PanicFI: An Infrastructure for Fixing Panic Bugs in Real-World Rust Programs

YUNBO NI*, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZIXI LIU*, State Key Laboratory for Novel Software Technology, Nanjing University, China
YANG FENG†, State Key Laboratory for Novel Software Technology, Nanjing University, China
RUNTAO CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
BAOWEN XU, State Key Laboratory for Novel Software Technology, Nanjing University, China

The Rust programming language has garnered significant attention due to its robust safety features and memory management capabilities. Despite its guaranteed memory safety, Rust programs suffer from runtime errors that are unmanageable, i.e., panic errors. Notably, traditional memory issues such as null pointer dereferences, which are prevalent in other languages, are less likely to be triggered in Rust due to its strict ownership rules. However, the unique nature of Rust's panic bugs, which arise from the language's stringent safety and ownership paradigms, presents a distinct challenge. Over half of the bugs in rustc, Rust's own compiler, are attributable to crashes stemming from panic errors. However, addressing Rust panic bugs is challenging and requires significant effort, as existing fix patterns are not directly applicable due to the design and feature of Rust language. Therefore, developing foundational infrastructure, including datasets, fixing patterns, and automated repair tools, is both critical and urgent.

This paper introduces a comprehensive infrastructure, namely PanicFI, aimed at providing support for understanding Rust panic bugs and developing automated techniques. In PanicFI, we construct a dataset, Panic4R, comprising 102 real panic bugs and their fixes from the top 500 most downloaded open-source crates. Then, through an analysis of the Rust compiler implementation, we identify Rust-specific patterns for fixing panic bugs, providing insights and guidance for generating patches. Moreover, based on these patterns, we develop an automated fixing tool, namely PanicKiller, as an artifact, which has already contributed to the resolution of 28 panic bugs in open-source projects. The practicality and efficiency of PanicKiller confirm the effectiveness of the patterns mined within PanicFI. Furthermore, Panic4R serves as a benchmark for evaluating APR tools focused on Rust panic bugs. We believe the construction and release of PanicFI could enable the expansion of automated repair research tailored specifically to Rust programs, addressing unique challenges and contributing significantly to advancements in this field.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools.*

Additional Key Words and Phrases: Rust, program repair, fault localization

*Both authors contributed equally to this research.
†Corresponding author.

Authors' addresses: Yunbo Ni, yunboni@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Zixi Liu, zxliu@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Yang Feng, fengyang@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Runtao Chen, 211220018@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Baowen Xu, bwxu@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2025/1-ART1 $15.00
https://doi.org/10.1145/3773991

ACM Reference Format:
Yunbo Ni, Zixi Liu, Yang Feng, Runtao Chen, and Baowen Xu. 2025. PanicFI: An Infrastructure for Fixing Panic Bugs in Real-World Rust Programs. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 37 pages. https://doi.org/10.1145/3773991

1 INTRODUCTION

As a statically typed programming language, Rust has gained popularity for its well-known memory safety guarantees and high performance. Recently, the White House Office of the National Cyber Director also emphasized the necessity of using programming languages that have fewer memory safety vulnerabilities [29], and nominated Rust as an example of a memory-safe programming language. The foundational principles of Rust, including ownership, borrowing, and lifetimes, enable developers to implement secure and efficient programs. Rust's emphasis on zero-cost abstractions and fearless concurrency has significantly contributed to its popularity in systems programming [39, 55, 56, 58]. This design has led to an increase in the development of widely recognized software projects written in Rust [25–28, 44].

Although Rust boasts security features and significantly reduces common bugs such as null pointer dereference, uninitialized variables, and data races, it could suffer from *panic errors*, which are caused by a Rust-specific error handling mechanism. In Rust, errors are categorized into two types: recoverable errors and unrecoverable errors. Recoverable errors can be caught and handled by the program, allowing execution to continue. In contrast, unrecoverable errors [22] cause the program to panic, typically indicating fatal issues that lead to runtime termination. Panic errors can have severe consequences, including program crashes and termination. Such errors may even introduce memory risks, particularly if they occur during critical operations like a drop operation, which could result in double frees [37]. Additionally, in a multithreaded environment, a panic in one thread can propagate, causing other threads to become poisoned [12] and jeopardizing the stability of the entire program. The severity of panic errors is underscored by their potential to leave resources improperly handled, such as unclosed file descriptors or network connections, contributing to program instability [98].

Rust's panic mechanism differs significantly from Java's structured exception handling. While Java's exceptions are designed for handling routine errors, Rust reserves panic for unrecoverable situations, which can have a substantial impact on program stability. In fact, the Rust compiler itself, rustc, written in Rust, also exhibits vulnerabilities to panic bugs. *Over half* of the issues in the official Rust GitHub repository are categorized as Internal Compiler Errors (ICE), indicating the panic bugs triggered in rustc [10]. The steep learning curve associated with Rust's unique memory management model, compounded by the lengthy stack traces generated during panic errors, presents a significant challenge for developers. These challenges make understanding and resolving panic bugs essential for ensuring the reliability and stability of Rust programs.

Although many automatic bug-fixing techniques have been developed for mainstream programming languages such as Java and C/C++, these techniques are difficult to adapt to Rust's panic bugs due to fundamental differences in language design, error models, and runtime behavior. First, Rust's panic mechanism stems from its unique ownership and borrowing system, which differs fundamentally from the error models targeted by traditional repair techniques. While memory safety errors are common in C/C++, Rust largely prevents them at compile time [4]. Instead, many runtime panics in Rust result from safety violations in abstractions like smart pointers [14], requiring fixes that are tightly coupled with its memory management model. Second, many strategies used in other languages are fundamentally inapplicable to Rust. For example, Java's null pointer exceptions are irrelevant in Rust, which uses the `Option` enum and pattern matching to enforce safe unwrapping [11]. Finally, the current Rust ecosystem lacks mature toolchain support for automated

repair. To the best of our knowledge, there are no publicly available and reproducible datasets or specialized tools for repairing Rust panic bugs. This absence of foundational infrastructure further limits the applicability of existing repair approaches in the Rust ecosystem.

Due to the lack of mature infrastructure, such as datasets and repair patterns, fixing panic bugs can be a tedious and challenging task for Rust developers. Recently, a few program repairing tools have been proposed for Rust, yet they are insufficient to address the most severe panic-related bugs. Rust-lancet [96] was developed to tackle bugs related to violations of ownership rules through three specific strategies. However, these strategies are tailored exclusively to Rust's ownership rules, which are verified prior to runtime, making them unsuitable for addressing panic bugs. Similarly, Ruxanne [78] and RustAssistant [47] focus on common compilation issues, such as incorrect data types, but these patterns do not effectively mitigate panic bugs.

Our work. To overcome the aforementioned challenges and fill the program understanding gap, in this paper, we design and implement the first infrastructure PanicFI aiming at automatically fixing panic bugs for real-world Rust programs. We have constructed a dataset Panic4R, containing more than 100 panic bug instances and corresponding fix patches, derived from open source projects in the ecosystem. Referring to the Rust implementation code, we further perform fix pattern mining with Rust-specific features to provide a reference for the community to better understand panic bugs and fixing strategies. Further, we implement an automated fixing tool for panic bugs, namely PanicKiller, which first applies dependency analysis for cross-file level pattern matching, then prioritizes the patches and combines them with semantic information, and finally outputs sorted patches with scores. We compared PanicKiller to state-of-the-art LLMs and LLM-based APR tools, employing both single and multi-round conversations as baselines. Results demonstrate that PanicKiller outperforms baseline tools in panic fixing, highlighting its practicality and reliability. Moreover, PanicKiller has effectively resolved issues in open-source Rust projects, with 28 panic bug fixes validated and merged by developers.

Contributions. In summary, the contributions of this work are as follows:

- **Dataset.** We constructed a reproducible dataset for fixing Rust panic bugs, named Panic4R, which consists of 102 real-world bugs and their corresponding fixes, all collected from the PR records of the top 500 most downloaded Rust crates. Each bug-fix pair is meticulously organized and manually validated to be reproducible on the latest stable Rust compiler, making the dataset a reliable foundation for future research.
- **Patterns.** We mined a series of fix patterns for panic bugs compliant with Rust-specific features. The potential application for these patterns could be developing automated repair tools, providing references for developers, serving as a dataset for fine-tuning LLMs, etc.
- **Tool.** We introduced PanicKiller, an automated repair tool specifically for Rust panic bugs, designed to address issues in real-world and large-scale Rust programs. Our experiment results show that PanicKiller is more efficient than commercial LLM-based tools and has successfully resolved 28 open issues in Rust projects on GitHub.

Paper Organization. The key components of our proposed infrastructure PanicFI are illustrated in Figure 1. In Section 2, we describe the process of dataset collection and present the corresponding statistical results. In Section 3, we outline the process of mining repair patterns and provide a detailed discussion of the causes of each type of panic bug and their respective repair strategies. Section 4 introduces a pattern-based APR tool, which combines and sorts patterns for application to fix real-world panics. In Section 5, we discuss the application scenarios of PanicFI and compare it with existing repair patterns. Section 6 provides a summary of related work, and in Section 7, we conclude the paper.

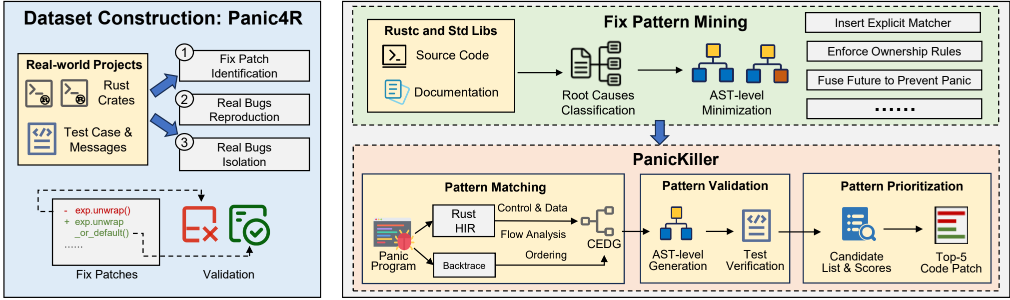


Fig. 1. Key components of the proposed infrastructure PanicFI.

2 DATASET CONSTRUCTION

To build a dataset containing real-world panic bugs and their patches, we follow the construction process of Defects4J [57], which is the most classic dataset in the APR task. We select the top 500 most downloaded crates from the public repository of Rust crates [30] for real bugs and fix patches extraction, so as to construct a real dataset. Specifically, we manually employ the following workflow to collect code and thus guarantee the quality and reliability of the dataset.

(1) Identifying Real Fix Patches. For each target crate, we refer to its official repository and review the list of issues. We filter for closed issues with merged pull requests (PRs) or corresponding fix commit. If an issue's title or description contains keywords indicating panics, such as "panic/thread panicked at...", it is flagged as a potential panic bug. Since user reports sometimes lack sufficient details to explicitly identify panics, we also examine PRs and commits containing relevant keywords. Developers often describe the fixed issue in PRs, which can reveal the presence of a panic bug. In such cases, we trace the corresponding issue for further analysis. After gathering all potentially valid issues, we meticulously review the code changes associated with their PRs to confirm that they address a panic bug. Any modifications unrelated to fixing the panic bug are excluded. For example, PRs that solely add or remove test cases without modifying the crate's source code are excluded. Similarly, changes limited to API documentation updates are outside the scope of our dataset.

(2) Reproducing Real Bugs. To ensure the reproducibility of panic bugs in each crate, which requires specific versions, we download the source code for the corresponding commit IDs before and after the PR commit. The two versions of the target crate are regarded as V_{bug} and V_{fix}, respectively. In most cases, users provide code examples to reproduce the specific panic bug, while developers often include test cases along with the fix patch. We refer to the descriptions in the PR or the code linked to the corresponding issue to derive a minimal test case, T_{test}, that triggers the bug—typically by invoking the target crate's API. To ensure the reliability of T_{test}, we verify that it successfully triggers the panic bug when executed on V_{bug} and that it compiles and runs without triggering the bug when tested on V_{fix}. This validation guarantees the reproducibility and accuracy of the identified panic bug and its fix.

(3) Isolating Real Bugs. In the target crate's repository, each PR commit may contain more than one code change, such as adding other functional logic or modifying data structures for new features, but not as a fix patch for panic. To ensure the accuracy of the fix patches in the dataset, we manually verified code changes and kept only the patches used to fix the panic bug. If a PR contains multiple fixes for panic bugs, we also split it into multiple code-fix pairs. As a result, the patches for V_{fix} versus V_{bug} have no irrelevant content and are the precise changes necessary to fix the panic bug. To summarize, each of the real datasets we build contains two crate versions, V_{bug} and V_{fix}, and a test case T_{test} that triggers panic bugs on V_{bug} and is compilable on V_{fix}.

Table 1 provides detailed information about the real-world dataset Panic4R. In total, we collected 102 panic bugs along with their corresponding fixing patches. These bugs were selected from 51 crates spanning 20 categories, each with an average of over 3, 000 stars, demonstrating their widespread adoption in practical applications. Moreover, these crates contain an average of more than 20, 000 lines of code, highlighting the considerable complexity involved in analyzing, debugging, and resolving panic issues in real-world projects.

Table 1. Crates and number of real bugs available in Panic4R.

Crate		**GitHub Info**				**Dataset Info**		
Name	**Category**	**Creation**	**# Stars**	**# Commits**	**# Contrib.**	**LoC**	**# Tests**	**# Bugs**
syn	development	2016-09	2,976	4,986	96	58,373	181	4
rand	algorithms	2015-02	1,711	3,903	270	15,662	325	2
regex	text processing	2014-12	3,601	1,525	185	96,367	399	6
aho-corasick	text processing	2015-06	1,053	293	30	12,432	78	1
num-traits	algorithms	2017-12	756	1,164	121	4,171	67	1
clap	CLI	2015-02	14,615	8,395	517	55,344	1,459	4
serde_json	encoding	2015-05	5,019	1,758	120	16,407	150	2
strsim	text processing	2015-02	418	116	19	1,108	96	1
time	date and time	2014-11	1,137	1,044	70	32,462	504	2
idna	algorithms	2013-12	1,352	1,377	170	10,021	105	7
hashbrown	data structures	2018-10	2,528	1,094	110	11,508	167	1
proc-macro2	development	2017-05	778	1,178	28	5,113	59	3
smallvec	data structures	2015-04	1,387	487	80	3,045	76	2
percent-encoding	encoding	2013-12	1,352	1,377	170	10,021	105	2
textwrap	CLI	2016-12	473	1,110	28	3,486	137	1
chrono	date and time	2014-03	3,405	1,678	211	19,272	299	13
uuid	data structures	2014-07	1,034	1,627	144	4,944	120	1
nom	parsing	2014-11	9,631	2,740	354	17,640	279	5
tokio	asynchronous	2016-09	27,690	4,001	855	86,782	895	5
hyper	network	2014-08	14,863	2,848	395	22,906	152	1
futures	asynchronous	2016-03	5,481	3,432	340	29,753	357	5
toml	config	2017-06	753	2,080	139	20,189	238	2
httparse	network	2015-02	606	235	45	6,481	64	1
object	binary analysis	2016-08	690	988	97	47,519	63	1
rustc demangle	debugging	2016-05	241	124	18	2,309	66	1
rustls	cryptography	2016-05	6,404	3,633	162	52,793	566	1
form_urlencoded	encoding	2013-12	1,352	1,377	170	10,021	105	1
gimli	debugging	2016-06	866	1,459	67	40,142	484	1
reqwest	web	2016-07	10,175	1,117	348	14,901	177	1
num-bigint	science	2017-12	548	1,336	119	12,963	205	1
rayon	concurrency	2014-10	11,349	2,211	135	28,167	445	1
bumpalo	memory	2018-11	1,502	426	51	5,641	71	2
filetime	filesystem	2015-05	123	187	44	1,173	11	1
fixedbitset	data structures	2015-04	130	178	26	3,231	63	1
phf	data structures	2014-01	1,865	793	67	2,654	60	1
prost	encoding	2017-06	4,058	699	155	13,014	138	1
pest	parsing	2016-04	4,739	1,310	128	34,820	520	1
serde-yaml	encoding	2016-02	974	947	37	10,445	90	1
libm	math	2018-07	557	666	55	15,566	105	1
prettyplease	development	2022-01	652	490	2	91,351	3	1
bytemuck	encoding	2019-09	763	422	59	4,924	59	1
cargo_metadata	development	2017-01	178	515	81	2,145	22	1
tinytemplate	template	2019-01	201	93	7	1,955	58	1
tar	filesystem	2014-07	638	523	76	5,665	91	1
plotters	visualization	2019-04	4,002	1,587	99	18,271	132	2
pretty_assertions	development	2017-03	1,220	188	15	1,285	48	1
yansi	CLI	2017-06	254	82	4	1,412	7	1
crossbeam	concurrency	2015-05	7,609	2,263	147	29,569	631	1
brotli-decompressor	compression	2016-04	54	190	13	31,180	106	1
indicatif	CLI	2017-04	4,552	794	127	6,678	90	1
md5	cryptography	2015-06	75	109	9	408	4	1

To better understand the bug resolution lifecycle, we quantitatively analyze the time spent in each stage of the process. Table 2 shows the distribution of durations in Panic4R, measured in

hours, divided into three stages: response time, discussion time, and integration time. Response time is the interval from issue creation to the first maintainer or contributor comment, reflecting the community's initial reaction speed. It is generally short, with a median of 0.64 hours, indicating prompt attention to panic-related issues. Discussion time covers the period from the first comment to the submission of the first pull request (PR) fixing the issue. This stage, involving problem investigation and patch development, is much longer, with a median of 11.62 hours and a mean exceeding 2, 000 hours, highlighting the complexity of diagnosis and fix preparation. Integration time spans from PR submission to its merge into the codebase, including review, testing, and integration. While the median is 8.86 hours, some patches take over 9, 000 hours, revealing considerable delays likely caused by human-in-the-loop bottlenecks. In some cases, an issue is resolved without any developer comment. Besides, a fix PR may be submitted immediately after the issue is created, or the code may be changed directly on the main branch without a PR. For these situations, we mark the related stage as N/A and count the number of N/A instances occur at each stage, shown in Table 2. Overall, the relatively short response times suggest that panic-related bugs tend to receive timely attention from the community. However, the duration of the entire resolution process appears to depend largely on developer involvement, as investigation, debugging, and patch development can require substantial time. The prolonged discussion and integration phases underscore the inherent difficulties in root cause identification and producing effective fixes.

Table 2. Statistics of time intervals (in hours) in issue resolution process in Panic4R.

Stage	Min	Max	Mean	Median	Count of N/A
Response Time	0.11	4635.28	208.25	0.64	43
Discussion Time	0.03	47210.66	2030.46	11.62	16
Integration Time	0.03	9060.21	384.12	8.86	0

3 FIX PATTERN MINING

To comprehensively identify the causes and fix patterns for panic bugs in Rust programs, we analyze and infer data from the Rust compiler implementation code [16]. We avoid mining fix patterns from Panic4R to ensure diversity and comprehensiveness in the repair patterns. The root causes of panic errors in Panic4R may not be exhaustive; for instance, overflow issues may stem from different operators, such as addition or subtraction, which could be impossible to cover fully through real-world examples. In contrast, official implementation code includes all such cases and provides concise examples, making it more suitable for our analysis and collection.

In the Rust compiler source code, developers annotate potential panic occurrences at the function level using the `# Panics` tag, which provides detailed descriptions of the causes and conditions under which the panic can occur. These function-level annotations often include supplementary tags such as `# Safety` or `# Examples`. The `# Safety` tag typically outlines strategies to prevent the panic or ensure safe behavior, while the `# Examples` tag often provides code snippets that illustrate scenarios triggering the panic or demonstrate correct usage of the functionality. This rich, annotated information serves as a valuable resource for analyzing and categorizing function-level fix patterns for different types of panics. By studying the annotations, we can identify common practices and strategies employed by developers to address these issues, which aid in understanding the underlying causes and the corresponding solutions. Additionally, file-level documentation provides a broader overview of each module's functionality and purpose. This higher-level context complements the code-specific annotations by offering insights into the design and intended usage of the module, further enriching our understanding of panic-related issues and their resolutions. To extract and analyze this information, we leverage both *rustdoc* [19] and *syn* [34]. We first use

rustdoc with the *–output-format json* flag to extract structured documentation metadata from the compiler codebase. This allows us to collect all public and private functions along with their associated doc-comments, including the `# Panics`, `# Safety`, and `# Examples` tags. We then locate the original source code of each function using the span information provided by *rustdoc*. For each extracted function, we use *syn* to parse the corresponding code at the AST level, enabling fine-grained analysis of the function body and its control-flow structure. By recursively traversing the parsed AST, we identify explicit panic-related constructs such as `panic!`, `unwrap()`, `expect()`, and assertion macros, linking them back to the documented annotations.

Moreover, we refer to the Request for Comments (RFCs) list [17] for detailed insights into specific designs related to panic bugs. The RFCs provide valuable information, including the motivation and design principles, offering a thorough explanation of the background and mechanisms involved. This detailed documentation serves as an excellent resource for understanding the intricacies of panics. Together, these annotations and documentation create a comprehensive knowledge base for studying and addressing panics in the Rust compiler.

```rust
/// # Panics: Panics if the value is
/// currently mutably borrowed.
/// For a non-panicking variant, use
/// [`try_borrow`] (#method.try_borrow).
/// # Examples:
/// let c = RefCell::new(5);
/// let borrowed_five = c.borrow_mut();
/// // causes panic
/// let borrowed_five2 = c.borrow();
pub fn borrow(&self) -> Ref<'_, T> { ... }
```

Listing 1. Annotated implementation of `borrow()` in `RefCell` [9].

Table 3. Categorization of Root Causes of Panics

Root Cause	Distribution
Improper `unwrap`	9 (2.25%)
Interior Mutability Violation	24 (6.00%)
Async Function Misuse	50 (12.50%)
Arithmetic Overflow	78 (19.50%)
Invalid String Boundary	31 (7.75%)
Unexpected Edge Cases	119 (29.75%)
Logical Panics	89 (22.25%)
Total	**400 (100.0%)**

For instance, Listing 1 presents a snippet of Rust source code that implements built-in type from `rust/library/core/src/cell.rs`, annotated to highlight a panic bug related to the ownership mechanism. Specifically, it examines the `borrow()` method in the context of ownership-related panics in `RefCell`, a Rust smart pointer that facilitates interior mutability. The `# Panics` annotation in this snippet provides details about the root cause of the panic, which occurs due to a double borrowing of a `RefCell` parameter. The annotation also suggests a mitigation strategy, recommending the use of the `try_borrow()` method as an alternative to avoid such panics. Additionally, the file-level annotations offer broader insights into borrowing rules and the concept of interior mutability, which are essential for understanding the underlying mechanism of this panic. Besides, RFC#1660 [1] documents the proposal for `try_borrow()`, detailing its motivation and design, offering further context and rationale for its introduction. These details are further discussed in Section 3.2. By systematically analyzing all documented panic cases and their corresponding resolutions in the Rust source code, we compile a comprehensive collection of fix strategies, providing valuable guidance for addressing similar issues.

To further delineate fix patterns, we convert code modifications into Abstract Syntax Trees (ASTs). By omitting irrelevant tokens, such as variable names, we identify similar AST transformation structures that reveal potential fixes for each identified root cause. To better understand the nature of panic bugs in Rust programs, we further categorize their root causes based on the contexts in which they occur. Table 3 presents a breakdown of these root causes into seven categories, along with their corresponding frequencies in the dataset. This categorization allows us to identify common

patterns in how panic bugs arise and how they are typically addressed in the Rust compiler's implementation. One commonly observed category is the unexpected edge cases, which often result from unanticipated input conditions or rare program states not sufficiently guarded against. Other prevalent categories include logical panics, arithmetic overflow, and async function misuse. These typically stem from violations of program invariants, unchecked arithmetic operations, or incorrect handling of asynchronous execution contexts, respectively. In the following subsections, we provide a detailed explanation for each root cause category, along with representative examples and the fix patterns commonly applied to resolve them. Building on this categorization, we conducted an in-depth analysis of the corresponding fixes in the Rust source code. As a result, we identified a total of 10 fix patterns, each potentially encompassing multiple sub-patterns. Below, we summarize 21 sub-patterns categorized under different scenarios.

3.1 Improper Error Handling Caused by `Unwrap`

Rust classifies errors into two main categories: *unrecoverable* and *recoverable* errors. *Unrecoverable* errors, which typically indicate unexpected issues such as missing features or logical flaws, are handled using the `panic!` macro. As for *recoverable* errors, Rust provides an enumeration `Result<T, E>`, where both `T` and `E` are generic types that can represent any specific data type. This enumeration is used for operations that may return either `Ok(T)`, representing success and containing a value of type `T`, or `Err(E)`, representing an error and containing an error value of type `E`. In addition, `Option<T>` is a more general enumeration that represents a value that is either `Some(T)`, containing a value of type `T`, or `None`, indicating the absence of a value. A common method for extracting values, `T`, from these enumerations is the `unwrap()` method, which returns the value if it is `Some(T)` or `Ok(T)`, but panics if the result is `None` or `Err(E)`.

For example, the code snippet in Listing 2 demonstrates a function `get_name()` that returns an `Option<String>`. When `get_name()` returns a `Some()` value, calling `unwrap()` method on line 6 works as expected, and the variable `name` is assigned the type `String`. However, if `get_name()` returns `None`, invoking `unwrap()` without any check triggers a panic error, causing the program to abort. Furthermore, as shown in lines 8, the panic message is not very clear, especially if the developer is unfamiliar with the details of the `get_name()` function. Consider the case where `get_name()` is a complex API function with more parameters. If a user provides invalid input and invokes `unwrap()`, resulting in a panic, the panic message does not clearly indicate the bug cause. Instead, the user would be unsure whether the issue lies with the input or with the internal implementation of the API. In such cases, the message provides little context about the actual issue, making it hard to diagnose and fix the problem.

```
1  fn get_name(flag: bool) -> Option<String> {
2      if flag { Some(String::from("Alice")) } else { None }
3  }
4
5  fn main() {
6      let name: String = get_name(true).unwrap(); // The program operates as expected
7      let no_name: String = get_name(false).unwrap();
8     // thread 'main' panicked at src/main.rs:7:43: called `Option::unwrap()` on a `None` value
9 }
```

Listing 2. An example of improper error handling in a Rust program using `unwrap()`, leading to a panic.

To fix panics caused by improper use of `unwrap()`, we introduce two repair patterns for constructing control flow that includes null value checking and error handling. Note that we focus on mining the patterns specifically suited for Rust programs. While logical structures such as `if-else`

can achieve similar effects, we do not list them here, as the conditional logic in these structures is less intuitive, making them a less recommended repair strategy. In addition, the standard library also provides other APIs to prevent panics, such as the `catch_unwind()` closure, which returns `Ok` with the closure's result if the closure does not panic, and returns `Err(cause)` if the closure panics. However, the official documentation[8] strongly discourages using this function as a general try-catch mechanism, as it does not guarantee catching all panics. Therefore, to enhance the robustness and generality of program fixes, we do not consider this approach as a repair pattern for panic bugs.

FP 1. Insert Explicit Matcher. The first fix pattern is to insert explicit matcher for pattern matching, which is powerful control flow construct and tightly connected to Rust's enum allowing hand the different variants of an enum explicitly. As recommended by the Rust Core Library [11], developers are suggested to apply pattern matching to explicitly handle the `None` cases. This repair pattern involves replacing the direct `unwrap()` call with `match` expressions. For valid values, the result of the original expression is returned, while for `None` values, an error is returned, thereby preventing the program from panicking at runtime.

```
-     x = exp.unwrap();
+     x = match exp {
+        Some(_) => { exp.unwrap() }
+        _ => { return Error }
+     };
```

Here, `exp` is a `Result<T, E>` or `Option<T>` type expression, and `Error` can be a user-defined error type specific to the corresponding crate. For generality, a placeholder (_) is used to catch all possible conditions.

FP 2. Replace with Combinator. In some cases, applying the match expression could be tedious, especially when the pattern matching only has two cases, i.e., `Some(T)` and `None`. In these cases, combinators [3, 23] can be used to manage control flow in a modular fashion. The built-in combinator functions are designed to simplify error handling by providing more concise and expressive alternatives to traditional control flow mechanisms. For `unwrap()` call, a commonly used combinator function is `unwrap_or()` to handle cases where unwrapping fails. Additionally, there is a corresponding set of variant functions designed for similar purposes. We divide this repair pattern into several sub-patterns, each corresponding to different application scenarios.

```
          -     let x = exp.unwrap()
FP 2.1    +     let x = exp.unwrap_or(VALUE)
FP 2.2    +     let x = exp.unwrap_or_default()
FP 2.3    +     let x = exp.unwrap_or_else(CLOSURE)
```

Here, `unwrap_or()` allows the developer to specify a default value to return if unwrapping fails, while `unwrap_or_default()` returns the default value for the type if it implements the `Default` trait. If the default value needs to be computed lazily, `unwrap_or_else()` can be used, taking a closure that generates the default value when needed. According to Rust official guide book [7], the `VALUE` is determined based on the data type. For integer types such as `i32` and `u32`, the default value is zero (`0`). For floating-point types such as `f32` and `f64`, the default value is zero point zero (`0.0`). For boolean types, the default value is false. For string slices (`&str`), the default value is an empty string (`""`). For `Option` types, the default value is `None`. For unit types (`()`), the default value is the unit value itself (`()`). For collection types such as `Vec` and `HashMap`, the default value is an empty collection.

3.2 Violation of Interior Mutability Rules

Rust ensures memory safety through the concepts of references and borrowing, with the core rule: at any given time, you can have either one mutable reference or any number of immutable references [15], and this is statically checked at compile time. However, these borrowing rules can sometimes be overly restrictive in certain use cases. For instance, consider caching mechanisms. The interface is typically designed to appear immutable to callers, providing read-only access to the requested data. Internally, however, the cache may need to perform mutations, such as updating access timestamps or inserting new entries when a cache miss occurs. These operations inherently require mutability, yet exposing a mutable interface to users would violate the abstraction of the cache as a read-only lookup mechanism.

To address the above issue, Rust provides a design pattern called interior mutability, which allows users to mutate data even when there are immutable references to that data. This pattern is implemented using Rust's smart pointers [14], such as `RefCell<T>`, which function like regular pointers but internally use unsafe code within a data structure to bypass Rust's usual rules of mutation and borrowing. The unsafe code signals to the compiler that these borrowing rules will be manually enforced, rather than relying on the compiler to check them. In single-threaded contexts, `RefCell<T>` enables mutation without the need for synchronization. It leverages Rust's lifetimes to allow dynamic borrowing, providing temporary, exclusive, mutable access to its inner value. The corresponding `borrow()` method grants an immutable reference (`&T`), while `borrow_mut()` allows for a mutable reference (`&mut T`). Both methods enforce Rust's borrowing rules at runtime, ensuring that mutable and immutable references do not coexist. However, violating the interior mutability rules by improperly invoking these borrowing methods leads to panics at runtime. It is important to emphasize that such panics occur specifically due to violations of these rules. In contrast, when regular references are used, any violation of borrowing rules will prevent compilation and will not trigger a panic. Panics can only occur in cases involving smart pointers, where manual borrowing checks are required.

```rust
1  use std::cell::RefCell;
2
3  fn main() {
4     let wrapper = RefCell::new(0);
5     let immutable_ref_1 = wrapper.borrow();
6     let immutable_ref_2 = wrapper.borrow(); // The program operates as expected
7     let mutable_ref = wrapper.borrow_mut();
8     // thread 'main' panicked at src/main.rs:7:32: already borrowed: BorrowMutError
9  }
10
11 fn violating_borrow_rules() {
12     let mut x = 0;
13     let immutable_ref = &x;
14     let mutable_ref = &mut x;
15     // error[0502]: cannot borrow `x` as mutable because it is also borrowed as immutable
16     println!("{}", immutable_ref);
17 }
```

Listing 3. An example of a Rust program with conflict between immutable and mutable references.

For example, in Listing 3, no panic occurs up to line 6. The variable `wrapper` is assigned as a `RefCell`, and two immutable shared references are created using the `borrow()` method, which is allowed under Rust's borrowing rules. However, on line 7, a panic is triggered because the

program attempts to create a mutable reference, `mutable_ref`, while the immutable references are still active. This violates the borrowing rules and causes a runtime panic, demonstrating that the program compiles successfully but panics during execution. In contrast, the function `violating_borrow_rules()` presents a typical example of a reference operation that violates the borrow rules. This function cannot pass compilation, and the Rust compiler will generate an appropriate error message, preventing it from running. To fix panics caused by violating the borrowing rules during runtime, two patterns are proposed to eliminate invalid references or choose a safer way to deal with possible violation.

FP 3. Enforce Ownership Rules. To adhere to the borrowing rules, within a block, this repair pattern involves replacing multiple mutable borrows with either an immutable borrow or removing one of the mutable borrows. Specifically, it can be subdivided into two repair modes as follows. FP3.1 indicates that when there is already an immutable reference and then a mutable reference, the mutable reference is replaced with an immutable reference. Similarly, FP3.2 states that if a mutable reference to the same variable has been used multiple times, only the first reference is retained. After removing the redundant `borrow_mut()` call, all references to the previously removed variable are replaced with the original, single reference.

```
FP 3.1          let x = a.borrow();
           -    let y = a.borrow_mut();
           +    let y = a.borrow();
FP 3.2          let x = a.borrow_mut();
           -    let y = a.borrow_mut();
           -    let tmp = *y
           +    let tmp = *x;
```

FP 4. Replace with Invariant. To enhance safety, non-panicking borrow methods for `RefCell<T>` were proposed in an early RFC [1]. These methods provide a safer alternative to the standard `borrow()` and `borrow_mut()` methods. Specifically, the `try_borrow()` and `try_borrow_mut()` methods attempt to borrow the inner value immutably or mutably, respectively, but without causing a panic if the borrowing rules are violated. Instead of panicking, these methods return a `Result`, allowing the programmer to handle the failure gracefully and maintain better control over borrowing logic. Furthermore, the fix pattern performs pattern matching on the returned `Result`, consistent with the logic of FP1. We divide this pattern into two sub-patterns, each corresponding to the methods for mutable and immutable borrowing.

```
FP 4.1     -    let x = a.borrow();
           +    let x = match a.try_borrow() {
           +              Ok(value) => value,
           +              Err(e) => { /* handle error */ },
           +            };
FP 4.2     -    let x = a.borrow_mut();
           +    let x = match a.try_borrow_mut() {
           +              Ok(value) => value,
           +              Err(e) => { /* handle error */ },
           +            };
```

3.3 Reuse of Ready Async Functions

Asynchronous programming in Rust is fundamentally based on the `Future` trait, which represents an asynchronous computation initiated by using the `async` keyword. The progress of a `future` is driven by the `poll()` method, which is used to attempt to resolve the `future` into a final state.

The `poll()` function returns one of two states: `Poll::Pending`, indicating that the computation is not yet complete and requires further progress, or `Poll::Ready`, signifying that the computation has successfully completed with a final result.

When a `future` returns `Poll::Pending`, it indicates that the computation is not yet complete and that progress is required. At this point, the `future` stores a clone of a `Waker` that is obtained from the current `Context`. The `Waker` is a key component of Rust's asynchronous model, enabling the task to be "woken up" when the `future` can make progress. This is accomplished by calling `Waker::wake`, which notifies the executor that the task is ready to continue. The executor then schedules the task again, and the `poll()` method is invoked to resume the computation. Once a `future` transitions to the `Poll::Ready` state, it signifies that the asynchronous operation has been successfully completed, and the `future` is in a terminal state. Any subsequent calls to the `poll()` method on this `future` are considered undefined behavior, as the `future` is no longer valid for polling. Typically, this results in a runtime panic, as no further progress can be made.

```rust
1  use futures::{Future, task::noop_waker};
2  use std::pin::Pin;
3  use std::task::{Context, Poll};
4
5  async fn example_future() -> i32 { 42 }
6
7  fn main() {
8      let mut future = Box::pin(example_future());
9      let waker = noop_waker();
10     let mut cx = Context::from_waker(&waker);
11
12     let result = Pin::new(&mut future).poll(&mut cx); // First poll
13     if let Poll::Ready(_) = result { /* ... */ }
14     let _ = Pin::new(&mut future).poll(&mut cx); // Second Poll
15     // thread 'main' panicked at src/main.rs:5:34: `async fn` resumed after completion
16 }
```

Listing 4. An example of a Rust program with reuse of ready async function, leading to a panic.

In Listing 4, the function `example_future()` is marked with the `async` keyword, indicating that calling it will return a `future` that represents an asynchronous computation. To interact with this `future`, in line 8-10, we provide a `Context` object, which includes a `Waker` that controls when the task should be scheduled to continue. In this case, the `Context` object, `cx`, is initialized by a built-in no-operation Waker `noop_waker()`, which effectively does nothing but satisfies the requirement for a `Waker` in the `poll()` method. This code snippet runs without issues up until line 13, where the `poll()` method is called on the `future`. At this point, the `poll()` method successfully advances the `future` to the `Poll::Ready` state, signaling that the asynchronous computation has completed and produced a result, 42. However, the second attempt to invoke `poll()` on the same `future` in line 14 will cause a panic. This occurs because once a `future` has transitioned to the `Poll::Ready` state, it is considered terminal and any further calls to `poll()` on the same `future` are undefined behavior. Typically, this results in a runtime panic, as the `future` is no longer valid for polling. We propose two fixing patterns to address such type of panic errors.

FP 5. Fuse Future to Prevent Panic. Rust provides the `fuse()` method to avoid immediate panics caused by multiple calls to `poll()` on a completed future. When it is anticipated that a future might be polled multiple times, `fuse()` can be used to ensure predictable behavior. For a fused future, subsequent calls to `poll()` after it has returned `Poll::Ready` will result in `Poll::Pending` rather

than causing undefined behavior or a runtime panic. By applying `fuse()` to the future beforehand, we can gracefully handle repeated polling.

```
-     let result = Pin::new(&mut future).poll(cx);
+     let mut fused_future = future.fuse();
+     let result = Pin::new(&mut fused_future).poll(cx);
```

FP 6. Postpone Error Propagation. The `ready!()` macro is commonly used to simplify the extraction of return values from repeated calls to `poll()`. It checks if the result is in the `Poll::Ready` state, extracting the value if it is. If the result is not ready, the macro ensures that the current function returns `Poll::Pending`, thus postponing further execution until the value becomes `Poll::Ready`. This macro is typically used when implementing the `Future` trait for custom types. After extracting the result, a state change is usually performed to ensure the future transitions correctly and remains consistent. However, while `ready!()` simplifies handling asynchronous results, issues can arise if an error is propagated using the ? operator within it. In such cases, state modifications may be skipped, leaving the future in an inconsistent state, which can cause potential issues in further execution. To avoid this, it is recommended to separate the extraction and error propagation steps. This ensures that any necessary state modifications are executed before errors are propagated.

```
-     let x = ready!(exp?);
+     let tmp = ready!(exp);
      /* state change */
+     let x = tmp?;
```

3.4 Arithmetic Overflow

In Rust, an unsigned integer can have one of the following types: `u8`, `u16`, `u32`, `u64`, `u128` and `usize`. The type denotes the number of bits used to store the integer, e.g., `u8` can hold values between 0 and 255. If an integer is assigned a value outside of its defined range, an overflow or underflow will occur. As demonstrated in the code snippet in Listing 5, the variable `value` is of type `u8`. Consequently, if the value exceeds 255, a panic will be triggered due to overflow.

Arithmetic overflow exhibits some peculiarities in Rust. In release or optimized mode, Rust silently ignores overflow by default, performing two's complement wrapping [5]. For example, when adding 1 to 255 in an `u8` integer, the result wraps around to 0. However, in debug mode, Rust introduces built-in checks for overflow and triggers a panic when an overflow occurs at runtime. This behavior contrasts with languages like C and C++, where arithmetic overflow is treated as undefined behavior, potentially leading to unpredictable results or security vulnerabilities. In comparison, Java truncates overflowed values by default, similar to the behavior in Rust's release mode. Given the unique design mechanisms of Rust, its standard library also provides safer computation APIs, which can be applied as fix patterns for these panics.

```
1 fn main() {
2     let mut value: u8 = 250; // Maximum value for u8 is 255
3     for _ in 0..10 { value += 10; }
4     // thread 'main' panicked at src/main.rs:3:22: attempt to add with overflow
5 }
```

Listing 5. An example of a Rust program with arithmetic overflow, leading to a panic.

FP 7. Mutate Arithmetic Operator. Rust provides explicit and safer ways for developers to handle potential overflows through built-in methods like `wrapping_add()`, `saturating_add()`, and

`checked_add()`, providing different additional processing when encountering boundary conditions. The `wrapping_op()` function wraps around the value on overflow, mimicking C/C++'s modular arithmetic behavior, making it suitable for scenarios like cyclic counters or ring buffers [24]. The `saturating_op()` function [20], on the other hand, clamps the result to the type's maximum or minimum value, which is ideal for cases requiring boundary enforcement, such as preventing values from exceeding physical limits. Finally, the `checked_op()` function [2] returns an `Option` type, yielding `None` on overflow, making it well-suited for cases where overflow must be explicitly handled or reported.

```
        -    a op b
FP 7.1  +    a.wrapping_op(b)
FP 7.2  +    a.saturating_op(b)
FP 7.3  +    a.checked_op(b).unwrap()
        +    a.checked_op(b).unwrap_or_default()
        +    a.checked_op(b).unwrap_or_else(|| fn())
```

3.5 Invalid String Boundary

Rust's String type is byte-indexed rather than character-indexed, a design choice made primarily for performance reasons [6]. This approach ensures that operations like indexing and slicing are efficient, as they directly manipulate the underlying UTF-8 byte array without requiring costly decoding or validation of character boundaries. However, this design also introduces potential pitfalls when handling non-ASCII input, since Unicode characters may span multiple bytes. In Rust, attempting to slice a `String` at a position within the middle of a multi-byte character is not allowed and will result in a runtime panic. This behavior contrasts sharply with languages like C and C++, where strings are often treated as null-terminated byte arrays, or Java, which uses UTF-16 encoding. In these languages, slicing strings at arbitrary positions typically does not produce an immediate error, but may lead to undefined or unexpected behavior, such as generating invalid strings or partial characters [21].

```rust
1 fn main() {
2     let text = String::from("Rust™ Programming");
3     let valid = 7; // Start of 'Programming'
4     let invalid = 5; // Splits inside the '™' character
5
6     let valid_slice = &text[valid..];  // The program operates as expected
7     let invalid_slice = &text[invalid..];
8     // thread 'main' panicked at src/main.rs:7:30: byte index 5 is not a char boundary
9 }
```

Listing 6. An example of a Rust program with invalid slice of `String`, leading to a panic.

For example, as shown in the code snippet in Listing 6, the variable `text` is assigned a string containing non-ASCII characters. In lines 6 and 7, we attempt to take slices that span both valid and invalid parts of the string. However, line 7 will cause the program to panic, as Rust is unable to properly handle the invalid string boundary, which violates its strict UTF-8 encoding rules. To fix panic bugs caused by this type of issue, the most common strategy is to validate the `String` and retain only the valid characters.

FP 8. Insert Char Matcher. A general fix strategy is to insert a match checker that iterates over the bytes of the `String`. The checker performs a validation step, examining each byte to ensure it conforms to the expected encoding rules, such as UTF-8. If the checker fails to retrieve a specified

byte due to an invalid boundary or an encoding error, it returns an empty `String`, effectively preventing further processing with corrupted data. If the validation is successful and all bytes are correctly retrieved, the function proceeds as usual, retaining the original behavior of the program. This fix pattern ensures that only valid data is processed, avoiding potential panics due to invalid string boundaries.

```
-     let _ = &data[index..];
+     let _ = match data.char_indices().nth(index) {
+        Some((offset, _)) => &data[offset..],
+        None => return String::new(),
+     };
```

3.6 Unexpected Edge Cases

Rust, like many other programming languages, is not immune to common runtime errors, such as index out-of-bounds access and division by zero. Listing 7 demonstrates two panic errors caused by unexpected edge cases. The function `get()` takes an index `idx` as input and attempts to retrieve the corresponding element from a vector `array` initialized with three elements. However, on line 2, the index 3 is passed to `get()`, which is out of the valid range, triggering a runtime panic error. Another runtime panic occurs due to division by zero. The function `get_zero()` returns the value `0`. However, on line 5, the number `10000` is divided by the result of `get_zero()`, which causes a runtime panic because division by zero is undefined. The strategy for fixing panics caused by these types of unexpected edge cases is similar to the corresponding fix patterns in other programming languages, which we summarize below.

```
1 fn get(idx: usize) -> i32 { let array = vec![1, 2, 3]; array[idx] }
2 fn out_of_bound() { get(3); }  //PANIC: index out of bounds: the len is 3 but the index is 3
3
4 fn get_zero() -> i32 { 0 }
5 fn division_by_zero() { let _ = 10000 / get_zero(); }  // PANIC: attempt to divide by zero
```

Listing 7. Two examples of Rust programs with unexpected edge cases, leading to a panic.

FP 9. Check Array Bound.

An `if` statement can be used to check whether the start or end index of a slice or array exceeds its length before accessing it. The subpatterns FP 9.1, FP 9.3, FP 9.4, and FP 9.5 all follow this handling logic: if the index is out of bounds, an error is returned to prevent a runtime panic. Additionally, FP 9.2 implements a subpattern in which, if an index exceeds the array's length, it can be adjusted by subtracting the excess to bring it within valid bounds. However, it is important to note that while modifying the index value can prevent a panic bug, it may also change the intended semantics of the program. Therefore, when applying FP 9.2, the surrounding code logic must be considered to ensure semantic consistency.

```
FP 9.1   -     x = array[index]
         +     if index > array.len() { return Error }
FP 9.2   +     x = array[index_new]
         -     x = array[start..end]
FP 9.3   +     if start > end { return Error }
FP 9.4   +     if start > array.len() { return Error }
FP 9.5   +     if end > array.len() { return Error }
```

FP 10. Modify Division Operation. Division operations can be modified to explicitly handle division-by-zero by checking the divisor with a condition or adjusting it to ensure it is not zero.

```
FP 10.1   -     let x = a / b;
          +     let x = a / b.max(1)
FP 10.2   +     if b == 0 { return Error }
                let x = a / b;
```

3.7 Logical Panics

In Rust, developers can manually trigger panics in several scenarios to handle exceptional conditions or to enforce assumptions during runtime. Listing 8 demonstrates common causes of panics, including assertion failures, the use of the `unreachable!()` macro, and the explicit invocation of the `panic!()` macro. These mechanisms allow developers to halt execution when the program encounters an unexpected state, ensuring that such conditions are handled appropriately.

```
1 fn assert_panic() { assert!(10000 == 10001); }  // PANIC: assertion failed: 10000 == 10001
2 fn unreachable_panic() { unreachable!(); }  // PANIC: internal error: entered unreachable code
3 fn macro_panic() { panic!("User-defined panic!");}  // PANIC: User-defined panic!
```

Listing 8. An example of a Rust program with different macros causing a panic.

These panics are typically tied to logical issues that developers are aware of during development. They often reflect the developer's intent to enforce invariants, signal incomplete functionality, or highlight known but unhandled cases in the code. For example, an assertion failure might ensure critical assumptions hold true, while the use of `unreachable!()` could indicate paths that are logically impossible under current conditions. Such panics may also act as placeholders for future functionality, marking areas where additional logic or features are expected to be added later. Alternatively, they can serve as safeguards for anticipated edge cases that are temporarily left unhandled due to prioritization or limitations during initial implementation. For these specific cases where developers intentionally trigger panics, we do not provide a fix pattern, as they are not considered bugs, and attempting to eliminate them may alter the intended semantics of the code.

4 AUTOMATED PANIC BUGS FIXING TOOL

As the first infrastructure aimed at addressing Rust panic bugs, we propose a pattern-based automated tool for fixing such bugs. The proposed tool, namely PanicKiller, leverages Panic4R and mined fix patterns to demonstrate the practical application of PanicFI. While numerous innovative APR techniques have been proposed, especially those based on large language models [49, 92], the primary objective of PanicKiller is to validate the effectiveness of our mined fix patterns. Accordingly, PanicKiller focuses on the iterative matching of suitable fix patterns, prioritization, and the generation of patches. The integration of additional techniques, such as static analysis and large language models, to fully leverage the potential of fix patterns represents a direction for future research.

4.1 Tool Design

The workflow of PanicKiller can be divided into three core components. First, PanicKiller matches fix patterns based on the stack trace information provided by the compiler and the dependency flow analysis of the original program. Then, utilizing the semantic information of the errors, PanicKiller generates a series of patches with the available fix patterns. Finally, PanicKiller ranks all generated

patches by calculating verification and matching scores, outputting the top-5 ranked patches along with their corresponding confidence scores.

Pattern matching. When a panic occurs, Rust initiates the unwinding process, tracing back through the stack to ensure proper cleanup. The error message is supplemented with stack trace information, from which PanicKiller automatically extracts suspicious locations, including file paths, column numbers, and row numbers.

For each candidate code element, PanicKiller iteratively employs the mined fix patterns. To facilitate this process, PanicKiller leverages *syn* [34] to parse the source code into its corresponding AST. If the corresponding AST has a structure that matches a specific pattern, it is applied to generate a patch; otherwise, it is discarded. AST transformations are applied directly to the parsed tree, enabling PanicKiller to perform fine-grained rewrites of Rust code while preserving both syntactic validity and semantic consistency. These transformations include expression replacements, rewrites of conditional expressions, and structural modifications based on matched patterns. The transformed AST is then converted back into source code using the *prettyplease* [13] crate, which produces readable and properly formatted output. Note that even after a pattern has been applied to an expression, additional patterns might be identified as one delves deeper into the AST. Consequently, a suspicious location may correspond to more than one resulting patch.

Patch validation. For the APR task, it is important to ensure that generated patches not only fix the bug but also do not alter the original program semantics. In our work, we utilize *cargo-test*, a built-in tool in Rust's package manager, to automatically execute the complete test suite included within each crate's source code. This allows us to comprehensively verify that patches pass the complete set of regression tests provided by the projects themselves. After performing the validation, the patches would have several cases: (1) The panic is eliminated and all the test cases execute with the same result as before. Our goal is to generate these patches, which indicate the most likely correct fix. (2) The panic is eliminated but the execution results of some test cases are not consistent with the original program. This suggests that the patch may have introduced logical modifications that cause the semantic inconsistency. (3) The panic is not eliminated and we don't evaluate the regression testing because it's not a correct fix. By regression testing, we can effectively filter out patches that might introduce new errors or exhibit overfitting to test suites, thereby enhancing the reliability and robustness of fixing.

Patch prioritization. During the pattern matching process, PanicKiller may match multiple code locations and fix patterns, resulting in a set of candidate patches. However, the exact faulty code element may not explicitly indicated in the panic stack trace. Prior studies suggest that the actual bug is frequently located in expressions that are structurally or semantically related to the code elements in the stack trace [80]. Consequently, the effectiveness of a fix pattern depends not only on the elements explicitly reported but also on their surrounding context and dependencies.

To better capture dependencies among code elements and assess the likelihood that a candidate patch addresses the fault, we construct a Code Element Dependency Graph (CEDG) [40] using Rust's High-level Intermediate Representation (HIR) [33], an intermediate form generated during compilation. Specifically, we leverage the official Rust compiler crate `rustc_hir` [18] to extract the HIR of the given code snippet, which provides rich structural information such as types, variable bindings, and source code locations. Based on this representation, we systematically capture dependencies and interactions among code elements. For instance, in assignment statements or expressions, variables on the left-hand side are considered dependent on the values on the right-hand side. In the case of function invocations, dependencies may exist among parameters or between the caller and callee, prompting us to further analyze function bodies to uncover deeper relationships. Exploiting the transitive nature of these dependencies, we iteratively construct the

CEDG to reflect both direct and indirect influences between code elements. Based on this graph, we then define a confidence score for each code element e_i as follows:

$$Con(e_i) = 1 - \frac{\min(Dist(\text{stacktrace}, e_i), \lambda)}{\lambda}, \quad \lambda \geq 1 \tag{1}$$

where $Dist(\text{stacktrace}, e_i)$ denotes the shortest distance between element e_i and any element in the stack trace on the dependency graph. A constant λ is used to normalize the confidence score, ensuring it is scaled within a meaningful range. In our work, we set $\lambda = 2$ according to an existing study [70], which has proved that it has the optimal performance. By incorporating both the precise locations in the stack trace and their dependent code elements via the CEDG, our approach enables a more comprehensive and fine-grained fault localization.

In addition to the confidence score, the presence of suspicious files within the stack trace can also influences the localization of code elements where fix patterns may be applied. On the one hand, files that frequently appear in stack traces tend to be more closely involved in the execution path leading to the error, and are therefore likely indicators of the fault location. On the other hand, in an expanded stack trace, code elements located at shallower depths are typically closer to the root cause of the error [91]. Thus, PanicKiller computes the suspicion score for each code element e_i as follows:

$$S(e_i) = N \times (1/D(e_i) + Con(e_i)) \tag{2}$$

where N denotes the number of times the file containing e_i appears in the stack trace, and D_i represents the depth of its location within the stack trace sequence. For each buggy program and its corresponding stack trace, PanicKiller constructs a dependency graph and calculates a score for each code element e_i. This score reflects the likelihood that e_i is a suitable candidate for applying fix patterns. Based on these scores, the suspect code elements are prioritized to guide the repair process.

For all the generated patches, PanicKiller combines the scores and validation results mentioned above to perform the patch prioritization. Specifically, patch prioritization is guided by two factors: (1) the suspicious score for each code element $S(e_i)$ and (2) the results from the regression test validation. PanicKiller first prioritizes each patch based on the calculated confidence score. For patches that achieve identical scores, preference is given to those that successfully pass regression testing. Finally, PanicKiller outputs a top-5 ranked list of fixing recommendations.

A running example. Figure 2 (a) presents a code example that triggers a panic bug caused by an index out-of-bounds error. Assuming that PanicKiller has identified lines 1 and 4 as suspicious locations, with confidence scores of 0.7 and 1, respectively, as determined by Equation 1. Subsequently, PanicKiller iteratively applies each pattern to these locations. For line 1, PanicKiller identifies a method call of `unwrap()`, thus the *Insert Explicit Matcher* pattern is applied, and *patch 1* is depicted in Figure 2 (b). For line 4, PanicKiller initially identifies an index expression, and the *Check Array Bound* pattern is employed, as illustrated as *patch 3* in Figure 2 (d). Further, PanicKiller analyzes the deeper structure and identifies a binary expression in the index of array, for which the *Mutate Arithmetic Operator* pattern is applied to `num2 - num3`. *Patch 2* is depicted in Figure 2 (c). The prioritization of patches is shown in Figure 2 (e). *Patch 3* ranks first with the highest score of 1.7, having passed the regression tests. *Patch 2* is second because, although it scored the same as *patch 1*, it passed the regression tests while *patch 1* failed. Since patch prioritization is closely tied to the outcomes of validation, it is important to understand why *patch 1* fails. While the inserted explicit matcher prevents the panic from the `unwrap()` call at line 1, it does not resolve the underlying out-of-bounds access at line 4. Consequently, the program still panics during the array indexing operation, leading to failed validation. By contrast, *patch 2* and *patch 3* both eliminate the root

```
1 arr[num1] = a.unwrap(); // patch 1
2 // ...
3 // index out of bounds panic!
4 if arr[num2 - num3] > 0 { // patch 2&3
5     return 0;
6 }
```

(a) The panic bug-triggering example.

```
- arr[num1] = a.unwrap();
+ arr[num1] = match a.unwrap() {
+     Some(_) => a.unwrap(),
+     _ => return Error, };
// ...
if arr[num2 - num3] > 0 { return 0; }
```

(b) Patch 1: Insert Explicit Matcher. (FP 1)

```
arr[num1] = a.unwrap();
// ...
- if arr[num2 - num3] > 0 {
+ if arr[num2.saturating_sub(num3)] > 0 {
    return 0;
}
```

(c) Patch 2: Mutate Arithmetic Operator. (FP 7.2)

```
arr[num1] = a.unwrap();
// ...
+ if num2 - num3 > arr.len() {
+     return Error;
+ }
if arr[num2 - num3] > 0 { return 0; }
```

(d) Patch 3: Check Array Bound. (FP 9.1)

Patch	Fix Pattern	Con(e_i)	S(e_i)	Validation	Rank
Patch 1	Insert Explicit Matcher	0.7	1.1	Test fail	3
Patch 2	Mutate Arithmetic Operator	1.0	1.1	Test pass	2
Patch 3	Check Array Bound	1.0	1.7	Test pass	1

(e) Prioritization results for the three patches.

Fig. 2. A panic bug-triggering example and its corresponding patches.

cause of the panic and therefore pass all regression tests. Consequently, PanicKiller outputs this ranked patch list with their interpretations, organized by scores and test results.

4.2 Tool Evaluation

In this section, we present the evaluation and experiment result for PanicKiller. We conduct experiments on Panic4R and categorize the real data in Panic4R into small-scale datasets and large-scale datasets based on the size of the source code. Specifically, projects with fewer than 20K lines of code (LoC) are considered small-scale, while those with 20K LoC or more are considered large-scale. We first evaluate the fault localization accuracy of all tools. Next, we report the effectiveness of different tools in fixing panics, using standard accuracy metrics at the top-1, top-3, and top-5 levels. We then conduct a comparative analysis of the patches generated by PanicKiller and baseline tools, highlighting their respective strengths and weaknesses through representative examples. Finally, we perform a case study by applying PanicKiller to four real-world Rust crates that are not included in the Panic4R benchmark, demonstrating its applicability beyond the evaluation set.

Baselines. To the best of our knowledge, no dedicated tool exists currently for the automated repair of Rust panic bugs. We are aware that Rust-lancet [96] was proposed to fix errors in Rust programs that violate ownership rules and Ruxanne [78] has constructed a series of fixing patterns related to bugs caused by the borrow checker. However, these bugs occur at compile time, indicating programs that violate ownership rules or fail to pass borrow checker cannot pass compilation. In contrast, panic bugs occur at runtime, implying that the program has already passed compilation. Therefore, Rust-lancet and Ruxanne are not suitable for fixing panic bugs. In this paper, we primarily compare the effectiveness of PanicKiller with LLM-based approaches for addressing panic bugs. We choose two categories of state-of-art LLM-based tools for evaluation:

(1) **Large Language Models (LLMs)**: We include ChatGPT-4.0 (GPT4.0), one of the most widely adopted commercial chat models. Additionally, we reference the performance of Rust-related tasks from the Big Code Models Leaderboard [45], which focuses on base multilingual code

generation models. From the subset of models supporting Rust tasks, we select two top-performing instruction-tuned candidates: Qwen2.5-Coder-32B-Instruct (Qwen) [77] and OpenCodeInterpreter-DS-33B (OCI) [68].

(2) **LLM-based Automated Program Repair (APR) Tools**: We consider Rust Assistant (Assist) [47], the leading APR tool specifically designed for fixing Rust compilation errors. To the best of our knowledge, it remains the only LLM-based APR tool tailored for the Rust programming language. Additionally, we include SWE-agent [95], the state-of-art agent-based tool for software engineering tasks, including automatic program repair.

For LLMs, we upload the entire Rust project, the test case triggering the panic, and relevant panic information. For APR tools, we adhere strictly to their respective usage guidelines. Specifically, for Rust Assistant, we employ its original prompt template, as the underlying source code has not been open-sourced. For SWE-agent, we collect the relevant issue URLs corresponding to each case in PanicKiller and supply them to the agent to initiate its automated repair process, using its default model, Claude 3.7 Sonnet [36].

We use a consistent prompt template for LLMs, as shown below:

> I have uploaded a source code package of a Rust crate **[crate-name]**, **[crate-zip]**. When using this crate with the following test case: **[main.rs]**, it went panic, the panic information is as follows: **[panic_info]**.
> (The error location is: **[perfect-location]**.) Please provide the top 5 **fixing code** to fix this panic. Don't explain anything about code, just show me the patches.

In this prompt template, the content within all square brackets is filled in based on the actual data from Panic4R. The phrase within the brackets is optional, as we conducted experiments both with and without the provision of perfect locations, and the corresponding prompt templates reflect this distinction. Because LLMs may fail to produce repair patches as instructed, we employ a baseline involving multiple interactions with LLMs. Initially, we assess LLMs' output for completeness and iterate the query process up to three times, correcting for any missing information. If, after three attempts, the generated patches remain unattained, we classify the effort as a repair failure of the LLM. The inquiry prompts for missing information are as follows:

> Based on the mislocalization information you provided, please provide me with the specific code to fix it.

Evaluation Metric. We assess fault localization effectiveness by verifying whether the faulty statement appears in the final output of each tool.

Referring to existing research on APR [75], we divide the generated patches into three categorization and verify their proportion:

(1) Panic-eliminated patches: the panic bug is resolved but some regression test cases fail;
(2) Plausible patches: the panic bug is successfully fixed and all test cases pass;
(3) Correct patches: plausible patches that have been manually verified for semantic correctness.

4.2.1 Effectiveness of Fault Localization. We firstly assess the fault localization capabilities of all baseline methods, with results summarized in Table 4. Overall, PanicKiller consistently outperforms all baseline tools in statement-level fault localization. As the size of the crates increases, the success rate of localization decreases across all the tools. For LLMs, this decline can be attributed to the growing complexity of larger codebases, which poses challenges for understanding and reasoning. For LLM-based APR tools, Rust Assistant yields the worst performance, primarily because its prompt design lacks consideration for explicit localization guidance. In contrast, SWE-agent achieves better results, benefiting from its agent-based mechanism that iteratively explores the project and infers inter-file relationships.

Table 4. The accuracy of statement-level fault localization of different tools on Panic4R.

Tool	Panic4R-Small (61)		Panic4R-Large (41)		Total (102)	
GPT4.0	30	49.2%	4	9.8%	34	33.3%
GPT4.0-multi	30	49.2%	4	9.8%	34	33.3%
Qwen	29	47.5%	13	31.7%	50	49.0%
Qwen-multi	30	49.2%	13	31.7%	51	50.0%
OCI	21	34.4%	5	12.2%	26	25.5%
OCI-multi	21	34.4%	5	12.2%	26	25.5%
Assist	8	13.1%	3	7.3%	11	10.8%
Assist-multi	8	13.1%	3	7.3%	11	10.8%
SWE-agent	26	42.6%	10	24.4%	36	35.3%
PanicKiller	**39**	**62.3%**	**20**	**48.8%**	**59**	**57.8%**

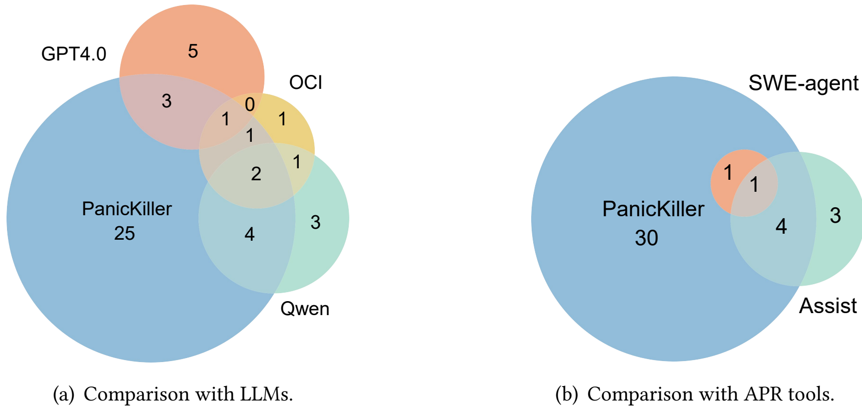
4.2.2 Effectiveness of Panic Fixing. To minimize the impact of inaccuracies in the localization of code elements, we conducted two sets of experiments. The first set focuses on identifying the code element method used by PanicKiller, which is based on dependency relationships. The second set establishes the perfect localization using information from Panic4R. Notably, SWE-agent only requires an issue URL and a repository URL as input, and does not support incorporating additional location information. As a result, it is excluded from the second set of experiments involving perfect localization. We then calculate the proportion of patches generated by all baseline methods, evaluating them using three different metrics on the fixing effectiveness.

The results for cases where the perfect localization is not provided are presented in Table 5. Overall, PanicKiller demonstrates a clear advantage over baseline tools in repairing panics. This superiority stems from its ability to analyze code dependencies and perform cross-file pattern matching, making it especially effective in large-scale projects. The limited effectiveness of LLMs in this setting is primarily due to their insufficient fault localization capability. Without accurately identifying the faulty code regions, it often generates non-compilable or contextually irrelevant patches, which constrains its ability to perform reliable repairs. SWE-agent shows the weakest repair performance among all baseline tools. This is primarily due to its reliance on general-purpose prompting strategies, which lack awareness of Rust-specific language features and runtime behaviors. Another contributing factor is the incomplete execution environment, as `rustc` is occasionally unavailable during SWE-agent's iterative prompting. This limits the quality of feedback and further hinders its repair performance.

As shown in Table 6, even with perfect location information, PanicKiller consistently outperforms other baseline approaches. This indicates that PanicKiller's patch generation capability is superior to all the baseline tools, even after multiple iterations. Compared with the results in Table 5, PanicKiller only slightly improves its fixing effectiveness with perfect location information. While it does generate new plausible or correct patches, it sometimes fails to recreate some of its previous patches without perfect locations. This is because some patches were applied at related but different locations, such as within a method call hierarchy, and still effectively fixed the issue. However, the provided location may interfere with the selection of fix patterns. As for the baseline tools, even when provided with perfect fault localization, they may still fail to generate correct patches due to the lack of awareness of repair patterns tailored to panic-related failures in Rust. For instance, Rust Assistant focuses primarily on fixing compilation errors, and thus may not generalize well to runtime panics. Another limitation lies in the syntactic correctness of generated patches. LLM-based approaches often overlook Rust's strict syntactic and typing constraints, leading to patches that fail to compile, as illustrated by a detailed example in Section 4.2.3.

Table 5. The number of panic-eliminated/plausible/correct patches of different tools on Panic4R.

	Tool	Panic4R-Small (61) Eliminated		Plausible		Correct		Panic4R-Large (41) Eliminated		Plausible		Correct		Total (102) Eliminated		Plausible		Correct	
Top-1	GPT4.0	6	9.8%	6	9.8%	0	0%	0	0%	0	0%	0	0%	6	5.9%	6	5.9%	0	0%
	GPT4.0-multi	8	13.1%	8	13.1%	0	0%	0	0%	0	0%	0	0%	8	7.8%	8	7.8%	0	0%
	Qwen	5	8.2%	5	8.2%	2	3.3%	2	4.9%	0	0%	0	0%	7	6.9%	5	4.9%	2	2.0%
	Qwen-multi	8	13.1%	7	11.5%	2	3.3%	2	4.9%	0	0%	0	0%	10	9.8%	7	6.9%	2	2.0%
	OCI	5	8.2%	3	4.9%	0	0%	0	0%	0	0%	0	0%	5	4.9%	3	2.9%	0	0%
	OCI-multi	5	8.2%	3	4.9%	0	0%	0	0%	0	0%	0	0%	5	4.9%	3	2.9%	0	0%
	Assist	7	11.5%	7	11.5%	5	8.2%	2	4.9%	2	4.9%	2	4.9%	9	8.8%	9	8.8%	7	6.9%
	Assist-multi	7	11.5%	7	11.5%	5	8.2%	2	4.9%	2	4.9%	2	4.9%	9	8.8%	9	8.8%	7	6.9%
	SWE-agent	1	1.6%	0	0%	0	0%	2	4.9%	2	4.9%	2	4.9%	3	2.9%	2	2.0%	2	2.0%
	PanicKiller	**28**	**45.9%**	**18**	**29.5%**	**4**	**6.6%**	**10**	**24.4%**	**7**	**17.1%**	**4**	**9.8%**	**38**	**37.3%**	**25**	**24.5%**	**8**	**7.8%**
Top-3	GPT4.0	6	9.8%	6	9.8%	1	1.6%	0	0%	0	0%	0	0%	6	5.9%	6	5.9%	1	1.0%
	GPT4.0-multi	8	13.1%	8	13.1%	1	1.6%	0	0%	0	0%	0	0%	8	7.8%	8	7.8%	1	1.0%
	Qwen	9	14.8%	8	13.1%	2	3.3%	3	7.3%	0	0%	0	0%	12	11.8%	8	7.8%	2	2.0%
	Qwen-multi	12	19.7%	10	16.4%	2	3.3%	3	7.3%	0	0%	0	0%	15	14.7%	10	9.8%	2	2.0%
	OCI	5	8.2%	3	4.9%	0	0%	0	0%	0	0%	0	0%	5	4.9%	3	2.9%	0	0%
	OCI-multi	5	8.2%	3	4.9%	0	0%	0	0%	0	0%	0	0%	5	4.9%	3	2.9%	0	0%
	Assist	7	11.5%	7	11.5%	5	8.2%	2	4.9%	2	4.9%	2	4.9%	9	8.8%	9	8.8%	7	6.9%
	Assist-multi	7	11.5%	7	11.5%	5	8.2%	2	4.9%	2	4.9%	2	4.9%	9	8.8%	9	8.8%	7	6.9%
	PanicKiller	**31**	**50.8%**	**22**	**36.1%**	**12**	**19.7%**	**13**	**31.7%**	**9**	**22.0%**	**7**	**17.1%**	**44**	**43.1%**	**31**	**30.4%**	**19**	**18.6%**
Top-5	GPT4.0	7	11.5%	7	11.5%	1	1.6%	0	0%	0	0%	0	0%	7	6.9%	7	6.9%	1	1.0%
	GPT4.0-multi	9	14.8%	9	14.8%	1	1.6%	0	0%	0	0%	0	0%	9	8.8%	9	8.8%	1	1.0%
	Qwen	9	14.8%	9	14.8%	2	3.3%	3	7.3%	0	0%	0	0%	12	11.8%	9	8.8%	2	2.0%
	Qwen-multi	12	19.7%	11	18.0%	2	3.3%	3	7.3%	0	0%	0	0%	15	14.7%	11	10.8%	2	2.0%
	OCI	5	8.2%	3	4.9%	0	0%	0	0%	0	0%	0	0%	5	4.9%	3	2.9%	0	0%
	OCI-multi	5	8.2%	3	4.9%	0	0%	0	0%	0	0%	0	0%	5	4.9%	3	2.9%	0	0%
	Assist	7	11.5%	7	11.5%	5	8.2%	2	4.9%	2	4.9%	2	4.9%	9	8.8%	9	8.8%	7	6.9%
	Assist-multi	7	11.5%	7	11.5%	5	8.2%	2	4.9%	2	4.9%	2	4.9%	9	8.8%	9	8.8%	7	6.9%
	PanicKiller	**34**	**55.7%**	**24**	**39.3%**	**12**	**19.7%**	**13**	**31.7%**	**9**	**22.0%**	**7**	**17.1%**	**47**	**46.1%**	**35**	**32.4%**	**19**	**18.6%**



(a) Comparison with LLMs.

(b) Comparison with APR tools.

Fig. 3. Comparison across PanicKiller and baseline tools.

4.2.3 *Comparison with Baselines.*

To better understand the strengths and limitations of each method, we analyze all successfully repaired cases with plausible patches generated by PanicKiller

Table 6. The number of panic-eliminated/plausible/correct patches of different tools on Panic4R with perfect locations.

	Tool	Panic4R-Small (61)						Panic4R-Large (41)						Total (102)					
		Eliminated		Plausible		Correct		Eliminated		Plausible		Correct		Eliminated		Plausible		Correct	
Top-1	GPT4.0	6	9.8%	4	6.6%	4	6.6%	3	7.3%	3	7.3%	3	7.3%	9	8.8%	7	6.9%	7	6.9%
	GPT4.0-multi	6	9.8%	4	6.6%	4	6.6%	3	7.3%	3	7.3%	3	7.3%	9	8.8%	7	6.9%	7	6.9%
	Qwen	9	14.8%	7	11.5%	2	3.3%	2	4.9%	0	0%	0	0%	11	10.8%	7	6.9%	2	2.0%
	Qwen-multi	10	16.4%	7	11.5%	2	3.3%	2	4.9%	0	0%	0	0%	12	11.8%	7	6.9%	2	2.0%
	OCI	5	8.2%	5	8.2%	0	0%	1	2.4%	1	2.4%	0	0%	6	5.9%	6	5.9%	0	0%
	OCI-multi	5	8.2%	5	8.2%	0	0%	1	2.4%	1	2.4%	0	0%	6	5.9%	6	5.9%	0	0%
	Assist	8	13.1%	6	9.8%	4	6.6%	2	4.9%	2	4.9%	2	4.9%	10	9.8%	8	7.8%	6	5.9%
	Assist-multi	8	13.1%	6	9.8%	4	6.6%	2	4.9%	2	4.9%	2	4.9%	10	9.8%	8	7.8%	6	5.9%
	PanicKiller	**31**	**50.8%**	**23**	**37.7%**	**7**	**11.5%**	**10**	**24.4%**	**7**	**17.1%**	**4**	**9.8%**	**41**	**40.2%**	**30**	**29.4%**	**11**	**10.8%**
Top-3	GPT4.0	6	9.8%	5	8.2%	4	6.6%	3	7.3%	3	7.3%	3	7.3%	9	8.8%	8	7.8%	7	6.9%
	GPT4.0-multi	6	9.8%	5	8.2%	4	6.6%	3	7.3%	3	7.3%	3	7.3%	9	8.8%	8	7.8%	7	6.9%
	Qwen	11	18.0%	8	13.1%	3	4.9%	6	14.6%	0	0%	0	0%	17	16.7%	8	7.8%	3	2.9%
	Qwen-multi	12	19.7%	9	14.8%	3	4.9%	6	14.6%	0	0%	0	0%	18	17.6%	9	8.8%	3	2.9%
	OCI	5	8.2%	5	8.2%	0	0%	1	2.4%	1	2.4%	0	0%	6	5.9%	6	5.9%	0	0%
	OCI-multi	5	8.2%	5	8.2%	0	0%	1	2.4%	1	2.4%	0	0%	6	5.9%	6	5.9%	0	0%
	Assist	8	13.1%	6	9.8%	4	6.6%	2	4.9%	2	4.9%	2	4.9%	10	9.8%	8	7.8%	6	5.9%
	Assist-multi	8	13.1%	6	9.8%	4	6.6%	2	4.9%	2	4.9%	2	4.9%	10	9.8%	8	7.8%	6	5.9%
	PanicKiller	**34**	**55.7%**	**26**	**42.6%**	**15**	**24.6%**	**12**	**29.3%**	**10**	**24.4%**	**8**	**19.5%**	**46**	**45.1%**	**36**	**35.3%**	**23**	**22.5%**
Top-5	GPT4.0	6	9.8%	5	8.2%	4	6.6%	4	9.8%	4	9.8%	4	9.8%	10	9.8%	9	8.8%	8	7.8%
	GPT4.0-multi	6	9.8%	5	8.2%	4	6.6%	4	9.8%	4	9.8%	4	9.8%	10	9.8%	9	8.8%	8	7.8%
	Qwen	12	19.7%	9	14.8%	4	6.6%	6	14.6%	0	0%	0	0%	18	17.6%	9	8.8%	4	3.9%
	Qwen-multi	13	21.3%	10	16.4%	4	6.6%	6	14.6%	0	0%	0	0%	19	18.6%	10	9.8%	4	3.9%
	OCI	5	8.2%	5	8.2%	0	0%	1	2.4%	1	2.4%	0	0%	6	5.9%	6	5.9%	0	0%
	OCI-multi	5	8.2%	5	8.2%	0	0%	1	2.4%	1	2.4%	0	0%	6	5.9%	6	5.9%	0	0%
	Assist	8	13.1%	6	9.8%	4	6.6%	2	4.9%	2	4.9%	2	4.9%	10	9.8%	8	7.8%	6	5.9%
	Assist-multi	8	13.1%	6	9.8%	4	6.6%	2	4.9%	2	4.9%	2	4.9%	10	9.8%	8	7.8%	6	5.9%
	PanicKiller	**34**	**55.7%**	**26**	**42.6%**	**15**	**24.6%**	**12**	**29.3%**	**10**	**24.4%**	**8**	**19.5%**	**46**	**45.1%**	**36**	**35.3%**	**23**	**22.5%**

and the baseline tools. This analysis reveals both overlaps and differences among the tools, as shown in Figure 3. PanicKiller successfully repairs 25 unique cases among LLMs, which accounts for 54.3% of the total, and 30 unique patches among LLM-based APR tools, which accounts for 76.9%. In summary, most of panic bugs are successfully repaired by PanicKiller but not by the baseline tools, while a smaller portion are handled by the baselines but missed by PanicKiller.

We further illustrate the strengths and weaknesses of each method by two representative examples in Figure 4. Figure 4 (a) shows a failed patch from GPT4.0 that does not pass the regression test of the crate proc-macro2 [48], whereas PanicKiller successfully repairs the case using pattern *FP8. Insert Char Matcher*. As illustrated in the figure, the patch generated by GPT4.0 correctly locates the buggy line where `trunc_lo` accesses an invalid boundary of a UTF-8 string. However, it mistakenly modifies the return type of the `source_text` function, leading to unintended side effects at all its call sites and making the patch impractical for real use. This example reflects a common issue with LLM-based methods, which often introduce speculative changes that break type constraints or alter program semantics. In addition, LLMs struggle with tasks that require precise numerical reasoning, resulting in incorrect patches for computation-intensive code. LLM-based approaches are less effective at addressing Rust-specific bugs, as they often lack the ability to recognize Rust-specific patterns and are not well adapted to the language's unique characteristics. Beyond model capabilities, prompt design also plays a crucial role in determining the effectiveness of LLM-based APR tools. Tools like Rust Assistant and SWE-agent use generic prompts not tailored

```
-  fn source_text(&self, span: Span) -> String {
-    let lo = (span.lo - self.span.lo) as usize;
-    let trunc_lo = &self.source_text[lo..];
+  fn source_text(&self, span: Span) -> Option<String> {
+    let lo = (span.lo - self.span.lo) as usize;
+    let trunc_lo = self.source_text.get(lo..)?;
     let char_len = (span.hi - span.lo) as usize;
     let source_text = match trunc_lo.char_indices().nth(char_len) {
       Some((offset, _ch)) => &trunc_lo[..offset],
       None => trunc_lo,
     };
-    source_text.to_owned()
+    Some(source_text.to_owned())
   }
```

(a) A wrong patch for the crate proc-macro2 generated by GPT4.0 that fails to pass the regression test.

```
   #[inline]
   pub fn bytes_of<T: Pod>(t: &T) -> &[u8] {
-    match try_cast_slice::<T, u8>(core::slice::from_ref(t)) {
-      Ok(s) => s,
-      Err(_) => unreachable!(),
+    if size_of::<T>() == 0 {
+      &[]
+    } else {
+      match try_cast_slice::<T, u8>(core::slice::from_ref(t)) {
+        Ok(s) => s,
+        Err(_) => unreachable!(),
+      }
     }
   }
```

(b) A plausible patch for the crate bytemuck generated by Qwen that fixes a panic bug caused by arithmetic overflow.

Fig. 4. Two patches generated by GPT4.0 and Qwen for Rust crates proc-macro2 and bytemuck.

for panic bug repair, leading to less effective results. In contrast, our structured, workflow-specific prompts yield better results.

Nonetheless, LLM-based methods also exhibit unique advantages. For instance, in the bytemuck [66] crate case shown in Figure 4 (b), both OCI and Qwen successfully repaired a subtle indexing error related to a custom Pod trait—an issue that triggered an unreachable logical panic and was not addressed by PanicKiller. The LLMs demonstrated an understanding of the user-defined constraints on the Pod trait and inserted an appropriate boundary check to safely handle the corner case. This fix required comprehension of both the underlying data structures and the semantics of the trait. By accurately inferring the trait's intended behavior and invoking the correct API, the LLMs produced valid patches. In contrast, PanicKiller struggled with such cases due to its lack of domain-specific knowledge and limited type inference capabilities. These findings align with prior research on LLM-based APR [54], which suggests promising directions for future development, as further discussed in Section 5.

Table 7. Time consuming for PanicKiller.

Datasets	Min	Max	Avg.
Panic4R-Small	0.51s	588.72s	65.77s
Panic4R-Large	0.34s	425.32s	61.29s

4.2.4 Efficiency. Table 7 presents the time consumption of the complete automated repair process conducted by PanicKiller on the Panic4R dataset. On average, PanicKiller completes the repair pipeline in approximately 65 seconds for both small and large-scale repositories, demonstrating a practical and acceptable runtime. The significantly large maximum values, such as 588.72s in Panic4R-small, are primarily due to the varying number of regression test cases across repositories. When a crate includes a large suite of test cases, the validation phase, where PanicKiller checks the correctness of generated patches, can become notably time-consuming. However, the results do not strictly support the assumption that larger crates always require more time, since the major time overhead comes from executing test cases, while the time spent on transforming code through node matching remains relatively stable regardless of the crate size. As discussed in Section 2, the typical lifecycle of an issue from its reporting to the merging of a pull request spans about 50 days on average. To improve efficiency, we believe that with the assistance of PanicKiller, developers can automatically generate patches and submit them after a brief manual review, thereby accelerating the resolution of panic bugs and enhancing overall maintenance productivity.

4.2.5 Case Study. To evaluate the practical effectiveness of PanicKiller, we apply it to real-world issues collected from four widely-used GitHub repositories: `hifitime` [31], `unicode-segmentation` [86],

Table 8. Distribution of panics in real-world crates and fixing effectiveness of PanicKiller.

Crates	Stars	LoC	Open issues Total	Open issues Confirmed	Closed issues Total	Closed issues Similar
hifitime	315	10,762	28	21	0	-
unicode	556	7,345	11	7	1	0
ratatui	9,086	41,534	1	0	18	7
fancy-regex	409	5,557	1	0	3	2
Total			**41**	**28**	**22**	**9**

`ratatui` [32] and `fancy-regex` [67]. These repositories are not included in Panic4R. For open issues, we submit a PR with the fix patch generated by PanicKiller, and we track whether they are confirmed and merged by developers. For closed issues, we use TF-IDF [83] to calculate the similarity between the generated patch with the official patch. If the score exceeds 0.75, and the panic is fixed with the regression test passed, we consider they are similar. If the PR is accepted by project developers or if our generated patch is similar to the official patch, it could further demonstrate that PanicKiller can be employed to real-world application scenarios and provide fixing guidance for large-scale real-world projects.

The overall evaluation results on real-world projects are presented in Table 8, showing that PanicKiller successfully resolved 28 of 41 open issues with varied root causes, with all patches merged by developers. As for unsuccessful fixes, some patches were plausible; they removed the panic but required further manual inspection and minor adjustments. In terms of closed issues, PanicKiller successfully generates 9 out of 22 patches that are similar to official patches. The proportion is not very high, largely due to differences in their repair locations. For example, while the official approach might add a branch before the parameter call, PanicKiller tends to insert it after creating the parameter. Nonetheless, both strategies achieve equivalent outcomes.

Figure 5 presents the distribution of fix patterns applied by PanicKiller when repairing panic bugs across four real-world Rust crates. Consistent with the root cause analysis in Table 3, two patterns, i.e., *FP7. Mutate Arithmetic Operator* and *FP8. Insert Char Matcher*, dominate the fixing space, together accounting for 86.4% of all patches. These patterns correspond to common and tractable panic causes such as arithmetic overflows and invalid character processing, which also contribute significantly to the root cause distribution observed in the Rust compiler.

Not all patterns have been applied in this evaluation, partly due to the limited number of real-world issues. On the other hand, since the patterns are extracted as a comprehensive set from official documentation, not all of them are expected to appear in a limited evaluation scope. Nevertheless, the application of a broad range of patterns across multiple crates highlights the practical relevance and generality of our pattern-mining process.

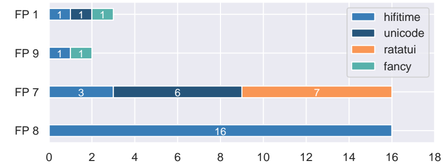
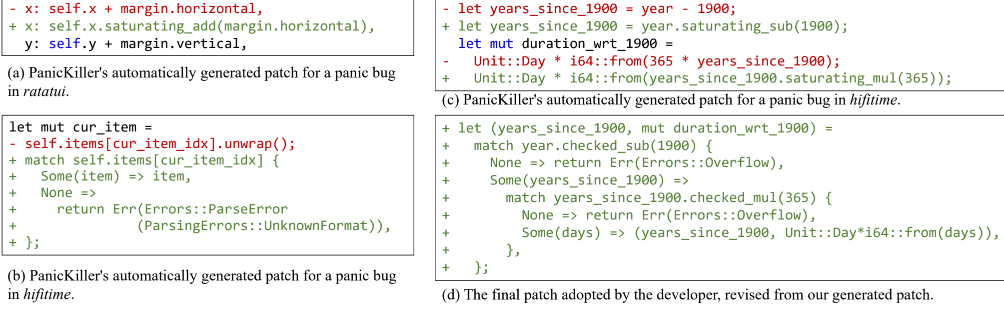


Fig. 5. Distribution of fix patterns applied by PanicKiller when repairing real-world bugs.

Below we illustrate some bug cases and their fixes.

Bug Case1: Figure 6 (a) shows the resolution of an arithmetic overflow error, which is a proposed patch of a closed issue for Rust crate ratatui[1]. The original code encountered unexpected edge cases when computing the variable `x`. One user triggered an arithmetic overflow by assigning a large integer to `margin.horizontal`, resulting in a panic caused by the overflow. Through fault localization, PanicKiller succeeded in pinpointing a list of potential fault locations, prioritizing

[1] https://github.com/ratatui/ratatui/pull/523



(a) PanicKiller's automatically generated patch for a panic bug in *ratatui*.

(b) PanicKiller's automatically generated patch for a panic bug in *hifitime*.

(c) PanicKiller's automatically generated patch for a panic bug in *hifitime*.

(d) The final patch adopted by the developer, revised from our generated patch.

Fig. 6. PanicKiller generated patches that successfully fixed real-world panic bugs in widely used Rust crates, including ratatui and hifitime.

`src/layout.rs:233:20` as the top candidate, which aligns exactly with the actual fix. Then, PanicKiller successfully generated 3 patches for this location. In this case, the fault was identified within a struct filed expression; however, a closer inspection through iterative analysis revealed that this match expression included a binary expression, making it suitable for applying the *Mutate Arithmetic Operator* pattern. Finally, we calculate the similarity score between the generated patch and the official patch, obtaining a score of 0.95, indicating the correctness.

Bug Case 2: The patch illustrated in Figure 6 (b), generated by PanicKiller[2], serves as an example of addressing a panic bug that arises from improper error handling caused by `unwrap`. The original code attempted to extract an item, `cur_item`, in a specific format, under the assumption that all possible user inputs had been accounted for. As a result, the developers opted to use `unwrap` as a quick and straightforward way to extract the value. However, a user later reported an input that did not match any expected format, leading to a panic error and causing confusion. This bug comes from an open issue of Rust crate hifitime [31], and our patch has been merged by developers. PanicKiller employs the *Insert Explicit Matcher* pattern, specifically designed to mitigate problems stemming from the misuse of unwrap. This approach transitions the unsafe usage of `unwarp()` to pattern matching, which results in returning an error message rather than triggering a panic error. Additionally, the patch created by PanicKiller takes into consideration the variable types to ensure consistency with the code context, thereby guaranteeing that the modified program passes regression testing.

Bug Case 3: Figure 6 (c) shows a patch automatically generated by PanicKiller to fix an arithmetic overflow bug in the *hifitime* crate[3]. This patch correctly identifies the faulty code and applies the *Mutate Arithmetic Operator* fix pattern by replacing the subtraction operation with a call to `saturating_sub()`. It passes all test cases and successfully eliminates the panic. Although functionally correct, the maintainers considered the patch as a plausible rather than the final fix. As shown in Figure 6 (d), their concern was not about the correctness of the patch, but about consistency with the project's coding style. Instead of silently handling overflows, the crate favors explicitly reporting such conditions. Nevertheless, they acknowledged the practical value of our proposed patch, noting that *"the use of* `saturating_sub()` *works for most applications."* Based on our plausible fix and accurate localization, the maintainers proposed a revised patch that better fits their intention. As shown in Figure 6(d), this version uses checked operations and explicit error propagation, along with a clearer error message that warns developers about incorrect API usage. This case illustrates the practical effectiveness of PanicKiller in real-world scenarios. While the

[2]https://github.com/nyx-space/hifitime/pull/285
[3]https://github.com/nyx-space/hifitime/pull/283

patches generated by PanicKiller may not fully match the developers' intended semantics, they are typically close enough to offer meaningful guidance, enabling developers to reach the final fix with minimal additional effort.

5 DISCUSSION

This section discusses our infrastructure's application scenarios, comparison with existing works, and threats to validity.

5.1 Application Scenarios

This work establishes a systematic infrastructure that offers a diverse dataset and comprehensive analysis of panic bugs specific to Rust projects. PanicFI offers a modular and extensible framework where each component can be substituted with alternative tools. For instance, the APR tool can be seamlessly integrated with other compatible systems, enabling greater flexibility and customization for various use cases. We now briefly discuss the application scenarios and the potential future research directions.

(1) Benchmark dataset for the Rust APR tools. The widely-used benchmark Defects4J has been crucial and influential for APR research in Java [41, 42, 59, 64, 65, 89, 90, 92]. Given the current limitations in the infrastructure for Rust program research, the dataset we propose, Panic4R, is designed to serve as a robust and reliable benchmark for subsequent studies. It includes macro switches and test case execution scripts, facilitating researchers to employ it as a benchmark dataset in their evaluations.

Panic4R offers detailed information for each case, encompassing the bug-triggering scenario and the exact patch applied to address the specific bug. This setup facilitates easy reproduction of the bug. Thus, Panic4R enables further analysis to uncover underlying patterns and common root causes of panic bugs. This can lead to better insights into Rust's common vulnerabilities and areas where APR tools can focus or improve, ultimately contributing to the robustness of Rust codebases and reducing the likelihood of runtime panics in production software. Moreover, it establishes a uniform evaluation metric, allowing for more objective and comparable assessments of APR tool effectiveness. Researchers and practitioners can measure improvements in patch accuracy, bug-fix efficiency, and detection of bug patterns across different tools, fostering a clearer understanding of APR advancements in the Rust ecosystem.

(2) Extendable dataset for Rust program repair. Rust, as an emerging language with significant advantages in memory safety, increasingly research and approaches designed for Rust have been proposed. In recent years, some of the existing tools [47, 78, 96] have been developed for fixing unique bugs in Rust programs. However, due to the less mature infrastructure of Rust compared to other mainstream languages, there has been a lack of structured and open-source datasets. This gap has made it challenging to standardize testing, repairing, and verification processes, and the collected datasets and code snippets are often not reusable for subsequent research.

In our proposed PanicFI, the project structure and test scripts included in Panic4R are both unified and highly extensible, including executable code snippets, test cases, and validation scripts. Future studies that aim to incorporate new panic bug repair datasets can easily extend Panic4R using a standardized format. Similarly, for other types of bug repair datasets, maintaining a similar repository structure allows for quick adaptation to standardized dataset formats, minimizing redundant and tedious work.

(3) Templates for pattern-based APR tools development. Pattern-based (also known as "template-based") APR tools was proved to be the most effective method compared to other approaches, such as search-based and constraint-based schemes [51]. After fault localization, the pattern-based APR tool could target these defects to select the corresponding fix templates to generate candidate patches. The mined patterns in our proposed PanicFI, containing 21 sub-patterns addressing 7 categories of panic root causes, can serve as templates for pattern-based APR tools. Based on this foundation, researchers can concentrate on developing improved techniques for fault localization and pattern prioritization. For example, multiple potential patterns may match for the identified fault locations, necessitating further research into efficient pattern selection and validation. Additionally, ensuring the correctness of program semantics following the application of these patterns merits deeper investigation.

In this paper, we demonstrate the effectiveness and practicality of these mined patterns through the implementation of PanicKiller. Future work could focus on developing more refined repair tools based on these patterns, aimed at enhancing the accuracy of program semantics and improving repair efficiency.

(4) Dataset for fine-tuning large language models (LLMs). In recent years, several research have employed deep learning models for automated program repair [43, 49, 92], and their results have demonstrated the feasibility of using AI technologies to assist in fixing programs. However, the effectiveness of these approaches is often constrained by several factors, such as the scale of the datasets for model training, the risk of overfitting due to model fine-tuning, and the loss of information associated with tokenizing unstructured data like code and text [97]. With the rapid advancement of large-scale models, it is highly potential that program repair techniques based on these LLMs will be implemented. This development could significantly enhance the precision and efficiency of tradition learning-based automated repairs, mitigating current limitations and opening new avenues for research in software maintenance.

To support research in LLM-based program repair, high-quality datasets are essential [38, 50]. The datasets and mined patterns included in our proposed PanicFI, are particularly well-suited for use as training data for LLMs. Notably, the patterns we have identified are diverse, encompassing 21 sub-patterns and covering 7 different root causes of panic errors. Moreover, our open-sourced patterns feature various data structures, including source code, abstract AST structures, and corresponding textual descriptions. These rich sources of information are highly beneficial for the learning and evolution of LLMs, providing a robust foundation for developing more effective automated program repair technologies.

(5) Helpful for fixing real-world opening panic bugs. The infrastructure we propose, PanicFI, is closely aligned with real-world application scenarios. The Panic4R is derived entirely from open-source Rust projects within the ecosystem, ensuring its relevance and applicability. The patterns we have mined originate from Rust's official code implementations, further validating the authenticity and utility of our infrastructure. Moreover, PanicKiller is capable of handling real, large-scale Rust projects. Our experiment results have shown PanicKiller is much more efficient than manual fixes, and PanicKiller has successfully resolved 28 panic bugs in real Rust crates, demonstrating its practical effectiveness.

5.2 Comparison with Existing Code Patterns

Comparison with Java/C/C++ code patterns. Although numerous bug-fixing patterns oriented towards Java, C, and C++ have been proposed to corresponding APR tools, the code patterns we have identified are specific to Rust and are utilized for fixing panic bugs. Some root causes of panic bugs in Rust are unique due to its language design. For instance, *Ignorance of Interior Mutability*

arises from Rust's borrowing rules, which are closely tied to its ownership system. This makes it impossible to find a similar pattern in other languages, as they do not enforce such strict ownership models. In particular, certain patterns, such as *Fuse Future to Prevent Panic*, are specifically designed to address concurrency panics. These patterns are deeply connected to Rust's `Future` trait and its execution mechanism, which differ fundamentally from concurrency models in Java or C++. Additionally, unlike Java or C++, where safety rules such as ownership are not enforced by the language, our proposed patterns carefully maintain these safety rules. This attention to Rust's unique ownership model ensures that the fixes not only resolve the bugs but also uphold the language's guarantees of memory safety and concurrency.

Comparison with Rust-specific code patterns. To date, only a few fix patterns have been specifically developed for Rust programs. Qin et al. [76] conducted an empirical study on Rust safety issues, with a primary focus on memory and thread safety. While the study also discussed certain safety bugs related to panics and their corresponding fix strategies, its bug analysis and solutions are largely intended as guidance for developers. Specifically, the study presents fix strategies such as the *Eliminate panic statements* approach, which, although useful, is too general to be directly applied in an automated tool. Since the described strategies are broad and cover only typical panic bugs, they do not fully account for all the underlying causes of panic issues. In contrast, our work takes a more focused approach. We have mined the root causes and fix patterns for panic bugs from the source code of the Rust compiler and abstracted these patterns at the level of AST nodes for the design of PanicKiller. Additionally, our experimental results demonstrate the effectiveness of the proposed fixes. From a pattern design perspective, our approach provides a more comprehensive solution by addressing a wider range of panic scenarios and enabling automated program repair.

Two pattern-based tools designed for Rust, Rust-lancet [96] and Ruxanne [78], have been proposed to address common bug types, such as ownership-related issues and missing attributes. Rust-lancet has designed three repair patterns focused on variable ownership and lifetimes. By applying these patterns and recompiling the program, it iteratively resolves errors related to ownership rules based on the compiler's error messages. Ruxanne, on the other hand, has proposed 12 general fix patterns and 8 patterns specifically for the borrow checker. Like Rust-lancet, all of these patterns are designed to address issues in programs that fail to compile. However, the panic bug is triggered at runtime, which suggests that these programs have already passed the compilation checks. Therefore, existing patterns designed to fix other types of Rust bugs are not applicable to resolving panic bugs. Besides, while most existing tools focus on resolving compilation issues, panic bugs lead to unexpected program termination at runtime. This not only has a more severe impact but also presents significant challenges for developers. Panic bugs are notoriously difficult to diagnose and resolve, highlighting the need for a systematic repair dataset and pattern to address them. In conclusion, this paper introduces the first infrastructure for exploring Rust panic bugs, identifying their root causes, and developing targeted repair patterns, significantly advancing the understanding and remediation of Rust programs.

5.3 Practical Implications and Future Work

The evaluation results in Section 4.2 show that PanicKiller can produce not only correct patches but also plausible ones. While plausible patches may not always match the developer's original implementation, they still eliminate runtime panics and pass all available regression tests, demonstrating practical value in real-world debugging. In particular, they help developers quickly identify the root cause of a panic and provide concrete, structurally sound suggestions that can serve as effective starting points for manual refinement. As shown in the bug cases demonstrated in Section 4.2.5, many of the patches generated by PanicKiller have been acknowledged by developers as useful or

even directly applicable, highlighting the practical value and generality of the fix patterns we have identified. To mitigate potential semantic inconsistencies introduced by these plausible patches, PanicKiller performs a patch validation phase in which each candidate is subjected to compilation checks and the crate's original regression test suite. This ensures that only behaviorally consistent and safe patches are presented to the user. In practice, we observe that PanicKiller rarely produces fixes that significantly alter program semantics. The combination of structurally correct patterns and thorough validation provides a strong safeguard against unsafe or misleading transformations.

Despite these strengths, we acknowledge that the fix patterns employed by PanicKiller are not comprehensive in all scenarios. In particular, PanicKiller is currently not designed to address panic bugs that arise from incorrect or incomplete program logic, as discussed in Section 3.7. Such panics often stem from unmet assertions, unhandled edge cases, or violations of user-defined invariants, which are closely tied to the intended semantics of the application. For instance, a panic caused by a failed assertion may be intentional, serving as a safeguard against invalid input, or it may indicate that critical constraints are missing. In such cases, applying a fix without a deeper semantic understanding of the program risks introducing incorrect or even unsafe behavior. Section 4.2.3 presents a representative case in which the LLM successfully fixed a logic-induced panic by understanding a user-defined trait constraint, whereas PanicKiller failed to generate a correct patch. Consequently, due to the inherent difficulty of repairing logic-related panics without semantic understanding, PanicKiller focuses primarily on language-level panics that exhibit well-defined and structurally consistent patterns, where automated repair is more tractable.

Addressing logic-related panics remains an open and challenging problem for automated program repair. We believe that integrating PanicKiller's pattern-based precision with the high-level reasoning capabilities of LLMs presents a promising future direction, which we plan to explore in subsequent work. Specifically, we aim to develop a hybrid repair framework that combines PanicKiller's deterministic, pattern-based transformations with the semantic understanding and context-aware patch generation strengths of LLMs. While PanicKiller offers precise AST-level edits based on well-defined fix patterns, LLMs are better suited for understanding high-level program intent and generating context-aware patches. Our preliminary investigation suggests that guiding LLMs with PanicKiller's structured patterns can produce higher-quality fixes for semantically complex panics that lie beyond the reach of traditional pattern-based methods. Conversely, LLMs can help generalize and extend PanicKiller's pattern space.

In addition, we plan to develop PanicKiller into a lightweight compiler plugin that integrates directly into existing development environments. By leveraging its precise fault localization and patch generation capabilities, this plugin would proactively warn developers about potential panic-inducing code and provide prioritized candidate fixes in real time. Such integration could streamline the debugging workflow, reduce the effort for manual inspection, and facilitate faster resolution of runtime panics.

5.4 Limitations

While our current work makes promising strides in addressing Rust panic bugs, there remain two major limitations that warrant discussion.

First, manually crafted fixing patterns have an inherently limited scope. While our mined pattern set covers many common manifestations of panic bugs, it cannot generalize to all possible scenarios. For instance, panics caused by unsafe memory access in multi-threaded environments with intricate synchronization are not well addressed by our existing patterns. Likewise, panic bugs that stem from deeper logical flaws in program semantics, rather than straightforward syntactic triggers, are beyond the scope of pattern-based fixes. In such cases, the occurrence of a panic is often closely tied to the intended program semantics. Directly matching a pattern may eliminate the panic, but

it may change the programmer's intended behavior. For example, a developer might deliberately design a program to panic under certain conditions as a safeguard. As a result, the mined and implemented patterns in this paper cannot cover the full scope of panic scenarios.

Second, the process of identifying new patterns and implementing the corresponding fixing mechanisms is not automated. The goal of this paper is to propose Rust panic bug fix patterns, and our primary focus has been on understanding the causes of panics and the associated fix strategies. Currently, extending the pattern set requires researchers to systematically read through compiler and standard library source code, analyze documentation, and manually abstract fix strategies into reusable patterns. This process could be complex and highly dependent on domain expertise, making it the most time-consuming part of our work. In the future, we consider automating the mining and integration of new patterns to reduce manual effort and improve scalability.

In summary, while our approach provides a solid foundation for understanding and addressing Rust panic bugs, overcoming these limitations may be essential for broader applicability and long-term impact. Future work aimed at expanding pattern coverage and automating the mining process may help unlock the full potential of pattern-based fixes in practice.

5.5 Threats to Validity

One of the potential threats to validity concerns the representativeness of our collected dataset, since all code and patches are sourced from open-source crates. However, we consider these crates to be relatively complex and representative of large-scale Rust projects. Panic4R comprises the top 500 most downloaded crates, reflecting the actual usage frequency and activity levels of these programs. We have also modeled our data collection process after the Defects4J dataset to enhance the reliability of our data. This adherence to proven methodologies in dataset construction supports the validity of our research findings.

Another potential threats lies in the incompleteness of the mined fixing patterns. In real-world scenarios, specific bug triggers may have unique or diverse repair strategies. To enhance the comprehensiveness of our mined patterns, we systematically explore and extract from Rust's official implementation code, which is considered to contain the most standard repair strategies. Additional patterns that may emerge in the future can also be easily integrated into our released PanicFI, further improving the effectiveness of pattern-based repair tools.

6 RELATED WORK

In this section, we compare our work with other APR approaches, as well as the testing and analysing work for Rust program.

6.1 Automated Program Repair

Automated Program Repair (APR) has seen substantial progress in recent years. Most APR methods [41, 42, 59, 63–65, 79, 89, 90, 92] are primarily focused on Java programs, with their effectiveness evaluated using Defect4J [57], a comprehensive dataset of real-world bugs from open-source Java projects. Defect4J [57] has long been considered the gold standard for benchmarking APR tools, providing a reliable foundation for research and development in the field [51]. However, no similar benchmark has been proposed for Rust panic bugs. To fill this gap, in this paper, we present the first dataset specifically for Rust panic bugs, drawn from real open-source Rust crates. This dataset not only offers an invaluable resource for understanding the characteristics and prevalence of panic bugs in Rust but also serves as a guide for future research and the development of APR techniques tailored to Rust.

A recent study [51] divides non-learning-based APR into three categories: search-based [61, 74, 82, 88, 89], constraint-based [35, 41, 42, 69, 72, 73, 87] and template-based [60, 79, 85] approaches.

VarFix [90], a search-based way of observing which combinations of edit operations pass the test. Constraint-based techniques like Nopol [94] and SemFix [71] transform the repair process into a constraint solving problem, reducing the search space. kPAR [63] and AVATAR [65] generate fix patterns collected by manual extraction and static violation analysis respectively, used by iFixR [59]. TBar [64] is proposed to assess the qualitative and quantitative diversity of previous repair templates. However, these fix patterns are mainly designed for common bugs in Java, and they are not very suitable for fixing panic bugs in Rust. To address the above issues, this paper explores a series of causes and fixes for panic bug from the development of the Rust compiler, summarizing them into corresponding patterns, which are used to design a pattern-based Rust panic APR tool.

As for learning-based APR, AlaphaRepair [92] achieves state-of-the-art results on both Java and Python programs via zero-shot learning. VulRepair [49] highlights the advancement of NMT-based automated vulnerability repairs with pre-trained models. According to a survey [97] on learning-based APR, extensive manual effort is required to produce high-quality test datasets to train a reliable model, which implies the high cost in learning-based APR techniques. Also, it has been questioned about the generalizability of learning-based APR tools when it comes to the strict syntactic structural features and complex semantic dependencies of PLs [53, 97], which our experiments with LLMs has demonstrated to some extent. Different from the existing learning-based APR tools, our work mainly focused on mining fix pattern for addressing panic bugs in Rust program, and our proposed PanicKiller has demonstrated the effectiveness of these patterns. We believe in the future, our proposed fix patterns combined with learning-bases approach can be applied to develop novel APR tools.

6.2 Rust Program Testing and Analysis

Due to Rust's innovative safety mechanism, new challenges have been posed in its testing and analyzing. RustSmith [81] employs random program generation to test the Rust compiler. As for RULF [52] and SyRust [84], they concentrate on testing Rust crates by generating API sequences. In the realm of Rust program analysis, RUPTA [62] introduces a context-sensitive pointer analysis framework for Rust, successfully applied to construct call graphs. Additionally, through static analysis, tools like SafeDrop [46] and Rudra [37] detect memory safety issues in large-scale Rust programs, contributing to enhanced program robustness. In contrast, RustCheck [93] employs dynamic analysis techniques to uncover memory safety vulnerabilities.

Different from existing testing approaches, our work proposed the first APR tool specifically tailored to address errors related to Rust's panic mechanism, and we focused on fixing the practical panic bugs. Besides, we have constructed a real-world code dataset and fix patterns, which serves as an infrastructure for Rust program comprehension and repair.

7 CONCLUSION

In this paper, we introduce an infrastructure PanicFI designed to support fixing panic bugs of real-world Rust programs. We construct the first Rust program's fixing dataset, Panic4R, containing 102 real-world panic bugs and their patches. Additionally, we conduct pattern mining based on Rust compiler source code to identify Rust-specific fixing patterns. We also introduce an APR tool, PanicKiller, which effectively localizes faults and generates patches, outperforming state-of-the-art LLM-based tools. Moreover, PanicKiller has successfully resolved 28 open issues related to panics, all of which have been confirmed and merged by developers.

DATA AVAILABILITY

The dataset Panic4R can be found at: https://anonymous.4open.science/r/Panic4R-1085/README.md. The source code of PanicKiller can be found at: https://anonymous.4open.science/r/PanicKiller-444E/README.md.

ACKNOWLEDGMENTS

We would like to thank reviewers for their constructive comments. This work was supported in part by the National Natural Science Foundation of China under Grant No. 62372225 and No. 62172209, and by the Fundamental Research Funds for the Central Universities (No. 020214380131 and No. ZZKT2025A10).

REFERENCES

[1] [n. d.]. 1660-try-borrow - The Rust RFC Book. https://rust-lang.github.io/rfcs/1660-try-borrow.html [Online; accessed 2025-01-14].
[2] [n. d.]. checked_add in u8 - Rust. https://doc.rust-lang.org/std/primitive.u8.html#method.checked_add. (Accessed on 01/14/2024).
[3] [n. d.]. Combinators: map - Rust By Example. https://doc.rust-lang.org/rust-by-example/error/option_unwrap/map.html [Online; accessed 2025-01-12].
[4] [n. d.]. Dangling References - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html#dangling-references. (Accessed on 01/17/2025).
[5] [n. d.]. Data Types - The Rust Programming Language. https://doc.rust-lang.org/book/ch03-02-data-types.html#integer-overflow [Online; accessed 2025-01-13].
[6] [n. d.]. Data Types - The Rust Programming Language. https://doc.rust-lang.org/book/ch03-02-data-types.html [Online; accessed 2025-01-18].
[7] [n. d.]. Default for Default Values - The Rust Programming Language. https://doc.rust-lang.org/book/appendix-03-derivable-traits.html#default-for-default-values. (Accessed on 01/13/2024).
[8] [n. d.]. Doc: catch-unwind - Rust. https://doc.rust-lang.org/std/panic/fn.catch_unwind.html. (Accessed on 01/06/2025).
[9] [n. d.]. Implementation of borrow() for RefCell. https://github.com/rust-lang/rust/blob/73c0ae6aec8f3d07467dfb9339761fa2eec92a44/library/core/src/cell.rs#L959-L998. (Accessed on 01/17/2025).
[10] [n. d.]. Issues · rust-lang/rust. https://github.com/rust-lang/rust/labels/I-ICE. (Accessed on 04/12/2024).
[11] [n. d.]. Option in core - Rust. https://doc.rust-lang.org/src/core/option.rs.html. (Accessed on 12/31/2024).
[12] [n. d.]. Poisoning - The Rustnomicon. https://doc.rust-lang.org/nomicon/poisoning.html. (Accessed on 01/17/2025).
[13] [n. d.]. prettyplease - crates.io. https://crates.io/crates/prettyplease. (Accessed on 05/21/2025).
[14] [n. d.]. RefCell<T> and the Interior Mutability Pattern - The Rust Programming Language. https://doc.rust-lang.org/book/ch15-05-interior-mutability.html [Online; accessed 2025-01-14].
[15] [n. d.]. References and Borrowing - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html [Online; accessed 2025-01-12].
[16] [n. d.]. rust-lang/rust: Empowering everyone to build reliable and efficient software. https://github.com/rust-lang/rust. (Accessed on 07/19/2024).
[17] [n. d.]. The Rust RFC Book. https://rust-lang.github.io/rfcs/. (Accessed on 01/17/2024).
[18] [n. d.]. rustc_hir - Rust. https://doc.rust-lang.org/beta/nightly-rustc/rustc_hir/index.html. (Accessed on 05/21/2025).
[19] [n. d.]. The Rustdoc Book. https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html. (Accessed on 05/21/2025).
[20] [n. d.]. saturating_add in u8 - Rust. https://doc.rust-lang.org/std/primitive.u8.html#method.saturating_add. (Accessed on 01/14/2024).
[21] [n. d.]. The Slice Type - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-03-slices.html [Online; accessed 2025-01-18].
[22] [n. d.]. Unrecoverable Errors with panic! - The Rust Programming Language. https://doc.rust-lang.org/book/ch09-01-unrecoverable-errors-with-panic.html. (Accessed on 01/17/2025).
[23] [n. d.]. Using Result Combinator Functions in Rust - Pat Shaughnessy. https://patshaughnessy.net/2019/11/19/using-result-combinator-functions-in-rust [Online; accessed 2025-01-12].
[24] [n. d.]. wrapping_add in u8 - Rust. https://doc.rust-lang.org/std/primitive.u8.html#method.wrapping_add. (Accessed on 01/14/2024).
[25] 2023. Redox - Your Next(Gen) OS - Redox - Your Next(Gen) OS. https://www.redox-os.org/. (Accessed on 12/06/2023).

[26] 2023. Servo, the embeddable, independent, memory-safe, modular, parallel web rendering engine. https://servo.org/. (Accessed on 12/06/2023).
[27] 2023. Stratis Storage. https://stratis-storage.github.io/. (Accessed on 12/06/2023).
[28] 2023. TiKV is a highly scalable, low latency, and easy to use key-value database. https://tikv.org/. (Accessed on 12/06/2023).
[29] 2023FixMiner: Mining relevant fix patterns for automated program repair. White House urges developers to dump C and C++ | InfoWorld. https://www.infoworld.com/article/3713203/white-house-urges-developers-to-dump-c-and-c.html. (Accessed on 03/17/2024).
[30] 2024. crates.io: Rust Package Registry. https://crates.io/. (Accessed on 04/03/2024).
[31] 2024. hifitime: a powerful Rust and Python library designed for time management. https://github.com/nyx-space/hifitime. (Accessed on 29/07/2024).
[32] 2024. ratatui: Rust library that's all about cooking up terminal user interfaces (TUIs). https://github.com/ratatui-org/ratatui. (Accessed on 29/07/2024).
[33] 2024. rust_hir: High-level Intermediate Representation. https://doc.rust-lang.org/stable/nightly-rustc/rustc_hir/hir/index.html. (Accessed on 29/07/2024).
[34] 2024. syn: Parser for Rust source code. https://github.com/dtolnay/syn. (Accessed on 29/07/2024).
[35] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2021. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2162–2181. https://doi.org/10.1109/TSE.2019.2944914
[36] Anthropic. [n. d.]. Claude 3.7 Sonnet. https://openrouter.ai/anthropic/claude-3.7-sonnet. (Accessed on 05/30/2025).
[37] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570
[38] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. https://proceedings.mlr.press/v139/berabi21a.html
[39] William Bugden and Ayman Alahmar. 2022. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503* (2022).
[40] K. Chen and V. Rajlich. 2000. Case study of feature location using dependence graph. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*. 241–247. https://doi.org/10.1109/WPC.2000.852498
[41] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 637–647. https://doi.org/10.1109/ASE.2017.8115674
[42] Liushan Chen, Yu Pei, and Carlo A. Furia. 2021. Contract-Based Program Repair Without The Contracts: An Extended Study. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2841–2857. https://doi.org/10.1109/TSE.2020.2970009
[43] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. SeqTrans: Automatic Vulnerability Fix via Sequence to Sequence Learning. arXiv:2010.10805 [cs.CR] https://arxiv.org/abs/2010.10805
[44] Cloudflare. 2023. Cloudflare - The Web Performance & Security Company. https://www.cloudflare.com/. (Accessed on 12/06/2023).
[45] Big Code. [n. d.]. Big Code Models Leaderboard. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard. (Accessed on 05/30/2025).
[46] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 82 (may 2023), 21 pages. https://doi.org/10.1145/3542948
[47] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing Rust Compilation Errors using LLMs. arXiv:2308.05177 [cs.SE] https://arxiv.org/abs/2308.05177
[48] dtolnay. [n. d.]. proc-macro2. https://github.com/dtolnay/proc-macro2. (Accessed on 06/05/2025).
[49] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. https://doi.org/10.1145/3540250.3549098
[50] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.
[51] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. arXiv:2303.18184 [cs.SE]

[52] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2022. RULF: rust library fuzzing via API dependency graph traversal. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 581–592. https://doi.org/10.1109/ASE51524.2021.9678813

[53] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[54] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1408–1420. https://doi.org/10.1145/3691620.3695513

[55] Ralf Jung. 2020. Understanding and evolving the Rust programming language. (2020).

[56] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152.

[57] 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* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[58] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.

[59] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 314–325. https://doi.org/10.1145/3338906.3338935

[60] 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. 213–224. https://doi.org/10.1109/SANER.2016.76

[61] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[62] Wei Li, Dongjie He, Yujiang Gui, Wenguang Chen, and Jingling Xue. 2024. A Context-Sensitive Pointer Analysis Framework for Rust and Its Application to Call Graph Construction. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction* (<conf-loc>, <city>Edinburgh</city>, <country>United Kingdom</country>, </conf-loc>) *(CC 2024)*. Association for Computing Machinery, New York, NY, USA, 60–72. https://doi.org/10.1145/3640537.3641574

[63] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 102–113. https://doi.org/10.1109/ICST.2019.00020

[64] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[65] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyandè. 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)*. 1–12. https://doi.org/10.1109/SANER.2019.8667970

[66] Lokathor. [n. d.]. bytemuck: A crate for mucking around with piles of bytes. https://github.com/Lokathor/bytemuck. (Accessed on 06/05/2025).

[67] Lokathor. [n. d.]. fancy-regex. https://github.com/fancy-regex/fancy-regex. (Accessed on 06/05/2025).

[68] m-a p. [n. d.]. OpenCodeInterpreter-DS-33B. https://huggingface.co/m-a-p/OpenCodeInterpreter-DS-33B. (Accessed on 05/30/2025).

[69] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 448–458. https://doi.org/10.1109/ICSE.2015.63

[70] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 151–160. https://doi.org/10.1109/ICSME.2014.37

[71] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.

1109/ICSE.2013.6606623

[72] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[73] Yu Pei, Yi Wei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. 2011. Code-based automated program fixing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 392–395. https://doi.org/10.1109/ASE.2011.6100080

[74] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In *2013 IEEE International Conference on Software Maintenance*. 180–189. https://doi.org/10.1109/ICSM.2013.29

[75] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[76] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. 2024. Understanding and Detecting Real-World Safety Issues in Rust. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1306–1324. https://doi.org/10.1109/TSE.2024.3380393

[77] Qwen. [n. d.]. Qwen2.5-Coder-32B-Instruct. https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct. (Accessed on 05/30/2025).

[78] Mohammad Robati Shirzad and Patrick Lam. 2024. A study of common bug fix patterns in Rust. *Empirical Softw. Engg.* 29, 2 (feb 2024), 34 pages. https://doi.org/10.1007/s10664-023-10437-1

[79] Seemanta Saha, Ripon k. Saha, and Mukul r. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 13–24. https://doi.org/10.1109/ICSE.2019.00020

[80] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE Computer Society, Los Alamitos, CA, USA, 118–121. https://doi.org/10.1109/MSR.2010.5463280

[81] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. https://doi.org/10.1145/3597926.3604919

[82] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. *SIGPLAN Not.* 50, 6 (jun 2015), 43–54. https://doi.org/10.1145/2813885.2737988

[83] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.

[84] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 899–913. https://doi.org/10.1145/3453483.3454084

[85] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 471–482. https://doi.org/10.1109/ICSE.2015.65

[86] unicode rs. [n. d.]. unicode-segmentation. https://github.com/unicode-rs/unicode-segmentation. (Accessed on 06/05/2025).

[87] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/1831708.1831716

[88] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 356–366. https://doi.org/10.1109/ASE.2013.6693094

[89] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[90] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 354–366. https://doi.org/10.1145/3468264.3468600

[91] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 181–190. https://doi.org/10.1109/ICSME.2014.40

[92] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. https://doi.org/10.1145/3540250.3549101

[93] Lei Xia, Yufei Wu, and Baojian Hua. 2023. Rustcheck: Safety Enhancement of Unsafe Rust via Dynamic Program Analysis. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. 871–872. https://doi.org/10.1109/QRS-C60940.2023.00102

[94] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Softw. Eng.* 43, 1 (jan 2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[95] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://arxiv.org/abs/2405.15793

[96] Wenzhang Yang, Linhai Song, and Yinxing Xue. 2024. Rust-lancet: Automated Ownership-Rule-Violation Fixing with Behavior Preservation *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 85, 13 pages. https://doi.org/10.1145/3597503.3639103

[97] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 55 (dec 2023), 69 pages. https://doi.org/10.1145/3631974

[98] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. 2023. A Closer Look at the Security Risks in the Rust Ecosystem. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 34 (dec 2023), 30 pages. https://doi.org/10.1145/3624738

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009