Unlocking LLM Repair Capabilities Through Cross-Language Translation and Multi-Agent Refinement

Wenqiang Luo
Department of Computer Science
City University of Hong Kong
Hong Kong, Hong Kong, Hong Kong
wenqialuo4-c@my.cityu.edu.hk

Jacky Wai Keung
Department of Computer Science
City University of Hong Kong
Hong Kong, Hong Kong, Hong Kong
Jacky.Keung@cityu.edu.hk

Boyang Yang
Beijing JudaoYouda Network Technology Co. Ltd.
Jisuan Institute of Technology
Beijing, Beijing, China
buaabarty@gmail.com

Jacques Klein
SnT
University of Luxembourg
Luxembourg, Luxembourg
jacques.klein@uni.lu

Tegawende F. Bissyande
SnT
University of Luxembourg
Luxembourg, Luxembourg
tegawende.bissyande@uni.lu

Haoye Tian*
Department of Computer Science
Aalto University
Espoo, Finland
haoye.tian@aalto.fi

Bach Le
School of Computing and Information System
University of Melbourne
Melbourne, Australia
bach.le@unimelb.edu.au

Abstract

Recent advances in leveraging LLMs for APR have demonstrated impressive capabilities in fixing software defects. However, current LLM-based approaches predominantly focus on mainstream programming languages like Java and Python, neglecting less prevalent but emerging languages such as Rust due to expensive training resources, limited datasets, and insufficient community support. This narrow focus creates a significant gap in repair capabilities across the programming language spectrum, where the full potential of LLMs for comprehensive multilingual program repair remains largely unexplored. To address this limitation, we introduce a novel cross-language program repair approach LANTERN that leverages LLMs' differential proficiency across languages through a multi-agent iterative repair paradigm. Our technique strategically translates defective code from languages where LLMs exhibit weaker repair capabilities to languages where they demonstrate stronger performance, without requiring additional training. A key innovation of our approach is an LLM-based decision-making system that dynamically selects optimal target languages based on bug characteristics and continuously incorporates feedback from previous repair attempts.

We evaluate our method on xCodeEval, a comprehensive multilingual benchmark comprising 5,068 bugs across 11 programming languages. Results demonstrate significant enhancement in repair effectiveness, particularly for underrepresented languages, with Rust showing a 22.09% improvement in *Pass*@10 metrics. Our research provides the first empirical evidence that cross-language translation significantly expands the repair capabilities of LLMs and effectively bridges the performance gap between programming languages with different levels of popularity, opening new avenues for truly language-agnostic automated program repair.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

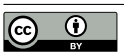
Automated Program Repair, Large Language Model, Cross-Language Translation, Multi-Agent Refinement

ACM Reference Format:
Wenqiang Luo, Jacky Wai Keung, Boyang Yang, Jacques Klein, Tegawende F. Bissyande, Haoye Tian, and Bach Le. 2026. Unlocking LLM Repair Capabilities Through Cross-Language Translation and Multi-Agent Refinement. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773227

1 Introduction

Software bugs remain an unavoidable challenge in modern software development where automated program repair (APR) [16, 31, 34, 62] has emerged as a promising solution by automatically identifying and generating patches for software defects. Driven by the rapid revolution in Large Language Models (LLMs) with unprecedented capabilities in generating complex code fixes [13, 14, 42, 50], LLM-based APR approaches have demonstrated remarkable success in

*Corresponding author.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773227

addressing diverse software defects, from semantic bugs [45, 47] and security vulnerabilities [15, 67] to syntax errors [4, 12] and hardware security issues [2, 57].

By benefiting from training or fine-tuning [18, 24, 32, 58] on diverse multilingual codebases, LLM-based repair approaches have achieved promising performance. However, these models exhibit substantial disparities in fixing bugs across languages, with significantly stronger repair capabilities in prevalent languages [1, 36] such as Java and Python [25, 51, 58] while struggling with others like Rust and Kotlin.

Notably, on the multilingual benchmark xCodeEval [27], the state-of-the-art LLM initially achieves *Pass*@10 scores of 89.02% and 89.93% for Python and PHP, respectively. However, for less common languages such as Rust, the *Pass*@10 score drops dramatically to only 65.58%, revealing a significant performance gap of over 24% points. This disparity primarily stems from the imbalanced distribution of training data, where mainstream languages such as Python dominate open source repositories with millions of contributors compared to relatively newer languages like Rust (according to the statistics of GitHub project contributors up to 2024 [37]). Furthermore, the study of Zhang et al. [63] demonstrates that the imbalance in programming languages also persists in current academia, where the majority of LLM-based APR research focuses on languages such as Java, Python, and C, while studies concerning less prevalent ones such as Rust, Go and Ruby constitute only 13% of the total research as of 2024. The scarcity of mature high-quality datasets, the high cost of training or fine-tuning, relatively less support in the APR community, and diverse language characteristics [60] further amplify these challenges, limiting the practicality and scalability of existing approaches.

Programming problems often have equivalent implementations across different languages that serve the same underlying intentions and functions [41, 44], which suggests a natural opportunity: if a bug proves difficult to fix in one language, it might be beneficial to draw from the repair expertise available in another language. Intuitively, in other words, an LLM that exhibits superior repair capabilities in one language could be leveraged to assist with resolving issues in another. In addition, as the repair process unfolds new insights and historical experiences [48], these can be fed back into the repair cycle to guide future refinement. Inspired by these intuitions, we propose to leverage cross-language knowledge by translating the buggy code to other languages with iterative refinement using historical feedback.

LLMs trained in one language can sometimes even outperform those trained in the target language itself [3], suggesting valuable transfer effects across language boundaries. With rapid progress in code translation [41, 68], the study of Pan et al. [39] demonstrates that LLMs can translate code better than non-LLM translation approaches for specific languages, demonstrating the potential of LLMs in code translation. Furthermore, benefited from advances in software engineering agents [30, 55], the integration of agent-driven [7, 17, 52, 65] and multi-agent [28, 61] approaches can enhance program repair with historical feedback while autonomously navigating complex repair workflows throughout the iterative repair process. Therefore, we introduce the key hypothesis:

> **"When the LLM fails to repair buggy code in a given programming language, translating the code into another language where the model demonstrates stronger repair capabilities may facilitate successful bug fixing."**

This work. In this paper, we present a novel program repair approach by leveraging cross-**LAN**guage **T**ranslation and multi-ag**E**nt **R**efi**N**ement (LANTERN) that strategically translates bugs to other programming languages with a multi-agent iterative repair paradigm. Instead of directly fixing the faulty code within its original language, LANTERN translates the code into a target programming language, which is selected based on the decision-making capabilities of the LLM that reasons about the bug characteristics and historical feedback. Once the bug is translated, repair is attempted in this alternative language, and the successfully repaired code is subsequently translated back to the original language. Repairs that fail to pass the test suites are not discarded, but instead scheduled for subsequent iterations of refinement, enabling a progressive improvement cycle. We conduct comprehensive experiments to evaluate whether and how LANTERN works. The evaluation results demonstrate that our approach successfully addresses most bugs that resist direct LLM-based repair attempts. Additional ablation studies confirm that improved repair performance stems from cross-language translation rather than simply repeated fix iterations, demonstrating the unique repair opportunities provided by cross-language perspectives.

Contributions. This paper makes the following contributions:

- **[Hypothesis]** We propose a novel hypothesis on enhancing cross-language repair capabilities through language translation.
- **[Technique]** Based on the hypothesis, we propose a program repair approach termed LANTERN that analyzes and translates buggy code to other programming languages in a multi-agent paradigm to enhance cross-language program repair through iterative multi-agent refinement.
- **[Evaluation]** We perform an extensive evaluation on xCodeEval, which is a state-of-the-art multilingual benchmark that consists of 5068 bugs across 11 programming languages. Evaluation results reveal that our approach can significantly enhance program repair across all languages. Notably, Rust achieves an improvement of 22.09% on *Pass*@10.
- **[Analysis]** We conduct an ablation study to validate the impact of the translator and analyzer, both of which contribute to performance improvement.

Availability. Artifacts of this work including code, dataset, and implementation details are publicly available at https://github.com/stringing/LANTERN.

2 Background and Related Work

2.1 Code Translation

Code translation refers to the process of transforming source code from one programming language to another, mainly employed for code adaptation scenarios or migrating legacy codebases written in deprecated languages to more modern alternatives [68]. Traditional rule-based translation approaches are costly, requiring substantial manual effort from programmers with specialized expertise in both

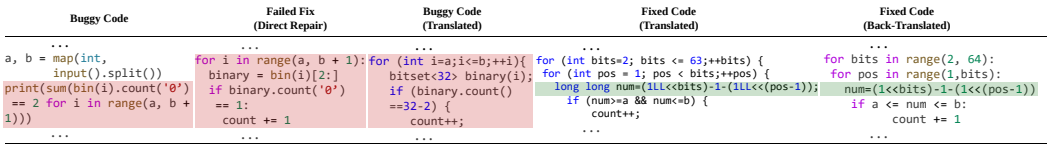


Figure 1: A motivating example of fixing a Python bug by translating it to C++.

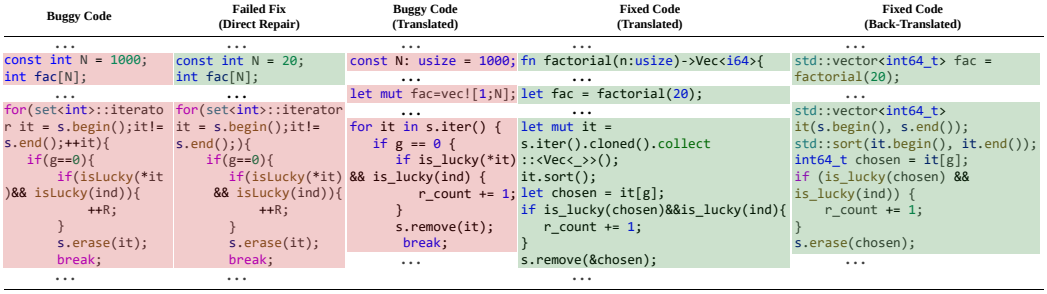


Figure 2: A motivating example of fixing a C++ bug by translating it to Rust.

the source and target languages [40]. Existing translation tools typically specialize in specific language pairs (e.g., Java to C# [9]) and require significant domain-specific knowledge to develop and maintain [6]. Meanwhile, unsupervised neural approaches, while promising, necessitate high-quality data for each language pair and computational resources for model training, making them also resource-intensive [41].

LLMs have shown promising capability in translating code between programming languages [39]. However, recent research highlights a significant gap between translating isolated code snippets and migrating entire software repositories, which involves complex dependencies, build configurations, and project-specific architectures [38, 46, 53, 64]. To tackle this, the field has shifted towards repository-level translation, developing new benchmarks and neuro-symbolic methods that combine program analysis with LLMs to manage complexity [20, 59]. These studies show that while fully automated, correct translation of large projects remains a challenge, the generated code can significantly reduce manual effort. Since program repair is the main objective of our paper rather than translation, we adopt LLM-based code translation for our approach to explore whether APR can be enhanced with code translation.

2.2 LLM-Based Program Repair Agent

LLM-based agents employ LLMs as the cognitive core to perceive and act based on environmental feedback, working toward specific goals through four essential components: Planning, Memory, Perception, and Action [30]. The planning component generates plans with reasoning strategies tailored to the task. The memory component maintains a record of historical data, while the perception component processes environmental feedback to facilitate more effective planning. The action component executes concrete steps based on decisions made by the planning component. Building upon this, existing program repair agents such as ChatRepair [52], AutoCodeRover [65], and RepairAgent [7] typically follow a pipeline consisting of four steps in an iterative paradigm: (1) the agent first generates potential fixes for the buggy code, (2) these fixes undergo patch validation through compilation, execution, and automated testing autonomously, (3) repair feedback from validation step is carefully analyzed to inform the next iteration of fixing, and (4) the process repeats until a solution passes all validation criteria or reaches a predetermined iteration limit. Moreover, multi-agent architectures further extend this paradigm by decomposing complex tasks into specialized modules. FixAgent [28] leverages multiple agents dedicated to autonomous bug fixing and bug localization. AgentCoder [19] integrates specialized agents responsible for code generation, test generation, and code execution. Similarly, ACFix [61] employs a configuration comprising a generator for proposing fixes and a validator tasked with ensuring their correctness. In our study, we leverage multiple LLM-based agents for program repair, code translation, and decision-making.

3 Illustrative Example

We illustrate how code translation can assist with repairing bugs that the LLM failed to fix with two real examples in our study. Figure 1 and Figure 2 present two examples demonstrating the efficiency of program repair with code translation. The examples show (1) the original buggy code, (2) the failed fix from direct repair, (3) the translated buggy code in a new programming language, (4) the fixed code in the target language, and (5) the back-translated fixed code. This detailed breakdown shows how the buggy code is first translated, then repaired in the target language, and finally translated back to its original language.

Figure 1 demonstrates a Python bug causing a time limit error due to inefficient brute-force iteration. Direct repair attempts only partially address this by removing the prefix issue but retain the performance bottleneck. When translated to C++, the algorithm was redesigned to iterate over bit positions rather than the entire range, dramatically reducing complexity. The optimized solution is then successfully back-translated to Python. Technically, **C++'s built-in facilities such as bitset and its emphasis on low-level efficiency** not only expose latent inefficiencies in the original language but also foster an algorithmic design that is both conceptually elegant and computationally superior.

Figure 2 shows a C++ memory limit error. While direct repair failed to address the unsafe iterator usage, translating to Rust exposed the flaw immediately due to Rust's strict ownership rules

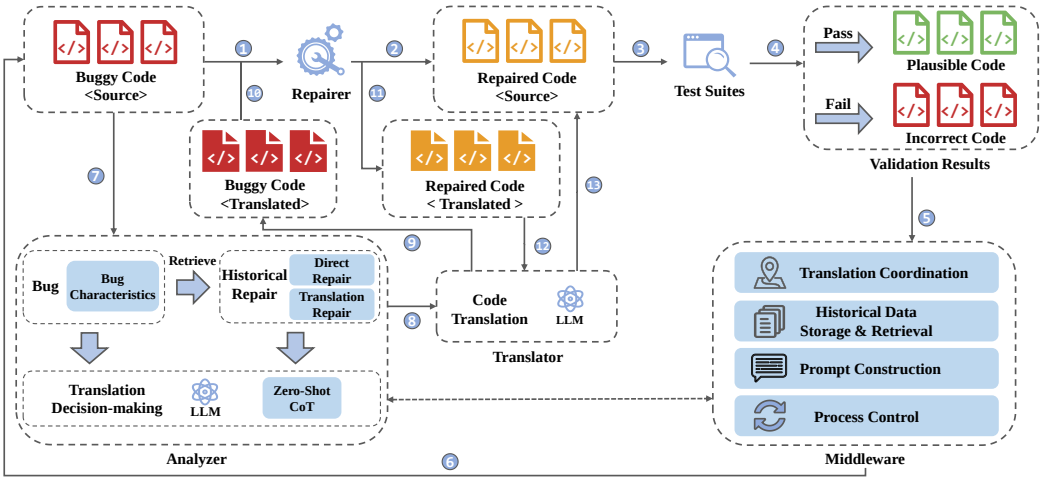


Figure 3: Overview of LANTERN.

(borrow checker). The Rust compiler's constraints forced a logic correction, which was preserved upon back-translation to C++, resolving the original memory limit issue. While the algorithm remains similar, **Rust's language-specific features regarding memory safety** help identify and fix memory and logical flaws that are less apparent in C++.

4 Approach

4.1 Overview

Figure 3 shows an overview of LANTERN. The workflow is fully automated and begins with buggy code in its source programming language as input (step 1) to the repairer, where we employ LLM to fix bugs. The repairer generates the fixed code (step 2), which then undergoes validation against corresponding test suites of the bugs (step 3). The plausible code, the incorrect code, and their corresponding detailed evaluation results (step 4) are then forwarded to the middleware for further processing (step 5). The middleware identifies the unfixed bugs for subsequent translation-based repair (step 6) and directs them to the analyzer (step 7). In addition to the bug characteristics, the analyzer retrieves historical records of fixing similar bugs through the middleware as historical feedback to inform its autonomous decision-making regarding the optimal target language for translation. Our approach constructs context-aware prompts for the reasoning process by dynamically integrating bug-specific details, historical fix insights from previous repair attempts. Once the target language is decided after a comprehensive analysis by the LLM, the unfixed bugs are translated to their corresponding target languages (steps 8-9), and subsequent repair attempts are made for the translated bugs (steps 10-11). The repaired translated code is then returned to the translator (step 12), where it is back-translated to its source programming language (step 13), enabling the next round of validation and further refinement through subsequent iterations. To ensure an efficient exploration of attempts on language diversity, each target language is employed only once per bug within a single iteration in our study.

4.2 Middleware

The middleware serves as the central coordination that orchestrates the key events of the entire workflow. As presented in Figure 3, the middleware comprises four primary components as follows:

Translation Coordination. This component receives the evaluation outcomes, identifies the failed fixes, and schedules the corresponding unfixed bugs for subsequent translation-based repair.

Historical Data Storage & Retrieval. Responsible for collecting and providing historical repair feedback for future reference. This component maintains a local vector database that encodes all evaluated bugs. The database is periodically updated after each iteration of repair. Historical repair feedback is sourced from two stages: (1) the initial direct repair attempts before translation-based repair, and (2) translation-based repair. While initial direct repair feedback is recorded at the beginning, translation-based repair feedback is updated in each iteration. Specifically, the evaluation results (e.g., repair performance, successful target languages where bugs are correctly fixed, etc.) and the bug characteristics (e.g., bug language, problem difficulty, execution outcome, etc.) from both sources of bugs are encoded into the vector database to facilitate effective bug retrieval.

Prompt Construction. This component dynamically constructs prompts for the LLM to support core actions in the workflow, including program repair, translation decision-making, code translation, and code back-translation according to specific bugs. In particular, it constructs prompts based on a designed prompt structure by incorporating detailed information such as bug characteristics along with auxiliary context like historical feedback from previous repair attempts, equipping the LLM with the necessary context to generate accurate responses for all tasks.

Process Control. This component monitors the overall pipeline and terminates the workflow when predefined conditions are met (e.g., reaching a maximum iteration limit). In our study, the maximum iteration limit is designed based on the number of supported languages in the benchmark, allowing the system to explore the full range of potential translation paths, which in our experiments ranges from 1 to 11 iterations corresponding to the language scope.

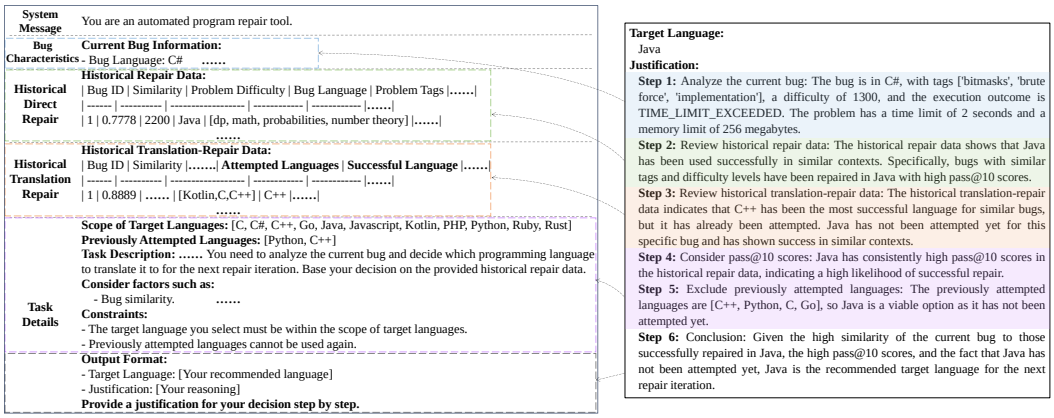


Figure 4: Prompt structure of translation decision-making (left) and an example of LLM response (right).

Subsequently, any code that has been repaired in the target language then undergoes back-translation to its original programming language, thereby facilitating further refinements in iterative repair cycles.

4.3 Analyzer & Translator

As the most important components of the pipeline, the analyzer and translator manage the translation process of each bug including translation decision-making, translation, and back-translation. The LLM-based analyzer reasons about the environmental feedback and makes decisions autonomously. The translator is also built on an LLM foundation to enable translation across diverse languages. Once received, the bug is encoded as a bug query to retrieve similar bugs through the middleware, which returns the top-k historical repair based on cosine similarity between the query bug and those from initial direct repair and previous translation-based repair. Consequently, the aggregated bug characteristics and the retrieved historical repair of similar bugs are provided as historical feedback for the analyzer. Specifically, the two sets of historical feedback serves a different purpose, respectively, where (1) the initial direct repair experience is provided for the analyzer to reason about which programming languages of bugs with similar characteristics the repairer can effectively resolve, and (2) the historical translation-based repair helps the LLM reveal optimal target languages previously employed in which similar bugs are successfully repaired. Once the optimal target language is determined, the bug is translated accordingly.

4.4 Prompt Design

Our program repair prompt integrates a concise problem description, error type, input/output specifications, the buggy code, and a repair instruction. Similarly, our code translation prompt specifies the source and target languages, the corresponding code, and the translation instruction, with back-translation using the reversed source and target languages.

We also design the prompt dedicated to translation decision-making as illustrated in Figure 4 in a zero-shot chain of thought (CoT) [23] paradigm, which systematically integrates bug characteristics, historical repair feedback, and task constraints into a structured reasoning framework. Through this step-by-step process, the model autonomously identifies optimal target languages that demonstrate superior capabilities in fixing similar bugs.

5 Evaluation

5.1 Research Questions

We evaluate our approach on the following research questions:

RQ1 (Overall Effectiveness):

- **RQ1.1 (Repair Performance)**: How effective is LANTERN in enhancing program repair capabilities across different programming languages compared to direct repair?
- **RQ1.2 (Language Selection Efficiency)**: How efficiently does LLM-based reasoning in LANTERN identify optimal target languages compared to other translation strategies?
- **RQ1.3 (Approach Comparison):** How does the repair performance of LANTERN compare against other state-of-the-art (SOTA) program repair approaches?

RQ2 (Translation Analysis):

- **RQ2.1 (Translation Outcomes)**: What underlying patterns behind the translation outcomes facilitate more efficient target language selection and enhance program repair?
- **RQ2.2 (Translation Consistency)**: To what extent do the translation and back-translation process preserve semantic consistency between source and target languages?

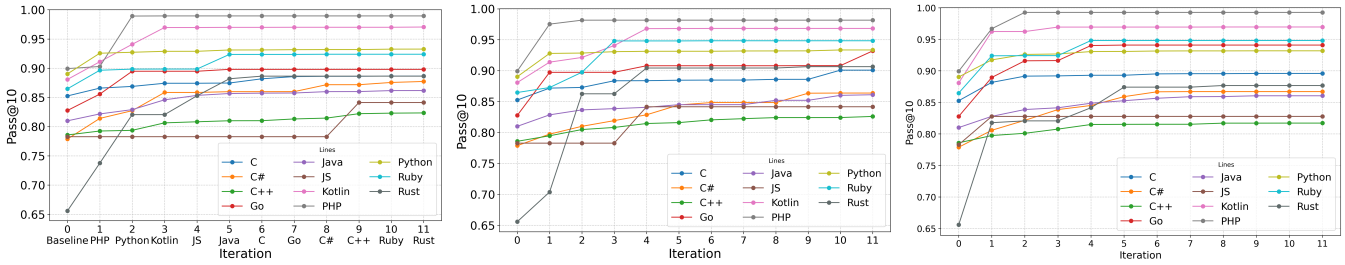
RQ3 (Ablation Study): How do cross-language translation and historical feedback contribute to LANTERN's repair performance and LLM-based reasoning efficiency in target language selection?

RQ4 (Generalizability):

- **RQ4.1 (Real-World Generalizability):** How effectively does LANTERN generalize to real-world APR benchmarks?
- **RQ4.2 (Model Generalizability):** Can LANTERN generalize to other LLMs and to what extent does LANTERN's effectiveness generalize across different LLMs?

5.2 Experiment Setup

5.2.1 Model. For primary experiments from RQ1 to RQ3, we use the open-source state-of-the-art DeepSeek-V3 [29] model as the



(a) Greedy strategy (target languages on x-axis). **(b) Random strategy.** **(c) Reasoning strategy.**

Figure 5: Comparison of different strategies on Pass@10 across iterations (iteration 0 being the baseline of initial direct repair).

LLM in the analyzer and translator, for both decision-making and code translation. We also leverage the same LLM for the repairer to align the language scope with the translator instead of adopting various APR approaches specialized for different languages of bugs. We choose DeepSeek-V3 because it is arguably among the best for coding tasks in the literature.

To validate the generalizability of our approach, we include two additional prominent LLMs in RQ4. We selected one closed-source model, Claude 3.5 Sonnet [5] and another powerful open-source model, Qwen2.5-72B-Instruct [43, 54], as a SOTA open-source alternative. This selection allows us to assess whether our approach's benefits are tied to a specific model architecture or are more broadly applicable across different foundational models.

5.2.2 Benchmark. We employ three distinct and challenging benchmarks. To assess our approach from RQ1 to RQ3, we use the xCodeEval [27] benchmark. Specifically, we use the Compact Set, which is a subset dedicated for academic research, containing 5,068 bugs across 11 languages (Table 1). It is sourced from Codeforces [11] and covers difficulty ratings from 800 to 3,500. Unlike other alternatives such as HumanEval [8], xCodeEval averages 50 tests per problem. In addition, xCodeEval's execution engine, ExecEval, provides dedicated runtime environments for all necessary compilers and interpreters, categorizing bug executions into six outcomes: compilation error, runtime error, memory limit exceeded, time limit exceeded, wrong answer, and passed.

For RQ4, we assess generalizability using Defects4J [26] and SWE-Bench [22]. We utilize Defects4J v1.2 (391 bugs) and v2.0 (438 bugs) for Java projects. For repository-level Python repair, we use SWE-Bench Lite, comprising 300 real-world GitHub issues. This subset is notably difficult, with relatively low SOTA resolve rates, making it ideal for evaluating robustness.

Table 1: Dataset statistics of xCodeEval for each programming language.

C	C#	C++	Go	Java	JS	Kotlin	PHP	Python	Ruby	Rust	Total
733	739	641	294	716	183	313	191	710	343	205	5068

5.2.3 Metrics. We employ evaluation metrics from multiple dimensions to rigorously assess the results in both program repair and decision-making.

Pass@k measures the likelihood that at least one of the top k generated patches fixes a bug [8, 66]. Since developers are generally willing to review a maximum of approximately 10 patches [35], we use *Pass*@10 under $n = 20$.

We also evaluate the reasoning ability of our approach, where the selected optimal target languages form a ranked list of languages over iterative attempts until the bug is fixed. Similar to recommendation systems that evaluate the rankings of **relevant** items, we assess our ranking quality based on **valid** iterations (i.e., where the selected languages result in successful fixes). We employ the ranking metrics [21] as follows:

Precision@k quantifies the proportion of valid iterations in the top k iterations.

Mean Average Precision at k (MAP@k) evaluates the overall quality of the rankings of target languages.

Normalized discounted cumulative gain at k (NDCG@k) measures the ranking quality by giving higher weights to valid iterations that appear earlier.

Recall@k measures the proportion of valid iterations among the top k iterations in the number of all relevant iterations.

F1@k is the harmonic mean of *Precision*@k and *Recall*@k, providing a balanced view between the two metrics.

5.3 RQ1: Overall Effectiveness

5.3.1 Experimental Design. To evaluate the effectiveness of our approach, we establish a maximum of 11 iterative cycles since there are 11 available programming languages in the benchmark, ensuring that all different target languages are attempted for each bug. In contrast to picking any language for each bug at each iteration, our approach leverages an LLM-based decision-making strategy that identifies the optimal target language by reasoning about the bug characteristics and historical feedback from previous repair attempts. Considering that all fixable bugs can always encounter a successful target language by the end of all 11 iterations, we compare with two more strategies as follows:

- **Greedy Strategy:** In this strategy, target languages are prioritized based on the historical performance of the LLM in the initial repair stage. For example, if the LLM achieved the best *Pass*@10 score on PHP initially, then PHP is selected as the target language in the first iteration for all bugs.
- **Random Strategy:** A target language is randomly selected for each bug in each iteration, serving as another baseline for evaluating the efficiency of the LLM-based decision.

- **Reasoning Strategy:** The LLM-based decision-making strategy that reasons about the optimal target language based on bug characteristics and historical feedback.

We compare the *Pass*@10 scores across iterations for all strategies to evaluate the repair performance in RQ1.1. We evaluate the quality of target language selection in RQ1.2 using ranking metrics that assess each strategy's efficiency in identifying valid repairs, which is critical since resolving more bugs in earlier iterations substantially reduces subsequent computational costs.

In RQ1.3, we compare LANTERN against SOTA approaches using DeepSeek-V3 on xCodeEval. Baselines include direct repair (original xCodeEval prompting), ChatRepair (iterative conversational refinement), Self-Planning (high-level algorithmic planning before coding), and Self-Collaboration (virtual development team with different agent roles for analysis, fixing, and validation).

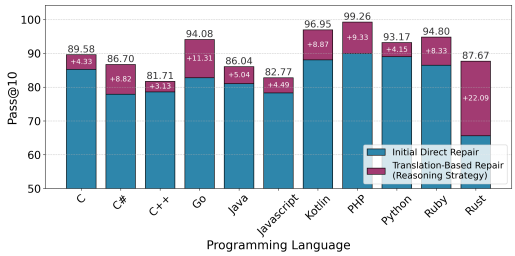


Figure 6: Pass@10 improvements of the reasoning strategy across programming languages.

5.3.2 Experimental Results for RQ1.1 (Repair Performance). Figure 5 shows the *Pass*@10 scores of the three strategies across iterations for bugs from 11 programming languages. We first observe in Figure 5a that under the greedy strategy, some languages require several iterations to yield fixes. For example, JavaScript (JS) bugs are only repaired in the ninth iteration after translation to C++, and PHP bugs cannot be fixed until the second iteration. In contrast, the random strategy produces a uniform distribution of target languages that leads to earlier fixes, where most PHP bugs are fixed immediately, with Go and Rust bugs achieving *Pass*@10 scores of 0.90 and 0.86 in the first and second iterations, respectively. However, JS bugs still cannot be fixed until the fourth iteration. Notably, the reasoning strategy as illustrated in Figure 5c fixes most bugs in the first iteration, with JS bugs resolved immediately and Kotlin and Ruby reaching *Pass*@10 scores of 0.96 and 0.92 in the first iteration, indicating the superior ability of LLM-based decision-making to select optimal target languages early on.

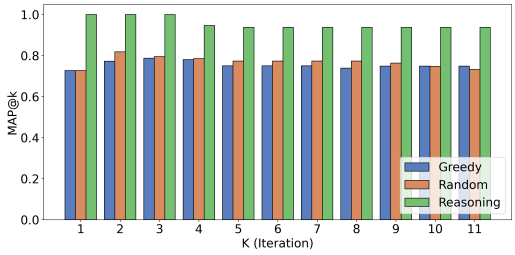


Figure 7: MAP@k of different strategies across iterations.

On the other hand, Figure 6 presents the final *Pass*@10 improvements of the reasoning strategy on all languages. It is worth noting that Rust bugs benefit the most from translation-based repair, which achieves an increment of 22.09%. Following are the Go, PHP, Kotlin, C#, and Ruby bugs that undergo improvements of 11.31%, 9.33%, 8.87%, 8.82%, and 8.33%. Our MWW tests [33, 49] further confirm the improvements with a p-value of 0.0126 (< 0.05) and a Cliff's Delta [10, 56] of 0.64 (large effect).

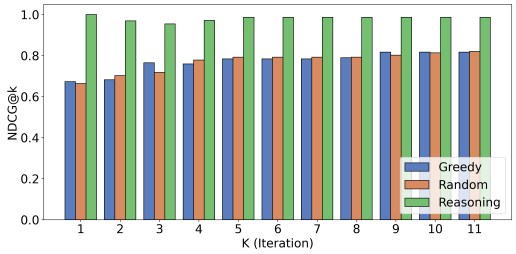


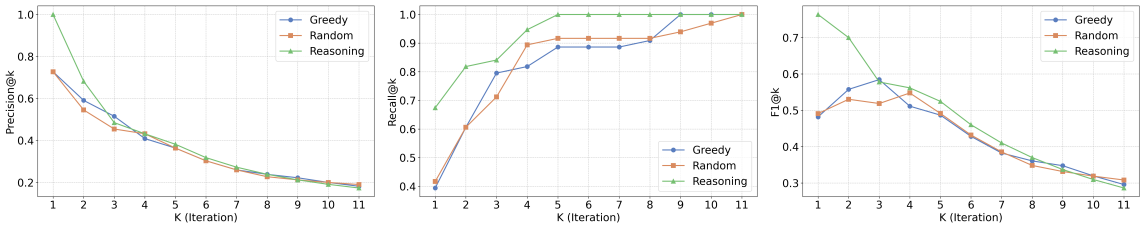
Figure 8: NDCG@k of different strategies across iterations.

5.3.3 Experimental Results for RQ1.2 (Language Selection Efficiency). Figure 7 shows that the reasoning strategy maintains an *MAP@k* of 1.00 for the first three iterations and converges to 0.938 by the fifth iteration, reflecting its consistent ability to identify optimal languages. Figure 8 exhibits that the reasoning strategy starts with a *NDCG@k* of 1.00 at the first iteration and consistently performs better than the other strategies at each iteration, highlighting its early advantages in the ranking.

According to Figure 9a, the reasoning strategy remains higher *Precision@k* until all strategies converge to the same level at the third iteration, demonstrating its early identification of valid fixes. Figure 9b shows that it achieves full recall with a *Recall@k* of 1.00 by the fifth iteration, ahead of greedy and random strategies, which converge only at the ninth and final iterations, respectively. Similarly, the *F1@k* scores in Figure 9c highlight its early advantage, aligning with our previous *Pass*@10 observations.

5.3.4 Experimental Results for RQ1.3 (Approach Comparison). Figure 10 shows LANTERN achieves the highest *Pass*@10 scores across xCodeEval, leading in 9 out of 11 languages. ChatRepair serves as a strong iterative baseline, demonstrating considerable gains from refinement within the original language (e.g., +18.31% for Rust). However, LANTERN consistently surpasses ChatRepair with improvements of 5.46%, 4.91%, and 6.76% on Go, Kotlin, and PHP respectively, suggesting that repetitive refinement has performance ceilings when confined to a single language. This gap stems from key architectural differences: ChatRepair iteratively refines solutions within one conversation, risking context accumulation and potential loss of critical early information, while LANTERN treats each repair as an independent session, ensuring context control and achieving one-shot solutions.

Self-Planning shows promise in some languages due to high-level planning abilities, achieving notable gains in JavaScript (+14.55%), but modest improvements elsewhere (e.g., +0.95% in C++) and significantly underperforming LANTERN. This indicates that generating high-level plans is insufficient when the LLM's underlying implementation ability in specific languages is weak.



(a) Precision@k. (b) Recall@k. (c) F1@k.

Figure 9: Comparison of different strategies on Precision@k, Recall@k, and F1@k across iterations.

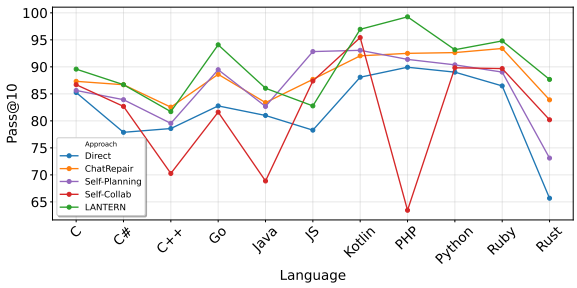
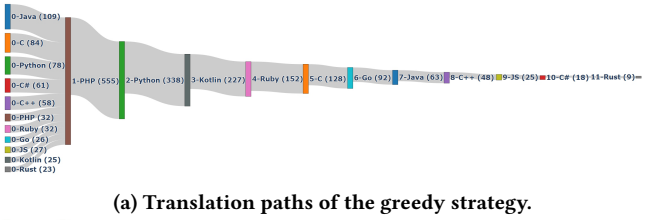
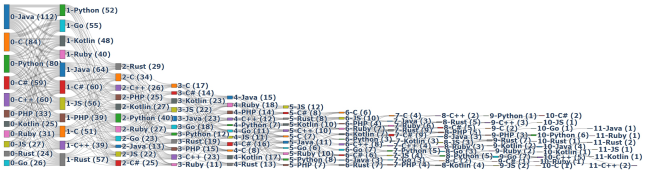


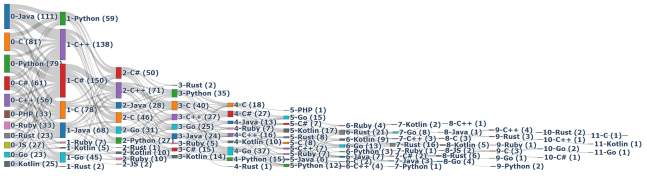
Figure 10: Pass@10 Performance of Different Approaches on xCodeEval



(a) Translation paths of the greedy strategy.



(b) Translation paths of the random strategy.



(c) Translation paths of the reasoning strategy.

Figure 11: Sankey diagrams of translation paths for successfully fixed bugs.

Self-Collaboration achieves large increments of 9.10% and 14.62% on JavaScript and Rust but causes severe performance degradation in several languages: C++ (-8.31%), Java (-12.13%), and most notably PHP (-26.48%). This suggests overly complex, simulated multi-role interactions can be inefficient, with the system potentially hallucinating incorrect analyses or validations that further mislead the model.

> ***[RQ-1] Findings:*** *(1) Simply defaulting to the language with the model's historically best performance does not guarantee the most effective repairs. Instead, a more tailored approach that analyzes specific bugs to select the optimal target language leads to more robust repair performance. (2) LANTERN can significantly enhance program repair, particularly for less common languages like Rust, achieving a maximal improvement of 22.09% in Pass@10. (3) The LLM reasoning capability can significantly enhance translation decision-making, with an average MAP@k of 0.957 and NDCG@k of 0.954, enabling efficient target language selection. (4) Changing the linguistic context of a bug is a more powerful enhancement technique than simply iterating or reasoning more deeply within the original language, especially for complex bugs or in languages where the LLM's native proficiency is limited.*

5.4 RQ2: Translation Analysis

5.4.1 Experimental Design. In RQ2.1, we investigate the translation outcomes to reveal the underlying differences between strategies. We compare the translation paths of the reasoning strategy against the greedy and random strategies, and analyze bugs fixed via cross-language translation versus direct repair. In particular, we assess the code difficulty of bugs fixed by LANTERN to determine if translation better addresses complex buggy code. Additionally, due to the partial semantic loss potentially caused by translation [39, 40], we validate semantic consistency in RQ2.2 for both bug translation and code back-translation by comparing test outcomes before and after translation.

5.4.2 Experimental Results for RQ2.1 (Translation Outcomes). Figure 11 shows the translation paths across iterations, with each node in the Sankey diagrams labeled as "*iteration number - bug language (number of bugs)*", where iteration 0 is the initial phase. As illustrated in Figure 11a, all bugs are translated to a single language per iteration by the greedy strategy, while the random strategy exhibited in Figure 11b selects nearly equal target languages each time. In contrast, Figure 11c demonstrates that the reasoning strategy gravitates towards specific languages. For example, the top-5 target languages in the first iteration, C#, C++, C, Java, and Python account for 150, 138, 78, 68, and 59 bugs, respectively. Similar outcomes can be observed particularly in the first three iterations after which the selection begins to shift toward the remaining languages since most languages have already been attempted. Our evaluation

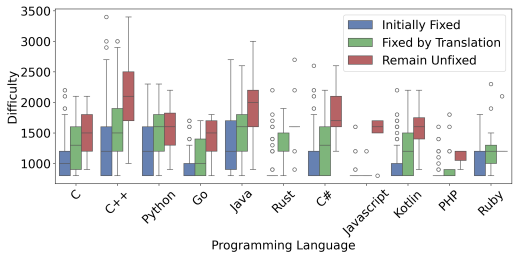


Figure 12: Difficulty distribution of the bugs fixed by initial repair, translation-based repair, and unfixed bugs.

demonstrates that the reasoning strategy achieves the shortest average translation path length (2.52), outperforming both random (2.69) and greedy (2.98) strategies, thereby minimizing resource consumption throughout the iterative process as unfixed bugs must propagate through all subsequent iterations.

Figure 12 exhibits the difficulty distribution of the bugs initially fixed by direct repair, fixed via translation, and those remain unfixed. We notice that the translation-based repair can fix bugs with 500 higher median difficulty for C#, while for Java, Kotlin, Ruby, and Rust, it enables successful repair of a group of bugs with median difficulty increased by 400. Our statistical analysis reveals a significant difference in difficulty between bugs fixed via translation and those initially fixed with a p-value of $5.25E-48$ and Cliff's delta of 0.37 (medium effect).

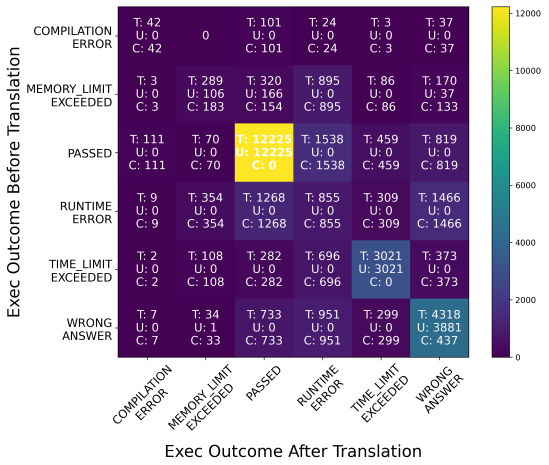


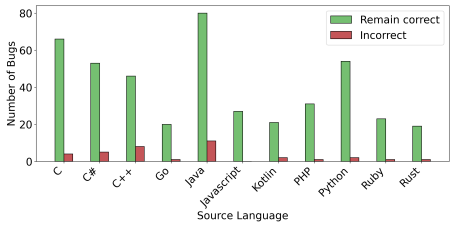
Figure 13: Heatmap of bug execution transition before and after translation (T: total number of test cases, U: test cases where the outcome remains unchanged after translation, C: test cases where the outcome changes after translation).

5.4.3 Experimental Results for RQ2.2 (Translation Consistency). Figure 13 presents a heatmap of transitions among the six bug categories (as introduced in Section 5.2.2) before and after translation. For each transition, we compare the test case outcomes of PASSED and WRONG ANSWER to validate the consistency, while the other categories output compiler-specific error messages, resulting mainly in changed outcomes after translation. In particular, two groups of transitions are illustrated in the heatmap, (1) consistent transitions (on the diagonal), and (2) the remaining inconsistent transitions.

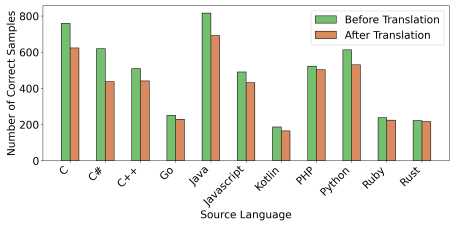
Before translation: <lang> C <exec_outcome> COMPILATION_ERROR <result> test.c: In function 'main': \ntest.c:18:20: expected ';' before 'else' **After translation:** <lang> Python <exec_outcome> PASSED <result> 7	**Before translation:** <lang> C <exec_outcome> MEMORY_LIMIT_EXCEEDED <result> 0 **After translation:** <lang>Rust <exec_outcome> PASSED <result> 0

Figure 14: Examples of inconsistent transition.

For the consistent transitions, we notice that a total of 20750 out of all 32277 cases remain consistent in coarse-grained categories. In particular, the consistent PASSED transitions account for 12225 cases. Moreover, 3881 out of 4318 WRONG ANSWER transitions, and all 3021 TIME LIMIT EXCEEDED transitions remain consistent after translation. Since compilers of different languages produce different error information even for the same category of bugs, the output of COMPILATION ERROR and RUNTIME ERROR is changed after translation, except for MEMORY LIMIT EXCEEDED transitions where C/C++ compilers still return answers for the problem while Java compilers throws an *OutOfMemory* exception, resulting in partial consistency in the outcomes.



(a) Number of fixed bugs that remain correct or become incorrect after back-translation.



(b) Number of correct samples before and after back-translation.

Figure 15: Validation for back-translation.

First, we notice that most bugs with COMPILATION ERROR can be fixed early in the translation phase, or transformed into other categories of bugs if minor additional issues remain within them. Figure 14 shows two examples of inconsistent transition. The first example presents a COMPILATION ERROR case of a C bug missing a semicolon ";", which is fixed after being translated to Python. Since Python syntax requires no semicolons, this compilation problem is naturally repaired after translation. Such compilation errors can be fixed even when translating to languages that require semicolons, revealing the inherent features of LLM-based code translation. Secondly, language-specific differences such as memory and time usage are also critical contributors to the transition inconsistency. The second example in Figure 14 illustrates the transition of a MEMORY LIMIT EXCEEDED case of a C bug to a PASSED case in Rust, which consumes less memory resource, thus enabling the repair. A very small subset of cases, where WRONG ANSWER cases are

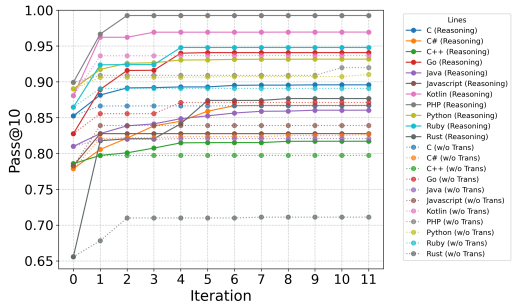


Figure 16: Pass@10 across iterations with and without translation (Reasoning and w/o Trans refer to the reasoning strategy and without translation).

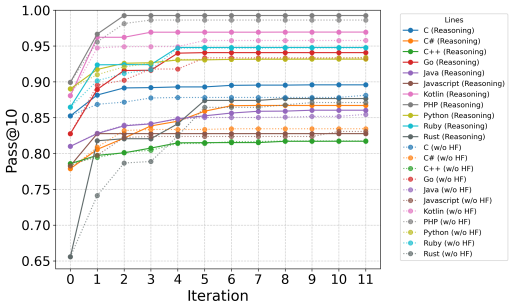


Figure 17: Pass@10 across iterations with/without historical feedback (w/o HF refers to without historical feedback).

transformed into PASSED ones after translation also contribute to the inconsistency.

Figure 15a shows the number of fixed bugs that remain correct and those that become incorrect after back-translation. Java experienced the greatest loss with 11 out of 91 fixed bugs incorrectly translated, which remains a small number of bugs compared to its correctly translated bugs. Figure 15b exhibits the number of correct samples generated for different bugs before and after back-translation, among which C# suffers from a maximal loss of 182 correct samples, whereas Rust only encounters a minimal loss of six correct samples. Our statistical tests on correct samples before and after back-translation yield a p-value of 0.39 (> 0.05) and a Cliff's Delta of 0.22, confirming that the loss caused by back-translation is not significant.

> ***[RQ-2] Findings:*** *(1) The LLM-based reasoning can effectively identify optimal target languages where model exhibits superior repair ability, achieving a minimal average translation path of 2.52. (2) The bugs addressed by LANTERN are complex and hard to tackle, with some resolved bugs having median difficulty 400 to 500 larger. (3) Although language-specific errors sometimes introduce inconsistencies, translation generally preserves most bug semantics (with 12225 consistent PASSED cases and 3881 out of 4318 consistent WRONG ANSWER cases) and can even naturally repair certain defects. (4) Back-translation preserves semantic consistency for most repaired code, with an overall loss of only 7.56% of fixed bugs while 85.95% of correct samples are preserved.*

5.5 RQ3: Ablation Study

5.5.1 Experimental Design. We perform an ablation study to investigate the contribution of the key component and design choice including code translation and historical feedback. We aim to determine the impact of code translation on program repair performance and whether incorporating historical feedback assists the LLM in performing accurate reasoning.

5.5.2 Experimental Results. Figure 16 exhibits the *Pass*@10 results of the reasoning strategy and direct repair without translation. We notice that repair without translation shows only a slight performance improvement during the initial iteration, after which further fixes become consistently difficult to achieve. Notably, the reasoning strategy achieves a 16.55% improvement in *Pass*@10 for Rust compared to direct repair, followed by PHP, Go, and Ruby, with improvements of 7.27%, 6.96%, and 5.71%, respectively. Our statistical analysis of the final Pass@10 improvements confirms that translation-based repair significantly outperforms direct repair with a p-value of $1.95E-03$ and Cliff's Delta of 0.42 (medium effect).

Figure 17 compares using the reasoning strategy with and without analysis on historical feedback. We see that the final *Pass*@10 performance of the two remains similar, which is expected since both are fundamentally translation-based repair like other strategies. However, the most significant discrepancy manifests during the early iteration cycles, where incorporating historical feedback provides noticeable advantages. For example, in Rust, our approach achieves a *Pass*@10 of 0.82 in the first iteration compared to only 0.74 without historical feedback. Without this feedback, the LLM selects target languages solely based on the bug itself, leading to suboptimal target languages.

> ***[RQ-3] Findings:*** *(1) The bug translation component contributes significantly to enhancing program repair effectiveness, with its most substantial impact observed in Rust, achieving a 16.55% improvement in Pass@10. (2) Identifying appropriate target languages is crucial, as historical feedback facilitates the early convergence of repairs by guiding the selection process toward the most promising options.*

5.6 RQ4: Generalizability

5.6.1 Experimental Design. In RQ4.1, we evaluate LANTERN on two real-world benchmarks. For Defects4J, we follow ChatRepair's settings, focusing on the most comprehensive single-function subset (255 bugs from v1.2) and the filtered set of 78 bugs from v2.0. While ChatRepair used GPT-3.5, we employ DeepSeek-V3 for a more current comparison. For SWE-bench, we align with AGENTLESS evaluation settings, leveraging their fault localization results and extracting 10-line context windows around bug locations. We use AGENTLESS 1.5 with GPT-4o and employ the same model for fair comparison, measuring performance by resolved instances. Both benchmarks use the same 11 target languages from xCodeEval.

Specifically, we translate the whole buggy function for Defects4J and the extracted context window (i.e., the bug location with a fixed number of context lines around it) for SWE-bench. In terms of bug repair, we provide the same context (e.g., problem description, buggy code, etc.) following the two baselines, respectively.

Table 2: Pass@10 Performance of Different LLMs on xCodeEval

LLM	Approach	C	C#	C++	Go	Java	JS	Kotlin	PHP	Python	Ruby	Rust
Claude 3.5 Sonnet	Direct	83.28	82.68	75.30	82.29	79.81	79.80	75.94	81.28	83.55	78.82	67.79
	LANTERN	90.42	87.34	80.81	90.46	85.99	86.40	92.89	99.23	90.66	85.89	86.09
QWen2.5-72B-Instruct	Direct	79.06	69.50	65.90	74.97	67.19	82.61	84.47	81.55	82.25	78.23	45.42
	LANTERN	86.09	79.92	75.26	85.11	76.89	88.30	88.21	93.22	88.37	86.69	72.00

Table 3: Performance on Defects4J and SWE-Bench Lite

Approach	D4J1.2	D4J2.0	Approach	SWE-Bench Lite
ChatRepair	141	54	AGENTLESS	95/300 (31.67%)
LANTERN	**170**	**60**	LANTERN	**110/300 (36.67%)**

For RQ4.2, we investigate LANTERN's generalizability across different models by replicating core xCodeEval experiments using Claude 3.5 Sonnet and Qwen2.5-72B-Instruct, with direct repair as the baseline. This determines whether our translation-based paradigm generalizes across model architectures.

5.6.2 Experimental Results for RQ4.1 (Real-world Generalizability). Table 3 shows LANTERN's performance on Defects4J and SWE-Bench compared to ChatRepair and AGENTLESS, respectively. On Defects4J 1.2, LANTERN resolves 170 bugs versus ChatRepair's 141 (+29 bugs). For Defects4J 2.0, LANTERN fixes 60 bugs compared to ChatRepair's 54, suggesting that while the source language is only Java, translating the buggy context to a different language (e.g., Python for its expressive syntax or C++ for its low-level control) appears to help the LLM overcome being stuck in a "local optimum" of incorrect repair attempts and produce more effective patches.

On SWE-Bench Lite, LANTERN achieves stronger results, resolving 110/300 instances (36.67%) versus AGENTLESS's 95/300 (31.67%). This benchmark requires resolving real-world GitHub issues across multiple files, yet LANTERN succeeds by translating only localized context windows around fault locations. This demonstrates that complete semantic representation of entire projects is unnecessary—focusing on relevant context maintains both scalability and efficiency for large-scale codebases.

The results indicate LANTERN's effectiveness extends beyond traditionally under-represented languages. Even for widely-used languages like Java and Python, LANTERN transforms difficult instances into more tractable problems by leveraging different programming languages' unique strengths, suggesting it functions as a general-purpose strategy for enhancing problem-solving capabilities on complex bugs.

5.6.3 Experimental Results for RQ4.2 (Model Generalizability). Table 2 presents Pass@10 results for Claude 3.5 Sonnet and QWen2.5-72B-Instruct on xCodeEval. LANTERN elevates Claude's performance across every language, with massive boosts in Kotlin (+16.94%), PHP (+17.95%), and Rust (+18.30%), demonstrating that even frontier proprietary models benefit significantly from cross-language translation.

Results with QWen2.5-72B-Instruct further highlight LANTERN's power for open-source models. QWen's baseline shows significant variance, particularly struggling with languages requiring strict compilation or complex type systems like Rust (45.42%) and C# (69.50%). LANTERN provides dramatic improvements on Rust (+26.58%), C# (+10.42%), Go (+10.13%), and PHP (+11.67%), revealing that LANTERN can elevate strong open-source models to achieve performance competitive with or superior to highly-capable proprietary models. For instance, QWen with LANTERN achieves 72.00% on Rust, surpassing Claude's direct repair score of 67.79%.

> ***[RQ-4] Findings:*** *(1) LANTERN outperforms ChatRepair and AGENTLESS on the challenging real-world benchmarks, which demonstrates strong generalizability to real-world issue resolution, revealing its cross-language translation paradigm is effective for complex, real-world software defects, including those that require understanding context across multiple files. (2) LANTERN demonstrates model-agnostic generalizability across different LLMs. It consistently and significantly enhances the repair capabilities of both state-of-the-art closed-source and leading open-source LLMs.*

6 Threats to Validity

Internal. The internal threat arises from the historical feedback, where as iterations progress and more historical feedback is generated, subsequent reasoning may increasingly bias toward previously successful languages. We mitigate this by limiting that each language can only be selected once, and comparing the reasoning strategy against both greedy and random strategies, showing that the feedback-based strategy outperforms the others.

Construct. We primarily use Pass@10 for evaluation, which might not capture all repair aspects. To address this, we incorporate multiple additional ranking metrics to provide a more comprehensive assessment. Another threat comes from potential semantic inconsistencies introduced during translation. We mitigate this by validating the semantic consistency for both bug translation and fixed code back-translation through extensive test cases.

External. We evaluate on a large number of bugs (xCodeEval), demonstrating effectiveness across 11 diverse programming languages and bug types. Furthermore, our significant improvements on less common languages like Rust suggest that the approach generalizes beyond mainstream languages. Our approach can also potentially be applied to repository-level codebases. The pipeline's design can also be extended to additional programming languages and models.

7 Conclusion

In this paper, we presented a novel program repair approach, LANTERN, which leverages cross-language code translation with multi-agent iterative refinement to fix bugs by translating buggy code to languages where the LLM demonstrates stronger repair capabilities based on the bug characteristics and historical feedback. Evaluation on xCodeEval with 5,068 bugs across 11 programming languages shows that our approach can significantly enhance the repair capability of the LLM, with notable improvements in less common languages like Rust (with a 22.09% increase in *Pass*@10). Our results demonstrate the potential of cross-language program repair in effectively extending the repair capabilities of the LLM, revealing new opportunities for agent-guided automated program repair.

References

[1] GitHut 2.0. 2024. Github Language Stats. https://madnight.github.io/githut/#/pull_requests/2024/1.

[2] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2024. On hardware security bug code fixes by prompting large language models. *IEEE Transactions on Information Forensics and Security* (2024).

[3] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*. 1443–1455.

[4] Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. 2022. SYNSHINE: improved fixing of syntax errors. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2169–2181.

[5] Anthropic. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet Software.

[6] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit Seshia, and Alvin Cheung. 2025. Verified code transpilation with LLMs. *Advances in Neural Information Processing Systems* 37 (2025), 41394–41424.

[7] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).

[10] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.

[11] Codeforces. 2024. Codeforces. https://codeforces.com/.

[12] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing rust compilation errors using llms. *arXiv preprint arXiv:2308.05177* (2023).

[13] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[14] Zhiyu Fan, Haifeng Ruan, Sergey Mechtaev, and Abhik Roychoudhury. 2024. Oracle-guided Program Selection from Large Language Models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 628–640.

[15] 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.

[16] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2018. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*. 1219–1219.

[17] Dávid Hidvégi, Khashayar Etemadi, Sofia Bobadilla, and Martin Monperrus. 2024. Cigar: Cost-efficient program repair with llms. *arXiv preprint arXiv:2402.06598* (2024).

[18] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1471–1493.

[19] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).

[20] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2454–2476.

[21] Aryan Jadon and Avinash Patil. 2024. A comprehensive survey of evaluation techniques for recommendation systems. In *International Conference on Computation of Artificial Intelligence & Machine Learning*. Springer, 281–304.

[22] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[23] Feihu Jin, Yifan Liu, and Ying Tan. 2024. Zero-shot chain-of-thought reasoning guided by evolutionary algorithms in large language models. *arXiv preprint arXiv:2402.05376* (2024).

[24] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[25] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[26] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[27] Mohammad Abdullah Matin Khan, M Saiful Bari, Do Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 6766–6805.

[28] Cheryl Lee, Chunqiu Steven Xia, Longji Yang, Jen-tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R Lyu. 2024. A unified debugging approach via llm-based multi-agent synergy. *arXiv preprint arXiv:2404.17153* (2024).

[29] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[30] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977* (2024).

[31] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 298–312.

[32] Wenqiang Luo, Jacky Wai Keung, Boyang Yang, He Ye, Claire Le Goues, Tegawende F Bissyande, Haoye Tian, and Bach Le. 2024. When Fine-Tuning LLMs Meets Data Privacy: An Empirical Study of Federated Learning in LLM-Based Program Repair. *arXiv preprint arXiv:2412.01072* (2024).

[33] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[34] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[35] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 2228–2240.

[36] Octoverse. 2022. The top programming languages. https://octoverse.github.com/2022/top-programming-languages.

[37] Octoverse. 2024. AI leads Python to top language as the number of global developers surges. https://github.blog/news-insights/octoverse/octoverse-2024/.

[38] Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xin Peng, and Zibin Zheng. 2024. Repository-level code translation benchmark targeting rust. *arXiv preprint arXiv:2411.13990* (2024).

[39] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[40] Daniel Ramos, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2024. BatFix: Repairing language model-based transpilation. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 1–29.

[41] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems* 33 (2020), 20601–20611.

[42] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 23–30.

[43] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/

[44] Ali Tehrani, Arijit Bhattacharjee, Le Chen, Nesreen K Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. 2024. CodeRosetta: Pushing the Boundaries of Unsupervised Code Translation for Parallel Programming. *Advances in Neural Information Processing Systems* 37 (2024), 100965–100999.

[45] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[46] Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, et al. 2024. Repotransbench: A real-world benchmark for repository-level code translation. *arXiv preprint arXiv:2412.17744* (2024).

[47] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.

[48] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2348–2368.

[49] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 196–202.
[50] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.
[51] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.
[52] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.
[53] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv preprint arXiv:2310.04951* (2023).
[54] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 Technical Report. *arXiv preprint arXiv:2407.10671* (2024).
[55] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2025. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2025), 50528–50652.
[56] Yanming Yang, Xing Hu, Zhipeng Gao, Jinfu Chen, Chao Ni, Xin Xia, and David Lo. 2024. Federated Learning for Software Engineering: A Case Study of Code Clone Detection and Defect Prediction. *IEEE Transactions on Software Engineering* (2024).
[57] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. HDLdebugger: Streamlining HDL debugging with large language models. *arXiv preprint arXiv:2403.11671* (2024).
[58] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 678–690.
[59] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2025. Scalable, validated code translation of entire projects using large language models. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1616–1641.
[60] Jie M Zhang, Feng Li, Dan Hao, Meng Wang, Hao Tang, Lu Zhang, and Mark Harman. 2019. A study of bug resolution characteristics in popular programming languages. *IEEE Transactions on Software Engineering* 47, 12 (2019), 2684–2697.
[61] Lyuye Zhang, Kaixuan Li, Kairan Sun, Daoyuan Wu, Ye Liu, Haoye Tian, and Yang Liu. 2024. Acfix: Guiding llms with mined common rbac practices for context-aware repair of access control vulnerabilities in smart contracts. *arXiv preprint arXiv:2403.06838* (2024).
[62] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–69.
[63] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A systematic literature review on large language models for automated program repair. *arXiv preprint arXiv:2405.01466* (2024).
[64] Xing Zhang, Jiaheng Wen, Fangkai Yang, Pu Zhao, Yu Kang, Junhao Wang, Maoquan Wang, Yufan Huang, Elsie Nallipogu, Qingwei Lin, et al. 2025. Skeleton-Guided-Translation: A Benchmarking Framework for Code Repository Translation with Fine-Grained Quality Evaluation. *arXiv preprint arXiv:2501.16050* (2025).
[65] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.
[66] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.
[67] 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 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE). *IEEE Computer Society* (2024), 872–872.
[68] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual code snippets training for program translation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 36. 11783–11790.