`R: Towards Detecting and Understanding Code-Document Violations in Rust

Wanrong Ouyang Baojian Hua
School of Software Engineering
University of Science and Technology of China
oywr@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—Documentation and comments are important for any software project. Although documentation is not executed, it is useful for many purposes, such as code comprehension, reuse, and maintenance. As a project evolves, the code and documentation can easily grow out-of-sync, and inconsistencies are introduced, which can mislead developers and introduce new bugs in subsequent developments. Recent studies have shown it is promising to use natural language processing and machine learning to detect inconsistencies between code and documentation. However, it's challenging to apply existing techniques to detect code-document inconsistency in Rust programs, as Rustdoc supports advanced document features like document testing, which makes existing solutions inapplicable. This paper presents the first software tool prototype, `R, to detect and understand code-document inconsistencies in Rust. To perform such analysis, `R leverages static program analysis, not only on Rust source code, but also on document testing code, to detect inconsistency indicating either bugs or bad documentation. To evaluate the effectiveness of `R, we applied it to 37 open source Rust projects from 9 domains, with a total of 6,192,251 lines of Rust source code (with 322,330 lines of comments). The results of the analysis give interesting insights, for example: the cryptocurrency domain has the highest documentation ratio (58.23%), documentation testing is rarely used (ratio 2.30% on average) in real-world Rust projects in all domains, etc. Based on these findings, we propose recommendations to guide the construction of better Rust documentation, better Rust documentation quality detection tools, and boarder adoption of the language.**

***Index Terms*—Rust, Documentation, Inconsistency**

I. Introduction

Despite the costly efforts to improve software robustness and reliability, software bugs and vulnerabilities continue to contribute to a significant percentage of system failures. To address the problem, documentation and comments continue to be the standard software engineering practice to increase reliability and maintainability. As documentation and comment are direct, descriptive, and easy-to-understand, it is a primary resource for programmers to understand the behaviors of any software systems [36], [41], [46], [49]. Furthermore, although documentation is not executed, it is useful for many purposes, such as code comprehension, reuse, or maintenance [35], [37]. Unfortunately, as software projects evolve, the document and code are changed and it's easy for them to grow out-of-sync and violations, thus inconsistencies are introduced. Such violations indicate 3 possible vulnerabilities: 1) wrong documentation; 2) buggy code; or 3) both of them. Unlike buggy code, although wrong documentation may not introduce bugs immediately, but existing studies have demonstrated that they can mislead developers and introduce new bugs in subsequent development tasks [44], [45], [48].

Recently, there have been a significant number of studies on detecting code-document violations. Existing techniques can be classified into 3 categories: 1) *the natural language processing (NLP) technique* [18], [19], to automatically analyze comments (in natural language) and detect violations and inconsistencies between code and documentation; this technique has been very successful in C++, by combining natural language processing, machine learning and program analysis techniques to automatically analyze documentation and detect inconsistencies between documentation and source code; 2) *The machine learning-based classification technique* [9], to implement documentation classifiers; this technique has been used to detect doc-document inconsistencies in C++ and java; 3) *The program analysis technique* [4], [13], to detect violations in the documentation components, this technique requires the document is of detecting whether there are null values or mismatched values in each component of the Javadoc.

Unfortunately, it's challenging to apply existing techniques to detect code-document violations in programs of Rust, a novel programming language for safe system programming [1]. Besides normal comments like in any other language, Rust introduced document testing [17], a feature that has been proven to be important in trusted systems. The key insight of document testing is to allow programmers to write unit testing code in documents, and such code will get executed when testing is triggered. As documents contain Rust code, so it's impossible to apply existing NLP or machine learning techniques to detect violations, as these tools are designed to analyze comments in natural languages, not program code. And it's hard to use existing program analysis techniques, for they can only process structured simple documentation language, not arbitrary Turing-complete programming languages.

Furthermore, The standard Rust distribution ships with a tool called `rustdoc` [17], which generates documentation for Rust projects, displays statistics like public API documentation ratio, and runs code examples as tests. However, Rustdoc cannot detect code-documentation violations.

This paper presents the first system, **`R**, to detect code-

document violations in Rust projects. The key insight of **`R** is to analyze *both* the documentation testing code and the Rust code being annotated, with the same program analysis infrastructure. Implementing this tool is challenging, as Rust documentation supports running arbitrary code examples as tests. To address this challenge, we implemented an algorithm to modify and extend the compiler's abstract syntax tree data structures extensively, by leveraging the Rust compiler's powerful API support. It should be noted that **`R** is of key difference with `rustc`, which can only process Rust code; and with `rustdoc`, which can only process documentation.

To evaluate this tool, we first build datasets for experiments and analysis. We selected and collected 36 open source Rust projects from the 8 domains and also the `rustc` compiler, consisting of a total of 6,192,251 lines of Rust source code. The domains we selected cover a wide range of applications and represent typical usage scenarios for the Rust language.

We conducted experiments on the selected data sets, and have obtained interesting insights and findings by analyzing the experiment results. We analyzed crates and Rust language 7 items (only analyzed the public items), and found that none of these items were documented at a very high ratio. In the documented public items, the incomplete documentation ratio is high for all the public items, even the public functions that need to be tested are up to 79.22% in our projects, and the incomplete documentation ratio for the public function in Rustc is up to 96.51%. We find that the matched ratio in complete documentation is high for public components.

Finally, we analyzed Rust projects in different domains and found that projects in the cryptocurrency and OS domains adhered to the Rust documentation specification more than others.

Based on these insights and findings, we propose recommendations to write better documentation and improve the software maintainability. First, we make some suggestions to guide writing better documentation. Second, we propose some ideas to increase the functionality of existing documentation detection tools and to build a healthier Rust ecosystem. Overall, these recommendations will make Rust a more reliable language for system programming.

To our knowledge, this work is the first software tool prototype to detect and understand code-document violations in real-world Rust projects. To summarize, our work makes the following contributions:

- The first software tool prototype **`R** to detect and understand code-document violations in real-world Rust projects.
- Insights and findings that indicate the presence of documentation violations, and the analysis of the root causes.
- Suggestions to guide better documentation of the Rust language, and to improve the reliability of Rust programs.

The rest of this paper is organized as follows. Section II presents an overview of the background information about Rust documentation. Section III discusses the general research methodology by presenting the research questions which guide our experiments. Section IV presents the approach we used to perform the analysis. Section V presents the experiments we performed along with the data set we used, and the answers to the research questions based on the experiment results. Section VII discusses the related work, and Section VIII concludes.

II. Background and Preliminaries

This section presents an overview of the background information about Rust documentation and violation patterns.

A. Rust Documentation Testing

Rust [1], as a novel safe system programming language, provides well-designed support for documentation. On one hand, Rust inherits the successful documentation design from other languages like Java or Python, and offers a tool `rustdoc` to generate easy-to-read HTML documentation. Although such documentation feature is very conventional, it's rather mature and successful, and makes the learning curve smooth for new Rust programmers.

On the other hand, Rust documentation offers a novel feature called document testing, which is not provided by Java or Python. Technically speaking, document testing allows programmers to write unit testing in documentation. And the `rustdoc` can run the testing code once being triggered.

Consider the sample Rust program in Fig. 1, although

```
1  //! This crate provides functionality for
       adding things.
2
3  /// Adds one to the number given.
4  ///
5  /// # Examples
6  ///
7  /// ```rust
8  /// let arg = 5;
9  /// let answer = my_crate::add_one(arg);
10 ///
11 /// assert_eq!(6, answer);
12 /// ```
13 pub fn add_one(x: i32) -> i32{
14     x+1
15 }
```

Fig. 1. A Rust documentation testing example

this example is simple, it demonstrates key characteristics: a documentation testing starts with three slashes `///`, instead of two slashes `//` (which are normal comments and are not documentations). The documentation testing consists of 3 components: the `//!` represents module-level or crate-level documentation (line 1), followed by a general and short description of the code (line 3); the third component is called *examples* (starting with a `#` symbol at line 5) indicating the start of the testing code.

The testing code is grouped into a pair of ``` symbol, which is usually Rust source code (as the ```rust indicates at line 7). Technically, although any Rust code can be placed into this section, however, as this example demonstrates, it's normal practice to put just testing-related code in.

The official software tool `rustdoc` executes documentation examples as unit tests, which makes it easier to guarantee testing examples within documentation are up to date and working. For instance, suppose we modify the code at line 14 to `x + 2`, but unfortunately forget to modify the documentation correspondingly, the `rustdoc` will trigger the assertion at line 11, indicating the failure of the unit testing and the violations of the code-documentation.

B. Documentation Violations

Although the basic structure of documentation testing looks straightforward to follow, writing such testing is nontrivial due to the complex syntax Rust documentation allows. Thus, the Rustdoc specification [20] presents detailed rules and best practices for writing good and maintainable documentation. To be specific, the Rustdoc specification recommends that all public APIs should have documentation, which includes modules, structs, functions, and macros. Furthermore, each crate should have documentation.

Based on the Rust documentation specification, we classified the Rust code-documentation violations into 3 categories: 1) missing documentation; 2) incomplete documentation, and 3) mismatched documentation. Missing documentation indicates that a public item has no associated documentation at all. Incomplete documentation means that a public item has documentation but no testing example in documentation. These two violations have a common pattern of vulnerability: they miss the opportunity to unit test the target code in the first place. Finally, mismatched documentation indicates the testing example in documentation and the source code item do not match, that is, the testing code does not test the code the programmers intend to. It should be noted that this classification of code-documentation violations is not unique to Rust, existing studies on code-documentation violations for other languages like Java, etc. also makes use of a similar classification [13].

Mismatched documentation violation is worth further discussion. As a software project evolves, the code is often changed and inconsistencies are introduced with the documentation, which can mislead developers and introduce new bugs in subsequent development [2], [13]. Consider the sample Rust program in Fig. 1 again, suppose a developer changes the function name to `add_1`, but forgets to update the documentation correspondingly, such inconsistency as in Fig. 2 can easily introduce new bugs, as developers may assume the code has passed the unit test and thus is bug-free. Unfortunately, We have observed that `rustdoc` may not report any inconsistency for the sample program in Fig. 2, for there may be another function with the name `add_one`. It should be noted that this is not a unique limitation of `rustdoc`, for instance, we have also conducted experiments with Tokei [15], another popular program analysis tool for Rust, which also does not report any inconsistency.

```
1  /// ```rust
2  /// let arg = 5;
3  /// let answer = my_crate::add_one(arg);
4  ///
5  /// assert_eq!(6, answer);
6  /// ```
7  pub fn add_1(x: i32) -> i32 {
8      x + 1
9  }
```

Fig. 2. A Rust documentation testing which may contain violations

It's difficult to apply existing techniques to address the aforementioned challenge, for two key reasons: 1) it's difficult to apply the NLP or ML techniques, because we need to analyze the code fragment in documentation, but not comments written in natural languages; and 2) it's impossible to apply the existing program analysis techniques, as we need to compare deep properties of code and documentation, but not superficial syntax issues. For instance, if the function `add_one` calls `add_1`, then this code should not be considered as a violation, as the unit test in the documentation may eventually invoke the function `add_1` through the call chain.

III. Research Methodology

In this section, we present the research questions that guide our study, as well as the selection criteria for the Rust projects we chose and explored.

A. Research Questions

The main goal of our study is to conduct a large-scale study to detect the code-documentation violations in real-world Rust projects and to gain insights into these violations. Specifically, we focus on the following research questions:

RQ1: What is the general quality of Rust documentation?

We have selected some commonly used features, and by observing the documentation violations of these public features, we can understand the documentation violations of each project.

The motivation for RQ1 is to gain insight into the distribution of documentation violations in real world projects, and thus get an understanding of the overall quality of documentation in Rust projects.

RQ2: How is the documentation testing used in real-world Rust projects?

As the Rustdoc violations are divided into three categories: missing documentation, incomplete documentation, and mismatched documentation. The important feature that distinguishes Rustdoc from documentation in other languages is testability. We will focus on incomplete documentation violation and mismatched documentation violations in Rust projects.

The motivation for RQ2 is to understand how the testable examples in Rust documentation are used in Rust projects, and to understand whether there are inconsistencies between the examples and the source code.

RQ3: How is the documentation used in different domains?

Rust projects from different domains have very different requests for documentation. The goal for RQ3 is to understand whether there are differences in the distribution of documentation violations for projects from different domains.

B. Data Selection

To detect and understand code-document inconsistencies in real-world Rust projects, we analyzed publicly available Rust code in the real world. Three principles are guiding our selection of the Rust projects.

First, to cover as many scenarios as possible, we include as many domains as possible in our study. We analyze Rust projects from 8 different domains: databases, operating systems, gaming, image processing, cryptocurrency, security tools, system tools, and Web. These domains represent the most important scenarios Rust is used for.

Second, in each of these domains, we select as many representative Rust projects as possible. However, as with any open ecosystem, there is a long tail of projects in Rust that are small, largely unused, and which may not be used to detect code-document inconsistencies. Therefore, we analyze the popular and still-maintained projects in our data set. When we download these Rust projects from the central Rust repository and GitHub, we measure popularity by having a higher number of downloads or stars.

Finally, to compare the difference in documentation quality between real-word large Rust projects and Rustc, in this work we select the Rustc source code. Rustc is a compiler for the Rust programming language, which is provided by the project itself. The compiler will take the source code and generate binary code in the form of libraries or executables.

IV. Approach

This section demonstrates our approach to answering the research questions by analyzing the source code of these Rust projects and by detecting code-documentation violations.

A. The Architecture

The architecture of the **`R** software prototype is shown in Fig. 3. First, the front-end of **`R** parses the Rust source code

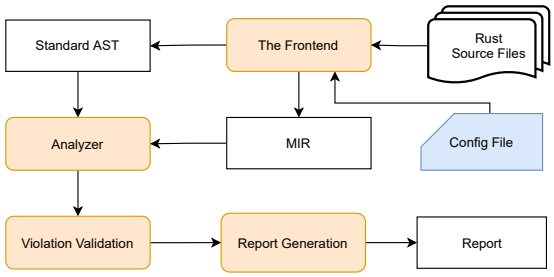

Fig. 3. **`R** Architecture

into both the AST and MIR data structures. Second, the analysis module analyzes both the AST and the MIR, to detect code-documentation violations of commonly used public features. Next, the validation module validates the generated violations. Finally, the statistics module generates statistics data which will be further processed by the data aggregation module to generate a code-documentation violation report.

In the following, we present some design and implementation details for each module.

B. The Frontend

The frontend of **`R** takes as input the Rust source files, and builds two intermediate representations: the Rust abstract syntax trees (AST) and the MIR. The AST is a tree representation of the source programs, especially, AST contains necessary documentation information for analysis in this work. The MIR is a control-flow graph (CFG) representation, in which each block is a sequence of statements. Blocks are connected by directed edges, which represent possible control transfers.

The design of **`R** differs from previous systems dramatically, in that it leveraged two IRs for Rust programs. The key consideration to make use of two IRs, instead of one, is that the documentation information is only kept on the Rust AST, and is erased before the AST is converted to lower MIR. On the other hand, our analysis engine makes use of the MIR extensively to build the call graph.

C. The Analysis Algorithm

Algorithm 1 takes as input a Rust program P, and calculates

Algorithm 1 : Calculating code-document violations

Input: P: The Rust program
Output: A set of code-documentation violations R

```
1:  procedure CAL-OVERFLOWS(P)
2:      R = ∅
3:      A, M = buildAstAndMir(P)
4:      G = buildCallGraph(M)
5:      for each public function f ∈ A do
6:          d = getDoc(f)
7:          if d == ∅ or incomplete(d) then
8:              R∪ = {f}
9:          for each call h(...) ∈ d do
10:             if h → f ∉ G then
11:                 R∪ = d
12:     return R
```

and returns a set of code-document violations in R. First, this algorithm builds an abstract syntax tree A and a MIR M as aforementioned in Section IV-A. The algorithm then builds a call graph G from the MIR M. The algorithm visits each public function f in the AST A to get its documentation. As the first two categories of violations (both the missing and incomplete ones) are not hard to detect, they will be added to the set R, once detected. The algorithm then analyzes each function call $h(\ldots)$ in the document d, if there is not a directed path $h \to f$ in the call graph G, then the documentation d is added to the set R.

This algorithm is efficient, it will build the AST, MIR, and call graph just once, and the examination of reachability $h \to f$ in the directed call graph G is quite fast, as the call graph for the program is of modest size, compared to the size of the AST or MIR.

D. Violation Validation

Like many static program analysis tools, the **`R** prototype is designed to be sound but not complete. To valid the soundness

of `R, we take a straightforward yet effective approach, we automatically insert a wrong assertion into each function that has been reported to contain violations. Then we run the official `rustdoc` tool to try to trigger this assertion. In all cases, this assertion should not be triggered, or else this is an implementation bug in `R.

E. Report Generation

Finally, the statistics module takes as input the information identifies each documentation violation type and finally records the node item type and violation type. The module ultimately generates a documentation violations report.

Since the standard Rust libraries are widely used in all Rust projects, we will ignore the Rust standard libraries when analyzing these projects, even though the Rust standard libraries can be handled by the same architecture. Because the Rust library is large, containing over 1,083,819 lines of Rust source code, such a design decision of `R speeds up analysis considerably.

V. Experiments and Results

This section presents our experiments and the analysis of the experimental results. We start with describing the experimental setup in Section V-A and continue with the detailed description of the data sets used in our experiments in Section V-B. We discuss the selected Rust features in Section V-C, and present the results by answering the research questions in Section V-D.

A. Experimental Setup

We execute the latest Rust compiler version 1.51.0. All these experiments and measurements are performed on a server with one 4 physical Intel i7 core (8 hyperthread) CPU and 8 GB of RAM running Ubuntu 18.04.

B. Data Sets

We have explained our data sets selection criteria in section III-B, here we give a detailed description of the selected domains and Rust projects in these domains.

The selected domains and projects are presented in Table I. There are 8 domains included: databases, operating systems, gaming, image processing, cryptocurrency, security tools, system tools, and Web. These domains are representatives of typical Rust application scenarios. In each domain, we give the the numbers of selected projects, the sizes of these projects (lines of source code), the numbers of Rust source files, the ratio of Rust source code, and the GitHub stars.

There are 3 to 7 projects respectively in each selected domains, and these projects are selected based on their importance and popularity in that domain, according to our data set selection criteria. Due to space limit, Table II presents five representative projects and the domains they belong to. The diversity of these projects is important in guaranteeing the fairness and precision of the experiment results.

Although not a perfect metric, we use the number of GitHub stars to measure the popularity of a Rust project. The top 4 domains with high stars on average are Web, Cryptocurrency, system tools, and databases, which stand for Rust's most popular usage scenarios. The project with highest GitHub stars (more than 75,000) is Deno [8], a secure JavaScript/TypeScript runtime.

Finally, we selected the Rustc source code for comparing the difference in documentation quality with real-world large-scale projects.

C. Rust Features

We have systematically studied the Rust documentation specification [17], to identify the language features which should have documentation. Table III shows the features we selected, which includes all the language features specified by the specification, except for the Impl feature, because we analyzed its internal public functions directly.

D. Results

We answer the research questions from Section III-A, based on the experimental results.

RQ1: What is the general quality of Rust documentation? Table IV shows the number of public features `#pub` (which should have documentations according to the Rustdoc specification), the number of documentations `#doc`, and the ratios for documentations $\frac{\#pub}{\#doc}$, for all the projects in our datasets. Although not a perfect metric, we believe that the ratio of documentation represents the general quality of Rust documentation. The RustScan project has the highest ratio of documentation (87.91%), and both the citybound and the dust projects do not contain any documentation and thus have the lowest ratio of documentation (0.0%). The `rustc` compiler and the substrate project have nearly the same number of documentations (2688 and 2686 respectively). The average documentation ratio for all projects is 39.83%, which is low.
Finding 1: *The document ratio in real-world Rust projects is not high in general. And smaller Rust projects tend to have lower document ratios.*

To get a better understanding of Rust documentation usage, we analyzed the documentation ratio for each public language feature in 36 real-world Rust projects. Fig. 4 presents the documentation ratios for these features. The trait feature has

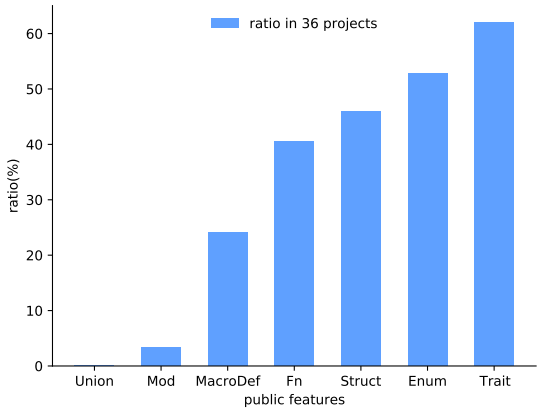

Fig. 4. Document ratio for public language features in 36 projects

TABLE I
THE SELECTED 8 DOMAINS OF RUST PROJECTS

Domains	Projects	LOC in Rust			No. of Rust Files			Ratio of Rust Code			GitHub Star		
		Avg.	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.	Min	Max
Database	4	114238.0	7139	258379	316.5	35	661	97.2	91.3	99.6	4075.0	1100	9200
Operating System	3	71014.0	13396	174924	307.0	117	676	98.3	97.2	99.4	1810.7	432	2800
Gaming	4	57313.0	11410	182129	225.5	49	671	93.9	84.1	99.7	2587.0	948	6400
Image Processing	3	14171.7	9370	20976	52.0	37	66	97.2	93.5	99.9	1565.3	996	2400
Cryptocurrency	5	171778.2	2351	336798	583.0	6	1338	79.2	2.1	99.7	6520.0	3200	15800
Security tools	4	11621.2	2154	28460	73.5	9	174	95.1	94.1	95.8	1729.5	667	3900
System tools	7	7920.9	1223	22681	36.7	4	94	96.3	89.7	99.9	5171.4	2300	13000
Web	6	39989.6	14723	66384	144.8	68	225	88.5	53.6	100.0	22820.0	6800	75000

TABLE II
REPRESENTATIVE PROJECTS

Projects	Domains	References
TiKV	a transactional and key-value database	[25]
Tock	an embedded operating system	[3]
Doom	a popular game renderer engine	[11]
Diem	a trusted financial network (A.K.A. Libra)	[12]
Servo	a Web browser engine from Mozilla	[22]

TABLE III
RUST FEATURES

Feature	Selected?	Description
Fn	✓	A function declaration
Mod	✓	A module declaration
Enum	✓	An enum definition
Union	✓	A union definition
Struct	✓	A struct definition
Trait	✓	A trait declaration
Impl	✗	An implementation.
MacroDef	✓	A macro definition.

the highest ratio (61.73%), whereas both the Union feature (0.02%) and the Mod feature (3.41%) have low ratio of documentation. We believe the root cause for this result is that the trait feature is used heavily in Rust, and thus get good documentations, whereas the Union and Mod features are used rarely.

Finding 2: *The public trait feature is of the highest documentation ratio, whereas the Union and Mod features have rather low document ratios.*

In addition, we need to understand how documentation testing are used in real-world projects and whether there are violations in the documentation testing examples, which will be explored in the next research question.

RQ2: How is the documentation testing used in real-world Rust projects? For Rust projects with documentation, Table V presents the incomplete documentation ratio, along with the mismatched documentation ratio in all Rust projects. The ratio for incomplete documentation is very high (96.81%), whereas the ratio for mismatched documentation is low (0.66%).

TABLE IV
DOCUMENTATION RATIOS IN ALL PROJECTS

Projects	#pub	#doc	doc ratio
rustc	8630	2686	31.12%
diem	3940	1507	38.25%
grin	708	483	68.22%
polkadot	1545	1135	73.46%
substrate	3764	2688	71.41%
zcash	73	27	36.99%
indradb	153	144	28.76%
materialize	1399	636	45.46%
noria	264	72	27.27%
tikv	2645	707	26.73%
citybound	498	0	0.00%
rust-doom	182	4	2.20%
veloren	2180	465	21.33%
zemeroth	250	17	6.80%
resvg	271	145	53.51%
svgbob	60	26	43.33%
svgcleaner	150	8	5.33%
kernel	152	74	48.68%
nebulet	673	196	29.12%
tock	330	70	21.21%
feroxbuster	2115	676	31.96%
RustScan	91	80	87.91%
sn0int	21	10	47.62%
sniffglue	695	3	0.43%
bandwhich	102	1	0.98%
dust	68	0	0.00%
exa	33	1	3.03%
fselect	111	0	0.00%
gitui	340	172	50.59%
lsd	75	27	36.00%
zoxide	36	6	16.67%
actix-web	390	261	66.92%
deno	899	152	16.91%
hyper	124	61	49.19%
Rocket	493	198	40.16%
servo	4268	1645	38.54%
zola	163	58	35.58%

Finding 3: *Documentation testing is rarely used in real-world projects.*

We carefully analyzed the root causes for this finding: the first reason is that large projects have dedicated test cases, so they do not use the Rust official documentation testing at all. For instance, the diem project from Facebook in cryptocurrency domain has a dedicated test suite based on docker. Another reason is documentation testing is under-appreciated and under-utilized. Many projects, for instance,

the bandwhich, fselect in system tools domain, do not have a dedicated test suite, but also use documentation testing rarely; for such projects, the code-documentation violations tend to occur.

TABLE V
INCOMPLETE AND MISMATCHED RATIO IN PROJECTS

Incomplete ratio	Mismatched ratio
96.81%	0.66%

To gain a deeper understanding of the use of documentation testing in Rust projects, we further study the ratio of the documentation testing used in our selected public feature except for union and mod, according to the results of the previous RQ1, the documentation ratio of both is inherently low. Table VI presents the distribution of public features' incomplete documentation ratio, along with the mismatched documentation ratio.

TABLE VI
INCOMPLETE AND MISMATCHED RATIO FOR PUBLIC FEATURES

Feature	Incomplete ratio	Mismatched ratio
Fn	96.51%	0.94%
MacroDef	100.00%	0.00%
Mod	96.73%	0.71%
Struct	98.88%	0.28%
Trait	97.65%	0.47%

We focused on the use of documentation testing in public fn, which requires documentation testing. We find that the ratio for incomplete documentation is high, but the ratio for mismatch documentation is low. This means that most code-documentations are consistent.

Finding 4: *The ratio for inconsistency between the code and the documentation is low.*

RQ3: How is the documentation used in different domains? Table VII presents the documentation ratios for crates and features in all projects.

TABLE VII
DOCUMENTATION USED IN DIFFERENT DOMAINS

Domain	Crate doc ratio	Feature doc ratio
Rustc	19.86%	31.12%
Cryptocurrency	43.29%	58.23%
Databases	22.33%	32.71%
Gaming	7.98%	15.63%
Image processing	3.75%	37.21%
Operating systems	57.83%	30.21%
Security tools	2.62%	10.34%
System tools	18.82%	34.36%
Web	25.48%	37.48%

The domains with the highest documentation ratios for crates are cryptocurrency and operating systems (43.29% and 57.83%, respectively). It is not surprising for this finding: both of these domains of high security requirements. The domain with the highest feature document ratio is still cryptocurrency (up to 58.23%), whereas `rustc` only has 31.12%.

Finding 5: *The cryptocurrency domain has the highest documentation ratios, both at crate-level and feature level.*

VI. TOWARDS BETTER DOCUMENTATION

In this work, we present the study of Rust code-documentation violations based on a large-scale study of the Rust projects, with the tool **`R**. Our analysis and results provide a basis for the future development of Rust language documentation, the construction of better Rust documentation analysis tools, and to promote border adoption of the language.

From the results of this work, we conclude that many projects may benefit from an increased documentation ratio. Thus, we recommend making more extensive use of documentation testing in any Rust projects. Recently, there have been a lot of researches on automatic documentation generation techniques [38]–[40], we believe that new IDE plugins can be developed to help generate Rust documents automatically.

For all existing rust documentation quality inspection tools [15], [17] only report per-crate documentation ratios. According to the Rust documentation specification, every public API should be documented. So we suggest that such tools should be improved to check for such violations.

For code-documentation inconsistency, the **`R** only detects inconsistency between the documentation testing examples and the code, and does not detect inconsistency between the documentation description part and the code. However, we believe that the inconsistency between the documentation description section and the code can be processed by existing techniques, such as natural language processing combined with machine learning [18].

Finally, as a relatively young language, Rust adoption is still growing rapidly [31], and in particular, the Rust documentation specification is not well adopted. To increase the adoption of Rust documentation (and the language in general), there should be a better specification, more easy-to-use tools, and more technical training.

VII. RELATED WORK

In recent years, the study of programming language documentation has received a lot of attention and there are many related studies. We divide these studies into two categories: documentation quality studies, and code-document inconsistency studies.

Documentation quality studies. Khamis et al. [10] introduced the JavadocMiner tool for analyzing the quality of Javadoc documentation. The tool evaluates the quality of the language used in documentation and the consistency between source code and documentation through a set of simple heuristics. However, the approach neither differentiates between different documentation types nor detects any inconsistencies between code and documentation beyond the structural requirements of Javadoc. Marcel [13] studied Javadoc violation, which focuses on the presence of Javadoc violation in source code items and whether specific Javadoc items are more likely to cause violations. Steidl et al. [9] performed documentation classification for Java and C/C++ programs based on machine

learning to distinguish between different types of comments. The model includes quality attributes for each documentation category based on four criteria: consistency across the project, completeness of the system documentation, consistency with the source code, and usefulness to the reader. However, none of these studies can be used for Rust documents because Rust documents are not composed for specific features, and Rust document testing presents a challenge for the application of these studies.

Code-document inconsistency studies. Tan et al. [4] proposed @TCOMMENT to test for inconsistencies between the items @param, @throws in Javadoc items and the method body, especially method attributes for null values and related exceptions. Raymond et al. [7] presented an algorithm for inferring conditions that may cause a given method to raise an exception. Importantly, the tool can be used to automatically generate documentation for the benefit of developers, maintainers and users of the system. Lin et al. [18] proposed a solution to automatically analyze documentation and detect inconsistencies between documentation and source code. The solution works by automatically analyzing documentation written in natural language to extract implicit program rules and uses these rules to automatically detect inconsistencies between documentation and source code, indicating incorrect or erroneous documentation. A combination of natural language processing (NLP), machine learning, statistical and program analysis techniques are ultimately used to achieve these goals. Then Lin et al. [19] designed and proposed aComment to detect concurrent errors in documentation and code related to interrupts in operating systems. The tool uses annotations extracted from code and documentation written in natural language and automatically propagates the documentation to the calling function to improve documentation and error detection. Svensson [14] presented comment validator to reduce the amount of outdated and inconsistent code comments. Stulova et al. [2] proposed a technique and a tool, upDoc, to automatically detect code-comment inconsistency during code evolution, by building a map between the code and its documentation, ensuring that changes in the code match the changes in respective documentation parts. However, the above tools can only detect inconsistencies between the documentation description section and the code, but not the documentation testing examples in Rust.

VIII. Conclusion

In this work, we present a software prototype **`R**, to detect the code-documentation violations in Rust programs. We conducted extensive experiments on real-world Rust projects using **`R**, and used the experiment results to answer the research questions. We found interesting findings and gained important insights for code-document violations. From these findings and insights, we make recommendations for guiding better Rust documentation, and for the construction of better Rust documentation checking tools.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work is supported by a graduate education innovation program of USTC under grant No. 2020YCJC41.

References

[1] Steve Klabnik and Carol Nichols. 2018. The Rust Programming Language. https://doc.rust-lang.org/stable/book/2018-edition/

[2] Stulova, N., Blasi, A., Gorla, A., and Nierstrasz, O. (2020, September). Towards Detecting Inconsistent Comments in Java Source Code Automatically. In 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 65-69). IEEE.

[3] Tock. 2019. Tock Embedded Operating System. https://www.tockos. org/

[4] Tan, S. H., Marinov, D., Tan, L., and Leavens, G. T. (2012, April). @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (pp. 260-269). IEEE.

[5] Zcash. https://github.com/zcash/zcash

[6] actix-web. https://github.com/actix/actix-web

[7] Raymond P. L. Buse, Westley Weimer: Automatic documentation inference for exceptions. ISSTA 2008: 273-282

[8] Deno. https://github.com/denoland/deno

[9] Steidl, D., Hummel, B., and Juergens, E. (2013, May). Quality analysis of source code comments. In 2013 21st international conference on program comprehension (icpc) (pp. 83-92). Ieee.

[10] Khamis, N., Witte, R., and Rilling, J. (2010, June). Automatic quality assessment of source code comments: the JavadocMiner. In International Conference on Application of Natural Language to Information Systems (pp. 68-79). Springer, Berlin, Heidelberg.

[11] Doom.https://github.com/cristicbz/rust-doom

[12] Libra. https://www.diem.com/en-us/

[13] Marcel Steinbeck, Rainer Koschke: Javadoc Violations and Their Evolution in Open-Source Software. SANER 2021: 249-259

[14] Svensson, A. (2015). Reducing outdated and inconsistent code comments during software development: The comment validator program.

[15] Tokei. https://github.com/XAMPPRocky/tokei

[16] Rocket. https://github.com/SergioBenitez/Rocket

[17] Rustdoc Book. https://doc.rust-lang.org/rustdoc/

[18] Lin Tan, Ding Yuan, Gopal Krishna, Yuanyuan Zhou: icomment: bugs or bad comments?. SOSP 2007: 145-158

[19] Lin Tan, Yuanyuan Zhou, Yoann Padioleau: aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. ICSE 2011: 11-20

[20] Guide to Rustc Development. https://rustc-dev-guide.rust-lang.org/

[21] Stefan Lankes, Jens Breitbart, Simon Pickartz: Exploring Rust for Unikernel Development. ACM Symposium on Operating Systems Principles (SOSP) 2019: 8-15

[22] Servo. The Servo Browser Engine. https://servo.org/

[23] TFS. https://github.com/redox-os/tfs

[24] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Tom Anderson: High Velocity Kernel File Systems with Bento. Computing Research Repository (CoRR) abs/2005.09723 (2020)

[25] TiKV. https://github.com/tikv/tikv

[26] TTstack. https://github.com/rustcc/TTstack

[27] Catalin Cimpanu. 2019. Microsoft to explore using Rust. https://www.zdnet.com/article/microsoft-to-explore-using-rust

[28] https://security.googleblog.com/2021/04/rust-in-android-platform.html

[29] https://security.googleblog.com/2021/04/rust-in-linux-kernel.html

[30] https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved

[31] https://insights.dice.com/2019/11/11/10-github-programming-languages/

[32] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, Philip Levis: The Case for Writing a Kernel in Rust. ACM SIGOPS Asia-Pacific Workshop on Systems (APSys) 2017: 1:1-1:7

[33] Parity. https://github.com/paritytech/parity-ethereum

[34] Mincheol Sung, Pierre Olivier, Stefan Lankes, Binoy Ravindran:Intra-unikernel isolation with Intel memory protection keys. Virtual Execution Environments (VEE) 2020: 143-156

[35] Fluri, B., Wursch, M., and Gall, H. C. (2007, October). Do code and comments co-evolve? on the relation between source code and comment changes. In 14th Working Conference on Reverse Engineering (WCRE 2007) (pp. 70-79). IEEE.

[36] Jiang, Z. M., and Hassan, A. E. (2006, May). Examining the evolution of code comments in PostgreSQL. In Proceedings of the 2006 international workshop on Mining software repositories (pp. 179-180).

[37] Vieira, C., Magana, A. J., Falk, M. L., and Garcia, R. E. (2017). Writing in-code comments to self-explain in computational science and engineering education. ACM Transactions on Computing Education (TOCE), 17(4), 1-21.

[38] Song, X., Sun, H., Wang, X., and Yan, J. (2019). A survey of automatic generation of source code comments: Algorithms and techniques. IEEE Access, 7, 111411-111428.

[39] Liang, Y., and Zhu, K. (2018, April). Automatic generation of text descriptive comments for code blocks. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 32, No. 1).

[40] Huang, Y., Huang, S., and Zhou, X. (2020). Towards automatically generating block comments for code snippets. Information and Software Technology, 127, 106373.

[41] Misra, V., Reddy, J. S. K., and Chimalakonda, S. (2020, March). Is there a correlation between code comments and issues? an exploratory study. In Proceedings of the 35th Annual ACM Symposium on Applied Computing (pp. 110-117).

[42] Shinyama, Y., Arahori, Y., and Gondow, K. (2018, December). Analyzing code comments to boost program comprehension. In 2018 25th Asia-Pacific Software Engineering Conference (APSEC) (pp. 325-334). IEEE.

[43] Padioleau, Y., Tan, L., and Zhou, Y. (2009, May). Listening to programmers—Taxonomies and characteristics of comments in operating system code. In 2009 IEEE 31st International Conference on Software Engineering (pp. 331-341). IEEE.

[44] He, H. (2019, August). Understanding source code comments at large-scale. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1217-1219).

[45] Rahman, M. M., Roy, C. K., and Keivanloo, I. (2015, September). Recommending insightful comments for source code using crowdsourced knowledge. In 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 81-90). IEEE.

[46] Ghosh, A., and Kuttal, S. K. (2018, October). Semantic clone detection: Can source code comments help?. In 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (pp. 315-317). IEEE.

[47] Salviulo, F., and Scanniello, G. (2014, May). Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals. In Proceedings of the 18th international conference on evaluation and assessment in software engineering (pp. 1-10).

[48] Panthaplackel, S., Nie, P., Gligoric, M., Li, J. J., and Mooney, R. J. (2020). Learning to update natural language comments based on code changes. arXiv preprint arXiv:2004.12169.

[49] Storey, M. A., Cheng, L. T., Singer, J., Muller, M., Myers, D., and Ryall, J. (2007, October). How programmers can turn comments into waypoints for code navigation. In 2007 IEEE International Conference on Software Maintenance (pp. 265-274). IEEE.

[50] Raskin, J. (2005). Comments are More Important than Code: The thorough use of internal documentation is one of the most-overlooked ways of improving software quality and speeding implementation. Queue, 3(2), 64-65.

[51] Tan, L., Yuan, D., and Zhou, Y. (2007, May). Hotcomments: how to make program comments more useful?. In HotOS (Vol. 7, pp. 49-54).

[52] Spadini, D., Çalikli, G., and Bacchelli, A. (2020, October). Primers or reminders? The effects of existing review comments on code review. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE) (pp. 1171-1182). IEEE.

[53] Corazza, A., Maggio, V., and Scanniello, G. (2015, August). On the coherence between comments and implementations in source code. In 2015 41st Euromicro Conference on Software Engineering and Advanced Applications (pp. 76-83). IEEE.

[54] Miyake, Y., Amasaki, S., Aman, H., and Yokogawa, T. (2017). A replicated study on relationship between code quality and method comments. In Applied Computing and Information Technology (pp. 17-30). Springer, Cham.