Learning and Programming Challenges of Rust: A Mixed-Methods Study

Anonymous Author(s)

ABSTRACT

Rust is a young systems programming language designed to provide both the safety guarantees of high-level languages and the execution performance of low-level languages. To achieve this design goal, Rust provides a suite of safety rules and checks against those rules at the compile time to eliminate many memory-safety and thread-safety issues. Due to its safety and performance, Rust's popularity has increased significantly in recent years, and it has already been adopted to build many safety-critical software systems.

It is critical to understand the learning and programming challenges imposed by Rust's safety rules. For this purpose, we first conducted an empirical study through close, manual inspection of 100 Rust-related Stack Overflow questions to understand 1) what safety rules are challenging to learn and program with, 2) under which contexts a safety rule becomes more difficult to apply, and 3) whether the Rust compiler is helpful enough in debugging safety-rule violations. We then performed an online survey with 101 Rust programmers to validate the empirical study. We invited participants to evaluate program variants that differ from each other, either in terms of violated safety rules or the code constructs involved in the violation, and compared the participants' performance on the variants. Our mixed-methods investigation revealed a range of consistent findings that can benefit Rust learners, practitioners, and language designers.

1 INTRODUCTION

Rust is a new programming language designed to build safe and efficient systems software [28, 33]. The key innovation of Rust is its suite of safety rules that are checked against during compilation to catch memory-safety and thread-safety issues. Alongside the language's safety mechanism, Rust maintains its compiled executable programs to be as efficient as C programs. Due to its safety and efficiency, Rust has become increasingly popular; it was rated the most beloved programming language every year since 2016 [51–55, 57] and was the fifth fastest growing language on GitHub in 2018 [34]. Rust has already been adopted by many open-source programmers and big tech companies to build safety-critical software [25, 31, 36, 37, 40, 48].

Rust's safety mechanism centers around two important concepts: *ownership* and *lifetime*. The basic safety rule requires each value to have exactly one owner variable, and the value is freed when its owner variable ends its lifetime. To improve programming flexibility, Rust extends this basic rule to a suite of extended rules, such as allowing ownership to be moved to another owner or to be borrowed using a reference, and still guarantees memory safety and thread safety. Rust's safety mechanism is elegant and effective. It essentially prohibits programs from having mutability and aliasing at the same time, and inherently avoids many severe memory bugs (*e.g.*, use after free) and concurrency bugs (*e.g.*, data race). A recent empirical study reports that if a program is written solely in safe Rust code, then it will have no memory bugs, confirming the effectiveness of Rust's safety mechanism in practice [38].

Unfortunately, Rust is known to have a steep learning curve and is difficult to program in practice [1, 66]. The ease with which programmers can write code that violates Rust's safety rules and is rejected by the Rust compiler comes down to two reasons. First, Rust's safety mechanism is unique, and the related grammar and semantics are very different from traditional systems programming languages (*e.g.*, C/C++) [27]. Thus, it is difficult for programmers to migrate their previous programming experience gained from other languages to coding in Rust [45]. Second, the design philosophy of Rust is to reject all suspicious code and force programmers to prove their code follows all safety rules. Rust's safety checks are strict, sometimes overly so, making Rust code hard to be compiled.

A piece of Rust code is shown in Figure 1a. Structs `Outer1` and `Outer2` are declared at lines 4 and 5, respectively. The two structs are similar to each other in the sense that both of them only contain one field with the same name and the same contents (two `Inner` objects). However, the field of `Outer1` is an array, while the field of `Outer2` is a tuple. Function `test()` takes two `Inner` objects as inputs. It uses a mutable reference to borrow the first `Inner` object and an immutable reference to borrow the second one. Function `test()` is called at line 14 using the two `Inner` objects in an `Outer1` object as inputs. However, the Rust compiler reports an error on this line (Figure 1b). The reason for the error is that elements of an array must be borrowed altogether in Rust (or after an element is borrowed, all other elements in the same array are also considered as being borrowed), since the Rust compiler conservatively assumes an index can access any element in an array. Rust does not allow a mutable reference to coexist with other references to the same object to prevent simultaneous mutability and aliasing. Array `out1.a` is already mutably borrowed as the first parameter. Thus, it cannot be borrowed again as the second parameter. Counter-intuitively, line 17 is allowed by the compiler, because different tuple fields can be borrowed separately.

Figure 1b shows the error messages reported by the Rust compiler. The compiler points out which ownership rule is violated, where it is violated, and how it is violated, but it fails to provide the most important information for the programmer that array elements are borrowed together in Rust, causing the programmer to go to Stack Overflow to ask for more explanations about the code and the error messages [49].

The above case demonstrates the complexity of applying Rust's safety mechanism under concrete coding scenarios and the difficulty in writing programs accepted by the Rust compiler. Besides the safety rule that a mutable reference cannot coexist with another reference to the same object, programmers must also know how array (and tuple) elements are borrowed to avoid similar mistakes. Moreover, the compiler may not always provide all necessary information for programmers to understand and fix the errors.

```
1  #![allow(unused_variables)]
2
3  struct Inner { inner: u8 }
4  struct Outer1 { a: [Inner; 2] }
5  struct Outer2 { a: (Inner, Inner) }
6
7  fn test(in1: &mut Inner, in2: &Inner){}
8
9  fn main() {
10   let mut out1 = Outer1 { a:
11     [Inner {inner: 1}, Inner {inner: 3}]};
12   let mut out2 = Outer2 { a:
13     (Inner {inner: 1}, Inner {inner: 3})};
14 -  test(&mut out1.a[0], &out1.a[1]);
15 +  let (first, rest) = out1.a.split_first_mut().unwrap();
16 +  test(first, &rest[0]);
17   test(&mut out2.a.0, &out2.a.1);
18 }
```

```
20 let r1 = &mut out1.a[0];
21 let r3 = &mut out2.a.0;
22 let r2 = &out1.a[1];
23 let r4 = &out2.a.1;
24 *r1 += 1;
25 *r3 += 1;

26 println!("{:?}", r2);
27 println!("{:?}", r4);
```

PC-1 changes to PC-3

(a) Rust program

```
error[E0502]: cannot borrow `out1.a[_]` as immutable because it is also
borrowed as mutable
  --> demo-snippet3.rs:14:24
   |
14 |   test(&mut out1.a[0], &out1.a[1]);
   |   ---- --------------  ^^^^^^^^^^ immutable borrow occurs here
   |   |    |
   |   |    mutable borrow occurs here
   |   mutable borrow later used by call
   |
note: `a` is an array and can only be borrowed as a whole
   |
 4 | struct Outer1 { a: [Inner; 2] }
   |                 ^

error: aborting due to previous error

For more information about this error, try `rustc --explain E0502`.
```

(b) Compiler error messages

Figure 1: A Rust program and its compile error. ***In Figure 1a, the program is PC-1 in the survey, tokens colored in red violate a safety rule, and "+" and "-" denote code added and deleted to fix the violation. We replaced lines 14 and 17 with the code in the cyan rectangle to create PC-3 in the survey. In Figure 1b, the "note" part in the green rectangle does not belong to the original error messages, and we added it in the survey.***

Our ultimate goal is to facilitate the learning and programming of Rust. We take the identification of the challenges imposed by Rust's safety rules as the first step. Those rules are unique and complex. As shown by the empirical study in Section 3, they indeed cause challenges to Rust programmers in the real world. Rust is still evolving [10, 60]. Learning Rust is a continuous process and programming Rust in practice often involves studying how to apply a safety rule under a particular coding context. Thus, we do not differentiate learning from programming in this paper. Overall, we aim to answer the following research questions (RQs).

- **RQ-1:** Which Rust safety rules are difficult to understand?
- **RQ-2:** Under which programming contexts is a safety rule more challenging to apply?
- **RQ-3:** How helpful is the Rust compiler in resolving programming errors due to safety-rule violations?

We took two approaches to answer the above questions. We first conducted an empirical study on Rust-related Stack Overflow questions, since programmers usually seek technical advice on Stack Overflow for issues they cannot resolve on their own [2, 14, 62, 67]. We then performed an online survey to validate the findings in the empirical study by closely examining how Rust programmers answer carefully designed survey questions.

We built two datasets for the empirical study. The larger one contains 15,509 Rust-related Stack Overflow questions, and the smaller one contains 100 questions caused by violations of Rust's safety rules. To answer **RQ-1**, we built a taxonomy for safety-rule violations in the small dataset. The taxonomy contains two major categories: *complex lifetime computation* and *violating ownership rules*. Each of these contains several sub-categories. To answer **RQ-2**, we applied the LDA model [9] to the large dataset, and computed the correlation between violated safety rules and involved code constructs for the small dataset. We manually interpreted the results to identify scenarios where a safety rule is more challenging to apply. To answer **RQ-3**, we examined whether the Rust compiler provides all necessary information for debugging safety-rule violations in the small dataset.

We gained several important findings from the empirical study. For example, Rust's safety rules are difficult for programmers to apply in practice, and computing a lifetime is more challenging than applying an ownership rule. Moreover, some safety-rule violations are highly correlated with particular code constructs, indicating the corresponding rules are more challenging to apply to those code constructs. In addition, the Rust compiler may not provide all necessary information for comprehending safety-rule violations.

In the online survey, we first asked for participants' demographic information, technical background, and previous experience in interacting with Rust's safety mechanism and the Rust compiler. We then showed them four small Rust programs, named PA, PB, PC, and PD. We only asked participants whether PA and PB could be compiled to test their Rust knowledge. We sampled PC and PD from two sets of similar program variants. All variants contained a safety-rule violation. They were different from each other either in the safety rules they violated or in the involved code constructs. For both PC and PD, we asked participants to 1) pinpoint error root causes by highlighting program tokens, 2) evaluate how difficult it was to comprehend the errors before and after seeing the error messages, 3) select the violated rules, 4) rate the helpfulness of the Rust compiler, and 5) describe the error root causes in their own words. We received 101 valid responses. After extensive data analysis, we confirmed many findings reported in the empirical study with significant confidence.

Overall, our mixed-methods investigation reveals what to learn about Rust, how to learn, and how to interpret compiler error messages, all of which can benefit Rust learners and programmers. Moreover, our investigation pinpoints information missed by the Rust compiler when reporting safety-rule violations, and thus provides valuable guidance for the evolvement of the Rust compiler. In sum, this paper makes the following key contributions.

- We performed the first empirical study on Stack Overflow questions related to violations of Rust's safety mechanism.
- We gained seven findings of programming challenges caused by Rust's safety rules and the helpfulness of the Rust compiler in debugging safety-rule violations. Those findings can be useful for Rust learners and programmers' references.
- We conducted an online survey and confirmed our findings with statistical significance.

All our study results and survey results can be found at https://bit.ly/3gXmccv.

```
1 fn max(a:Vec<i8>) ->i8 {71}
2 //fn max(a:&Vec<i8>) ->i8 {71}
3 fn min(a:Vec<i8>) ->i8 {8}
4 fn main() {
5   let foo = vec![71, 23, 8];
6   let mx = max(foo);
7   let mn = min(foo);
8   println!("{} {}", mx, mn);
9 }
```

Figure 2: An example of ownership move. ***The program cannot be compiled, since*** `foo` ***is moved at line 6 and it cannot be used at line 7.***

```
1 fn bar(x: &mut i32) {
2   println!("{}", x);
3 }
4 fn main() {
5   let mut a = 100;
6   let y = &a;
7   println!("{}", y);
8   bar(&mut a);
9 }
```

Figure 3: An example of ownership borrowing. ***The program can be compiled.***

2 BACKGROUND

This section gives some background for this project, including Rust's safety mechanism and the information provided by the Rust compiler for safety-rule violations.

2.1 Rust's Safety Mechanism

Rust's safety mechanism centers around two critical concepts, *ownership* and *lifetime*. The basic rule requires that a value is associated with *one and only one* owner variable, and that the value is dropped (freed) when its owner variable's lifetime ends. Sometimes, the place where a variable's lifetime ends is easy to determine, such as at the end of a function or at a matched curly bracket. However, there are cases where lifetime computation is much more complex than inspecting a variable's lexical scope. To improve its programming flexibility, Rust extends its basic safety rule into a suite of rules, while still guaranteeing memory safety and thread safety.

Ownership Move. Rust allows a value's ownership to be moved to a different owner variable or to a different scope (*e.g.*, a function, a closure), but prohibits any access to the previous owner variable after the move. For example, array `foo` is moved to function `max()` at line 6 in Figure 2, since the parameter type of function `max()` is "`Vec<i8>`", not "`& Vec<i8>`" like the function at line 2. Thus, the Rust compiler reports an error at line 7, since `foo` has already been moved and it cannot be accessed anymore.

Ownership Borrow. Rust allows to temporarily borrow a variable's ownership using a reference, which can be immutable for read-only accesses or mutable for read-write accesses. A borrow ends at the last usage site of the reference, and Rust requires that a reference can only be used within its borrowed variable's lifetime. Rust permits multiple immutable references to a variable to exist at the same time, but only allows at most one mutable reference to a variable at any time. Those rules essentially guarantee all accesses to a variable are within its lifetime and forbid simultaneous mutability and aliasing, avoiding many severe memory and concurrency bugs. For example, in Figure 3, variable `a` is immutably borrowed by `y` at line 6 and mutably borrowed when calling `bar()` at line 8. Although the lexical scope of `y` does not end until the end of function `main()` at line 9, because `y` is not used after line 7, the Rust compiler decides the borrow ends at line 7 and does not overlap with the mutable borrow at line 8. Thus, the compiler compiles the program in Figure 3.

Lifetime Annotation. Rust allows programmers to explicitly annotate a variable's lifetime with an apostrophe followed by an annotation name. Lifetime annotations are commonly used at function declaration sites to specify the relationship between two parameters or the relationship between a parameter and the return, and at struct declaration sites to describe the lifetime requirement between a struct object and its reference fields. When checking a function or a struct, the compiler reports errors when safety-rule violations are inferred based on the lifetime annotations of the function or struct. When calling a function, the compiler inspects whether the real parameters satisfy the corresponding annotations. Rust also allows lifetime elision to reduce the annotation burden, and the compiler automatically infers elided annotations during safety checks.

Safe vs. Unsafe. All code discussed so far has been safe Rust code. Rust permits programmers to use the "`unsafe`" keyword to bypass some safety checks and conduct unsafe operations (*e.g.*, pointer operations, calling an unsafe function). Unsafe code is similar to the traditional C programming language. A piece of code or a function can be unsafe. A function can also be interior unsafe by containing unsafe code internally but exposing a safe API externally, and it can be used as a safe function. In this paper, we focus on understanding programming challenges when coding safe code, since safe code must strictly follow Rust's safety mechanism, and it is used much more often than unsafe code in Rust programs [38].

2.2 Rust's Compiler Error Messages

The Rust compiler serves as the primary communication channel between programmers and Rust's safety mechanism. It checks against the above-mentioned safety rules and reports an error when detecting a rule violation. Typically, a piece of error messages contains three components: 1) the violated safety rule and its corresponding error code, 2) the lines of code or program tokens that violate the rule, and 3) some explanations about the violation. For example, Figure 1b shows the error messages for the program in Figure 1a, which present the error code ("E0502") and the violated rule ("cannot borrow ... as mutable") at the beginning, underline program tokens violating the rule in red, and underline several other tokens in blue to provide more information. Sometimes, error messages contain suggestions about how to fix an error or even directly give a concrete patch. Moreover, the Rust compiler provides a generic explanation for every error code, which can be obtained by executing rustc (*e.g.*, "`rustc --explain E0502`" in Figure 1b).

Unfortunately, safety rules are complex [13, 46] and some are counter-intuitive [38]. Moreover, compiler error messages may be imprecise [20] or even contain misleading information [18, 19]. Thus, compiler error messages may not be good enough for Rust programmers to debug and fix safety-rule violations. In Section 3.4, we will combine cognitive task analysis (CTA) [29] and manual inspection of safety-rule violations in real Rust programs to systematically evaluate error messages reported by the Rust compiler.

3 STUDYING STACK OVERFLOW QUESTIONS

This section presents our empirical study on Stack Overflow questions. Our study aims to answer the research questions outlined above. Its results can guide the learning process of Rust and improve the interaction between programmers and the Rust compiler.

3.1 Methodology

We construct a large dataset and a small dataset for statistical analysis and manual inspection, respectively.

Root Causes	Safe	Unsafe	SL	IUL	UL	No	Total
Complex Lifetime Computation							
Intra-procedural	31	0	1	10	0	2	44
Inter-procedural	19	2	1	4	0	4	30
Simple Syntax Error	3	0	0	0	0	0	3
Violating Ownership Rules							
Move Rule	12	0	1	5	0	0	18
Borrowing Rule	9	1	3	8	0	2	23
Total	74	3	6	27	0	8	118

Table 1: Root causes and fixes of violations in the small dataset. ***Safe/Unsafe: directly writing safe/unsafe code; SL: safe libraries; IUL: interior unsafe libraries; UL: unsafe libraries; and No: eight violations do not have fixes.***

3.1.1 Large Dataset. The large dataset contains all Stack Overflow questions that are labeled with tag "Rust", have a score greater than or equal to zero, and have at least one answer as of February 17, 2021. In total, there are 15,509 questions in the large dataset.

We randomly sampled 100 questions from the large dataset and manually inspected why programmers asked them on Stack Overflow. Common reasons include not knowing how to use a library function (26%), being unable to understand Rust's safety rules (23%), being confused by type conversions and type checks in Rust (14%), not knowing how to implement or use a trait (similar to an interface in Java) (9%), and failing to use FFI properly (7%).

Finding 1: *Rust shares many programming challenges with traditional programming languages, and its complex safety rules also bring unique difficulties.*

3.1.2 Small Dataset. We randomly sampled 100 questions related to Rust's safety mechanism from the large dataset to build the small dataset. Each sampled question contains a code snippet for describing the problem. Based on those snippets, we successfully implemented standalone programs and reproduced all problems offline. For eight questions, the programs can be compiled, but the compilation contradicts the questioners' understandings. We consider each of these questions as a case where a programmer's understanding violates a safety rule. For all other questions, the programs cannot be compiled. Among these, 76 programs contain one violation of a safety rule, 14 programs contain two violations, and the remaining two contain three violations. In total, there are 118 safety-rule violations in the small dataset.

3.2 Which Safety Rules are Difficult?

To figure out which safety rules are difficult and are more likely to cause usage violations, we build a taxonomy for the root causes of the violations in the small dataset. As shown in Table 1, we first divide the root causes into complex lifetime computation and violating ownership rules. We then separate each of these categories into several sub-categories.

3.2.1 Complex Lifetime Computation. Lifetime computation may be much more complicated than referring to a variable's lexical scope. 77 violations are due to complex lifetime computation. For most of them, programmers estimate a variable's lifetime to be longer or shorter than it really is, thus violating a safety rule. We further divide the violations into those due to intra-procedural lifetime computation, those due to inter-procedural lifetime computation, and those caused by syntax errors when declaring a struct.

```
1  struct Foo {}
2  struct Bar2<'b> { x: &'b Foo,}
3
4  impl<'b> Bar2<'b> {
5  -  fn f(&'b mut self)-> &'b Foo {
6  +  fn f(&mut self)-> &'b Foo {
7        self.x
8     }
9  }
```

```
10 fn f4() {
11    let foo = Foo {};
12    let mut bar2 = Bar2 {
13       x: &foo };
14    bar2.f();
15    let z = bar2.f();
16 }
```

Figure 4: An example of complex inter-procedural lifetime computation. ***Tokens colored in red are the root-cause tokens. "+" and "-" denote code added and deleted to fix the violation.***

Intra-procedural Lifetime Computation. Lifetime computation may be difficult even for cases within a single function. 44 violations are in this category, 32 of which are cases where programmers miscompute variable lifetimes when using particular code constructs, including control flow constructs (*e.g.*, if, loop), data structures (*e.g.*, hashmap, vector), temporary variables, and program constants. For example, SO#65682678[1] is caused by miscomputing the lifetime of a reference held by a closure. As another example, both SO#63428868 and SO#51044568 are caused by errors when computing lifetime inside a `match` block. 11 out of the 44 violations are due to unsatisfied lifetime requirements at a function declaration or a struct declaration. For example, when declaring an `async` function, Rust requires an explicit lifetime annotation for each input object, and violating this requirement is the root cause of SO#62440972. The remaining violation is due to a lack of basic understanding of Rust.

Inter-procedural Lifetime Computation. Thirty violations are due to lifetime computation across function boundaries. Among them, 22 are cases where a real parameter does not satisfy the lifetime requirement of its corresponding formal parameter. The remaining eight are caused by unexpected lifetime extensions through a function call. Figure 4 shows one such example. Function `f()` is implemented for struct `Bar2` at line 5. It borrows a `Bar2` object and returns its field `x`. Lifetime annotation `'b` is specified for both input reference `self` and the return value, so that the questioner thought the borrowing of a `Bar2` object ended when the corresponding return terminated its lifetime. The questioner also believed that since the return value at line 14 is not saved to any variable, both the lifetime of the return and the borrow of `bar2` ended at line 14. He was confused why the compiler complained that two mutable references to `bar2` exist at line 15. The reason is that lifetime annotation `'b` is also applied to struct `Bar2` at line 4, so that the borrow conducted by function `f()` does not stop until the borrowed object ends its lifetime. Thus, the borrow of `bar2` at line 14 ends line 16, which is out of the questioner's expectation.

When declaring a function or a variable, programmers may explicitly specify all lifetime annotations or choose to elide some annotations. Among the 30 violations in this category, 16 only involve explicit lifetime annotations (*e.g.*, Figure 4), and the lifetime miscomputation or mismatch happens at an elided annotation for the remaining 14 cases (*e.g.*, SO#40053550).

Simple Syntax Errors. Three violations are caused by the misuse of lifetime annotations when declaring or implementing a struct. For

[1] SO#65682678 denotes Stack Overflow question 65682678 [50].

example, the questioner of SO#62422857 only uses an apostrophe to annotate a struct field without providing an annotation name.

3.2.2 Violating Ownership Rules. There are 41 ownership-rule violations in the small dataset, including 18 violations of a move rule and 23 violations of borrowing rules.

Move Rule Violations. A variable cannot be accessed after it is moved. Non-compliance with this requirement causes 18 violations. Of these, 16 violations involve (complex) program constructs. For example, SO#65873356 is caused by accessing an object that is already moved to a called function. As another example, when an object is moved to a closure, it may be unclear to programmers whether the move happens at the closure creation site or at the location where the closure is firstly used, which is the root cause of SO#62125100. The remaining two cases are very simple, and we think the questioners asked the corresponding questions because they did not know the move rule.

Borrowing Rule Violations. Misuse of references leads to 23 violations. Two of them are due to mistakenly borrowing a collection of objects altogether, instead of a single element (*e.g.*, Figure 1a). Another two are cases where programmers wanted to copy an object using a reference, but mistakenly copied the reference. Moreover, 11 cases are due to using a reference to move an object, which is not allowed in Rust. For example, "`a = *x`" moves the object referenced by `x` if the object does not implement the `Copy` trait, which confused the questioner of SO#35649968. In addition, mutability mismatches (*e.g.*, changing a variable using an immutable reference) cause two violations. Rust prohibits a closure from returning a mutable reference, since it leads two mutable references existing simultaneously (one is returned and the other is held by the closure). Not complying with this rule causes three violations. The remaining three are due to not knowing how to use the reference counted library (*i.e.*, `Rc`) or library APIs that take a reference as input.

Finding 2: *Rust's safety mechanism may be difficult to apply in concrete usage scenarios.*

Finding 3: *More lifetime-related questions are asked on Stack Overflow than ownership-related questions, indicating lifetime computation is more challenging in Rust programming.*

3.2.3 How Violations are Fixed? We examine whether unsafe code is used in the violation patches to understand whether programmers can achieve the desired functionalities while complying with all safety rules. Since 8 violations are cases where programmers' understanding (not implementation) conflicts with the safety rules and therefore have no fix, we focus on the remaining 110 cases.

As shown in Table 1, only three violations are fixed by writing unsafe code directly (column "Unsafe"). For example, the questioner of SO#64274964 wants to use two editor objects to modify the same image at the same time. Since the two editors change two different parts of the image, there is no bug logically. However, the Rust compiler does not allow the two editors to have two mutable references to the image at the same time. The patch uses pointers in unsafe code to have two writers for the same image simultaneously.

Another 27 violations are patched with interior unsafe library functions (column "IUL" in Table 1). Although the interfaces of those functions are safe and programmers can use them as safe functions, they actually contain unsafe code internally. For example, SO#57766918 is patched by calling interior unsafe function `into_iter()` [43].

All other violations are fixed by writing safe code directly or using safe library functions (columns "Safe" and "SL" in Table 1). For example, SO#39827244 in Figure 4 is fixed by removing the lifetime annotation of `self` to break the lifetime binding between the borrow conducted by function `f()` and the borrowed `Bar2` object, and SO#62491845 in Figure 1a is fixed by calling safe standard library function `split_first_mut()` [44], which returns the first element of the input array. These patches only involve safe code.

Finding 4: *The majority of safety-rule violations are fixed with safe code, and a small portion of violations are patched using well-encapsulated interior unsafe libraries. Programmers usually do not have to write unsafe code by themselves to fix safety-rule violations.*

3.3 When a Safety Rule is More Confusing?

To detect when a safety rule is more difficult, we first apply the LDA model [9] to the large dataset. We then use a statistical metric *lift* [23, 26, 61] to compute the correlation between root cause categories and involved code constructs for the small dataset.

3.3.1 LDA Model. The LDA model can pinpoint the hidden topics of analyzed documents, and the hidden topics of Rust-related Stack Overflow questions describe when programmers feel Rust is more challenging. We take two steps to apply LDA. We first identify questions related to safety rules. We then run LDA on the questions and manually interpret the identified topics.

We use Stack Overflow tags to identify questions related to safety rules. Following the taxonomy in Section 3.2, we divide safety rules into three groups: lifetime-related rules, move rules, and borrowing rules. We find 790, 28, and 848 questions respectively for these rule groups in the large dataset.

For each group of rules, we use the Gensim package [41] to run bigram LDA on all its identified questions. We remove Rust code in those questions and only consider question titles, descriptions, and answers in the analysis. We preprocess the texts using NLTK [7] to lemmatize words and to remove stop words and punctuations. We try all numbers from 5 to 30 as the topic number to configure the model. We manually inspect results for the topic number with coherence value [30] closest to zero, since a coherence value closer to zero represents a better clustering result. The topic numbers with the best coherence value for lifetime, borrowing, and move are 5, 5, and 9, respectively.

After reading the top words and representing questions reported by LDA, we identify several challenging scenarios for each group of rules. For example, 204 questions contain the topic about how to use lifetime annotations in a trait, 32 questions contain the topic about how to borrow from an iterator or a container, and and three questions are about how to use move with pattern matching. Rust programmers can refer to our identified topics to enhance their understanding of Rust's safety rules.

Finding 5: *Statistically analyzing Stack Overflow questions helps pinpoint programming challenges due to Rust's safety mechanism.*

3.3.2 Lift Correlation. We use the lift metric to measure the correlation between root cause categories in Section 3.2 and code

constructs. The lift of category A and code construct B is computed as $lift(AB) = \frac{P(AB)}{P(A)P(B)}$, where $P(AB)$ represents the probability of a violation that is due to A and also involves B, $P(A)$ means the probability of a violation caused by A, and $P(B)$ denotes the probability of a violation involving B. If $lift(AB)$ equals 1, A is independent of B. If $lift(AB)$ is larger than 1, A and B are positively correlated, indicating when the safety rule of A is applied to B, it is more likely to cause problems and it is more challenging. The larger the lift value is, the more positively A and B are correlated. If $lift(AB)$ is smaller than 1, A and B are negatively correlated.

Among all code constructs with at least ten violations, root cause "inter-procedural lifetime computation" is most correlated with the `'static` code construct. The lift value is 2.36. Self-defined annotations and generics are ranked as the second and the third most correlated code constructs with "inter-procedural lifetime computation." Their lift values are 2.32 and 2.14, respectively. "Intra-procedural lifetime computation" is most correlated with standard library `Box`, function declarations, and `return` statements, with lift values 2.19, 1.89, and 1.87, respectively. The top three code constructs correlated with "move rule violations" are loops (1.96), vectors (1.38), and function calls (1.19). The largest three lift values for "borrowing rule violations" are 1.57 for hashmaps, 1.35 for iterators, and 1.31 for closure declarations.

Many of those widely used code constructs have different lift values with different root cause categories. For example, function declarations are positively correlated with "intra-procedural lifetime computation", but it is negatively correlated with the other three categories. As another example, generics are positively correlated with "inter-procedural lifetime computation"; however, they are roughly independent of "borrowing rule violations."

Finding 6: *The same rule has different difficulty levels when applied to different code constructs, and different rules have different difficulty levels when applied to the same code construct.*

3.4 Evaluating Compiler Error Messages

As we discussed earlier, 110 rule violations in the small dataset can trigger compiler errors[2]. The Rust compiler associates 20 different error codes to 103 of the 110 violations. Error code "E0382" (*i.e.*, accessing a variable after it is moved) appears most frequently at 19 times. The compiler does not provide an error code for the remaining seven violations, which are caused by four different uncommon problems.

We leverage the 110 violations to evaluate whether the compiler provides all necessary information for programmers to comprehend safety-rule violations. This evaluation comes in two steps. We first conduct cognitive task analysis (CTA) [15] to identify the steps taken by Rust experts to comprehend error messages. We then follow those steps to analyze whether the information required at each step is provided in the error messages for each violation.

Cognitive Task Analysis (CTA). A CTA typically interviews three to five experts in the investigated subject [11]. Moreover, our CTA aims to identify *how* experts comprehend Rust's compiler error messages, other than to sample their opinions on particular compiler errors. Thus, we chose three paper authors as the CTA participants and another author as the CTA analyst. All participants have at least one year's experience in programming Rust and use Rust on a weekly basis.

Our CTA contains a think-aloud observation and a semi-structured interview [12]. In the think-aloud observation, participants were asked about their general impression of Rust's compiler errors and the steps they take to understand compiler error messages. The analyst recorded each participant's think-aloud and analyzed the recording to identify key steps in comprehending error messages. In the semi-structured interview, the analyst asked participants questions about the key steps for the validation purpose. The analyst interviewed the three participants separately to avoid premature consensus.

Both the think-aloud observation and the semi-structured interview were audio-recorded and automatically transcribed. The analyst combined all responses into a description of the procedures, decisions/actions, concepts, and conditions/situations used by experts to comprehend Rust's compiler error messages.

We further invited three *external* experts to evaluate the description. All experts have at least two years' Rust experience and use Rust on a daily basis. The initial average proportion of agreement was 0.7, a satisfactory agreement level [24], indicating our method is *sufficient* to capture how programmers comprehend Rust's compiler errors. We updated a few description components based on the external experts' comments. In the end, the description was agreed upon by both internal and external experts.

Studying Violations. We then follow the description to examine whether error messages contain all needed information for comprehending each violation. For 59 out of the 110 violations, their error messages contain all necessary information. Programmers can use the highlighted code and the compiler's explanations to figure out why the Rust compiler rejects the code. Error messages miss some important information for the remaining 51 violations; these fall into three categories.

First, for nine violations, the Rust compiler fails to explain how a safety rule works on a particular code construct. For example, the error messages in Figure 1b do not mention that elements of an array cannot be borrowed individually in Rust.

Second, for 32 violations, the compiler fails to explain key steps in computing a lifetime or a borrowing relationship. For example, the error messages of SO#39827244 in Figure 4 do not explain why the borrow of `bar2` at line 14 does not end until the borrowed object terminates its lifetime at line 16.

Third, in the remaining ten cases, the compiler fails to explain the relationship between two lifetime annotations, making it difficult to understand the annotation mismatch in the error messages. For example, SO#53835730 is due to using two references with different lifetimes to call a function that requires two inputs to have the same lifetime. The compiler simply complains that the second reference does not live as long as the elided lifetime annotation of the first reference. However, it does not explain that the first reference is the input of the caller function and has a lifetime longer than the caller. In contrast, the second reference is a reference to a local variable and has a lifetime shorter than the caller.

Finding 7: *The Rust compiler may not provide all information necessary to comprehend violations of Rust's safety rules.*

[2]The compiler version we evaluate is 1.50.0, which was released in February 2021.

ID	Root Cause	Code Construct	Err. Code	#Responses
PC-1	borrowing	function, array	E0502	34 (16, 18)
PC-2	move	function, array	E0508	31
PC-3	borrowing	local variable, array	E0502	32
PD-1	inter lifetime	function, annotation	E0499	36 (19, 17)
PD-2	move	function, annotation	E0382	29
PD-3	intra lifetime	closure, annotation	E0499	32

Table 2: Program variants of PC and PD. ***PC-1 in Figure 1a and PD-1 in Figure 4 are the two base programs. (x, y) represents x participants were assigned with the original error messages and y participants were assigned with enhanced messages of PC-1 or PD-1.***

4 SURVEYING RUST PROGRAMMERS

We conducted an online survey on Qualtrics [39] to validate the findings in Section 3. The survey was approved by the institutional review board (IRB) office at the authors' university. This section gives the survey details and survey results.

4.1 Methodology

4.1.1 Recruitment. We required participants to be at least 18 years old, not be residing in the European Economic Area, and have some Rust coding experience. To recruit participants, we distributed our survey by posting threads on Rust-specific forums and newsletters, sending emails to programmers who recently committed code to open-source Rust projects, and contacting industrial collaborators.

4.1.2 Stimuli. We presented four Rust programs (PA, PB, PC, and PD) to each participant. All the programs were designed based on the studied Stack Overflow questions in Section 3. PB can be compiled, while the other three all contain safety-rule violations. PA is shown in Figure 2 and PB is shown in Figure 3. They are identical for all participants. PC and PD are sampled from two program sets. Each set contains a base program (PC-1 or PD-1) and two variant programs that are synthesized either by changing the base program's violated safety rule (PC-2 or PD-2) or involved code constructs (PC-3 or PD-3). Thus, we can compare survey results between a base program and its variants (*e.g.*, PC-1 vs. PC-3) to validate Finding 6 in Section 3.3.

Table 2 shows the information of program variants in the two sets. PC-1 is the program in Figure 1a and it is the base program in the PC set. The Rust compiler complains a mutable reference to `out1` coexists with an immutable reference to `out1` with error code "E0502." We created PC-2 by changing function `test()` to move its first parameter. Then a different safety rule which states that an array element cannot be moved out of its array was violated (error code "E0508"). As shown in Figure 1a, we changed the code constructs involved in the error to create PC-3. We replaced the two function calls at lines 14 and 17 with several borrowing operations and assignments, with the purpose being to have `out1`'s mutable reference `r1` to coexist with `out1`'s immutable reference `r3`. In addition, we enhanced the compiler error messages of PC-1 by explicitly explaining that an array can only be borrowed as a whole in Rust (the green rectangle in Figure 1b).

PD-1 in Figure 4 is the base program of the PD set. How we created PD-2 and PD-3 is similar to how we created PC-2 and PC-3. See Table 2 for the violated rules and involved code constructs.

4.1.3 Procedure. Our survey consisted of three phases. We discuss their details as follows.

Phase 1: Demographics and on-board Experience in Rust. We collected demographic information at the beginning of the survey to validate whether participants were qualified. We asked participants to provide their age groups, locations, and number of years of Rust experience. If a participant did not satisfy any recruitment requirement, our survey automatically terminated. We then asked participants about their genders, races, and ethnic groups.

We gauged participants' expertise levels in Rust by asking them how long they have learned Rust, how often they program with Rust, how many lines of code are in the largest Rust programs they have written, and whether Rust is their most frequently used language. We also asked participants to self-rate their expertise on a 10-point scale ("1" means "beginner" and "10" means "expert").

To examine participants' previous Rust coding experience, we asked participants how often they feel confused about Rust's ownership/lifetime rules and how often they can understand compiler error messages when their code violates the safety rules. Possible options include "Never", "Sometimes", "Most of the time", and "Always." We also asked whether Rust has other language features they consider challenging.

Phase 2: Evaluating Rust Programs. We showed participants the four programs at this phase. We first asked them whether PA and PB could be compiled. We then explicitly told them that both PC and PD contained a safety-rule violation, and asked them to answer the following six questions.

- **Q1:** We asked participants to highlight the program tokens that were the error's root cause (*i.e.*, tokens where safety rules were violated). Figure 1a and Figure 4 show our expected highlighting for PC-1 and PD-1, respectively.
- **Q2:** We requested participants to rate the difficulty of identifying the root cause on a 10-point scale, where "1" means "very easy" and "10" means "very difficult."
- **Q3:** We asked participants to choose the violated rule among ten options, which included the correct answer, four rules similar to the violated one, and five rules different from the violated one. The similarity between two safety rules was computed as the absolute value between their error codes [32].
- **Q4:** We asked participants to rate the difficulty of root cause identification again after showing them the compiler error messages. If PC-1 or PD-1 was sampled, either the original error messages or the enhanced version would be presented with equal probability.
- **Q5:** We requested participants to gauge the helpfulness of the error messages on a 10-point scale, where "1" means "not helpful" and "10" means "extremely helpful."
- **Q6:** In this open-ended question, we asked participants to describe how the safety rule was violated. We suggested they should consider the involved program constructs, how the rule was applied to the code context, and why the compiler highlighted some code in their answers.

Among the questions, Q1, Q3, and Q6 had correct answers, while the other three were based on participants' subjective assessments. Q4–Q6 were asked after showing participants compiler error messages, and their results can reveal the effects of viewing the error messages.

Phase 3: Post-session Questions. We asked about participants' overall technical background at this stage, including years of programming, favorite programming languages, self-rated programming expertise levels, how many lines of code were in the largest programs they had ever worked on, and job titles. We also asked participants why they choose to learn or use Rust.

4.2 Survey Results

We distributed the survey from March 12 to April 6, 2021 and received 502 completed responses. Three paper authors inspected these responses together and identified 101 valid responses. We then focused our data analysis on the valid responses. Invalid responses included those that were finished in a very short time (*e.g.*, less than five minutes), had open-ended question answers identical to other responses, or came from unwanted sources.

4.2.1 Phase 1. The majority of the 101 participants were male (91.1%) and in the 18–34 age range (85.1%). The top two most common locations were the U.S. (53.5%) and China (9.0%). Most participants were White (47.5%) and Asian (27.7%). There were also responses from Hispanics (5.9%) and African Americans (2.0%). Overall, the demographics of our participants revealed considerable diversity and reflected the demographic distribution of real-world Rust programmers [46, 47].

The participants were relatively experienced in Rust (69.3% had learned Rust for more than one year and 63.3% had implemented a Rust program with more than 1000 lines of code). Rust was the most frequently used language for 48.5% of the participants, and 64.4% used Rust on a daily or weekly basis. The average level of self-rated Rust expertise was 5.3 out of 10, and the median was 6.

Most participants (85.1%) were at least "sometimes" confused by lifetime rules, but only 52.4% of them held the same feeling for ownership rules. A chi-square test confirmed the difference was significant ($\chi^2_{(1)} = 23.6, p < 0.001$). Moreover, the proportion of participants who could "always" understand compiler errors for lifetime-rule violations (10.0%) was significantly smaller than the proportion of participants who could "always" understand compiler errors for ownership-rule violations (39.6%, $\chi^2_{(1)} = 22.4, p < 0.001$). These results are consistent with Finding 3 in Section 3.2.

Besides ownership and lifetime, other challenging language features mentioned by more than ten participants were Rust's type systems (26), asynchronous programming (25), trait bounds and generics (22), and macros (12).

4.2.2 Phase 2. Most (83.2%) of the participants correctly answered that PA could not be compiled, 77.2% correctly answered that PB could be compiled, and 70.3% correctly answered both of the two questions. These results show the participants had reasonably adequate knowledge of Rust.

As discussed in Section 4.1.2, PC and PD were sampled from two different program sets. Each set contained three program variants. Table 2 shows the number of participants assigned with each variant. We conducted a one-way ANOVA test for each combination of a program set and a research question (Q1–Q6). The null hypothesis was that there was no difference among the results obtained from the three variants. The significance level was 0.05. Since there were three pairwise comparisons in a program set, we adjusted all computed pairwise p-values using Bonferroni correction [8]. We denoted adjusted p-values using $p_{adj.}$. We mainly focused on (PC-1, PC-2), (PC-1, PC-3), (PD-1, PD-2), and (PD-1, PD-3), since variants in these comparison pairs are different from each other either in terms of the violated rules or involved code constructs in Table 2.

ID	Q1	Q2	Q3	Q4	Q5	Q6
PC-1	0.74 (0.06)	6.12 (0.44)	0.76 (0.07)	3.71 (0.47)	8.50 (0.32)	0.69 (0.05)
PC-1_o	-	-	-	3.46 (0.74)	8.93 (0.32)	0.56 (0.08)
PC-1_e	-	-	-	3.89 (0.62)	8.15 (0.51)	0.78 (0.07)
PC-2	0.39 (0.08)	5.27 (0.40)	0.48 (0.09)	4.03 (0.44)	7.45 (0.44)	0.56 (0.05)
PC-3	0.63 (0.06)	6.32 (0.37)	0.84 (0.06)	3.34 (0.41)	7.76 (0.40)	0.74 (0.03)
PD-1	0.53 (0.09)	7.63 (0.47)	0.22 (0.07)	5.28 (0.56)	7.19 (0.42)	0.66 (0.04)
PD-1_o	-	-	-	4.67 (0.86)	7.33 (0.63)	0.68 (0.04)
PD-1_e	-	-	-	5.82 (0.72)	7.06 (0.59)	0.65 (0.06)
PD-2	0.75 (0.07)	5.59 (0.51)	0.59 (0.09)	3.24 (0.46)	8.24 (0.38)	0.74 (0.05)
PD-3	0.46 (0.09)	7.09 (0.43)	0.37 (0.08)	4.57 (0.47)	6.83 (0.41)	0.58 (0.04)

Table 3: Average scores and standard errors. ***Standard errors are in parentheses. o (e) in a subscript denotes original (enhanced) error messages. Error messages were shown in between Q3 and Q4, and thus they did not impact Q1–Q3.***

Q1: Error Token Highlighting. We categorized program tokens into three types for grading: root-cause tokens, relevant tokens, and irrelevant tokens. Root-cause tokens represented where safety rules were violated (*e.g.*, the tokens colored in red in Figure 1a and Figure 4). Relevant tokens were those close to or related to root causes, but that did not directly cause programming errors (*e.g.*, the uncolored tokens at line 14 in Figure 1a). Considering participants might accidentally highlight extra tokens close to the root causes, we did not penalize the selection of relevant tokens. Irrelevant tokens had nothing to do with the root causes. Highlighting irrelevant tokens indicated participants' misunderstanding of programming errors, and thus we penalized the marking of irrelevant tokens. We computed a score for each answer by dividing the number of highlighted root-cause tokens by the number of highlighted root-cause tokens and irrelevant tokens. A score was in the range from 0 to 1, and a larger score represented a better answer.

The average highlighting scores fell in the range from 0.39 to 0.75 in Table 3, implying that applying Rust's safety rules and identifying error tokens are challenging in general (Finding 2 in Section 3.2).

The ANOVAs confirmed that the main effect of program variant was significant for both the PC set ($p = 0.004$) and the PD set ($p = 0.030$). Among the four comparison pairs, only the difference between PC-1 and PC-2 was significant with $p_{adj.}$ equal to 0.003. Since PC-1 and PC-2 shared the same code constructs but violated different safety rules (Table 2), this result reveals that *different safety rules can have different difficulty levels when applied to the same code construct* (Finding 6 in Section 3.3).

To understand how different code constructs impacted root-cause highlighting, we further examined the highlighting results for the PC variants, since PC-1, PC-2, and PC-3 all contained two similar code constructs: array `out1.a` and tuple `out2.a`. For all of these, tokens related to the array were the root cause tokens, while tokens related to the tuple (*e.g.*, tokens at line 18 in Figure 1a) were irrelevant and misleading. We counted how many participants selected tuple-related tokens. The results for PC-1, PC-2, and PC-3 were 10, 9, and 14, respectively. Such results suggest that *programmers can be confused when applying the same safety rule to different code constructs,* which explains Finding 6 in Section 3.3.

Q2 & Q4: Difficulty Ratings. Column "Q2" in Table 3 shows the average difficulty ratings before participants saw the error messages. The main effect of program variant was significant for the PD set ($p = 0.009$) only. For the two comparison pairs in the PD set, PD-1 was significantly more difficult than PD-2 ($p_{adj.} = 0.01$), indicating that participants perceived the error due to "inter-procedural lifetime computation" to be more difficult to comprehend than the error caused by "move rule violations" on the same code construct (Finding 3 in Section 3.2 and Finding 6 in Section 3.3). There was no significant difference between PD-1 and PD-3. The average difficulty rating showed the same pattern after participants viewed the compiler error messages (column "Q4" in Table 3).

Q3: Violated Rule Selection. Column "Q3" in Table 3 shows the correct answer rates for selecting violated safety rules. The main effect of program variant was significant for both PC variants ($p = 0.003$) and PD variants ($p = 0.008$). Among the four comparison pairs, only the correct answer rate of PD-1 was significantly lower than that of PD-2 ($p_{adj.} = 0.008$), demonstrating PD-1's violated safety rule was more difficult to identify than PD-2's (Finding 6).

To understand the difficulty in identifying violated rules, we inspected the wrongly selected options. In total, there were four wrong options chosen by at least five participants, one for PC-2, two for PD-1, and one for PD-3. PC-2 violated a move rule, but six participants thought an object was borrowed again after being mutably borrowed in PC-2, violating a borrowing rule. Those participants did not notice that function `test()` moved (not borrowed) its input in PC-2. The lifetime computation of a mutable reference in PD-1 was complex, and the lifetime overlapped with another mutable reference to the same object. However, seven participants thought the rule that had been violated was that an owner variable cannot be used after the ownership is borrowed. This rule was also violated in PD-1, but it was not as precise as the correct answer. Five participants thought a reference was used beyond the borrowed object's lifetime. The participants noticed that the lifetime computation was complex, so that they guessed this lifetime rule was violated. For PD-3, six participants said that the violated rule was the one that forbids moving an object through a reference, but the program actually conducted a copy (not move). These results show that *without knowing the correct violated safety rule, programmers easily inspected a programming error in a wrong direction.*

Q5: Helpfulness of Compiler Error Messages. Column "Q5" in Table 3 shows the average helpfulness ratings of the error messages. PC-1 had the highest average rating (8.50), and PD-3 had the lowest average rating (6.83). The main effect of program variant was significant for the PD set ($p = 0.039$) only, and there was no significant difference for the four comparison pairs.

Q6: Error Description in Programmers' Own Words. We first developed a scoring rubric for grading. We identified two scoring schemes representing the "what" aspect and the "how" aspect of error root causes. We gave more weight to the "how" aspect (60%). For each theme, we defined three score levels (30%, 60%, and 100%) and formalized the criteria at each level. Based on the rubric, two paper authors first graded participants' responses independently. The average percentage of agreement between the two graders was 50%. Then, the two graders resolved the discrepancies between them by revisiting the criteria through multiple discussions.

Column "Q6" in Table 3 shows the average score for each program variant. The highest score was 0.74 for PD-2, and the lowest score was 0.56 for PC-2. The main effect of program variant was significant for both the PC set ($p = 0.03$) and the PD set ($p = 0.03$), but there was no significant difference for any comparison pair.

We further inspected how participants answered Q6 together with their answers to previous questions to deeply understand participants' error comprehension process. For PC-2, six participants selected that a borrowing rule was violated in Q3, which was wrong, but five of them correctly mentioned move or ownership in their Q6 answers. We anticipated that the error messages helped the five participants figure out the correct root cause. For PD-1, five participants noticed the violation was due to the lifetime extension of the borrow of `bar2` at line 14 (Figure 4). However, they thought the extension was caused by the return at line 7, indicated by their explanations of Q6. This understanding was wrong, because the extension was actually caused by the lifetime annotations (*i.e.*, `'b`) at lines 4 and 5. This result demonstrates that *Rust's safety rules are complex in practice and although programmers know that a safety rule is violated, they may not know the correct reason for the violation.*

Effect of Enhanced Error Messages. We created enhanced versions of compiler error messages for PC-1 and PD-1. Since the error messages were shown after Q3, we compared the effects of the enhanced version and the original version on Q4–Q6 using two-sample t tests. The enhanced version of PC-1 showed a significantly better average score than the original version on the objective question Q6 ($p = 0.015$). Moreover, four participants explicitly mentioned the extra information we provided in PC-1's error messages in their Q6 answers. Those results confirm that *compiler error messages may not contain all necessary information for Rust programmers to debug safety-rule violations, and providing more facilitating information can improve programmers' performance* (Finding 7 in Section 3.4). The enhanced error messages did not significantly impact participants' performance for both PC-1 and PD-1 on the two subjective questions (Q4 and Q5) and for PD-1 on Q6.

4.2.3 Phase 3. About 75.2% of the participants had at least three years' programming experience. The average self-rated programming expertise was 6.3 and the median was 7. A bit less than half (46.5%) of the participants had worked on a program with more than 10,000 lines of code. The general technical background of the participants is consistent with their Rust background as surveyed in Phase 1. The top two most common job titles/roles were "software engineers" (43.6%) and "students" (31.2%). Besides Rust, the top three programming languages participants had the most experience in were Python, C/C++, and JavaScript. The top three most favorite Rust features were safety, performance, and language features (*e.g.*, functional programming styles, pattern matching). Those results reflect our expectations of Rust programmers.

5 DISCUSSION

This section discusses the implications of our studies, threats to its validity, and our procedures to reduce those threats' influence.

Implications to the Rust community. Our findings can guide the learning and programming processes of Rust, the evolution of the Rust compiler, and future research on Rust. First, Rust's safety

rules are difficult to apply in practice, and they may have different challenge levels when applied to various code constructs. We suggest Rust learners not only learn the basic concepts of Rust but also read different code examples to gain a deep understanding of how a rule works in different code contexts. Second, we find that compiler error messages may miss information important to comprehending safety-rule violations. Thus, programmers should not solely rely on compiler feedback when debugging safety-rule violations. Novel IDE tools can be built to provide more information about Rust's safety rules. Third, we pinpoint several types of information missed by the compiler when reporting safety-rule violations. In the future, the Rust compiler can improve itself by providing such missed information. Fourth, we find most violations of Rust's safety rules can be fixed using safe code, and future research work can leverage this finding to automatically fix safety-rule violations.

Values beyond Rust. Our work can benefit other programming languages in two ways. First, we demonstrate how to statistically analyze Stack Overflow questions to identify programming challenges for Rust and use code constructs to describe when a safety rule is more confusing. Future work can leverage similar methods to detect programming difficulties for other languages. Second, we construct pairs of program variants by changing code constructs or involved grammar to compare survey participants' performance in a controlled way. Researchers and practitioners can use similar methods to build program variants in other programming languages for testing and learning purposes.

Threats to Validity. Similar to previous empirical studies and user studies, our findings need to be considered with our methodology in mind. They have several potential threats to their validity.

There are several internal validity threats. Our survey participants might not have been representative enough, they might have referred to Rust tutorials in the survey, malicious persons or online bots might have submitted responses, and both the study on Stack Overflow questions and the grading of open-ended questions were based on subjective assessments. We took several methods to ensure internal validity. First, we recruited a relatively large number of participants from multiple channels. Second, we explicitly required participants not to refer to external resources multiple times in the survey. Third, three authors inspected all responses and filtered out invalid ones together. Fourth, at least two authors studied each Stack Overflow question and graded each open-ended question.

There are two possible external validity threats to our study's validity. First, we mainly leveraged Stack Overflow questions for identifying programming challenges of Rust. Those questions could not be resolved by the questioners, and thus they are more difficult than programming errors in daily practice. We also agree that some of the programming challenges of Rust are never submitted to Stack Overflow and thus cannot be identified through studying Stack Overflow questions. Second, our survey was conducted on Qualtrics. Reading Rust code on Qualtrics is different from coding Rust in a real development environment. Thus, participants' performance in our study may not reflect their common behaviors.

6 RELATED WORK

User Studies on Rust. The Mozilla Rust team conducts annual surveys to understand Rust programmers' backgrounds and figure out ways to improve Rust. The survey in 2020 reported *lifetime* and *ownership* are the two most difficult topics for programmers to grasp [47], which aligns with our observations in Section 3. Zeng and Crichton [66] analyzed Rust-related posts and comments collected from online Rust communities and identified several obstacles to the adoption of Rust. Crichton [13] conducted a case study to show the challenges of interpreting Rust's compiler error messages. Abtahi and Dietz [1] conducted a laboratory study to examine the methods employed by programmers when they learn Rust. They found that online code examples and compiler errors were helpful to Rust learners. They also reported that sometimes Rust learners found compiler error messages hard to interpret because the messages were full of terminologies. Fulton et al. [17] identified benefits and challenges of adopting Rust through a semi-structured interview and an online survey. Our study differs from those existing ones in study goals. Specifically, we aim to identify programming challenges incurred by Rust's safety rules and pinpoint scenarios where a safety rule is more difficult to understand.

Empirical Studies on Rust Code. Researchers have conducted empirical studies to understand real-world Rust code from different points of view, like how unsafe code is used [3, 16, 35], how many Rust libraries depend on external C/C++ libraries [58], and the buggy code patterns of safety issues that bypass Rust's compiler checks [38, 64]. Those empirical studies focus on Rust programs that can be compiled. However, we focus on Rust programs that are rejected by the Rust compiler, because we aim to identify programming challenges imposed by Rust's compile-time checks.

Leveraging and Improving Stack Overflow. Stack Overflow is an open community for developers to ask technical questions and share their knowledge [56]. Previous researchers leveraged Stack Overflow data to understand real-world development problems [2, 14, 22, 59, 65, 67] and built tool to improve the usages of Stack Overflow [6, 21, 42, 63, 68]. However, there is no prior work on studying Rust-related Stack Overflow questions and our study in Section 3 is the first one to examine those questions.

Comprehending Compiler Errors. Researchers performed several studies to understand how programmers interpret compiler error messages for traditional programming languages (*e.g.*, C/C++) [4, 5]. However, there is no similar study for Rust. Rust features strict compile-time checks that depend on complex safety rules. Compiler error messages are critical feedback for programmers, especially when it comes to safety-rule violations. Thus, it is particularly important to study how Rust programmers comprehend compiler error messages and to improve error messages accordingly.

7 CONCLUSION

Rust conducts extensive static checks at the compile time to catch memory-safety and thread-safety issues. Given the increasing popularity of Rust, it is critical to understand the language's learning and programming challenges, especially those due to its safety checks and its underlying safety mechanism. For this purpose, we conduct the first comprehensive, empirical study on Rust-related Stack Overflow questions. We expect that our findings can guide the learning, programming, and compiler evolution of Rust. In addition, we further perform a survey on 101 Rust programmers and confirm many of our findings with significant confidence.

REFERENCES

[1] 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.

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In Proceedings of the 2019 International Conference on Software Engineering (ICSE '19). Montreal, QC, Canada.

[3] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? Proceedings of the ACM on Programming Languages 4, OOPSLA (2020), 1–27.

[4] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How should compilers explain problems to developers?. In Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE '18). Lake Buena Vista, FL, USA.

[5] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In Proceedings of the 39th International Conference on Software Engineering (ICSE '17). Buenos Aires, Argentina.

[6] Stefanie Beyer, Christian Macho, Massimiliano Di Penta, and Martin Pinzger. 2018. Automatically classifying posts into question categories on stack overflow. In Proceedings of the 2018 International Conference on Program Comprehension (ICPC '18). Gothenburg, Sweden.

[7] Steven Bird, Ewan Klein, and Edward Loper. 2009. Natural Language Processing with Python (1st ed.). O'Reilly Media, Inc.

[8] J Martin Bland and Douglas G Altman. 1995. Multiple significance tests: the Bonferroni method. BMJ 310, 6973 (1995).

[9] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. J. Mach. Learn. Res. 3 (March 2003), 993–1022.

[10] Mara Bos. 2021. The Plan for the Rust 2021 Edition | Rust Blog. https://blog.rust-lang.org/2021/05/11/edition-2021.html (Accessed on 09/01/2021).

[11] Richard E Clark, Carla M Pugh, Kenneth A Yates, Kenji Inaba, Donald J Green, and Maura E Sullivan. 2012. The use of cognitive task analysis to improve instructional descriptions of procedures. Journal of Surgical Research 173, 1 (2012), e37–e42.

[12] Nancy J Cooke. 1994. Varieties of knowledge elicitation techniques. International Journal of Human-Computer Studies 41, 6 (1994), 801–849.

[13] Will Crichton. 2020. The Usability of Ownership. arXiv:2011.06171 [cs.PL]

[14] Alex Cummaudo, Rajesh Vasa, Scott Barnett, John Grundy, and Mohamed Abdelrazek. 2020. Interpreting Cloud Computer Vision Pain-Points: A Mining Study of Stack Overflow. In Proceedings of the 2020 International Conference on Software Engineering (ICSE '20). Han River, Seoul, South Korea.

[15] Dan Diaper. 2004. Understanding task analysis for human-computer interaction. The handbook of task analysis for human-computer interaction (2004), 5–47.

[16] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is rust used safely by software developers?. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE '20). Han River, Seoul, South Korea.

[17] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In USENIX Symposium on Usable Privacy and Security (USENIX SOUPS 2021).

[18] Github#67651. 2019. Confusing/incorrect error message with incoherent implementations and async blocks. https://github.com/rust-lang/rust/issues/67651

[19] Github#71584. 2020. Wrong error message for missed type inference. https://github.com/rust-lang/rust/issues/71584

[20] Github#79429. 2020. Inaccurate error message for const operations in type parameters. https://github.com/rust-lang/rust/issues/79429

[21] Yi Huang, Chunyang Chen, Zhenchang Xing, Tian Lin, and Yang Liu. 2018. Tell Them Apart: Distilling Technology Differences from Crowd-Scale Comparison Discussions. In Proceedings of the 2018 International Conference on Automated Software Engineering (ASE '18). Montpellier, France.

[22] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20). Han River, Seoul, South Korea.

[23] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 12). Beijing, China.

[24] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. Biometrics (1977), 159–174.

[25] 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.

[26] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software. In Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID '06). San Jose, CA.

[27] Alexey Lozovsky. 2018. Rust vs C++ Comparison. https://www.apriorit.com/dev-blog/520-rust-vs-c-comparison

[28] Nicholas D Matsakis and Felix S Klock. 2014. II. 2014. The Rust Language. In Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT'14). 103–104.

[29] Laura G Militello and Robert JB Hutton. 1998. Applied cognitive task analysis (ACTA): a practitioner's toolkit for understanding cognitive task demands. Ergonomics 41, 11 (1998), 1618–1641.

[30] David Mimno, Hanna M. Wallach, Edmund Talley, Miriam Leenders, and Andrew McCallum. 2011. Optimizing Semantic Coherence in Topic Models. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '11). Punta Cana, Dominican Republic.

[31] Mozilla. 2017. Quantum - MozillaWiki. https://wiki.mozilla.org/Quantum

[32] Mozilla. 2020. Rust Compiler Error Index. https://doc.rust-lang.org/error-index.html

[33] Mozilla. 2021. Rust Programming Language. https://www.rust-lang.org/

[34] Nick Kolakowski. 2019. 10 Fastest-Growing Programming Languages on GitHub. https://insights.dice.com/2019/11/11/10-github-programming-languages/.

[35] Alex Ozdemir. 2019. Unsafe in Rust: Syntactic Patterns. https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax

[36] Parity Technologies. 2021. Parity Bitcoin Client. https://www.parity.io/bitcoin/

[37] Parity Technologies. 2021. The Parity Ethereum Client. https://www.parity.io/ethereum/

[38] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20). London, UK.

[39] Qualtrics. 2021. Qualtrics XM // The Leading Experience Management Software. https://www.qualtrics.com/

[40] Redox. 2020. Redox - Your Next(Gen) OS. https://www.redox-os.org/

[41] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks. ELRA.

[42] Anastasia Reinhardt, Tianyi Zhang, Mihir Mathur, and Miryung Kim. 2018. Augmenting Stack Overflow with API Usage Patterns Mined from GitHub. In Proceedings of the 2018 Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE '18). Lake Buena Vista, FL, USA.

[43] Rust. 2020. IntoIterator. https://doc.rust-lang.org/std/vec/struct.Vec.html#impl-IntoIterator-1

[44] Rust. 2020. split_first_mut. https://doc.rust-lang.org/std/primitive.slice.html#method.split_first_mut

[45] Rust Forum. 2017. Quick introduction to Rust for C++ programmers. https://users.rust-lang.org/t/quick-introduction-to-rust-for-c-programmers/13769

[46] Rust Survey Team. 2019. Rust Survey 2019 Results. https://blog.rust-lang.org/2020/04/17/Rust-survey-2019.html

[47] Rust Survey Team. 2020. Rust Survey 2020 Results. https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html

[48] Servo. 2020. Servo. https://servo.org/

[49] SO#62491845. 2020. Ownership: differences between tuples and arrays in Rust. https://stackoverflow.com/questions/62491845/ownership-differences-between-tuples-and-arrays-in-rust

[50] SO#65682678. 2021. Rust Closures concept. https://stackoverflow.com/questions/65682678/rust-closures-concept

[51] Stack Overflow. 2016. Stack Overflow Developer Survey 2016. https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted

[52] Stack Overflow. 2017. Stack Overflow Developer Survey 2017. https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted

[53] Stack Overflow. 2018. Stack Overflow Developer Survey 2018. https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted

[54] Stack Overflow. 2019. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted

[55] Stack Overflow. 2020. Stack Overflow Developer Survey 2020. https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted

[56] Stack Overflow. 2021. Stack Overflow - Where Developers Learn, Share, & Build Careers. https://stackoverflow.com/

[57] Stack Overflow. 2021. Stack Overflow Developer Survey 2021. https://insights.stackoverflow.com/survey/2021#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages

[58] Mingshen Sun, Yulong Zhang, and Tao Wei. 2018. When Memory-Safe Languages Become Unsafe. In DEF CON China (DEF CON China '18'). Beijing, China.

[59] Mohammad Tahaei, Kami Vaniea, and Naomi Saphra. 2020. Understanding Privacy-Related Questions on Stack Overflow. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20). Honolulu, HI.

[60] The Rust Core Team. 2021. Planning the 2021 Roadmap | Rust Blog. https://blog.rust-lang.org/2020/09/03/Planning-2021-Roadmap.html (Accessed on 09/01/2021).

[61] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19). Providence, RI.

[62] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An empirical study on challenges of application development in serverless computing. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE/ESEC '21). Virtual Event.

[63] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: Automated Generation of Answer Summary to Developers' Technical Questions. In Proceedings of the 2017 International Conference on Automated Software Engineering (ASE '17). Urbana-Champaign, IL, USA.

[64] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael Lyu. 2020. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. arXiv preprint arXiv:2003.03296 (2020).

[65] Xin-Li Yang, David Lo, Xin Xia, Zhiyuan Wan, and Jian-Ling Sun. 2016. What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts. Journal of Computer Science and Technology 31 (09 2016), 910–924.

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

[67] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In Proceedings of the 2018 International Symposium on Software Testing and Analysis (ISSTA '18). Amsterdam, Netherlands.

[68] Jing Zhou and Robert J. Walker. 2016. API Deprecation: A Retrospective Analysis and Detection Method for Code Examples on the Web. In Proceedings of the 2016 International Symposium on Foundations of Software Engineering (FSE '16). Seattle, WA.