An Empirical Study of Bugs in the rustc Compiler

ZIXI LIU, Nanjing University, China
YANG FENG*, Nanjing University, China
YUNBO NI, Nanjing University, China
SHAOHUA LI, Chinese University of Hong Kong, China
XIZHE YIN, Nanjing University, China
QINGKAI SHI, Nanjing University, China
BAOWEN XU, 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, resulting in bugs that are uncommon 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. Such understanding could provide the foundation for developing more effective testing strategies tailored to `rustc`. Improving the quality of `rustc` testing is essential for enhancing compiler reliability, which in turn strengthens the safety and correctness of all Rust programs, as compiler bugs can silently propagate into every compiled program. Yet, we still lack a large-scale, detailed, and in-depth study of `rustc` bugs.

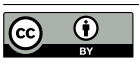
To bridge this gap, this work presents a comprehensive and systematic study of `rustc` bugs, specifically those originating in semantic analysis and intermediate representation (IR) processing, which are stages that implement essential Rust language features such as ownership and lifetimes. Our analysis 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.

CCS Concepts: • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: Rust, compiler, testing, empirical study, bug study

*Yang Feng is the corresponding author.

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



ACM 2475-1421/2025/10-ART411
https://doi.org/10.1145/3763800

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 Bugs in the rustc Compiler. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 411 (October 2025), 33 pages. https://doi.org/10.1145/3763800

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].

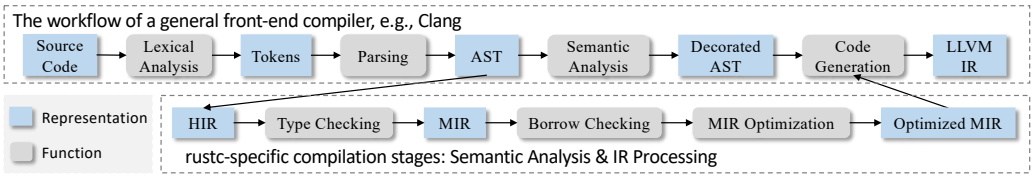


Fig. 1. The high-level workflow of `rustc` and a general front-end compiler.

The primary compiler for Rust is the official open-source `rustc` [Rust 2023c], which is written in Rust and uses LLVM [Lattner and Adve 2004] as its default backend. Like traditional compilers, `rustc` follows the general compilation workflow comprising lexical analysis, parsing, semantic analysis, and code generation. However, to support Rust's unique language features (such as ownership, lifetimes, traits and so on) and memory-safety guarantee, `rustc` introduces additional intermediate representations (IRs) and specialized compilation mechanism that distinguish it from conventional compilers like Clang and GCC. 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` performs several optimizations over the MIR to ensure both runtime efficiency and the memory safety. This multi-layered design makes semantic analysis and IR processing in `rustc` both unique and central to its compilation pipeline.

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[1]. Despite their importance, existing tools and studies have overlooked the challenges introduced by compilation mechanisms that enforce Rust's core language features, leaving a gap in understanding their impact on testing `rustc`. This gap is especially concerning because bugs in `rustc` can propagate silently into every compiled Rust application, posing significant risks to reliability and security. To date, the only empirical study on Rust compilers was by Xia et al. [Xia et al. 2023], which provides comprehensive statistics but lacks in-depth 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 such as miscompilations. In the research domain, RustSmith [Sharma et al. 2023] was 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 rely on macro-based strategies, but they can only uncover a small subset of real `rustc` bugs. We doubt that the key limitation of these tools lies in a lack of deep understanding of the unique bug characteristics in `rustc`. For example, for C/C++ compilers, CSmith is a well-known tool that uncovered hundreds of C compiler bugs despite supporting only basic language features [Yang et al. 2011]. By contrast, RustSmith [Sharma et al. 2023] adopts a similar strategy but has proven far less effective for Rust, as it uncovered only a few historical bugs and none in recent versions of `rustc`. 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 bugs related to the implementation of memory-safety guarantee mechanisms, we conduct a quantitative and qualitative study on the official Rust compiler, `rustc`. *This study focuses on bugs originating in the two critical compilation stages, i.e., semantic analysis and IR processing,* which implements essential language features such as ownership and lifetimes in Rust [2]. These include phases such as IR lowering and optimization, as illustrated in Figure 1. 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 features related Rust language mechanisms. In particular, our study answers the following research questions.

- **RQ1 (Bug Causes): What are the main causes of `rustc` bugs?** 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 `rustc` bugs?** 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?

These research questions are designed to provide a comprehensive understanding of `rustc` bugs from multiple perspectives. We begin with RQ1 and RQ2, which together characterize `rustc`

[1] https://github.com/rust-lang/rust/issues/132186

[2] In this paper, "`rustc` bugs" refers specifically to errors arising from the implementation of semantic analysis and IR processing in the Rust compiler.

bugs by examining their underlying causes and observable symptoms. Understanding both where bugs originate and how they manifest offers a complete picture of the challenges faced by `rustc`. To answer these questions, we collect a list of issues and their corresponding pull requests from Rust's official GitHub [Rust 2023b] over the past three years. Each bug is manually labeled with its symptoms, cause, and the compilation stage where it occurs. Building on this foundation, RQ3 investigates how these bugs are triggered in practice. Specifically, we examine the characteristics of bug-revealing test cases, including the language features, input patterns, and compilation configurations that tend to trigger `rustc` bugs. To this end, we extract test cases and compilation commands from the collected issues and parse their abstract syntax trees (ASTs) to analyze the involved language features. Finally, RQ4 evaluates the effectiveness of existing testing tools for `rustc` by collecting the number and types of bugs these tools have detected. Furthermore, we apply these tools to a specific version of `rustc` to assess their bug-finding capabilities in a controlled setting. Based on these results, we examine the strengths and limitations of current tools and discuss the broader challenges and future directions for testing `rustc`.

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

- We manually construct a three-year dataset of `rustc` bugs, including test cases, issues, and fixes, providing a foundation for this study and future research on testing and verification.
- We conduct a comprehensive empirical study of `rustc` bugs from multiple perspectives, 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:

(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 (51.1%) stem from type resolution and well-formedness checks, while MIR-related bugs mainly relate to MIR transformation (50.0%).
(2) Crash is the most common symptom (39.9%), followed by correctness issues (25.9%), 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 (24.3%) and specific compilation settings or optimization levels (18.9%). 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 6.1% 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 is summarized in Figure 2. Firstly, we perform bug data collection (Section 2.1) by collecting all closed issues from the official Rust GitHub repository within a specified time frame (2022-01-01 to 2025-01-01). We apply an initial filter using official issue labels, focusing on those related to Rust safety guarantee mechanisms. Then, we manually filter irrelevant or ineligible issues, such as duplicates or those without test cases. For each remaining issue, we identify the corresponding pull request (PR) and extract the test case provided in the issue

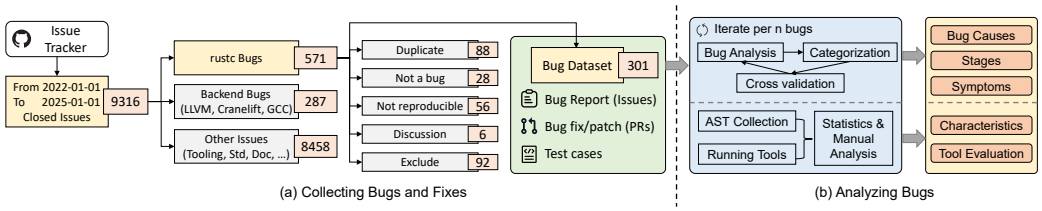


Fig. 2. The overview of our bug collection and analysis process.

description. The final result is a set of `rustc` bugs, each comprising an issue, a PR, and a test case. This dataset serves as the input for our bug analysis process (Section 2.2).

We iteratively analyze the dataset, where each bug is independently labeled by two researchers across multiple dimensions. In cases of disagreement, all researchers engage in discussion until a consensus is reached.

Table 1. Bug labels and corresponding descriptions in Rust's official GitHub issue tracker [rust team 2025b].

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) - https://blog.rust-lang.org/2016/04/19/MIR.html
	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-traits	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	45	The borrow checker
Backend (Excluded)	A-LLVM	275	Code generation parts specific to LLVM.
	A-gcc	2	Things relevant to the [future] GCC backend
	A-cranelift	10	Things relevant to the [future] cranelift backend

2.1 Collecting Bugs and Fixes

To capture the evolution of `rustc` bugs in recent development, we collect all issues reported between January 1, 2022, and January 1, 2025. This period aligns with the usage of Rust 2021 Edition which was released on October 21, 2021, while Rust 2024 was released on February 20, 2025. To ensure all collected bugs are reviewed and patched, we only include closed issues. This yields a total of 9,316 closed issues, forming the complete set for our study.

The official Rust GitHub repository hosts not only the source code of `rustc` but also a wide range of related components, including the standard library, documentation tools (e.g., `rustdoc`), build systems, package managers (e.g., `Cargo`), and backend integrations for code generation (e.g., LLVM). The `rustc` development team maintains a comprehensive labeling system and they employ labels prefixed with "*A-*" to denote individual compiler area, component, or language feature. Thus, to identify `rustc` bugs accurately, we follow this convention and choose the labels that are related to the implementation of Rust language features, including HIR, MIR, Type, and Lifetimes, resulting in 571 issues.

Table 1 lists the selected labels, along with their descriptions and issue counts. Since a single issue may be assigned multiple labels, the total label count exceeds the number of issues. For completeness, the table also includes backend-related labels, e.g., for LLVM, Cranelift, and GCC [rust team 2025a], which correspond to 287 issues and are excluded from our following analysis. Notably, there are 7 issues containing both our selected labels and backend-related labels; considering their possible relation to Rust compilation process, we include them in our study. In addition, 8, 458 issues are not associated with any of the labels listed in Table 1. These issues typically involve bugs or discussions related to the Rust standard library and toolchain and are excluded from our study. Note that while the tracker also offers general-purpose labels such as *C-bug* and *T-compiler*, these are unsuitable for our analysis. The *C-bug* label includes all types of errors, including general compiler bugs, such as those in parsers, while *T-compiler* merely refers to the responsible team and includes many non-bug or organizational issues. Moreover, issues are not consistently tagged with either label, making them unreliable for precise filtering.

For the 571 `rustc` bugs, we manually review and filter out unsuitable ones based on the classification criteria outlined in Table 2. Specifically, 88 issues are labeled as *duplicate*, indicating that they have already been reported and confirmed, typically marked by developers as "closed as a duplicate." 28 issues are classified as *not a bug*, meaning they describe expected behaviors rather than actual defects. There are 56 issues marked as *not reproducible* because they can no longer be reproduced, suggesting the underlying problems may have already been fixed. These are excluded due to the absence of a verifiable fix. There are 6 issues that fall under the *discussion* category, which includes inquiries, suggestions, or vague reports lacking concrete symptoms or test cases. In addition, 92 issues are marked as *exclude*, either because they are unrelated to `rustc` (e.g., documentation bugs) or cannot be reproduced in the 2021 edition (e.g., bugs specific to the 2015 edition). After filtering, the remaining 301 issues are confirmed as valid `rustc` bugs, forming the core dataset for our subsequent empirical analysis.

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 `rustc` improvement, but not a bug.	6
Exclude	(1) Does not contain a test case; (2) Unrelated to `rustc`; (3) Not reproducible on 2021 edition.	92
Valid	The bug has been confirmed as a `rustc` bug, with a corresponding test case and fix.	301
Total	-	**571**

2.2 Analyzing Bugs

This section presents our methodology for analyzing `rustc` bugs to answer RQ1 and RQ2. Because the analysis for RQ3 and RQ4 involves additional data and tools beyond the collected issues and PRs, we present their detailed methodologies separately in Section 5 and Section 6.

Following prior bug analysis approaches [Chaliasos et al. 2021; Drosos et al. 2024; Xiong et al. 2023], we systematically study the collected issues and PRs with reference to the principles of Qualitative Content Analysis (QCA) [Schreier 2012]. To answer RQ1 and RQ2, we employ a mix of theory-based and data-driven approaches to build the coding frame across the following three dimensions: (1) the cause of the bug, (2) the compilation stage during which `rustc` encounters the bug, and (3) the bug symptom. Specifically, we first define an initial set of main categories for each dimension using a theory-based approach grounded in prior studies of compiler bugs [Chaliasos et al. 2021; Chen et al. 2020; Romano et al. 2021; Sun et al. 2016; Tang et al. 2020] and our domain expertise in `rustc`. Then, we apply a data-driven iterative refinement process, guided by observations during labeling bug reports, to evaluate and modify the categories. In addition to the main categories, we also define a set of subcategories to support more fine-grained analysis, which are detailed in Sections 3 and 4. Specifically, for bug causes, we categorize them based on Rust language mechanisms implemented in `rustc`, and classify them into type system errors, lifetime-related errors, MIR optimization errors, and more general logic or implementation mistakes. To determine the bug cause, we review the associated test cases, fix patches, and developer discussions to infer the underlying reason for each bug. For compilation stages, we follow the official Rust compiler development guide [Rust 2023a] and identify three core stages: the generation of AST, HIR, and MIR. Additionally, we treat utility components and code generation as separate modules. We then label the compilation stage by reviewing the files and modules modified in the corresponding PR. For bug symptoms, we refer to prior studies on compiler bugs and define several major categories, including crashes, miscompilations, diagnostic issues, misoptimizations, and performance problems. We determine the symptom by analyzing the descriptions in each bug report to identify discrepancies between the expected and actual behavior of `rustc`.

To ensure rigorous and consistent labeling across all three aspects, we evaluate and modify our coding frame through iterative refinement. We conduct 5 rounds of iterative annotation, each involving 20 randomly selected bug reports. In each round, we employ double-coding, with the first two authors independently labeling the bugs according to the current set of categories. After each round, they compare their results, discuss discrepancies, and refine the definitions of ambiguous or insufficient categories. During these initial rounds, we evaluate the inter-rater reliability using Cohen's Kappa coefficients [Seaman 1999], obtaining values of 0.667 for bug causes, 0.651 for compilation stages, and 0.683 for symptoms, which guide the refinement of the categories. After five rounds covering 100 bug reports, both the main categories and subcategories became stable. A comprehensive description of both the initial and finalized coding frames, along with their definitions and hierarchical structure, is presented in the appendix (see Appendix A). Once finalized, the two authors annotate all the issues and compare their annotations. If a discrepancy occurs, the third co-author independently annotates the same bug report. If the third annotation matches one of the initial results, that label is adopted as the final category; otherwise, all three annotators share their perspectives and discuss until reaching consensus. For the results on all data, the Cohen's Kappa coefficients are 0.913 for bug causes, 0.952 for compilation stages, and 0.946 for symptoms, indicating strong agreement between the two annotators. The full manual annotation effort requires substantial domain expertise in both the Rust language and the `rustc` implementation, and takes approximately six person-months to complete.

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	12.3%
	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	12.6%
	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	2.3%
	Well-formedness	The errors were caused by `rustc`'s well-formedness checking, including ownership, lifetime, type system, and the borrow checker.	9	3.0%
	Subtotal	-	**91**	**30.2%**
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	2.3%
	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	11.3%
	Subtotal	-	**41**	**13.6%**
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	11.3%
	Missing cases	Some specific corner cases of the optimization algorithm were not considered thoroughly.	12	4.0%
	Subtotal	-	**46**	**15.3%**
General Errors	Basic structure	Bugs caused by `rustc` errors in processing features like closures and internal data structures.	38	12.6%
	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	24.9%
	Compatibility	The bugs were triggered by certain operating systems, bugs in the back-end LLVM, or errors specific to the Rust edition.	10	3.3%
	Subtotal	-	**123**	**40.9%**

3 RQ1: Bug Causes

In our collected bug dataset, each issue is linked to a corresponding fix PR. We analyze PR descriptions and code changes to classify bug causes, as summarized in Table 3. We first introduce four major categories of bug causes that stem from the implementation of Rust's core language mechanisms or other specific reasons within the compiler. 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*. In Section 3.5, we present another perspective by investigating the compilation stages in which these bugs are triggered within the `rustc` pipeline. Note that the compilation stage where a bug manifests is related to but distinct from its bug cause. For example, a bug caused by an error in the type system may actually arise during type inference or trait solving within the HIR stage. We provide a detailed discussion of these relationships in Section 3.5.

3.1 Type System Errors

Type system errors are a major cause of bugs in `rustc`, accounting for 30.2% 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*.

Trait & Bound Related Errors: Trait-related errors account for 12.3% 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 12.6% 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 2.3% 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 2025c] 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 3.0% of all causes, arise from `rustc` improperly processing WF checking, leading to incorrect behaviors or panics when validating the well-formedness.

Example. The patch in Figure 3 (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 3 (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 & Lifetime Errors: lifetime errors.

Fig. 3. 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 ensures memory safety without a garbage collector. Bugs caused by ownership and lifetime errors make up 13.6% of all `rustc` bugs. Specifically, `rustc` verifies reference validity over their lifetimes, prevents conflicts between mutable and immutable references, and enforces ownership rules to avoid use-after-move or use-after-drop errors. 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 2.3% 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 11.3% 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 3 (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 3 (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 15.3% 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 (4.0%). From our study, most MIR optimization bugs require modifications to the algorithm's logic, rather than merely fixing a minor overlooked case.

Example. Figure 4 (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 & structure.

Fig. 4. The fix patch for explaining MIR optimization bug cause PR 110569) and general errors PR 112266).

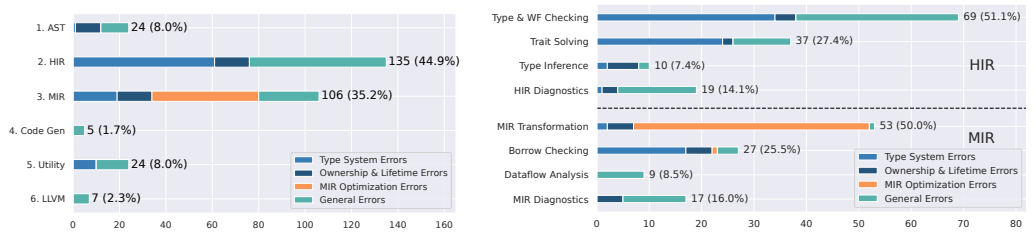
3.4 General Errors

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 40.9% 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 closures or internal data structures (12.6%); **(2)** ***error handling and reporting issues***, where exceptional cases or error reports are mishandled, leading to misleading messages or incorrect error locations (24.9%); and **(3)** ***compatibility issues***, where bugs arise from specific operating system configurations, backend LLVM problems, or Rust edition-specific errors (3.3%).

Example. Figure 4 (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



(a) Bug distribution across `rustc` compilation stages.

(b) Bug distribution across HIR and MIR components.

Fig. 5. Comprehensive analysis of bug distribution in `rustc` pipeline and its HIR/MIR components.

the bug cause to the stage where the error originates. Figure 5(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 5(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 5(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 *diagnostics* in both the HIR and MIR components, most bugs stem from general programming errors, especially improper error handling, which can misclassify the bug 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 39.9% of cases, followed by correctness issues (25.9%) and diagnostic issues (19.3%). Miscompilation and misoptimization are less common, making up 10.0% and 5.0%, respectively.

4.1 Crash

Similar to all software systems, `rustc` also suffers from crashes. Among the bugs we collected, 36.5% involve crash errors. Based on the compilation stage where the crash occurs, we categorize them into front-end panics and back-end crashes.

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

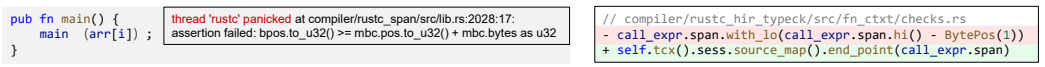
Symptoms		Occurrence		Type System	Ownership & Lifetime	MIR Optimization	General Errors
Crash	Front-end Panic (valid)	42 (14.0%)	120 (39.9%)	30 (25.0%)	3 (2.5%)	19 (15.8%)	**68 (56.7%)**
	Front-end Panic (invalid)	75 (24.9%)					
	Back-end Crash	3 (1.0%)					
Correctness Issues	Completeness Issues	56 (18.6%)	78 (25.9%)	**43 (55.1%)**	17 (21.8%)	7 (9.0%)	11 (14.1%)
	Soundness Issues	22 (7.3%)					
Miscompilation	Inconsistent Output Issues	18 (6.0%)	30 (10.0%)	5 (16.7%)	4 (13.3%)	**12 (40.0%)**	9 (30.0%)
	Safe Rust Causes UB	12 (4.0%)					
Diagnostic Issues	Incorrect Warning/Error	20 (6.6%)	58 (19.3%)	12 (20.7%)	16 (27.6%)	1 (1.7%)	**29 (50.0%)**
	Improper Fix Suggestion	38 (12.6%)					
Misoptimization	Incorrect Optimization	9 (3.0%)	15 (5.0%)	1 (6.7%)	1 (6.7%)	**7 (46.7%)**	6 (40.0%)
	Performance Issues	6 (2.0%)					

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 38.9% of all observed symptoms. Among them, 14.0% are triggered by valid programs, and 24.9% 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.

A small number of back-end crashes exist in our dataset, accounting for 1.0% of all cases. Although we excluded 287 issues labeled as backend-related, as mentioned in Section 2.1, 7 issues were tagged with both backend-related labels and labels of interest to our study, and were therefore retained. Manual inspection confirms that some of these issues lead to back-end crashes. It is worth noting that such crashes are not necessarily specific to the current LLVM-based backend. Similar failures may still occur with alternative backends, such as the in-development Cranelift backend, since these bugs may stem from the backend implementation itself.

Bug cause analysis. The primary causes of crash bugs in `rustc` are general errors (56.7%), such as inadequate error handling and compatibility issues. When `rustc` encounters an unexpected program state, its error recovery mechanisms may be incomplete, leading to `rustc` front-end panic instead of graceful handling. The second major category involves the type system (25.0%) and MIR optimization (15.8%). 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 panic. 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.



(a) A Rust program that triggers an ICE.

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

Fig. 6. The example of a crash bug (Issue 128717) and corresponding fix patch (PR 128864).

Example. Figure 6 (a) shows a code snippet that triggers a front-end panic, 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 6 (b) presents the fix patch, which corrects the positioning approach by eliminating the use of `BytePos`.

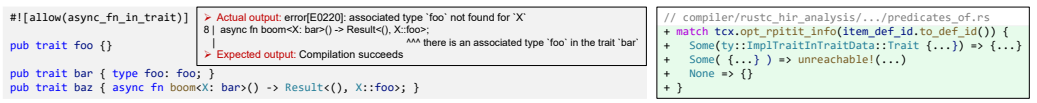
4.2 Correctness Issues

Correctness bugs occur when `rustc` fails to enforce Rust's syntax or semantic rules, leading to the unintended rejection or acceptance of programs, thereby undermining its ability to accurately validate Rust code. We classify these issues into two distinct subcategories: incorrect rejections of valid programs (completeness issues) and incorrect acceptances of invalid programs (soundness issues). We make this distinction because soundness issues are generally considered more severe, as they may allow invalid or unsafe programs to compile and potentially execute incorrectly or unsafely. The correctness issues account for 25.9% of all cases. Among them, 18.6% are triggered by completeness issues, and 7.3% 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 (55.1%) and ownership management (21.8%). 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 (9.0%) and general errors (14.1%), 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.



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

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

Fig. 7. The example of a correctness issue (Issue 132372) and corresponding fix patch (PR 132373).

Example. Figure 7 (a) presents a test case that exposes a *correctness issue*, which is caused by handling *opaque types* and Return-Position Impl Trait in Trait (`RPITITs`), 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 bug cause 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 7 (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.

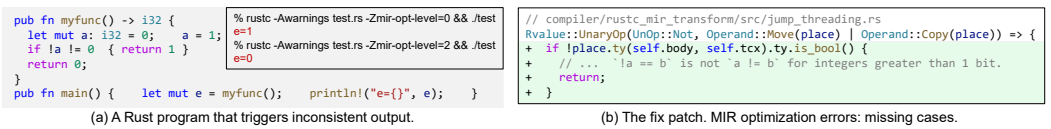
4.3 Miscompilation

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 10.0% 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 6.0% of all symptoms.

Safe Rust Program Causes Undefined Behaviors: This symptom is particularly unique for `rustc` due to the language's strict division between *safe* and *unsafe* code. A core design principle of Rust is that code written entirely in the safe subset should never cause undefined behaviors (UB) [Wikimedia 2004]. This guarantee is strictly upheld by the Rust compiler [Rust 2025b]. When UB occurs in a safe Rust program, it indicates a critical violation of this principle and is therefore considered a compiler bug rather than a user error. This differs from many other languages, where UB is typically attributed to improper use of low-level or unsafe operations by the programmer. In Rust, by contrast, `rustc` is solely responsible for ensuring memory safety in safe code. As such, when `rustc` compiles a safe Rust program that leads to UB, it represents a distinct category of `rustc` bug that violates Rust's core safety guarantees and compromises the trust developers place in the compiler. In our study, such bugs account for 4.0% of all observed symptoms.

Bug cause analysis. Miscompilation bugs in `rustc` are primarily caused by MIR optimization errors (40.0%). Faulty optimization logic can lead to semantic differences between optimized and unoptimized code, directly affecting program correctness. General errors account for 30.0%, 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 (16.7%) and ownership-related errors (13.3%), 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.



(a) A Rust program that triggers inconsistent output.

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

Fig. 8. The example of a miscompilation bug (Issue 131195) and corresponding fix patch (PR 131201).

Example. Figure 8 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 8 (b) resolves this by introducing a boundary check, ensuring that only boolean operands are considered for jump threading.

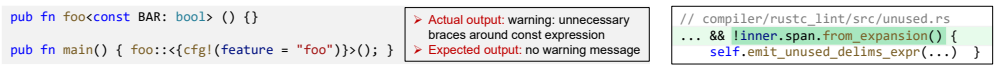
4.4 Diagnostic Issues

`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 19.3%. Among them, incorrect warning/error issues account for 6.6%, and improper fixing suggestion issues account for 12.6%.

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 (50.0%), including shortcomings in error handling and suggestion-matching mechanisms. Similar to `rustc` front-end panics, 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 (20.7%) and ownership-related issues (27.6%) also contribute significantly, as imprecise error branch selection within these checkers can result in misleading or unclear messages. Finally, MIR optimization (1.7%) 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.



(a) A Rust program that triggers a diagnostic issue. (b) The fix patch. General errors: Error handling& Reporting.

Fig. 9. The example of a crash bug (Issue 104141) and corresponding fix patch (PR 105515).

Example. Figure 9 (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 9 (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 5.0% of all.

Incorrect Optimization: `rustc` may apply unexpected optimization strategies, resulting in MIR that deviates from intended semantics or fails to incorporate expected transformations. These issues reflect flaws in the optimization logic but do not necessarily cause incorrect execution.

Performance Issues: In some cases, missing or ineffective optimizations lead to runtime performance or prolonged compilation time. Such issues are relatively rare and typically arise from the absence of expected optimizations rather than from functional errors in the compiled code.

Bug cause analysis. Misoptimization bugs in `rustc` are primarily caused by MIR optimization errors (46.7%) and general errors (40.0%). 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 (6.7%) and ownership-related issues (6.7%) 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 10 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 10 (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 10 (b) fixes this by introducing stricter validation for pointer-to-integer casts, ensuring they are preserved when necessary.

```rust
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.

```
// 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 _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: 2:6
    _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.

Fig. 10. The example of a misoptimization bug (Issue 97421) and corresponding fix patch (PR 97597).

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.

Analysis Method. 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.

Analysis Results. 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. Analyzing test case sizes provides valuable insight into the complexity required to trigger `rustc` bugs. Consistent with prior studies on compilers such as GCC, LLVM [Sun et al. 2016], JVM [Chaliasos et al. 2021], and WebAssembly [Romano et al. 2021], our findings confirm that compiler bugs are often revealed by relatively short programs—typically under 100 lines. This suggests that even small code fragments containing key language features are sufficient to expose critical issues in `rustc`, highlighting the importance of targeted, fine-grained testing.

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

	mean	median	min	max
Original tests	17.8	12	2	346
Reduced tests	14.2	11	2	123

Table 6 presents the distribution of *Item* and *Type* nodes across test cases. 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, representing the fully qualified name of types, e.g., `Vec<i32>`, `std::fs::File`. *Reference* is the second most common type, appearing in 43.9% 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* (20.7%) and *Trait Object* (10.7%) 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 (17.8%) allows constant expressions in generic parameters, enabling more flexible compile-time computations. The `type_alias_impl_trait` feature (15.1%) simplifies complex trait bounds by allowing type aliases with `impl Trait`, making generic code more concise. Meanwhile, the `const_trait_impl`

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

Item	Total[1]	Prevalence	Mean per File[2]	Max per File[2]	Type	Total[1]	Prevalence	Mean per File[2]	Max per File[2]
Function	524	100.0%	1.93	8	**Path**	1262	88.2%	5.28	41
Struct	130	37.6%	1.27	4	**Reference**	276	43.9%	2.32	10
Impl	157	37.6%	1.54	6	**Tuple**	161	30.3%	1.96	8
Trait	144	34.3%	1.55	6	**Impl Trait**	87	20.7%	1.55	10
Use	64	20.3%	1.16	3	**Array**	55	11.4%	1.77	10
Type	29	7.4%	1.45	6	**Trait Object**	49	10.7%	1.69	3
Enum	8	3.0%	1	1	**Ptr**	35	7.8%	1.67	4
Macro	11	3.0%	1.38	2	**Infer**	18	4.8%	1.38	2
Extern Crate	7	2.6%	1	1	**BareFn**	21	4.1%	1.91	5
Static	7	2.2%	1.17	2	**Slice**	13	3.0%	1.62	3
Mod	8	1.9%	1.6	3	**Never**	1	0.4%	1	1
Const	5	1.9%	1	1	**Paren**	1	0.4%	1	1
Verbatim	4	1.1%	1.33	2	**Group**	0	0.0%	0	0
Foreign Mod	2	0.7%	1	1	**Macro**	0	0.0%	0	0
Trait Alias	1	0.4%	1	1	**Verbatim**	0	0.0%	0	0
Union	0	0.0%	0	0					

[1] The total occurrences of each node.
[2] The average occurrences per file, and the highest count in a single file.

feature (4.1%) 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 (12.3%) provides direct access to compiler intrinsics for performance-critical operations. The `custom_mir` feature (11.0%) allows custom transformations on MIR, enabling experimental optimizations and analysis. The most common compilation flags are related to optimization. The `-Zmir-opt-level=X` (45.6%) and `-Copt-level=X` flag (14.0%) controls MIR and LLVM optimizations, respectively. The `-Zmir-enable-passes=+X` flag (15.8%) enables specific MIR passes. The `+nightly` flag (14.0%) 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 (%)**
#![feature(generic_const_exprs)]	17.8%	-Zmir-opt-level=X	45.6%	(?)Sized	49.2%	lifetimes	34.6%
#![feature(type_alias_impl_trait)]	15.1%	-Zmir-enable-passes=+X	15.8%	FnOnce	12.3%	std API	18.6%
#![feature(core_intrinsics)]	12.3%	-Copt-level=X	14.0%	Iterator	7.8%	dyn	10.0%
#![feature(custom_mir)]	11.0%	+nightly	14.0%	Copy	6.2%	async	7.3%
#![feature(const_trait_impl)]	4.1%	–edition=X	12.3%	FnMut	4.6%	core API	6.3%
Total: 73	**24.3%**	**Total: 57**	**18.9%**	**Total: 65**	**21.6%**	-	-

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 21.6% of all cases. Additionally,

18.6% of cases import standard library traits (`use std`), and 6.3% 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 (49.2%) ensures a type has a known size at compile time, while `?Sized` allows dynamically sized types like `str` and `dyn Trait`. The `FnOnce` trait (12.3%) applies to types callable at most once, typically due to ownership constraints. The `Iterator` trait (7.7%) enables value generation, whereas `Copy` (6.2%) allows duplication via bitwise copying instead of moves. The `FnMut` trait (4.7%) permits multiple calls, modifying the captured environment each time. For other language features, lifetimes play a crucial role, with 34.6% 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.

To investigate the performance of these tools, we conduct a *two-step* analysis. In the first step, we determine whether each `rustc` bug in our dataset falls within the scope of an existing tool's capability by examining the submitter's identity. If the issue was submitted by a known developer of a testing tool, we attribute the bug to that tool. If the submitter is a member of the Rust development team, we classify it as reported by the Rust team. All other reports are attributed to general Rust users. 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 cause `rustc` crashes, 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

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[1]	Program generation approaches			Supported features			Testing Method	#Reported Bugs[2]	#Tested Bugs
			Method	Representation	Validity	Unstable	Flag	API			
Community	Fuzz-rustc	2019-07	mutation	Byte Stream	◐	◐	○	◐	Fuzzing	49	1
	Tree-splicer	2023-03	splicing	AST	◐	○	○	◐	Fuzzing	27	0
	ICEMaker	2020-12	mutation	AST	◐	◐	●	◐	Fuzzing	873	0
Academia	RustSmith	2022-04	Rule-based	AST	●	○	●	○	Differential	3	0
	Rustlantis	2023-01	Rule-based	MIR	●	◐	◐	○	Differential	8	0
	Rust-twins	2024-10	LLM-based	Rust code	◐	●	●	●	Differential	8	2
	CLP-Fuzzer	2015-10	Rule-based	Rust code	●	○	○	○	Fuzzing	14	-

[1] 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.

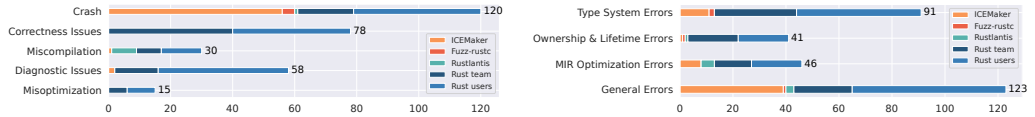
[2] The bugs detected by these tools do not fully align with our dataset. For community-sourced tools, we use their official bug statistics (up to March 3, 2025), and for paper-proposed tools, we record the data from their publications.

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. As shown in validity column in Table 8, the success rate of generating compilable programs varies across different tools. Community-developed fuzzers often produce invalid programs due to 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. As shown in the Supported features column, each tool supports a subset of the high-frequency features summarized in Section 5. 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. Also, we can observe that Fuzz-rustc found 1 bug, and Rust-twins identified 2 bugs, all of which are `rustc` front-end panics. Upon our careful check, the bug found by Fuzz-rustc was previously submitted by ICEMaker[3] and is still in an open state, which seems that it has not been actively maintained or verified since then. The two `rustc` front-end panics discovered by Rust-twins are duplicates, with identical error messages and bug causes. They overlap with another issue submitted by ICEMaker in the past[4], which was closed after Rust developers determined it to be an intentional behavior. Among all detected results, front-end panic is the most frequent and observable bug symptom in `rustc`. We believe that the effectiveness of testing tools may be influenced by a longer testing time and the quality of seed programs.

[3] https://github.com/rust-lang/rust/issues/114920

[4] https://github.com/rust-lang/rust/issues/123950



(a) Distribution of bug symptoms across existing tools.

(b) Distribution of bug causes across testing tools.

Fig. 11. Distribution of bug causes and symptoms across existing tools.

Figure 11 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 11(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 11(b) indicates, the majority of the bugs detected by ICEMaker are caused by general errors, rather than issues related with Rust language mechanism, highlighting ICEMaker's ability to uncover corner cases within `rustc`. An interesting observation from Figure 11(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.

7 Implications and Discussion

The primary goal of this study is to systematically characterize bugs in `rustc`'s semantic analysis and IR processing stages. This characterization offers actionable guidance for testing and analysis of `rustc`, with insights that are valuable both for understanding Rust's type system and for advancing compiler validation techniques.

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.** In our dataset, although 40.9% of the bugs are attributed to general programming errors (Table 3), the HIR (44.9%) and MIR (35.2%) stages remain the most error-prone, as shown in Figure 5(a). This is because HIR and MIR are the stages where high-level constructs are desugared and processed by complex analyses, such as trait resolution, borrow checking, and MIR optimizations, which increases the likelihood of subtle interactions manifesting as bugs. The characteristics of bug-revealing test cases further support this observation. As shown in Table 6, trait-related constructs including traits, `impl traits`, and trait objects frequently appear in both item and type nodes. Moreover, certain unstable trait-related features and the explicit use of lifetimes, as reported in Table 7, also contribute to `rustc` bug manifestation, indicating that these language features may interact with the HIR and MIR modules and thereby increase the likelihood of `rustc` errors.

➤ **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 crash bug often causes panic with

safety protection, setting it apart from other compilers where crash 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: `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 5(b) illustrates that error reporting is scattered across different components, including HIR (14.1%) and MIR (16.0%), 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 4: Existing `rustc` testing tools are less effective at detecting non-crash bugs.** Figure 11(a) shows that about 50% of the crash bugs are detected by existing `rustc` testing tools. On the one hand, non-crash bugs such as soundness and completeness issues often lack directly observable symptoms, making them difficult to detect during development or testing. On the other hand, this suggests that current testing tools are limited to finding easily observed crash bugs with obvious symptoms while remaining unaware of the syntactic and semantic validity of generated programs. As shown in Table 4, certain bug symptoms such as partial front-end panics and completeness issues can only be triggered by valid programs, which indicates that testing tools need to be aware of the validity of programs to find such bugs.

7.2 Actionable Suggestions and Takeaways

➤ **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 5 indicates that 44.9% and 35.2% 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, 24.3% of the bug-revealing test cases apply unstable features, and 18.9% 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 internal validity concerns the selection criteria and representativeness of the bugs analyzed. We focus on fixed bugs accompanied by both a patch and a test case, as they provide concrete, developer-acknowledged issues with sufficient context for analysis. New feature requests, enhancements, non-reproducible issues, and discussion-based reports were excluded to reduce noise. This filtering strategy aligns with prior studies[Chaliasos et al. 2021; Di Franco et al. 2017; Jin et al. 2012; Sun et al. 2016] that similarly concentrate on fixed bugs. We acknowledge that our dataset may not include all reported bugs, particularly those that remain undiscovered or unresolved. While this limitation is inherent to studies based on historical data, we believe our dataset is sufficiently representative for characterizing common issues and also present the developers' knowledge in Rust's semantic analysis and IR processing. Notably, the selected time frame spans the entire lifetime of the Rust 2021 Edition (from January 1, 2022 to January 1, 2025), capturing a complete cycle of development and bug resolution under this edition.

To identify `rustc` bugs relevant to our study, we relied on the official labeling system maintained by the Rust team and filtered for core components related to Rust's safety guarantees. Specifically, we excluded backend-related labels such as "*A-LLVM*" and general compiler labels like "*A-Parser*", as they are not directly involved in semantic analysis or the Rust-specific IR stages we target. While LLVM plays a crucial role in target code generation and low-level optimizations, our focus

is on bugs arising from Rust compilation stages—such as type checking, trait solving, and the generation of HIR and MIR, before lowering to LLVM IR. To ensure precision, we used a label-based filtering strategy and manually verified all selected issues. Despite these efforts, a small number of LLVM-related bugs remain in our dataset. As discussed in Section 2.1, 7 of the 287 backend-labeled issues were also tagged with labels of interest to our study and thus retained. Some of these lead to back-end crashes or are caused by LLVM-level errors. However, the low counts in these categories do not indicate that LLVM-related bugs are uncommon in practice. Instead, they are underrepresented in our dataset, as our study specifically targets bugs tied to the implementation of Rust's core language mechanisms.

Another potential threat lies in the subjectivity of our manual bug analysis. 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 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 studies, we propose the first systematic bug study of `rustc`, which is resilient to memory safety issues and has unique IRs designed for type checking and borrow checking.

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 presents a comprehensive empirical study of `rustc` bugs, analyzing their causes, symptoms, affected compilation stages, and test case characteristics. Our findings offer insights, suggestions, and potential research directions for testing and debugging `rustc`. We observe that bugs involving HIR and MIR occur at comparable rates, with most issues stemming from Rust-specific analyses, checks, and MIR-based optimizations. Moreover, existing test generation techniques for `rustc` are limited, with insufficient support for both correctness and misoptimization bugs. 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 is publicly available [Liu 2025]. We have already submitted the artifact for evaluation via this link: https://zenodo.org/records/16600026.

Acknowledgments

We would like to thank reviewers for their constructive comments. This project was partially funded by the National Natural Science Foundation of China under Grant No. 62372225 and No. 62272220.

A Coding Frame for Issue Labeling

This appendix presents the coding frame used for labeling `rustc` bugs, which involves three manually classified dimensions: bug cause, the compilation stage in which `rustc` bug occurs, and the observed symptom. These categories formed the basis of our manual analysis and supported our empirical findings and suggestions. As described in Section 2.2, our labeling process begins with a predefined set of categories and subcategories for these three dimensions. These categories were informed by prior compiler bug studies and our domain expertise in `rustc`. The annotation was performed iteratively by the first two authors. We conducted 5 rounds of labeling, each round involving 20 randomly selected bug reports. In each round, the two authors independently annotated the bugs using the current taxonomy, then compared their results and discussed any discrepancies. This iterative process allowed us to refine ambiguous or insufficient category definitions and stabilize the taxonomy after labeling 100 reports. Once the labeling taxonomy was finalized, the authors re-annotated the previously labeled bug reports to ensure consistency, and then continued to annotate the remaining dataset using the stable taxonomy. Each batch of 20 bug reports was independently labeled by the same two authors. In cases of disagreement, the third author joined the discussion to facilitate consensus. This rigorous process ensured high reliability of the manual annotations across the entire dataset.

The classification of bug causes and its detailed criteria are presented in Section 3 (see Table 3). The classification scheme for bug causes, which remained unchanged from the initial to the finalized version, is fully presented in Section 3 and thus not repeated here. Therefore, this appendix focuses on the other two dimensions, compilation stage and bug symptom, providing both their initial classification schemes and the refinements made during the analysis process, along with the rationale behind the adjustments.

A.1 Bug Prone Compilation Stages

For compilation stages, we follow the official Rust compiler development guide [Rust 2023a] and identify five categories that correspond to key phases and components of `rustc`: *AST*, *HIR*, *MIR*, *Code Generation*, and *Utility*. Among these, *HIR* and *MIR* are the most critical for enforcing Rust's safety guarantees. To better reflect their internal complexity, we further subdivide these two stages based on functionality. Table 9 summarizes our classification of each compilation stage along with representative modules where bugs are typically fixed. Each module is accompanied by a brief description based on `rustc` documentation and source code comments. For diagnostic-related modules under *HIR* and *MIR*, the table includes selected examples of the module name. In practice, most of these modules contain sub-modules or files dedicated to diagnostics, which are too numerous to list exhaustively.

The classification in Table 9 reflects the final taxonomy, refined through multiple rounds of iterative labeling and revision. Initially, our predefined labels did not include a separate category for "*LLVM*". However, during the labeling process, we encountered several bug-fix pull requests that resolved issues by upgrading LLVM. To better capture such cases, we added "*LLVM*" as a distinct category. Our original taxonomy also did not account for diagnostic-related categories. In practice, we found that many bug-fix PRs modified only diagnostic code. Although these modules vary in naming, such as `error_reporting` or `diagnostics`, they are generally well-structured and clearly identifiable in the `rustc` source code. To explicitly represent them, we introduced a dedicated "*Diagnostics*" subcategory under both the *HIR* and *MIR* stages.

Finally, we clarify the scope of modules listed in Table 9. All of them are located under the "`compiler/`" directory in the official Rust repository. Most of these modules contain several sub-modules and numerous Rust source files, which are not exhaustively shown in the table. Besides,

Table 9. The typical modules and descriptions for compilation stages.

Compilation Stages		Typical Modules	Description
1. AST		`rustc_ast`	Contains syntax-related components such as the AST, token definitions, AST mutation utilities, and shared structures used by the lexer and macro expansion.
		`rustc_ast_lowering`	Lowers the AST to the HIR.
		`rustc_ast_passes`	Implements validation passes over the AST produced by `rustc_parse`, prior to its lowering by `rustc_ast_lowering`.
		`rustc_parse`	Represents the main parser interface.
		`rustc_const_eval`	Evaluates compile-time constant expressions.
		`rustc_resolve`	Responsible for the part of name resolution that doesn't require type checker.
2. HIR	Type & Checking	`rustc_hir_analysis`	Performs semantic checks on HIR.
		`rustc_hir_typeck`	Responsible for: 1. Determining the type of each expression. 2. Resolving methods and traits. 3. Guaranteeing that most type rules are met.
	Trait Solving	`rustc_trait_selection`	Defines the trait resolution method.
		`rustc_next_trait_solver`	Contains the implementation of the next-generation trait solver, along with shared components that were generalized from the old solver during the transition.
	Type Infer	`rustc_infer`	Defines the type inference engine.
		`rustc_hir`	Defines the structure of the HIR.
	Diagnostics	`{*hir}/src/error`	Generates diagnostics for all HIR-related components.
3. MIR	MIR Transform	`rustc_mir_transform`	Applies MIR-based optimizations and passes.
	Borrow Checking	`rustc_borrowck`	Performs borrow checking and ownership analysis.
	Dataflow Analysis	`rustc_mir_dataflow`	Defines the type of the dataflow state, the initial value of that state at entry to each block, as well as the direction of the analysis.
	Diagnostics	`{*mir}/src/diagnostics`	Emits diagnostics for borrow check failures.
4. Code Gen		`rustc_codegen_ssa`	Contains code generation code that is used by all backends (LLVM and others).
5. Utility		`rustc_middle`	Defines shared compiler data structures.
		`rustc_ty_utils`	Provides type system utility functions.
		`rustc_metadata`	Handles crate metadata encoding and decoding.
		`rustc_lint`	Implements lints and warning infrastructure.
6. LLVM		The bug can be resolved by updating LLVM.	

the table does not cover all modules in the "`compiler/`" directory. While we initially assigned compilation stages to all core modules in this directory, some of them did not appear in our dataset and are therefore not included.

A.2 Bug Symptoms

Following the classification principles adopted in prior bug studies [Romano et al. 2022; Shen et al. 2021; Zhou et al. 2021], we initially predefined four categories of bug symptoms to characterize how bugs manifest in `rustc`. The final taxonomy with examples is presented in Section 4 (Table 4). The overview of the mapping from initial to final bug symptom taxonomies is shown in Figure 12. In the following, we describe our predefined categories, the refinements made in the final taxonomy, and the rationale behind each adjustment.

Initial Category (1): ICE. This category refers to internal compiler errors that cause `rustc` to terminate unexpectedly and abnormally. Such failures are typically indicated by panic messages and

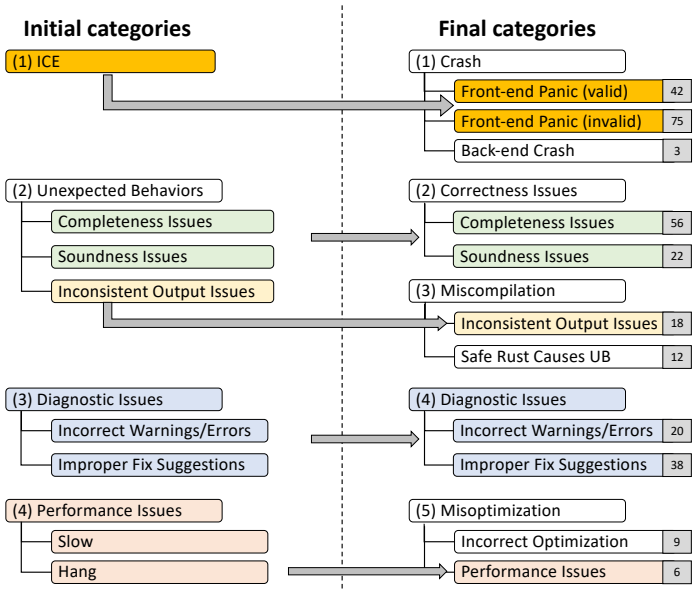


Fig. 12. An overview of the mapping from initial to final bug symptom taxonomies.

explicit mentions of "internal compiler error (ICE)." ICEs reflect violations of compiler invariants and generally reveal severe faults within `rustc`'s implementation.

Final Category (1): Crash. After several iterations of labeling and refinement, the final taxonomy includes a new top-level category, "*Crash*", comprising "*Front-end Panic*" and "*Back-end Crash*". Initially, our focus was exclusively on ICEs caused by faults in `rustc`'s implementation, and we assumed that all crashes would arise from panics during the front-end stages. However, during manual inspection of issues, we observed a small number of cases where the crash was triggered by bugs in the backend components, particularly in LLVM. To account for such cases, we extended our taxonomy to explicitly include "*Back-end Crash*". Furthermore, within the "*Front-end Panic*" category, we distinguished between panics triggered by *valid* and *invalid* Rust programs. This distinction arose from the observation that many panics were caused by malformed inputs, such as random symbols or incomplete fragments, often generated by fuzzing tools. We separately recorded panics from invalid programs, as they may reflect robustness issues in the parser or early-stage analysis, rather than logic errors in handling valid code.

Initial Category (2): Unexpected Behaviors. This category encompasses cases where the behavior of `rustc` deviates from the expected outcomes defined by the Rust language specification. These deviations may compromise correctness, user expectations, or language soundness. The following subcategories are predefined:

- **Completeness Issues:** `rustc` erroneously rejects well-formed Rust programs that should compile successfully according to the Rust specification.
- **Soundness Issues:** `rustc` incorrectly accepts ill-formed code that should have been rejected, allowing it to pass compilation.
- **Inconsistent Output Issues:** `rustc` produces results that deviate from the expected program output. Alternatively, `rustc` yields inconsistent results across different compilation settings, despite the input program remaining semantically unchanged.

Final Category (2): Correctness Issues & (3): Miscompilation. In the final taxonomy, we refined the predefined category of "*Unexpected Behaviors*" by reorganizing it into two higher-level categories: "*Correctness Issues*" and "*Miscompilation*". The "*Correctness Issues*" category includes the predefined subcategories of "*Completeness Issues*" and "*Soundness Issues*", which respectively capture scenarios where `rustc` rejects valid programs or accepts invalid ones. The newly introduced "*Miscompilation*" category covers symptoms where `rustc` produces incorrect code despite successful compilation. This category includes two subcategories: "*Inconsistent Output Issues*" and a new subcategory we define as "*Safe Rust Causes UB*". The latter represents a Rust-specific bug symptom not observed in most other language compilers. According to Rust's safety guarantees, code written entirely in `safe` Rust should not exhibit UB. However, during the annotation process, we encountered several issues where developers reported UB triggered by programs that did not involve any use of `unsafe` blocks. These cases were also acknowledged by `rustc` developers as miscompilations. Thus, we introduced "*Safe Rust Causes UB*" as a distinct subcategory, thereby making our classification more comprehensive and aligned with Rust's unique language semantics.

Initial Category (3): Diagnostic Issues. This category covers bugs in `rustc`'s diagnostic system, which helps developers identify and fix issues. We predefine the following subcategories:

- **Incorrect Warnings/Errors:** `rustc` emits warnings or error messages that are spurious or misleading, potentially resulting in unnecessary code modifications or confusion.
- **Improper Fix Suggestions:** The suggestions or fix hints provided by `rustc` are irrelevant, misleading, or ineffective in resolving the actual issue.

Final Category (4): Diagnostic Issues. We did not revise the category of "*Diagnostic Issues*", as our labeling process consistently revealed that related issues could be attributed to either incorrect or misleading error messages, or problematic compiler suggestions. No additional patterns were observed that would justify a further split or reorganization within this category.

Initial Category (4): Performance Issues. This category includes bugs that degrade the performance of `rustc`, without necessarily affecting the correctness of its output. We predefine the following subcategories:

- **Slow:** `rustc` exhibits unusually long compilation times for input programs.
- **Hang:** `rustc` enters a non-terminating state (e.g., infinite loop or deadlock), failing to complete compilation without external interruption.

Final Category (5): Misoptimization. In the final taxonomy, we replaced the predefined category "*Performance Issues*" with a broader category named "*Misoptimization*", which encompasses both incorrect optimizations and performance-related problems. This change was driven by two main considerations. First, we observed that the number of issues explicitly reporting compilation slowness or runtime performance degradation was very small. Given their rarity and the inherent difficulty in detecting or reproducing such issues, we found it unnecessary to maintain a separate category for them. Second, we identified some issues involving unexpected optimization behaviors. For example, in some cases, specific optimization flags had no effect, or the generated MIR did not match the expected optimizations. We classified these cases as "*Incorrect Optimization*". Because performance problems often result from missing or ineffective optimizations, we consider both types as performance issues. Therefore, we merged them into the unified category "*Misoptimization*", which better reflects the nature of the underlying compiler defects observed in our study.

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.

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

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.

Matthias Krüger. 2020. matthiaskrgr/icemaker: automatially find crashes in the rust compiler & tooling. https://github.com/matthiaskrgr/icemaker. (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

Zixi Liu. 2025. *An Empirical Study of Bugs in the rustc Compiler*. https://doi.org/10.5281/zenodo.16600026

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 Yiying 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(Gen) OS - Redox - Your Next(Gen) 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, Yonghwi Kwon, and Weihang Wang. 2021. An empirical study of bugs in webassembly compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–54.

Alan Romano, Xinyue Liu, Yonghwi 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. Compiletest - Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/tests/compiletest.html. (Accessed on 12/05/2023).

Rust. 2023b. rust-lang/rust: Empowering everyone to build reliable and efficient software. https://github.com/rust-lang/rust. (Accessed on 11/29/2023).

Rust. 2023c. 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. On Undefined Behavior - High Assurance Rust: Developing Secure and Robust Software. https://highassurance.rs/chp3/undef.html [Online; accessed 2025-06-21].

Rust. 2025c. 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. 2025a. Code generation - Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/backend/codegen.html [Online; accessed 2025-06-20].

rust team. 2025b. 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].

Margrit Schreier. 2012. Qualitative content analysis in practice. (2012).

C.B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

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

Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

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.

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

Yixuan Tang, Zhilei Ren, Weiqiang Kong, and He Jiang. 2020. Compiler testing: a systematic literature analysis. *Frontiers of Computer Science* 14 (2020), 1–20.

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 [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 2025-03-26; accepted 2025-08-12