HapRepair: Learn to Repair OpenHarmony Apps

Zhihao Lin*
Beihang University
China
mathieulin@buaa.edu.cn

Mingyi Zhou*
Beihang University
China
zhoumingyi@buaa.edu.cn

Wei Ma
Singapore Management University
Singapore
weima93@gmail.com

Chi Chen
Fudan University
China
15110240004@fudan.edu.cn

Yun Yang†
Yunnan University
China
yangyun@ynu.edu.cn

Jun Wang†
Beihang University
China
junwang.lu@gmail.com

Chunming Hu
Beihang University
China
hucm@buaa.edu.cn

Li Li†
Beihang University
Yunnan Key Laboratory of Software Engineering, China
lilicoding@ieee.org

Abstract

Software defect detection and repair are essential software engineering tasks that mitigate potential risks in the early development stages. Large Language Models (LLMs) have demonstrated significant capabilities in software defect detection and repair. However, it is hard for LLMs to handle the new programming languages such as ArkTS (which is predominantly used in the OpenHarmony platform) due to training data shortage. Additionally, LLM-based multi-defect repair suffers from the limitation of the context window of LLMs. These issues significantly affect the performance of LLM-based defect repair in new programming languages. To address the above challenges, we propose *HapRepair*, a defect repair framework that integrates static analysis tools with retrieval-augmented generation (RAG) to improve the effectiveness of the defect repair. Specifically, we integrate *CodeLinter* into our iterative defect repair framework for defect detection, which is the basis of defect repair, and utilize RAG together with *ArkAnalyzer* to improve the quality of our repair solutions. To overcome the context window limitation of LLMs, we propose the *Surrounding Context Extractor* and the *Context Combination Tool*. Experiment results show that *HapRepair* effectively repairs defects in OpenHarmony Apps, demonstrating high reliability and efficiency in addressing code issues, achieving a defect repair rate of about 99% on the test set, compared to only about 37% when directly using LLMs for defect repair based on the defect information. Our approach demonstrates a robust solution for defect repair on new programming languages that have limited code corpus.

*Equal Contribution
†The corresponding author

FSE Companion '25, June 23–28, 2025, Trondheim, Norway
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728556

ACM Reference Format:
Zhihao Lin, Mingyi Zhou, Wei Ma, Chi Chen, Yun Yang, Jun Wang, Chunming Hu, and Li Li. 2025. *HapRepair*: Learn to Repair OpenHarmony Apps. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3696630.3728556

1 Introduction

Software defects can result in a series of severe consequences such as system crashes, functional failures, and security vulnerabilities [3, 43], which significantly impact the user experience. Additionally, the accumulation of defects increases both the complexity and cost of maintaining the software [7]. Therefore, it is essential to detect and repair the defects in the early stage [45] to ensure the quality and long-term sustainability of the software.

Existing approaches for defect detection and repair can be divided into two categories: traditional approaches using static analysis and learning-based approaches. Traditional approaches, which often rely on static analysis, examine source code without executing it. Tools like SonarQube [49] and ESLint [16] are typical traditional approaches, which have been widely used to detect defects such as memory leaks, unused variables, and inconsistent code styles. These tools show high performance when detecting defects with predefined rules. However, their performance in defect repair is limited, as these tools primarily provide examples and recommendations or highlight issues without offering automated solutions. This limitation hinders their effectiveness in repairing tasks automatically, especially in dynamic and fast-evolving ecosystems like OpenHarmony.

Learning-based approaches usually train models on large amount of data [11, 50, 52]. Recent advances in deep learning, particularly transformer-based models like CodeBERT [19], have demonstrated significant capability in learning patterns from existing datasets and achieving high performance on evaluation benchmarks (e.g., Defects4J [26], CWE-bench) to suggest defect repairs. With the advent of LLMs, many researchers have been drawn to their powerful capabilities in understanding both the syntax and semantics of code [35–37]. It's obvious that the performance of learning-based approaches relies on the availability of high-quality and diverse training data [2]. However, the lack of publicly available data is a significant issue for ArkTS, which will directly affect the performance of LLMs for effective defect detection and repair due to a lack of training samples. In addition, LLMs often suffer from the maximum length of the context window, which is a limitation for multi-defect repair in a large code file.

To address these challenges, we introduce *HapRepair*, a novel approach designed to detect and repair code defects in OpenHarmony apps. **Firstly**, to overcome the challenge of the lack of training data, we make full use of the advantage of static analysis. We leverage and integrate a static analysis tool *CodeLinter* for defect detection, which not only effectively detects defects, but also interacts with LLMs to evaluate the defect repair quality. In addition, we utilize RAG to retrieve specific knowledge with in-context learning to alleviate the problem of the lack of training data. **Secondly**, to overcome the challenge of the limitation of the context window for multi-defect repair, we propose two analysis tools: *Surrounding context extractor* and *Context combination tool. Surrounding context extractor* focuses on extracting relevant context snippets from the source code, removing redundant or unnecessary parts, and retaining the key information for the repair task. *Context combination tool* further combines the overlapping context. Experiment results show that *HapRepair* achieves a defect repair rate of about 99% on the test set, compared to only about 37% when directly using LLMs for defect repair based on the defect information. By integrating static analysis tools with LLMs and RAG, *HapRepair* combines the precise defect detection capabilities of static analysis with the generative repair strengths of LLMs. Furthermore, the static analysis feedback not only aids in identifying defects but also provides structured insights that improve repair quality. In summary, this approach offers a more robust defect solution for low-resource new programming languages like ArkTS, ultimately improving their code quality.

The main contributions of this study are shown as follows:

- We present *HapRepair*, the first framework combining static analysis tool, RAG and LLMs, specifically designed to repair defects in low-resource programming languages like ArkTS.
- To mitigate the issue of LLMs dealing with multi-defection in a single code file, we design the *Surrounding context extractor* and *Context combination tool* to overcome the limitation of the context window.
- We manually curate a specialized dataset of 271 defect-repair pairs across 37 performance-related rules tailored to ArkTS, providing a foundational resource for future research and development in ArkTS defect detection and repair.
- We have open-sourced our tool by making it available on our repository: https://github.com/SMAT-Lab/HapRepair.

2 Background

In this section, we will introduce the basic background of this paper.

2.1 OpenHarmony and ArkTS

OpenHarmony is an operating system governed by the OpenAtom Foundation. It aims to create a unified and intelligent platform for connecting all devices. OpenHarmony has evolved into a community-driven operating system supported by multiple organizations and developers worldwide. Its modular architecture facilitates seamless interconnection and cooperation across a wide range of smart devices, including smartphones, smart homes, automotive systems, and wearables. OpenHarmony has shown significant potential, especially in the rapidly expanding Internet of Everything (IoE) and smart device markets. It has been deployed in millions of devices across various domains. It also has attracted many developers delving into the OpenHarmony ecosystem. Current researches on OpenHarmony explore enhancing its technical capabilities and application scenarios, with a focus on distributed computing, security, and real-time processing. As highlighted by Li et al. [30], OpenHarmony presents a unique opportunity for advancing software engineering practices tailored for IoE and distributed systems.

ArkTS, the official programming language of developing OpenHarmony Apps, extends TypeScript with features tailored for developing applications in multi-platform environments. It not only retains the strengths of TypeScript, but also adds some new features to optimize performance and usability for mobile applications. One of such newly added features is the ArkUI framework, which is a declarative framework that simplifies UI development by directly binding the interface to reactive data. This feature makes it easier for developers to develop the OpenHarmony Apps. However, the fundamental difference between ArkTS and TS, such as ArkTS-specific syntax, semantics, and unique features, makes the existing defect detection and repair tools for TS unable to be applied to ArkTS. Therefore, we need to build a new tool for defect detection and repair for ArkTS.

2.2 Defect Detection and Repair

2.2.1 ***Defect Detection.*** Defect detection is a crucial process in software engineering that detects potential issues directly from source code before execution. Traditional approaches often rely on static analysis tools [5, 8], such as SonarQube and ESLint, which have been widely adopted for detecting coding errors, memory leaks, and security risks. These tools remind developers to proactively address potential issues and prevent future problems.

Recent advancements in deep learning-based approaches (including LLMs) have shown the potential for defect detection [21, 35–37]. By training on large datasets of labeled code snippets, these models can predict defect types and provide actionable insights. However, the performance of these approaches heavily depends on the availability of high-quality and diverse training data [2]. In scenarios involving new programming languages, such as ArkTS in OpenHarmony Apps, the lack of datasets significantly limits their ability to detect defects.

2.2.2 ***Defect repair.*** Defect repair is to provide repair suggestions for defective code or automatically repair defective code. Traditional approaches for defect repair are often rule-based and template-based approaches that provide common repair suggestions [27, 34, 38]. However, these approaches struggle to repair defects that do not belong to predefined rules and templates. In addition, these approaches have difficulty in repairing complex and context-dependent defects [42]. LLM-based approaches for defect repair have gained considerable attention in recent years [18]. These approaches leverage LLMs to understand the syntax and semantics of code and then to automatically repair defects. For example, CodeX and GPT-3 developed by OpenAI, which have been trained on large amounts of code corpora, possess the capability of defect repair. However, these approaches also heavily depend on the availability of high-quality and diverse training data [55]. In addition, these approaches often suffer from the maximum length of the context window [47], which is a limitation for multi-defect repair. Furthermore, the generated repairs by these approaches may introduce other defects [25]. Thus, we propose an iterative defect repair framework to mitigate limitations in current approaches.

2.3 *ArkAnalyzer*

ArkAnalyzer [9] is a static analysis framework. It has a set of common features (e.g., call graph construction) that are recurrently required when implementing in-depth static analyzers such as privacy leak detectors and compatibility issue detectors, providing high efficiency and effectiveness in analyzing the code of ArkTS. Thus, *ArkAnalyzer* is a natural and good choice for us to integrate into *HapRepair* for functionality verification.

2.4 *CodeLinter*

CodeLinter is a static analysis tool specifically designed for ArkTS. It designs an extensible rule set that is capable of detecting defects across multiple dimensions, including security, performance, and coding style. *CodeLinter* analyzes source code and provides developers with detailed defect information, such as the violated rules, their descriptions, and the corresponding line numbers. Additionally, it supports developers in defining custom rules, enhancing its flexibility and scalability. It is suitable for us to integrate *CodeLinter* into *HapRepair* for defect detection.

2.5 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is a hybrid approach that combines the strengths of retrieval systems and large language models. Unlike traditional LLM approaches, which rely solely on the internal knowledge of the model itself, RAG leverages retrieval techniques to incorporate external knowledge to improve generation. The retrieved knowledge from RAG is used in the prompt of LLMs; thus, it is very flexible. In the scenarios of defect detection and repair, RAG is an important way to retrieve external defect knowledge, such as similar defects, fixes, and best practices, to improve the generation of LLMs.

3 Preliminary Study

As mentioned before, the approaches for defect detection and repair can be divided into two categories, which are traditional approaches and learning-based approaches. Thus, we need to conduct a preliminary experiment to evaluate the performance of these two categories for defect detection and repair, which will help us to choose a suitable approach for defect detection and repair respectively. We focus on the defect detection and repair for ArkTS, so we aim to look for traditional approaches designed for ArkTS. However, *CodeLinter* is an off-the-shelf tool designed for defect detection for ArkTS and is not suitable for defect repair for ArkTS. Thus, we compare *CodeLinter* with LLMs in the scenario of defect detection to choose a better one. For defect repair, there are no existing tools specifically designed for ArkTS, so we can directly utilize LLMs.

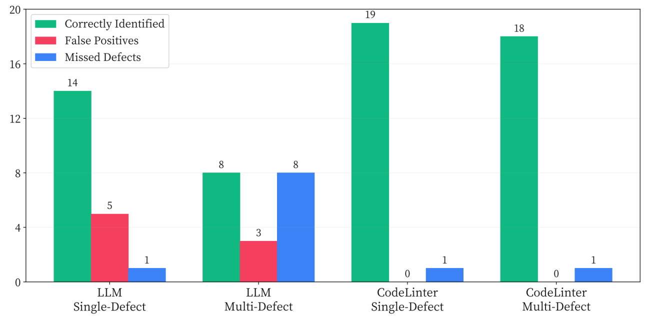

Figure 1: In-context Learning Defect Detection for ArkTS

3.1 Setting

3.1.1 ***Model.*** We use a powerful LLM, the GPT-4o, to conduct the preliminary experiment. The decision to use GPT-4o for defect detection is based on the following considerations:

- GPT-4o performs better than other LLMs in many tasks, which indicates its strong abilities.
- GPT-4o is trained with TypeScript datasets, which makes it possible to perform defect detection with in-context learning.
- In-context learning requires a sufficient context to make more accurate predictions, and GPT-4o has the capacity to handle long context.
- It is difficult to prepare enough training data of defect detection for ArkTS to fine tune a model.

3.1.2 ***Dataset.*** We manually create 47 example pairs of ArkTS, each containing a defect and its corresponding defect-free version, and the location of the defect. These pairs are then used for in-context learning. For testing, we prepare 20 files with single defects and 5 files with multiple defects, which totally contain 39 defects.

3.2 Comparison

3.2.1 ***Steps.*** Initially, we explore the use of in-context learning with LLM for defect detection in ArkTS. For each defect type, we manually craft positive and negative examples according to the defects' rules and then provide these along with defect types and descriptions as prompts to LLMs. The result is then assessed by human experts to determine whether a specific defect is present in the target. We also use *CodeLinter* for the same defect detection task. We use both files containing a single defect and files containing multiple defects to thoroughly test the capability of the model in detecting ArkTS-specific defects.

3.2.2 ***Results.*** As shown in Figure 1, LLM demonstrates the ability to effectively detect defects in single-defect files, which achieves a detection rate of 14 out of 20. However, for multi-defect files, LLMs can only detect 8 out of 19 defects and also generate numerous false positives and missed detections. In contrast, *CodeLinter* shows superior performance, successfully identifying 19 out of 20 defects in single-defect files and 18 out of 19 defects in multi-defect files, with no false positives and minimal missed detections in both scenarios. As a result, it is better to use *CodeLinter* for defect detection for ArkTS.

Answer for this study: LLM only detect 8 out of 19 defects for multi-defect files. It shows that we need to integrate the static analysis tool *CodeLinter* into our method, which performs better than LLM in all scenarios.

3.3 Motivation of Our Approach

From the result of our preliminary study, we can see that the traditional approach has its own advantages, specifically in the scenario of defect detection for ArkTS. Thus, we are motivated that it is necessary to propose an overall framework to leverage the advantages of different approaches to overcome the limitations mentioned before. Specifically, we are motivated to propose a defect repair framework that combines static analysis tools, RAG and LLMs to repair defects in low-resource programming languages like ArkTS. We leverage and propose static analysis tools to analyze and extract information for RAG and LLMs. RAG and LLMs are responsible for generating more precise defect repairs.

4 Approach

In this section, we will detail our approach *HapRepair*.

4.1 Overview

To address the challenges of defect detection and repair for ArkTS in OpenHarmony applications, we propose *HapRepair*, an automated iterative defect repair framework that combines static analysis tools and RAG with LLMs. This integrated approach leverages the strengths of static analysis for precise defect detection and precise context extraction while utilizing LLMs and RAG for generating intelligent and targeted repair patches. The overall workflow of *HapRepair* is illustrated in Figure 2.

HapRepair has four main phases, which are Defect Detection phase, Defect Context Extraction phase, Knowledge-Augmented Repair phase, and Functionality Verification phase. These phases form an iterative process. Defect Detection phase is responsible for utilizing *CodeLinter* to detect defects in the original input code and checking whether there is any defect in the repaired code. Defect Context Extraction phase aims to analyze, extract, and combine related surrounding context of a defect using defect information provided by Defect Detection phase. Knowledge-Augmented Repair phase is an important phase to retrieve related external knowledge related to defect repair using defect context from Defect Context Extraction through a knowledge database and a vector database, which will improve the final generation of LLM. Once the repair code is generated, Functionality Verification phase is responsible for verifying whether the repaired code has the same meaning with the original code. Finally, to evaluate the quality of the repaired code, Defect Detection phase will execute again.

By integrating static analysis tools, LLMs, and RAG, *HapRepair* achieves a balance between precision and flexibility. Its iterative and context-aware design enables it to handle both simple and complex defects on low-resource programming languages like ArkTS, and ensure high-quality repairs tailored to the needs of OpenHarmony applications.

4.2 Defect Detection

As shown in Section 3, directly using LLMs as a defect detection tool is not the best choice, particularly in scenarios like ArkTS where publicly available data are limited. To address this challenge, it is better for us to leverage *CodeLinter* as the defect detection tool in our framework.

CodeLinter is specifically designed to identify defects in source code based on predefined rules and patterns, ensuring comprehensive coverage of typical issues in ArkTS applications. Since *CodeLinter* is an off-the-shelf tool, we can directly integrate it into *HapRepair* for defect detection. Given a code, *CodeLinter* can provide precise defect detection information such as the type, description, and localization information of a defect, which serves as the foundation for defect repair. Compared to the LLM-based defect detection approaches, we can reliably detect defects and minimize noise and inaccuracies by using the static analysis tool *CodeLinter*. This guarantees that defect repair processes are built on accurate insights.

In addition, *CodeLinter* can also be used to evaluate whether there is any defect in the repaired code. Since the repaired code generated by the LLM is not always absolutely correct, which means that it may introduce other defects or the original defect is not repaired. Thus we need to make an inspection. *CodeLinter* is the natural choice. If the repaired code has any defect detected by *CodeLinter*, *HapRepair* will start an iteration for defect repair until there is no defect in the repaired code or reaching the maximum number of iterations.

By integrating *CodeLinter* into the *HapRepair* framework, we not only address the limitation of LLM-based defect detection but also create a seamless workflow for iterative defect detection.

4.3 Defect Context Extraction

To effectively repair defects, it is crucial to accurately analyze and extract the context of the defect, which will be used as the context in the retrieval query and prompt for LLMs to improve the generation.

The defect localization information provided by *CodeLinter* reflects the line number where the defect occurs. However, it is not enough to just use the line that a defect occurs as context for repairing a defect. We need to precisely analyze and extract the surrounding context of the defect for repair. For example, the rule 'no-state-var-access-in-loop' used in *CodeLinter* requires avoiding frequent access to state variables within loops, such as 'for' or 'while' structures. When this defect is detected, it is necessary to locate the surrounding loop structure from the defect line.

We propose two specialized tools, which are the *Surrounding Context Extractor* and the *Context Combination Tool* to analyze, extract, and combine context. The *Surrounding Context Extractor* is responsible for analyzing and extracting the surrounding context

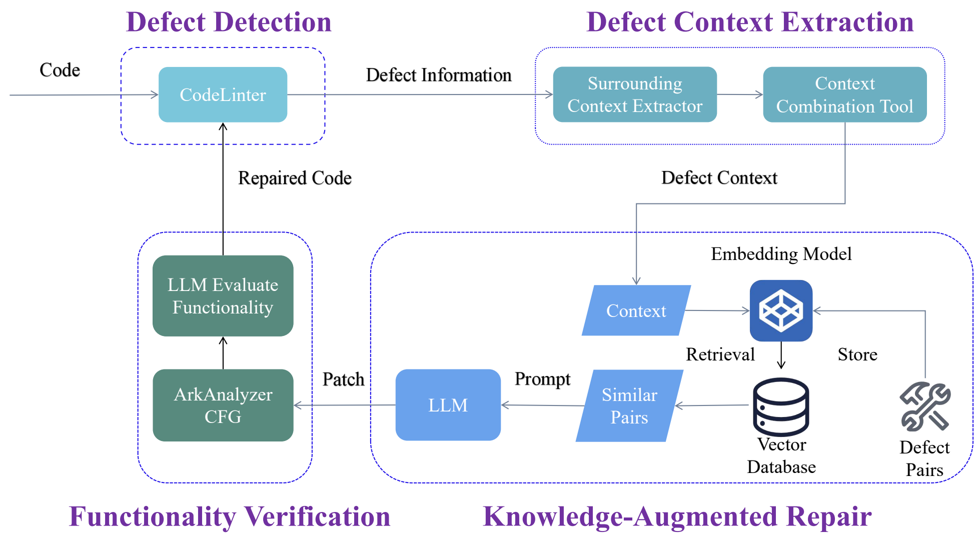

Figure 2: Overview of the *HapRepair*. *HapRepair* has four main steps, including defect detection, defect context extraction, knowledge-augmented repair, and functionality verification.

of defect code, which helps identify the broader code context and dependencies. The *Context Combination Tool* then merges defects that share overlapping contexts, allowing for better repair suggestions for multi-defect files. Initially, we try to utilize LLMs to extract and combine surrounding context. However, the result is not what we expect, which contains irrelevant context and hallucination.

Thus, we propose the static analysis approach to extract and combine context. The algorithm is shown in Algorithm 1, which contains the following steps:

(1) **Rule and Defect Code Initialization**: Rules are loaded from a predefined file to determine whether the defect needs more context or not. If additional context is needed, the corresponding defect code is extracted for analysis; otherwise, the defect line itself is directly used for retrieval and repair. Information on whether a rule requires additional context is open-sourced in our code.
(2) **Defect Extraction**: Each file is scanned by *CodeLinter* to identify defect lines, along with the associated defect type and description.
(3) **Context Extraction**: Extract surrounding code blocks for the defect's line based on structural patterns and definitions or usages of key variables (e.g., @State). The nested blocks (e.g., inner scopes) will be filtered out to retain the blocks that are most relevant to the defect's context.
(4) **Context Combination**: Overlapping or adjacent blocks are detected and combined using a grouping mechanism (e.g., UnionFind), ensuring comprehensive coverage of the context of the defect.

Surrounding Context Extractor contains the first three steps and *Context Combination Tool* contains the final step. These two tools ensure precise and reliable extraction of context, which overcomes the limitations of the context window observed in LLMs.

Algorithm 1: Defect Context Extraction Algorithm

```
Input: Project Directory proj_dir, Defect Directory
       defect_dir, Rules File rules.json
Output: Merged defect contexts for all files
// Step 1: Rule and Defect Code Initialization
rules_dict ← LoadRulesFromJson(rules.json);
Initialize empty merged_results;
foreach file in defect_dir do
    // Step 2: Defect Extraction
    code_lines, defects ← GetDefectsFromFile(file);
    if defects is empty then
        continue
    Initialize empty blocks;
    foreach defect in defects do
        // Step 3: Context Extraction
        context ← ExtractContext(code_lines,
          defect.line, rules_dict);
        Append context to blocks;
    // Step 4 : Context Combination
    merged_blocks ←
      MergeOverlappingBlocks(blocks);
    Extend merged_results with merged_blocks;
return merged_results;
```

4.4 Knowledge-Augmented Repair

First of all, we need to construct a knowledge database that contains defect pairs for RAG. In real-world scenarios, various defect types are encountered. For instance, in OpenHarmony applications, defects are broadly classified into six main categories: General Rules (@typescript-eslint), Security Rules (@security), Performance Rules (@performance), Preview Rules (@previewer), Cross-Device Deployment Rules (@cross-device-app-dev), and ArkTS Code Style Rules (@hw-stylistic). Considering the user interaction in OpenHarmony applications, we primarily focus on performance rules, which impact response related to a smooth user experience.

We manually craft 271 defect pairs that cover 38 types of performance rules. Each performance rule is equipped with 3 to 10 defect pairs. Each defect pair contains a defective code and a corresponding repair. The defective code is designed according to the description of each rule, and the corresponding repair is applied manually. Additionally, we provide an explanation of the reason for each defective code. Furthermore, we utilize the 'difflib' library to analyze the changes between defective code and repaired code.

In order to support the retrieval process, each defective code in the knowledge database is embedded as vectors, which form a vector database. When a surrounding context is given from the Defect Context Extraction phase, it is also embedded as a vector and used as a query to retrieve the most similar and related defective codes under the same defect type of the query from the vector database. Once the defective codes are retrieved, the corresponding repairs and other related knowledge can also be retrieved. We do not use the entire file containing the defect as a query because it contains noise (irrelevant) information, which leads to worse retrieval results.

Once the related knowledge has been retrieved, the prompt can be construed. The prompt consists of the defect type, the description of the defect type, the surrounding context of the defect, a representative similar defective code and its corresponding repaired code together with the explanation, and of the repaired code. The defect type, the description of the defect type, and the surrounding context of the defect are basic information. We assume that the similar defect repair example will act as a reference for LLM, so we add a representative similar defective code and its corresponding repaired code together with the explanation in the prompt. We also assume that the changes of the repaired code reflect the fine-grained operations, which will be a good guidance for the LLM. Thus, we add the changes of the repaired code in the prompt. Then, we input the prompt to the LLM and get the generated patch.

4.5 Functionality Verification

Ensuring the functionality equivalence between repaired code and original code is a critical phase in the automated defect repair process. However, it is difficult and expensive to dynamically prepare and run test cases for verification. Therefore, we use static functionality verification to check whether the repaired code has the same meaning with the original code. The functionality verification is achieved through a combination of static analysis and LLM-based evaluation. This dual-layer approach ensures that the repaired code not only resolves the detected defects but also preserves the overall functionality of the code.

Firstly, we leverage *ArkAnalyzer* to construct the CFG of both the original and repaired code. The CFG provides a structural representation of the execution flow of code, capturing the relationships between different functional blocks. Secondly, LLM is used to compare the CFG of the original and repaired code to ensure the functionality correctness. By comparing the CFG of the original and repaired code, we can detect whether any functional blocks from the original code have been omitted in the repaired code. Additionally, this comparison ensures that the modified functional blocks in the repaired code preserve the same execution logic and intended behavior as the original. Any repaired code that fails to pass the functionality verification will be rejected by *HapRepair*. Through this process, we verify that the repair not only resolves the detected defect but also maintains the overall functionality and correctness of the code.

4.6 Implementation

The implementation of *HapRepair* integrates static analysis tools, an embedding model, a vector database, and LLMs in a workflow to detect and repair defects in OpenHarmony Apps. For generating effective code fixes, *HapRepair* leverages RAG that combines the capabilities of a vector database and a powerful large language model. We utilize the Pinecone vector database to store and retrieve defect pairs, where defective code vectors are indexed alongside metadata such as defect rules, descriptions, defective code, repair code, and repair suggestions. This structured storage enables efficient retrieval of relevant examples based on the defect type and context, providing critical repair guidance.

We adopt GPT-4o as the LLM in *HapRepair*. We use the OpenAI API to interface with GPT-4o. The embedding model we use is stella_en_1.5B_v5 [17].

To facilitate interaction with static analysis tools, we set up a backend to execute commands for CodeLinter and ArkAnalyzer. *HapRepair* sends code files to the backend, where CodeLinter generates an xlsx file containing detailed defect reports, and ArkAnalyzer returns a CFG file representing the structure of both original code and repaired code. *HapRepair* processes these files to refine the repair and validation workflow.

5 Evaluation

In this section, we will evaluate the performance of our proposed *HapRepair*.

5.1 Experiment Setup

5.1.1 Research Question. The research questions evaluated in our study are shown below:

- **RQ1: How effective is *HapRepair* in repairing OpenHarmony apps developed by ArkTS in real-world scenarios?**
- **RQ2: How do the different factors of *HapRepair* contribute to its effectiveness?**
- **RQ3: What is the performance of *HapRepair* using different LLMs?**

In RQ1, we evaluate the overall performance of *HapRepair* in real-world OpenHarmony apps developed by ArkTS. Then, in RQ2, we conduct an ablation study to analyze the influence of different factors (such as top-n retrieved results) to explore the best setting.

Finally, we discuss the influence of the LLM chosen on repairing the defect of new programming languages.

5.1.2 ***Dataset****.* To collect data of the knowledge-database that can be used for RAG, we obtain the current defect types supported by *CodeLinter* from the official Huawei Developer website [13]. In this work, we focus on performance rules (@performance) because performance primarily impacts the smooth of user experience and performance defect is an important type when a defect occurs. Based on performance rules, we manually construct a knowledge database, which consists of 271 defective codes, along with their corresponding repairs, explanations for the applied repairs, and changes in the repairs. This dataset will then be leveraged as examples to support in-context learning, which are used to augment the prompts prepared for LLMs.

To construct the test dataset, we select data from three official open-sourced OpenHarmony organization repositories: OpenHarmony [20], OpenHarmony-SIG [23] and OpenHarmony-TPC [48]. We check each project and select the project that contains at least 10 defects of performance type. A certain number (such as 10) of defects in a project can avoid the occasionalism of the repair and increase the diversity of the repair.

5.1.3 ***Model****.* In our experiment setup, we utilize GPT-4o for both RQ1 and RQ2. For RQ3, we focus on comparing different models, including Llama-3.2-70b and Qwen2-72b-instruct, to evaluate whether the choice of LLMs would impact the performance of *HapRepair*.

5.1.4 ***Embedding model****.* We use the stella_en_1.5B_v5 model, which is driven by a balance between performance and model size [41], to convert the defective code into an 8192-dimensional vector to construct the vector database. With 1.5 billion parameters, the model provides a good trade-off between computational efficiency and embedding accuracy, making it suitable for large-scale language tasks without requiring excessive computational resources. Embedding provides a semantic understanding of code and enables efficient retrieval for defect detection and repair.

5.1.5 ***Metric****.* To evaluate the performance of *HapRepair*, we adopt two key metrics:

- **Number of Remaining Defects:** We count the number of defects detected by *CodeLinter* after the repair process is completed as the number of remaining defects.
- **Repair Rate:** It measures the proportion of defects that have been successfully repaired. The repair rate is calculated as the number of fixed defects divided by the total number of defects identified, typically expressed as a percentage.

5.1.6 ***Experimental Environment****.* *HapRepair* is evaluated on a workstation with Intel(R) Xeon(R) Platinum 8352V 3.50GHz CPU (144 cores), 256GB RAM, 4 NVIDIA A100 PCIe 80GB GPUs, and Ubuntu 24.04.1 LTS operating system.

Table 1: Number of Remaining Defects and Repair Rate of Projects Across Iterations

Project	Original	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
A21_C	20	1 (95%)	1 (95%)	0 (100%)	0 (100%)	0 (100%)
AdaptiveCapability	30	7 (77%)	5 (83%)	5 (83%)	5 (83%)	4 (87%)
ComponentCollection	223	18 (92%)	14 (94%)	14 (94%)	13 (94%)	7 (97%)
GSYVideo	15	1 (93%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
Gallery	301	29 (90%)	24 (92%)	10 (97%)	2 (99%)	0 (100%)
HealthyDietIX	28	16 (43%)	9 (68%)	3 (89%)	4 (86%)	0 (100%)
Photos	58	14 (76%)	7 (88%)	6 (90%)	6 (90%)	8 (86%)
StageModelAbilityDevelop	66	3 (95%)	2 (97%)	2 (97%)	2 (97%)	2 (97%)
amqplib	10	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
Arkts_tutorial	20	1 (95%)	1 (95%)	1 (95%)	1 (95%)	1 (95%)
utilCode	15	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
wifi_testapp	1116	126 (89%)	113 (90%)	45 (96%)	27 (98%)	7 (99%)
node-pop3	58	6 (90%)	4 (93%)	2 (97%)	1 (98%)	0 (100%)
Total	**1952**	**222** (89%)	**180** (91%)	**88** (96%)	**61** (97%)	**29** (99%)

Table 2: Number of Remaining Defects and Repair Rate of Different Types of Defects Across Iterations

Type	Original	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
remove-unchanged-state-var	672	44 (93%)	32 (95%)	12 (98%)	7 (99%)	1 (100%)
remove-redundant-state-var	605	23 (96%)	19 (97%)	8 (99%)	4 (99%)	0 (100%)
foreach-args-check	134	21 (84%)	9 (93%)	3 (98%)	3 (98%)	2 (99%)
no-state-var-access-in-loop	104	23 (78%)	35 (66%)	3 (97%)	3 (97%)	0 (100%)
remove-container-without-property	96	0 (100%)	0 (100%)	0 (100%)	1 (99%)	0 (100%)
remove-redundant-nest-container	70	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
use-row-column-to-replace-flex	70	7 (90%)	7 (90%)	7 (90%)	7 (90%)	6 (91%)
use-local-var-to-replace-state-var	62	0 (100%)	10 (84%)	7 (89%)	7 (89%)	0 (100%)
init-list-component	40	28 (30%)	26 (35%)	9 (78%)	5 (88%)	5 (88%)
multiple-associations-state-var-check	37	14 (62%)	4 (89%)	4 (89%)	0 (100%)	0 (100%)
high-frequency-log-check	14	3 (79%)	3 (79%)	3 (79%)	3 (79%)	3 (79%)
avoid-empty-callback	10	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
set-cache-count-for-lazyforeach-grid	8	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
use-reusable-component	7	5 (29%)	3 (57%)	4 (43%)	2 (71%)	2 (71%)
no-closures	6	2 (67%)	2 (67%)	2 (67%)	2 (67%)	2 (67%)
use-grid-layout-options	3	3 (0%)	3 (0%)	3 (0%)	0 (100%)	0 (100%)
use-transition-to-replace-animateto	3	3 (0%)	3 (0%)	3 (0%)	0 (100%)	0 (100%)
use-id-in-get-resource-sync-api	2	2 (0%)	2 (0%)	2 (0%)	2 (0%)	2 (0%)
no-stringify-in-lazyforeach-key-generator	2	1 (50%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
suggest-use-effectkit-blur	2	2 (0%)	2 (0%)	0 (100%)	0 (100%)	0 (100%)
use-scale-to-replace-attr-animateto	2	2 (0%)	2 (0%)	0 (100%)	0 (100%)	0 (100%)
use-onAnimationStart-for-swiper-preload	1	1 (0%)	1 (0%)	1 (0%)	1 (0%)	1 (0%)
combine-same-arg-animateto	1	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
waterflow-data-preload-check	1	1 (0%)	1 (0%)	1 (0%)	1 (0%)	1 (0%)
Sum	**1952**	**222** (89%)	**180** (91%)	**88** (96%)	**61** (97%)	**29** (99%)

5.2 RQ1: Repairing Performance

In order to evaluate the performance of *HapRepair*, we use it to repair the defects that are detected by *CodeLinter* in the projects we get in the test dataset. As introduced before, *HapRepair* extracts and combines the surrounding context of defects, retrieves the most similar detective code and repaired code pair from the vector database to construct a prompt. As a result, the prompt contains the defect type, the description of the defect type, the surrounding context of the defect, a representative similar defective code and its corresponding repaired code, the explanation for the repaired code, and changes of the repaired code. An example of prompt is shown in Figure 3. Given the prompt, the LLM will generate a patch. The generated patch is applied to the original code, and its functionality is verified for equivalence with the original code using its CFG. Subsequently, we re-evaluate the repaired code with *CodeLinter* for the next repair iteration. We use *HapRepair* to conduct a 5-iteration repair process. We evaluate the number of remaining defects and repair rate of each project using *HapRepair* at each iteration. Furthermore, we also evaluate the number of remaining defects and repair rate of different types of performance defects using *HapRepair* at each iteration. The results are shown in Table 1 and Table 2 respectively.

Table 1 shows that the number of remaining defects reduces for each project and achieves a high overall repair rate, which is about 99%. For example, the project wifi_testapp starts with the largest number of defects (1116) and is reduced to only 7 after five iterations. The repair rate for the project is about 99%. For another example, the project A21_C completely eliminates its initial defects after the third iteration. The repair rate for the project is 100%. From the results, we can see that *HapRepair* is generally effective for different projects.

Prompt Template for Fixing Defects

You are a code defect repair expert in ArkTS. I will give you an example of the same type of defect of the code and the defect code surrounding context, please help me repair.
Defect Type:
`hp-arkui-set-cache-count-for-lazyforeach-grid`
Description:

```
It is recommended to set a reasonable
cacheCount when using LazyForEach under
the Grid.
```

Demo 1:
Problem Code:

```
LazyForEach(this.data, (item:number) => {
GridItem() } .margin(10).height(500))
```

Fix Explanation:

```
Should set a reasonable cacheCount when
using LazyForEach
```

Repaired Code:

```
LazyForEach(this.data,
(item:number) => { GridItem() }
.margin(10).height(500)).cacheCount(2)
```

Following is the difflib results of repairing buggy code into fixed code:
`[Changes]`

Here's the surrounding context to be fixed:
`[Surrounding Context]`

Your output should follow the following json format:

```
[
    {
        "type": "modify | add |delete",
        "line": [line number],
        "code": "[line code]"
    }
]
```

Figure 3: Example of Fixing an ArkTS's Defects

Table 2 shows the result of different types of performance defects. There are totally 24 different types of performance defects of the projects in the test dataset, which indicates that the types are various. *HapRepair* successfully reduces the number of remaining defects and achieves high repair rates for 21 types of performance defects. The results show that *HapRepair* is generally effective for different types of performance defects.

From Table 1 and Table 2, we can also see that as the iteration progresses, the number of remaining defects gradually decreases and the repair rate increases. After five iterations, most of the defects have been successfully repaired. This result shows that the iterative strategy of *HapRepair* makes sense.

Answer for RQ1: *HapRepair* is capable of achieving overall high performance on automated defect repair in real-world scenarios.

5.3 RQ2: Ablation Study

In this RQ, we analyze the factors that can affect the performance of *HapRepair*. There are several factors that will affect the performance of *HapRepair*, including the number of similar defect repair examples retrieved for in-context learning during RAG, the context used as the retrieval query and prompt, and the changes of repaired code. For each factor, we conduct an ablation study to evaluate the influence. Take the cost and time of using the LLM into consideration, we randomly select several projects from the test dataset and set the iteration of *HapRepair* to 1.

Table 3: Number of Remaining Defects and Repair Rate of Different Number of Similar Defect Repair Examples

Project	Original	Without RAG	Top-1	Top-3	Top-5
Photos	58	33 (43%)	**14 (76%)**	23 (60%)	24 (59%)
HealthyDietIX	28	**12 (43%)**	16 (57%)	18 (64%)	18 (64%)
Arkts_tutorial	20	11 (45%)	**1 (95%)**	5 (75%)	1 (95%)
Component Collection	223	179 (20%)	**18 (92%)**	24 (89%)	21 (79%)
StageModelAbilityDevelop	66	17 (74%)	17 (74%)	17 (74%)	17 (74%)
A21_C	20	15 (25%)	**1 (95%)**	3 (85%)	6 (70%)
AdaptiveCapability	30	20 (33%)	**7 (77%)**	11 (63%)	10 (67%)
Gallery	301	184 (39%)	**29 (90%)**	30 (90%)	41 (86%)
Total	**746**	471 (37%)	**103 (86%)**	131 (82%)	138 (81%)

We design four settings, which are without RAG (zero example), top-1 example, top-3 examples and top-5 examples, to evaluate the influence of the number of similar defect repair examples retrieved for in-context learning during RAG. The result is shown in Table 3. We can see that *HapRepair* achieves a significantly lower number of remaining defects and a higher repair rate with similar defect repair examples than without similar defect repair examples. This indicates that the LLM can utilize the repair experience provided by similar defect repair retrieved through RAG, which gains significant improvement. Regarding the number of retrieved examples, we observe that the difference among using Top-1 example, Top-3 examples, and Top-5 examples is not substantial. Using the Top-1 example performs best in most projects. This indicates that additional repair examples do not necessarily improve repair performance and may introduce noise. However, it is worth noting that HealthyDietIX without RAG performs better. This anomaly occurs because the LLM tends to mistakenly insert the 'key-generator' in the wrong position when handling the 'foreach-args-check' defect, which results in cascading errors that prevent other defects in the file from being repaired. Consider the cost and time of using the LLM, Top-1 example seems to be the best choice.

Besides, the context used for retrieval is an important factor that will affect the retrieval results. Also, the context used in the prompt will directly affect the generation of the LLM. Thus, we design two types of context. One is the surrounding context used in *HapRepair* and the other one is the entire code of the defect. The result is shown in Table 4. We can see that using the surrounding context generally achieves better performance than that of using the entire code as context. We make an analysis and find that directly

Table 4: Number of Remaining Defects and Repair Rate of Different Context Types for Retrieval and Prompt

Project	Original	Surrounding Context	Entire Code
Photos	58	**14 (76%)**	18 (69%)
HealthyDietIX	28	16 (43%)	**9 (68%)**
Arkts_tutorial	20	**1 (95%)**	10 (50%)
Component Collection	223	**18 (92%)**	45 (80%)
StageModelAbilityDevelop	66	17 (74%)	17 (74%)
A21_C	20	**1 (95%)**	4 (80%)
AdaptiveCapability	30	**7 (77%)**	8 (73%)
Gallery	301	**29 (90%)**	71 (76%)
Total	746	**103 (86%)**	182 (76%)

Table 5: Number of Remaining Defects and Repair Rate with-/without Changes of Repaired Code

Project	Original	Without Changes	With Changes
Photos	58	16 (72%)	**14 (76%)**
HealthyDietIX	28	**13 (54%)**	16 (43%)
Arkts_tutorial	20	1 (95%)	1 (95%)
Component Collection	223	44 (81%)	**18 (92%)**
StageModelAbilityDevelop	66	17 (74%)	17 (74%)
A21_C	20	10 (50%)	**1 (95%)**
AdaptiveCapability	30	12 (60%)	**7 (77%)**
Gallery	301	57 (81%)	**29 (90%)**
Total	746	170 (77%)	**103 (86%)**

using the entire code as a query for retrieval may get less relevant retrieval results. In addition, the entire code in the prompt may cause the LLM to lose focus on the defect-related segment, as the attention mechanism can be distracted by irrelevant portions of the code. By contrast, extracting and using surrounding context allows the LLM to focus specifically on defect-related information and address the limitation of the context window, resulting in better generation.

As we assume that the changes of the repaired code reflect the fine-grained operations, which will be good guidance for the LLM, we design an ablation study to evaluate the effectiveness of the changes of the repaired code. The result is shown in Table 5. The result shows that adding changes in the prompt can further reduce the number of remaining defects and increase the repair rate. This indicates that changes indeed guide the LLM to analyze the repair process to improve the performance.

Answer for RQ2: The ablation study reveals that both surrounding context, RAG examples, and repair changes, contribute to *HapRepair*'s performance. Specifically, using precise surrounding context, retrieving the top-1 similar defect example, and incorporating repair changes yield the highest repair rates, optimizing both accuracy and efficiency.

5.4 RQ3: Performance of different LLMs in *HapRepair*

The LLM used in our framework is one of the most important components. Therefore, we need to analyze how to choose the LLMs for repairing the new programming languages such as ArkTS. To compare the performance of various LLMs in *HapRepair*, we conduct evaluations using different LLMs including GPT-4o, Llama-3.2-70b-instruct, and Qwen-2.5-72b-instruct.

Table 6: Number of Remaining Defects and Repair Rate of *HapRepair* using Llama-3.2-70b-instruct

Project	Original	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
A21_C	20	3 (85%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
AdaptiveCapability	30	1 (97%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
ComponentCollection	223	41 (82%)	13 (94%)	7 (97%)	5 (98%)	2 (99%)
GSYVideo	15	5 (67%)	5 (67%)	5 (67%)	5 (67%)	0 (100%)
Gallery	289	37 (87%)	5 (98%)	2 (99%)	0 (100%)	0 (100%)
HealthyDietIX	27	4 (85%)	5 (81%)	4 (85%)	1 (96%)	0 (100%)
Photos	58	22 (62%)	16 (72%)	13 (78%)	8 (86%)	3 (95%)
StageModelAbilityDevelop	72	14 (81%)	14 (81%)	11 (85%)	0 (100%)	0 (100%)
amqplib	10	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
demo	20	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
node-pop3	57	39 (31%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
utilCode	15	4 (73%)	4 (73%)	0 (100%)	0 (100%)	0 (100%)
wifi_testapp	1116	396 (64%)	274 (75%)	144 (87%)	84 (92%)	33 (97%)
Total	**1952**	**566** (71%)	**336** (83%)	**186** (90%)	**103** (94%)	**38** (98%)

The results are presented in Table 1, Table 6, and Table 7. It is evident that GPT-4o, Llama-3.2-70b-instruct, and Qwen-2.5-72b-instruct demonstrate remarkable performance after the 5-iteration repair process. Although these LLMs are not pretrained on ArkTS data, their in-context learning capabilities enable them to effectively repair performance defects in OpenHarmony Apps by leveraging defect repair examples provided through the RAG framework. We can see that it is easy to migrate *HapRepair* to different LLMs and achieve remarkable performance.

Table 7: Number of Remaining Defects and Repair Rate of *HapRepair* using Qwen-2.5-72b-instruct

Project	Original	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
A21_C	20	2 (90%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
AdaptiveCapability	30	1 (97%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
ComponentCollection	223	16 (93%)	10 (96%)	0 (100%)	6 (97%)	6 (97%)
GSYVideo	15	7 (53%)	5 (67%)	5 (67%)	5 (67%)	5 (67%)
Gallery	289	29 (90%)	14 (95%)	0 (100%)	13 (95%)	1 (99%)
HealthyDietIX	27	3 (89%)	1 (96%)	1 (96%)	0 (100%)	0 (100%)
Photos	58	15 (74%)	10 (83%)	8 (86%)	7 (88%)	7 (88%)
StageModelAbilityDevelop	72	6 (92%)	1 (99%)	1 (99%)	0 (100%)	0 (100%)
amqplib	10	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
demo	20	0 (100%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
node-pop3	57	21 (63%)	13 (77%)	0 (100%)	0 (100%)	0 (100%)
utilCode	15	5 (67%)	0 (100%)	0 (100%)	0 (100%)	0 (100%)
wifi_testapp	1116	299 (73%)	185 (83%)	163 (85%)	142 (87%)	8 (99%)
Sum	**1952**	**404** (79%)	**239** (88%)	**178** (90%)	**173** (89%)	**27** (99%)

Answer for RQ3: For defect detection and repair of the ArkTS program, GPT-4o, Llama-3.2-70b-instruct, and Qwen-2.5-72b-instruct achieve remarkable performances.

6 Threats to Validity

Threats to internal validity. First, the defect pairs are constructed manually, so the quality of the defect pairs in the knowledge database may affect the performance. However, we construct these defect pairs according to the performance rules strictly, and the experiment results indicate the effectiveness of these defect repairs. Second, since there are various LLMs, the LLM used in *HapRepair* may not be the best choice. However, we conducted an experiment to evaluate the performance of three LLMs, and the results show that GPT-4o that we used performs best. In addition, if there is a better LLM, it is easy to use it in *HapRepair*.

Threats to external validity. First, *HapRepair* focuses on the types of performance defects, so it is unclear whether *HapRepair* can work well for other types of defects. However, as LLMs have shown a strong ability in repairing defects through context-learning, *HapRepair* can easily adapt to other types of defects after constructing the defects pairs dataset. Second, *HapRepair* is designed for ArkTS, so it is unclear whether *HapRepair* can work well for other programming languages. However, we think that *HapRepair* can be easily adapted to other types of defects and other programming languages.

7 Related Work

7.1 Traditional Approaches for Defect Detection and Repair

Static analysis tools play an important role in software defect detection by analyzing code without being executed. IccTA [29] is a tool that focuses on detecting inter-component communication vulnerabilities in Android applications. It uses static taint analysis to track data flows between components to help developers identify security flaws such as data leakage. FlowDroid [4] is another static analysis tool focusing on precise taint tracking in Android application. It provides lifecycle-aware analysis, improving the accuracy of vulnerability detection in complex application flows. In contrast, dynamic analysis focuses on examining program behavior during execution[6, 14, 51]. Tools such as Valgrind [44] are widely used for detecting runtime errors, memory leaks, and performance bottlenecks. By leveraging runtime data, dynamic analysis ensures comprehensive coverage of execution paths. Besides, Holmes [12] utilizes statistical debugging techniques to identify error-prone execution paths efficiently. Besides, traditional program repair approaches rely on manual intervention and rule-based mechanisms to generate patches [28, 40]. Tools like AutoFix [24] and Angelix [39] utilize program synthesis and fault localization techniques to generate repair patches. These tools reduce the manual effort involved in fixing defects by automating the patch generation process, which is especially valuable for addressing complex defects. Another widely used approach involves rule-based methods, which rely on predefined patterns or templates to detect and repair defects. For instance, Abstract Syntax Tree based matching and regular expressions are commonly used to identify syntactic and semantic issues in source code [33]. These methods are implemented in tools that enforce compliance with specific coding guidelines.

7.2 Learning-Based Approaches for Defect Detection and Repair

The advent of LLMs has reshaped software engineering, especially for the code domain such as code generation and code summarization [1, 31]. Benefiting from the strong capability of understanding both the syntax and semantics of code, LLMs can also finish some tasks like defect detection and repair [22, 46, 54, 56]. Recent learning-based approaches have created a significant impact on defect detection, which autonomously learns the patterns of defective codes in a large scale of code datasets. For example, Devign uses GNN [53] to understand code semantics to detect defects. It can represent code as a graph to identify the complex defects that static analysis tools miss. VulDeePecker [32] is another example of a learning-based defect detection approach. VulDeePecker relies on binary classification for detecting vulnerable parts of code by examining the syntactic and semantic structure, with which it identifies snippets of bug-prone code. It is able to perform well on common programming languages but fails when applied to new languages without extensive retraining. Both Devign and VulDeePecker illustrate their strengths in detecting defects. CodeX [10] is a general-purpose LLM that has demonstrated strong capabilities in the code domain. Its pretraining on extensive code datasets enables it to provide intelligent code suggestions and repairs for widely used programming languages such as Python, JavaScript, and Java. AutoCodeRover [57] is an agent that integrates LLMs with a context retrieval mechanism and debugging techniques such as spectrum-based fault localization. It can iteratively extract the relevant surrounding context based on AST representations to autonomously solve real-world GitHub issues. Vul-RAG[15] is another approach using RAG to enhance the performance of LLMs in detecting and repairing defects. It retrieves the relevant external knowledge from a vector database to generate more precise and informed repair suggestions.

8 Conclusion

In conclusion, we address the challenges of defect detection and repair in the low-resource programming languages, particularly for ArkTS. By integrating LLMs with static analysis tools and RAG, we propose an innovative approach *HapRepair* to automatically detect and repair defects. *HapRepair* leverages and integrates a static analysis tool *CodeLinter* for defect detection and utilizes RAG to retrieve specific knowledge with in-context learning for defect repair to address the challenge of the lack of data. *HapRepair* designs two analysis tools to extract and combine context to overcome the challenge of the limitation of the context window. Experiment results show that *HapRepair* can achieve a low number of remaining defects and a high repair rate. In the future, we will further evaluate the generalizability of *HapRepair* across other programming languages. Additionally, we plan to explore more complex defect scenarios, such as cross-file defects, which require an understanding of broader code contexts and dependencies. These efforts aim to extend the applicability of *HapRepair* and address challenges in repairing sophisticated defects in emerging and evolving programming environments.

Acknowledgments

This work is partially supported by the National Key Research and Development Program of China (No. 2024YFB4506300) and the Open Foundation of Yunnan Key Laboratory of Software Engineering under Grant No.2023SE102. Besides, this research was primarily conducted after Mingyi Zhou joined Beihang University. However, part of his involvement in this work began during his time at Monash University. We acknowledge his previous affiliation and appreciate the support received throughout the research process.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[3] Sultan Alneyadi, Elankayer Sithirasenan, and Vallipuram Muthukkumarasamy. 2016. A survey on data leakage prevention systems. *Journal of Network and Computer Applications* 62 (2016), 137–152.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269.

[5] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[6] Thoms Ball. 1999. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 216–234.

[7] Keith H Bennett and Václav T Rajlich. 2000. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 73–87.

[8] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

[9] Haonan Chen, Daihang Chen, Yizhuo Yang, Lingyun Xu, Liang Gao, Mingyi Zhou, Chunming Hu, and Li Li. 2025. ArkAnalyzer: The Static Analysis Framework for OpenHarmony. arXiv:2501.05798 [cs.SE] https://arxiv.org/abs/2501.05798

[10] 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).

[11] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[12] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. 2009. Holmes: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 34–44.

[13] Huawei Developer. 2024. CodeLinter Rule Set for HarmonyOS. https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/ide-codelinter-rule-V5. Accessed on: 2024-04-10.

[14] Ulrich Doraszelski and Ariel Pakes. 2007. A framework for applied dynamic analysis in IO. *Handbook of industrial organization* 3 (2007), 1887–1966.

[15] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. *arXiv preprint arXiv:2406.11147* (2024).

[16] ESLint. [n. d.]. ESLint. https://eslint.org/.

[17] Hugging Face. 2024. stella_en_1.5B_v5: A Large Language Model for English Text Generation. https://huggingface.co/transformers. Accessed: 2024-12-25.

[18] 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.

[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] OpenAtom Foundation. 2024. OpenHarmony: A Comprehensive Open Source Project for All-Scenario, Fully-Connected, and Intelligent Era. https://gitee.com/openharmony. Accessed on: 2024-04-10.

[21] Görkem Giray, Kwabena Ebo Bennin, Ömer Köksal, Önder Babur, and Bedir Tekinerdogan. 2023. On the use of deep learning in software defect prediction. *Journal of Systems and Software* 195 (2023), 111537.

[22] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 31.

[23] Open HarmonySIG. 2024. Openharmony sig organization and pr command support list. https://gitee.com/openharmony-sig. Accessed on: 2024-06-07.

[24] Shi-Yu Huang, Kuang-Chien Chen, and Kwang-Ting Cheng. 1999. AutoFix: A hybrid tool for automatic logic rectification. *IEEE transactions on computer-aided design of integrated circuits and systems* 18, 9 (1999), 1376–1384.

[25] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.

[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] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 802–811.

[28] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 3–13.

[29] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[30] Li Li, Xiang Gao, Hailong Sun, Chunming Hu, Xiaoyu Sun, Haoyu Wang, Haipeng Cai, Ting Su, Xiapu Luo, Tegawendé F Bissyandé, et al. 2023. Software engineering for openharmony: A research roadmap. *arXiv preprint arXiv:2311.01311* (2023).

[31] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[32] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[33] Changsong Liu, Yangyang Zhao, Yibiao Yang, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2015. An ast-based approach to classifying defects. In *2015 IEEE International Conference on Software Quality, Reliability and Security-Companion*. IEEE, 14–21.

[34] 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.

[35] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The scope of ChatGPT in software engineering: A thorough investigation. *arXiv preprint arXiv:2305.12138* (2023).

[36] Taslim Mahbub, Dana Dghaym, Aadhith Shankarnarayanan, Taufiq Syed, Salsabeel Shapsough, and Imran Zualkernan. 2024. Can GPT-4 aid in detecting ambiguities, inconsistencies, and incompleteness in requirements analysis? A comprehensive case study. *IEEE Access* (2024).

[37] Dung Nguyen Manh, Thang Phan Chau, Nam Le Hai, Thong T. Doan, Nam V. Nguyen, Quang Pham, and Nghi D. Q. Bui. 2024. CodeMMLU: A Multi-Task Benchmark for Assessing Code Understanding Capabilities of CodeLLMs. arXiv:2410.01999 [cs.SE] https://arxiv.org/abs/2410.01999

[38] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22 (2017), 1936–1964.

[39] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

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

[41] MTEB 2024. Overall MTEB English leaderboard. https://huggingface.co/spaces/mteb/leaderboard. Accessed on: 2024-04-10.

[42] María Isabel Murillo and Marcelo Jenkins. 2021. Technical Debt Measurement during Software Development using Sonarqube: Literature Review and a Case Study. In *2021 IEEE V Jornadas Costarricenses de Investigacion en Computación e Informática (JoCICI)*. IEEE, 1–6.

[43] John D Musa. 2004. *Software reliability engineering: more reliable software faster and cheaper*. AuthorHouse.

[44] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[45] C Lakshmi Prabha and N Shivakumar. 2020. Software defect prediction using machine learning techniques. In *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*. IEEE, 728–733.

[46] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[47] Julian Aron Prenner and Romain Robbes. 2024. Out of context: How important is local context in neural program repair?. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[48] OpenHarmony Third Party Components SIG. 2024. Openharmony-tpc: Third party components repository for openharmony applications. https://gitee.com/openharmony-tpc. Accessed on: 2024-04-10.

[49] SonarQube. [n. d.]. SonarQube. https://www.sonarqube.org/.

[50] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[51] Dimitrios Vamvatsikos and C Allin Cornell. 2002. Incremental dynamic analysis. *Earthquake engineering & structural dynamics* 31, 3 (2002), 491–514.

[52] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720* (2019).

[53] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[54] 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.

[55] Aidan ZH Yang, Sophia Kolak, Vincent J Hellendoorn, Ruben Martins, and Claire Le Goues. 2024. Revisiting unnaturalness for automated program repair in the era of large language models. *arXiv preprint arXiv:2404.15236* (2024).

[56] Bing Yu, Hua Qi, Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Lei Ma, and Jianjun Zhao. 2021. Deeprepair: Style-guided repairing for deep neural networks in the real-world operational environment. *IEEE Transactions on Reliability* 71, 4 (2021), 1401–1416.

[57] 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.