Trusting Code in the Wild:

Exploring contributor reputation measures to review dependencies in the Rust ecosystem

Literature Review

August 19, 2024

Muhammad Hassnain

Questions

- How do developers choose to review dependencies in the Rust Ecosystem?
- How can network centrality measures, as proxies for contributor reputation, signal the need to review dependencies?

Background

- **Upstream Dependency** : Packages we import in our projects.
- **Downstream Dependents** : Packages that include upstream dependencies.

“Part of trusting code from others also involve determining if someone is trustworthy”

Network centrality measure as a proxy for contributor reputation.

Developer Social Networks

Methodology

1) **Rust Developer Network**
   - 1000 most downloaded packages and their dependency packages.
   - Total of 1724 packages.

TABLE 1: Overview of the collected data to construct a developer network in Rust.

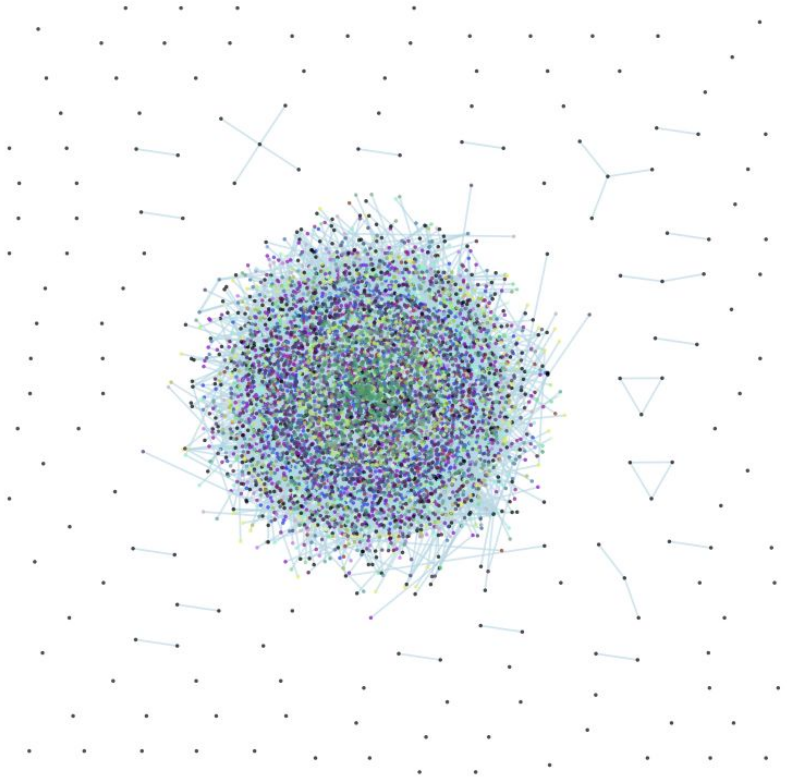
Crates.io	1,644
Repositories	1,088
Commits	109,512
Reviewed Commits	57,592 (52.6%)
Rejected Pull Requests	2,975
Developers	6,949
Authors	6,616 (95.2%)
Reviewers	2,891 (41.6%)
Relationships	26,448
File Co-edition Relationships	18,461 (69.8%)
Author-Reviewer Relationships	14,363 (54.3%)

Methodology

1) Rust Developer Network

TABLE 2: Rust developer network structural overview.

Vertices	6,949
Edges	26,448
Components	132
Isolates	111
Network density	0.001
Avg. clustering coefficient	0.398
Total communities	164
Communities with more than a hundred vertices	21
Largest Component	
Vertices	6,789 (97.7%)
Edges	26,417
Avg. shortest path length	3.674



Methodology

1) Rust Developer Network

Centrality Measures → indicate and rank central actors within a network

Metric	Definition	Interpretation
Degree centrality [71]	The fraction of the total vertices in the network a vertice is connected to.	The extent of direct collaboration with other contributors indicates a contributor's number of direct interactions.
Closeness centrality [72]	The reciprocal of the average shortest path distance from the vertice to all other reachable vertices in the network.	The reciprocal average number of collaboration relationships needed to reach other contributors indicates how quickly a contributor can reach the entire community.
Betweenness centrality [73]	The fraction of all pairs of vertices in the network whose shortest paths pass through that vertice.	The number of contributors that pass through the contributor as their shortest collaboration path to others indicates how often a developer acts as an intermediary.
Eigenvector centrality [74]	The centrality of a vertice is calculated based on the centrality of its neighbors.	The direct collaboration with higher-ranked contributors in the network indicates contributor influence.
PageRank [75]	A variation of eigenvector centrality, using the number and quality of edges.	The contributors who engage in higher-ranked collaborations within the network indicate contributor influence.

Methodology

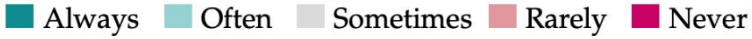
2) Survey

ID	Question	Options
SC1	Please rate the below statements based on how you review the incoming upstream code changes in your dependency packages (when adding or updating a package). — The question presents six statements which are listed in Table 5 when discussing the findings.	Always, Often, Sometimes, Rarely, Never
SO1	What other factors impact how carefully you review the upstream code changes before adding/updating a package?	*Open-ended*
SC2	In the Rust package ecosystem context, choose one of the five options for the below-listed GitHub users. — The respondent is presented with a list of ten GitHub users who are sampled separately for each respondent. The sampling strategy is explained in Section 3.2.2.	1. I have never heard of this person before. 2. I recognize this name, but I don't know much about them. 3. I know this person, but I don't know anyone who has worked with them. 4. I know this person, and I have worked with people who have worked with them. 5. I have directly worked with this person.
SC3	When adding or updating a package, how carefully would you review upstream code changes coming from the below-listed GitHub users? Please choose one of the four options based on your knowledge/association with them. Note that the code changes are within the dependency packages of your project. For example, when updating a package, how do you review the new changes that come with the update? — The respondent is presented with the same list of ten GitHub users from SC2.	1. I would not include code from this person in my project. 2. I would review each line of upstream code change coming from this person. 3. I would skim through the upstream code changes coming from this person. 4. I do not feel the need to review the upstream code changes coming from this person.
SO2	Can you explain your reasoning if you have chosen the same option for every user? Skip if not applicable.	*Open-ended*
SO3	Please add anything more you have to say.	*Open-ended*

RQ1 : Developer Dependency Review

TABLE 5: Likert scale responses on how strategies impact review process of upstream code for SC1.

ID	Statement	SC1 Distribution
SC1-1	I review the incoming upstream code changes before adding/updating a package.	24% 40%
SC1-2	I look at the authors and reviewers [6] of the incoming upstream code changes before adding/updating a package	29% 49%
SC1-3	The authors'/reviewers' [6] reputations within the Rust ecosystem impact how carefully I review before adding/updating a package.	51% 26%
SC1-4	My past collaborations with the authors/reviewers [6] impact how carefully I review before adding/updating a package.	45% 35%
SC1-5	My familiarity with the authors'/reviewers' [6] past work impacts how carefully I review before adding/updating a package.	50% 28%
SC1-6	I do not review the incoming upstream code changes before adding/updating a package.	46% 33%



RQ1 : Developer Dependency Review

TABLE 6: Top seven reasons developers review or not dependencies mentioned in SO2 and SO3 with their SC1 answers.

Codes (# Respondents) – Summary description	SC1-1 Distribution	SC1-3 Distribution
Light skim updates (44) – While reviewing a dependency update, the developer lightly skims the changes (e.g., changelog).	17% 46%	44% 32%
Review the same for everyone (36) – The developer treats every contributor the same during review.	30% 24%	41% 41%
Trust ecosystem (32) – The developers trust and depend on the ecosystem to provide security guarantees for dependencies. For example, dependency maintainers will review the dependencies.	12% 53%	37% 47%
Project context-dependent (32) – The level of review of a project varies on the context of the project (e.g., toy project, security-critical dependency).	7% 66%	36% 43%
Hard to review dependencies (29) – The dependencies are hard to review for developers due to reasons including the number of dependencies, time available, and tools.	18% 64%	46% 40%
Review when adding a dependency (28) – Once a dependency is reviewed during adoption, a lower level of review occurs for updates. Factors considered are mentioned in SO1.	4% 52%	45% 41%
Decide review level based on reputation (26) – The level of review considers contributor reputation, with respondents mentioning more relevance of the maintainers or leads reputation.	26% 31%	47% 21%

Always Often Sometimes Rarely Never

RQ1 : Developer Dependency Review

TABLE 7: Top 10 non-contributor reputation dependency review factors mentioned in SO1.

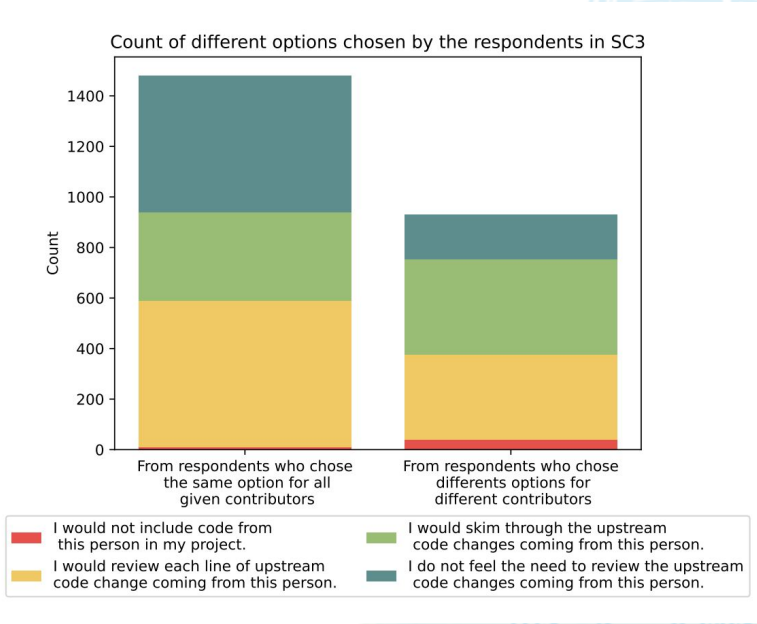
Factor	# Respondents
Popularity	52
Usage	27
Project context	25
Size	24
Security implication	21
Code quality	19
API changes	18
Maintenance	18
Changelog	16
Community	16

Review based on Contributor Identity

→ 241/285 responses were complete.

→ 148 (61%) choose same option for all developers.

“Respondents who do not consider identity are less likely to review code, those who consider are more likely to skim.”

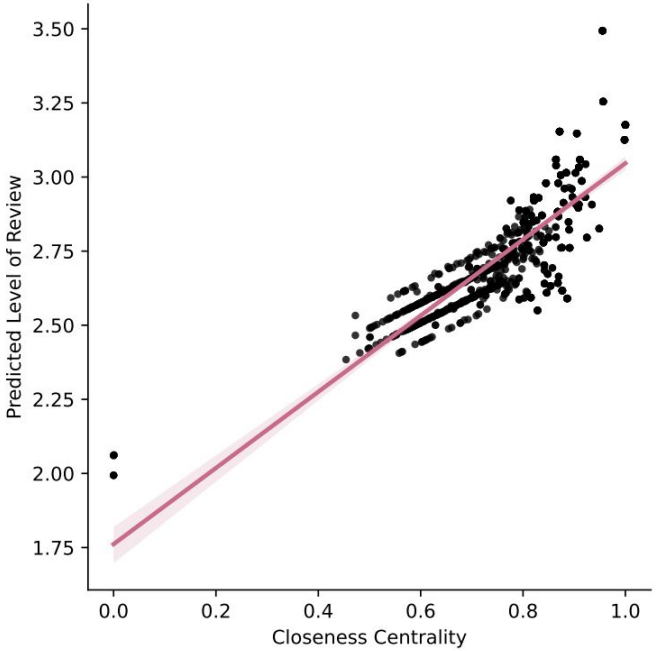


RQ2 : Network Centrality Signals

→ Measure the influence of network centrality measures on the level of dependency review employed.

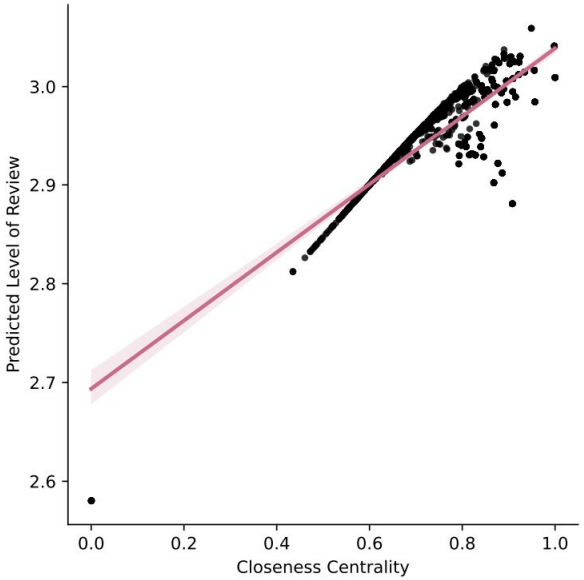
→ “engaging in highly-ranked collaborations within the network leads to less review from downstream”

→ “collaborating with highly-ranked contributors in the network leads to a higher level of review from the downstream”

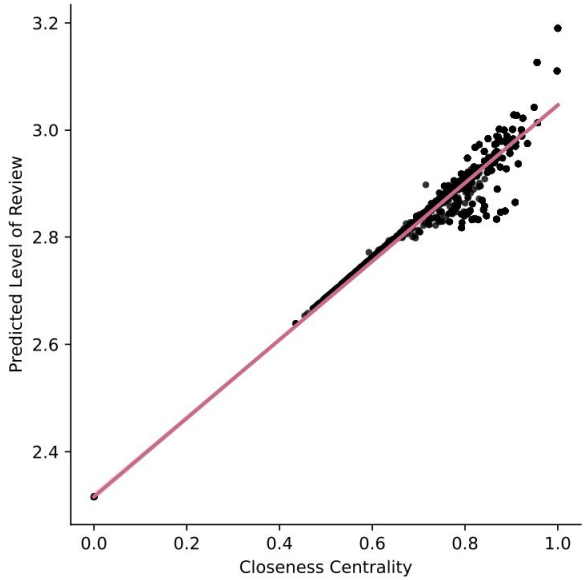


(a) Different level of review

RQ2 : Network Centrality Signals



(b) Same level of review



(c) All

Discussion / Suggestions

→ Ecosystems should provide a contributor reputation badge.

→ Developers should consider contributor reputation badges when reviewing dependencies

→ Researchers and companies should assist developers in reviewing dependencies

Limitations

→ Network centrality measures are limited.

- People can make mistakes.
- Accounts can be compromised.

→ Might not have incorporated all developer relations etc.