Rust-lancet: Automated Ownership-Rule-Violation Fixing with Behavior Preservation

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

Linhai Song
songlh@ist.psu.edu
Pennsylvania State University
Pennsylvania, USA

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

ABSTRACT

As a relatively new programming language, Rust is designed to provide both memory safety and runtime performance. To achieve this goal, Rust conducts rigorous static checks against its safety rules during compilation, effectively eliminating memory safety issues that plague C/C++ programs. Although useful, the safety rules pose programming challenges to Rust programmers, since programmers can easily violate safety rules when coding in Rust, leading their code to be rejected by the Rust compiler, a fact underscored by a recent user study. There exists a desire to automate the process of fixing safety-rule violations to enhance Rust's programmability.

In this paper, we concentrate on Rust's ownership rules and develop RUST-LANCET to automatically fix their violations. We devise three strategies for altering code, each intended to modify a Rust program and make it pass Rust's compiler checks. Additionally, we introduce mental semantics to model the behaviors of Rust programs that cannot be compiled due to ownership-rule violations. We design an approach to verify whether modified programs preserve their original behaviors before patches are applied. We apply RUST-LANCET to 160 safety-rule violations from two sources, successfully fixing 102 violations under the optimal configuration — more than RUSTC and six LLM-based techniques. Notably, RUST-LANCET avoids generating any incorrect patches, a distinction from all other baseline techniques. We also verify the effectiveness of each fixing strategy and behavior preservation validation and affirm the rationale behind these components.

CCS CONCEPTS

• **Software and its engineering → General programming languages**; **Error handling and recovery**; **Software development techniques**; **Error handling and recovery**.

KEYWORDS

Rust, Program Repair, Compiler Error

*Also with Suzhou Institute for Advanced Study University of Science and Technology of China.
†Yinxing Xue is the corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639103

ACM Reference Format:
Wenzhang Yang, Linhai Song, and Yinxing Xue. 2024. Rust-lancet: Automated Ownership-Rule-Violation Fixing with Behavior Preservation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639103

1 INTRODUCTION

As a statically-typed young programming language, Rust has rapidly gained popularity [28]. The language's core innovation lies in its safety mechanism and the associated safety rules. Rust enforces these rules to check and eliminate memory bugs during compilation, allowing it to maintain the execution speed of its compiled executable programs comparable to those written in C/C++. Rust's combination of safety and performance has led to its adoption in building numerous safety-critical low-level software [1, 15, 33, 47, 50].

Unfortunately, Rust has a steep learning curve [10, 62], and its safety rules pose programming challenges to its programmers [66]. Rust programmers can easily write code that violates Rust's safety rules, leading to compiler errors, given that Rust's safety rules are distinctive, and both the related grammar and semantics differ significantly from those of traditional programming languages. To make things worse, the Rust compiler typically fails to provide sufficient information for programmers to comprehend the compiler errors resulting from safety-rule violations [66]. Furthermore, safety-rule violations hinder the application of many automated techniques to Rust, such as code generation for Rust [54], translating C/C++ programs to Rust [19, 22, 63], and fuzzing Rust libraries [24]. Consequently, there is a need to automatically patch compiler errors stemming from safety-rule violations, with the goal of enhancing Rust's programmability and facilitating the utilization of other Rust-related techniques.

Automatically fixing safety-rule violations in Rust programs presents two primary challenges, underscoring the inadequacy of existing automated program repair (APR) techniques for this task. The first challenge is how to design suitable code-change strategies to handle safety-rule violations. Rust's safety rules are exceptionally unique, and conventional APR techniques tailored for C/C++ and Java programs are unaware of Rust's safety rules [11, 13, 16, 20, 27, 34, 41, 49, 52, 59]. Consequently, their fixing templates and machine learning models are ill-equipped to address safety-rule violations specific to Rust. The second challenge lies in determining how to validate whether patches maintain the original behaviors of programs. Since the programs, prior to applying the patches, cannot be compiled, we are unable to follow Rust's semantics to infer their behaviors or execute the compiled programs to observe their behaviors like existing techniques [12, 30, 32, 32, 43, 56–58, 60, 61].

In this paper, we build RUST-LANCET to automatically fix compiler errors stemming from ownership-rule violations in Rust programs while preserving the original program behaviors. RUST-LANCET specifically focuses on ownership-rule violations, given that ownership is a key concept in Rust's safety mechanism, and such violations (*e.g.*, using a moved value, having two mutable references to the same object coexisting) are commonly encountered by Rust programmers in their daily practices [66]. RUST-LANCET takes a Rust program with a compiler error as input and attempts to fix the error through multiple rounds. The process continues until RUST-LANCET either discovers a patch or reaches a configured round limit. In each round, RUST-LANCET systematically applies its three fixing strategies, and validates whether the modified program can be successfully compiled and whether the fixing preserves the program's behaviors for each strategy.

To tackle the first challenge, we design the three fixing strategies by taking into account Rust's safety mechanism comprehensively: owner loop (OL), reordering (REO), and lattice-based weaken (WKN). The OL strategy targets errors where an older owner is used after its ownership has been moved to a new owner. It accomplishes this by substituting the use of the old owner variable with the new one. REO relocates an instruction to a preceding location. It can relocate the use of a value before where it is moved and resolve the issue of two mutable references to the same object coexisting. WKN substitutes an operation with a high likelihood of causing a violation with another one carrying a lower likelihood, determined by a constructed lattice. Each of these strategies contributes distinct value in enhancing the likelihood of passing Rust's compiler checks for Rust programs.

To address the second challenge, we devise mental semantics to model the intentions of programmers when coding in Rust. Mental semantics ease the constraints imposed by Rust's safety rules, as their purpose is to model the behaviors of programs that have safety-rule violations and cannot be compiled. To perform the detailed validation for a patch, RUST-LANCET first computes the condition under which the program's behaviors remain unchanged after the patch is applied. Subsequently, RUST-LANCET instruments an assertion and other relevant code to examine this condition. In the end, RUST-LANCET performs symbolic execution to validate whether the inserted assertion holds true for all program paths, with reference to mental semantics. If successful, RUST-LANCET concludes that the patch preserves the program's behaviors.

To evaluate RUST-LANCET, we collect 160 safety-rule violations from two sources [9, 66] and compare RUST-LANCET with RUSTC and six other large language model (LLM)-based techniques. In total, RUST-LANCET patches 102 violations under the optimal setting, addressing 57 more violations than RUSTC and 18 more violations than the best LLM-based baseline technique. Moreover, RUST-LANCET does not generate any wrong patches (false positives), whereas all baseline techniques exhibit false positives, with the potential for such occurrences reaching 115. We meticulously examine RUST-LANCET's results by systematically disabling each of its fixing strategies and behavior preservation validation. We observe that the fully-featured RUST-LANCET patches most violations, and disabling behavior preservation validation leads to two false positives, underscoring the rationality of RUST-LANCET's design. In a user study involving three experts, we observe that RUST-LANCET patches a number of violations comparable to Rust experts, but with a significantly shorter processing time.

```
1 let a = "1".to_string();
2 let new_a = a;
3 println!("{}", a);
```

(a) Intra-scope movement

```
1 fn foo(a: String){
2     loop {
3         foo(a);}}
```

(b) Inter-scope movement

```
1 let p1 = &mut a;
2 let p2 = &a;
3 println!("{}", *p1);
4 println!("{}", *p2);
```

(c) End at distinct locations

```
1 let p1 = &mut a;
2 let p2 = &a;
3 println!("{}{}",
4     *p1, *p2);
```

(d) End at the same location

```
1 let p1 = &mut a;
2 let p2 = &a;
3 println!("{}", *p2);
4 println!("{}", *p1);
```

(e) Lifetime cover

```
1 // hellothere.x is a reference
2 match hellothere.x {
3     box E::Bar(x) =>
4       println!("{}",x),
5     _ => {}}
```

(f) Move through a reference

Figure 1: Error categories

Overall, we make the following three contributions:

- We devise three fixing strategies tailored to address ownership-rule violations in Rust programs by taking careful consideration of the relevant safety rules.
- We introduce mental semantics to model behaviors of Rust programs with ownership-rule violations. Additionally, we develop an approach to validate whether patches addressing ownership-rule violations preserve program behaviors, utilizing mental semantics as a reference.
- We build RUST-LANCET by integrating the fixing strategies and behavior preservation validation. We conduct thorough experiments to evaluate RUST-LANCET and confirm its fixing capability, accuracy, and advancement over the baseline techniques.

All our code and experimental data can be found at https://sites.google.com/view/rust-lancet/index.

2 BACKGROUND

This section gives the background of this paper, including Rust's unique safety mechanism, programming challenges caused by Rust's safety checks, and the problem scope of this paper.

2.1 Rust's Safety Mechanism

Rust constructs its safety mechanism based on two fundamental concepts: *ownership* and *lifetime*. In essence, Rust enforces that each value has one and only one owner variable (*i.e.*, the name-binding variable of the value), and the value is dropped (freed) when its owner's lifetime ends, such as at the end of the owner variable lexical scope. However, this basic safety rule can be overly restrictive, particularly when implementing low-level systems software. Thus, Rust provides some extensions to the basic rule to enable better programming flexibility.

First, Rust allows moving the ownership of a value to a different owner or a different scope but it prohibits the former owner from being used anymore after the move. For example, in Figure 1a, variable a is the owner of the string after line 1, and the ownership

is then moved to `new_a` at line 2. Thus, Rust prohibits the use of `a` at line 3. Similarly, when function `foo` recursively calls itself at line 3 in Figure 1b, the ownership of `a` is moved to the callee (a different scope). As this move happens in a loop and has already occurred in the first iteration, Rust disallows the use of `a` in following iterations.

Second, Rust provides the ability to temporarily borrow the ownership of a value through references. Those references can be mutable, supporting both read and write accesses, or immutable, only allowing read accesses. Rust enforces the rules that a mutable reference cannot coexist with other references to the same variable and that the ownership cannot be moved through a reference. For example, in Figure 1c, variable `a` is borrowed with mutable reference `p1` at line 1, and then it is borrowed again with immutable reference `p2` at line 2. Moreover, `p2` is dereferenced at line 4. Thus, Rust prohibits the use of `p1` at line 3. Another reference error is illustrated in Figure 1f. `hellothere.x` is moved to the `match` block at line 2. However, `hellothere.x` is a reference and thus Rust disallows this move.

All the above rules essentially disallow having alias and mutability at the same time, and they can prevent many severe memory safety issues and thread safety issues. Moreover, all the rules are checked by the Rust compiler, so that Rust ensures its compiled executable programs to be as efficient as C/C++ programs.

2.2 Rust's Programming Challenges

Regrettably, Rust's safety mechanism and corresponding safety rules pose unique programming challenges for Rust programmers. It is easy for them to write code that violates Rust's safety rules and is subsequently rejected by the Rust compiler. Recent research confirms this observation through an empirical study on Rust-related Stack Overflow questions and a survey of real-world Rust programmers [66]. The researchers report that the second most common reason for Rust programmers to ask questions on Stack Overflow is to figure out why their code violates Rust's safety rules. Furthermore, the researchers also note that a safety rule may be more difficult to follow in certain programming contexts, and that the error messages produced by the Rust compiler — the most immediate feedback for safety-rule violations — may not provide enough information for programmers to comprehend the violations. Thus, it is critical to tackle the programming difficulties caused by Rust's safety checks and safety rules.

Our solution to these programming challenges is to develop a tool that can automatically fix safety-rule violations. Upon applying our tool, the modified Rust file can pass the Rust compiler's stringent safety checks, while preserving its original programming semantics. We envision our tool can be integrated into the CI/CD process. It will resolve safety-rule violations automatically and greatly improve Rust's programmability.

2.3 Program Scope

The authors of the recent research paper also develop a taxonomy for the root causes of Rust's safety-rule violations [66]. They categorize the violations into those caused by violating ownership rules and those due to complex lifetime computation, since ownership and lifetime are the two most important concepts of Rust's safety mechanism. They further separate each category into several subcategories. For example, ownership-rule violations are divided into move-rule violations and borrow-rule violations.

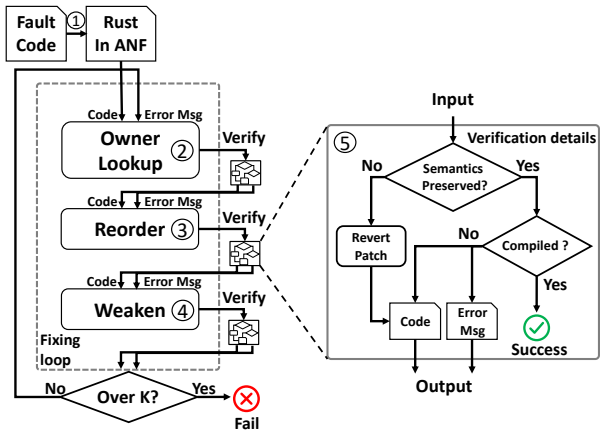

Figure 2: Workflow of RUST-LANCET.

In our work, we employ this taxonomy with a specific emphasis on addressing ownership-rule violations, deferring the handling of those arising from complex lifetime computation for future research. Specifically, we resolve three types of safety-rule violations: using an already moved object (*e.g.*, Figure 1a, Figure 1b), borrowing an object while it has already been mutably borrowed (*e.g.*, Figure 1c, Figure 1d, Figure 1e), and moving an object through a reference (*e.g.*, Figure 1f). Section 3.2 will provide a more comprehensive explanation of these three error types and their formal definitions.

3 PROPOSED APPROACH

In this section, we commence by introducing the workflow of RUST-LANCET, accompanied by a motivating example. Subsequently, we delineate the three types of safety-rule violations that we aim to address. Following that, we delve into the repair process. Finally, we detail how we validate that a patch preserves the original semantics.

3.1 Overview

Workflow. Figure 2 shows the workflow of RUST-LANCET. It takes a program with safety-rule violations as input. It either produces a modified program that is free of violations and preserves the semantics of the original program (indicated by "Success" in Figure 2), or it reports that it cannot patch the program after several rounds of attempts (indicated by "Failure" in Figure 2).

In summary, RUST-LANCET consists of five key steps (denoted by ①-⑤ in Figure 2). In Step ①, RUST-LANCET converts the input program into administrative normal forms (ANF) [48] to facilitate the application of the fixing strategies. Steps ②-④ correspond to the three fixing strategies, respectively. They are executed one by one in the main loop. After applying a strategy to modify the input program, RUST-LANCET proceeds to Step ⑤ to verify whether the modified program satisfies two criteria: it preserves the original semantics, and it passes the Rust compiler's checks. If so, RUST-LANCET identifies a patch and concludes the fixing process. If not, RUST-LANCET attempts the next fixing strategy. Particularly, if the modified program retains the original semantics, the modification

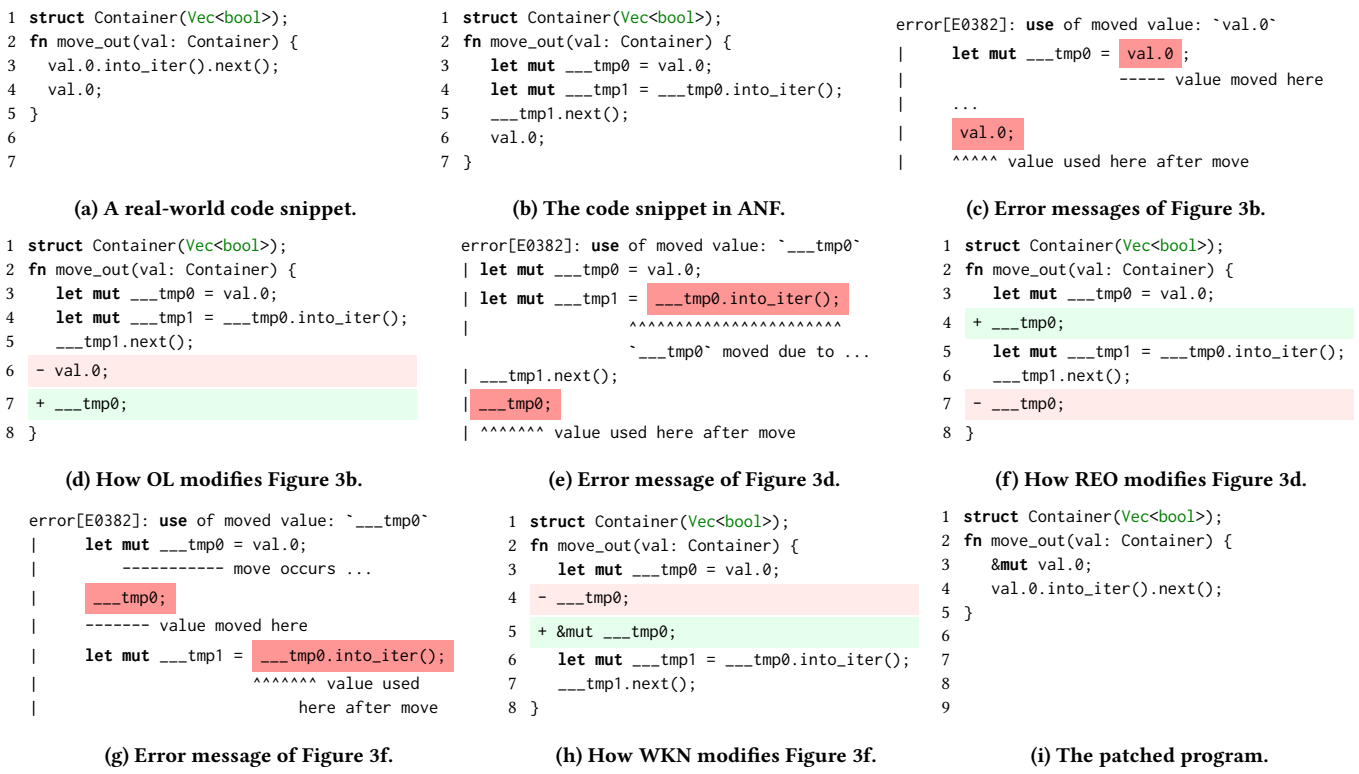

Figure 3: How RUST-LANCET repairs a real-world code snippet. Figures (c), (e), and (g) show the compiler error messages of the original program and modified programs at intermediate steps. Violations are highlighted in red in those figures.

conducted by RUST-LANCET according to the strategy is retained for subsequent patching steps. Otherwise, the program is rolled back to the version before applying the strategy. After each iteration of the main loop, RUST-LANCET checks whether the iteration number is larger than a configurable number K. If so, RUST-LANCET abandons the fixing and reports that it cannot patch the input program.

The three patching strategies are Owner Lookup (**OL**), which substitutes a variable whose ownership is moved with a new variable that takes the moved ownership, Reordering (**REO**), which swaps the order of two statements, and Weakening (**WKN**), which attempts to alter the type of an operation. We will provide detailed explanations of these three strategies in Section 3.3.

Motivating Example. Figure 3 illustrates the step-by-step process by which RUST-LANCET fixes a safety-rule violation, along with the corresponding intermediate results. Figure 3a shows the original code snippet with the violation. The code snippet comes from the official Rust compiler test suites. At line 3, `val.0` is moved into the function `into_iter()`. Consequently, the uses of `val.0` at line 4 violates the safety rule that a moved value cannot be used. As shown in Figure 3b, RUST-LANCET initiates the process by transforming the input program into ANF (Step ① in Figure 2). Subsequently, it employs RUSTC to compile the program in ANF, leading to the error message in Figure 3c. RUST-LANCET performs fault location and extracts statements that violate the safety rule based on the error message for subsequent steps. The modification of the program resulting from the application of OL is shown in Figure 3d (Step ②). As demonstrated in Figure 3e, the altered program does not successfully pass Rust's compiler checks (Step ⑤). Consequently, RUST-LANCET continue the repairing process. Since the modification carried out by OL does not alter the program's semantics, it is retained for the following steps. Similarly, REO also fails to patch the program, but the modification it introduces is preserved (Step ③). Ultimately, WKN patches the program (Step ④). To enhance the readability of the patch, RUST-LANCET transforms the modified program in ANF in Figure 3h back to its original form in Figure 3i.

3.2 Error Categories

We categorize the compiler errors addressed by RUST-LANCET into three distinct types:

Intra-scope Movement. Compiler errors falling under this category violate the safety rule that the old owner cannot be accessed after a move operation. Additionally, for such errors, the new owner variable exists within the same program scope as where the old owner is accessed. One such example is shown in Figure 1a. The variable `a` declared at line 1 is the owner of the string. The ownership is moved to `new_a` at line 2. Consequently, the use of `a` at line 3 violates the safety rule. Importantly, the new owner `new_a` is within the same scope as the use of `a` at line 3.

Inter-scope Movement. Errors of this type violate the same safety rule as the preceding category. However, in this case, the value is moved to a distinct program scope, presenting a significant challenge for repair. For instance, consider Figure 1b, where string `a`

is moved to function `foo` (a different scope) at line 3 within a loop. Consequently, the use of `a` in the subsequent loop iteration violates the safety rule.

Borrowing. RUST-LANCET fixes two types of borrow-rule violations. The first involves having both a mutable reference and a reference to the same object simultaneously, while the second pertains to moving ownership through a reference. To handle the former type, we introduce several symbolic representations, with a crucial focus on the lifetime tuple $(l^\circ, l^\bullet)$ and related positional relations denoted by $=$ and $\prec$.

Definition 1 (Lifetime Tuple). *A lifetime tuple signifies the living range of a value, represented as $(l^\circ, l^\bullet)$. It denotes the starting and termination locations within the control flow, where l° signifies the starting location, and $l^\bullet$ denotes the termination location.*

Definition 2 (Lifetime Relation). *The binary relation $l_1^\circ \prec l_2^\circ$ signifies that location l_1° precedes location l_2° within the control flow, establishing an order between the two locations. The notation $l_1^\circ = l_2^\circ$ indicates that the two locations l_1° and l_2° are the same within the control flow.*

When two references coexist, their lifetime tuples, denoted as $(l_1^\circ, l_1^\bullet)$ and $(l_2^\circ, l_2^\bullet)$, exhibit one of the following three patterns:

- $l_1^\circ \prec l_2^\circ \prec l_1^\bullet \prec l_2^\bullet$. Both the starting and termination locations of the first lifetime tuple precede those of the second lifetime, as illustrated in Figure 1c.
- $l_1^\circ \prec l_2^\circ$, $l_1^\bullet = l_2^\bullet$. The termination locations of the two lifetime tuples are the same. Figure 1d shows one such example, where both the lifetimes of `p1` and `p2` end at line 4.
- $l_1^\circ \prec l_2^\circ$, $l_2^\bullet \prec l_1^\bullet$. The lifetime scope of the first reference encompasses the lifetime scope of the second reference. Figure 1e provides an example of this pattern, where the lifetime scope of `p1` covers that of `p2`. As `p1` is a mutable reference, the Rust compiler generates a compilation error, indicating that a mutable reference is not allowed to coexist with another reference to the same object. Nevertheless, Rust provides a reborrowing mechanism [7] to afford programmers more flexibility. By utilizing this mechanism, the compilation error of Figure 1e can be resolved by replacing line 2 with "`let p2 = &*(p1);`".

Errors arising from moving a value through a reference can manifest in any location where a reference is dereferenced. These errors become more challenging to comprehend when the dereference operation occurs implicitly. For instance, in Figure 1f, `hellothere.x` is a reference used as the condition of a **`match`** expression at line 2. Notably, the **`ref`** keyword is absent in the pattern of the arm at line 3. Consequently, the reference is dereferenced implicitly, and the value is moved to **`box`** `E::Bar(x)`, resulting in a violation of the rule that prohibits moving a value through a reference.

3.3 Fixing Loop

As depicted in Figure 2, RUST-LANCET iterates through the fixing loop until it either discovers a patch or reaches a pre-configured threshold K for loop iterations. In each iteration, RUST-LANCET applies the three fixing strategies one by one. For each strategy, RUST-LANCET analyzes the error message generated by RUSTC to pinpoint the AST nodes violating the safety rule. Subsequently, it modifies the program in accordance with the strategy. Applying a strategy is one single attempt to patch the program. If its code modification maintains the original semantics, it is preserved for subsequent steps. A complex error may involve the application of multiple strategies, as illustrated by the error in Figure 3.

Table 1: How fixing strategies and behavior preservation instrumentation modify the input program. *(BP is short for behavior preservation, and $\omega(exp)$ is an operation weaker than exp.)*

ID	Patch	BP Instrumentation

OL	`let foo = exp;`
	`...`
	`- let bar = exp;`
	`+ let bar = foo;`
		`let foo = exp;`
		`...`
		`+ assert(exp == foo);`
		`let bar = exp;`

REO	`+ let foo = exp2;`
	`let bar = exp1;`
	`... // no I/O stmts`
	`- let foo = exp2;`
		`+ let c1 = exp2;`
		`let bar = exp1;`
		`... // no I/O stmts`
		`+ let c2 = exp2;`
		`+ assert(c1 == c2);`
		`let foo = exp2;`

WKN	`- let foo = exp;` `+ let foo = ω(exp);`	Clone Checking

Fault Localization. When compiling a file, if the additional compilation parameter "--error-format=json" is provided to RUSTC, RUSTC generates a JSON file containing all error messages encountered during compilation. RUST-LANCET analyzes the JSON file for fault location. Each object in the file corresponds to a compiler error. The `code` field of an object represents the code of the violated rule. One `label` field of an object corresponds to one source-code line involved in the error. It also describes why the line is involved. Each `label` has a `line_start` field denoting the line number of the source-code line. We use the line-number information to locate the faulty AST nodes. Although this approach may not be perfect, our experience demonstrates its accuracy is sufficient for constructing RUST-LANCET.

For example, the JSON file for Figure 3c contains only one object, as there is only one compiler error. There are two source-code lines involved: one represents where `val.0` is moved (line 3 in Figure 3b), and the other represents where `val.0` is used after the move (line 6 in Figure 3b). Thus, the object contains two `label` fields for the two lines, and the `line_start` fields of the two `label` fields are 3 and 6, respectively.

Owner Lookup (OL). This strategy aims to resolve errors caused by intra-scope movement. As the new owner is in the same scope as the use of the old owner, the strategy replaces the use of the old owner with the use of the new owner. Specifically, RUST-LANCET checks if the input compiler error is "use of moved value". If not, RUST-LANCET stops applying the strategy and does not modify the input program. Otherwise, RUST-LANCET further extracts the expression accessing the old owner (*e.g.*, "`let bar = exp`" in Table 1). Since the input program has already been converted to ANF, the

Table 2: How different operations change the numbers of values and entities. ***(An entity could be an owner or a reference, and "/": not applicable.)***

Value \ Entity	−1	−0	+1
−1	move	/	/
−0	/	& mut	&
+1	/	/	clone

expression cannot be a chain call (*e.g.*, line 3 in Figure 3a). RUST-LANCET examines whether the expression falls into one of the following four types: accessing a variable[1], accessing an object's field, dereferencing a reference, and conducting the "?" operation. If not, RUST-LANCET gives up this strategy. Otherwise, RUST-LANCET identifies the new owner by analyzing the assignments preceding the use (*e.g.*, "`let foo = exp`" in Table 1) and replaces the use of the old owner with the new owner.

For instance, in Figure 3c, RUST-LANCET identifies `val.0` at line 6 in Figure 3b as the expression using the moved value. It then recognizes the new owner as `__tmp` declared at line 3 in Figure 3b. Consequently, it substitutes `val.0` at line 6 with `__tmp`, as depicted in Figure 3d.

Reordering (REO). As illustrated in Table 1, the REO strategy initially identifies a pair of instructions that violate a safety rule. Subsequently, it repositions the latter instruction immediately before the former instruction. The rationale behind REO is to potentially relocate the instruction that uses a moved value before the move operation. Additionally, it is likely to move the final use of a reference ahead of the declaration of another reference to the same object, effectively eliminating the overlap in the lifetimes of the two references, since Rust's NLL mechanism terminates the lifetime of a reference at its last use, such as changing $l_1^\circ \prec l_2^\circ \prec l_1^\bullet \prec l_2^\bullet$ to $l_1^\circ \prec l_1^\bullet \prec l_2^\circ \prec l_2^\bullet$.

To determine the two instructions for reordering in an input program, REO examines the JSON file containing error messages generated by RUSTC during compilation. If the object corresponding to a compiler error has only one `label` field, REO refrains from patching the error, as there is only one instruction involved. In all other cases, REO depends on a set of heuristic rules to decide which two `label` fields should be considered. These rules take into account their descriptions, explaining why they are involved, and the type of the compiler error.

Prior to relocating an instruction, REO conducts an additional check for any I/O operations (*e.g.*, `println!`) between the new and old locations. If such operations exist, the movement could potentially affect the side effects of the I/O operations. As a result, REO conservatively abandons the patching process.

For example, REO identifies the instructions at lines 4 and 7 in Figure 3d for reordering through an analysis of the error message presented in Figure 3e. It relocates line 7 just before line 4 in Figure 3d, as depicted by the patch in Figure 3f.

Lattice-based Weaken (WKN). Rust provides multiple types of operations to modify entities (*e.g.*, owner, reference) through which a value is accessed. Some of these operations are more prone to violating safety rules than others, and we consider these to have a stronger capability. The WKN strategy transforms operations with stronger capabilities into those with weaker capabilities, thereby reducing the risk of safety rule violations. Specifically, we consider four types of operations: move, mutable borrow, immutable borrow, and clone. As shown in Table 2, we define their capabilities based on two aspects. The first aspect is whether an operation creates one more entity to access a value. The second aspect is whether an operation creates one more value. A move operation can move a value to a different scope, reducing the number of entities and values of the current scope by one. Furthermore, neither a mutable borrow nor an immutable borrow creates a new value. However, a mutable borrow exclusively borrows the value, thus not increasing the number of entities to access the value, while multiple immutable references are allowed in Rust, and an immutable borrow creates one more immutable reference. In contrast, cloning an object creates one more value and also an owner of the new value. In summary, the total order of the four types of operations is determined as: $clone \subset \& \subset \&mut \subset move$.

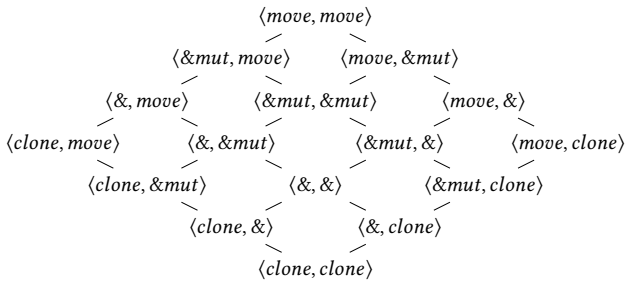

Figure 4: The lattice for two operations.

At times, operations that modify entities are implicitly carried out through standard library calls. We notice a pattern in the standard library where functions with a similar functionality but different ways of accessing one of their parameters are named as follows: `{fn}` moves the parameter, `{fn}_mut` mutably borrows the parameter, and `{fn}_ref` immutably borrows the parameter. Therefore, we can establish the order of these functions by analyzing their names.

Given a compiler error, WKN first identifies the operations contributing to the violation by examining the `label` fields. Subsequently, WKN constructs a lattice where the number of fields in each lattice element corresponds to the number of involved operations. For instance, the lattice for cases involving two operations is depicted in Figure 4. WKN locates the lattice element corresponding to the operations causing the error and then weakens their capabilities by traversing the lattice from top to bottom and from left to right. After weakening operations based on a lattice element, WKN verifies whether the semantics are preserved and whether the modified program remains free of compiler errors. If either of these conditions is not met, WKN proceeds to the next lattice element. If the semantics are unchanged, the code modification is retained.

For instance, RUST-LANCET identifies the two operations causing the compiler error in Figure 3g as "`__tmp0`" and "`__tmp0.into_iter()`".

[1]The access may be performed through a plain path if the variable is in a different module

$$\frac{loc(S, v) \;=\; \ell_v}{S \rhd o \longrightarrow S \rhd \ell_w} \qquad \frac{S_2 \;=\; write(S_1, o_1, o_2)}{S_1 \rhd o_1 \;=\; o_2 \longrightarrow S_2 \rhd \epsilon}$$

(a) Move **(b) Assignment**

Figure 5: The move and assignment reduction rules in mental semantics. ***(S: the memory state before and after an operation if the operation does not alter the state; S_1 and S_2: the memory state before and after an operation; o, o_1, and o_2: the owners of values v, v_1, and v_2.)***

Both are move operations. According to the lattice in Figure 4, RUSTLANCET transforms the first move into a mutable borrow, as shown in Figure 3h.

3.4 Behavior Preservation

Fixing a compiler error in Rust doesn't simply end with passing the compiler's checks. To ensure that the modified program behaves the same as the original, a comparison of their behaviors is essential. However, this task is significantly challenging as the Rust compiler cannot compile the original program, preventing us from generating an executable and comparing its behavior with that of the modified program.

This section addresses the challenge through three steps. First, we define mental semantics to capture the intentions of programmers when writing Rust code. Second, we instrument assertions and some other related code into the original programs based on the applied fixing strategy. The conditions of these assertions ensure that the program semantics remain unchanged after the strategy is applied. Lastly, we perform a specialized symbolic execution by referencing the mental semantics to verify whether the inserted assertions are satisfied across all potential program executions. If successful, the program semantics are preserved.

3.4.1 Mental Semantics. We introduce mental semantics to represent the behaviors of Rust programs with compiler errors. The mental semantics avoid dropping any values, permit one value to have multiple owners, and allow multiple mutable references to the same object to coexist. Our rationale is that if Rust did not enforce these rules, programs with compiler errors could be compiled, run, and exhibit the behaviors desired by programmers.

We employ symbols and helper functions from [46] to formally define mental semantics. Due to space constraints, we refrain from presenting the complete set of reduction rules in this paper[2]. Figure 5 illustrates two rule examples. Figure 5a demonstrates that when the ownership of v is moved from o, o still holds the ownership. To achieve this, after the move, the memory location of o still retains the address of v, allowing all accesses to v to be performed through o. Figure 5b shows that when o_2 is moved to o_1, we do not drop the value owned by o_1 before the move. All references to o_1 before the move still point to the old value v_1.

3.4.2 BP Instrumentation. As shown by Table 1, we devise different instrumentation methods for different fixing strategies.

The OL strategy involves replacing the old owner with the new owner. Therefore, an assertion is instrumented before the code line where the replacement is performed to ensure that the value of the new owner is equal to the old owner.

REO relocates an instruction from its current location to a preceding one. Since there is no use of the instruction's value between the current location and the preceding location, we insert an assertion immediately after the current location, which ensures that the evaluation result of the instruction at the preceding location is equal to the evaluation result at its current location. (*e.g.*, lines marked by "+" in Figure 6)

Since WKN patches changes entities to access values, they typically do not alter program behaviors according to the mental semantics, except in cases where object cloning occurs. When an object is cloned, subsequent accesses to the cloned version and the original object now target different objects. The original program behaviors are preserved only when the values of the two objects are the same. To validate this, we adopt a conservative approach by examining whether there is any update to the cloned version or the original version after the clone operation. If such updates exist, we consider the patch to modify the original program semantics.

```
1  struct C(Vec<bool>);
2  fn move_out(val: C) {
3      // σ = {val -> C(vec)}
4      let mut ___tmp0 = val.0;
5      // σ = {val -> C(vec), ___tmp0 -> vec}
6  +   let c1 = ___tmp0;
7      // σ = {val -> C(vec), ___tmp0 -> vec, c1 -> vec}
8      let mut ___tmp1 = ___tmp0.into_iter();
9      // σ = {val -> C(vec), ___tmp0 -> vec, c1 -> vec,
10     // ___tmp1 -> into_iter(vec)}
11     ___tmp1.next();
12     // σ = {val -> C(vec), ___tmp0 -> vec, c1 -> vec,
13     // ___tmp1 -> into_iter(vec)}
14 +   let c2 = ___tmp0;
15     // σ = {val -> C(vec), ___tmp0 -> vec, c1 -> vec,
16     // ___tmp1 -> into_iter(vec), c2 -> vec}
17 +   assert(c1 == c2);
18     // true
19 }
```

Figure 6: BP instrumentation for Figure 3d and the symbolic execution results. ***("+" denotes lines instrumented to validate behavior preservation, and comments following a code line show the symbolic execution results after executing the line.)***

3.4.3 Validation. We conduct partial symbolic execution to verify whether the instrumented assertion holds true for an OL or REO patch. The symbolic execution initiates from the function's entry point where the patch is applied and halts at the insertion point of the assertion. All values defined outside the function are treated as symbolic variables, and the execution states are updated according to the mental semantics after each instruction is executed. If the instrumented assertion proves true for all execution paths, we can affirm that the patch maintains the original program semantics.

Figure 6 shows an example. The initial state of symbolic execution is "`σ = val -> C(vec)`" at line 3. The execution state is updated by referring to the mental semantics. For example, line 6 moves the ownership of `___tmp0` to `c1`, but `___tmp0` can still be used to access `vec`, since mental semantics allows for multiple owners. Similarly, `c2` also becomes the owner of `vec` after line 14. Thus, "`c1==c2`" is true

[2]All reduction rules can be found at our project website [5].

at line 17, affirming that the REO patch preserves the program's behaviors.

4 EVALUATION

We implement RUST-LANCET using RUSTC version `1.67.0-nightly`. RUST-LANCET takes a Rust source code file with a compiler error as input and outputs the patch for the error or reports that it cannot fix the error. RUST-LANCET performs its analysis and code transformation on the Abstract Syntax Tree (AST) of the file. To achieve this, it utilizes the syn crate [3] and the quote crate [2] to convert a stream of Rust source-code tokens into an AST and dump an AST into source code. Our experiments are designed primarily to address the following four questions:

- **Effectiveness**: How effective is RUST-LANCET in fixing Rust's safety-rule violations?
- **Necessity**: How does each component of RUST-LANCET contribute to its capability?
- **Advancement**: Is RUST-LANCET better than state-of-the-art techniques?
- **Comparison with Experts**: How does RUST-LANCET perform compared with Rust experts?

We conduct all of our experiments on a MacBook Pro machine equipped with a 2GHz Quad-Core Intel Core i5 CPU, 16GB RAM and kernel version macOS 13.1.

4.1 Experimental Settings

Benchmarks. We collect a total of 160 safety-rule violations from two sources. First, we select 36 testing files with ownership-rule violations from the official test suite of RUSTC (RustcTS) [9]. Since one file may contain multiple violations, there are a total of 111 violations from these files. To simplify our experiments, we split each testing file into multiple files to ensure that each split file only contains one violation. Second, we choose 49 violations from the dataset constructed by Zhu *et al.* [66]. These violations are in distinct source-code files. Among them, 42 are ownership-rule violations, and they are within the problem scope of RUST-LANCET (Zhu). The remaining seven are intentionally picked to explore how RUST-LANCET behaves when handling compiler errors beyond its problem scope (OoS). Among them, four are caused by type mismatches, two stem from mutating an immutable variable, and one is caused by a missed type name.

Techniques. As shown by Table 3, we set the maximum iteration number of RUST-LANCET's fixing loop (denoted as "K" in Figure 2) to one and four, and evaluate RUST-LANCET under these two settings.

We choose seven baseline techniques. One of them is the Rust compiler (RUSTC), since it sometimes offers direct suggestions for modifying the program to pass compiler checks. The remaining six techniques are developed by harnessing the capabilities of large language models (LLMs) provided OpenAI [6]. Those techniques vary in three aspects. First, three of them utilize the GPT-3.5 model, while the other three employ the GPT-4 model. Second, for four techniques, we use scripts to automatically extract modified code suggested by the LLM from each LLM response, test whether the code still contains any compiler error, and request the LLM to fix again if necessary (marked by "K" in Table 3). For the other, we manually read the response and follow the response to modify the code (marked by "H" in Table 3). Third, among the automated techniques, we further distinguish them based on the number of fixing attempts. Two techniques attempt only once ("K1" in Table 3), while the other two attempt four times ("K4" in Table 3). For all the techniques relying on LLMs, we set the "temperature" parameter to 0.5 to regulate the randomness of the LLMs. We employ a consistent prompt template, as illustrated below. For each request instantiation, the placeholders src and err in the template are substituted with the specific Rust code and the compiler error, respectively.

```
Try to fix my rust compilation error with behavior
preservation, and give me a completely fixed version.
Don't explain the code, just generate the rust code block itself.
[ [The code in your answer is still uncompilable.]
   Here is the code: {src}
   Here is the compilation error message: {err} ]*
```

Table 3: Experimental Results. ***(K: automated techniques, H: techniques need human involvement, and the number following K indicates the maximum number of attempts.)***

	RustcTS (111)		Zhu (42)	
	TP	**FP**	**TP**	**FP**
RUSTC	40 (36.0%)	8 (7.2%)	5 (11.9%)	7 (16.7%)
GPT-3.5-K1	26 (23.4%)	85 (76.6%)	12 (28.6%)	30 (71.4%)
GPT-3.5-K4	40 (36.0%)	71 (64.0%)	17 (40.5%)	25 (59.5%)
GPT-3.5-H	45 (40.5%)	66 (59.5%)	16 (38.1%)	26 (61.9%)
GPT-4-K1	42 (37.8%)	69 (62.2%)	17 (40.5%)	25 (59.5%)
GPT-4-K4	51 (45.9%)	60 (54.1%)	22 (52.4%)	20 (47.6%)
GPT-4-H	61 (55.0%)	50 (45.0%)	**23 (54.8%)**	19 (45.2%)
RUST-LANCET-K1	76 (68.5%)	0 (0%)	13 (31.0%)	0 (0%)
RUST-LANCET-K4	**89 (80.2%)**	**0 (0%)**	13 (31.0%)	**0 (0%)**

Metrics. We follow a two-step process to assess whether a rule violation is fixed. At the beginning, we employ RUSTC to compile the modified Rust file and check whether it does not have any compiler error. If so, we proceed to the subsequent step. In the second phase, a Rust expert meticulously reviews the modified file to confirm its adherence to the original semantics. The Rust expert makes the decision entirely on his expertise in Rust, without consulting any techniques described in Section 3. When a technique successfully corrects a violation, we count a true positive (TP) for the technique. If a technique generates a wrong patch (either rejected by the compiler or losing the original semantics), we count a false positive (FP) for the technique.

To quantify the size of a patch, we begin by formatting both the original and modified source code using the command-line tool rustfmt [8]. Subsequently, we use the command-line tool diff [4] to determine the number of modified code lines with bash command "`diff -y -- suppress-common-lines foo.rs bar.rs | wc -l`."

To assess the execution time of a tool, we execute the tool *ten* times and report the average execution time.

4.2 Experimental Results

4.2.1 Effectiveness. As indicated in Table 3, when the maximum iteration number of the fixing loop is set to four (RUST-LANCET-K4), RUST-LANCET successfully addresses 89 out of 111 violations (80.2%) in RustcTS and 13 out of 42 violations in Zhu (31.0%). Notably, RUST-LANCET does not produce any false positives while patching

Table 4: Component contributions of RUST-LANCET-K4. ***(RL is short for RUST-LANCET-K4, w/o is short for without, x_y denotes x fixed violations and y false positives, and "-" represents both fixed violations and false positives are zero.)***

	w/o OL	w/o REO	w/o WKN	w/o BP	RL
RustcTS	86_0	46_0	59_0	89_0	89_0
Zhu	11_0	10_0	8_0	13_2	13_0
OoS	-	-	-	-	-
Total	97_0	56_0	67_0	102_2	102_0

violations in the two datasets, showcasing its accuracy. When RUST-LANCET executes the fixing loop only once (RUST-LANCET-K1), the number of patched violations decreases to 76 for RustcTS, affirming that increasing the iteration number of the fixing loop enhances RUST-LANCET's fixing capability. The number of patched violations remains unchanged for Zhu. Similarly, RUST-LANCET-K1 does not report any false positives.

While scrutinizing the seven bugs in OoS, RUST-LANCET does not discover any patches upon reaching the configured maximum iteration number of the fixing loop. Despite being unable to address violations beyond its problem scope, RUST-LANCET does not generate incorrect patches when analyzing these violations.

Patch Size. Out of the 102 violations addressed by RUST-LANCET-K4, the average patch size is 2.25 lines of code (LOC). It is widely recognized that smaller patches are easier for programmers to review and are more likely to be accepted. Consequently, the patches generated by RUST-LANCET are likely to meet acceptance from programmers.

Execution Time. With RUST-LANCET-K1, the analysis of the 160 violations of the three datasets takes 365.74 seconds. On average, RUST-LANCET-K1 spends 2.28 seconds analyzing each violation. When increasing the maximum fixing-loop iteration number to four, the overall analysis time rises to 850.45 seconds, and the average analysis time per violation increases to 5.31 seconds. Among all the components, WKN is the most time-consuming as it attempts all cells in the constructed lattice. The worst-case complexity is $O(N^n)$, where N denotes the number of operation types[3], and n represents the number of operations involved in a compiler error. For all errors in our experiments, n is less than 5.

Answer to Effectiveness: RUST-LANCET successfully patches the majority of rule violations when the maximum fixing loop iteration is set to four. Additionally, the generated patches are small, increasing the likelihood of acceptance by programmers. The efficient execution time of RUST-LANCET makes it a viable candidate for integration into the CI/CD process.

4.2.2 Necessity. To discern the contribution of each component in RUST-LANCET — namely, OL, REO, WKN, and BP validation — we run RUST-LANCET-K4 on all three datasets with each component disabled individually. The experimental results are shown in Table 4.

Fixing Strategies. In terms of overall effectiveness, REO stands out as the most impactful fixing strategy. Its absence results in a significant reduction in RUST-LANCET-K4's fixing capability, with only 56 violations addressed, leaving 46 unpatched compared to the complete version of RUST-LANCET-K4. Conversely, OL emerges as the least effective fixing strategy. When disabled, RUST-LANCET-K4 addresses only five fewer violations. WKN demonstrates effectiveness on the Zhu dataset. When deactivated, RUST-LANCET-K4 misses five violations, the highest number among all three strategies.

[3] N is four for our current implementation

```
1    let mut a = Vec::new();
2  + println!("{:?}", a);
3    let mut foo = Foo::new(move |v| {
4      for i in v { a.push(i);}});
5    foo.fun(1);
6  - println!("{:?}", a);
```

Figure 7: A patch example rejected by BP validation.

BP Validation. If behavior preservation validation is disabled, RUST-LANCET still fixes the same number of bugs but introduces two incorrect patches (*i.e.*, false positives). These two false positives, when BP validation is activated, would be rejected by symbolic execution and clone checking, respectively. Figure 7 illustrates one such example. The compiler error arises from using the vector `a` at line 6 after `a` is moved to the closure at line 4. To resolve this error, REO attempts to relocate line 6 to line 2 to ensure the use of `a` precedes the move. However, the closure is invoked at line 5, and a number is pushed into `a`. The vector `a` at line 6 contains one more element than `a` at line 2. Symbolic execution can precisely capture this distinction and reject the patch.

The programs used in our evaluation are relatively simple, primarily involving operations such as moving, borrowing, and cloning values, with infrequent value updates. Consequently, we believe the importance of the BP validation component is undervalued, given that the patches generated by the three strategies are unlikely to alter the original semantics. However, in real-world scenarios, programs are more prone to modifying values [23], and thus BP validation will be more impactful.

Answer to Necessity: Each fixing strategy contributes to enhancing the patching capability of RUST-LANCET. Among them, REO stands out as the most effective. Additionally, behavior preservation validation proves to be successful in effectively eliminating false positives.

4.2.3 Advancement. As shown by Table 3, both RUST-LANCET-K1 and RUST-LANCET-K4 successfully patch more violations than the baseline techniques on RustcTS, whereas GPT-4-H proves to be the most effective technique on Zhu. When considering the two datasets together, RUST-LANCET-K4 fixes the highest number of violations — 102 in total. This is 18 more than the best baseline technique, GPT-4-H. Furthermore, all baseline techniques exhibit false positives, with the count reaching as high as 115 for GPT-3.5-K1 (85 from RustcTS and 30 from Zhu). In contrast, neither RUST-LANCET-K1 nor RUST-LANCET-K4 generates any false positives. Overall, RUST-LANCET outperforms the baseline techniques in both fixing capability and fixing accuracy.

Figure 8 illustrates a compiler error that can only be fixed by RUST-LANCET. Two mutable references, `a` and `b`, are created at lines 2 and 3, both mutably borrowing `x.0`. Additionally, `a` is used at line

```
1    let mut x = (1, 2);
2    let a = &mut x.0;
3  - let b = &mut x.0;
4  + let _b = &mut x.1;      GPT-3.5
5    a.use_ref();
6  + let b = &mut x.0;       Rust-lancet
7  + a.use_mut();            GPT-4
```

Figure 8: The patches for RustcTS-298.

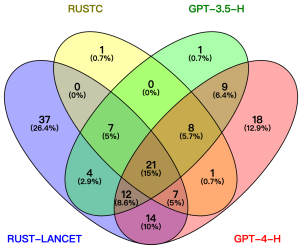

Figure 9: Violation coverage.

```
1    struct Foo<A> { f: A }
2    fn touch<A>(_a: &A) {}
3    let x = "hi".to_string();
4  - let _y = Foo { f:x };
5  + let _y = Foo { f:x.clone() };  Expert
6    touch(&x);
7  + let _y = Foo { f:x };      Rust-lancet
```

Figure 10: The patches for RustcTS-246.

5, extending its lifetime from line 2 to line 5. Consequently, the two mutable references to `x.0` coexist at line 3, triggering the compiler error. To address the error, all techniques delete code line 3, but they add different code lines. GPT-3.5-H adds line 4, where a different value (`x.1`) is mutably borrowed. Although this patches the compiler error, it alters the program semantics. GPT-4-H inserts a method call at line 7, but does not create `b` anymore. It also changes the program semantics. In contrast, RUST-LANCET relocates the declaration of `b` after the use of `a`. After this relocation, the lifetime of `a` no longer overlaps with that of `b`, effectively patching the violation and also preserving the semantics.

GPT-4 fixes more violations than RUST-LANCET on the Zhu dataset. Specifically, GPT-4-H fixes 10 more violations than RUST-LANCET-K4. We attribute this to two factors. First, all violations in Zhu originate from Stack Overflow questions posted before September 2021, potentially contributing to the training data for GPT-4. Second, GPT-4 has internet access, and it is plausible that it retrieves Stack Overflow information when responding to our requests. To confirm our hypothesis, we examine the original Stack Overflow web pages corresponding to the 23 violations successfully patched by GPT-4-H. Among them, ten web pages contain the patched code, and the patched code precisely matches the patches generated by GPT-4-H. Additionally, for one violation, although the Stack Overflow page does not provide the patch, it described the main idea to fix the violation in plain text, and this description precisely aligns with the patch generated by GPT-4-H.

Violation coverage. Figure 9 illustrates the violation coverage when RUST-LANCET-K4, RUSTC, GPT-3.5-H, and GPT-4-H analyze the three datasets (RustcTS, Zhu, and OoS). There are 37 violations that can only be patched by RUST-LANCET-K4, demonstrating that RUST-LANCET complements the baseline techniques.

> **Answer to Advancement:** RUST-LANCET successfully patches the highest number of violations among all evaluated techniques, and there is a substantial number of violations that can only be addressed by RUST-LANCET. Thus, RUST-LANCET represents an advancement in the state-of-the-art for compiler-error fixing techniques for Rust.

4.2.4 Comparison with Rust Experts. We conduct a user study to compare patches written by Rust experts with those generated by RUST-LANCET.

Study Setup. We recruit study participants from two Rust-specific forums, requiring them to be at least 18 years old and have a minimum of two years of experience in Rust programming. We conduct interviews to verify their background and Rust expertise and ultimately recruit three senior software engineers. We compensate each participant with 500 RMB.

During the study, each participant is assigned the task of fixing all 160 compiler errors in the three datasets. We explicitly specify the use of RUSTC as the exclusive tool, prohibiting participants from utilizing other tools or referencing external resources. Participants are permitted to make multiple attempts for each compiler error, and we measure the time taken by each participant on an error from the moment they open the code file until they believe they have successfully fixed the error or decide to abandon the attempt.

Overall Results. As shown in Table 5, the three experts successfully fix 143, 153, and 102 compiler errors, respectively, with only two of them patching more errors than RUST-LANCET-K4. The remaining expert fixes the same number of violations as RUST-LANCET-K4. Additionally, on average, each expert spends 22,495 seconds analyzing all compiler errors, which is 25 times more than the time expended by RUST-LANCET-K4. RUST-LANCET demonstrates a fixing capability comparable to Rust experts and proves to be faster in analyzing errors and generating patches.

Patch Size. As shown by Table 5, the average sizes of the patches created by the three experts are 2.20, 2.33, and 2.13, respectively. These sizes are comparable to the average size of the patches generated by RUST-LANCET-K4. Moreover, sizes one and two are the most common patch sizes for the three experts and RUST-LANCET-K4. The size of a patch usually represents the readability of the patch. Thus, the readability of the patches created by RUST-LANCET-K4 is comparable to those created by Rust experts.

At times, despite RUST-LANCET-K4 generating a patch with a larger size than the patch created by a Rust expert for the same error, RUST-LANCET-K4's patch is better. An example of this is shown in Figure 10. In this case, string `x` is moved to struct `_y` at line 4. Consequently, the borrowing of `x` at line 6 is a use of `x` after the move. To fix the error, all the three Rust experts replace the move operation with a clone operation. The diff tool interprets this as a modification to a single source-code line, resulting in a patch size of one. On the other hand, RUST-LANCET-K4 relocates line 4 just after line 6, ensuring that the use precedes the move. The diff tool then considers the patch size to be two. Despite the larger size of RUST-LANCET-K4's patch, we consider it better as it avoids the need for an additional clone operation.

Table 5: Patch size distribution.

	Patch Size					Sum	Average
	1	2	3	4	>4		
Expert#1	55	54	15	8	11	143	2.20
Expert#2	50	69	12	9	13	153	2.33
Expert#3	58	21	10	5	8	102	2.13
RUST-LANCET-K4	26	64	5	1	6	102	2.25

Answer to Comparison with Experts: RUST-LANCET patches a comparable number of violations with similar patch sizes as Rust experts, but it is 25 times faster than the Rust experts.

5 DISCUSSION

Limitations. RUST-LANCET focuses on addressing violations of ownership-related rules. Another fundamental aspect of Rust's safety mechanism is lifetime, and numerous Rust compiler errors arise from breaching lifetime rules [66]. We defer the augmentation of RUST-LANCET to patch such errors for future work.

We fail to evaluate RUST-LANCET on real Rust projects for two reasons. First, extracting compiler errors from the commit histories of open-source Rust projects is difficult since developers typically ensure their commits can be compiled before merging them to the projects. Second, injecting errors into real Rust projects raises concerns about whether the errors injected by us can represent those made by real Rust programmers. Nevertheless, we are confident that RUST-LANCET can effectively work on large, real Rust projects. This confidence is grounded in the fact that both RUST-LANCET's patching strategies and behavior preservation validation are designed to operate within small program scopes (*e.g.*, a single function). None of them requires analyzing the entire input program, eliminating any scalability issues.

Threats to Validity. For symbolic execution in Section 3.4.3, we implement only a subset of Rust language features. Consequently, there is a risk that RUST-LANCET may incorrectly validate a patch's preservation of the program's original semantics, affecting internal validity. In our experiments, we manually review the responses from GPT-3.5 and GPT-4. The assessment is subjective, and our decisions may be influenced by our expertise, posing a potential impact on external validity.

6 RELATED WORK

Automated Program Repair (APR). In recent years, many methodologies for Automated Program Repair (APR) have emerged, broadly falling into non-learning-based and learning-based categories.

In non-learning-based APR, approaches vary from manual templates [26, 44] to automatic pattern mining [18, 29, 35]. TBar [36], a state-of-the-art method, consolidates 35 fix templates, outperforming counterparts. Semantic-based techniques like CSNIPPEX [53] and HoBuFF [38] offer solutions for import declarations and perform build fixing using dataflow analysis, respectively. HeteroGen [64], HireBuild [21], and RULF [24] address compilation errors. HeteroGen uses pattern-oriented edits, HireBuild calculates log similarity, and RULF, a Rust library fuzzer, circumvents ownership-rule violations by adding tags. In learning-based APR, techniques such as MACER and TEGCER [14, 17] classify compilation errors or compute the similarity with associated patches. Rete [45] addresses the challenge of learning program namespaces, and TENURE [40] combines template-based and Neural Machine Translation (NMT) methods, representing a significant evolution in APR methodologies. Jiang et al. [25] compare code language models, while Tare [65] proposes a type-aware neural program repair. TransRepair [34] uses program context and error messages for repair.

Patch Correctness. Testing-driven patch correction, ensuring behavior correction by passing all tests [26], has overfitting limitations [31] and could cause more serious harm [51] than original fixed bugs. After that, researchers focus on patch precision [35, 39, 55]. Techniques ensure semantic equivalence, such as Alive2 [37] encoding LLVM IR Semantics in SMT to verify the correctness of optimizations. Invalidator [32] uses semantic and syntactic reasoning via program invariants to automatically assess the correctness of the patch. Verifix [12] aligns the incorrect program and the reference program, then performs MaxSMT to find a minimal repair with behavioral equivalence. Relational Hoare logic [42], a variant of Hoare logic, verifies the equivalence between programs. These approaches contribute to robust patch correctness validation.

7 CONCLUSION

In this paper, we introduce RUST-LANCET, an automated technique designed to patch ownership-rule violations in Rust programs. To construct RUST-LANCET, we design three effective fixing strategies and an approach to assess whether a patch maintains the program's original semantics. We evaluate RUST-LANCET with 160 violations. The experimental results show that RUST-LANCET can effectively patch ownership-rule violations with small patch sizes and short analysis time, and RUST-LANCET outperforms RUSTC and other LLM-based baseline techniques in terms of patching capability and accuracy. In the future, we plan to extend RUST-LANCET to patch other types of Rust compiler errors.

8 ACKNOWLEDGMENT

Wenzhang Yang and Yinxing Xue were supported by Chinese National Natural Science Foundation (Grant#: 61972373) and CAS Pioneer Hundred Talents Program. Linhai Song was supported by NSF grants CNS-1955965 and CCF-2145394.

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 quote. https://crates.io/crates/quote Accessed: 2023-03-26.
[3] 2023. The crate syn. https://crates.io/crates/syn Accessed: 2023-03-26.
[4] 2023. diff. https://www.gnu.org/software/diffutils/ Accessed: 2023-03-26.
[5] 2023. Mental Semantic Details. https://sites.google.com/view/rust-lancet/index Accessed: 2023-07-31.
[6] 2023. OpenAI. https://openai.com/ Accessed: 2023-03-26.
[7] 2023. Reborrow in Rust. https://github.com/rust-lang/reference/issues/788 Accessed: 2023-03-26.
[8] 2023. rustfmt. https://github.com/rust-lang/rustfmt Accessed: 2023-03-26.
[9] 2024. Rustc. https://github.com/rust-lang/rust/tree/master/tests Accessed: 2024-01-22.
[10] Parastoo Abtahi and Griffin Dietz. 2020. Learning Rust: How Experienced Programmers Leverage Resources to Learn a New Programming Language. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems (CHI EA '20)*. Honolulu, HI.

[11] Toufique Ahmed, Noah Rose Ledesma, and Premkumar T. Devanbu. 2021. SYNFIX: Automatically Fixing Syntax Errors using Compiler Diagnostics. *CoRR* abs/2104.14671 (2021). arXiv:2104.14671 https://arxiv.org/abs/2104.14671

[12] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified Repair of Programming Assignments. *ACM Transactions on Software Engineering and Methodology.* 31, 4 (2022), 74:1–74:31. https://doi.org/10.1145/3510418

[13] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE'2018 SEET), Gothenburg, Sweden.* https://doi.org/10.1145/3183377.3183383

[14] Umair Z. Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. 2019. Targeted Example Generation for Compilation Errors. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE'2019), San Diego, CA, USA.* https://doi.org/10.1109/ASE.2019.00039

[15] 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/

[16] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *CoRR* abs/1603.06129 (2016). arXiv:1603.06129 http://arxiv.org/abs/1603.06129

[17] Darshak Chhatbar, Umair Z. Ahmed, and Purushottam Kar. 2020. MACER: A Modular Framework for Accelerated Compilation Error Repair. In *Proceedings of the 21st International Conference on Artificial Intelligence in Education (AIED'2020), Ifrane, Morocco (Lecture Notes in Computer Science, Vol. 12163).* https://doi.org/10.1007/978-3-030-52237-7_9

[18] Reudismam Rolim de Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *Proceedings of the 35th Brazilian Symposium on Software Engineering (SBES'2021), Joinville, Santa Catarina, Brazil.* https://doi.org/10.1145/3474624.3474650

[19] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to safer Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485498

[20] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'2017), San Francisco, California, USA.* http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[21] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'2018), Gothenburg, Sweden.* https://doi.org/10.1145/3180155.3180181

[22] Jaemin Hong. 2023. Improving Automatic C-to-Rust Translation with Static Analysis. In *Proceedings of 45th IEEE/ACM International Conference on Software Engineering (ICSE'2023 Companion), Melbourne, Australia.* https://doi.org/10.1109/ICSE-Companion58688.2023.00074

[23] Wei Huang, Ana L. Milanova, Werner Dietl, and Michael D. Ernst. 2012. Reim & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2012),Tucson, AZ, USA.* https://doi.org/10.1145/2384616.2384680

[24] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. In *Proceedings of 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'2021), Melbourne, Australia.* https://doi.org/10.1109/ASE51524.2021.9678813

[25] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'2023), Melbourne, Australia.* https://doi.org/10.1109/ICSE48619.2023.00125

[26] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 36th IEEE/ACM International Conference on Software Engineering (ICSE'14). Hyderabad, India.* https://doi.org/10.1109/ICSE.2013.6606626

[27] Youngjae Kim, Seungheon Han, Askar Yeltayuly Khamit, and Jooyong Yi. 2023. Automated Program Repair from Fuzzing Perspective. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'2023), Seattle, WA, USA.* https://doi.org/10.1145/3597926.3598101

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

[29] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[30] Xuan-Bach Dinh Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina S. Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'2019), Montreal, QC, Canada.* https://doi.org/10.1109/ICSE.2019.00064

[31] Xuan-Bach Dinh Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'2018), Gothenburg, Sweden.* https://doi.org/10.1145/3180155.3182536

[32] Thanh Le-Cong, Duc-Minh Luong, Xuan-Bach Dinh Le, David Lo, Nhat-Hoa Tran, Bui Quang Huy, and Quyet-Thang Huynh. 2023. Invalidator: Automated Patch Correctness Assessment Via Semantic and Syntactic Reasoning. *IEEE Trans. Software Eng.* 49, 6 (2023), 3411–3429. https://doi.org/10.1109/TSE.2023.3255177

[33] 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.

[34] Xueyang Li, Shangqing Liu, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, and Yang Liu. 2022. TransRepair: Context-aware Program Repair for Compilation Errors. In *Proceedings of 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022), Rochester, MI, USA.* https://doi.org/10.1145/3551349.3560422

[35] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'2019), Hangzhou, China.* https://doi.org/10.1109/SANER.2019.8667970

[36] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'2019), Beijing, China.* https://doi.org/10.1145/3293882.3330577

[37] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'2021), Virtual Event, Canada.* https://doi.org/10.1145/3453483.3454030

[38] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'2019), Beijing, China.* https://doi.org/10.1145/3293882.3330578

[39] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 15:1–15:37. https://doi.org/10.1145/3241980

[40] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'2023), Melbourne, Australia.* https://doi.org/10.1109/ICSE48619.2023.00127

[41] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: learning to repair compilation errors. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'2019), Tallinn, Estonia.* https://doi.org/10.1145/3338906.3340455

[42] David A. Naumann. 2020. Thirty-Seven Years of Relational Hoare Logic: Remarks on Its Principles and History. In *Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods (ISLAFM'2020), Rhodes, Greece.* https://doi.org/10.1007/978-3-030-61470-6_7

[43] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E. Hassan, Xuan-Bach Dinh Le, and David Lo. 2023. Multi-Granularity Detector for Vulnerability Fixes. *CoRR* abs/2305.13884 (2023). https://doi.org/10.48550/arXiv.2305.13884 arXiv:2305.13884

[44] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering.* 14, 3 (2009), 286–315. https://doi.org/10.1007/s10664-008-9077-5

[45] Nikhil Parasaram, Earl T. Barr, and Sergey Mechtaev. 2023. Rete: Learning Namespace Representation for Program Repair. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'2023), Melbourne, Australia.* https://doi.org/10.1109/ICSE48619.2023.00112

[46] David J Pearce. 2021. A lightweight formalism for reference lifetimes and borrowing in Rust. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 1 (2021), 1–73.

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

[48] Amr Sabry and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation* 6 (1993), 289–360.

[49] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'2018), Campobasso, Italy.* https://doi.org/10.1109/SANER.2018.8330219

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

[51] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'2015),*

Bergamo, Italy. https://doi.org/10.1145/2786805.2786825

[52] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the Importance of Building High-quality Training Datasets for Neural Code Search. In *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE'2022), Pittsburgh, PA, USA*. https://doi.org/10.1145/3510003.3510160

[53] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: automated synthesis of compilable code snippets from Q&A sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'2016), Saarbrücken, Germany*. https://doi.org/10.1145/2931037.2931058

[54] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI EA'2022), New Orleans, LA, USA*. https://doi.org/10.1145/3491101.3519665

[55] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'2018), Gothenburg, Sweden*. https://doi.org/10.1145/3180155.3180233

[56] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'2017), Santa Barbara, CA, USA*. https://doi.org/10.1145/3092703.3092718

[57] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'2018), Gothenburg, Sweden*. https://doi.org/10.1145/3180155.3180182

[58] Ke Xu, Yao Xiao, Zhaoheng Zheng, Kaijie Cai, and Ram Nevatia. 2023. PatchZero: Defending against Adversarial Patch Attacks by Detecting and Zeroing the Patch. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV'2023), Waikoloa, HI, USA*. https://doi.org/10.1109/WACV56688.2023.00461

[59] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. In *Proceedings of the 37th International Conference on Machine Learning (ICML'2020), Virtual Event*. http://proceedings.mlr.press/v119/yasunaga20a.html

[60] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2022. Automated Classification of Overfitting Patches With Statically Extracted Code Features. *IEEE Trans. Software Eng.* 48, 8 (2022), 2920–2938. https://doi.org/10.1109/TSE.2021.3071750

[61] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empir. Softw. Eng.* 26, 2 (2021), 20. https://doi.org/10.1007/s10664-020-09920-w

[62] Anna Zeng and Will Crichton. 2018. Identifying Barriers to Adoption for Rust through Online Discourse. In *PLATEAU@SPLASH '18*. Boston, MA.

[63] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership Guided C to Rust Translation. In *Proceedings of the 35th International Conference on Computer Aided Verification (CAV'2023), Paris, France*. https://doi.org/10.1007/978-3-031-37709-9_22

[64] Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. 2022. HeteroGen: transpiling C to heterogeneous HLS code with automated test generation and program repair. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'2022), Lausanne, Switzerland*. https://doi.org/10.1145/3503222.3507748

[65] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'2023), Melbourne, Australia*. https://doi.org/10.1109/ICSE48619.2023.00126

[66] 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 (ICSE'22)*. Pittsburgh, Pennsylvania. https://doi.org/10.1145/3510003.3510164