influence the experimental results and findings. To mitigate this issue and support future reproducibility, we have explicitly documented the configuration settings of all LLMs used in this study and publicly released the fully reproducible replication package [31].

Construct Threats primarily stem from the construction of vulnerable function pairs and metrics that evaluate the automated vulnerability approaches. We relied on the tool namely Tree-sitter to parse the commit data and collect function pairs, which means there is a possibility of incorrect function pair matching. To mitigate this threat, we performed a sanity check on a group of randomly selected samples to ensure the robustness of the tool. For our evaluation metrics, we used established measures in the vulnerability domain, including Exact Match, BLEU, and ROUGE scores.

7 CONCLUSION

This work presents a large-scale empirical study to examine the effectiveness of the existing software vulnerability repair approaches (learning-based approaches and pre-trained language models) and LLMs in the context of multilingual vulnerabilities across seven programming languages. Our key findings show that GPT-4o with instruction tuning and few-shot prompting is both the best-performing LLM and competitive with VulMaster, the leading AVR approach, in multilingual vulnerability repair. Furthermore, instruction-tuned GPT-4o with few-shot prompting outperforms VulMaster in repairing unique vulnerabilities and is more effective at addressing the most dangerous vulnerabilities. In addition, instruction-tuned GPT-4o with few-shot prompting demonstrates strong generalization capabilities on vulnerabilities in previously unseen language, outperforming existing approaches. Our manual analysis reveals that error localization is the main factor preventing instruction-tuned GPT-4o from successfully repairing vulnerabilities.

Meanwhile, our work opens up several promising future research directions, including: exploring advanced prompting strategies, expanding the scale of multilingual vulnerability datasets, evaluating AVR techniques on dynamically verifiable benchmarks, incorporating multimodal approaches to address specific vulnerability types across diverse programming languages, and performing comprehensive robustness testing prior to real-world deployment.

8 DATA AVAILABILITY

We released all the experimental data and source code on the project homepage for replication, future research, and practical use [31].

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFB4506300), National Natural Science Foundation of China (Grant Nos. 62322208, 12411530122), JSPS for the KAKENHI grants (JP21H04877, JP22K18630), Bilateral Program grant JPJSBP120239929, Japan Science and Technology Agency (JST) as part of Adopting Sustainable Partnerships for Innovative Research Ecosystem (ASPIRE), Grant Nos JPMJAP2415, and the Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship.

REFERENCES

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.
[2] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*. PMLR, 780–791.
[3] 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.
[4] Leyla Bilge and Tudor Dumitras. [n. d.]. An empirical study of zeroday attacks in the real world. *CCS'12* ([n. d.]), 16–18.
[5] Max Brunsfeld. 2024. tree-sitter/tree-sitter: v0.23.0. https://doi.org/10.5281/zenodo.13375512
[6] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. 2022. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 464–468.
[7] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th international conference on software engineering*. 1456–1468.
[8] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-end object detection with transformers. In *European conference on computer vision*. Springer, 213–229.
[9] Junkai Chen, Li Zhenhao, Hu Xing, and Xia Xin. 2024. Nlperturbator: Studying the robustness of code llms to natural language variations. *ACM Transactions on Software Engineering and Methodology* (2024).
[10] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.
[11] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. Seqtrans: automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering* 49, 2 (2022), 564–585.
[12] Hyung Won Chung, Le Hou, S. Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Wei Yu, Vincent Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. 2022. Scaling Instruction-Finetuned Language Models. *JMLR* (2022).
[13] CVE. 2024. https://www.cve.org/About/Overview.
[14] CWE. 2024. https://cwe.mitre.org/about/index.html.
[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
[16] Edgescan. 2024. 2024 Vulnerability Statistics Report. https://www.edgescan.com/stats-report/ Accessed: 2024-10-24.
[17] Mohamad Fakih, Rahul Dharmaji, Halima Bouzidi, Gustavo Quiros Araya, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. 2025. LLM4CVE: Enabling Iterative Automated Vulnerability Repair with Large Language Models. arXiv:2501.03446 [cs.SE] https://arxiv.org/abs/2501.03446
[18] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.
[19] 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).

[20] Nicole Forsgren, Bas Alberts, Kevin Backhouse, Grey Baker, Greg Cecarelli, Derek Jedamski, Scot Kelly, and Clair Sullivan. 2021. 2020 state of the octoverse: Securing the world's software. *arXiv preprint arXiv:2110.10246* (2021).
[21] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–29.
[22] 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.
[23] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le. 2023. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, Los Alamitos, CA, USA, 632–636. https://doi.org/10.1109/APSEC60848.2023.00085
[24] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2023. Security weaknesses of copilot generated code in github. *arXiv preprint arXiv:2310.02059* (2023).
[25] Deep Ganguli, Danny Hernandez, Liane Lovitt, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova Dassarma, Dawn Drain, Nelson Elhage, et al. 2022. Predictability and surprise in large generative models. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*. 1747–1764.
[26] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. 2023. What makes good in-context demonstrations for code intelligence tasks with llms?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 761–773.
[27] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International conference on mining software repositories*. 18–21.
[28] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[29] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
[30] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, and peter chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/68abef8ee1ac9b664a90b0bbaff4f770-Paper.pdf
[31] Homepage. 2024. https://github.com/salted-yu/AVRStudy.
[32] Shahin Honarvar. 2025. Evaluating Correct-Consistency and Robustness in Code-Generating LLMs. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 797–800.
[33] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2023).
[34] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).
[35] Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. 2024. Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[36] Gautier Izacard and Édouard Grave. 2021. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. 874–880.
[37] Tiantian Ji, Yue Wu, Chang Wang, Xi Zhang, and Zhongru Wang. 2018. The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques. In *2018 IEEE third international conference on data science in cyberspace (DSC)*. IEEE, 53–60.
[38] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.
[39] Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. 2024. A Case Study of LLM for Automated Vulnerability Repair: Assessing Impact of Reasoning and Patch Validation Feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*. 103–111.
[40] Wen Li, Li Li, and Haipeng Cai. 2022. On the vulnerability proneness of multilingual code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 847–859.
[41] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1035–1047.
[42] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[43] Evie McCrum-Gardner. 2008. Which is the correct statistical test to use? *British Journal of Oral and Maxillofacial Surgery* 46, 1 (2008), 38–41.
[44] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure coding practices in java: Challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*.
[45] MITRE Corporation. 2023. CWE-1000: Research Concepts. https://cwe.mitre.org/data/definitions/1000.html Accessed: 2025-07-29.
[46] David Mueller, Mark Dredze, and Nicholas Andrews. 2024. Multi-Task Transfer Matters During Instruction-Tuning. In *Annual Meeting of the Association for Computational Linguistics*.
[47] Manisha Mukherjee and Vincent J Hellendoorn. 2023. Stack over-flowing with results: The case for domain-specific pre-training over one-size-fits-all models. *CoRR* (2023).
[48] National Institute of Standards and Technology (NIST). 2024. Software Vulnerability - Glossary | CSRC. https://csrc.nist.gov/glossary/term/software_vulnerability. Accessed: 2024-10-19.
[49] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[50] Vadim Okun, Aurelien Delaitre, Paul E Black, et al. 2013. Report on the static analysis tool exposition (sate) iv. *NIST Special Publication* 500 (2013), 297.
[51] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt.
[52] OpenAI. 2024. https://openai.com/.
[53] OpenAI. 2024. GPT-4o: A Flagship Model by OpenAI. https://openai.com/index/gpt-4o-and-more-tools-to-chatgpt-free. Accessed: 2024-10-19.
[54] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.
[55] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
[56] 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)*. 2339–2356. https://doi.org/10.1109/SP46215.2023.10179324
[57] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2024. Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology* 175 (2024), 107523. https://doi.org/10.1016/j.infsof.2024.107523
[58] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[59] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.
[60] Sofia Reis and Rui Abreu. 2021. A ground-truth dataset of real security patches. *arXiv preprint arXiv:2110.09635* (2021).
[61] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
[62] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[63] Rico Sennrich. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).
[64] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 537–549.
[65] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *arXiv preprint arXiv:2502.07049* (2025).
[66] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming Jiang, and Nachiappan Nagappan. 2024. A systematic literature review on automated software vulnerability detection using machine learning. *Comput. Surveys* 57, 3 (2024), 1–36.
[67] Honglin Shu, Dong Wang, Antonio Mastropaolo, Gabriele Bavota, and Yasutaka Kamei. 2025. An Empirical Study on Language Models for Generating Log Statements in Test Code. *ACM Trans. Softw. Eng. Methodol.* (Aug. 2025). https://doi.org/10.1145/3759915 Just Accepted.
[68] The MITRE Corporation. 2024. CVE Metrics - Common Vulnerabilities and Exposures. https://www.cve.org/About/Metrics. Accessed: 2024-10-19.
[69] Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. 2024. Large Language Models for Equivalent Mutant Detection: How Far Are We? *arXiv preprint arXiv:2408.01760* (2024).
[70] Chaozheng Wang, Zongjie Li, Yun Pena, Shuzheng Gao, Sirong Chen, Shuai Wang, Cuiyun Gao, and Michael R Lyu. 2023. Reef: A framework for collecting real-world vulnerabilities and fixes. In *2023 38th IEEE/ACM International Conference on Automated Software*

Engineering (ASE). IEEE, 1952–1962.
[71] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).
[72] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2024. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395* (2024).
[73] Xin Wang, Xiao Liu, Pingyi Zhou, Qixia Liu, Jin Liu, Hao Wu, and Xiaohui Cui. 2022. Test-Driven Multi-Task Learning with Functionally Equivalent Code Transformation for Neural Code Generation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–6.
[74] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
[75] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).
[76] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
[77] WhiteSource. 2022. Mend bolt. https://www.mend.io/free-developer-tools/bolt
[78] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).
[79] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1282–1294.
[80] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.
[81] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* 36 (2023), 11809–11822.
[82] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1274–1286.
[83] Junji Yu, Honglin Shu, Michael Fu, Dong Wang, Chakkrit Tantithamthavorn, Yasutaka Kamei, and Junjie Chen. 2025. A Preliminary Study of Large Language Models for Multilingual Vulnerability Detection. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Clarion Hotel Trondheim, Trondheim, Norway) *(ISSTA Companion '25)*. Association for Computing Machinery, New York, NY, USA, 161–168. https://doi.org/10.1145/3713081.3731746
[84] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240* (2023).
[85] Haoxiang Zhang, Shaowei Wang, Heng Li, Tse-Hsun Chen, and Ahmed E Hassan. 2021. A study of c/c++ code weaknesses on stack overflow. *IEEE Transactions on Software Engineering* (2021).
[86] Lan Zhang, Qingtian Zou, Anoop Singhal, Xiaoyan Sun, and Peng Liu. 2024. Evaluating Large Language Models for Real-World Vulnerability Repair in C/C++ Code. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics*. 49–58.
[87] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing* (2023).
[88] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology* (2024).
[89] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[90] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. 2020. Deformable detr: Deformable transformers for end-to-end object detection. *arXiv preprint arXiv:2010.04159* (2020).