

# An Empirical Study of Rust-Specific Bugs in the rustc Compiler

ZIXI LIU, State Key Laboratory for Novel Software Technology Nanjing University, China

YANG FENG, State Key Laboratory for Novel Software Technology Nanjing University, China

YUNBO NI, State Key Laboratory for Novel Software Technology Nanjing University, China

SHAOHUA LI, The Chinese University of Hong Kong, China

XIZHE YIN, State Key Laboratory for Novel Software Technology Nanjing University, China

QINGKAI SHI, State Key Laboratory for Novel Software Technology Nanjing University, China

BAOWEN XU, State Key Laboratory for Novel Software Technology Nanjing University, China

ZHENDONG SU, ETH Zurich, Switzerland

Rust is gaining popularity for its well-known memory safety guarantees and high performance, distinguishing it from C/C++ and JVM-based languages. Its compiler, `rustc`, enforces these guarantees through specialized mechanisms such as trait solving, borrow checking, and specific optimizations. However, Rust's unique language mechanisms introduce complexity to its compiler, leading to Rust-specific compiler bugs that are less common in traditional compilers. With Rust's increasing adoption in safety-critical domains, understanding these language mechanisms and their impact on compiler bugs is essential for improving the reliability of both `rustc` and Rust programs. Yet, we still lack a large-scale, detailed, and in-depth study of Rust-specific bugs in `rustc`.

To bridge this gap, this work conducts a comprehensive and systematic study of Rust-specific bugs in `rustc`, with a particular focus on the components that support its unique language features. Our analysis examines issues and fixes reported between 2022 and 2024, with a manual review of 301 valid issues. We categorize these bugs based on their causes, symptoms, affected compilation stages, and test case characteristics. Additionally, we evaluate existing `rustc` testing tools to assess their effectiveness and limitations. Our key findings include: (1) `rustc` bugs primarily arise from Rust's type system and lifetime model, with frequent errors in the High-Level Intermediate Representation (HIR) and Mid-Level Intermediate Representation (MIR) modules due to complex checkers and optimizations; (2) bug-revealing test cases often involve unstable features, advanced trait usages, lifetime annotations, standard APIs, and specific optimization levels; (3) while both valid and invalid programs can trigger bugs, existing testing tools struggle to detect non-crash errors, underscoring the need for further advancements in `rustc` testing.

---

Authors' Contact Information: Zixi Liu, zxliu@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology Nanjing University, Nanjing, China; Yang Feng, fengyang@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology Nanjing University, Nanjing, China; Yunbo Ni, yunboni@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology Nanjing University, Nanjing, China; Shaohua Li, shaohuali@cse.cuhk.edu.hk, The Chinese University of Hong Kong, China; Xizhe Yin, xizheyin@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology Nanjing University, Nanjing, China; Qingkai Shi, qingkaishi@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology Nanjing University, Nanjing, China; Baowen Xu, bwxu@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology Nanjing University, Nanjing, China; Zhendong Su, zhendong.su@inf.ethz.ch, ETH Zurich, Switzerland.

---

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.

ACM 1557-735X/2025/8-ART

<https://doi.org/XXXXXX.XXXXXXXX>

### ACM Reference Format:

Zixi Liu, Yang Feng, Yunbo Ni, Shaohua Li, Xizhe Yin, Qingkai Shi, Baowen Xu, and Zhendong Su. 2025. An Empirical Study of Rust-Specific Bugs in the `rustc` Compiler. *J. ACM* 37, 4 (August 2025), 26 pages. <https://doi.org/XXXXXX.XXXXXXXX>

## 1 Introduction

As the demand for more secure programming paradigms grows, the need for languages with fewer memory vulnerabilities becomes more recognized. For instance, United States White House recently emphasized the importance of adopting memory-safe languages, with Rust recognized as a leading example [InfoWorld 2023]. Rust's unique principles, such as ownership, borrowing, and lifetimes, enable developers to write both secure and efficient code. Additionally, Rust's focus on zero-cost abstractions and fearless concurrency has made it particularly popular in system programming [Jung et al. 2021; Klabnik and Nichols 2023]. Recently, there is an increasing trend to re-engineer widely used software systems in Rust [Cloudflare 2023; RedoxOS 2023; Servo 2023; STRATIS 2023; TiKV 2023].

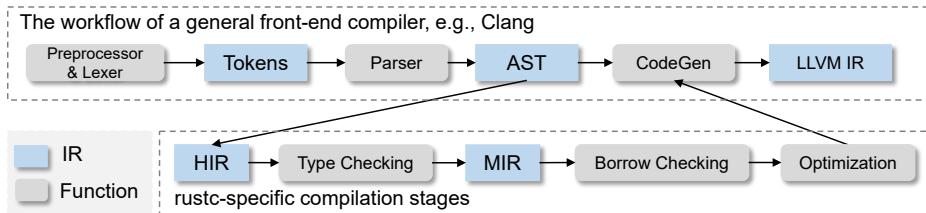


Fig. 1. The high-level workflow of `rustc` and a general front-end compiler.

The primary compiler for Rust is the official open-source `rustc` [Rust 2023b], which is written in Rust and uses LLVM [Lattner and Adve 2004] as its back-end. Similar to classic front-end compilers, such as Clang, `rustc` translates source code into LLVM IR, but it introduces additional intermediate representations (IRs) and compilation components to support Rust's unique language mechanisms. As illustrated in Figure 1, `rustc` follows a multi-stage compilation workflow tailored to enforce Rust's strict safety guarantees and advanced type system. After parsing the input program, the Abstract Syntax Tree (AST) is transformed into the High-Level Intermediate Representation (HIR), which abstracts over syntactic details to facilitate type inference, type checking, and trait resolution. This processing is crucial yet complex due to Rust's trait system, which enables zero-cost abstractions while supporting highly flexible usage patterns. Additionally, some data types are annotated with lifetimes, posing challenges for `rustc`'s type inference. Then, the HIR is lowered to Mid-Level Intermediate Representation (MIR), a control-flow-oriented representation crucial for enforcing Rust's ownership model, borrow checking, and move semantics. Before generating LLVM IR, `rustc` applies a series of Rust-specific optimizations on the MIR, ensuring efficient code generation while maintaining memory safety and preventing data races.

While these specific IRs and components are essential for enforcing memory safety and preventing data races, they also introduce significant complexity to compilation. Bugs in `rustc` may weaken these guarantees and compromise Rust's memory safety. For instance, a recent `rustc` bug led to an unsound borrow check, allowing a program that should have been rejected to compile, potentially causing Use-After-Free<sup>1</sup>. Despite their importance, existing tools and studies have overlooked the unique challenges posed by Rust-specific compilation mechanisms, leaving a gap in understanding their impact on testing `rustc`. To date, the only empirical study on Rust compilers is conducted by Xia et al. [Xia et al. 2023]. While it provides comprehensive statistics, it lacks an in-depth

<sup>1</sup><https://github.com/rust-lang/rust/issues/132186>

analysis. For example, it identifies `src/test`, `librustc`, and `librustcdoc` are the three most error-prone modules in `rustc`, yet they belong to the testing suite and standard library rather than the compiler itself. This misclassification may mislead our understanding of `rustc`'s design flaws. Besides, there is currently limited tooling available to effectively test and improve the reliability of `rustc`. In the open-source community, fuzzing scripts are commonly used to generate random programs for detecting crash bugs, but they often fail to identify compile-time issues like miscompilations. In the research domain, RustSmith [Sharma et al. 2023] is proposed as a program generator but provides limited support for Rust-specific features, including traits and generics. Other `rustc` testing techniques [Dewey et al. 2015; Wang and Jung 2024; Yang et al. 2024] attempt to generate MIR or use macros to generate test cases, but they can only detect a limited number of real `rustc` bugs. We consider that the lack of an effective testing tool specifically designed for `rustc` stems largely from an insufficient understanding of the unique bug characteristics within `rustc`.

To bridge the gap in understanding Rust-specific compiler bugs, we conduct a comprehensive quantitative and qualitative bug study on the official Rust compiler, `rustc`. Our study is explicitly designed to focus on Rust-specific bugs, particularly those arising from the unique compilation stages that distinguish `rustc` from traditional compilers. Therefore, we selected labels related to Rust-specific IRs from Rust's official issue tracker, thus naturally excluding generic compiler bugs, such as those in the parser. We focus on `rustc` because it is the only compiler currently capable of handling large-scale Rust projects. Other unofficial Rust compilers, such as Rust-GCC [Rust-GCC 2024], remain in early development stages and lack the maturity for real-world use. Moreover, their bug histories are more related to build processes, such as cleanup [Xia et al. 2023], rather than Rust-specific features. In particular, our study answers the following research questions.

- **RQ1 (Bug Causes): What are the main causes of Rust-specific bugs in `rustc`?** What is the frequency of these bug causes? Which stages/components in `rustc` are more prone to bugs?
- **RQ2 (Symptoms): What are the symptoms of Rust-specific bugs in `rustc`?** What is the frequency of these symptoms? What is the relationship between bug causes and symptoms?
- **RQ3 (Test Case Characteristics): What are the main characteristics of the bug-revealing test cases?** What kind of test settings are required to trigger `rustc` bugs?
- **RQ4 (Status of Existing Tools): What are the existing testing techniques for `rustc`?** What kind of bugs can they detect? What are their limitations?

To answer these questions, we collect a list of issues and their corresponding pull requests from Rust's official GitHub [Rust 2023a] over the past three years. Each bug is manually labeled with its symptoms, cause, and the compilation stage where it occurs. Further, we identify frequently used language features and analyze their occurrences in both test cases and compilation commands. Moreover, we investigate the existing testing tools for `rustc` and discuss their advantages and limitations, as well as the challenges and difficulties in testing `rustc`.

**Contributions.** The contributions of this paper can be summarized as follows.

- We manually construct a dataset of Rust-specific bugs in `rustc`, covering the past three years. It includes test cases, issues, fixes, providing a foundation for our study and future research on testing and verifying `rustc`.
- We conduct a comprehensive empirical study on Rust-specific compiler bugs to understand the diverse dimensions of `rustc` bugs, including bug causes, bug-prone compilation stages, symptoms, and test case characteristics.
- Based on our analysis, we enumerate the implications of our findings, providing actionable suggestions for Rust users, `rustc` developers, and programming language researchers to shed light on detecting `rustc` bugs and improving its design.

**Summary of findings.** Some representative findings include:

Table 1. Rust-specific bug labels in `rustc` and the corresponding descriptions [rust team 2025].

Category	Label	# Num	Description
HIR	A-HIR	20	The high-level intermediate representation (HIR)
	A-THIR	1	Typed HIR
MIR	A-MIR	43	Mid-level IR (MIR) - <a href="https://blog.rust-lang.org/2016/04/19/MIR.html">https://blog.rust-lang.org/2016/04/19/MIR.html</a>
	A-mir-opt	78	MIR optimizations
	A-mir-opt-inlining	23	MIR inlining
	A-mir-opt-GVN	0	MIR opt Global Value Numbering (GVN)
	A-mir-opt-nrvo	0	Fixed by the Named Return Value Opt. (NRVO)
	A-stable-MIR	1	stable MIR
Type	A-type-system	25	Type system
	A-inference	29	Type inference
	A-closures	29	Closures ( ... {...})
	A-coercions	13	implicit and explicit expr as Type coercions
	A-const-generics	70	const generics (parameters and arguments)
	A-DSTs	0	Dynamically-sized types (DSTs)
	A-zst	0	Zero-sized types (ZST).
	A-trait-system	77	Trait system
	A-impl-trait	68	Universally/existentially quantified anonymous types with static dispatch
	A-trait-objects	27	trait objects, vtable layout
	A-auto-trait	14	auto traits (e.g., auto trait Send {})
	A-implied-bounds	9	Implied bounds / inferred outlives-bounds
	A-coinduction	0	Concerning coinduction, most often for auto traits
	A-coherence	14	Coherence
Lifetimes	A-lifetimes	70	Lifetimes / regions
	A-borrow-checker	46	The borrow checker

- (1) The `rustc`-specific IRs and components are prone to bugs due to the complex interplay of ownership, lifetimes, and trait resolution. In the HIR-processing component, most bugs (51.11%) stem from type resolution and well-formedness checks, while MIR-related bugs mainly relate to MIR transformation (50.00%).
- (2) Crash is the most common symptom (39.87%), followed by correctness issues (25.91%), where valid programs are mistakenly rejected or invalid ones are accepted. These often stem from the unique type checker and borrow checker within `rustc`. While existing tools can detect many crash bugs, they struggle with deeper correctness and misoptimization bugs.
- (3) Key contributors to `rustc` bugs include unstable features (24.25%) and specific compilation settings or optimization levels (18.94%). Features like trait objects often introduce edge cases that evade conventional testing, and their interactions with core language mechanisms can expose soundness and correctness issues.
- (4) Existing testing tools have detected only 6.07% of non-crash bugs, likely due to gaps in program generation. Current approaches lack support for Rust-specific features like higher-order trait bounds, advanced lifetime annotations, and complex borrowing, limiting the detection of correctness-critical issues in valid Rust programs.

## 2 Study Methodology

Our bug collection and analysis approach can be summarized in several steps. Firstly, we perform bug data collection, which is detailed in Section 2.1. We collect issues and corresponding pull requests from the official Rust GitHub repository, categorized by issue labels. Then, in the next step of post-filtering, we filter out certain issues, such as duplicate bugs and those without test cases. The final dataset consists of a series of bugs related to `rustc` IRs within a specific time range (from 2022-01-01 to 2025-01-01), along with their corresponding test cases and fix patches. Then, the resulting dataset serves as the foundation for our bug analysis approach, detailed in

Table 2. Status and description of collected bugs.

Status	Description	# Num
Duplicate	The bug duplicates other bugs that have already been confirmed.	88
Not a bug	It is not a bug because the feature is intentional and designed this way.	28
Not reproducible	When the developer confirmed the bug, it was no longer reproducible.	56
Discussion	(1) A question about a certain feature; (2) Suggestions for <code>rustc</code> improvement, but not a bug.	6
Exclude	(1) Does not contain a test case; (2) Unrelated to <code>rustc</code> ; (3) Not reproducible on 2021 edition.	92
Valid	The bug has been confirmed as a <code>rustc</code> bug, with a corresponding test case and fix.	301
<b>Total</b>	-	<b>571</b>

Section 2.2. The bug analysis is conducted iteratively by four researchers. In each iteration, two researchers manually analyze a random sample of bugs, categorize them based on various aspects, and cross-validate the results. When they can not reach a consensus, two additional researchers verify the categorization, resolving conflicts through discussion until a consensus is reached.

## 2.1 Collecting Bugs and Fixes

To collect bugs related to specific components of `rustc`, we select issues from the official tracker with relevant labels. The Rust team categorizes various labels in the tracker, which currently contains 814 labels in total. Labels prefixed with "A-" indicate different areas of the compiler, corresponding to various Rust language features and components, such as *A-rustdoc*. Table 1 lists the selected labels, descriptions, and occurrence counts. We focus on labels associated with Rust-specific IRs (HIR and MIR) and key language features, including the type system and memory management. Since an issue can have multiple labels, the total label count exceeds the actual number of collected issues. Although some dedicated tags are provided in the issue tracker, they are not suitable for our data collection scenario. For example, *C-bug* label includes all errors, including those related to parsing, which are beyond the scope of this study. Similarly, the *T-compiler* label indicates issues assigned to the developer team, but also includes non-bug issues. In addition, issues are not necessarily tagged with these two labels simultaneously.

To ensure that all collected bugs have been reviewed by developers and have corresponding patches, only closed issues are selected. The time frame for collected issues is chosen to cover the period from January 1, 2022, to January 1, 2025. This period aligns with the usage time of the latest Rust 2021 Edition, which was released on October 21, 2021, while Rust 2024 is scheduled for release on February 20, 2025. We crawled all closed issues that met the time frame and contained at least one tag listed in Table 1, resulting in a total of 571 issues. Then, a preliminary manual post-filtering is performed to exclude unsuitable issues, with the classification criteria outlined in Table 2. Specifically, The *duplicate* status refers to issues that have already been reported and confirmed, typically marked by developers as "closed as a duplicate." Issues labeled as *not a bug* indicate that they are not actual bugs but rather expected behaviors. Some issues are categorized as *not reproducible* because they can no longer be reproduced, suggesting that the underlying bug has already been resolved. These issues are excluded due to the absence of an identifiable fix patch. The *discussion* category includes issues without test cases or specific symptoms and instead involves inquiries about usages or suggestions for alternative `rustc` implementations. Finally, the *exclude* issues are unrelated to `rustc` (e.g., documentation-related issues), or cannot be reproduced in the 2021 edition (e.g., issues that only occurred in the 2015 edition). Finally, a total of 301 issues are classified as valid bugs and used for the subsequent analysis in our study.

## 2.2 Analyzing Bugs

To better understand the nature of the Rust-specific bugs in `rustc`, and reduce the possibility of getting biased, we follow the existing bug analyzing approaches [Chaliasos et al. 2021; Drosos et al. 2024; Xiong et al. 2023] and uniformly study bugs by an iterative process. Specifically, in each iteration, we randomly select 20 bugs, which are then independently examined by two co-authors. Each co-author categorizes the bugs based on their symptoms, bug causes, and compilation stages according to their individual understanding. Following this, they cross-validate and discuss their classifications until a consensus is reached. In cases where consensus cannot be achieved, the other two co-authors join the discussion to make the final decision. This iterative process was repeated 16 times until all bugs were thoroughly analyzed. After 2-3 iterations, the independent categorization results of the first two co-authors showed a high degree of agreement, with around 95% of the cases resulting in consistent labeling that required no further discussion. The manual analysis process necessitated substantial domain-specific expertise in both the `rustc` implementation and the Rust programming language, requiring approximately six person-months to complete.

To answer RQ1 and RQ2, the first two co-authors assign every bug to categories based on (1) its symptom, (2) the compilation stage when `rustc` encounters the bug, and (3) its bug cause. Specifically, regarding the bug symptom, we analyze the descriptions in each bug report to identify discrepancies between the expected and actual behavior of `rustc`. For the compilation stage, we examine the fixes for each bug to pinpoint the exact compilation procedure and the component responsible for the issue. Regarding the bug causes, we review the test cases, fix patches, and developer discussions to determine the root causes of each bug. To answer RQ3, we collect test cases from each issue, including Rust programs and compilation commands. We then extract their abstract syntax trees to identify frequent node types and analyze frequently used features and compilation commands. To address RQ4, we gather all existing tools for testing `rustc` and analyze the types and number of bugs they have identified. Additionally, we run each tool for 12 hours on a specific `rustc` version to assess their effectiveness.

## 3 RQ1: Bug Causes

In our collected bug list, each issue is linked to a corresponding fix pull request (PR). We analyze PR descriptions and code changes to classify the bug causes, as shown in Table 3. Three categories are closely tied to Rust’s language mechanisms: *the type system*, *the ownership system*, and errors from *MIR optimizations*. Other categories include bugs from *basic Rust syntax implementation in rustc*, *error handling and reporting*, and *compatibility issues*. The following sections provide a detailed discussion of each bug cause, with examples for clarification.

### 3.1 Type System Errors

Type system errors are a major cause of bugs in `rustc`, accounting for 30.23% of all cases. These issues stem from `rustc` handling Rust’s complex type mechanism, which emphasizes zero-cost abstractions, allowing high-level, expressive code without runtime overhead. For example, traits enable polymorphism, and generics allow code to operate on multiple types while maintaining type safety. However, their interaction with Rust’s other mechanism such as the ownership model introduces significant complexity, often leading to intricate type relationships and related bugs. We classify an error as a type system error if `rustc` fails to correctly handle Rust’s type mechanisms, leading to incorrect behavior or a compilation failure. Type system-related bugs belong to one of the following groups: (1) *trait & bound related errors*, (2) *opaque types related errors*, (3) *new solver related errors*, or (4) *well-formedness related errors*.

Table 3. The taxonomy of bug causes.

Category	Subcategory	Description	# Bugs	Ratio
Type System Errors	Trait & Bound	The errors were caused by rustc's handling of traits and its enforcement of type parameter constraints, such as requiring specific traits or conditions.	37	12.29%
	Opaque types	The errors were caused by issues within rustc's handling of opaque types, which rely on the ownership system, zero-cost abstractions, and the design of generics and traits.	38	12.62%
	New solver	The errors are caused due to the interaction between rustc's new solver, which is designed to improve trait-bound resolution and reduce workload, and the existing old solver.	7	2.33%
	Well-formedness	The errors were caused by rustc's well-formedness checking, including ownership, lifetime, type system, and the borrow checker.	9	2.99%
	<b>Subtotal</b>	-	<b>91</b>	<b>30.23%</b>
Ownership & Lifetime Errors	Borrow & Move	The errors were caused by issues in implementing the ownership model, which ensures memory safety and concurrency safety through the move and borrow semantics.	7	2.33%
	Lifetime	The errors were caused by issues in rustc's lifetime checking, which ensures that every reference is valid and does not outlive the data it points to.	34	11.30%
	<b>Subtotal</b>	-	<b>41</b>	<b>13.62%</b>
MIR Optimization Errors	Wrong implementations	The errors were caused by incorrect implementations of rustc's MIR-based optimizations (e.g., constant propagation, dead code elimination, inlining).	34	11.30%
	Missing cases	Some specific corner cases of the optimization algorithm were not considered thoroughly.	12	3.99%
	<b>Subtotal</b>	-	<b>46</b>	<b>15.28%</b>
General Errors	Basic structure	Bugs caused by rustc errors in processing features like closures and internal data structures.	38	12.62%
	Error handling & Reporting	The errors were caused by rustc's failure to handle exceptional cases properly or its misprocessing of reports, leading to misleading error messages or incorrect error locations.	75	24.92%
	Compatibility	The bugs were triggered by certain operating systems, bugs in the back-end LLVM, or errors specific to the Rust edition.	10	3.32%
	<b>Subtotal</b>	-	<b>123</b>	<b>40.86%</b>

**Trait & Bound Related Errors:** Trait-related errors account for 12.29% of all bugs. Traits define shared behaviors across types, while bounds constrain the types that can be used with generics. These bounds work with traits to ensure type safety and enable polymorphism. Errors in this category occur when rustc struggles to resolve trait bounds or apply constraints during type inference or checking. Typically, this happens when rustc fails to match types to their associated trait bounds, leading to incorrect type assignments or failure to resolve the required traits.

**Opaque types Related Errors:** Opaque types allow defining a type alias that only exposes certain traits as its interface. The actual concrete type is inferred from its usage in the code context [rustc-dev guide 2025]. Examples include types introduced by `impl Trait` and associated types within traits. For rustc, handling opaque types requires resolving these types and their associated properties during type checking and inference while maintaining their abstraction across different scopes. Errors in this category, which account for 12.62% of all causes, occur when rustc encounters difficulties in properly resolving opaque types or their associated properties, often due to scope-related issues. These challenges can lead to incorrect behavior, such as type mismatches or compilation failures, revealing flaws in rustc's type resolution for opaque types.

**New solver Related Errors:** The Rust team has been actively developing and integrating a new trait solver to replace some of the existing core implementations [Rust 2025a]. This effort aims to address unsoundness issues in the previous solver and enhance compilation efficiency. Currently, both the old and new trait solvers coexist within rustc, leading to challenges during the transition. Errors in this category, accounting for 2.33% of all causes, usually result from issues in the new trait solver, especially when resolving complex trait bounds.

**Well-formedness Related Errors:** Well-formedness (WF) [Rust 2025b] ensures that declarations in a Rust program follow its language's rules, validating types, bounds, and relationships. The WF checker generates a logical goal for each declaration and attempts to prove it using the type system's rules. If successful, the declaration is deemed well-formed; otherwise, an error is reported. Errors in this category, accounting for 2.99% of all causes, arise from rustc improperly processing WF checking, leading to incorrect behaviors or ICEs when validating the well-formedness.

**Example.** The patch in Figure 2 (a) addresses a WF-related error (tracked as [Issue 118876](#)) caused by incorrect WF checking for built-in traits. The built-in `Fn*` traits, including `Fn`, `FnMut`, and `FnOnce`, allow closures to be used like function pointers, passed as arguments, or stored in structs. Before explaining the bug cause, we first clarify some definitions. The unnormalized signature refers to function signatures that may include unresolved associated types, whereas the normalized signature resolves all associated types to their concrete definitions. Rust's type system assumes that if a type is well-formed, its normalized form is also well-formed. As a result, `rustc` only checks the WF of the unnormalized signature and ignores the normalized form during type checking. However, this assumption is violated because the implementations of built-in `Fn*` traits do not explicitly declare certain required lifetime bounds, particularly the '`'s`: `'static` bound. Consequently, `rustc` fails to enforce these implicit lifetime bounds, leading to an unexpected compiler behavior. The patch in Figure 2 (a) adds checks for the normalized signature to ensure that all associated types are resolved and necessary lifetime bounds are explicitly declared. This ensures that `rustc` applies the same WF rules to both built-in `Fn*` traits and user-defined traits.

```
// compiler/rustc_borrowck/src/type_check/mod.rs
- let sig = self.normalize(sig, term_location);
+ let sig = self.normalize(unnormalized_sig, term_location);
// WF(sig) does not imply WF(normalized(sig)) with built-in
// `Fn` implementations, since the impl may not be well-formed itself.
+ if sig != unnormalized_sig { ... }
```

(a) Type System Errors: Well-formedness related errors.

```
// compiler/rustc_borrowck/src/type_check/input_output.rs
- if body.yield_ty().is_some() != universal_regions.yield_ty.is_some() { ... }
+ if let Some(mir_yield_ty) = body.yield_ty() {
+   let yield_span = body.local_decls[RETURN_PLACE].source_info.span;
+   ...
+ }
```

(b) Ownership &amp; Lifetime Errors: lifetime errors.

Fig. 2. Two snippets of fix patch for explaining ownership & lifetime bug cause ([PR 118882](#) and [PR 119563](#)).

### 3.2 Ownership & Lifetime Errors

Rust's ownership and lifetime system is a core language feature, ensuring memory safety without needing a garbage collector. Bugs in `rustc` related to ownership and lifetime errors account for 13.62% of all causes. Specifically, `rustc` checks whether references are valid for their expected lifetimes, ensuring that mutable and immutable references do not conflict, and enforcing the ownership rules that prevent data from being accessed after it has been moved or dropped. We classify an error as an ownership and lifetime error when Rust's ownership model fails, causing compilation issues. These bugs belong to one of the following groups: (1) *borrow & move related errors* or (2) *lifetime related errors*.

**Borrow & Move Related Errors:** The borrow and move mechanisms are fundamental to Rust's ownership system, yet bugs arising from them are relatively rare, accounting for only 2.33% of all identified causes. The borrow model enables references to a value without transferring ownership, permitting either multiple immutable references or a single mutable reference, but never both simultaneously. The move model, in contrast, transfers ownership of a value, rendering the original variable invalid and preventing further use. Bugs in this category typically stem from `rustc` mismanaging mutable and immutable borrowing or incorrectly tracking ownership transfers.

**Lifetime Related Errors:** The lifetime is a key feature of Rust's ownership system, which describes the scope for which a reference is valid, preventing issues like dangling references or data races. The bugs caused by lifetime-related errors account for 11.30% of all. The borrow checker in `rustc` utilizes lifetimes to track the validity of references and enforce that they do not outlive the referenced data. The errors caused by this category are typically because `rustc` improperly infer or check the lifetimes of references.

**Example.** The patch in Figure 2 (b) shows an example of lifetime-related errors. In the test case (tracked as [Issue 119564](#)), `coroutines` for asynchronous programming are utilized. Unlike traditional functions, `coroutines` in Rust allow execution to be paused and resumed at different points, forming an implicit state machine. This mechanism introduces challenges for the borrow checker, as it must ensure that all references captured inside the `coroutine` remain valid across

suspension points. However, in this case, `rustc` failed to properly enforce lifetime constraints on values produced by `yield`, allowing a yielded value to be assigned a stricter lifetime than it should have. Since `yield` effectively acts as a suspension point, any borrowed reference tied to it must remain valid when the `coroutine` resumes. Without proper checks, this could lead to dangling references or memory safety violations. The patch shown in Figure 2 (b) improves soundness in `rustc`'s coroutine handling by enforcing stricter lifetime checks at yield and resumption points. When a `yield` expression is detected, `rustc` captures the `yield_span` to determine the scope of the yielded value. Then, `rustc` uses this span to perform further checks, ensuring that `coroutines` correctly enforce lifetime constraints.

### 3.3 MIR Optimization Errors

MIR optimization in `rustc` refines MIR to enhance performance and reduce resource consumption. These optimizations, including constant folding, dead code elimination, and loop unrolling, refine the code before it is passed to the backend compiler. While most algorithms have been implemented within classic compilers, applying them to MIR can introduce subtle interactions and edge cases. Bugs arising from these challenges, categorized as MIR optimization errors, account for 15.28% of all causes. An MIR optimization error occurs when incorrect transformation or optimization causes misbehavior or compilation failure. These bugs fall into two categories: (1) **wrong implementations**, where `rustc` incorrectly implement the intended transformations (11.3%), and (2) **missing cases**, where certain corner cases or program patterns are not properly addressed, leading to incomplete optimizations (3.99%). From our study, most MIR optimization bugs require modifications to the algorithm's logic, rather than merely fixing a minor overlooked case.

**Example.** Figure 3 (a) illustrates an example of incorrect MIR optimization. The bug (tracked as Issue 111355) occurs when inlining results in redundant unreachable blocks. It is caused by the interaction between two key MIR optimization passes: `InstCombine`, which simplifies instructions by combining constant expressions and redundant operations, and `SimplifyCfg`, which simplifies control flow graphs by removing unnecessary branches and loops. Initially, the function responsible for merging duplicate targets was placed within `InstCombine`, but this placement was ineffective because `InstCombine` runs before `SimplifyCfg`. Since duplicate unreachable blocks are only introduced after `SimplifyCfg` is applied, the function was executed too early to have the intended effect. The patch corrects this by relocating the function, ensuring it properly merges duplicate unreachable blocks when they actually appear.

```
// compiler/rustc_mir_transform/src/instcombine.rs
- fn combine_duplicate_switch_targets(...) {...}
// compiler/rustc_mir_transform/src/simplify.rs
+ fn combine_duplicate_switch_targets(...) {...}
```

(a) MIR optimization errors: wrong implementations.

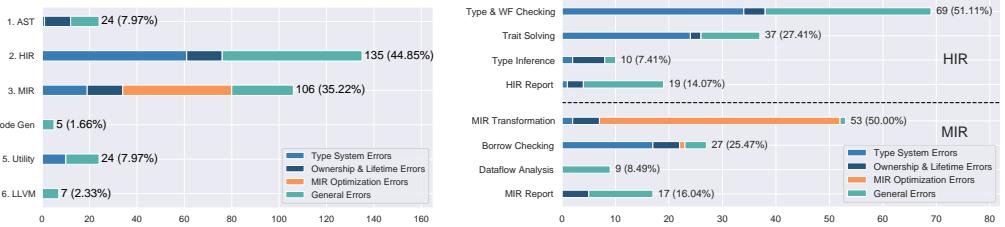
```
// compiler/rustc_hir_typeck/src/fn_ctxt/checks.rs
- let is_closure = matches!(arg.kind, ExprKind::Closure { .. });
+ let is_closure = if let ExprKind::Closure(closure) = arg.kind { ... }
+ else {false};
```

(a) Other general errors: basic syntax &amp; structure.

Fig. 3. The fix patch for explaining MIR optimization bug cause PR 110569 and general errors PR 112266).

### 3.4 General Errors

The remaining bug causes are not directly tied to core language features but instead, result from more fundamental issues in how `rustc` processes certain constructs, handles edge cases, or interacts with its backend systems. These errors can stem from various issues within `rustc`'s internal logic, structure, or interaction with external components. These bugs account for 40.86% of all causes. We classify an error as a general error when flaws in `rustc`'s design or implementation cause expected compilation behaviors. General errors in `rustc` can be classified into three categories: (1) **basic structure errors**, where `rustc` incorrectly processes fundamental constructs, such as



(a) Bug distribution across `rustc` compilation stages. (b) Bug distribution across HIR and MIR components.

Fig. 4. Comprehensive analysis of bug distribution in `rustc` pipeline and its HIR/MIR components.

closures or internal data structures (12.62%); (2) **error handling and reporting issues**, where exceptional cases or error reports are mishandled, leading to misleading messages or incorrect error locations (24.92%); and (3) **compatibility issues**, where bugs arise from specific operating system configurations, backend LLVM problems, or Rust edition-specific errors (3.32%).

**Example.** Figure 3 (b) illustrates an example of basic structure errors, which is a regression in Rust 1.70 (tracked as [Issue 112225](#)) affecting type inference in argument-position closures and `async` blocks. The issue arises from how `rustc` evaluates `async` blocks, where improper closure handling leads to incorrect type resolution. Unlike regular functions, `async` blocks are implicitly transformed into state machines, which affects closure inference and evaluation order. This transformation caused `rustc` to misidentify closures in arguments, leading to inference failures. The patch adds an explicit check to verify whether an argument is a closure, preventing misclassification.

### 3.5 Bug Prone Compilation Stages

The workflow of `rustc` involves several specific components, including HIR and MIR, as well as various specialized checks and analyses based on these IRs that support Rust’s unique memory management system. To investigate the stages of `rustc` compiler pipeline prone to bugs, we decompose its workflow and divide it into several core stages. We then quantify the error rates at each stage and analyze the underlying causes. In some cases, a bug involves modifications across multiple stages. To handle such cases, we identify all affected modules in the fixing PR and trace the root cause to the stage where the error originates. Figure 4(a) provides an overview of the distribution of bug causes across different compilation stages. General errors appear throughout all stages, while MIR optimization bugs predominantly occur in the MIR-processing stage. Beyond HIR-processing and MIR-processing, most bugs stem from general errors.

To further understand the bugs triggered in the core HIR and MIR components, we subdivide them and investigate their bug causes, as shown in Figure 4(b). A closer look at the HIR and MIR processing stages reveals that bugs related to the type system and ownership mechanisms are spread across multiple components rather than being isolated to a single stage. Regarding Figure 4(b), most components contain bugs caused by type system errors. For instance, issues in *type & WF checking* may allow invalid types, while errors in *trait solving* can lead to unexpected type mismatches. Bugs in *MIR transformation* and *borrow checking* can also stem from type system errors. This is partly because some WF checks are performed during borrow checking, as certain lifetime information may still be incomplete during the HIR-based type-checking phase. Similarly, multiple components are affected by ownership and lifetime errors. In the *type & WF checking* and *type inference* components, incorrectly inferred types and constraints can lead to unsound borrowing rules. Additionally, incorrect trait resolution in the *trait solving* may introduce errors that propagate to later stages, ultimately affecting ownership analysis. Errors in *borrow checking* can

directly cause ownership-related issues. Furthermore, the *MIR transformation* involves optimization algorithms related to lifetimes, which can also introduce related issues. Regarding *error reporting* in both the HIR and MIR components, most bugs stem from general programming errors, especially improper error handling, which can misclassify the root causes of compilation failures.

#### 4 RQ2: Bug Symptoms

To categorize the bug symptoms of `rustc`, we manually review bug descriptions from GitHub's bug reports and analyze the discrepancies between expected and actual behaviors. Specifically, we categorized the bugs into five distinct bug symptom categories: *Crash*, *Correctness Issues*, *Miscompilation*, *Diagnostic Issues* and *Misoptimization*. The distribution of bug symptoms is shown in Table 4. Among them, crashes are the most prevalent, accounting for 39.87% of cases, followed by correctness issues (25.91%) and diagnostic issues (19.27%). Miscompilation and misoptimization are less common, making up 9.97% and 4.98%, respectively.

Table 4. Distribution of bug symptoms and the distribution of bug symptoms per cause.

Symptoms		Occurrence		Type System	Ownership & Lifetime	MIR Optimization	General Errors
Crash	Front-end Panic (valid)	42 (13.95%)	120 (39.87%)	30 (25.00%)	3 (2.50%)	19 (15.83%)	68 (56.67%)
	Front-end Panic (invalid)	75 (24.92%)					
	Back-end Crash	3 (1.00%)					
Correctness Issues	Completeness Issues	56 (18.60%)	78	43 (55.13%)	17 (21.79%)	7 (8.97%)	11 (14.10%)
	Soundness Issues	22 (7.31%)	(25.91%)				
Miscompilation	Inconsistent Output Issues	18 (5.98%)	30	5 (16.67%)	4 (13.33%)	12 (40.00%)	9 (30.00%)
	Safe Rust Causes UB	12 (3.99%)	(9.97%)				
Diagnostic Issues	Incorrect Warning/Error	20 (6.64%)	58	12 (20.69%)	16 (27.59%)	1 (1.72%)	29 (50.00%)
	Improper Fix Suggestion	38 (12.62%)	(19.27%)				
Misoptimization		15 (4.98%)	15 (4.98%)	1 (6.67%)	1 (6.67%)	7 (46.67%)	6 (40.00%)

#### 4.1 Crash

Similar to all software systems, `rustc` also suffers from crashes. Among the bugs we collected, 36.51% involve crash errors. Based on the compilation stage where the crash occurs, we categorize them into front-end panics and back-end crashes.

**Front-end Panic:** In Rust, a panic occurs on an unrecoverable error, followed by a cleanup operation before termination. In `rustc`, an internal compiler error (ICE) often manifests as a panic, indicating that `rustc` has encountered an unexpected state or an unhandled scenario. The front-end panic accounts for 38.87% of all observed symptoms. Among them, 13.95% are triggered by valid programs, and 24.92% are triggered by invalid programs.

**Back-end Crash:** Back-end crashes happen due to low-level failures like segmentation faults (SIGSEGV) or abnormal terminations (SIGABRT), often linked to issues with code generation or LLVM. The proportion of back-end crashes is notably minimal, accounting for 1% of all cases.

**Bug cause analysis.** The primary causes of crash bugs in `rustc` are general errors (56.67%), such as inadequate error handling and compatibility issues. When `rustc` encounters an unexpected program state, its error recovery mechanisms may be incomplete, leading to an ICE instead of graceful handling. The second major category involves the type system (25.00%) and MIR optimization (15.83%). The complexity of type checking and trait resolution can introduce subtle inconsistencies, especially with advanced generics and associated types. Additionally, since MIR serves as the bridge between high-level Rust code and low-level machine code, incorrect optimizations or misinterpretations of type transformations at this stage can also lead to ICE. Finally, ownership and

lifetime errors account for 2.5% of crash bugs. As most violations in this area are caught at compile time, incorrect checks are more likely to cause correctness issues rather than immediate crashes.

```
pub fn main() {  
    main(arr[i]);  
}
```

thread 'rustc' panicked at compiler/rustc\_span/src/lib.rs:2028:17:  
assertion failed: bpos.to\_u32() >= mbc.pos.to\_u32() + mbc.bytes as u32

(a) A Rust program that triggers an ICE.

```
// compiler/rustc_hir_typeck/src/fn_ctxt/checks.rs  
- call_expr.span.with_lo(call_expr.span.hi()) - BytePos(1))  
+ self.tcx().sess.source_map().end_point(call_expr.span)
```

(b) The fix patch. General errors: Error handling& Reporting.

Fig. 5. The example of a crash bug ([Issue 128717](#)) and corresponding fix patch ([PR 128864](#)).

**Example.** Figure 5 (a) shows a code snippet that triggers an ICE, which is caused by a *general error*. This code incorrectly passes an argument into the main function and uses a multi-byte brace as the closing delimiter. When rustc detects that the main function involves parameters, it attempts to provide a fix suggestion and shifts one byte to remove the extra parameter. However, rustc fails to handle multi-byte characters because it assumes that every closing delimiter is a single byte. This misalignment violates Unicode boundaries, triggering an assertion failure. Figure 5 (b) presents the fix patch, which corrects the positioning approach by eliminating the use of BytePos.

## 4.2 Correctness Issues

Correctness bugs occur when rustc fails to enforce Rust's syntax rules, leading to the unintended rejection or acceptance of programs, undermining its ability to accurately validate Rust programs. The correctness issues account for 25.91% of all cases. Among them, 18.60% are triggered by completeness issues, and 7.31% are triggered by soundness issues.

**Completeness Issues:** Completeness bugs occur when rustc fails to compile a syntactically and semantically valid Rust program as defined by the language specification. These bugs typically manifest when rustc incorrectly rejects such a program, either by displaying a false error message or failing to complete compilation.

**Soundness Issues:** Soundness bugs refer to situations where rustc mistakenly accepts programs that should be rejected due to violating language rules. Rust is known for its strict rules around syntax and semantics, and soundness bugs occur when rustc incorrectly allows code that violates these rules to compile successfully.

**Bug cause analysis.** Correctness bugs in rustc primarily stem from issues in the type system (55.13%) and ownership management (21.79%). Unlike crash bugs, correctness issues are not as immediately apparent, as they often result from logical flaws in rustc's core checking mechanisms rather than explicit failures. Other causes, such as MIR optimization errors (8.97%) and general errors (14.10%), are relatively less common. MIR optimization bugs can introduce subtle miscompilations when incorrect transformations alter program semantics, particularly in aggressive optimization scenarios. General errors, including missed edge cases in rustc logic, may propagate inconsistencies, leading to undetected violations of Rust's safety guarantees.

```
#![allow(async_fn_in_trait)]  
pub trait foo {}  
pub trait bar { type foo: foo; }  
pub trait baz { async fn boom<X: bar>() -> Result<(), X::foo>; }
```

> Actual output: error[E0220]: associated type 'foo' not found for 'X'  
8 | async fn boom<X: bar>() -> Result<(), X::foo>;  
 | ^^^ there is an associated type 'foo' in the trait 'bar'  
> Expected output: Compilation succeeds

(a) A Rust program that triggers a completeness bug of rustc.

```
// compiler/rustc_hir_analysis/.../predicates_of.rs  
+ match tcx.opt_rpitit_info(item_def_id.to_def_id()) {  
+     Some(ty::ImplTraitInTraitData::Trait {..}) => {...}  
+     Some({..}) => unreachable!(..)  
+     None => {}  
+ }
```

(b) The fix patch. Type System Errors: Opaque types related errors.

Fig. 6. The example of a correctness issue ([Issue 132372](#)) and corresponding fix patch ([PR 132373](#)).

**Example.** Figure 6 (a) presents a test case that exposes a *correctness issue*, which is caused by handling *opaque types* and Return-Position Impl Trait in Trait (**RPIITs**), categorized under *type system errors*. The test case defines three public traits: foo, bar, and baz. The baz trait includes an asynchronous method, boom, which is generic over a type X constrained to implement bar. Here, the asynchronous method can be defined in the trait because the corresponding unstable feature

is enabled. While this code previously compiled successfully, it now fails with the latest `rustc` version. Although the test case does not explicitly use `impl Trait`, the `async` function implicitly returns `impl Future<Output=T>`, thereby involving RPITITs. The root cause of the bug is that RPITITs are incorrectly assigned the `def id` of a Generalized Associated Type (GAT) instead of the correct opaque type identifier. Consequently, shorthand projections such as `T::Assoc` fail to resolve properly. The patch in Figure 6 (b) corrects this by modifying `rustc` to detect cases where an item originates from RPITIT lowering and ensuring that queries are forwarded to the appropriate item. As this case demonstrates, flaws in the complex type system can lead to correctness issues, underscoring the challenges of maintaining a reliable type system in Rust.

#### 4.3 Miscompilation

Miscompilation bugs occur when `rustc` generates incorrect machine code or behaves unexpectedly during compilation, leading to incorrect program execution. Miscompilation issues are particularly important, as they may compromise the safety and performance guarantees that Rust provides to its users. Bugs classified as miscompilation account for 9.97% of the total.

**Inconsistent Output Issues:** These bugs arise when `rustc` produces different outputs based on compilation levels or optimization settings. Rust's debug and release modes apply varying optimizations, but miscompilation can cause inconsistencies in both the generated machine code and the program's execution results across configurations. Bugs classified as inconsistent output issues account for 5.98% of all symptoms.

**Safe Rust Program Causes Undefined Behaviors:** This symptom is unique to `rustc`, as it guarantees that safe Rust code does not cause undefined behavior (UB). UB occurs when a program executes code that is not defined by the language specification, leading to unpredictable or incorrect results [Wikimedia 2004]. However, a `rustc` bug is triggered if a safe Rust program causes UB, which accounts for 3.99% of all symptoms.

**Bug cause analysis.** Miscompilation bugs in `rustc` are primarily caused by MIR optimization errors (40.00%). Faulty optimization logic can lead to semantic differences between optimized and unoptimized code, directly affecting program correctness. General errors account for 30.00%, as mistakes in internal data structures or improper handling of basic syntax can propagate through the compilation process, leading to incorrect code generation. Other causes, including type system issues (16.67%) and ownership-related errors (13.33%), are relatively less common. Type system bugs may lead to miscompilations due to incorrect type inference or trait resolution. Similarly, ownership errors could result in unintended memory access patterns, potentially causing miscompilations.

```
pub fn myfunc() -> i32 { % rustc -Awarnings test.rs -Zmir-opt-level=0 && ./test
    let mut a: i32 = 0; a = 1;
    if !a != 0 { return 1 }
    return 0;
}
pub fn main() { let mut e = myfunc(); println!("e={}", e); }
```

(a) A Rust program that triggers inconsistent output.

```
// compiler/rustc_mir_transform/src/jump_threading.rs
Rvalue::UnaryOp(Unop::Not, Operand::Move(place) | Operand::Copy(place)) =>
+ if !place_ty(self.body, self.tcx).ty.is_bool() {
+ // ...
+ if !a == b` is not `a != b` for integers greater than 1 bit.
+ return;
+ }
```

(b) The fix patch. MIR optimization errors: missing cases.

Fig. 7. The example of a miscompilation bug ([Issue 131195](#)) and corresponding fix patch ([PR 131201](#)).

**Example.** Figure 7 illustrates an *inconsistent output* bug, caused by a *MIR optimization error*. The test case defines a function `myfunc`, where the variable `a` is initialized as 0 with type `i32`. In Rust, the `!` operator performs bitwise negation on integers and logical negation on booleans. Therefore, applying `!` to `a` should produce a 32-bit value with all bits set to 1, equivalent to `-1` in two's complement representation, so the expected output is `e = 1`. However, under MIR optimization level 2, the actual result is `e = 0`, leading to an inconsistent output. The bug originates from the `jump_threading` optimization pass, where `rustc` incorrectly applies optimizations to non-boolean operands. The optimizer fails to differentiate between integer and boolean negation, causing

incorrect jump threading in specific cases. The patch in Figure 7 (b) resolves this by introducing a boundary check, ensuring that only boolean operands are considered for jump threading.

#### 4.4 Diagnostic Issues

`rustc` generates error messages for compilation failures and warnings for potential misuse, often accompanied by corresponding fix suggestions. Therefore, we subdivide diagnostic issues into two categories. In total, diagnostic issues account for 19.27%. Among them, incorrect warning/error issues account for 6.64%, and improper fixing suggestion issues account for 12.62%.

**Incorrect Warning/Error:** After the compilation, `rustc` may generate warning or error messages. Nevertheless, these messages may be inaccurate or deceptive.

**Improper Fixing Suggestion:** When handling invalid programs, `rustc` often provides fix suggestions, yet these may be imprecise, or there could be a more optimal recommendation.

**Bug cause analysis.** Diagnostic issues in `rustc` primarily stem from general errors (50.00%), including shortcomings in error handling and suggestion-matching mechanisms. Similar to ICEs, while `rustc` correctly identifies that the input program is non-compilable, incomplete error handling may lead to unclear diagnostics or ineffective fix suggestions. In addition, type system (20.69%) and ownership-related issues (27.59%) also contribute significantly, as imprecise error branch selection within these checkers can result in misleading or unclear messages. Finally, MIR optimization (1.72%) rarely causes diagnostic issues, as it involves minimal error reporting, but fix suggestions may still be affected by transformations like incorrect dead code elimination, which can remove useful code and lead to inaccurate suggestions.

```
pub fn foo<const BAR: bool> () {}
pub fn main() { foo::<{cfg!(feature = "foo")}>(); }
```

➤ Actual output: warning: unnecessary braces around const expression  
➤ Expected output: no warning message

(a) A Rust program that triggers a diagnostic issue.

```
// compiler/rustc_lint/src/unused.rs
... && !inner.span.from_expansion() {
    self.emit_unused_delims_expr(...)
}
```

(b) The fix patch. General errors: Error handling& Reporting.

Fig. 8. The example of a crash bug ([Issue 104141](#)) and corresponding fix patch ([PR 105515](#)).

**Example.** Figure 8 (a) shows a code snippet that triggers a warning message suggesting the removal of unnecessary braces around a const expression. However, this warning is not correct, and applying it leads to a compilation failure. Because the braces are required for const generics combining with the `cfg!` macro. This bug is classified as an error in *error handling and reporting*, which falls under *general errors*. Figure 8 (b) presents the corresponding fix, refining the linting process to exclude edge cases involving macros in const generics.

#### 4.5 Misoptimization

Misoptimization bugs in `rustc` occur during the optimization phase. While the final execution results may be correct, the MIR generated by `rustc` may not match the expected optimizations. Besides, intermediate compilation stages can introduce inefficiencies or subtle issues, affecting soundness or performance. In total, misoptimization issues account for 4.98% of all.

**Bug cause analysis.** Misoptimization bugs in `rustc` are primarily caused by MIR optimization errors (46.67%) and general errors (40.00%). Unlike crash or diagnostic issues, misoptimizations are not explicitly detected but instead manifest as deviations in the generated MIR from expected behavior. These issues often stem from flaws in MIR optimization algorithms or unhandled corner cases in `rustc`. Additionally, the type system (6.67%) and ownership-related issues (6.67%) contribute to a smaller portion of misoptimizations. In these cases, incorrect analyses can propagate errors into MIR lowering, leading to unintended transformations in the optimized code.

**Example.** Figure 9 illustrates a *misoptimization* caused by an *ownership & lifetime error*, specifically a *borrow and move error*. The issue arises in the `SimplifyLocals` optimization pass, which

removes unused variables and redundant code at the MIR level. As shown in Figure 9 (c), the optimizer incorrectly eliminates `_2` (a `usize` variable) and `_3` (a raw pointer of type `const T`), highlighted in red. Under Rust's strict provenance model, pointer-to-integer conversions must retain provenance information, as they encode the pointer's origin. While the program may still compile and execute, a deeper MIR-level analysis reveals deviations from expected behavior. The patch in Figure 9 (b) fixes this by introducing stricter validation for pointer-to-integer casts, ensuring they are preserved when necessary.

```
pub fn expose_addr<T>(a: &T) { a as *const T as usize; }
```

(a) A Rust program that trigger a misoptimization bug.

```
// compiler/rustc_middle/src/mir/mod.rs
- pub fn is_pointer_int_cast(&self) -> bool {
-   matches!(self, Rvalue::Cast(CastKind::PointerExposeAddress, _, _))
+ pub fn is_safe_to_remove(&self) -> bool {
+   match self {
+     Rvalue::Cast(CastKind::PointerExposeAddress, _, _) => false,
+     Rvalue::Use(_) | ... => true,
+   }
}
```

(b) The fix patch. Ownership & Lifetime Errors: borrow and move errors.

Fig. 9. The example of a misoptimization bug ([Issue 97421](#)) and corresponding fix patch ([PR 97597](#)).

```
// These lines are removed after SimplifyLocals optimization.
fn expose_addr(_1: &T) -> () {
  debug a => _1; // in scope 0 at a.rs:1:23: 1:24
  let mut _0: (); // return place in scope 0 at a.rs:1:30: 1:30
  let mut _2: usize; // in scope 0 at a.rs:2:5: 2:27
  let mut _3: *const T; // in scope 0 at a.rs:2:5: 2:18
bb0: {
  _3 = &raw const (*_1); // scope 0 at a.rs:2:5: 2:16
  _2 = move _3 as usize (Misc); // scope 0 at a.rs:2:5: 2:27
  return; // scope 0 at a.rs:3:2: 3:2
}
```

(c) The MIR before/after optimization.

## 5 RQ3: Test Case Characteristics

In this section, we analyze the characteristics of the bug-revealing test cases. By studying the characteristics and properties of these test cases, we can identify specific aspects of Rust that contribute to rustc bugs, offering guidance for test case design.

A test case consists of a Rust program and a compilation command, which we collect from each issue. If both original and reduced test cases are reported, we collect them separately. If only a single test case is reported, it is categorized as an original test case. If a minimized version is included in the corresponding comment, it is collected as a reduced test case; otherwise, we record the reduced case as the same as the original case. Additionally, if a bug is reproducible only within a separate Rust project, we exclude it, as the excessive presence of unrelated elements may obscure the test case characteristics. Among the 301 valid bugs, we have collected 276 original test cases and 293 reduced test cases. The number of original test cases is less than the reduced ones because some original cases are separate projects, which we do not collect, yet their reduced versions are included. There are 8 issues without test cases, due to: (1) unavailable external links, (2) separate projects, and (3) test cases with only compilation commands, not executable programs. To analyze the characteristics of test cases, we convert the reduced test cases into an AST and count the node types that reflect syntactic structures. Specifically, we use `syn` library [Tolnay 2025] to parse the AST and extract the occurrences of `item` and `type` nodes. The `item` nodes represent top-level constructs in Rust, such as functions, structs, traits, and enums, which define the overall structure of a program. The `type` nodes capture the different kinds of types in Rust, including primitives, references, and more complex types like trait objects, providing insights into how values are represented and manipulated in the code. Excluding a few cases where no test case is provided or where severe syntax errors prevent generating an AST, we collect a total of 293 test cases and 271 corresponding ASTs. Since `syn` library cannot parse Rust code fragments without a main function, we manually supplement such snippets. If the test case only defines items like functions or structs, we append an empty main function. Otherwise, if it contains statements, we wrap the entire snippet in a main function. Additionally, we also analyze features triggering rustc bugs from various perspectives, including unstable features, compilation flags, built-in traits, and other keywords or APIs. For the compilation command, we identify the most frequently used commands and their usage frequency.

Table 5 presents some general statistics on test cases. The average size of original test cases is 17.83 lines of code (LoC), while the median is 12 LoC. Since not every test case has a reduced version, the average LoC for reduced test cases is 14.17, with a median of 11. The difference from the original test cases is not significant, although the maximum LoC has decreased from 346 to 123. Based on these statistics, we could infer that rustc bugs are mainly triggered by small fragments of code.

Table 5. Statistics on test case sizes: lines of code (LoC).

	mean	median	min	max
Original tests	17.83	12	2	346
Reduced tests	14.17	11	2	123

Table 6. Summary of AST node types and their occurrence across test cases.

Item	Total	Prevalence	Mean per File	Max per File	Type	Total	Prevalence	Mean per File	Max per File
<b>Function</b>	524	100.00%	1.93	8	<b>Path</b>	1262	88.19%	5.28	41
<b>Struct</b>	130	37.64%	1.27	4	<b>Reference</b>	276	43.91%	2.32	10
<b>Impl</b>	157	37.64%	1.54	6	<b>Tuple</b>	161	30.26%	1.96	8
<b>Trait</b>	144	34.32%	1.55	6	<b>Impl Trait</b>	87	20.66%	1.55	10
<b>Use</b>	64	20.30%	1.16	3	<b>Array</b>	55	11.44%	1.77	10
<b>Type</b>	29	7.38%	1.45	6	<b>Trait Object</b>	49	10.70%	1.69	3
<b>Enum</b>	8	2.95%	1	1	<b>Ptr</b>	35	7.75%	1.67	4
<b>Macro</b>	11	2.95%	1.38	2	<b>Infer</b>	18	4.80%	1.38	2
<b>Extern Crate</b>	7	2.58%	1	1	<b>BareFn</b>	21	4.06%	1.91	5
<b>Static</b>	7	2.21%	1.17	2	<b>Slice</b>	13	2.95%	1.63	3
<b>Mod</b>	8	1.85%	1.6	3	<b>Never</b>	1	0.37%	1	1
<b>Const</b>	5	1.85%	1	1	<b>Paren</b>	1	0.37%	1	1
<b>Verbatim</b>	4	1.11%	1.33	2	<b>Group</b>	0	0.00%	0	0
<b>Foreign Mod</b>	2	0.74%	1	1	<b>Macro</b>	0	0.00%	0	0
<b>Trait Alias</b>	1	0.37%	1	1	<b>Verbatim</b>	0	0.00%	0	0
<b>Union</b>	0	0.00%	0	0					

Table 6 presents the distribution of *Item* and *Type* nodes across test cases. "Total" is the total occurrences of each node, "Prevalence" is the percentage of test cases containing the node, "Mean per File" is its average occurrences per file, and "Max per File" is the highest count in a single file. Among *Item* nodes, *function* is the most common node, followed by *struct*, *impl*, and *trait*, accounting for around 35%. These nodes often appear multiple times per file, suggesting that a test case usually defines several custom data structures. About 20% of test cases contain *use* statements, mainly for standard library imports and some third-party dependencies. *Type*, representing custom types like type aliases, appears in nearly 10%, common in Rust's trait-based generics for abstract and reusable code. Among data type nodes, *Path* is the most frequent, appearing in 88.19% of test cases. It represents the fully qualified name of types, e.g., `Vec<i32>, std::fs::File`. *Reference* is the second most common type, appearing in 43.91% of cases, reflecting Rust's ownership and borrowing system. This is also linked to *ptr* (raw pointers), which bypass safety checks in advanced use cases. Trait-related types such as *Impl Trait* (20.66%) and *Trait Object* (10.70%) support compile-time and runtime polymorphism, respectively.

The other features triggering rustc bugs are listed in Table 7. Around 25% of the test cases involved unstable features, while about 20% required specific compilation flags. In total, we identified 42 distinct unstable features and 41 different compilation flags. Many frequently used unstable features are applied to support advanced trait usages. The `generic_const_exprs` feature (17.81%) allows constant expressions in generic parameters, enabling more flexible compile-time computations. The `type_alias_impl_trait` feature (15.07%) simplifies complex trait bounds by allowing type aliases with *impl Trait*, making generic code more concise. Meanwhile, the `const_trait_impl` feature (4.11%) enables trait implementations in constant contexts, further extending Rust's compile-time capabilities. These features enhance Rust's type system but also introduce complexity to

trait resolution, type inference, and constant evaluation. The interplay of traits, generics, and compile-time computation boosts expressiveness while increasing the edge cases `rustc` must handle. Unstable trait-related features often reveal subtle issues in type checking, trait coherence, and monomorphization. Consequently, testing `rustc` becomes more challenging, as ensuring soundness while supporting richer abstractions demands rigorous validation against an increasingly intricate trait system. The other two unstable features are primarily related to low-level optimizations. The `core_intrinsics` feature (12.33%) provides direct access to compiler intrinsics for performance-critical operations. The `custom_mir` feature (10.96%) allows custom transformations on MIR, enabling experimental optimizations and analysis. The most common compilation flags are related to optimization. The `-Zmir-opt-level=X` (45.61%) and `-Copt-level=X` flag (14.04%) controls MIR and LLVM optimizations, respectively. The `-Zmir-enable-passes=+X` flag (15.79%) enables specific MIR passes. The `+nightly` flag (14.04%) specifies the nightly `rustc` version, and `-edition=X` specifies the Rust edition.

Table 7. The five most frequent unstable features and compilation flags required by test cases.

Most frequent unstable features		Most frequent compile flags		Most frequent traits		Other features	
Feature	Occ (%)	Flag	Occ (%)	Trait	Occ (%)	Feature	Occ (%)
<code>#![feature(generic_const_exprs)]</code>	17.81%	<code>-Zmir-opt-level=X</code>	45.61%	<code>(?)Sized</code>	49.23%	<code>lifetimes</code>	34.55%
<code>#![feature(type_alias_impl_trait)]</code>	15.07%	<code>-Zmir-enable-passes=+X</code>	15.79%	<code>FnOnce</code>	12.31%	<code>std API</code>	18.60%
<code>#![feature(core_intrinsics)]</code>	12.33%	<code>-Copt-level=X</code>	14.04%	<code>Iterator</code>	7.69%	<code>dyn</code>	9.97%
<code>#![feature(custom_mir)]</code>	10.96%	<code>+nightly</code>	14.04%	<code>Copy</code>	6.15%	<code>async</code>	7.31%
<code>#![feature(const_trait_impl)]</code>	4.11%	<code>-edition=X</code>	12.28%	<code>FnMut</code>	4.72%	<code>core API</code>	6.31%
<b>Total: 73</b>	<b>24.25%</b>	<b>Total: 57</b>	<b>18.94%</b>	<b>Total: 65</b>	<b>21.59%</b>	-	-

Given the frequent occurrence of traits in test cases, we further analyze the usage of built-in traits, which can increase test case complexity, as demonstrated by the examples in Section 3.1 and Section 3.2. Test cases involving at least one built-in trait account for 21.59% of all cases. Additionally, 18.60% of cases import standard library traits (`use std`), and 6.31% use core library traits (`use core`). This suggests that the flexible use of Rust’s built-in traits contributes to triggering `rustc` bugs. Table 7 shows the five most frequently used build-in traits. The `Sized` trait (49.23%) ensures a type has a known size at compile time, while `?Sized` allows dynamically sized types like `str` and `dyn Trait`. The `FnOnce` trait (12.31%) applies to types callable at most once, typically due to ownership constraints. The `Iterator` trait (7.69%) enables value generation, whereas `Copy` (6.15%) allows duplication via bitwise copying instead of moves. The `FnMut` trait (4.72%) permits multiple calls, modifying the captured environment each time. For other language features, lifetimes play a crucial role, with 34.55% of test cases using lifetime annotations. Additionally, the usage of `dyn` (for dynamic dispatch via trait objects) and `async` (for asynchronous programming) also contributes to detecting `rustc` bugs.

## 6 RQ4: Status of Existing Techniques

A major concern for developers is how to automate the testing and verification of `rustc` as it evolves. Several `rustc`-specific testing tools have been proposed by the Rust community and academia, and they differ in program generation and testing methods. This section reviews existing automated techniques for finding `rustc` bugs.

**Analysis Method.** Table 8 lists the selected testing tools, their first release time, program generation approaches, supported features, and testing methods. In the Rust community, several individual projects have been developed to perform fuzz testing on `rustc`. Fuzz-rustc [Renshaw 2019] adapts LibFuzzer [LLVM 2023] into a custom script to systematically mutate input byte stream and uncover crashes in `rustc`. Tree-splicer [Barrett 2023] constructs new test cases by recombining

ASTs extracted from existing programs, though it is constrained by the structures present in its seed inputs and often produces syntactically invalid programs. ICEMaker [Krüger 2020], the most widely used fuzzing tool, combines elements of both Fuzz-rustc and Tree-splicer, leveraging iterative mutations and employing tools like Miri [Miri 2023] and Clippy [Rust-clippy 2023] to analyze generated programs. In academia, several tools have been developed to test rustc by generating Rust programs using different methodologies. RustSmith [Sharma et al. 2023] constructs ASTs that conform to Rust’s grammar, ensuring syntactically valid programs, and uses differential testing to detect inconsistencies across rustc versions or optimization levels. Rustlantis [Wang and Jung 2024] generates custom MIRs via the `mir!()` macro, making it effective at detecting bugs in MIR-based optimizations. Rust-twins [Yang et al. 2024] employs differential testing by generating semantically equivalent programs using macros and comparing their HIRs and MIRs, aided by Large Language Models (LLMs) for generation. Typecheck-fuzzer, an early work by Dewey et al. [Dewey et al. 2015], uses Constraint Logic Programming (CLP) to generate well-typed programs and uncover type-checking bugs in Rust’s type system.

Table 8. Information and statistical results of existing tools for detecting rustc bugs. (Validity: ● indicates all of the generated programs are valid, ○ indicates approximately half of the generated programs are valid, and ○ indicates the generated programs are mostly invalid. Support features: ○ means unsupported, ● means fully supported, and ○ means partially supported for specific features.)

Tool	First Release <sup>1</sup>	Program generation approaches			Supported features			Testing Method	#Reported Bugs <sup>2</sup>	#Tested Bugs
		Method	Representation	Validity	Unstable	Flag	API			
Community	Fuzz-rustc	mutation	Byte Stream	○	●	○	●	Fuzzing	49	1
	Tree-splicer	splicing	AST	○	○	○	●	Fuzzing	27	0
	ICEMaker	mutation	AST	○	●	●	●	Fuzzing	873	0
Academia	RustSmith	Rule-based	AST	●	○	●	○	Differential	3	0
	Rustlantis	Rule-based	MIR	●	●	●	○	Differential	8	0
	Rust-twins	LLM-based	Rust code	○	●	●	●	Differential	8	2
	CLP-Fuzzer	Rule-based	Rust code	●	○	○	○	Fuzzing	14	-

<sup>1</sup> For tools proposed in academic papers without open-source availability, we document the publication date of the paper, the actual tool development likely preceded this date.

<sup>2</sup> The bugs detected by these tools do not fully align with our dataset. For community-sourced tools, we use their official bug statistics (up to March 3, 2025), and for paper-proposed tools, we record the data from their publications.

To investigate the performance of these tools, we conduct a *two-step* analysis. In the first step, we identify which rustc bugs in our dataset fall within the tool’s capability scope by matching the user names of the issue submitters. If the submitter is a tool developer, we attribute the bug detection to that tool. If the submitter belongs to the Rust development team, we classify the issue as detected by a Rust team member; otherwise, it is reported by a Rust user. In addition, we examine all these open-source tools and review their corresponding papers to understand their techniques. In the second step, we run each tool for 12 hours to test a specific historical version of rustc (**v1.58.0**), recording the number and types of detected bugs. For tools that require seed programs, we use the official test suite of the rustc being tested. Since the official test suite includes many cases expected to trigger ICE, we exclude these cases and apply the remaining ones as the seed set. After excluding these cases, there are a total of 6,876 test cases. We did not run CLP-Fuzzer because the code link is no longer available and it was tested on an early 1.0-alpha version of rustc, which is very different from modern rustc. We follow the default setup (e.g., verification commands, LLM settings) of each tool in our experiment, and all experiments are conducted in the same environment.

**Analysis Results.** In Table 8, the “Validity” column indicates whether the generated program can be successfully compiled. Community-developed fuzzers often produce invalid programs due to

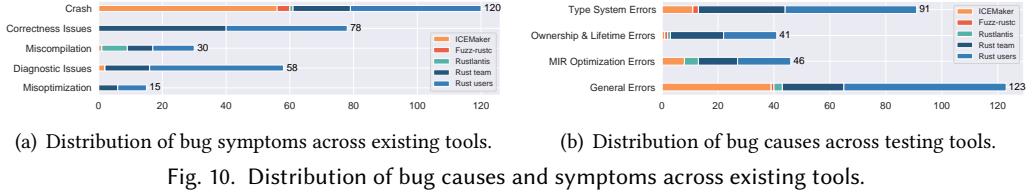


Fig. 10. Distribution of bug causes and symptoms across existing tools.

the randomness of program generation and the coarse-grained nature of their mutation and splicing rules. In contrast, academic research tends to focus more on generating valid programs, which is particularly useful for uncovering deeper `rustc` bugs, such as miscompilations and misoptimizations. The "*Supported features*" column shows which high-frequency features summarized in Section 5 are supported by each tool. Community-developed fuzzers rarely provide explicit support for unstable features and the std/core API, relying instead on seed programs. If present in seeds, these features may be incorporated during mutation. However, due to the lack of semantic awareness, generated test cases may declare an unstable feature or API without actually exercising its functionality, limiting their effectiveness in systematically testing such features. Academic research often explores various compilation flag combinations, which is particularly beneficial for differential testing, yet rarely supports unstable features and APIs explicitly. Among these four tools, only Rust-twins fully supports them, leveraging LLMs for code generation. Rustlantis supports a few unstable features for custom MIR and low-level optimizations, while RustSmith and CLP-Fuzzer overlook them. The column "#Tested Bugs" in Table 8 shows the number of bugs detected after we run each tool for 12 hours. Among them, Fuzz-rustc found 1 bug, and Rust-twins identified 2 bugs, all of which are ICEs. Upon our careful check, the bug found by Fuzz-rustc was previously submitted by ICEMaker<sup>2</sup> and is still in an open state, which seems that it has not been actively maintained or verified since then. The two ICEs discovered by Rust-twins are duplicates, with identical error messages and root causes. They overlap with another issue submitted by ICEMaker in the past<sup>3</sup>, which was closed after Rust developers determined it to be intentional behavior. From the detection results, ICE is the most easily triggered and detected bug symptom. We believe that the effectiveness of testing tools may be influenced by a longer testing time and the quality of seed programs.

Figure 10 presents the detection status of the 301 valid bugs we collected across different tools. Among them, ICEMaker, Fuzz-rustc, and Rustlantis have detected some bugs, while issues submitted by other tools are not included in our collected bug list. This discrepancy may be due to some tools submitting issues beyond our dataset collection timeframe, their reported issues lacking the labels we collected, or their submitted issues remaining open and thus not included. As shown in Figure 10(a), ICEMaker and Fuzz-rustc, as fuzzing tools, are capable of detecting crash bugs, while Rustlantis specializes in identifying miscompilation bugs. Additionally, ICEMaker has also detected a few miscompilation and diagnostic issues, thanks to its more comprehensive verification approach, which leverages a wider range of compilation flags and integrates tools like Miri to detect UB. However, as Figure 10(b) indicates, the majority of the bugs detected by ICEMaker are caused by general errors, rather than Rust-specific issues, highlighting ICEMaker's ability to uncover corner cases within `rustc`. An interesting observation from Figure 10(a) is that existing tools fail to detect correctness issues and misoptimizations. This highlights the limitations of current tools in identifying deep `rustc` bugs, which require a deep understanding of Rust's language rules and extensive experience.

<sup>2</sup><https://github.com/rust-lang/rust/issues/114920><sup>3</sup><https://github.com/rust-lang/rust/issues/123950>

## 7 Implications and Discussion

This section analyzes the conclusions and findings derived from our study, and we further discuss potential future research directions for `rustc`, and threats to the validity.

### 7.1 Findings

- **Finding 1: A large number of `rustc` bugs in the HIR and MIR modules are caused by Rust's unique type system and lifetime model.** As shown in Table 3, most bugs are related to the type system (30.23%), ownership and lifetime management (13.62%), and MIR optimizations (15.28%). Unlike many compilers that lower the AST directly to LLVM IR [Kruse and Finkel 2018], `rustc` introduces HIR and MIR for advanced type and lifetime analysis, as well as compilation optimizations. Figure 4(a) shows that bug distributions in HIR (44.85%) and MIR (35.22%) are similar, underscoring frequent errors tied to Rust's advanced type and memory management mechanisms. These findings highlight the challenges posed by Rust-specific features, such as zero-cost abstractions and strict borrow checking, making these components more error-prone than general programming features.
- **Finding 2: `rustc` bugs share many symptoms with other compiler bugs but also introduce unique types, such as undefined behavior in safe Rust.** Like other compilers, `rustc` experiences various compilation and runtime bugs. However, its ICE bug often causes panic with safety protection, setting it apart from other compilers where ICE typically results in segmentation faults or abnormal terminations. Another unique symptom is undefined behavior in safe Rust code, tied to Rust's safety guarantees. While performance-related bugs are absent in our analysis, this doesn't mean `rustc` is free of performance issues. Rather, these issues tend to appear less frequently in Rust-specific issues or may be categorized as misoptimizations related to code efficiency.
- **Finding 3: Both valid and invalid programs are critical in triggering `rustc` bugs.** As shown in Table 4, certain bug symptoms such as partial ICES, completeness issues, and inconsistencies require valid programs to reveal, while issues related to soundness, diagnostics, and some ICES can only be triggered by invalid programs. This highlights the importance of both valid and invalid programs in exposing different types of `rustc` bugs. This contrasts with the typical approaches in compiler testing, where front-end bugs are often triggered by invalid programs [Chaliasos et al. 2022; Chen et al. 2016; Sun et al. 2018], while optimization bugs are primarily tested with valid programs [Le et al. 2014; Livinskii et al. 2020; Yang et al. 2011]. It's important to note that invalid programs are not entirely outside language rules; they often contain minor errors that trigger bugs in the core components, such as trait solving and optimization.
- **Finding 4: `rustc`'s diagnostic module still has considerable potential for enhancement, with many issues distributed across different IR-processing modules.** As shown in Table 3, diagnostic issues account for about 20% of all bugs. Figure 4(b) illustrates that error reporting is scattered across different components, including HIR (14.07%) and MIR (16.04%), with each component having its own dedicated module for error analysis and reporting. Moreover, gaps in these modules still exist, causing some errors to be inaccurately detected or reported.
- **Finding 5: Trait-related features, unstable features, compilation flags, lifetime annotations, and standard APIs are key contributors to triggering `rustc` bugs.** Trait-related constructs, including traits, `impl` traits, and trait objects, often appear in both item and type node types (Table 7), highlighting their contribution in triggering `rustc` bugs. From Figure 4(b), trait-solving issues account for 27.41% of HIR-related bugs, reflecting the complexity of trait components. Traits, frequently combined with generics and trait-bound constraints, create intricate edge cases that challenge `rustc`'s resolution. Additionally, unstable features, compilation flags, and

lifetime annotations also impact the likelihood of triggering rustc bugs. Applying standard APIs further exposes rustc bugs, underscoring the interplay of multiple factors in bug occurrence.

► **Finding 6: Existing rustc testing tools are less effective at detecting non-crash bugs.** Figure 10(a) shows that about 50% of the crash bugs are detected by existing rustc testing tools. However, other issues, such as miscompilation and diagnostic errors, are largely overlooked. The effectiveness of existing tools may be limited by their test program generation methods, for example, lacking effective guidance metrics or mutation rules.

## 7.2 Actionable Suggestions and Takeaways

After presenting our study results and findings, we now briefly discuss further suggestions for Rust developers, rustc developers, and programming language researchers, respectively.

► **Suggestion 1: (For Rust developers) Be cautious with unstable features and custom optimization settings.** As shown in Table 7, unstable features account for over 20% of triggering rustc bugs, indicating that these features may introduce flaws leading to unexpected behavior. Additionally, custom optimization settings, such as enabling specific MIR passes or adjusting optimization levels, can cause unintended side effects or instability. *(1) Rust developers should first execute programs with the default optimization level to check results before applying higher-level optimizations, ensuring consistency and avoiding potential rustc bugs. (2) Rust developers should avoid using unstable features and employ a stable version of rustc when developing system-level software, which is beneficial for ensuring software reliability.*

► **Suggestion 2: (For Rust developers) The suggestions provided by rustc may be inaccurate.** As shown in Table 4, nearly 20% of rustc bugs are linked to the feedback provided by rustc, including error messages and suggested fixes. This suggests that rustc's diagnostic tools may not always provide accurate or effective solutions. *If rustc's suggestion does not resolve the issue, Rust developers should consider alternative approaches. Reporting the bug to the Rust team can also be beneficial for improving the reliability of rustc.*

► **Suggestion 3: (For rustc developers) Designing testing and verification techniques for rustc components across different IRs.** The core process of rustc involves HIR and MIR lowering, along with type checking, borrow checking, and optimization. Figure 4 indicates that 44.85% and 35.22% of the issues occur in the modules responsible for processing HIR and MIR, respectively. However, existing fuzzers rarely employ specialized testing techniques for these components. Currently, Rustlantis is the only tool capable of generating valid MIR, but it lacks support for other modules, such as type checking and lifetime analysis. *To verify the key rustc components, rustc developers should generate valid HIRs and MIRs under specific constraints. For example, generating HIRs to ensure well-formedness in different scenarios, such as for build-in traits and user-defined traits.*

► **Suggestion 4: (For rustc developers) Investing more effort in implementing and maintaining new language features and compilation settings.** As shown in Table 7, 24.25% of the bug-revealing test cases apply unstable features, and 18.94% employ special compilation commands. This indicates that some less frequently used or newly proposed features still have many flaws, which should receive attention from rustc developers. *For newly proposed unstable features or syntax rules, developers should discuss thoroughly their potential use cases and rustc's expected behaviors in RFC meetings. This helps design diverse test cases, ultimately enhancing rustc's reliability.*

► **Suggestion 5: (For researchers) Building better Rust program generators that fully support Rust's unique type system.** Research on testing, debugging, and analyzing C/C++ compilers

often relies on CSmith [Yang et al. 2011], a random generator that produces valid C programs covering a wide range of syntax features. For Rust, the only preliminary tool, RustSmith [Sharma et al. 2023], generates complex control flow and extensive use of variables and primitive types but has limited support for Rust’s higher-level abstractions. As shown in Table 3, many rustc bugs stem from improper handling of advanced features like traits, opaque types, and references. Additionally, Table 6 indicates that test cases combining these abstractions are more likely to trigger bugs. *Researchers should create a Rust program generator that supports Rust’s advanced features like generics, traits, and lifetime annotations, for example, by enhancing RustSmith.*

► **Suggestion 6: (For researchers) Generating well-designed, both valid and invalid Rust programs to test rustc’s type system.** Our analysis shows that over half of rustc bugs originate from the HIR and MIR modules, particularly in type and WF checking, trait resolution, borrow checking, and MIR transformation. Many corner cases expose weaknesses in rustc’s type handling. (1) *Researchers should develop Rust-specific mutation rules, such as altering lifetimes, to introduce minor errors into valid programs and generate invalid ones for detecting soundness bugs.* (2) *Researchers should synthesize test programs from real-world Rust code, which provides diverse unstable features, std API usage, lifetime annotations, and complex trait patterns that benefit for testing rustc.*

### 7.3 Threats to Validity

One potential threat to validity is that the selected bugs may not be representative because we only collected closed issues with fixing patches and did not consider the issues still open. We consider that unclosed issues represent the possibility that the bug has not yet been recognized by the developer or fixed, which makes it difficult to analyze the root causes and compilation stages. We used several methods to ensure the validity. On the one hand, we crawled all issues that were tagged with a series of labels with Rust-specific features to make sure we didn’t miss the rustc-related bugs. In addition, we excluded non-bug issues, such as questions and discussions during the manual review process. This is in line with previous bug study work [Chaliasos et al. 2021; Di Franco et al. 2017; Jin et al. 2012; Sun et al. 2016], which mainly focuses on closed issues and bug reports for further analysis. On the other hand, we have manually checked all closed issues since the release of Rust Edition 2021, covering the entire three-year period, which allows for more accurate statistical analysis in large-scale bug research.

Another potential threat is the possibility of incorrection in our labeling results. To mitigate this issue, we establish criteria for classifying each label, drawing references from existing compiler bug studies and the official Rust documentation. Additionally, each issue is independently inspected by two co-authors and then cross-checked the results between themselves and the other co-authors to achieve consensus. This aligns with the bug analysis approach from prior empirical studies [Chaliasos et al. 2021; Sun et al. 2016; Xie et al. 2021; Xiong et al. 2023], where each bug was manually reviewed and labeled by multiple researchers.

## 8 Related Work

In this section, we primarily focus on two perspectives of closely related research: (1) the empirical studies of compiler bugs, and (2) the studies of Rust programs.

### 8.1 Understanding Compiler Bugs

The most relevant bug study to our work is conducted by Chaliasos et al. [Chaliasos et al. 2021], which analyzes typing-related bugs in four JVM compilers: Java, Scala, Kotlin, and Groovy. It highlights numerous overlooked type-related bugs in JVM compilers. While some findings align with ours, the design differences between rustc and JVM compilers are significant. Notably, Rust’s

use of associated functions, types, and borrow checking introduces new type-related bugs. Another closely related study by Xia et al. [Xia et al. 2023] provides the first analysis of historical bugs in two Rust compilers, `rustc` and Rust-GCC. However, their analysis relies solely on statistical data, such as lines of code in issues, variable counts, label classifications, and affected modules in pull requests, without delving into the `rustc`'s implementation details. The analysis lacks depth, for example, it fails to elucidate the symptoms and root causes of the errors within `rustc`. In contrast, our work presents the first comprehensive bug analysis specifically for `rustc`, the only official and mature Rust compiler. We manually reviewed and annotated issues and PRs related to Rust features covering a three-year period, categorizing and quantifying their bug causes and symptoms. Our study also examines the susceptibility of different compilation stages to bugs and compares existing testing techniques for `rustc`. By offering deeper insights into `rustc`'s design and prevalent bugs, we aim to inform researchers and guide future improvements in Rust compiler development.

Empirical studies on compiler bugs have been conducted extensively, especially for C/C++ compilers, such as the investigation proposed by Sun et.al. [Sun et al. 2016], which focused on understanding compiler bugs in GCC and LLVM. Subsequently, Zhou et al. [Zhou et al. 2021] conducted further research and analysis on the characteristics of optimization bugs in GCC and LLVM, providing some testing and debugging guidance for testing compilers. Another study [Xie et al. 2021] analyzed LLVM's tool-chain bugs, summarizing typical reasons for their interaction and their corresponding fixing commits. Additionally, an empirical study on WebAssembly compilers [Romano et al. 2022] investigated the bugs' lifecycle, impact, and sizes of bug-inducing inputs and bug fixes. Unlike these works, which all focus on investigating the bug characteristics of the compiler back-end, our work is the first systematic study towards `rustc` as a front-end compiler.

## 8.2 Empirical Studies of Rust Programs and Testing Approaches

Most existing studies focus on the unsafe usages of Rust, such as investigating how programmers employ unsafe Rust [Astrauskas et al. 2020; Cui et al. 2024; van Oorschot 2023; Zhang et al. 2023], the potential risks associated with unsafe code [Höltervennhoff et al. 2023], and whether Rust programs are used safely [Evans et al. 2020]. Zhu et al. [Zhu et al. 2022] analyzed the difficulty of understanding, application, and challenges associated with Rust safety rules. Xu et al. [Xu et al. 2021] conducted an in-depth analysis of Rust CVEs, exploring bugs related to memory safety. Qin et al. [Qin et al. 2020] conducted research on memory and thread safety issues in real Rust programs. Zheng et.al. [Zheng et al. 2023] performed an investigation into the security risks in the Rust ecosystem, discussing the characteristics of the vulnerabilities in Rust programs. Different from existing empirical studies of Rust programs, we propose the first systematic bug study of the Rust compiler, which is resilient to memory safety issues and has unique intermediate representations designed for type checking and borrow checking. We not only present the symptoms and root causes of `rustc` bugs but also analyze the syntactic features that are prone to triggering `rustc` bugs, along with the modules within `rustc` that are more prone to bugs.

With Rust's powerful type system and memory management model, some research has been conducted on the testing and verification of Rust programs. For instance, SyRust [Takashima et al. 2021] automatically generates Rust programs to effectively test Rust libraries. Verus [Lattuada et al. 2023] is an SMT-based verifier for Rust programs, while Aeneas [Ho and Protzenko 2022] translates lightweight functions for verification. RustHornBelt [Matsushita et al. 2022] employs a semantic model to check Rust's soundness. Additionally, some approaches [Astrauskas et al. 2019; Wolff et al. 2021] leverage Rust's type system for verification. Unlike these works focusing on testing and verification, our study examines the Rust compilation process, particularly the reliability of its type-checking and borrow-checking implementations. We believe our findings can benefit

compiler developers, Rust programmers, and programming language researchers while opening new directions for Rust research.

## 9 Conclusion

This paper conducts a comprehensive empirical study specifically dedicated to bugs in the Rust compiler, categorizing and analyzing bug symptoms, causes, bug-prone compilation stages, and test case characteristics. Many insights, suggestions, and potential research directions for testing and debugging `rustc` are provided in our study. We found that the number of `rustc` bugs caused by HIR and MIR are comparable, with errors resulting from Rust-specific analyses and checks, as well as MIR-based optimizations, accounting for the majority. Additionally, existing testing generation techniques for `rustc`'s testing tools are insufficient, and there is a lack of automated testing technologies for both correctness and misoptimization bugs in `rustc`. We expect our research to deepen the understanding of bugs in `rustc` and provide guidance for `rustc`'s testing and development, as well as research on Rust's compilation and optimization.

## Data-Availability Statement

All source code and data for this study can be found at [https://anonymous.4open.science/r/rustc\\_bug\\_study-4106](https://anonymous.4open.science/r/rustc_bug_study-4106).

## References

- Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.
- Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (oct 2019), 30 pages. <https://doi.org/10.1145/3360573>
- Langston Barrett. 2023. langston-barrett/tree-splicer: Simple grammar-based test case generator. <https://github.com/langston-barrett/tree-splicer>. (Accessed on 12/05/2023).
- Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-typed programs can go wrong: A study of typing-related bugs in jvm compilers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.
- Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 183–198.
- Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) (PLDI '16). Association for Computing Machinery, New York, NY, USA, 85–99. <https://doi.org/10.1145/2908080.2908095>
- Cloudflare. 2023. Cloudflare - The Web Performance & Security Company. <https://www.cloudflare.com/>. (Accessed on 12/06/2023).
- Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 106, 13 pages. <https://doi.org/10.1145/3597503.3639136>
- Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 482–493. <https://doi.org/10.1109/ASE.2015.65>
- Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 509–519. <https://doi.org/10.1109/ASE.2017.8115662>
- Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. 2024. When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 359 (Oct. 2024), 31 pages. <https://doi.org/10.1145/3689799>
- Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust used safely by software developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 246–257.

- Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (aug 2022), 31 pages. <https://doi.org/10.1145/3547647>
- Sandra Höltervennhoff, Philip Klostermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. 2023. {“I} wouldn’t want my unsafe code to run my {pacemaker}”: An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2509–2525.
- InfoWorld. 2023. 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).
- Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *SIGPLAN Not.* 47, 6 (jun 2012), 77–88. <https://doi.org/10.1145/2345156.2254075>
- Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152.
- Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.
- Michael Kruse and Hal Finkel. 2018. User-Directed Loop-Transformations in Clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 49–58. <https://doi.org/10.1109/LLVM-HPC.2018.8639402>
- Matthias Krüger. 2020. matthiaskrgr/icemake: automatically find crashes in the rust compiler & tooling. <https://github.com/mattiaskrgr/icemake>. (Accessed on 12/05/2023).
- Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 75–86.
- Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (apr 2023), 30 pages. <https://doi.org/10.1145/3586037>
- Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.
- Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. <https://doi.org/10.1145/3428264>
- LLVM. 2023. libFuzzer – a library for coverage-guided fuzz testing. – LLVM 18.0.0git documentation. <https://llvm.org/docs/LibFuzzer.html>. (Accessed on 12/09/2023).
- Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 841–856. <https://doi.org/10.1145/3519939.3523704>
- Miri. 2023. rust-lang/miri: An interpreter for Rust’s mid-level intermediate representation. <https://github.com/rust-lang/miri>. (Accessed on 12/14/2023).
- Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiyi Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 763–779.
- RedoxOS. 2023. Redox - Your Next(�) OS - Redox - Your Next(�) OS. <https://www.redox-os.org/>. (Accessed on 12/06/2023).
- David Renshaw. 2019. dwrensha/fuzz-rustc: setup for fuzzing the Rust compiler. <https://github.com/dwrensha/fuzz-rustc>. (Accessed on 12/05/2023).
- Alan Romano, Xinyue Liu, Yonghui Kwon, and Weihang Wang. 2022. An empirical study of bugs in webassembly compilers. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) (ASE ’21). IEEE Press, 42–54. <https://doi.org/10.1109/ASE51524.2021.9678776>
- Rust. 2023a. rust-lang/rust: Empowering everyone to build reliable and efficient software. <https://github.com/rust-lang/rust>. (Accessed on 11/29/2023).
- Rust. 2023b. What is rustc? - The rustc book. <https://doc.rust-lang.org/rustc/what-is-rustc.html>. (Accessed on 12/06/2023).
- Rust. 2025a. Next-gen trait solving - Rust Compiler Development Guide. <https://rustc-dev-guide.rust-lang.org/solve/trait-solving.html> [Online; accessed 2025-03-03].
- Rust. 2025b. Well-formedness checking. <https://rust-lang.github.io/chalk/book/clauses/wf.html> [Online; accessed 2025-03-03].
- Rust-clippy. 2023. rust-lang/rust-clippy: A bunch of lints to catch common mistakes and improve your Rust code. Book: <https://doc.rust-lang.org/clippy/>. <https://github.com/rust-lang/rust-clippy>. (Accessed on 12/14/2023).
- Rust-GCC. 2024. Rust-GCC/gccrs: GCC Front-End for Rust. <https://github.com/Rust-GCC/gccrs>. (Accessed on 09/07/2024).
- rust team. 2025. Labels rust-lang/rust. <https://github.com/rust-lang/rust/labels> [Online; accessed 2025-01-19].
- rustc-dev guide. 2025. Opaque Types - Rust Compiler Development Guide. <https://rustc-dev-guide.rust-lang.org/opaque-types-type-alias-impl-trait.html> [Online; accessed 2025-03-06].

- Servo. 2023. Servo, the embeddable, independent, memory-safe, modular, parallel web rendering engine. <https://servo.org/>. (Accessed on 12/06/2023).
- 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 (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. <https://doi.org/10.1145/3597926.3604919>
- STRATIS. 2023. Stratis Storage. <https://stratis-storage.github.io/>. (Accessed on 12/06/2023).
- Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th international symposium on software testing and analysis*. 294–305.
- Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering (Gothenburg, Sweden) (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 361–371. <https://doi.org/10.1145/3180155.3180236>
- 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>
- TiKV. 2023. TiKV is a highly scalable, low latency, and easy to use key-value database. <https://tikv.org/>. (Accessed on 12/06/2023).
- David Tolnay. 2025. syn - crates.io: Rust Package Registry. <https://crates.io/crates/syn> [Online; accessed 2025-03-03].
- Paul C. van Oorschot. 2023. Memory Errors and Memory Safety: A Look at Java and Rust. *IEEE Security & Privacy* 21, 3 (2023), 62–68. <https://doi.org/10.1109/MSEC.2023.3249719>
- Qian Wang and Ralf Jung. 2024. Rustlantis: Randomized Differential Testing of the Rust Compiler. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 340 (Oct. 2024), 27 pages. <https://doi.org/10.1145/3689780>
- Wikimedia. 2004. Undefined behavior - Wikipedia. [https://en.wikipedia.org/wiki/Undefined\\_behavior](https://en.wikipedia.org/wiki/Undefined_behavior) [Online; accessed 2025-03-23].
- Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 145 (oct 2021), 29 pages. <https://doi.org/10.1145/3485522>
- Xinmeng Xia, Yang Feng, and Qingkai Shi. 2023. Understanding Bugs in Rust Compilers. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. 138–149. <https://doi.org/10.1109/QRS60937.2023.00023>
- Xiaoyuan Xie, Haolin Yang, Qiang He, and Lin Chen. 2021. Towards Understanding Tool-chain Bugs in the LLVM Compiler Infrastructure. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–11.
- Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (Seattle, WA, USA) (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1319–1331. <https://doi.org/10.1145/3597926.3598138>
- Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–25.
- Wenzhang Yang, Cuifeng Gao, Xiaoyuan Liu, Yuekang Li, and Yinxing Xue. 2024. Rust-twins: Automatic Rust Compiler Testing through Program Mutation and Dual Macros 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, 631–642. <https://doi.org/10.1145/3691620.3695059>
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.
- Yuchen Zhang, Ashish Kundu, Georgios Portokalidis, and Jun Xu. 2023. On the Dual Nature of Necessity in Use of Rust Unsafe Code (*ESEC/FSE 2023*). Association for Computing Machinery, New York, NY, USA, 2032–2037. <https://doi.org/10.1145/3611643.3613878>
- 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>
- Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884.
- Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 2022. Learning and programming challenges of rust: A mixed-methods study. In *Proceedings of the 44th International Conference on Software Engineering*. 1269–1281.

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