Rust-twins: Automatic Rust Compiler Testing through Program Mutation and Dual Macros Generation

Wenzhang Yang
yywwzz@mail.ustc.edu.cn
University of Science and Technology of China
Anhui, China

Cuifeng Gao
gcf20225162@mail.ustc.edu.cn
University of Science and Technology of China
Anhui, China

Xiaoyuan Liu
lxyyyy@mail.ustc.edu.cn
University of Science and Technology of China
Anhui, China

Yuekang Li
yuekang.li@unsw.edu.au
University of New South Wales
Sydney, Australian

Yinxing Xue*
yxxue@ustc.edu.cn
University of Science and Technology of China
Anhui, China

ABSTRACT

Rust is a relatively new programming language known for its memory safety and numerous advanced features. It has been widely used in system software in recent years. Thus, ensuring the reliability and robustness of the only implementation of the Rust compiler, RUSTC, is critical. However, compiler testing, as one of the most effective techniques to detect bugs, faces difficulties in generating valid Rust programs with sufficient diversity due to its stringent memory safety mechanisms. Furthermore, existing research primarily focuses on testing RUSTC to trigger crash errors, neglecting incorrect compilation results - miscompilation. Detecting miscompilation remains a challenge in the absence of multiple implementations of the Rust compiler to serve as a test oracle.

This paper tackles these challenges by introducing RUST-TWINS, a novel and effective approach to performing automated differential testing for RUSTC to detect both crashes and miscompilations. We devise four Rust-specific mutators and adapt fourteen general mutators for Rust, each intends to produce a syntax and semantic valid Rust program to trigger RUSTC crashes. Additionally, we develop a macroize approach to rewrite a regular Rust program into dual macros with equivalent behaviors but in different implementations. Furthermore, we design an assessment component to check the equivalence by comparing the expansion results with a simple macro input. Finally, RUST-TWINS attempts to expand the two macros with numerous complex inputs to detect differences. Due to the macro expansion mechanism, the root causes of differences can arise not only from the macro expansion part but also from any other mis-implemented compiler code.

We have evaluated RUST-TWINS on the latest version of RUSTC. Our experimental results indicate that RUST-TWINS achieves twice the total line coverage and identifies more crashes and differences than the best baseline technique, RUSTSMITH, after 24 hours of testing. In total, RUST-TWINS triggered 10 RUSTC crashes, and 229 of the generated macros exposed RUSTC differences. We analyzed and reported 12 previously unknown bugs, of which 8 have already been confirmed and fixed.

*Yinxing Xue is the corresponding author.

KEYWORDS

Rust, Compiler Testing, Differential Testing

ACM Reference Format:
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 *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695059

1 INTRODUCTION

As a relatively new system programming language, Rust has gained widespread use and has become one of the world's most admired languages for years [5]. Rust ensures memory safety through several safety rules, and its performance is comparable to traditional system programming languages such as C and C++. To benefit from safety and performance, numerous safety-critical, low-level system software applications are being rewritten in Rust [1, 6, 21, 29, 31]. However, any defects in Rust compiler (RUSTC) can result in unreliable behaviors and security breaches in these systems. Consequently, there is a need to develop a novel test approach to effectively detect defects for RUSTC to improve the quality of Rust applications.

Compiler Testing is an effective technique to find compiler bugs [9]. However, automatically employing it for RUSTC remains two primary challenges, preventing the existing techniques from effectively exposing defects in RUSTC. The **challenge❶** is generating valid Rust programs that can reach the mid- and back-end compiler passes of RUSTC, since any Rust programs that violate safety rules [37, 43] would be rejected in the early stages of RUSTC. Existing fuzzing techniques tailored to safety rules are incomplete for Rust. For example, RUSTSMITH [32] does not support advanced Rust features like concurrency, traits, and unsafe code, all of which are heavily related to safety rules, and the CLP-based RUSTC fuzzer [13] only focuses on type-checker bugs. The **challenge❷** lies in how to construct an oracle to detect the miscompilation bug without multiple Rust compiler implementations. Due to the short history of Rust, it has only a single complete compiler implementation, whereas differential testing requires multiple implementations as test oracles to identify errors. Although there are some incomplete and work-in-progress third-party Rust compilers, such as GCCRS and MRUSTC, using them as oracles like existing techniques [7, 11, 22, 24, 32, 42] would result in numerous false positives and would not be able to test unsupported programming language features.

In this paper, we present RUST-TWINS, a tool designed to automatically test RUSTC for both crashes and miscompilations. The key novel insight is that, in Rust, there are two kinds of independent macro systems, each with different expansion mechanisms: one expands in the front end of RUSTC, while the other one involves the entire compilation process from parsing to code generation. Consequently, any bugs in the compilation would result in incorrect macro expansion results. Therefore, we can consider these dual macro systems as differential implementations to construct a test oracle within a single compiler implementation.

Overall, RUST-TWINS automatically mutates Rust programs and rewrites a regular Rust program into a pair of 'twin' macros to serve as a testing oracle. Specifically, RUST-TWINS takes a seed pool as the initial corpus and selects one seed to be mutated with the highest priority based on four metrics. In the mutation component, RUST-TWINS randomly picks a mutation prompt from the prompts pool and asks a large language models (LLMs) to modify the seed according to the corresponding prompt. Consequently, RUST-TWINS leverages RUSTC to compile the mutated seed and detect potential defects. Furthermore, the selected seed is sent to the macroize component to help generate two equivalent macros using the LLMs. However, since the generated macros are not always equivalent, RUST-TWINS employs an assessment component to improve the success rate of generation by expanding it with our predefined default arguments and comparing the expanded results. Finally, RUST-TWINS asks the LLMs to generate different macro inputs and combines some historical macro inputs to expand the dual macros into High-level IR (HIR) [30] and identify differences.

To address **challenge❶**, we propose a mutation approach that consists of seed prioritization and eighteen LLM-driven mutators designed to mutate Rust programs without syntax and semantic errors. For seed prioritization, we use four metrics to measure the probability of triggering defects: coverage, function size, execution time, and lifetime complexity. Notably, the lifetime complexity is computed by the number of lifetime variables in a Rust function; the more lifetime variables, the higher the likelihood of triggering errors in the type- and borrow-checker. For LLM-driven mutators, we devise four Rust-specific mutators and adapt fourteen existing mutators as mutation prompts for the LLMs. Consequently, the LLMs can modify Rust programs by the mutation prompts without safety rules violations, and each of these prompts contributes to mutation diversity, increasing code coverage.

To tackle **challenge❷**, we design the macroize component to automatically obtain dual equivalent macro implementations: macro by example (MBE) and procedural macro (PCD) [3]. MBE is a declarative-style macro, similar to the macros in C/C++, while PCD is an imperative-style macro with stronger programmability. To ensure the equivalent behavior of the dual implementations, we devised a list of default macro expansion values covering all potential input types. Once the expansion results of the dual macros are equivalent with the same input, RUST-TWINS considers them a valid oracle. Otherwise, a regeneration process for macroize would be performed. Consequently, RUST-TWINS attempts to expand the macros with more complicated inputs from historical storage and currently generated inputs from LLMs, and identifies the differences in the expanded results.

To evaluate RUST-TWINS, we pre-processed RUSTC's unit tests as initial seeds and applied RUST-TWINS to RUSTC. We compared RUST-TWINS with LIBFUZZER and the state-of-the-art RUSTC fuzzer RUSTSMITH. The experimental results show that RUST-TWINS's coverage is 2x more than RUSTSMITH's. In total, we generated 6048 Rust programs and 3552 macros with 73.5% and 49% stillborn rate respectively; 239 of them triggered differences and crashes in RUSTC. We reported 12 previously unknown bugs, of which 8 have already been confirmed or fixed.

The main contributions of this paper are as follows.

- **Mutation Component**: We devise 18 program mutation prompts, 4 of which are tailored to Rust's unique features, such as lifetime annotations and ownership. The remaining 14 prompts are adapted from previous mutation-based fuzzers. Using these prompts, we can generate valid Rust programs as test cases by LLMs to trigger compiler crashes.
- **Macroize Component**: To the best of our knowledge, we are the first to develop an approach that generates oracles for compiler testing using its two advanced macro systems within a single compiler implementation. Due to the mechanism of macro expansion, this approach allows us to detect defects caused by any part of the compiler, not just those limited to the front end.
- **RUST-TWINS**: We build an effective differential testing tool named RUST-TWINS for Rust by integrating mutation and macroize components. The experimental results demonstrate that RUST-TWINS outperforms baseline techniques and can detect bugs in the wild. In total, 12 bugs were found in RUSTC, with 8 of them being fixed or confirmed by the developers.

To support the open scientific, all the code of RUST-TWINS and experimental data can be found at https://sites.google.com/view/rust-twins/index

2 BACKGROUND

This section provides the background for this paper, covering Rust's two kinds of macro systems and code generation challenges caused by Rust's safety rules.

2.1 Macro System of Rust

Macro is a meta-programming feature designed for expert programmers. Generally, a macro accepts some code representations and generates new code at compile time. Macro systems can greatly enhance the expressive capability of programming language. In Rust, there are two different kinds of macro systems. Specifically, MBE is a declarative macro that matches the input by token pattern, while PCD provides a programmable way to generate a token stream as

```
1  #[macro_export]
2  macro_rules! mbe {
3      ($main_body:expr) => {
4          fn main() {
5              $main_body
6          }
7      };
8  }
9
10
```

(a) Macro by example (MBE)

```
1  #[proc_macro]
2  pub fn pcd(input: TokenStream)
3      -> TokenStream {
4      let input_expr: Expr =
5          parse(input).unwrap();
6      let expanded = quote! {
7          fn main() {#input_expr}
8      };
9      TokenStream::from(expanded)
10 }
```

(b) Procedural macro (PCD)

Figure 1: Rust's macro systems

```
1  let foo = String::from("foo");
2  let bar = foo;
3  println!("{}", foo);
4
```

(a) Ownership rule violation

```
1  fn foo<'a: 'b, 'b>(fst: &'a i8,
2      snd: &'b i8) -> &'a i8 {
3      return snd;
4  }
```

(b) Lifetime rule violation

Figure 2: Safety rule violations

code. Since PCD supports complete Rust features, RUSTC first compiles PCD as a library, and uses it to execute the generation logic in link time. All Rust macros can be invoked using the ! operator, such as the commonly used `println`! macro.

As shown in Figure 1a, a built-in macro named `macro_rules` is used to define the MBE. In line 1, the attribute declares the macro visibility. This macro definition contains a single pattern, from lines 3 to line 7, which accepts an expression token and generates a main function definition with the accepted expression. In MBE, the parameter variable is referenced by starting symbol `$`, such as `$main_body` in line 5.

In Figure 1b, the function pcd with attribute `#[proc_macro]` is a procedural macro. A PCD is simply a function that accepts a token stream and returns another token stream. In line 5, the input is parsed as an expression token. Then, from lines 6 to 8, the built-in macro `quote`! is invoked with the parsed expression token `#input_expr` to construct a main function. In line 9, the expanded result is converted to a token stream.

Both macro systems provide approaches to transform and generate code at compile time, with partially overlapping functions. Additionally, RUSTC implements MBE in the compiler frontend, while PCD must be compiled through all the compiler phases first.

2.2 Safety Rules of Rust

To ensure memory safety, Rust leverages two fundamental concepts: ownership and lifetime. Each value in Rust can be owned by only one variable. Once the variable's lifetime ends (i.e., the execution flow exits the variable's lexical scope), the value is freed. For example, in Figure 2a, the variable `foo` owns the string value `"foo"` at line 1, and this value is moved to the variable `bar` at line 2. Consequently, Rust forbids the usage of `foo` at line 3 until another value is assigned to `foo`.

The prevention of memory errors in Rust extends beyond ownership. Rust provides lifetime annotations for value references in

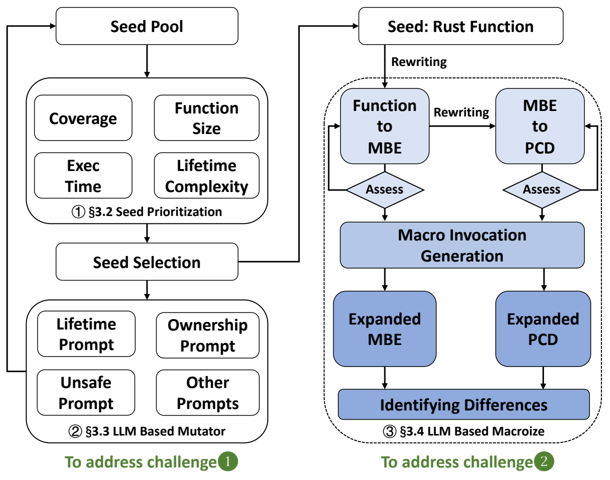


Figure 3: Workflow of RUST-TWINS.

function type signatures. In Rust, the lifetime of a reference is represented by a quote following a name, such as `'a`. Additionally, developers can specify relationships between lifetime variables using the typing `'a: 'b`, indicating that `a`'s lifetime must be at least as long as `b`'s. If these lifetime constraints are not satisfied, RUSTC raises a compile-time error. For example, in Figure 2b, the function `foo` has two arguments that are references to `i8`, with `fst`'s lifetime being at least as long as snd's. The return type `&'a` indicates that its lifetime is also at least as long as `snd`'s. However, the argument `snd`, which has a shorter lifetime, is returned at line 3, causing a violation of the lifetime rules.

All the compile-time rules restrict developers from casually creating aliases and mutability. These rules force users to carefully consider how a value is used and how long it lasts.

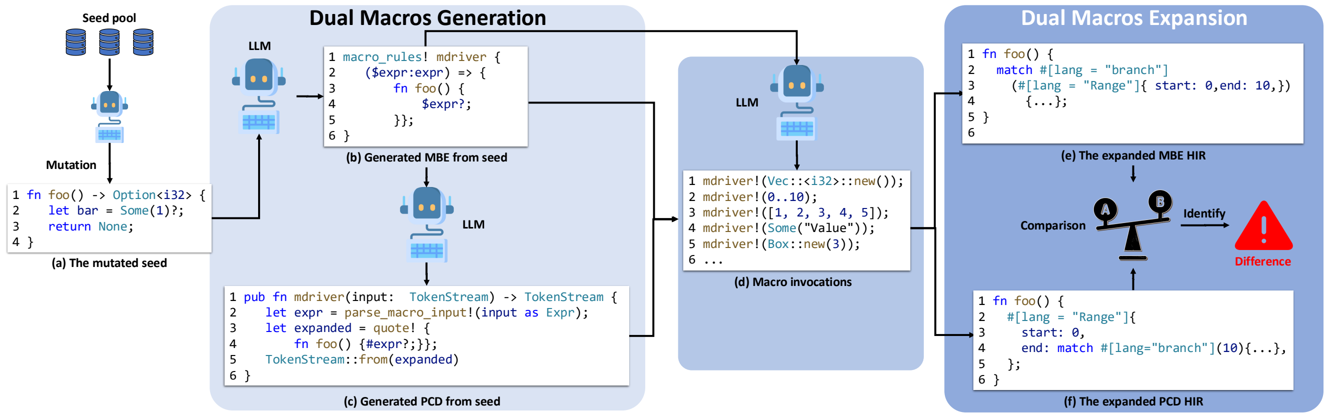
3 PROPOSED APPROACH

In this section, we begin by introducing the workflow of RUST-TWINS. Then, we illustrate how RUST-TWINS prioritizes seeds to effectively select them. After that, we demonstrate the Rust-specific mutators that address the challenges of Rust code generation caused by the ownership and borrow checker. Finally, we detail how to use the large language model to rewrite Rust functions as macros and identify the differences.

3.1 Overview

Workflow. Figure 3 illustrates the workflow of RUST-TWINS. The input to the overall framework is a set of initial seeds and the outputs are detected differential cases. We extract functions from the RUSTC code base and the unit test cases as RUST-TWINS's seed pool. RUST-TWINS consists of three components: ①*seed prioritization*, ②*mutator* and ③*macroize*.

In step ①, RUST-TWINS uses a multi-objective optimization to prioritize the seeds based on various objectives. We propose four objectives using both static and dynamic information. Notably, the lifetime complexity metric is designed specifically for Rust's unique lifetime features. In step ②, RUST-TWINS takes the highest priority seed as input and randomly selects a mutation prompt to modify



```
fn foo() -> Option<i32> {
   let bar = Some(1)?;
   return None;
}
```

```
macro_rules! mdriver {
   ($expr:expr) => {
       fn foo() {
           $expr?;
       }};
}
```

```
pub fn mdriver(input:  TokenStream) -> TokenStream {
   let expr = parse_macro_input!(input as Expr);
   let expanded = quote! {
       fn foo() {#expr?;}};
   TokenStream::from(expanded)
}
```

```
mdriver!(Vec::<i32>::new());
mdriver!(0..10);
mdriver!([1, 2, 3, 4, 5]);
mdriver!(Some("Value"));
mdriver!(Box::new(3));
...
```

```
fn foo() {
  match #[lang = "branch"]
    (#[lang = "Range"]{ start: 0,end: 10,})
      {...};
}

```

```
fn foo() {
  #[lang = "Range"]{
    start: 0,
    end: match #[lang="branch"](10){...},
  };
}
```

Figure 4: How RUST-TWINS detects the bugs

the seed using LLMs. The mutated seed is then placed back into the seed pool for the next round of testing. Simultaneously, the selected seed is also sent to step ③, where RUST-TWINS queries LLMs to rewrite the seed as MBE. Once the first rewriting is successful, RUST-TWINS asks LLMs to rewrite the generated MBE into PCD. After rewriting, RUST-TWINS assesses the correctness of these two generated macros by checking whether they expand to the same results in a simple context. If so, RUST-TWINS queries LLMs to generate numerous macro invocations and attempts to identify any differences. If not, RUST-TWINS repeats the rewriting process until it passes the correctness assessment.

Motivating Example. Figure 4 illustrates the details of each step to detect a miscompile bug, along with the corresponding generated code and results. In Figure 4a, the code serves as an initial seed for macroize and undergoes several mutation processes. In the function body, an option value `Some(1)?` attempts to extract the value `1` and assign it to the variable `bar`.

Consequently, RUST-TWINS queries the LLMs to rewrite the seed and obtain a MBE as shown in Figure 4b. Specifically, this MBE is named `mdriver`, and the LLMs take the function body as the macro's argument `$expr` following the operator `?`. Subsequently, a PCD is generated based on the MBE by the LLMs, as depicted in Figure 4c.

After RUST-TWINS collects the two macros, it performs a simple assessment to ensure that these two macros exhibit identical behavior. Consequently, RUST-TWINS instructs the LLMs to generate ten invocations for the MBE based on the input token type `expr`. The generated invocations maintain sufficient diversity of inputs, as shown in Figure 4d. Among them, the invocation `mdriver`(0..10)! triggers a difference between the MBE and the PCD as illustrated in Figure 4e and Figure 4f.

For the expanded MBE HIR, the outermost code, the **`match`** expression, is derived from the operator `?`. However, in the expanded PCD HIR, the outermost code is the `Range` expression, which is derived from the operator `(0..10)`. Clearly, these two compile results lead to a miscompile bug. RUST-TWINS identifies the differences in HIR by comparing the HIR dump strings character by character and raises a detected error

3.2 Seed Prioritization

RUST-TWINS utilizes multi-objective optimization to prioritize the seeds. In this section, we introduce the metrics that RUST-TWINS employs to construct the objectives and explain the rationale behind their selection.

3.2.1 Metrics. In summary, there are four metrics in RUST-TWINS: *coverage*, *function size*, *compilation time*, and *lifetime complexity*. The reasons for using these metrics are as follows:

- *Coverage (M_1).* Since the mutators we used tend to combine multiple seeds and increase the programming language features of seeds, the coverage function is almost monotone increasing. Consequently, higher coverage seeds are more likely to reach new code regions and good coverage.
- *Function size (M_2).* The larger a function's codebase, the greater opportunities to detect expansion differences.
- *Compilation time (M_3).* In Rust, a constant function's execution during compilation can result in long compilation time. RUST-TWINS attempts to detect unterminated compilation by favoring seeds with longer compilation time.
- *Lifetime complexity (M_4).* Intuitively, seeds with complex lifetime cover more code of ownership- and borrow-checker. Thus, RUST-TWINS prefers functions with more lifetime variables and lifetime subtyping relations. Notably, the metric M_4 has two different impact factors: lifetime variables (l_1) and lifetime subtyping relations (l_2). We compute M_4 for a Rust function using the formula $M_4 = |l_1| + 2 \times |l_2|$, where $|l_1|$ is the number of lifetime variables and $|l_2|$ is the number of lifetime subtyping relations.

3.2.2 Objective Function of RUST-TWINS. We revise the objective function for RUSTC fuzzing with the aforementioned multiple metrics. The definition is as follows:

Definition 3.1 (Objective Function of Seed Prioritization.). Given a set of seeds $\mathbb{S}$ and multiple objectives O_n for n ranging from 1 to k, the goal of seed prioritization is to select a seed s^* from $\mathbb{S}$ that:

$$s^* = \arg\max_{s \in \mathbb{S}} (O_1(s), O_2(s), \ldots, O_k(s))$$

In RUST-TWINS, we introduce the above four metrics as optimization objectives. Here is the mapping between metrics and objectives: $O_n = M_n$, we aim to maximize all the metrics to achieve the best testing performance. To optimize the objective function, we adopt the nondominated sorting (NS) algorithm from CEREBRO [23]. The basic idea of NS is to efficiently calculate the most edge seeds (Pareto frontier) by ranking the seeds, and pop seeds in lower rank.

3.3 Mutator

Due to Rust's strict compilation rule checking, designing mutators for it can be tedious. Thus, RUST-TWINS leverages large language models to assist in mutating Rust programs. We devise four Rust-specific mutation prompts to accommodate the static checker of Rust. Additionally, we adapt 14 mutators for Rust from previous works [14, 28]. In each mutation round, RUST-TWINS attempts to randomly select a mutation prompt using the under-mutate seed to construct the LLM prompt.

3.3.1 *Mutation Prompts.*

The prompts are summarised in Table 1, and are categorised into Rust-specific, GrayC [14] and DIE [28].

Rust-specific(lines 1-4 in Table 1). For this category, all the mutators are related to Rust's unique features. For example, the LIFETIME mutator converts non-reference type arguments to reference type arguments and adds lifetime variables to these references. If a Rust function has two lifetime variables, the OUTLIVE mutator adds outlive lifetime bounds for these two variables. The OWNERSHIP mutator attempts to introduce more ownership transformations and additional borrows to complicate the function body. The last mutator, UNSAFE, adds unsafe code blocks, allowing dereferencing of raw pointers, which may cause memory unsafety.

GrayC and DIE(lines 5-18 in Table 1). Besides the Rust-specific mutators, we adapt fourteen mutators from previous mutation-based compiler testing: DIE and GrayC. From lines 5 to 16, the mutators manipulate single statements or expressions to mutate the original program, while lines 17-18 contain mutators that are multiple-input transformers. The three mutators in DIE tend to change the program without any structural modifications. In GrayC, the mutators do not preserve program structures. For example, DUPLICATE and INJECT-CONTROL-FLOW may insert break or continue statements into a loop structure, causing control flow modifications. The function combination mutators (lines 17-18) take a source program and a destination program and combine them by replacing, concatenating, and interleaving their bodies. In previous Abstract Syntax Tree (AST) based mutation approaches [14], these types of mutators have to maintain the symbol table to avoid duplicate names and ownership conflicts in Rust. Fortunately, we can combine two functions more efficiently by leveraging LLMs to satisfy semantic checking.

3.3.2 *Query Template.*

The mutation query prompt consists of three components: prompt head, mutate action and under-mutate program. The query template is shown as follows:

```
Return the pure Rust code between ```rust and ```.
Don't explain the code and generate the rust code
    block itself.
{mutation_prompt}
{under-mutate Rust programs}
```

In each mutation round, RUST-TWINS randomly selects a mutation action prompt. If the prompt requires two programs (source and destination), RUST-TWINS randomly picks another program from the seed set as the second input. After preparing the inputs, it replaces the placeholders in the query template with the mutation action prompt and the selected programs. Notably, we do not design an iterative querying mechanism for mutation, such as re-querying if the programs produced by the LLMs are uncompilable. In our opinion, this approach benefits the testing of the error handling component of RUSTC when the mutation results in invalid output.

Algorithm 1 Macroize Component. *($\mathbb{E}(m)$ is m's expanded result with default input, $\overline{\mathbb{E}}(m, i)$ is m' expanded result with input i)*

```
Input: Rust function
Output: Identified difference
1: function MACROIZE(F)
2:     $mbe \leftarrow requireLLMGenMBE(F)$
3:     $pcd \leftarrow requireLLMGenPCD(mbe)$
4:     while $\mathbb{E}(mbe) \neq \mathbb{E}(pcd)$ do
5:         $mbe \leftarrow requireLLMGenMBE(F)$
6:         $pcd \leftarrow requireLLMGenPCD(mbe)$
7:     end while
8:     $typ \leftarrow getInputType(mbe)$
9:     $invs \leftarrow requireLLMGenInvocations(typ)$
10:    $selectedInvs \leftarrow selectK(invsStore[typ]) + invs$
11:    $invsStore[typ].add(invs)$
12:    while selectedInvs.isNotEmpty() do
13:        $inv \leftarrow selectedInvs.pop()$
14:        if $\overline{\mathbb{E}}(pcd, inv) \neq \overline{\mathbb{E}}(mbe, inv)$ then
15:            return difference
16:        end if
17:    end while
18: end function
```

3.4 Macroize

Finding errors in RUSTC doesn't end with generating data and detecting crashes. To ensure the correctness of the compiled results, we need to compare the behaviors of the compiled artifacts. However, the challenge lies in collecting pairs of code that have the same behaviors but different implementations.

RUST-TWINS tackles this challenge through three steps in the macroize component. First, we use LLMs to rewrite a Rust function into both an MBE macro and a PCD macro. The goal of these two macros is to produce identical code but in different ways. Next, we leverage a behavior assessment to ensure these two macros expand to the same HIR for the same arguments. To achieve this, we design default macro arguments covering all token types to expand these two macros into HIR. If the expanded HIRs differ, RUST-TWINS returns to the rewriting phase and prompts LLMs to regenerate the macros. Finally, if the HIRs are the same, RUST-TWINS generates ten macro invocations for each macro using LLMs. In the differential checker, RUST-TWINS reports a potential error if differences exist in the ten expanded HIRs.

Table 1: Mutation prompts of RUST-TWINS

#	Type	Short Name	Prompt
1	Rust-specific	LIFETIME	Change certain function types to references, ensuring to add the corresponding lifetime annotations. If any types are currently missing, please add them.
2		OUTLIVE	Modify function types to use references where appropriate and add the corresponding lifetime annotations. If multiple lifetime variables are present, establish an outlive relationship between these variables. If there are no existing lifetime variables, introduce additional reference types and lifetime variables, and create an outlive relationship for them.
3		OWNERSHIP	Make the ownership of this function more complex by transforming and swapping the positions of two statements.
4		UNSAFE	Change a part of this function body to use unsafe block. Some references should be changed to raw pointers.
5	DIE	REPLACE-AP	Select a piece of code and replace it with new code, the selected code serves no structural purpose
6		USE-VAR	Locates a statement block (e.g., the body of an if statement, a function or simply the global region) and randomly selects a code point inside the block for insertion. Next, generates a new expression statement by using the existing variables declared at the point.
7		INTRO-VAR	Insert the declarations of new variables at random code points, the variable is initialized by an expression.
8	GrayC	DUPLICATE	Duplicate a statement within the same block excluding variable declarations.
9		EXPAND	Expand sub-expression with other sub-expressions from the corpus program; e.g. in an assignment or loop condition.
10		REPLACE-BY-CONSTANT	Replace an expression with a random valid constant expression of the same data type.
11		FLIP-BIT	Flip a bit in a constant expression.
12		REPLACE-DIGIT	Similar to Flip-Bit but on the number's decimal representation: either flip the sign or change a single digit.
13		CHANGE-TYPE	Change the type of an expression (short, long, unsigned, float, etc.).
14		REPLACE-UNARY-OPERATOR	Replace unary operator with an assignment using the same variable
15		FLIP-OPERATOR	Replace one operator with another (arithmetic operators).
16		INJECT-CONTROL-FLOW	Add a break, continue or return statement inside a loop. The statement is guarded by a condition based on an auxiliary loop counter so that it is only invoked on certain iterations.
17		REPLACE-FUNCTION-BODY	Replace the body of a function with that of another function with the same number of arguments.
18		COMBINE FUNCTION	Combine the body of a function with another function with the same number of arguments, either by concatenating bodies or interleaving their statements.

Table 2: Default inputs for each token type. (TT is short for Token Type.)

TT	block	expr	ident	item	lifetime	meta	tt
Value	`{1}`	`1`	`a`	`fn a() {}`	`'a`	`#[a]`	`1`
TT	**pat**	**path**	**literal**	**stmt**	**ty**	**vis**	/
Value	`a`	`a`	`1`	`let a = 1;`	`i32`	`pub`	/

Algorithm 1 details how RUST-TWINS generates macros. As our goal is to use LLMs to generate two identical behavior macros from single Rust function, we provide the following prompt to the LLMs: "*I have a piece of Rust code. Please write a Rust macro to generate this code. There are some requirements for the generated macro: 1. The macro name is 'mdriver'. 2. The macro type is a macro by example. 3. The macro is a function-like macro. 4. The macro has only one parameter. 5. Do not forget export the macro, such as using '#[macro_export]'. 6. The code should be wrapped by Rust code block. 7. Do not use any external crate except syn and quote*". In this prompt, we specify seven requirements to facilitate easy expansion and error prevention, such as a universal macro name and a single parameter. In the motivating example, shown in Figure 4e and Figure 4f, the generated macros have the same name and input parameters

Due to hallucination [39] of LLMs, the generated macro might miss some of the given requirements. To ensure the dual macros have same behavior, RUST-TWINS compares the expanded results with simple inputs to assess the generation correctness. From lines 4-7, RUST-TWINS repeatedly prompts LLMs to generate MBE and PCD macros until the expanded results match. The function $\mathbb{E}$ expands the given macro with a default argument. We revise 13 default arguments for $\mathbb{E}$ to cover all possible token types, details are shown in Table 2. We intentionally use simple inputs as the default for filtering out inequivalent generations. The rationale is based on an observation that inequivalences are obvious and can be filtered out easily using simple inputs. If we use more complex test inputs, we may trigger real bugs instead of detecting inequivalences, which could cause false negatives. For instances, in Figure 4e and Figure 4f, RUST-TWINS use the default argument `1` to expand the dual macros since the input type in Figure 4e is `expr`.

Once the pair of macros is confirmed, RUST-TWINS attempts to extract the input token type from MBE instead of PCD at line 8, since the former exhibits an explicit type signature while PCD's input type is always `TokenStream`. To reduce the cost of the LLM token fees, RUST-TWINS reuses historically generated invocations, which are stored and categorized by their corresponding types. In lines 9-10, RUST-TWINS requires the LLMs to generate invocations that satisfy the extracted type of MBE, selects `K` invocations from the invocation store by `typ`, and combines the generated invocations to construct `selectedInvs`. We provide the following prompt to the LLMs: "*Please give me an invoke for this macro, There are some requirements for the generated invocation:1. 10 differnet macro invocations are needed. 2. The invocation parameter diversity should be strong enough. 3. The code should be wrapped by "' rust "'. Here are the macro type: MBE input type*". Consequently, the newly generated invocations are saved in `invsStore` at line 11. In lines 12-17, RUST-TWINS expands PCD and MBE with these inputs and compares the expanded HIR to identify any differences. For motivating example, the invocations listed in Figure 4d, it covers many features of Rust, such as `range`, `array`, `Box`, etc. Consequently, RUST-TWINS gets two expanded results by the macro invocation `mdriver`(0..10)!, as shown in Figure 4e and Figure 4f. By comparing the dumped results, RUST-TWINS identifies a difference as a potential bug in RUSTC.

4 EVALUATION

Research Questions. We conduct experiments to evaluate RUST-TWINS by addressing the following questions:

- **RQ1:** Can RUST-TWINS generate a sufficient amount of valid Rust code?
- **RQ2:** How effective is RUST-TWINS in testing RUSTC?
- **RQ3.** What are the root causes of detected differences and crashes?

Implementation. We implemented RUST-TWINS using RUSTC version 1.80.0-nightly, the Rust libfuzzer binding libfuzzer-sys version 0.4.7. For seed prioritization, we implemented the nondominated sorting algorithm in C++ by modifying the `ChooseUnitToMutate` function in libfuzzer. RUST-TWINS mutates and rewrites Rust programs using the GPT-4-turbo API. Since MBE's pattern arms are difficult to parse with the crate syn [2], RUST-TWINS extracts the token type using regex. The generated macro is expanded into HIR, which is then dumped as a string for comparison. The compilation flag for expansion and dump is `-Z unpretty=hir`.

Baseline techniques. Two techniques are chosen as our experimental baseline. One of them is RustSmith [32], a generative compiler fuzzing technique that generates Rust code through the Rust grammar and carefully maintains ownership during generation. The other tool is libFuzzer [4], a byte-level mutation-based fuzzer without initial corpus.

Evaluation Setup. Our experiments are performed on a Linux server with Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz and distribution Ubuntu 20.04. RUST-TWINS's initial corpus is a collection of single-file programs from RUSTC's unit tests. We filter the uncompilable and crash-causing programs, remove all comments, newlines, and blanks from each file to reduce the LLM token fees. We set the temperature parameter at 0.5 for the GPT-4o.

Metrics. Four metrics are designed to evaluate RUST-TWINS against other techniques.

- **Stillborn rate.** To quantify the Rust mutation and rewriting capability of LLMs, we compute the stillborn rate of both mutator and macroize components.

$$SR = 1 - \frac{|Valid|}{\#Queries} \times 100\% \tag{1}$$

  Here, $Valid$ refers to the valid seeds or rewritten macros, $|Valid|$ is their count, and $\#Queries$ is the number of requests made to the LLMs. In this paper, stillborn artifacts are Rust programs with invalid semantics or syntax, or macros that cannot pass our assessment. The poorer the generation capability of the LLMs, the higher this rate will be.
- **Code diversity.** To evaluate the range of Rust features covered by the generated code, we sample 1,000 files from the program set produced by RUST-TWINS and RUSTSMITH respectively. We then counted the number of different kinds of Rust AST nodes, categorized into 167 types by the crate syn version 1.0.66. The greater the range of types a seed set has, the better its diversity.
- **Code coverage.** To measure whether our mutation strategies can generate effective and sufficient seeds to achieve higher coverage than the state-of-the-art tools, we compute two types of coverage: overall coverage and module coverage. Overall coverage records the line coverage of the entire RUSTC per hour for RUST-TWINS and baseline techniques. Additionally, to determine the coverage improvement for each module in RUSTC, we compute the line coverage of each module for RUST-TWINS and RUSTSMITH.
- **Differences and crashes.** We collect the expansion differences between MBE and PCD and document RUSTC crashes. Each detected case is analyzed, and potential bugs are reported to RUSTC developers via GitHub issues.

Table 3: Stillborn rate. (SR is short for stillborn rate, and RUST-TWINS$_{mut}$ and RUST-TWINS$_{mac}$ denote the mutation component and macroize component of RUST-TWINS respectively)

	RustSmith	*Libfuzzer*	RUST-TWINS$_{mut}$	RUST-TWINS$_{mac}$
Valid	20712	0	6048	3552
Invalid	68	335540	16800	3420
Total	20784	335540	22848	6972
SR	0.33%	100%	73.5%	49%

4.1 Experimental Results

4.1.1 RQ1: Can RUST-TWINS generate a sufficient amount of valid Rust code? To evaluate the performance of RUST-TWINS, we initiate a 24-hour fuzzing campaign. We compare the stillborn rate and code diversity of RUST-TWINS against the baseline techniques, RUSTSMITH and LIBFUZZER. Consequently, we discuss the generation cost and the compilation errors of invalid programs.

Stillborn rate. As indicated in Table 3, RUST-TWINS successfully generates 6,048 Rust programs in the mutation component and 3,552 pairs of macros in the macroize component. Since the rewriting task is easier than the mutation task for GPT-4o, the SR of macroize is better than that of the mutation component. Although LIBFUZZER can generate the highest number of Rust programs, it cannot generate any valid Rust programs for RUSTC with its random byte-level mutation strategy. RUSTSMITH has the best SR, 0.33%, nearly generating correct Rust programs. Our analysis of the invalid programs it generated shows that the most common error is missing identifiers. Despite this, RUSTSMITH demonstrates impressive performance in terms of generation speed. However, the code it generates tends to be rigid, a point that we discuss in the code diversity paragraph.

Code diversity. To measure the diversity of generated Rust programs, we count the number of AST nodes by their type. As shown in Figure 5, we compute the percentage of each node type for both RUST-TWINS and RUSTSMITH. The larger the area the tool occupies, the higher the percentage of that node type. In total, RUST-TWINS generates seeds that cover all node types, while RUSTSMITH generates only 84 types and misses 83 types. The largest differences in the number of generated AST types between RUST-TWINS and RUSTSMITH are found in `Item` and `Expr`. RUST-TWINS generated 7.68 times more `Item` AST node than RUSTSMITH due to its tendency to generate more functions and function calls. Conversely, RUSTSMITH generates 21% more `Expr` AST node than RUST-TWINS. For both of them, the top five node types are `Span`, `Ident`, `Expr`, `PathArgument`, `PathSegment`, and `Path`. However, the standard deviation of RUST-TWINS outperforms that of RUSTSMITH by 8.3%. The node type distribution results show that the diversity of RUST-TWINS is better than that of

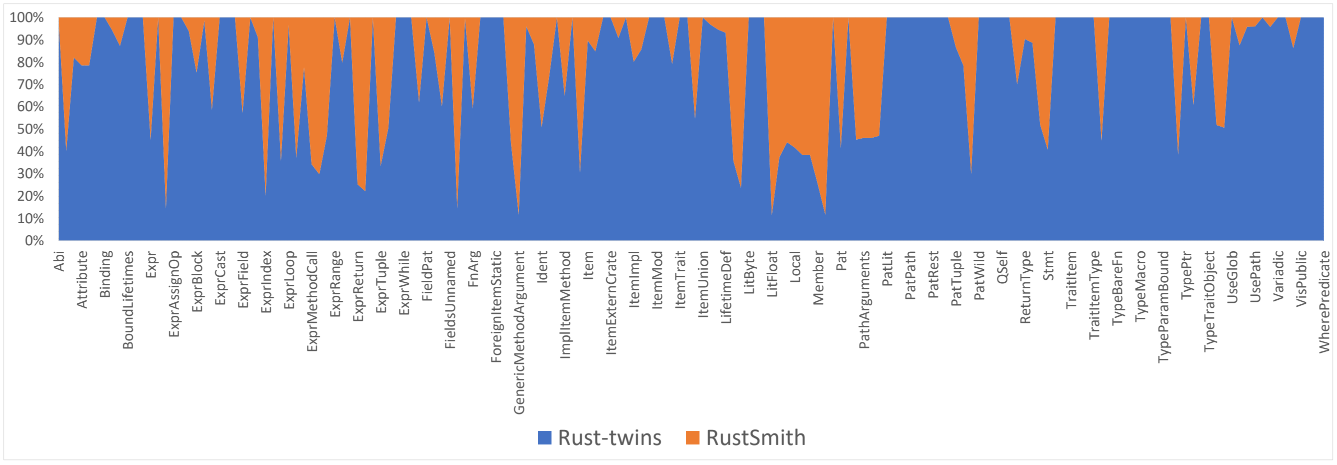


Figure 5: The AST type distribution of generated code of RUST-TWINS and RUSTSMITH.

RUSTSMITH. Consequently, RUST-TWINS can increase the number of generated programs by easily increasing the processes while maintaining good diversity.

Generation cost. To ensure the generated Rust's correctness, RUST-TWINS sends 22848 requests to LLMs, and successfully generates 6048 valid Rust programs for RUSTC within 24 hours with the cost of $48.1 for querying the GPT-4o in total.

Invalid programs. We sampled and analyzed 100 files from the mutation and macroize components, categorizing the root causes of invalid programs into five types as follows:

- **Unresolved Symbols**: The majority, 40 cases, had unresolved symbol errors, including the use of third-party crates (26) and unresolved symbols.
- **Syntax Errors**: 27 invalid programs had syntax errors, including incorrect attributes and inappropriate identifier modifiers.
- **Type Errors**: 17 cases involved type mismatches and incorrect generic function usage.
- **Safety Rules Violations**: Only 5 cases violated safety rules, such as returning variables with shorter lifetimes and lifetime type mismatches.
- **Other Errors**: The remaining 11 cases had other errors, such as network issues, environmental requirements, and pattern errors.

The analysis results show that the majority of generation errors for RUST-TWINS are due to unresolved symbols. GPT-4 tends to generate code that depends on popular crates not included in our compilation environment.

Answer to RQ1: RUST-TWINS can mutate and rewrite Rust programs with correct syntax and semantics by GPT-4o. Additionally, the generated programs exhibit greater diversity for compiler testing compared to grammar-based generators. The majority of generation errors are due to unresolved symbols.

4.1.2 *RQ2: How effective is rust-twins in testing RUSTC?*

Figure 6 shows the line coverage achieved in RUSTC by the baseline techniques RUSTSMITH (orange), LIBFUZZER (gray) and our tool RUST-TWINS (blue). We sampled and merged the coverage per tool hourly by `grcov` and `lcov`.

Overall coverage. Figure 6 indicates that LIBFUZZER achieves the lowest coverage, with only 24,884 lines covered after 24 hours of fuzzing. Additionally, its coverage saturates within the first hour. Despite LIBFUZZER efficiently producing inputs for the remaining time, the increase in coverage is minimal. Over the 24-hour fuzzing period, it covers only 700 more lines than it did in the first hour. As shown in Figure 6, the coverage of RUSTSMITH is similar to that of LIBFUZZER. It covers 100287 lines in the first hour and does not increase significantly during the remaining fuzzing period. Our tool, RUST-TWINS, achieves the highest overall coverage, with more than 198470 lines covered after 24 hours of fuzzing, which is 98183 more lines than RUSTSMITH. Due to its mutation strategies and seed prioritization, RUST-TWINS can discover more code regions and increase coverage compared to baseline techniques after the first hour of fuzzing.

Although LIBFUZZER does not generate any valid inputs for RUSTC, it still covers 24,884 lines. It is likely that RUSTC has many error analysis, recovering and reporting features, and the numerous invalid inputs achieve substantial code region of error-handling in the front-end of RUSTC. Since RUSTSMITH is a grammar-based generator without runtime information feedback, its rigid generation pattern fails to cover rare cases and deep code region of RUSTC. Despite its effective generation strategies for producing valid Rust programs, the limitation of poor diversity makes it difficult to continually increase coverage throughout the entire 24-hour fuzzing period.

Module coverage. In Figure 7, we selected the top 25 highest coverage modules and highlighted in orange those modules where RUST-TWINS achieves more than 35% higher coverage compared to RUSTSMITH. The IR lowering-related modules, `ast_lowering`, `hir_typeck`, `mir_build`, and `hir_analysis`, show significantly higher coverage, indicating that RUST-TWINS can generate better diversity programs and perform more efficient HIR and MIR lowering than RUSTSMITH.

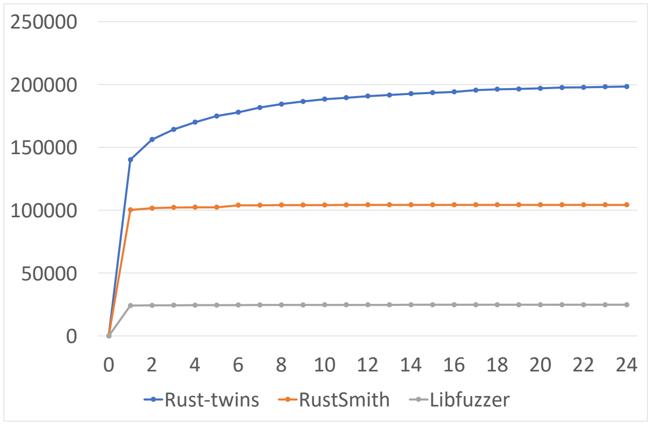


Figure 6: RUST-TWINS and baseline techniques line coverages over 24h of fuzzing.

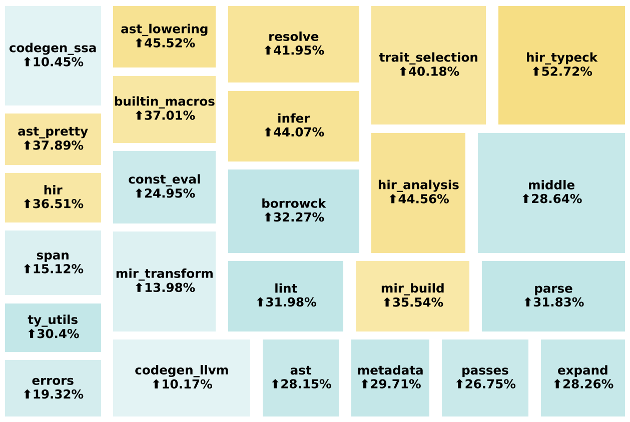


Figure 7: Module coverage comparison for RUST-TWINS and RUSTSMITH. (Larger areas indicate more total lines in those modules. The percentage number denotes how much higher coverage RUST-TWINS achieves compared to RUSTSMITH.)

Notably, RUST-TWINS can generate more built-in macros such as `println`, as evidenced by the 37.01% higher coverage in `builtin_macros`. In the middle-end, RUST-TWINS also has 26.44% higher coverage. However, in the back-end, RUST-TWINS achieves only 10.17% and 10.45% higher coverage in the `codegen_llvm` and `codegen_ssa` modules, respectively. We believe this is because many programming language features are eliminated before code generation, and the lower the IR, the fewer IR types there are. Additionally, due to the macroize component, RUST-TWINS outperforms RUSTSMITH in macro-related modules such as `expand` and `metadata`.

Answer to RQ2: RUST-TWINS achieves the highest overall coverage, demonstrating that its fuzzing strategies are significantly more effective than baseline techniques. Furthermore, RUST-TWINS explores more coverage across all modules in RUSTC. Specifically, RUST-TWINS outperforms RUSTSMITH in macro-related modules due to the macroize component.

4.1.3 RQ3: Root Cause of Differences and Crashes. We analyzed and reported some RUSTC differences to the Rust team using Github issue, 8 of which have been confirmed as RUSTC bugs. We summarize some typical differences and crashes as follows.

Different termination timing of expansion. Due to the differences in the expanded workflows of MBE and PCD, the timing of expansion termination also varies. RUST-TWINS generates multiple macro invocations sequentially within a single file. If any invocations result in type errors, the subsequent ones are no longer expanded in PCD. However, macro invocations in MBE are independent of each other, meaning a macro expansion error does not affect the other expansion processes. Therefore, MBE can expand more macros than PCD when errors occur. We do not consider this difference to be a bug.

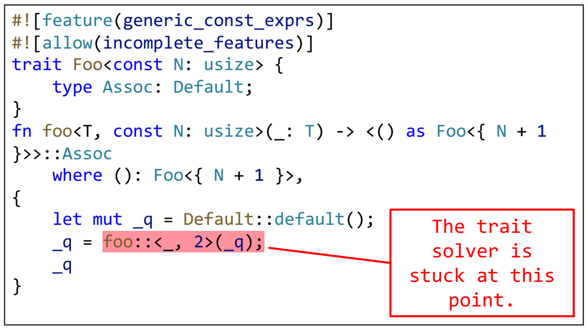
Different dump format. We compare the HIR dump results character by character. If differences are detected, RUST-TWINS indicates a potential error. However, there are some false positives because the dump formats of PCD and MBE are not identical. Specifically, comments, blanks, and newlines are not preserved in the HIR, and RUSTC attempts to guess their positions when dumping. Unfortunately, the guessing strategies for PCD and MBE are different.

Different scoping. The significant difference between PCD and MBE is variable scope. MBE is a hygienic macro with static variable scoping, while PCD has dynamic scoping. Before MBE expansion, RUSTC resolves the actual arguments at the call site. In contrast, for PCD, RUSTC first expands the macro and then resolves the actual arguments in the expanded results. Therefore, due to these differing variable scoping, MBE raises more symbol undefined errors than PCD. This difference does not indicate a bug in Rust.

False positive in equivalence assessment. Since LLMs are black box tools with a randomized mechanism, the code it generates is unstable and sometimes ignores the user's requirements. As a result, the two generated twin macros may exhibit different behaviors in detail. For example, in Rust, both `to_string` and `stringify!` are used to convert an object to a string. However, a regular type token such as &i32 is stringified as "& i32" by `to_string` and as "&i32" by `stringify!`. The simple default inputs cannot discern the slight semantic difference between these two functions. Fortunately, we found only 1 false positive in the equivalence assessment out of 229 differences in our experiments.

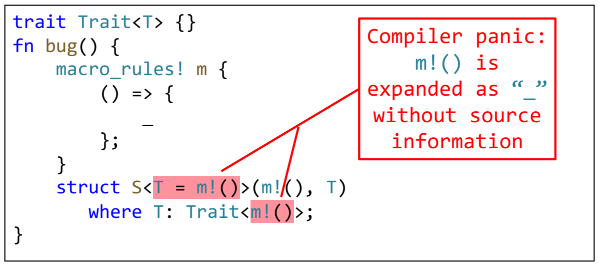
Bugs. In total, we have detected 12 bugs for RUSTC, and 8 of them are confirmed or fixed. The found bugs are mainly due to the following root causes.

- **Incorrect implementations (3).** We classify two types of implementations that might cause incorrect expansion. First, the expansion rules directly impact the correctness of the macro. Second, the whole translation correctness of RUSTC affects the expansion result since PCD is an actual function and must be executed during the expansion stage. More specifically, the mechanism of PCD involves the entire compilation process from parsing to code generation. PCD will be compiled to a binary and executed to produce a token stream as the macro expansion result. Thus,



```
#![feature(generic_const_exprs)]
#![allow(incomplete_features)]
trait Foo<const N: usize> {
    type Assoc: Default;
}
fn foo<T, const N: usize>(_: T) -> <() as Foo<{ N + 1
}>>::Assoc
    where (): Foo<{ N + 1 }>,
{
    let mut _q = Default::default();
    _q = foo::<_, 2>(_q);
    _q
}
```

Figure 8: The detected infinite compilation bug.



```
trait Trait<T> {}
fn bug() {
    macro_rules! m {
        () => {
            _
        };
    }
    struct S<T = m!()>(m!(), T)
        where T: Trait<m!()>;
}
```

Compiler panic: m!() is expanded as "_" without source information

Figure 9: The detected panic of assertion.

any general setting bugs in the compilation would result in incorrect expansion results of PCD, causing the difference between the results of PCD and MBE. One example of this type of bug is shown in Figure 4 as a motivating example.

- **Unterminated compilation (2).** Rust provides compile-time calculation features to support its zero-cost abstractions. However, implementing these features poses a significant challenge for the compiler. We discovered that RUSTC might not terminate when compiling a program with large types, such as `&'static [i32; 32768]`. Additionally, the integer computation in generics can cause infinite compilation. For example [1], as shown in Figure 8, the generic type parameters are constant expressions `N` and `N + 1`, which lead to infinite compilation when the generic type inference exists.
- **Panic of debug assertion (3).** In RUSTC, each HIR node has a span that records source code information, and this span should be non-empty during compilation after the parsing stage. We found that some edge cases can trigger the non-empty checking assertion in RUSTC's debug mode. These cases were combined and generated by LLMs from RUSTC's original unit test cases. As shown in Figure 9, the macro `m` is expanded as the type placeholder "_", which turns the type `T = m`()! into `T = _`. However, the placeholder lacks any source code information, causing a compiler panic. [2]

[1]https://github.com/rust-lang/rust/issues/126106
[2]https://github.com/rust-lang/rust/issues/116502

Answer to RQ3: RUST-TWINS detects 8 confirmed and fixed bugs. A difference might be caused by different termination timing of expansion, dump format, variable scoping, generated code and expansion implementation. A crash might be caused by the unterminated compile-time calculation and debug assertion of the parser.

5 RELATED WORK

Rust compiler testing. Due to the aforementioned challenges, few existing works focus on RUSTC testing. The state-of-the-art differential testing tool, RUSTSMITH[32], is the first randomized generator of Rust programs suitable for differential compiler testing. Another work[13] addresses typechecker defects in RUSTC through a CLP-based code generation approach.

Differential testing. In recent years, many differential testing approaches have been proposed for compiler testing to resolve the test-oracle problem. These approaches can be roughly categorized into intra-compiler and inter-compiler differential testing. In intra-compiler differential testing, the same compilers with different configurations or versions are grouped as an oracle. Kitaura et al.[18] use multiple older versions of a compiler to detect performance bugs in the latest version. Some works[8, 35] attempt to couple JIT-enabled and JIT-disabled compilers with different configurations and trigger seeds to identify optimization bugs.

In inter-compiler differential testing, multiple compilers are taken as an oracle. CSMITH[38] collects different versions of LLVM and GCC and compares the compilation results produced by these compilers. GRAYC[14] introduces a novel post-processing tool that transforms programs to produce a single output, which can be easily compared across different compilers. Beyond C/C++, there are various implementations of JVM and WebAssembly [27]. Numerous JVM fuzzers [11, 12, 41] generate class files to trigger differences between JVMs. Similarly, some fuzzers [17, 42] generate WebAssembly instructions to detect discrepancies among multiple WebAssembly engines.

Metamorphic testing. The test-oracle problem can also be addressed by metamorphic testing [10]. In short, the core of this approach is to construct metamorphic relations, which can be used to generate equivalent input data for the programs under test. In compiler metamorphic testing, researchers generally devise code transformers to generate programs that are seemingly different but actually equivalent [25].

Many transformers have been proposed in previous works. Orion, Athena, and Hermes [19, 20, 33] attempt to modify seeds in a semantic-preserving manner by deleting and inserting statements in dead code. Besides static approaches, other researchers derive metamorphic relations through runtime information [15, 36]

Furthermore, there are stronger code transformers that can be performed without considering the context of the transformed program. Mettoc [34] proposed equivalent expression templates, equivalent statement manipulations, and equivalent control flow transformations that can always generate equivalent programs for any seeds. MetaFuzz [40] is designed for a specific programming language and implements six identically equivalent mutation rules based on new programming language features.

6 DISCUSSION

Metamorphic Testing vs Differential Testing. The key difference between metamorphic testing and differential testing is the existence of multiple implementations in the former and metamorphic relations in the latter. Consequently, there are two reasons why we claim that RUST-TWINS belongs to differential testing. First, we generate identical macros as two different implementations with the same behavior and use the actual arguments of expansion as inputs to test their correctness. In contrast, in metamorphic testing, the generated seeds are used as inputs directly for the programs under test. Second, there are no well-defined metamorphic relations to generate equivalent seeds in our approach since the generation task is done by the LLMs.

Limitations of practice. As a testing technique, its throughput is one of the significant factors in triggering potential bugs. However, RUST-TWINS, driven by LLMs, suffers from the computing power required for inferencing and cannot generate large volumes of input data for testing tasks. Furthermore, while the LLMs can mutate and rewrite Rust programs to help produce valid Rust programs, the stillborn rate remains unsatisfactory. We defer the augmentation of RUST-TWINS to improve its generation success rate through prompt engineering and by fine-tuning a Rust-specific LLM.

7 THREATS TO VALIDITY

Internal. The internal threats come from three aspects. First, we only use the lifetime complexity while there exist many other general code complexity metrics such as MCCABE'S CYCLOMATIC COMPLEXITY [26] and HALSTEAD COMPLEXITY MEASURES [16]. We leave the metrics augmentation in the future. Second, for the mutation component, we use only four Rust-specific mutators tailored to the language's unique features. There are other features we have not yet considered, such as trait objects and higher-rank trait bounds. The third threat is the lack of experiments using different LLMs instead of GPT-4o to mutate and rewrite in RUST-TWINS.

External. One of the external threats is the hallucination phenomenon of large language models (LLMs). Although we designed the assessment component to reduce the impact of hallucination, it can still generate inconsistent macros in the macroize component. Furthermore, we identify differences by comparing the dumped expanded macros in HIR, which can result in more false positives if the dumped format changes. For macro systems, there is a risk that our approach may not detect widespread defects in RUSTC due to the instability of the PCD expansion mechanism.

8 CONCLUSION AND FUTURE WORK

We have presented the design of our LLM-driven differential testing approach for RUSTC and detailed its implementation through RUST-TWINS. Our evaluation demonstrates that, under strict static checking of RUSTC, RUST-TWINS can generate a sufficient number of Rust programs to achieve the highest coverage for RUSTC compared to other baseline techniques. Additionally, RUST-TWINS has detected and reported numerous differences and crashes to the RUSTC developers.

In the future, we plan to fine-tune a Rust-specific mutation LLM to address the majority generation errors (unresolved symbol and syntax errors), rather than using the universal LLM GPT-4o, to better support Rust fuzzing and differential testing tasks. Furthermore, we aim to apply RUST-TWINS to other software that accepts Rust programs as input, such as Rust Analyzer, to enhance the code quality of the entire Rust community.

9 ACKNOWLEDGMENT

Wenzhang Yang and Yinxing Xue were supported by Chinese National Natural Science Foundation (Grant#: 61972373).

REFERENCES

[1] 2023. announcing-windows-11-insider-preview-build-25905. https://blogs.windows.com/windows-insider/2023/07/12/announcing-windows-11-insider-preview-build-25905/ Accessed: 2023-07-21.

[2] 2023. The crate syn. https://crates.io/crates/syn Accessed: 2023-03-26.

[3] 2023. Introduction of Rust macros. https://doc.rust-lang.org/book/ch19-06-macros.html Accessed: 2024-06-04.

[4] 2023. Libfuzzer. https://www.llvm.org/docs/LibFuzzer.html Accessed: 2023-03-26.

[5] 2023. Stack Overflow 2023 developer survey. https://survey.stackoverflow.co/2023/ Accessed: 2024-06-04.

[6] Matt Asay. 2020. Why AWS loves Rust, and how we'd like to help. https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/

[7] Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th international conference on compiler construction.* 82–92.

[8] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* 351–364.

[9] 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.

[10] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

[11] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 1257–1268.

[12] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 85–99.

[13] 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).* IEEE, 482–493.

[14] Karine Even-Mendoza, Arindam Sharma, Alastair F Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 1219–1231.

[15] Shikai Guo, He Jiang, Zhihao Xu, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Rong Chen. 2022. Detecting Simulink compiler bugs via controllable zombie blocks mutation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1061–1072.

[16] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc.

[17] Gilang Hamidy. 2020. Differential fuzzing the webassembly. (2020).

[18] Kota Kitaura and Nagisa Ishiura. 2018. Random testing of compilers' performance based on mixed static and dynamic code comparison. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018).* Association for Computing Machinery, New York, NY, USA, 38–44. https://doi.org/10.1145/3278186.3278192

[19] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[20] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *Acm Sigplan Notices* 50, 10 (2015), 386–399.

[21] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).* Shanghai, China.

[22] Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *Proceedings of the 28th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, Volume 3. 238–251.

[23] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 533–544.

[24] Yifei Lu, Weidong Hou, Minxue Pan, Xuandong Li, and Zhendong Su. 2024. Understanding and Finding Java Decompiler Bugs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1380–1406.

[25] Haoyang Ma. 2023. A Survey of Modern Compiler Fuzzing. *arXiv preprint arXiv:2306.06884* (2023).

[26] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[27] Mozilla. 2024. WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly

[28] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP).* 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

[29] Redox. 2023. Redox - Your Next(Gen) OS. https://www.redox-os.org/

[30] Rust. 2024. HIR. https://rustc-dev-guide.rust-lang.org/hir.html

[31] Servo. 2023. Servo. https://servo.org/

[32] 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.* 1483–1486.

[33] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications.* 849–863.

[34] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An automatic testing approach for compiler based on metamorphic testing technique. In *2010 Asia Pacific Software Engineering Conference.* IEEE, 270–279.

[35] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. {FuzzJIT}:{Oracle-Enhanced} Fuzzing for {JavaScript} Engine {JIT} Compiler. In *32nd USENIX Security Symposium (USENIX Security 23).* 1865–1882.

[36] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28.

[37] Wenzhang Yang, Linhai Song, and Yinxing Xue. 2024. Rust-lancet: Automated Ownership-Rule-Violation Fixing with Behavior Preservation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* 1–13.

[38] 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.

[39] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).

[40] Yingquan Zhao, Junjie Chen, Ruifeng Fu, Haojie Ye, and Zan Wang. 2023. Testing the compiler for a new-born programming language: An industrial case study (experience paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 551–563.

[41] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering.* 1133–1144.

[42] Shiyao Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. 2023. WADIFF: A Differential Testing Framework for WebAssembly Runtimes. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 939–950.

[43] 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.