The Open University

Open Research Online

Citation

Li, Chunmiao; Yu, Yijun; Wu, Haitao; Carlig, Luca; Nie, Shijie and Jiang, Lingxiao (2024). Unleashing the Power of Clippy in Real-World Rust Projects. In: IEEE/ACM 46th International Conference on Software Engineering, 14-20 Apr 2024, Lisbon, Portugal.

URL

https://oro.open.ac.uk/96391/

License

(CC-BY-NC-ND 4.0) Creative Commons: Attribution-Noncommercial-No Derivative Works 4.0

https://creativecommons.org/licenses/by-nc-nd/4.0/

Policy

This document has been downloaded from Open Research Online, The Open University's repository of research publications. This version is being made available in accordance with Open Research Online policies available from Open Research Online (ORO) Policies

Versions

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding

Unleashing the Power of Clippy in Real-World Rust Projects

Chunmiao Li
Huawei Beijing Research Center
chunmiaoli1993@gmail.com

Yijun Yu
The Open University
y.yu@open.ac.uk

Haitao Wu
Huawei Waterloo Research Center
haitao.wu@huawei.com

Luca Carlig
Huawei Ireland Research Center
luca.carlig@huawei.com

Shijie Nie
National Institute of Informatics
nieshijie2011@gmail.com

Lingxiao Jiang
Singapore Management University
lxjiang@smu.edu.sg

ACM Reference Format:
Chunmiao Li, Yijun Yu, Haitao Wu, Luca Carlig, Shijie Nie, and Lingxiao Jiang. 2024. Unleashing the Power of Clippy in Real-World Rust Projects. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3639478.3643096

1 INTRODUCTION

The error messages generated by the Rust compiler (rustc) are useful for developers to identify and diagnose suspicious code segments. Complementing the compiler, linters can also play an important role in promoting the adherence to certain coding style conventions and best practices. Prominent linters utilized in the Rust ecosystem include Clippy [1] and Rustfmt [2]. Among them, the Rust community particularly emphasizes on the importance of heeding the warnings provided by Clippy to mitigate common errors and promote the adoption of idiomatic conventions. Clippy provides a set of more than 600 lints in addition to the built-in rustc lints. These lints are divided into nine distinct categories that address correctness and style aspects. Each category is assigned a default lint level, namely Allow, Warn, or Deny, indicating the severity with which the lints are reported.

Nevertheless, there is a conspicuous **absence** of research examining the impact of Clippy in real-world Rust projects, despite similar studies being conducted for other programming languages like Python [6] . Consequently, the distribution of warnings and the prevalence of specific warnings in idiomatic projects remain unclear. It is important to emphasize that the adoption of varying conventions by different projects may lead to the selection of distinct subsets of available lint rules for gate-keeping purposes. Therefore, it is imperative to undertake such studies to provide valuable guidance and insights for developers, ultimately improving code quality and best practices within the Rust ecosystem. Moreover, investigating developers' perceptions and responses to Clippy's warnings would be invaluable in comprehending the practical application of Clippy and promoting its widespread adoption. Similar studies [3] conducted for other linters have provided helpful insights, but **none** have specifically addressed how developers evaluate the effectiveness of Clippy's warnings in their projects and how they treat these warnings. Additionally, the types of Clippy warnings that developers typically configure or prioritize in their projects remain uncertain. Understanding these preferences and configurations is crucial in optimizing Clippy's impact and catering to developers' specific needs and coding conventions. Lastly, exploring lint violations and their resolutions is not an unfamiliar research direction [4], it is interesting to study the measures to fix Clippy warnings. While Clippy is more capable of identifying and raising alerts for security threats than rustc, its reports may not be accurate or complete.

To address these unresolved problems, we first conducted a landscape study to analyze the distribution of Clippy warnings across all projects hosted at crates.io, along with a longitudinal survey examining the evolution of warnings in four representative official Rust projects. Our observations revealed a highly skewed distribution of warnings, with over 50% of warnings attributed to the top-1 lints, and over 80% of warnings identified by the top-5 lints. This raised uncertainty regarding the automatic resolution of these high-frequency warnings. Furthermore, we observed that certain projects clearly adopt gate-keeping practices, incorporating Clippy as part of their CI/CD process to ensure code with minimal warnings before committing, while others do not follow such practices. Consequently, the rationale behind adopting Clippy as a gate-keeping step, particularly concerning its auto-fix capabilities, remained ambiguous.

Following on these observations, we performed a user survey on the usage of Clippy in real-world Rust projects. The responses revealed that Clippy is indeed regarded as a valuable linter. However, Clippy is frequently not selected as the gate-keeper in continuous integration due to the high false positives and limited automated fixes for most warnings.

Finally, to foster the auto-fix capabilities for Clippy warnings, we proposed three promising strategies through the combination of rule transformation, direct integration of fix operations into Clippy, and the development of specialized shell script. They offer effective means for developers to remove warnings. It is worth noting that, to the best of our knowledge, we are the **first** to tackle this specific issue. As a result, the warning density in Rosetta benchmarks has significantly decreased from 195/KLOC to an impressive 18/KLOC, already lower than the average density (21/KLOC) of the crates-io Rust projects.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0502-1/24/04
https://doi.org/10.1145/3639478.3643096

2 METHODS AND RESULTS

RQ1. Data Mining *How many warnings exist in the Rust development landscape? How are they distributed over warning types? Are they fixed by developers?*

We developed a tool to detect warning messages of types. The analysis of over 8 million warnings from the crates.io code landscape reveals a significant imbalance in the distribution of warning types. Additionally, the longitudinal study highlights that while some projects have made commendable efforts in reducing Clippy warnings, others have chosen to overlook these warnings, raising questions about the adoption of best practices.

RQ2. User Survey *Is Clippy considered a useful linter in real-world projects? What are the key factors that obstruct the widespread adoption of Clippy?*

We conducted a user survey with aim to answer this. Totally, we gathered a total of 31 responses from developers with varying levels of expertise in rust programming, as self-assessed by the participants. From their responses, we found two key points. Clippy is regarded valuable for real-world projects, especially for adhering to coding practices and style guides. However, its widespread adoption is mainly hampered by false positives and less effective warning auto-fix techniques.

RQ3. Auto Fixes *How to fix high-frequency Clippy lints to reduce the warning density for benchmarks?*

We present three viable approaches to effectively mitigate the occurrence of the four most frequent types of warnings found in the Rosseta Rust dataset, as identified in CRustS [5]. The first approach involves the utilization of rule transformation, wherein we manually constructed TXL rules to reduce the arithmetic_side_effects warnings. Subsequently, we propose a modification to Clippy's existing functionalities, aiming to seamlessly integrate a warning fix operation directly into Clippy itself to fully eliminate default_numeric_fallback warnings. Thirdly, we devised shell commands to alleviate the two common types of warnings: undocumented_unsafe_blocks and missing_debug_implementations. [1]

We collected the number of warnings using Clippy, the total lines of code, and computed the KLOC for three Rust variations of the Rosseta dataset: c2rust, CRustS, and the refactored data using our three automated approaches. The performance change statistics are presented in Table 1, which demonstrate the effectiveness of our warning fixes, reducing the number of warnings from 9704 in c2rust to 812 using our methods. Particularly noteworthy is the significant decrease in warning density from 195/KLOC in c2rust to an impressive 18/KLOC after applying our methods, already lower than the average density of 21 KLOC observed in crates-io Rust projects. We carefully examined 100 programs (around 1/3 of all programs) to ensure their functional correctness. As we move forward, we plan to expand this verification process to include more programs for a thorough examination.

Table 2 presents how our approach had a significant impact on the top four common types of warnings before and after using the CRustS dataset. It was exciting to witness the substantial effectiveness of our approach in reducing warnings on the CRustS baseline, even though the dataset already had fewer warnings compared to c2rust. All four types of warnings experienced significant decreases. We observed that CRustS improved the safety of c2rust by sinking the "unsafe" on function identifiers into the function body, which reduced the ratio of unsafe functions but increased the ratio of unsafe blocks. As a result, the "undocumented_unsafe_blocks" warnings from CRustS (806) were more than those from c2rust (325). However, our script managed to greatly decrease this type of warning (5), further enhancing the safety of CRustS by alleviating warnings. Moreover, we successfully eliminated all instances of the default_numeric_fallback warnings, which showcases the inspiring effectiveness of our study.

[1]This lint belongs to rustc compiler's lints.

	Warnings	Lines	KLOC
c2rust	9704	49791	195
CRustS	6885	43863	156
Our approaches	**812**	43669	**18**

Table 1: Clippy-reported warnings data for three types of Rust codesets: c2rust, CRustS, and our approaches. Our methods significantly reduced warnings to only 18 KLOC, compared with 195 (c2rust) and 156 (CRustS).

	c2rust	CRustS	Our methods
arithmetic_side_effects	2480	1919	682
default_numeric_fallback	5789	3842	0
undocumented_unsafe_blocks	325	806	5
missing_debug_implementations *(rustc lints)	197	160	8

Table 2: Numbers regarding the top four most frequently occurring warnings in three Rust codesets (c2rust, CRustS, and our approach). Our methods effectively reduced warnings, especially eliminating default_numeric_fallback.

3 CONCLUSION

In this work, we have conducted the `first` study of Clippy's impact to real-world Rust projects, unleashing the distribution of warnings and revealing user feedback to Clippy. As a result, we proposed three effective solutions to fix Clippy warnings that could not be automatically removed via its auto-fix functionality, which foster Clippy's wider adoption to enhance the safety and idiomaticity of Rust programs.

REFERENCES

[1] 2023. Clippy. https://doc.rust-lang.org/clippy/. Accessed: 2024-01-15.
[2] 2023. Rustfmt. https://github.com/rust-lang/rustfmt. Accessed: 2024-01-15.
[3] Nathaniel Ayewah and William Pugh. 2008. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*. 1–5.
[4] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.
[5] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James Cordy, and Ahmed Hassan. 2022. CRustS: A Transpiler from Unsafe C to Safer Rust. (2022).
[6] Naelson Oliveira, Márcio Ribeiro, Rodrigo Bonifácio, Rohit Gheyi, Igor Wiese, and Baldoino Fonseca. 2022. Lint-Based Warnings in Python Code: Frequency, Awareness and Refactoring. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 208–218.