Error Delayed Is Not Error Handled: Understanding and Fixing Propagated Error-Handling Bugs

HAORAN LIU*, National University of Defense Technology, China
SHANSHAN LI*, National University of Defense Technology, China
ZHOUYANG JIA, National University of Defense Technology, China
YUANLIANG ZHANG, National University of Defense Technology, China
LINXIAO BAI, National University of Defense Technology, China
SI ZHENG, National University of Defense Technology, China
XIAOGUANG MAO, National University of Defense Technology, China
XIANGKE LIAO, National University of Defense Technology, China

Error handling is critical for software reliability. In software systems, error handling may be delayed to other functions. Such propagated error handling (PEH) could easily be missed and lead to bugs. Our research reveals that PEH bugs are prevalent in software systems and, on average, take 44.1 days to fully address. Existing approaches have primarily focused on the error-handling bug within individual functions, which makes it difficult to fully address PEH bugs.

In this paper, we conducted the first in-depth study on PEH bugs in 11 mature software systems, examining how errors propagate and how they should be handled. We introduce EH-Fixer, an LLM-based tool for automated program repair specifically designed to address PEH bugs. For each PEH bug, EH-Fixer constructs its propagation path, and repairs them through retrieval-augmented generation. To assess the performance of our approach, we collected 89 historical PEH bugs from the Linux Kernel as well as 9 widely used applications. The experimental results show that EH-Fixer can fix 83.1% (74/89) of PEH bugs.

CCS Concepts: • **Software and its engineering** → *Error handling and recovery*.

Additional Key Words and Phrases: Error-Handling Bug, Automatic Program Repair

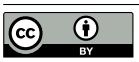
ACM Reference Format:
Haoran Liu, Shanshan Li, Zhouyang Jia, Yuanliang Zhang, Linxiao Bai, Si Zheng, Xiaoguang Mao, and Xiangke Liao. 2025. Error Delayed Is Not Error Handled: Understanding and Fixing Propagated Error-Handling Bugs. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE114 (July 2025), 24 pages. https://doi.org/10.1145/3729384

1 Introduction

Software systems frequently encounter errors, necessitating appropriate handling to maintain stability. A prevalent issue impacting software reliability is the presence of error-handling bugs, which are failures to properly handling specific errors.

*Co-first author.

Authors' Contact Information: Haoran Liu, National University of Defense Technology, Changsha, China, liuhaoran@nudt.edu.cn; Shanshan Li, National University of Defense Technology, Changsha, China, shanshanli@nudt.edu.cn; Zhouyang Jia, National University of Defense Technology, Changsha, China, jiazhouyang@nudt.edu.cn; Yuanliang Zhang, National University of Defense Technology, Changsha, China, zhangyuanliang13@nudt.edu.cn; Linxiao Bai, National University of Defense Technology, Changsha, China, linxiao_b@nudt.edu.cn; Si Zheng, National University of Defense Technology, Changsha, China, zhengsi@qiyuanlab.com; Xiaoguang Mao, National University of Defense Technology, Changsha, China, xgmao@nudt.edu.cn; Xiangke Liao, National University of Defense Technology, Changsha, China, xkliao@nudt.edu.cn.



ACM 2994-970X/2025/7-ARTFSE114
https://doi.org/10.1145/3729384

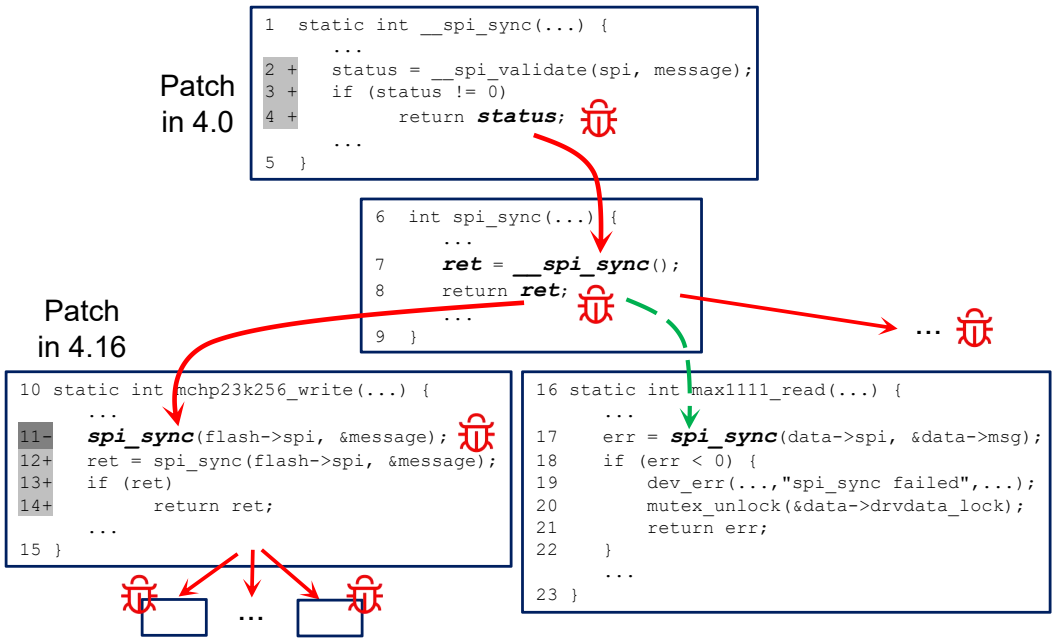


Fig. 1. Limitations of existing approaches.

Existing approaches that fix error-handling bugs usually rely on manually created templates. For instance, ErrDoc [55] identified four categories of repair methods and summarized corresponding templates. Such methods require developers to possess extensive domain knowledge, struggle to adapt to the software evolution, and tend to be inaccurate. With the rapid development of large language models (LLMs), there is growing interest in utilizing LLMs for automatic program repair (APR). For example, ChatRepair [63] guided the APR by feeding the test results back to LLMs, and RAP-Gen [56] fine-tuned LLMs for APR by retrieving similar patches. However, such efforts are limited in two aspects.

Firstly, error-handling bugs can affect multiple functions within a software system. We illustrate this with an example from the Linux Kernel in Fig. 1. Developers fixed an error-handling bug in version 4.0 (lines 2-4) and returned a variable "status" (line 4). This variable represents an abnormal state of the software system, referred to as an ***error***. This error was propagated to other functions along the data/control flow (lines 4, 7, 8, 11, 17). Such a propagation path is referred to as the ***error propagation path***. The error reaches function *mchp23k256_write* and causes a new error-handling bug in line 11, and was not fixed until version 4.16. Error-handling bugs in line 2 and 11 are caused by the propagation of the same error, and we refer to the collection of such bugs as a **propagated error-handling (PEH) bug**. Existing approaches usually focus on single-hunk fixes[55], are inadequate for addressing PEH bugs.

Secondly, error-handling strategies vary across different software systems and even among different functions. For instance, the error in *mchp23k256_write* only needs to be passed to its caller (line 14), whereas the error in *max1111_read* necessitates additional logging (line 19) and cleanup (line 20). Choosing the appropriate handling requires an understanding of how the software and each function handles errors. Existing template-based approaches often replicate near-by error-handling code snippets [55]. They overlook the diversity of handling strategies, and could result in inappropriate repairs.

To better understand the PEH bug, we conducted studies on 153 (44+109) PEH bugs from the Linux Kernel and 10 applications. Our findings indicate that PEH bugs are prevalent in software systems and, on average, take 44.1 days to fully address. To tackle PEH bugs, we proposed an LLM-based automated program repair (APR) approach EH-Fixer. Our key insight is that resolving error-handling bugs requires a step-by-step fix along the propagation path, not just within a single function. Accordingly, EH-Fixer constructs the error propagation path for the PEH bug, identifies functions necessitating repair, and generate patches. The design of EH-Fixer presents two primary challenges:

- Firstly, constructing the error propagation path is challenging. To address this, we analyzed 153 PEH bugs, studying their propagation characteristics to inform our construction of the propagation path.
- Secondly, fixing functions on the propagation path is not easy. We analyzed 2,550 functions in 153 PEH bugs, identifying contextual information that aids LLMs in selecting the appropriate handling. This information guides LLMs using a retrieval-augmented technique.

We evaluated the performance of EH-Fixer in fixing real-world PEH bugs. We used EH-Fixer to repair 9 new error-handling bugs in the Linux Kernel. Two have been confirmed, while the rest are still under review. Furthermore, we collected 89 (44+45) historical PEH bugs from the Linux Kernel and 9 highly-rated applications on GitHub, including 233 functions necessitating repair. To mitigate data leakage, we partitioned the dataset based on the training cut-off time of LLMs [47]. Bugs fixed post-cut-off are free from data leakage since their patches were not included in the training data of LLMs. For earlier bugs, we employed source code transformation methods maintaining semantic equivalence [69]. Our experiments revealed that EH-Fixer successfully resolved 83.1% (74/89) of the PEH bugs within five attempts, and notably, 48.6% (36/74) of these PEH bugs cannot be fixed by all comparative approaches.

The key contributions of this paper include:

- We conducted a study on 153 historical PEH bugs from the Linux Kernel and 10 applications. These findings contribute to a deeper understanding of PEH bugs, reveal limitations of existing approaches, and inform the design of our proposed method.
- We proposed an APR approach EH-Fixer. By using LLMs and a retrieval-augmented technique, EH-Fixer is able to fix PEH bugs.
- We constructed a dataset containing 89 historical PEH bugs, and compared the performance of EH-Fixer against 3 state-of-the-art (SOTA) approaches. Experiment results show that EH-Fixer outperforms SOTA approaches by fixing an additional 48.6% (36/74) PEH bugs.

2 Understanding Propagated Error-Handling Bug

In this section, we take an in-depth look into the PEH bug through an empirical study. We will first outline the methodology used in this study, then present our findings. Our findings involve the prevalence and resolution time of PEH bugs, and characteristics of PEH bugs concerning their propagation and handling.

2.1 Study Methodology

2.1.1 Studied Subjects. As shown in Table 1, we studied 11 software systems from different domains. We select these projects because they are: a) mature and widely used, each having at least 500 GitHub stars; b) open-source and have well-maintained evolution histories. These criteria ensure the accuracy and generality of our findings.

2.1.2 Data Collection. An error may propagate through multiple functions, and some functions may not handle the error properly, resulting in a PEH bug. To better understand PEH bugs, we analyze

Table 1. Studied subjects.

Domain	Name	Line Number	PEH Bugs
Operating System	Linux Kernel	36,780,452	44
Database	TimescaleDB	659,465	23
	MonetDB	452,349	14
Image Editor	Darktable	664,703	14
FTP	ProFTPD	845,882	13
Machine Learning	H2O	918,805	11
Security	Hashcat	1,790,923	11
	MASSCAN	59,803	3
Data Transfer	Zstandard	157,915	9
Web server	Lighttpd	132,521	7
Player	Audacious	49,795	4

existing errors in the software, collect functions on their propagation paths, and identify PEH bugs accordingly. First, we use the method of an existing approach [31] to identify error-handling code snippets. We then manually analyze the error propagation path based on data dependencies, and document the functions involved. A PEH bug is identified if functions along the propagation path fails to appropriately handle the error. For example, in Fig. 1, we first identify error-handling code snippets in $\_\_spi\_sync$, $mchp23k256\_write$, and $max1111\_read$. We then construct the error propagation path based on data/control dependencies and collect functions including spi_sync. Finally, we searched the commit history and found that the error in $\_\_spi\_sync$ was handled at version 4.0, whereas the error in $mchp23k256\_write$ was not handled until version 4.16, so they were considered as a PEH bug.

2.2 Resolution Time of PEH bugs

We investigated 876 error-handling code snippets to study the prevalence of PEH bugs and their resolution time, and found:

> **Finding 1**: A significant 41.9% (367/876) of errors propagate to multiple functions. Errors affecting 17.5% (153/876) of functions persist across multiple software versions (classified as PEH bugs). On average, these errors propagate through 16.7 functions and require 44.1 days for complete resolution.

Taking Fig. 1 as an example, the error in function $\_\_spi\_sync$ propagated through function return values to multiple functions, and some functions such as $mchp23k256\_write$ fail to handle this error properly. Moreover, the fix in $mchp23k256\_write$ also propagated the error to its callers, and may result in more functions necessitating repair.

Implication: PEH bugs are prevalent in software systems and affect multiple functions. Resolving them usually takes a significant amount of time, posing a threat to software reliability.

2.3 Characteristics of the error propagation

This study examines how errors propagate across functions, and how such propagation causes PEH bugs. We investigated 2,550 functions in the error propagation path of 153 PEH bugs to study how they propagate the error. We found that:

Finding 2: 44.9% (1,145/2,550) of the functions propagate errors. Of these, 85.7% (981/1,145) propagate errors via their return values, 12.3% (141/1,145) by altering parameters, and 2.3% (26/1,145) by modifying global variables.

On the one hand, this finding suggests that functions mainly propagate the error through its return values or parameters, and the error propagation path can be identified accordingly. On the other hand, this finding suggests that, on the error propagation path, functions in leaf nodes need to be repaired according to their parent nodes. For example, as depicted in Fig. 1, callers of $mchp23k256\_write$ may need to be repaired because the patch in $mchp23k256\_write$ returns the error (line 14).

Further investigation into these 1,145 functions revealed the rationale behind their error propagation:

Finding 3: 66.6%(763/1,145) chose to propagate errors because their errors could affect subsequent program execution, while the remaining 33.4%(382/1,145) did so because the software uniformly handles certain types of errors.

We classify errors that fall into both categories as uniform handled, as such errors are propagated to specific code snippets, making them easy to identify. This finding indicates that the decision to propagate an error depends on the potential impact of the error on subsequent program execution and the handling strategy of the software. We define the term ***error impact*** as the influence of an error on subsequent program execution. This impact is twofold: it affects both the function in which the error occurs and any functions that call this function. For instance, as illustrated in Fig. 1, the role of the $max1111\_read$ function is to read, process, and return data. The error at line 17 not only halts the execution of $max1111\_read$ due to a failure in data input, but also prevents the calling functions from receiving the processed data, necessitating the return of an error code.

Implication: The propagation path of an error can be traced through return values and parameters, while the fixing of error-handling bugs requires an understanding of the error impact and the handling strategy of the current software.

2.4 Characteristics of the error-handling action

We term the individual statements within error-handling code snippets as ***error-handling action***, and refer to the combination of these actions as the ***action set***. The correlation between the error impact and the action set is termed the ***error-handling strategy***. This strategy exhibits considerable diversity across different software systems and even within individual functions. For instance, in Fig. 1, the error-handling action set in $mchp23k256\_write$ differs from that in $max1111\_read$.

We first studied 2,151 handling actions in error-handling code snippets from these 1,145 functions, and found that:

Finding 4: The studied error-handling code snippets include 704 distinct actions, categorized as follows: resource cleanup 56.1% (395/704), returning 37.9% (267/704), and logging 6.0% (42/704).

Cleanup actions are statements like exit, free, close, delete, and unlock, while return actions include return, goto, break, and continue. The variety of these actions underscores the complexity of

selecting an appropriate action set to handle the error. For instance, to generate the error-handling code snippet in *max*1111_*read* of Fig. 1, it is critical to acknowledge that: 1) The error propagated from *spi_sync* is severe for *max*1111_*read*, and handling it requires ending *max*1111_*read* and return the error; 2) *data→drvdata_lock* utilized by *mutex_lock* can be cleaned up using *mutex_unlock*; 3) *dev_err* can log the error; 4) "*err*" can be used as the return value. Note that, It is common for error-handling code snippets to rely on specific functions for logging, resource cleanup, and returning standardized error codes (e.g., "*log_err*" for logging, "close" for files, "ENOMEM" for return values). These functions and return values are commonly repeated across multiple code snippets, resulting in 704 distinct actions out of 2,151 handling actions.

We further investigated these error-handling code snippets to determine the contexts that could inform the selection of handling action sets:

> **Finding 5**: Almost all log actions and return actions repeatedly occur within the error propagation path or in functions from the same file, and 96.0% (458/477) of cleanup actions correspond to specific resources.

Functions within the same module usually use the same logging function. This usage allows for inferences about log actions, such as the *dev_err* in Fig. 1. Moreover, return values used in functions of the same error propagation path are similar. For instance, the return action in *mchp*23*k*256_*write* (line 14) can be inferred from the handling in *spi_sync* (line 8). However, 35.4% (169/477) of the cleanup actions remain unidentified in these contexts (functions from the same propagation path or the same file). Through our analysis, we discovered that 96.0% (458/477) of the cleanup actions are associated with specific resources. For example, the cleanup actions for *data→drvdata_lock* in *max*1111_*read* can be informed by observing how similar resources with the same data type as *data→drvdata_lock* are managed in other error-handling code snippets.

Implication: Choosing the appropriate action set necessitates specific contextual information, including the error-handling strategies of the current software system, available handling actions, and resource usage. Existing error-handling code snippets in the software can guide this selection.

3 EH-Fixer Design

This section outlines the design of EH-Fixer, an APR approach employing LLMs to repair PEH bugs automatically. Error handling in C/C++ is usually manually implemented by developers, so it is more flexible and error-prone [49, 51]. Therefore, this paper focuses on error-handling bugs in C/C++ programs.

The design of EH-Fixer is illustrated in Fig. 2. EH-Fixer comprises three main components: the EH Database, the Repair Agent, and the Validate Agent. The EH Database aids LLMs in understanding and fixing errors using retrieval-augmented technology. The Repair Agent identifies the error propagation path of an PEH bug, and generates patches accordingly, whereas the Validate Agent scrutinizes and guides the refinement of these patches.

This design faces two primary technical challenges. First, although our study in Sec. 2 has significantly reduced the input data, the remaining volume can still impair the performance of LLMs and increase computational costs. Second, tracing error propagation across multiple functions step-by-step is inherently time-consuming and resource-intensive. To address the first challenge, we developed the EH Database, which features summarization methods designed specifically for different types of contextual information, effectively shortening the input length. For the second challenge, we address this by employing clustering techniques that allow for the collective repair

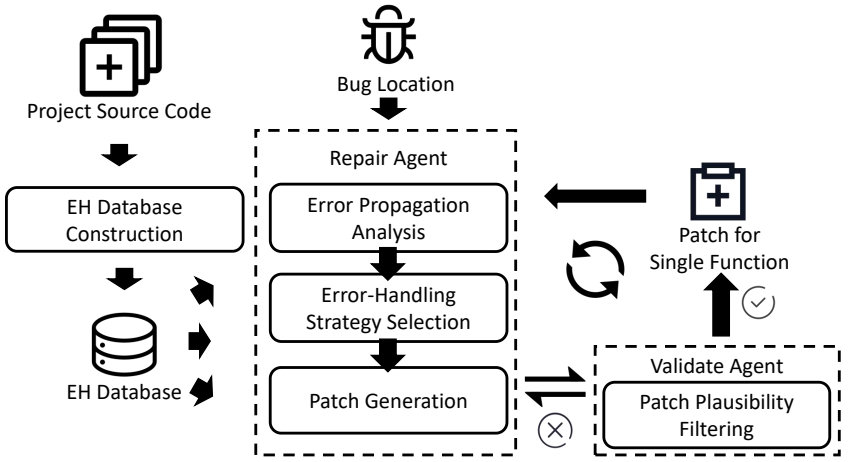


Fig. 2. Overview of EH-Fixer.

of functions with similar contextual information during a single LLM interaction, thus decreasing both interaction frequency and token usage.

Note that, EH-Fixer operates based on static analysis in conjunction with LLMs, allowing it to analyze software without compilation. This feature enables EH-Fixer to be used during software development.

3.1 EH Database

As outlined in Sec. 2, fixing PEH bugs necessitates tracing the error propagation path, comprehending its impact, and choosing the appropriate action set that aligns with the handling strategy of the target software. To this end, the dataset must encapsulate three types of contextual information from the target software: function semantics and dependencies, available handling actions, and illustrative examples that mirror the error-handling strategy used in the software. The primary technical challenge lies in representing this information with clarity and precision. EH-Fixer tackles this challenge by conducting an inter-procedural static analysis[1] and applying distinct extraction methods for each type of information. We depict an example of the three types of information in the EH Database in Fig. 3.

3.1.1 Semantics and Dependencies. In the context of function semantics and dependencies, the Repair Agent primarily utilizes this information to aid LLMs in tracing error propagation paths, identifying origins, and analyzing impacts. The EH Dataset initially stores the source code for each function. Upon retrieval for analysis, EH-Fixer applies the Chain of Thought (COT) methodology during the analysis, and will first generates a natural language summary detailing inputs, outputs, and logic (data 1 in Fig. 3). This summary is stored in the EH Dataset and is directly provided when the function is subsequently retrieved. Furthermore, EH-Fixer constructs function call graphs and data/control dependency graphs via inter-procedural static analysis (data 2 in Fig. 3). These graphs aid in the tracing of error propagation and program slicing on code snippets related to the error.

3.1.2 Available Action. Turning to the second category of information, LLMs should be aware of the available handling actions in the current software to make informed decisions. Logging actions and returning actions, which typically can be identified by specific keywords [19, 25, 31, 50], are less

[1]The specific details and examples of the static analysis can be found in link. [2]

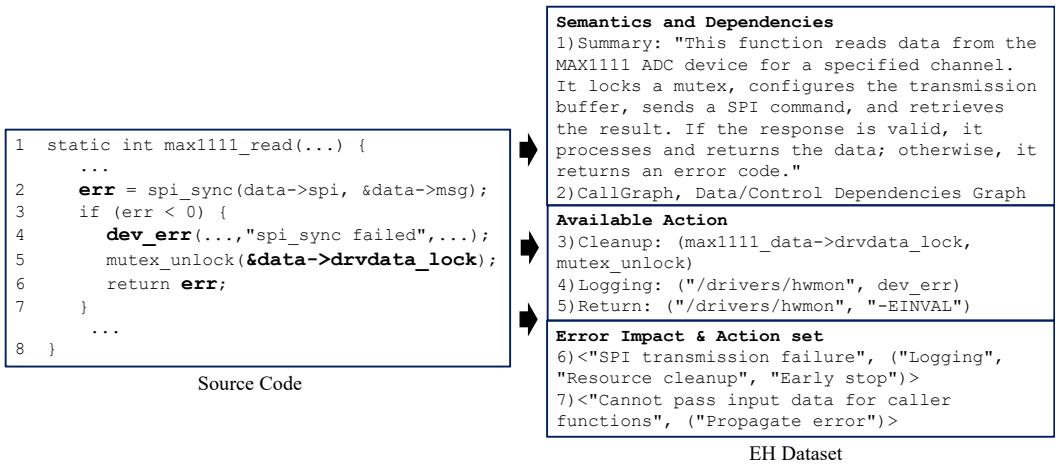


Fig. 3. Construction of EH Database.

diverse. Conversely, cleanup functions show more diversity and are scattered across error-handling code snippets, posing a challenge for direct representation to LLMs. EH-Fixer systematically extracts these actions from existing error-handling code snippets in three steps. Initially, EH-Fixer utilizes the method outlined in Sec. 2.1.2 to collect existing error-handling code snippets, and document their return values. Subsequently, EH-Fixer categorizes functions containing keywords such as "log" or "err" in hard-coded strings as logging actions, while it considers other function invocations as cleanup actions. The impact of such a simple classification method on the performance of EH-Fixer will be discussed in Sec. 4.1.2. Finally, EH-Fixer establishes relationships between cleanup actions and specific resource types based on data/control dependencies. For example, in Fig. 3, EH-Fixer first identifies the error-handling code snippet in line 3-6, and document its return value in line 6. Since it returns the return value of *spi_sync* (line 2, 6), EH-Fixer retrieves the return value of function *spi_sync*, and stores it with the file path of *maxllll_read* (data 5). Similarly, EH-Fixer identifies the logging function *dev_err* and correlates it with the same file path (data 4). Finally, we record the *mutex_unlock* with the datatype of its input variable *data→drvdata_lock* (data 3). For each cleanup function, EH-Fixer archives up to TH_e code snippets as examples to help LLMs understand its applications. The threshold TH_e will be discussed in Sec. 4.3. When retrieving available actions, EH-Fixer may identify multiple actions of the same type, such as several logging functions. All these actions will be returned and provided to the LLM.

3.1.3 Error-Handling Strategy. Regarding the third category of information, in line with Finding 2 and 3, selecting the appropriate handling action set necessitates comprehension of both the error impact, and the handling strategy of the current software. EH-Fixer analyzes existing error-handling code snippets to study the relationships between the error impact and the associated handling action set. These relationships serve as demonstrative examples of the handling strategy. Referring to Finding 5, functions within the same error propagation path or the same file usually have a similar action set. Consequently, EH-Fixer learns the handling strategy of the target software from these functions. To achieve this, we utilize an LLM to analyze existing error-handling code snippets and summarize data pairs of <Error Impact, Action Set>. These pairs aid LLMs in grasping the handling strategies through the In Context Learning (ICL) technique. Specifically, EH-Fixer constructs these data pairs in 3 steps. Initially, for each existing error-handling code snippet, EH-Fixer establishes

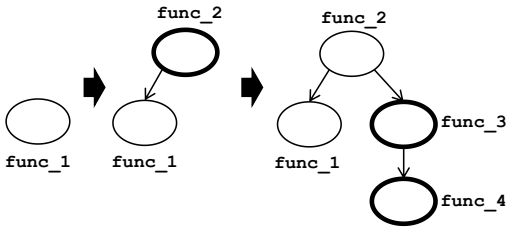


Fig. 4. Error propagation path construction.

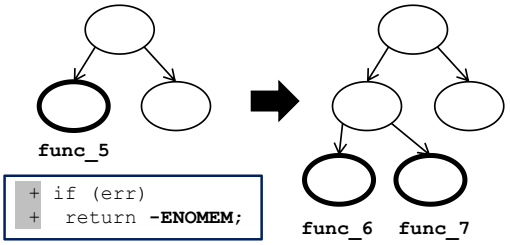


Fig. 5. Error propagation path extension.

the propagation path for each error-handling code snippet as described in Sec. 3.2.1. Following Finding 5, EH-Fixer then retrieves relevant contextual information, and summarizes a natural language describing the error impact using LLMs. Finally, EH-Fixer utilizes LLMs to analyze and pair error impacts with handling action set, outputting as <Error Impact, Action Set>. For instance, as shown in Fig. 3, the impact on the buggy function (*maxllll_read*) necessitates logging and an early termination (data 6). A failure in *maxllll_read* could hinder its caller from obtaining a valid device, thus necessitating error propagation (data 7). To minimize unnecessary overhead, the EH-Fixer only learns these data pairs when an error-handling code snippet is retrieved.

The EH-Database is designed to store the error-handling strategies of target software, and needs to be rebuilt for different software. The database is constructed using automated static analysis, with interaction with the LLM occurring only when relevant code snippets are retrieved. As a result, the rebuilding process is simple and fast. For example, the EH-Database for software containing 250,000 lines of code can be built in under 5 minutes on an i9-10900K CPU.

3.2 Repair Agent

The Repair Agent, powered by LLMs, undertakes two primary tasks: analyzing error propagation paths and generating patches. This section outlines the technical challenges associated with these tasks and describes our solutions.

3.2.1 Error Propagation Path Analysis. The Repair Agent analyzes the error propagation path through the call graph. The main technical challenge is that functions in the error propagation path need to be fixed according to their parent nodes. For instance, as illustrated in Fig. 1, callers of *mchp23k256_write* necessitate repairs only if *mchp23k256_write* itself has been successfully repaired and returns an error code. To address this challenge, the Repair Agent first constructs the error propagation path, and then repairs it step-by-step from the root node.

When a bug location is provided to the Repair Agent, it may not necessarily be the root node of the error propagation path. The Repair Agent constructs the propagation path based on two criteria: 1) A node that does not handle return values or parameters of custom functions is identified as the root node. 2) Nodes that neither return values nor alter their parameters are considered leaf nodes. As illustrated in Fig. 4, with $func\_1$ as the input bug location, the Repair Agent first identifies the root node $func\_2$, and subsequently locates $func\_3$ and $func\_4$. After the above steps, the Repair Agent analyzes the nodes along the error propagation path and generates corresponding patches. If these patches cause further error propagation, the Repair Agent will expand the propagation path. Using Fig. 5 as an example, since the patch for $func\_5$ returns "-ENOMEM", causing the error to propagate to its callers. As a result, the Repair Agent includes $func\_6$ and $func\_7$ to the propagation path. The Repair Agent does not consider error propagation involving global variables. This is a trade-off between accuracy and recall, which is discussed in Sec. 5.3.

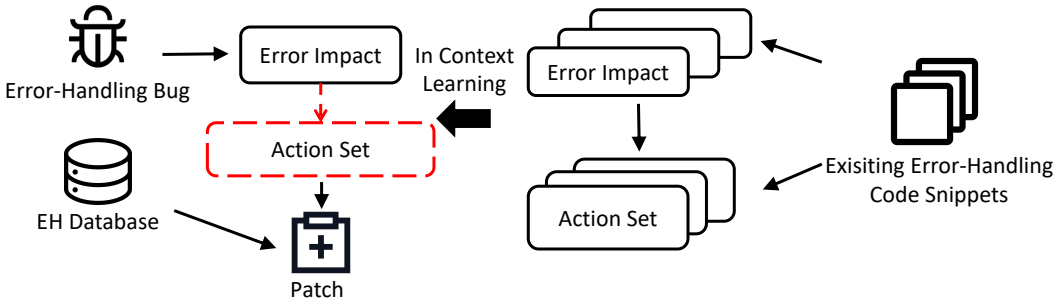


Fig. 6. Patch generation.

3.2.2 Patch generation. For each function in the propagation path, the Repair Agent first evaluates the error impact, then predicts a set of handling actions considering the error-handling strategy of the current software, and eventually retrieves available actions from the EH Database to generate patches. As shown in Fig. 6, the main idea of this step is to derive <Error Impact, Action Set> relationship pairs by learning relevant error-handling code snippets, and guide LLMs to predict the appropriate action set based on the error impact through ICL. This process presents two technical challenges. Firstly, PEH bugs effect multiple functions. Our experiments indicate that repairing all affected functions in a single LLM session is difficult, and individual analysis of functions leads to increased overhead. Secondly, the extensive retrieval of contextual information, particularly regarding cleanup actions, results in significant unnecessary costs.

To tackle the first challenge, we conduct studies on 96 PEH bugs discussed in Sec. 2. We find that many functions along the propagation path have similar contexts. Independently repairing these functions entails redundant context provision to LLMs, thus unnecessarily increasing computational overhead. To streamline this process, we grouped functions by context similarity and implemented collective repairs within a single session. Specifically, as discussed in Sec. 3.1, LLMs necessitate three contextual inputs for repairs: summaries of relevant functions, code snippets having data/control dependencies with the error, and data pairs describing the error-handling strategy of the target software. We use the Jaccard coefficient [52] to quantify the contextual similarity between two functions. Defined by the formula $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$, this metric calculates the proportion of shared to total unique contexts, offering a concise quantification of contextual overlap between functions. Utilizing the Agglomerative Hierarchical Clustering (AHC) technique [24], we clustered the functions with a distance threshold (TH_d) set at 0.4. Functions that are in the same cluster are fixed collectively. The rationale behind the chosen value of TH_d is discussed in Sec. 4.4.

Regarding the second challenge, the Repair Agent instructs LLMs to provide feedback on the resources that require cleanup, and supplies only the relevant cleanup functions. Furthermore, the Repair Agent simplifies the action set to a combination of three types of handling actions. For example, code snippets containing only the logging action are considered to have the same action set. For each type of action set, the Repair Agent will only retrieve up to TH_e code snippets from the EH Database. Consequently, the total number of input code snippets will not exceed $TH_e * 7$ (calculated as $C(3,1) + C(3,2) + C(3,3)$). The parameter setting is discussed in Sec. 4.4. These approaches can reduce a lot of unnecessary inputs and can effectively reduce the overhead.

The prompt of Repair Agent comprises 4 components, including **Instruction**, **Contextual Information**, **Constraint**, and **Example**. The Instruction guides LLMs to analysis the error impact, predict the appropriate handling action set, and generate patches. The Example contains data pairs for In Context Learning (ICL) (discussed in Sec. 3.1.3). Contextual Information retains

semantics and dependencies related to the error (discussed in Sec. 3.1.1). Constraint restricts LLMs from outputting extraneous information and specifies the format of the output. If additional information is required during patch generation, EH-Fixer retrieves and supplies the relevant information to LLMs (discussed in Sec. 3.1.2). Throughout this process, LLMs receive only the source code of the buggy function and the predicted set of handling actions as historical data.

Algorithm 1 Generate patches for a PEH Bug

Require: Bug location *BugLocation*
Ensure: Generate patches for PEH bugs
1: Initialize *RepairList* and *PatchList*
2: *RepairList* = *get_error_propagation_path*(*BugLocation*)
3: **while** not *RepairList*.isEmpty() **do**
4: Let *ContextInfo* = EHDataset(RepairList)
5: Let *RepairCluster* = RepairList.cluster().*sort_and_pop*()
6: Let *InfoRetrieval*, *Patch*, *NewRepairList* = RepairAgent(*form_prompt*(*RepairCluster*, *ContextInfo*))
7: **while** not *InfoRetrieval*.isEmpty() **do**
8: Let *AdditionalInfo* = EHDataset(*InfoRetrieval*)
9: *InfoRetrieval*, *Patch*, *NewRepairList* = RepairAgent(*AdditionalInfo*)
10: **end while**
11: *ValidationResult* = ValidateAgent(*Patch*)
12: **while** not *ValidationResult*.isEmpty() **do**
13: *Patch* = RepairAgent(*ValidationResult*)
14: *ValidationResult* = ValidateAgent(*SinglePatch*)
15: **end while**
16: *PatchList*.append(*Patch*)
17: *RepairList*.extend(*NewRepairList*)
18: **end while**
19: **return** *PatchList*

3.3 Validate Agent

Verifying the correctness of generated patches is a challenge faced by all APR methods. Even with test cases, these methods can only produce plausible patches, and the presence of bugs still requires manual verification. Therefore, EH-Fixer is designed to generate plausible patches. Due to the lack of sufficient test cases, the Validate Agent leverages static analysis and LLMs to assess the plausibility of these patches. The main challenge is to design the criteria for the check. To address this challenge, we attempted to fix all 153 historical PEH bugs in the study with the Repair Agent, and analyzed 600 incorrect patches generated by LLMs. Our analysis indicated that: 52.8% (317/600) of incorrect patches used non-existent return values; 43.3% (260/600) referenced non-existent cleanup functions; 43.1% (259/600) included irrelevant modifications. Consequently, the Validate Agent scrutinizes patches for fabricated function calls and return values based on specified keywords, and assesses unnecessary modifications via LLMs.

First, the Validate Agent examines error-handling actions within the input patch. If they are not included in the input of the Repair Agent, the Validate Agent assumes that they are made-up and returns these mistakes. Subsequently, the Validate Agent checks the patch using LLMs for irrelevant modifications and returns found mistakes. Similar to the Repair Agent, the prompt of the Validate

Agent consists of 2 components, including **Instruction** and **Constraint**. The Instruction describes the buggy function and code snippets, the input patch, and guides LLMs to confirm whether the patch fixes only the target error-handling bug. The Constraint directs LLMs to produce outputs in a structured format.

We detail the EH-Fixer method for addressing PEH bugs in Algorithm 1. *BugLocation* serves as the input, providing the location of the error-handling bug. EH-Fixer constructs the error propagation path, logs all affected functions in *RepairList* (action 2), and stores generated patches in *PatchList*. It retrieves contextual information from the EH Dataset (action 4), clusters the *RepairList*, and selects the largest cluster *RepairCluster* that closest to the root node (action 5). EH-Fixer constructs prompts for functions in *RepairCluster*, which are then processed by the *RepairAgent* (action 6). If the *RepairAgent* requires more information, it initiates *InfoRetrieval* (action 7), and data is fetched from the EH Database (action 8) to continue the repair(action 9). The *RepairAgent* generates the *Patch*, which is reviewed by the Validate Agent (action 11). If modifications are needed, the patch is updated by the *RepairAgent* (actions 13 and 14). Finally, EH-Fixer secures the validated patch in *PatchList* (action 16) and updates the *RepairList* (action 17). This process is repeated TH_p times, and this parameter is discussed in Sec. 4.4.

4 Experiment

We conduct experiments to evaluate our approach by answering the following research questions:

- **RQ1:** How does EH-Fixer performs on real-world PEH bugs?
- **RQ2:** Does EH-Fixer outperform SOTA approaches?
- **RQ3:** How does each design affect the performance of EH-Fixer?
- **RQ4:** How do parameters affect the performance of EH-Fixer?

The experiments were conducted on a machine running Linux-18.04 with 64GB of RAM and an Intel i9-10900K CPU.

4.1 Answer to RQ1: Performance on Real-World PEH Bugs

In this section, we evaluate the performance of EH-Fixer on real-world PEH bugs through experiments conducted on the Linux Kernel and open-source applications.

4.1.1 Experiment Setup. As shown in Table 2, we evaluated EH-Fixer on the Linux Kernel and 9 open-source applications across different domains. Each application has over 500 stars on GitHub and is distinct from those in our study. Our evaluation consists of two main parts: fixing new PEH bugs and addressing historical PEH bugs. On the one hand, we used an existing approach [31] to detect error-handling bugs in the Linux Kernel, and repaired them using EH-Fixer. Newly discovered PEH bugs, which involve multiple function fixes, were reported to the developers. On the other hand, following the methods outlined in Sec. 2.1.2, we collected 44 historical PEH bugs from the Linux Kernel, and 45 from the applications, involving 125 and 108 buggy functions respectively. It is important to note that a total of 88 PEH bugs were collected from the Linux Kernel, with 44 bugs used for research and 44 used for experimentation. These two sets of bugs are from different modules of the Linux Kernel.

To evaluate the efficacy of EH-Fixer, we restored the source code of each historical PEH bug to its pre-repair state to ascertain whether EH-Fixer could generate patches that are semantically equivalent to the original fixes. Notably, some patches for historical bugs might have been included in the LLM training dataset. To mitigate the risk of data leakage, we segregated the historical bugs by the training cut-off time of the LLM. Given that EH-Fixer utilizes gpt-4-0613, we selected September 2021 as the cutoff time [47]. Consequently, we categorized the historical PEH bugs into two groups: 71 (37+34) and 18 (7+11) bugs, respectively. For historical PEH bugs prior to the training

Table 2. Performance on real-world PEH bugs.

Domain	Name	PEH Bugs	Precision	Repair Rate
Operating System	Linux Kernel	44	68.6%(151/220)	81.8%(36/44)
Networking	OpenVPN	11	74.5%(41/55)	81.8%(9/11)
Developer Tools	ESP-IDF	7	77.1%(27/35)	85.7%(6/7)
	BPF Compiler Collection	4	75.0%(15/20)	75.0%(3/4)
Database	Redis	6	73.3%(22/30)	83.3%(5/6)
Window Manager	Sway	5	68.0%(17/25)	80.0%(4/5)
	Mutter	2	70.0%(7/10)	100.0%(2/2)
Emulator	iSH	4	95.0%(19/20)	100.0%(4/4)
Media	HandBrake	4	65.0%(13/20)	75.0%(3/4)
Data Transfer Tool	Curl	2	90.0%(9/10)	100.0%(2/2)

cut-off time, we adopted the methodological of existing studies [69] and manually transformed the code to ensure equivalence. This transformation includes changes to function and variable names, as well as code structure. For instance, we may rename "$free\_page$" to "$release\_page\_memory$", reverse a condition like "$a > b$" to "$b < a$", or rewrite "$if(a)\{s_1\}else\{s_2\}$" as "$if(!a)\{s_2\}else\{s_1\}$". Afterward, we use a separate LLM to check if it can identify the original source of the code. If the LLM fails to recognize it, we consider the transformation valid and use it in our experiment.

We set the thresholds TH_e (in Sec. 3.2.2), TH_d (in Sec. 3.2.2), and TH_p (in Sec. 3.3) to 3, 0.4, and 5, respectively. The rationale behind these parameter settings is discussed in Sec 4.4. We use evaluation metrics commonly used in program repair, including precision and repair rate. Precision represents the ratio of correct patches to the total number of generated patches, whereas the repair rate indicates the proportion of successfully repaired bugs. A bug is considered successfully repaired if at least one candidate patch is correct.

4.1.2 Experiment Result. We used EH-Fixer to fix 34 newly detected error-handling bugs in the Linux Kernel, and found that the fixes for 9 of them involved multiple functions (PEH bugs). We submitted these patches to the developers. Currently, 2 have been confirmed, while the rest are still under review. As for the historical bugs, the results are shown in Table 2. On the Linux Kernel, EH-Fixer achieved a repair rate of 81.8% (36/44) with a precision of 68.6% (151/44*5). As for applications, it achieved a repair rate of 84.4% (38/45) with a precision of 75.6% (170/45*5). Note that, regarding historical bugs, EH-Fixer achieved a precision of 72.7% (258/355) and a fix rate of 81.7% (58/71) on the data before the training cut-off time. On the data after the training cut-off time, it achieved a precision of 70.0% (63/90) and a repair rate of 88.9% (16/18). The performance of EH-Fixer on these two types of data is similar, suggesting that the experimental results were not notably affected by the data leakage problem.

EH-Fixer failed to fix 15 PEH bugs, which we have classified into 3 categories. Firstly, some errors propagate through global variables. EH-Fixer trade-offs precision and repair rate to drop this case. As a result, it causes 2 buggy function in 2 PEH bug to be missed. We will discuss this trade-off in Sec. 5.3. Secondly, static analysis may be inaccurate. EH-Fixer performs static analysis based on AST without compiling the target software system. This allows EH-Fixer to be used in the software development phase, but it also causes it to perform less well than compilation-based methods. This resulted in 8 buggy functions in 5 PEH bugs were not found. Finally, 8 PEH bugs were not fixed due to the lack of relevant information. The propagation path of PEH bugs contains multiple functions, and a single inappropriate prediction of handling action set can cause the fix to fail. These 8 PEH bugs contained 22 buggy functions, and their propagation path affected 119 functions. Among these,

Table 3. Comparison in real-world bugs.

	Precision	Repair Rate
ChatRepair	13.7%(61/445)	14.6%(13/89)
ErrDoc	28.3%(126/445)	30.3%(27/89)
RLCE	38.0%(169/445)	42.7%(38/89)
EH-Fixer	**72.1%(321/445)**	**83.1%(74/89)**

12 buggy functions have less than 3 error-handling code snippets retrieved from the EH Dataset, and therefore are failed to be fixed by EH-Fixer. Note that, although a simple approach using keywords was employed for classifying logging actions, no repair failures due to misclassification were observed in the experiments. This is because logging functions and return statements often repeat within the same file or error propagation path, so occasional misclassification of logging actions does not impact the repair process.

Result 1: EH-Fixer has successfully repaired 9 new PEH bugs, with 2 confirmed by developers and the rest currently under review. In addressing historical bugs from the Linux Kernel and applications, EH-Fixer demonstrated its effectiveness in fixing real-world PEH bugs by achieving a repair rate of 83.1% (74/89).

4.2 Answer to RQ2: Comparison with the State-of-the-art

This section presents a comparison of EH-Fixer against SOTA approaches, focusing primarily on PEH bugs. Due to the lack of test cases, general-purpose APR techniques can hardly be used to fix error-handling bugs, let alone PEH bugs. Therefore, we compare our approach with existing approaches designed for error-handling bug fixes. Furthermore, we include several LLM-based APR approaches in our comparative experiments, as these can implement repairs using feedback information like the test outcome.

4.2.1 Experiment Setup. EH-Fixer was evaluated against three SOTA methods: a template-based error-handling bug repair approach ErrDoc [55], and two LLM-based APR approaches, ChatRepair [63] and RLCE [10]. ErrDoc identifies the type of error-handling bug and addresses it by duplicating adjacent error-handling code snippets. ChatRepair produces candidate patches by continuously integrating test results into LLMs, Whereas RLCE retrieves relevant code snippets from the software source code, assisting LLMs in implementing repairs.

We utilized the 89 (44+45) historical PEH bugs discussed in Sec. 4.1 as the test data, and employed precision and repair rate as evaluation metrics. Since these compared approaches were not specifically designed for PEH bugs, this study explores their theoretical maximum performance. ErrDoc and RLCE, which primarily target single-hunk repairs, were provided with all buggy functions (125+108) within the error propagation path. A PEH bug was deemed resolved if all supplied buggy functions were successfully repaired. As for ChatRepair, since it can be applied to multi-hunk repair, we provided it with the entire error propagation path, and used the compile result as the test outcome. To maintain fairness, both ChatRepair and RLCE were replicated using the gpt-4-0613 model as EH-Fixer, generating five candidate patches per PEH bug.

4.2.2 Experiment Result. The experimental results, as shown in Table 3, indicate that EH-Fixer exhibits superior performance in repairing PEH bugs, achieving a 40.4% increase in repair rates (from 42.7% to 83.1%). 48.6% (36/74) PEH bugs fixed by EH-Fixer cannot be fixed by all comparative

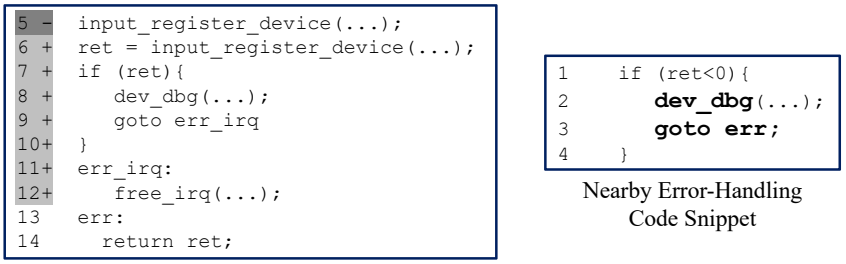


Fig. 7. Case study.

approaches. ChatRepair successfully addressed only 13 PEH bugs with a precision of 13.7%. This low performance stems from two issues: First, the lack of context often causes ChatRepair to select inappropriate action sets. Second, the error-handling code snippet is rarely covered by test cases. This makes it challenging to ensure the quality of generated patches based solely on the "passes compilation" criterion. Consequently, ChatRepair frequently produces patches that compile but are incorrect. The performance of ErrDoc on PEH bugs is also limited. This is because that ErrDoc fills templates with handling actions copied from nearby error-handling code snippets. This method led to 38 failed fixes due to unsuitable actions from nearby snippets. Additionally, it caused 24 failed fixes because some required error-handling actions were absent in the nearby snippets. RLCE employs a retrieval-augmented strategy to identify relevant function definitions and usages, slicing code snippets that exhibit data or control dependencies with errors from the buggy function and its callers. These snippets, as a subset of the context information—semantics and dependencies—within EH-Fixer, help LLMs understand the error impact. Therefore, RLCE achieves higher performance than the other two comparison approaches. However, RLCE failed to fix 51 PEH bugs. On the one hand, it still lacks some essential contexts, including available actions, and illustrative examples of the error-handling strategy. This deficiency can lead RLCE to select inappropriate action sets. On the other hand, RLCE does not validate the generated patches, occasionally resulting in irrelevant changes.

To illustrate our contribution, we conducted a case study. Fig. 7 illustrates a patch for the Linux Kernel, it fixes one of the functions affected by a PEH bug. This patch includes a logging function *dev_dbg* (line 8) and a resource cleanup function $free\_irq$. None of the three comparison approaches succeeded in generating this patch. ChatRepair, provided with all functions along the error propagation path, attempted to modify the target function only once in five tries but did not succeed. ErrDoc, using only the buggy function as input, identified *dev_dbg* but overlooked $free\_irq$. RLCE, similarly using the buggy function as input, failed to recognize the need to use the cleanup action. ChatRepair struggle to identify this function due to the lack of error propagation path analysis. Moreover, these approaches cannot select the appropriate handling action sets because $free\_irq$ is not retrieved. In contrast, EH-Fixer, accessing this function along the error propagation path in all five attempts, successfully retrieved $free\_irq$, leading to effective repairs in every instance.

Result 2: The repair rate of EH-Fixer outperformed comparative approaches in fixing PEH bugs, and 48.6% (36/74) PEH bugs fixed by EH-Fixer cannot be fixed by all comparative approaches.

Table 4. Ablation study.

	Input Token	Precision	Repair Rate
EH-Fixer-wp	11,075	17.3%(77/445)	29.2%(26/89)
EH-Fixer-wr	6,217	20.2%(90/445)	33.7%(30/89)
EH-Fixer-wc	23,020	**76.2%(339/445)**	83.1%(74/89)
EH-Fixer-wv	13,906	49.7%(221/445)	55.1%(49/89)
EH-Fixer	15,011	72.1%(321/445)	**83.1%(74/89)**

4.3 Answer to RQ3: Ablation Study

The design of EH-Fixer includes 4 key components: analyzing error propagation paths, employing a retrieval-augmented technique, collectively fixing functions with similar contexts, and filtering patches using the Validate Agent. We evaluate the impact of these components on the performance of the EH-Fixer through an ablation study.

4.3.1 Experiment Setup. We developed 4 variants for this study: EH-Fixer-wp, EH-Fixer-wr, EH-Fixer-wc, and EH-Fixer-wv. We removed the error propagation path analysis from EH-Fixer to form EH-Fixer-wp. EH-Fixer-wp directly inputs all functions that have data/control dependencies with the input function to LLMs, and requires LLMs to fix the PEH bugs within; We removed the retrieval-augmented technique to construct EH-Fixer-wr. EH-Fixer-wr removes "Contextual Information" and "Example" from the prompt, and no longer actively returns information that requires additional retrieval; EH-Fixer-wc removes function clustering during patch generation, and analyze only one function at a time; Lastly, we remove the Validate Agent from EH-Fixer to form EH-Fixer-wv. EH-Fixer-wv considers a patch as plausible if it compiles successfully.

We applied these variants to 89 historical PEH bugs discussed in Sec.4.1. We selected precision, repair rate, and cost as evaluation metrics. The cost metric assesses whether EH-Fixer can reduce unnecessary computational expenses, measured by the average number of tokens inputted into LLMs.

4.3.2 Experiment Result. We present the experimental results in Table 4. Firstly, EH-Fixer-wp managed to repair only 26 PEH bugs. As discussed in Finding 1, PEH bugs typically affect an average of 16.7 functions. Attempting repairs through a single interaction without considering the propagation path of errors complicates the repair process. This requires LLMs to process longer inputs and address multiple functions concurrently, often leading to incomplete or incorrect fixes, and resulting in a significant performance drop. Secondly, EH-Fixer-wr only fixes 30 PEH bugs, with a precision of 20.2%. This is consistent with the conclusion of Sec. 2.4 that contextual information is critical for selecting the appropriate action set. Thirdly, although EH-Fixer-wc has a similar performance to EH-Fixer, it uses tokens amounting to 153.4% of those used by EH-Fixer (23,020/15,011). This is due to the aggregation of functions with similar contexts, which substantially reduces the duplication of information and consequently decreases input length. The results suggest that clustering methods can effectively reduce the computational overhead associated with repairs without compromising performance. Finally, EH-Fixer-wv, despite successfully addressing 49 PEH bugs, demonstrates a notable performance decline compared to the EH-Fixer. This version frequently uses fabricated function/variable names and induces extraneous modifications, such as altering variable names or inserting unrelated code snippets. This result indicates that successful compilation does not necessarily guarantee the plausibility of patches for error-handling bugs.

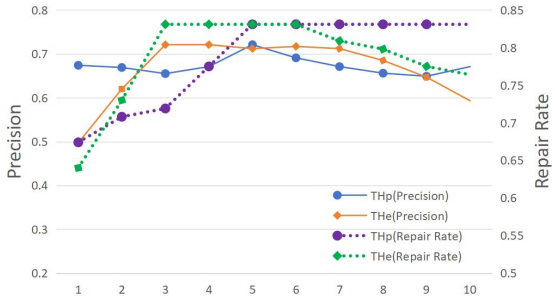


Fig. 8. Performance under different TH_e and TH_p.

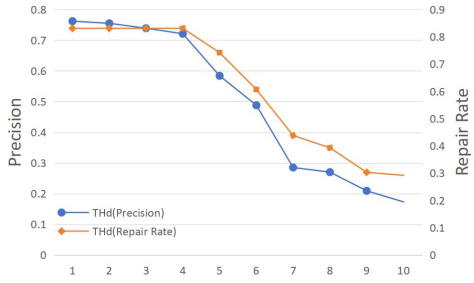


Fig. 9. Performance under different TH_d.

Result 3: Analyzing error propagation paths, employing a retrieval-augmented technique, and filtering patches are critical for the performance of EH-Fixer. Employing clustering methods can significantly decrease computational overhead without degrading performance.

4.4 Answer to RQ4: Impact of Parameters

EH-Fixer requires three parameters to be pre-defined, including TH_e and TH_d in Sec. 3.2.2, and TH_p in Sec. 3.3. TH_e affects the number of error-handling code snippets retrieved during patch generation. TH_d influences both the number and size of clusters, impacting the collective repair of functions. TH_p controls the quantity of candidate patches. All these parameters can impact the performance of EH-Fixer. This section evaluates how variations in TH_e, TH_d, and TH_p affect EH-Fixer.

4.4.1 Experiment Setup. We applied EH-Fixer to the 89 historical PEH bugs detailed in Sec. 4.1, and evaluated its precision and repair rate with different parameter settings. Initially, we set TH_e, TH_d, and TH_p to 3, 0.4, and 5, respectively. We then fixed two parameters and adjusted the remaining one. Specifically, TH_e and TH_p were varied from 1 to 10 in increments of 1, and TH_d ranged from 0 to 1 in increments of 0.1.

4.4.2 Experiment Result. The results are depicted in Fig. 8 and Fig. 9. As TH_e increases, both the precision and repair rate of EH-Fixer increase significantly. After TH_e reaches 3, the repair rate remains stable, while the precision rate fluctuates slightly. This is because these error-handling code snippets can help LLMs understand the usage of each available action. However, after TH_e exceeds 7, the precision and repair rates of EH-Fixer decrease. This is likely because error-handling code snippets tend to be repetitive. Excessive code, while not contributing additional context, augments the input size and may impair LLM performance. To minimize this overhead, we set TH_e to 3. As TH_p increases, the precision of EH-Fixer remains stable. This is because we simply run EH-Fixer repeatedly. Variability in LLM outputs does not affect this consistency, as the Validation Agent ensures stable precision. Conversely, the repair rate initially increases significantly as TH_p rises and then stabilizes once TH_p reaches 5. This result indicates that multiple iterations are necessary to fix some PEH bugs. Given that more outputs require developers to review more patches, we set TH_p at 5 to optimize the balance between the repair rate and the volume of outputs. As the threshold TH_d increases, the precision and repair rate of the method remains stable. However, when TH_d exceeds 0.4, both precision and repair rate begin to decline significantly. This decline can be attributed to

the impact of TH_d on clustering: a high threshold can result in excessively large clusters, which diminish the performance of EH-Fixer.

Result 4: TH_e, TH_d and TH_p influence the performance of EH-Fixer. To balance the repair rate against overhead, we set TH_e, TH_d, and TH_p to 3, 0.4, and 5, respectively.

5 Discussion

5.1 Generalizability

Both the survey and the experiments utilized PEH bugs from the Linux Kernel and high-star repositories on GitHub. Due to the significantly larger codebase of the Linux Kernel, a notable portion of the data was sourced from it, accounting for 28.8% (44/153) and 49.4% (44/89), respectively. The high proportion of data derived from the Linux Kernel could potentially affect the generalizability of our study and method.

Regarding the generalizability of the study, we present the data for both the Linux Kernel and other applications separately in link [3]. While there are minor differences—such as PEH bugs in the Linux Kernel affecting an average of 21.3 functions, compared to 14.8 in other applications—the characteristics of these PEH bugs are similar across both sets. These differences do not impact the conclusions of the study or the design of our method.

As for the generalizability of our method, as detailed in Sec. 4.1, EH-Fixer achieved similar repair rates (81.8% for the Linux Kernel and 84.4% for other applications) and precision rates (68.6% for the Linux Kernel and 75.6% for other applications). The slight decrease in performance on the Linux Kernel is primarily due to the higher number of functions affected by PEH bugs, which makes them more challenging to repair.

5.2 Static Analysis

To trace the relevant context and repair PEH bugs, EH-Fixer conducts program analysis on the AST to construct data/control-flow dependencies. This design allows EH-Fixer to analyze code without the need to compile the target project, and thereby can be applied to more projects automatically. This approach may generate more erroneous results compared to tools requiring compilation, such as LLVM. This is a trade-off between precision and scalability. According to the results of our experiments in Sec. 4.1, the number of mistakes caused by program analysis is comparatively low (4 buggy functions in 3 PEH bugs not been fixed). At present, EH-Fixer is implemented exclusively for C/C++. EH-Fixer employs tree-sitter [8] to obtain the AST of the code. Since tree-sitter supports more than 20 programming languages, EH-Fixer can be easily adapted to other programming languages with minimal code modifications.

5.3 Error Propagation Path Analysis

According to Finding 2, errors are mainly propagated through function return values, pointer parameters, and global variables. However, EH-Fixer excludes global variables when constructing error propagation paths. This is because such cases are uncommon, accounting for only 2.8% of the cases in the study (Sec. 2.3) and only causing one failed repair in our experiments (Sec. 4.1). Static analysis struggles to accurately trace the propagation paths of global variables. Incorporating global variables could marginally enhance the repair rate, yet it would substantially compromise the precision. Therefore, we trade off repair rate for precision, and do not consider global variables.

5.4 Patch Validation

Program repair typically depends on test cases to assess the plausibility of generated patches. However, the error-handling code snippet often features complex triggering conditions. For instance, certain memory-related error handling requires a specific runtime environment. This complexity results in a scarcity of test cases for these snippets, complicating the evaluation of patch plausibility for error-handling bugs. We utilize developer-implemented error-handling code snippets as the ground truth to understand the characteristics of incorrect patches generated by LLMs. These characteristics serve as an oracle to develop the Validation Agent, which assesses the patch plausibility. The experiments detailed in Sec. 4.3 demonstrate the efficacy of the Validate Agent in aiding EH-Fixer to rectify error-handling bugs.

5.5 Language Limitation

EH-Fixer consists of two main components: static analysis and LLM interaction. The static analysis is implemented using tree-sitter, which supports multiple programming languages, and the LLM can be applied to other languages as well. Although EH-Fixer is currently designed for C/C++, it can be extended to other languages with minimal modifications. Specifically, when extending to different languages, the static analysis code needs to be adjusted to track data/control dependencies according to the variations in AST structures.

6 Related Work

6.1 Error-Handling Bug Management

Research related to error-handling bugs has mainly focused on detection, existing methods for detecting error-handling bugs can be broadly divided into two categories: those based on manually constructed templates or error specifications, and those driven by learning-based approaches. In the first category, these methods typically begin with an analysis from a specific perspective, identify common patterns, and design corresponding templates [54, 64, 65, 68]. For instance, Tian and Ray [55], Jana et al.[18], and Li et al.[30] classified error-handling bugs and designed templates accordingly. Pakki et al.[48] studied cases where errors are handled too severely, while Wu et al.[62] examined disordered handling. These approaches often require substantial domain knowledge and struggle to adapt to software evolution. On the other hand, learning-based approaches typically leverage existing error-handling code snippets to derive templates or error specifications. Several studies [1, 12, 30, 32, 61, 70, 71] have explored the differences between normal paths and error paths, using features like the length of paths and return expressions. Liu et al.[31] focused on learning Error-Fault-Failure patterns from error-handling code snippets and performed inter-procedural detection. Jia et al.[21] investigated the equivalence of check conditions, while Shen et al. [53] concentrated on specific functions and pointers. These efforts are effective in detecting error-handling bugs, but do not focus on how to fix them. This makes them only able to detect partial functions in PEH bugs, and results in PEH bugs usually requiring several versions to be gradually discovered and fixed. Only a few existing approaches further implements the repair, and they mainly rely on manually defined templates. For example, ErrDoc [55] categorizes the error-handling bugs and constructs templates individually, and achieves the repair by copying the nearby error-handling actions. However, these approaches mainly focus on a single function, ignoring error propagation, and thus making it difficult to address PEH bugs.

Exception-handling mechanisms like try-catch, which are built-in features in programming languages such as Java and Python, have been extensively studied [6]. Existing approaches focus on this problem from two perspectives: whether a correct exception is thrown, and whether the thrown exceptions are properly handled. Some existing approaches [7, 9, 20, 46, 70] study whether a correct

exception is thrown, they usually studied the inconsistency of the error with the thrown exception. Some existing approaches [5, 15, 29, 42, 45, 57, 58] study whether the thrown exception is properly handled throw a data-flow analysis. These works focus on exception-handling mechanisms like try-catch statements, while EH-Fixer fix bugs for error handling of variable constrain violation, which is widely used in languages like C/C++.

6.2 Automatic Program Repair

Automated Program Repair (APR) is a key area in software engineering that has been the subject of numerous studies [55]. Arcuri and Yao [4] originally proposed a basic framework for the co-evolution of test cases and programs. Subsequent APR approaches like GenProg [59] have employed test cases and heuristic/constraint to guide and select program variants for repair [13, 26–28, 36, 44, 60]. In the area of template-based methods [14, 17, 33, 43], TBar [34] stands out by integrating templates from various sources to implement fixes. LSRepair [35], on the other hand, dynamically sources repair components from the code repositories to address bugs.

Recent advancements in machine learning [38–41] have led to an increase in studies using deep learning for program repair. This includes efforts to train repair models that transform the repair process into a sequence-to-sequence (seq2seq) task [23, 37, 66, 67], as demonstrated by DeepFix [16] and SequenceR [11]. The rise of large models has further enhanced these capabilities [22], with some research directing LLMs to perform repairs by incorporating test results [63], or by using retrieval-augmented techniques to provide more contextual information to LLMs [10]. Most related research focuses on generic bug fixes, relying primarily on test cases, and is not readily applicable to PEH bug repairs. EH-Fixer analyzes the characteristics of PEH bugs, directs LLMs to trace error propagation, and selects suitable action sets for handling, thereby enhancing its effectiveness in repairing PEH bugs.

7 Conclusion

This paper explores APR for PEH bugs. We studied 96 PEH bugs in the Linux Kernel and 10 applications, and summarized characteristics of error propagation and error-handling action. Based on these studies, we propose EH-Fixer, an APR approach powered by LLMs. EH-Fixer works by analyzing error propagation paths, and generating patches through a retrieval-augmented technique. We applied EH-Fixer to the Linux Kernel and 9 applications. EH-Fixer fixed 9 new PEH bugs, 2 of which have been confirmed by developers, and the remainder currently under review. Meanwhile, EH-Fixer demonstrated a repair rate of 83.1% (74 /89), significantly outperforming comparable approaches by fixing 48.6% (36/74) more PEH bugs. These results confirm EH-Fixer as a SOTA approach for repairing PEH bugs.

8 Data Availability

The source code and dataset can be found in the repository:
https://github.com/EH-Fixer/EH-Fixer

9 Acknowledgements

This research was funded by NSFC No. 62272473, the Science and Technology Innovation Program of Hunan ProvinceNo.2023RC1001 and No. 2023RC3012), NSFC No.U2441238 and No.62202474, National University of Defense Technology Research Project No.ZK24-01, and State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology.

References

[1] Mithun Acharya and Tao Xie. 2009. Mining API error-handling specifications from source code. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 370–384.

[2] Anon. 2025. Examples of Static Analysis and Key Prompts. https://github.com/EH-Fixer/EH-Fixer/blob/main/Example/Example.pdf Accessed: 2025-02-23.

[3] Anon. 2025. Separate Results of the Study. https://github.com/EH-Fixer/EH-Fixer/blob/main/Study/Separate%20Result.csv Accessed: 2025-02-23.

[4] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 162–168.

[5] Eiji Adachi Barbosa and Alessandro Garcia. 2018. Global-aware recommendations for repairing violations in exception handling. In *Proceedings of the 40th International Conference on Software Engineering*. 858–858.

[6] Pan Bian, Bin Liang, Yan Zhang, Chaoqun Yang, Wenchang Shi, and Yan Cai. 2018. Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering* 45, 10 (2018), 984–1001.

[7] Islem Bouzenia. 2022. Detecting Inconsistencies in If-Condition-Raise Statements. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–3.

[8] M. Brunsfeld. 2023. Tree-sitter. https://tree-sitter.github.io/tree-sitter/ Accessed 1. October 2021.

[9] Haicheng Chen. 2021. *Combating Fault Tolerance Bugs in Cloud Systems*. The Ohio State University.

[10] Yuxiao Chen, Jingzheng Wu, Xiang Ling, Changjiang Li, Zhiqing Rui, Tianyue Luo, and Yanjun Wu. 2024. When Large Language Models Confront Repository-Level Automatic Program Repair: How Well They Done?. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 459–471.

[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] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V Thakur. 2019. Effective error-specification inference via domain-knowledge expansion. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 466–476.

[13] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*. 30–39.

[14] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[15] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. 2016. Automatic runtime recovery via error handler synthesis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 684–695.

[16] 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.

[17] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 888–891.

[18] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications.. In *USENIX Security Symposium*. 345–362.

[19] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 61–71.

[20] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, and Ji Wang. 2019. Automatically detecting missing cleanup for ungraceful exits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 751–762.

[21] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, Ji Wang, Xiaodong Liu, and Yunhuai Liu. 2019. Detecting error-handling bugs without error specification input. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 213–225.

[22] 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.

[23] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[24] Stephen C Johnson. 1967. Hierarchical clustering schemes. *Psychometrika* 32, 3 (1967), 241–254.

[25] Van-Hoang Le and Hongyu Zhang. 2023. Log parsing with prompt-based few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2438–2449.

[26] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604.

[27] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[28] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[29] Yan Lei, Chengnian Sun, Xiaoguang Mao, and Zhendong Su. 2018. How test suites impact fault localisation starting from the size. *IET software* 12, 3 (2018), 190–205.

[30] Chi Li, Min Zhou, Xinrong Han, and Ming Gu. 2021. Sensing Error Handling Bugs in SSL Library Usages. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 686–692.

[31] Haoran Liu, Zhouyang Jia, Shanshan Li, Yan Lei, Yue Yu, Yu Jiang, Xiaoguang Mao, and Xiangke Liao. 2024. Cut to the Chase: An Error-Oriented Approach to Detect Error-Handling Bugs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1796–1818.

[32] Huqiu Liu, Yuping Wang, Lingbo Jiang, and Shimin Hu. 2014. PF-Miner: A new paired functions mining method for Android kernel in error paths. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 33–42.

[33] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–12.

[34] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 31–42.

[35] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 658–662.

[36] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[37] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[38] Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2024. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622* (2024).

[39] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At which training stage does code data help llms reasoning? *arXiv preprint arXiv:2309.16298* (2023).

[40] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? *arXiv preprint arXiv:2406.01422* (2024).

[41] Yingwei Ma, Yue Yu, Shanshan Li, Zhouyang Jia, Jun Ma, Rulin Xu, Wei Dong, and Xiangke Liao. 2023. Mulcs: Towards a unified deep representation for multilingual code search. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 120–131.

[42] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89 (2014), 51–62.

[43] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th international symposium on software testing and analysis*. 441–444.

[44] 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.

[45] Tam Nguyen, Phong Vu, and Tung Nguyen. 2019. Recommending exception handling code. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 390–393.

[46] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and Fernando Castor. 2018. Do android developers neglect error handling? a maintenance-Centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software* 136 (2018), 1–18.

[47] OpenAI. 2023. *GPT-4 Turbo and GPT-4 Models*. https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4 Accessed: 2024-09-11.

[48] Aditya Pakki and Kangjie Lu. 2020. Exaggerated error handling hurts! an in-depth study and context-aware detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1203–1218.

[49] Martin P Robillard and Gail C Murphy. 2000. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*. 2–10.
[50] Guoping Rong, Yangchen Xu, Shenghui Gu, He Zhang, and Dong Shao. 2020. Can you capture information as you intend to? A case study on logging practice in industry. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 12–22.
[51] Cindy Rubio-González, Haryadi S Gunawi, Ben Liblit, Remzi H Arpaci-Dusseau, and Andrea C Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 270–280.
[52] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.
[53] Qintao Shen, Hongyu Sun, Guozhu Meng, Kai Chen, and Yuqing Zhang. 2023. Detecting API Missing-Check Bugs Through Complete Cross Checking of Erroneous Returns. In *International Conference on Information Security and Cryptology*. Springer, 391–407.
[54] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1069–1084.
[55] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 752–762.
[56] 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.
[57] Westley Weimer. 2004. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. 419–431.
[58] Westley Weimer and George C Necula. 2008. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–51.
[59] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.
[60] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering*. 1–11.
[61] Baijun Wu, John Peter Campora III, Yi He, Alexander Schlecht, and Sheng Chen. 2019. Generating precise error specifications for c: A zero shot learning approach. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.
[62] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and detecting disordered error handling with precise function pairing. In *the 30th USENIX Security Symposium (Security'21)*.
[63] Chunqiu Steven Xia and Lingming Zhang. 2024. Keep the Conversation Going: 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 (ISSTA'24)*.
[64] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 512–523.
[65] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 499–510.
[66] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
[67] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th international conference on software engineering*. 1506–1518.
[68] Dongyang Zhan, Xiangzhan Yu, Hongli Zhang, and Lin Ye. 2022. ErrHunter: Detecting Error-Handling Bugs in the Linux Kernel Through Systematic Static Analysis. *IEEE Transactions on Software Engineering* 49, 2 (2022), 684–698.
[69] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 535–547.
[70] Hao Zhong. 2022. Which Exception Shall We Throw?. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[71] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 307–318.

Received 2024-09-13; accepted 2025-04-01