# Poster: Unsafe Rust – Conscious Choice or Spiky Shortcut?

Sandra Höltervennhoff[†], Philip Klostermeyer[†], Noah Wöhler[§], Yasemin Acar[‡], Sascha Fahl[§]

[†]Leibniz University Hannover, Germany, {hoeltervennhoff, klostermeyer}@sec.uni-hannover.de
[§]CISPA Helmholtz Center for Information Security, Germany, {noah.woehler, sascha.fahl}@cispa.de
[‡]George Washington University, USA, acar@gwu.edu

***Abstract*—The Rust programming language features guarantees regarding memory safety, which get enforced during compile time. So called unsafe Rust partly lifts guarantees to enable a programmer to write code which cannot be verified by a compiler to be memory safe, thus, creating critical points during development susceptible to serious security bugs. In this project, we investigate the usage of unsafe Rust by conducting qualitative, semi-structured interviews with Rust developers on (1) how the process of coding unsafe Rust looks like, (2) whether they reconsider the usage of this critical (and potentially dangerous) part of the language, and (3) how they verify the code's correctness and soundness.**

**The interviews coded so far show that the awareness of the potential for security bugs, when using unsafe Rust, can be considered high among participants, but also that there are still misunderstandings about what unsafe Rust does technically. As for help in writing secure unsafe Rust, no procedures were found besides official documentation. Mostly, the lively exchange in Rust communities and personal experiences help developers to evaluate their unsafe code fragments.**

***Index Terms*—Rust, Unsafe Code, Interview Study**

## I. Introduction

Rust is a more recently emerged multi-paradigm programming language with its first stable release in 2015, designed so that specific errors, such as memory and concurrency bugs, are prevented. This concept seems well-received. The usage of Rust has slightly increased in recent years, as seen in the *Most Popular Technologies* section of the Stack Overflow developer survey, and it has held the 1st place as the *Most Loved Language* for six consecutive years now[1]. However, not all concepts or use cases can be implemented using secure *safe* Rust code. As Rust also offers low-level programming functionalities, developers have to resort to so-called *unsafe* Rust to interact with components on a system- and hardware-level, where the guarantees of the language cannot be ensured anymore. In these unsafe code fragments, critical security vulnerabilities can still arise if developers act carelessly. Accompanied by the increasing popularity of Rust, it is necessary to closely examine how unsafe code is understood, used, and verified for security. In this project, we conduct interviews to learn about the developers' mindset around unsafe code and to understand what processes in connection to unsafe code look like in real-life scenarios.

When starting functions, methods, or code blocks with the `unsafe` keyword, the developer can use so-called *"unsafe superpowers"*[2], that would otherwise be rejected by the compiler. Within these blocks, e. g., dereferencing pointers could lead to undefined behavior and security issues at runtime.

The philosophy behind Rust is to put the responsibility that comes with the usage of unsafe Rust into the hands of developers to verify their code for correctness, safety, and security and to insert appropriate documentation.

We structure our work into four main research questions, examining several sub-questions in greater detail as follows.

- **RQ1.** *How is unsafe code perceived and utilized?* We want to identify typical use cases of unsafe Rust, how decisions about the usage of unsafe Rust are made, and what aids developers in writing secure unsafe code.
- **RQ2.** *Are safe and unsafe code understood correctly?* Following up, we want to understand how unsafe code is perceived and what opinions about unsafe Rust exist.
- **RQ3.** *How is unsafe code verified for security/safety?* Here, we are interested to learn how testing and code reviewing in Rust, specifically in unsafe parts, are carried out. This also included tools used for these purposes.
- **RQ4.** *What are the consequences when unsafe code is used incorrectly?* Lastly, we want to determine actual experiences where unsafe code has led to troubles or exploits during development or in a later stage of a project's life cycle.

## II. Methodology

In this section, we present our approach for the conducting the interviews, the recruitment process, and for the analysis of the transcribed data.

### A. Interview Procedure

To address the RQs, we decided to conduct semi-structured interviews. The interviews are conducted online and offered in German or English to reach more participants. We aim for a maximum duration of one hour for each interview. After each interview, we discuss the interview guide and make minor adjustments, if appropriate, to improve its quality. We also conducted three piloting interviews to verify that our guide is suitable and that the RQs could be answered.

[1] https://insights.stackoverflow.com/survey/2021

[2] https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

### B. Recruiting

For the main interview, we searched for Rust developers with at least some experience in using unsafe Rust. To identify possible participants, we used the GitHub API to search for contributors to repositories which can be found on the "*Awesome Rust*"[3] list – a curated list of Rust repositories. We restricted the search to developers whose contribution included code fragments with unsafe Rust and who had a public contact email address on their profile. This resulted in 860 possible contacts. Of these 860, 50 people were randomly chosen and were invited to our study, leading to seven interviews by the time of writing. All participants were compensated with an $80 Amazon voucher.

### C. Analysis

For evaluating the data, we first transcribe the interviews and code them in an open coding approach. Following the pilot interviews, we established an initial codebook, containing the labels for topics and themes that emerged during the interviews. In a first step, each interview is coded independently by two researchers. During the second step, conflicts are resolved and, if necessary, the codebook is adjusted. Currently, we are coding the remaining interviews using this initial codebook. We plan to contact more developers to create a more diverse sample.

## III. Findings

In the following, we want to present a preliminary look into early findings from our ongoing process of coding and analyzing interview data. As a limitation, it has to be made clear that these findings are hand-picked examples by the authors, are by no means exhaustive and might be subject to change in their relevance after we gain deeper insights through more interviews. The results refer to the three pilot interviews and the seven interviews conducted thereafter, from which quotes were taken.

**Unsafe Rust Perception and Understanding.** When asked how the participants would describe unsafe Rust in their own words to understand how they perceive it, we received largely varying answers, ranging from the existence of the specific *"unsafe superpowers"* to the assumption that unsafe is "*C mode in Rust*" and all checks are turned off. This suggests that there still might be some misunderstanding of what exactly happens within unsafe Rust and a knowledge gap between developers.

**Unsafe Rust Usage.** A clear majority of the participants generally try to avoid unsafe code if possible and try to use safe Rust instead. One participant used unsafe code only for the Foreign Function Interface because there is no alternative to it. When confronted with the question in what situations unsafe code could be preferable, performance was mentioned but also situations where the criticality of the concerned code passages was perceived as lesser or irrelevant: "*I wouldn't want my unsafe code to run my pacemaker.*". Whenever unsafe code cannot be avoided, good documentation and isolating unsafe code into small functions up to whole external crates could be identified as dominant strategies. Participants often tried to provide a safe interface. The answers suggest a hesitant but also reflected attitude around the how and when of using unsafe Rust.

**Guidance.** Talking about policies, aid and guidelines, apart from the official documentation, the Rustonomicon[4] was utilized most often. Participants also asked questions in channels like Discord or Reddit. No project or employer-issued standardized guidelines were mentioned. This suggests that developers are often on their own when programming unsafe code but know of channels they can turn to when in doubt. This is also reflected in who decides when to use unsafe code – our participants' answers largely agreed in that they decide it themselves.

**Tooling.** Regarding the tooling, developers tend to make a more thoughtful and balanced use of additional tools from the Rust toolchain, e. g., the linter tool Clippy[5], the additional interpreter MIRI[6], which can be used to analyze unsafe Rust code, or Cargo Deny[7] to check their dependencies for security issues. Most participants knew at least some of these tools, however, they tended only to make use of them if the effort seemed reasonable.

**Misuse & Vulnerabilities.** Only a few of our participants were able to contribute when asked about past experiences with the misuse of unsafe code so far. We found that bugs were perceived to occur more often in unsafe code, but "misuse" concerning unsafe code could only be observed once where an actual advisory had been filed. "*You just fill an advisory, do a patch release [..] and you move on.*" The general belief was that, most times, security vulnerabilities do not have a negative impact on the project if developers take care of the advisories in time.


## Acknowledgment

This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CaSa – 390781972.


[3] https://github.com/rust-unofficial/awesome-rust

[4] https://doc.rust-lang.org/nomicon/

[5] https://github.com/rust-lang/rust-clippy

[6] https://github.com/rust-lang/miri

[7] https://github.com/EmbarkStudios/cargo-deny