"I wouldn't want my unsafe code to run my pacemaker": An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust

Sandra Höltervennhoff[L] Philip Klostermeyer [C] Noah Wöhler [C] Yasemin Acar[PW]
Sascha Fahl[C]

[L] *Leibniz University Hannover*
[P] *Paderborn University*
[W] *George Washington University*
[C] *CISPA Helmholtz Center for Information Security*

Abstract

Modern software development still struggles with memory safety issues as a significant source of security bugs. The Rust programming language addresses memory safety and provides further security features. However, Rust offers developers the ability to opt out of some of these guarantees using unsafe Rust. Previous work found that the source of many security vulnerabilities is unsafe Rust.

In this paper, we are the first to see behind the curtain and investigate developers' motivations for, experiences with, and risk assessment of using unsafe Rust in depth. Therefore, we conducted 26 semi-structured interviews with experienced Rust developers. We find that developers aim to use unsafe Rust sparingly and with caution. However, we also identify common misconceptions and tooling fatigue that can lead to security issues, find that security policies for using unsafe Rust are widely missing and that participants underestimate the security risks of using unsafe Rust.

We conclude our work by discussing the findings and recommendations for making the future use of unsafe Rust more secure.

1 Introduction

Security issues are prevalent in software products and more vulnerabilities are reported every year. In 2021, the CVE Dictionary reported 20,161 newly published CVEs, of which 4,074 were classified between high and critical severity by NIST. In the following year, 25,059 new CVEs were filed [1], [2].

A root cause for this alarming number of bugs is the deployment of low-level or system-oriented programming languages that usually require developers to take care of the entire memory management and, thus, impose the burden of not introducing security vulnerabilities into the systems onto them. As regularly reported in the news, negligent actions can easily lead to severe security incidents [3]–[6].

To counter this trend and aid developers in secure coding, the Rust programming language emerged, constructed to support both secure programming and good performance. Rust provides concurrency and memory safety while simultaneously being designed to be fast and efficient. The language achieves this through a so-called ownership system and the borrow checker, finding many security issues at compile time [7]. Therefore, Rust offers security guarantees which are not bypassable in normal "safe" Rust code [8]–[10].

However, there are use cases for which safe Rust code is insufficient. Such use cases, for example, require raw memory access and are not supported by regular Rust code. To support these use cases, developers can use the so-called "unsafe" Rust feature. Unsafe Rust provides developers with an extended feature set, such as dereferencing raw pointers and calling "unsafe" functions. In return, developers lose safety guarantees, including memory safety, and must implement safety measures themselves. It is therefore generally recommended to use unsafe Rust sparingly and with great care and attention. However, there is little research on how developers approach writing, reviewing, and testing unsafe code fragments and whether they use security policies or guidance.

Contributions. While previous work explored the use of unsafe Rust code in open source projects [11]–[13], our work aims to investigate decision-making and development processes for unsafe Rust. Therefore, we conducted 26 semi-structured interviews with experienced Rust developers who had used Rust's unsafe feature. To the best of our knowledge, we are the first to conduct qualitative interviews with Rust developers with a focus on better understanding the use and motivation of unsafe code. We provide insights into why and how developers use unsafe code in Rust. We identify potentially dangerous misconceptions around unsafe code and explore developers experiences with unsafe code-specific security incidents. In the course of our work, we contribute to the security community by addressing the following research questions:

RQ1. *What are common practices of deciding for and implementing software using unsafe Rust?* Deciding for and implementing unsafe Rust is critical with potentially severe security consequences. We explore decision processes for using unsafe code and how developers implement it. Moreover, we investigate security policies and resources for writing unsafe code which developers follow.

RQ2. *How do developers assess unsafe Rust's features, limitations, and security risks?* To make informed decisions regarding the use of unsafe code, developers need to properly assess its features, benefits, limitations, and risks. Therefore, we contribute to understanding developers' comprehension of the concept of unsafe Rust and uncover potential misconceptions that might lead to insecurities.

RQ3. *What security code reviewing and testing practices around unsafe Rust are used by developers?* Proper quality control and security checks are crucial for developers to keep vulnerabilities out of a code base. Therefore, we explore unsafe code specific code reviewing and testing practices.

RQ4. *How do developers experience security incidents as the result of incorrect use of unsafe Rust?* Using unsafe code incorrectly might lead to serious security incidents. Therefore, we explore developer experiences with such incidents.

Availability. To support the replication of our work and help other researchers build upon it, we provide a replication package (cf. Availability 7).

2 Background

Below, we provide background information for both safe and unsafe Rust and their core principles and safety features.

Rust and Core Safety Features. Rust is a multi-paradigm programming language. It includes high-level programming concepts and provides good runtime performance with zero-cost abstractions since memory management checks happen during compilation rather than runtime. Rust also provides memory safety using "ownership" and "borrowing" concepts following three core ideas [14]:

- Each value has an owner.
- Each value can only have one owner at a time.
- A value will be dropped if the owner goes out of scope – its memory is automatically returned.

All rules are enforced by the "borrow checker" in the compiler. This allows static analysis to reject code that would lead to uncertainties about ownership and that could, thus, potentially lead to memory errors. Rust also eliminates the need for a conventional garbage collector [13]. However, the ownership concept has the downside of a steeper learning curve when starting to work with Rust compared to other programming languages like C or C++ [15].

Unsafe Rust. To work around some of the limitations of safe Rust, e.g., integrating code written in another language or working with memory directly, Rust provides unsafe, a superset of safe Rust. Unsafe Rust code is written using the `unsafe` keyword. Unsafe lifts certain safety guarantees and provides ways to perform operations that cannot be covered by a static analysis step during compilation or through runtime checks and would therefore be rejected by the safe Rust compiler. Unsafe Rust grants the developer an additional set of capabilities that can be used only in those unsafe code regions and consists of five major features called "unsafe superpowers", including [16]:

- Dereferencing raw pointers
- Calling unsafe functions and methods
- Implementing unsafe traits
- Accessing or modifying mutable static variables
- Accessing fields of unions

Rust's "Foreign Function Interface" (FFI) allows calling code written in other languages. It must be called using unsafe code since the compiler cannot provide safety guarantees for external languages. Unsafe blocks still provide certain safety features, e.g., the borrow checker is still enabled. Nevertheless, with this feature set, developers are given great responsibility upon using unsafe, as they could inadvertently introduce critical vulnerabilities into their code.

Ecosystem and Tooling of Rust. As we asked developers for their interaction with tools and the Rust ecosystem, we give a brief background around emerging tools and topics.

Rust includes the dependency manager and build tool "cargo" to compile external packages called "crates" and to add them to a project. At the time of writing in 2022, "crates.io", the central repository, provides more than 77,400 external crates with a plethora of extensions and abstractions.

To facilitate the debugging of safe Rust and especially unsafe blocks, developers may rely on multiple tools provided by the Rust ecosystem. This includes "MIRI", an additional interpreter that triggers on various undefined behavior errors, including out-of-bounds memory accesses, invalid uses of uninitialized data, and other memory-related errors by testing all resulting binaries [17]. In addition, there is also an implementation of the tool "Valgrind" for Rust that can also detect many memory management and threading errors and be used to profile programs in detail [18].

Furthermore, the Rust compiler provides several linting mechanisms out of the box. In general, a linter can be used to detect programming or stylistic errors, which in turn can result in better code. Rust uses a system of linting levels between *allow* and *forbid*, with the lint `unsafe_code` defaulting to *allow*, in so far generally allowing the usage of unsafe. A developer could thus also prevent the use of unsafe code completely by including the lint `#![forbid(unsafe_code)` at the top of their code base.

Moreover, the additional linter "Clippy" provides several more lints for safe and unsafe Rust. As an example, there are lints such as `undocumented_unsafe_blocks` which alerts a developer that an unsafe block has not yet been documented. Clippy is also available through the standard Rust crate repository [19].

Cargo offers some further extensions through crates for automatic verification of project dependencies. Among them is `cargo-audit` of the Rust Secure Code working group, which builds a crate dependency tree and checks it against the RustSec Advisory Database. This can be used to determine whether known vulnerabilities occur in dependencies [20]. In addition, there is `cargo-deny`, which allows fine tuning of dependencies, e.g. by displaying whether license agreements of dependencies meet expectations and requirements [21].

3 Related Work

We discuss related work in two key areas: research on the security of (unsafe) Rust and research on the usability of Rust.

Rust Security. We are among a number of researchers investigating the security of Rust and the unsafe keyword. First, the Rust Belt projects aims to verify and prove Rust's safety claims, placing them on a formally secure footing [9], [10], [22]. Moreover, different tooling and verification approaches were designed to validate Rust programs and identify bugs such as memory safety issues [23]–[29]. Regarding tool support for transitioning code to Rust, Emre *et al.* explored the problems of translating C code into Rust via the "C2Rust" tool and listed causes and categories for resulting unsafe code fragments [30]. Building on that, Ling *et al.* presented the tool "CRustS" with several improvements [31].

Other researchers studied the isolation of unsafe Rust. While Lamowski *et al.* isolated Rust from unsafe C libraries by executing unsafe functions in separated child processes [32], Almohri *et al.* presented a system to isolate Rust memory from unsafe Rust code by utilizing the Linux kernel [33]. An approach pursued by Liu *et al.* was to create different heap regions for safe and unsafe memory [34] and Rivera *et al.* introduced a technique for isolating the heap of safe Rust from languages included via FFI [35]. Moreover, several works have analyzed unsafe code. Evans *et al.* examined Rust code gathered from crates.io to investigate the occurrence, propagation, and usage of unsafe code [12]. Astrauskas *et al.* also analyzed code from crates.io. They found that 23.9% of all crates used unsafe code, with a small unsafe-to-safe code ratio [13]. Qin *et al.* studied the usage of unsafe Rust code in different environments and extracted the main reasons for the usage [11]. Xu *et al.* investigated Rust-related CVEs with memory-safety issues and examined their consequence and patterns of occurrence [8].
The presented research differs from ours in that the "as-is" state of code was studied. By conducting interviews with developers, we aim to gain insights into Rust developers' mindsets and decision-making processes regarding unsafe code. Using our research approach, we enrich previous research with qualitative insights that cannot be derived from unsafe code fragments.

Rust Usability. Rust usability has been studied in previous works, often with a focus on the adoption process. Zeng *et al.* examined difficulties when adopting Rust by analyzing several online platforms [36]. Zhu *et al.* investigated problems in Rust programming and learning by inspecting Stack Overflow questions and conducting a survey [37]. With Rust as an example, Abtahi *et al.* examined which resources developers find most helpful in learning a new language [38]. Various tooling options were constructed to aid developers in learning Rust [39]–[41]. Mindermann *et al.* identified the most important Rust crypto APIs and analyzed them regarding usability [42].

Through interviewing and surveying developers, Fulton *et al.* gained further insight into the adoption process of the Rust language. They identified a steep learning curve when switching to Rust and concerns regarding the novelty of the language (e.g., whether the ecosystem is already mature enough). However, most participants rated the use of Rust positively. Regarding unsafe code, most study participants claimed to have used this feature. The authors found that companies mostly seemed to have no or only vague guidelines on how to review unsafe code [15].

Since using unsafe code in Rust is such a safety-critical component, this is where our work ties in. We conduct interviews mainly concentrating on this feature and questioning developers' mindset and risk awareness to study the effects insufficient security policies might have.

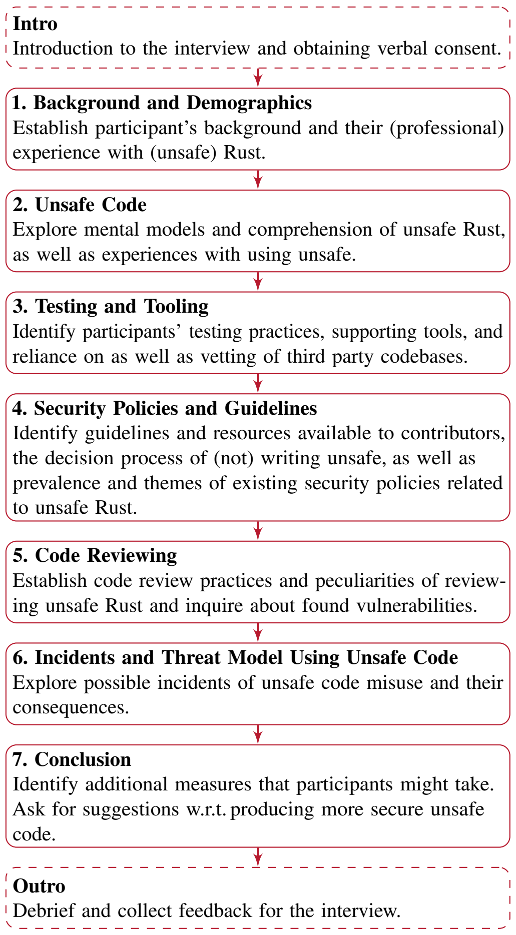
4 Methodology

Below we describe the structure and procedure of the interview study, the data analysis and interview coding, our recruitment procedure, as well as ethical considerations and limitations of our work.

4.1 Interview Guide

We decided for interviews, since we were interested in in-depth insights into the experiences and motivation for using unsafe Rust. We based our interview questions on previous work exploring Rust and its unsafe keyword [8], [11]–[13], [15] and phrased them with our research questions in mind. The interview pilots with three participants served to assure the quality and applicability of our interview guide. During the pilots, we tested and improved the wording and order of the interview questions.

We conducted all interviews online to reach a wider and more diverse audience. Semi-structured interviews allowed us to ask open-ended questions and explore the field without limiting response options, as well as ask follow-up questions if needed. We created the interview guide in German and English and updated both guides synchronously in case of minor changes or clarifications were added after an interview. Each interview was conducted by one researcher, with a shadow interviewer as a backup. Overall, three different researchers were involved in the interviewing process. All interviewers were knowledgeable about unsafe Rust and interview studies


Intro
Introduction to the interview and obtaining verbal consent.

1. Background and Demographics
Establish participant's background and their (professional) experience with (unsafe) Rust.

2. Unsafe Code
Explore mental models and comprehension of unsafe Rust, as well as experiences with using unsafe.

3. Testing and Tooling
Identify participants' testing practices, supporting tools, and reliance on as well as vetting of third party codebases.

4. Security Policies and Guidelines
Identify guidelines and resources available to contributors, the decision process of (not) writing unsafe, as well as prevalence and themes of existing security policies related to unsafe Rust.

5. Code Reviewing
Establish code review practices and peculiarities of reviewing unsafe Rust and inquire about found vulnerabilities.

6. Incidents and Threat Model Using Unsafe Code
Explore possible incidents of unsafe code misuse and their consequences.

7. Conclusion
Identify additional measures that participants might take. Ask for suggestions w.r.t. producing more secure unsafe code.

Outro
Debrief and collect feedback for the interview.

Figure 1: Procedure and structure of our interview guide, consisting of seven blocks surrounding the comprehension, usage, and handling of unsafe Rust.

with developers. The interview consisted of seven thematic blocks that primarily focused on unsafe Rust and secure Rust programming. Figure 1 illustrates the interview structure. We asked all participants to fill out a pre-survey to collect their demographics and consent. The interview guide and the pre-survey can be found in the replication package.

4.2 Data Analysis and Coding

We recorded and transcribed all interviews using a GDPR-compliant third-party provider. We reviewed all transcripts for correctness and completeness. We coded all interviews using an iterative, open coding approach [43]–[45] looking for recurring themes and established an initial codebook based on the interview guide and interview impressions. In an iterative approach, all researchers applied the codebook to all interviews until no further codes or themes emerged [46], [47]. The same researchers who conducted the interviews did the coding. We resolved conflicts by consensus or by adding new (sub)codes after each iteration. This coding approach does not require the reporting of an inter-coder agreement since all conflicts were resolved when they emerged, resulting in a hypothetical final agreement of 100%. This approach follows best practices in our field [48]–[50]. Our final codebook contained 151 unique codes. We assigned each interview 79 codes on average. Using affinity diagramming [51], we extracted the most relevant information from the codebook and condensed them into results.

4.3 Recruitment

To gain in-depth insights into the usage, perception, reviewing practices, and testing of unsafe Rust, we recruited experienced Rust developers who had used unsafe Rust before. We piloted the interview with one developer from our professional network and two freelancers from Upwork [52]. We chose Upwork as our initial recruitment option as it offers access to many freelancers and has been successfully used in previous studies [53], [54]. We made only minor changes during the piloting. Hence, we included the three pilot interviews in our final data set. For participant diversity and to attract sufficiently many participants, we recruited further participants from GitHub. We selected projects from the curated "Awesome Rust" [55] list and manually identified GitHub users who had committed at least one unsafe code fragment to one of the projects. From those, we randomly invited 250 developers who provided a mail address in their public GitHub profile [1]. We recruited further unsafe Rust experienced participants, who were referred to us by peers or who contacted us after visiting our study website. Overall, we could recruit two participants from Upwork, 21 participants from GitHub, and three participants via peers. We conducted all 26 interviews between October 2021 and August 2022. We offered all participants a compensation of $80 via bank transfer or voucher. As common for qualitative interview studies, we aimed for participant diversity and not generalizability. In our study, we aimed for diversity in terms of Rust experience and professional background.

4.4 Ethical Considerations and Data Protection

This work was approved by our Ethical Review Board (ERB). Additionally, we followed the ethical principles of the Menlo report for research involving information and communications technologies [56]. For the data collection and processing, as well as the used transcription service, we adhered to the European General Data Protection Regulation (GDPR). We stored the data in our secure cloud, to which only the researcher associated with the project had access. We removed all personally identifiable information from the transcripts. All participants

[1]According to an update of GitHub's ToS, recruiting participants for research studies is no longer permitted and should be avoided for future research, as determined by the USENIX Security Research Ethics Committee.

filled out a consent form before the interviews. The consent form informed them of the content and procedure of the interview study and provided further contact information. At the beginning of each interview, we explained that the interview was recorded and how we stored and processed participant data. We obtained verbal consent from each participant for this procedure.

Disclaimer: Recruiting participants on GitHub was a common recruitment option for developers in previous work [49], [57]–[59]. Since a change in GitHub's 'Acceptable Use Policies' [60], this type of recruitment is prohibited. Therefore, we discourage this recruiting method in future studies. However, we carefully filtered for potential participants and did not send mass invite emails. We manually invited small batches of developers who had committed unsafe Rust code and added their contact email addresses to their public profiles. We stopped inviting developers once we reached saturation. Additionally, we refrained from contacting developers who did not want to be contacted, contacted developers only once, and offered to add them to an ignore list. We received no complaints from any of the developers we contacted. Therefore, we consider the damage caused by the recruitment justifiable.

4.5 Limitations

Our work has several limitations inherent for this type of interview study and should be interpreted in context.

Participants self-reported their experiences. They might have forgotten, or omitted information, e.g., exhibiting social-desirability bias. This could lead to over- or understating of individual experiences. As a countermeasure, we aimed to phrase sensitive inquiries accordingly. Moreover, before each interview, we explained that we were only interested in experiences and opinions and did not judge any answers. We acknowledge that some interview questions were closed-ended or could have influenced the participants answers.

We invited most participants by sending out emails to Rust developers that had at least one GitHub commit containing unsafe code or reached them via snowball sampling. We sampled randomly from all identified developers and did not explicitly account for developers from varying project sizes or application areas. Our email invites may have lead to a self-selection bias: Developers with high interest in the topic might have been more likely to participate or more motivated to talk about their encounters with unsafe Rust than other Rust developers. Our sample is therefore regarded as a convenience sample, with a greater part of the participants residing in Europe, and is not generalizable to the Rust population as a whole. Interviews with participants from other regions in the world might haven given different answers.

In particular, we only interviewed participants with unsafe Rust experience and cannot provide insights for developers who did not use unsafe code.

However, we are confident that we interviewed a diverse set of developers and thus are able to provide a rich set of valuable insights for the security research community.

5 Results

Below, we report results from 26 semi-structured interviews with experienced Rust developers regarding their use, understanding, reviewing practices, and testing of Rust unsafe code. In order to make our results more tangible, we report counts from our codebook. However, as our work is solely of qualitative nature, these counts should not be interpreted as quantitative results, but help to give weight to the different topics. The complete codebook is available in our replication package.

5.1 Participant Demographics

We begin with reporting general demographic information gathered from the pre-survey and in the interviews [2]. Important demographics are summarized in Table 1. The majority of our participants identified as male (n=20), except for one female, one non-binary, and one agender participant. The age ranged from 18 to 46 years (median=28, mean=29.17). Our participants came from 15 different countries. On average, an interview lasted 57 minutes (median: 58 minutes). 11 participants had used Rust for two to five years, while 12 participants had used Rust for more than five years. Apart from Rust, nearly every participant reported a background in C or C++. Second most popular was Python, followed by Java and JavaScript. Three participants self-reported having considerable security experience, 14 participants had some, and six indicated that they had little security experience.

During the interviews, 17 participants reported using Rust for the maintenance of or the contribution to open source projects, with some participants mentioning significant contributions to the Rust ecosystem and two participants being part of the Rust Core Team. Using Rust in their company has been stated by eleven participants, three were using Rust for freelance work, while four participants used Rust for academic purposes, and three participants indicated they use Rust only for private projects. Further, seven participants indicated that they perceived the security sensitivity of their project as relatively low. Participants often indicated working with Rust in small teams or environments. Eight participants stated to work in medium to large teams or environments with more than 40 developers.

Regarding contacts that could offer security advice, twelve participants mentioned regularly reaching out to colleagues or personal contacts for security-related Rust questions, while eleven participants stated that they did not need security expert advice since they were experienced in writing secure Rust code or were more likely to be the ones asked for advice

[2] We only collected pre-survey information after the piloting phase. Hence, some demographic information is missing for the first three participants.

ID	Language	Length	Country	Rust Exp.	Security Exp.	Usage Areas[1]
p01	English	1:09h	DE	> 5 years	Considerable	A, P
p02	English	0:48h	NO	2–5 years	-	C, F
p03	English	0:34h	NO	> 5 years	Little	C, O, P, F
p04	English	1:03h	RU	> 5 years	Considerable	O
p05	English	0:53h	US	2–5 years	Some	C, O, P
p06	English	0:34h	LT	2–5 years	Little	P
p07	English	1:18h	HR	> 5 years	Little	C, O
p08	English	1:00h	DK	2–5 years	Some	O
p09	German	1:01h	DE	> 5 years	Little	O, P
p10	English	0:47h	BR	2–5 years	Some	O, P
p11	English	1:20h	NL	2–5 years	Considerable	C, O
p12	English	1:13h	GB	> 5 years	Some	C, O
p13	English	0:47h	GB	2–5 years	Some	C, O
p14	English	0:45h	NL	2–5 years	Some	F
p15	English	1:07h	PL	> 5 years	Some	O, P
p16	English	0:50h	US	> 5 years	Considerable	C
p17	English	1:03h	IT	> 5 years	Little	A, O, P
p18	English	0:56h	CN	2–5 years	Some	A
p19	German	0:41h	DE	2–5 years	Some	C, O
p20	English	0:50h	CA	> 5 years	Some	P
p21	English	0:44h	IN	2–5 years	Some	O, P
p22	German	1:04h	DE	> 5 years	Little	O
p23	German	1:13h	DE	> 5 years	Some	P
p24	English	0:42h	US	2–5 years	Some	A, P
p25	English	1:08h	DE	> 5 years	Little	C, O
p26	English	1:05h	NL	> 5 years	Some	C, O

[1] C: Company, F: Freelancing, A: Academia, P: Private Work, O: Open Source

Table 1: Overview of the 26 interviews and associated participant information.

on Rust security themselves. Ten participants reported to not have dedicated Rust security experts in their company or professional network. However, 15 participants used platforms such as Reddit or Discord to ask for security advice, and six participants specifically mentioned Stack Overflow.

5.2 Unsafe Rust Application Areas

We asked participants about application areas and reasons for using unsafe Rust. While this topic was already examined in previous research by analyzing unsafe code fragments [11]–[13], we asked about the use cases to better relate and contextualize later answers. Moreover, we are the first to confirm previous research with developers directly.

Asked about possible use cases for unsafe, our participants mentioned a multitude of application areas. As far as “unsafe superpowers” were concerned, pointer operations and raw memory accesses were elaborated by 23 participants. Further, eight participants mentioned the calling of unsafe functions inside an unsafe block. In contrast, implementing unsafe traits, mutating a global static, or working with unions were mentioned less often. Apart from the fundamental unsafe operations, another important purpose our participants reported was Rust’s FFI, often associated with OS or hardware interfacing. Next in line, 13 participants reported data structures like doubly linked lists, and eleven participants mentioned the usage of unsafe code for concurrency reasons. Further use cases our participants mentioned were more diverse, such as in-place or intrinsic mutation, lazy loading, and working with memory or in-line assembly.

These unsafe applications reflect our participants’ actual deployment of unsafe code, particularly concerning the FFI. All but one participant used unsafe code with the FFI, with some trying to limit their unsafe code use exclusively towards it. Most participants primarily mentioned the inevitable use of FFI and performance improvements as reasons for using unsafe code. However, bringing up performance, seven participants stated they would not use unsafe code for this reason and four participants pointed out that the performance gain had to be significant or necessary. Participants also indicated that they had used unsafe code before since it had saved them effort or allowed them to write code more efficiently.

5.3 Common Practices of Deciding for and Implementing Software Using Unsafe Rust

To answer RQ1, we asked for motivation and decision processes, including policies for using unsafe, and programming practices.

5.3.1 Motivation and Decisions for (not) Using Unsafe

Over the course of the interviews, 18 participants explicitly stated that they tried to avoid unsafe code or only used unsafe if really necessary, e.g., there was no alternative to using the FFI. Further, five participants emphasized that they tried not to use unsafe code in critical parts, like network stacks or work for company customers. As one participant stated: “*I wouldn’t want my unsafe code to run my pacemaker.*” (p07) Eleven participants mentioned that they preferred to write safe code, even if the code was more complex or less performant than the unsafe counterpart. However, six participants declared that sometimes it was justified to write purely unsafe functions. Their examples included lower-level code, such as architecture-specific operations or calls and direct access to registers or memory. Additionally, six participants would consider using unsafe if the code performance was (noticeably) better:

> “*[T]hat’s a pretty hot piece of code, so it’s fairly important that, yes, the bounds checks probably would not hurt that much, but at the same time we just don’t want to bother risking [using safe code].*” - p05

On the contrary, nine participants explained that they were satisfied with the optimization done by the compiler and that the optimization was sometimes even better or equal to unsafe code. In this vein, some participants indicated that they later rewrote unsafe code in their project or tested safe alternatives alongside the unsafe code:

> “*We started that project with more unsafe code in it than we have right now, even though we’ve gained features because that developer has slowly learned to trust that there is usually a safe way to do things, and that’s usually the better way to do things.*” - p25

In general, the use of unsafe code was mostly based on common sense and trust in the developers' capabilities to decide if unsafe code should be used or not, as illustrated by one participant:

> "*Typically, in a project when the topic of unsafe code comes up, it is because it is clear that something should be solved with unsafe code.*" - p04

Four participants mentioned that unsafe code had to pass their internal code review process. Additionally, three participants stated that lead maintainers or senior developers made final judgments for unsafe code fragments. Four participants discussed the use of unsafe code in their team and evaluated the pros, cons, and alternatives. However, one interviewee brought up team decisions as a potential for conflicts:

> "*Some members of our team were very adamant that we should not use unsafe code here [...]. There were periods in time when the other team, the front-end team, puts blame onto us that their code fails because we did something wrong in our unsafe code.*" - p12

Despite this potential for conflicts, only very few participants worked on projects that had explicit security policies for (not) writing unsafe code. Two participants worked on crates that used `#![forbid(unsafe_code)]` or `#[deny(unsafe_code)]` inside their modules. Both are lints that produce errors if the `unsafe` keyword is used. Moreover, three participants used mandatory security comments for unsafe code or mentioned explicit API design policies, which included locking the unsafe code behind a safe interface.

We categorized answers into different team and project sizes, but found no influence on the presence or absence of formal unsafe policies. However, discussing the use of unsafe in a team was only done in smaller projects. Though also reported by smaller projects, more than half of the medium to large projects reported trust in the common sense when writing unsafe. While we consider the differences in deciding for unsafe between different team and project sizes to be rather limited, four participants working in smaller projects hold the sentiment that they would expect more formal policies or formalities for larger projects. Our findings are in line with research comparing different company sizes. They often found only small or no differences in security practices [48], [61]–[63].

5.3.2 Writing Unsafe Rust

Asked about their own writing and thought process around unsafe code, 16 participants indicated that they tried to identify and enforce all invariants and contracts. This includes the preservation of all properties that must be upheld at all times, as well as pre- and post-conditions. A few participants mentioned exceptions to that rule, like pure FFI calls or code, which was perceived as trivial. One participant used the `unsafe_block_in_unsafe_fn` lint, so that specific unsafe operations within an unsafe function still require unsafe blocks.

Isolation of unsafe code from their safe code has been described by 15 participants. Some specified it as explicitly outsourcing unsafe code to other modules or crates. Also, 14 participants stated that they tried to make the unsafe part as small as possible and confine it to small code blocks. However, one participant mentioned that, while still trying to keep the unsafe block small, they rather put everything that needs to be checked inside the unsafe block. Safe interfacing and safe APIs were mentioned by 22 participants, providing the possibility for the end user, or at higher levels of the project, to only interact with safe code. If no viable safe-interface solution existed, two participants explained that they tried to wrap their unsafe code inside other unsafe code, creating a facade that provides an easier-to-use interface. Four participants mentioned that, for performance reasons, they provided a safe as well as an unsafe function to the end user.

Reading the documentation thoroughly was mentioned by nine participants, also comprising documentation of code they wanted to include via the FFI, to convince themselves that they had upheld all invariants. Writing good documentation was also important. Writing safety comments were mentioned by 16 participants, exactly stating why the block is safe to use and which invariants have to be upheld. One participant used Clippy lints to remind themselves of these comments. Another participant even indicated that they wrote documentation before starting to actually code, to have a reference for orientation. Contrary, one participant stated that they only wrote little documentation and that it would be hard to convince other developers about the correctness of their unsafe code. Yet another participant indicated that they might forget to write safety comments, as this would be something they left for the last. Again, if code fragments were assessed to be trivial, some participants refrained from writing detailed comments or wrote none at all.

In terms of encapsulating unsafe use cases, 13 participants mentioned that they used crates that already encapsulate their use cases. By this, they do not have to write unsafe code themselves. If they still had to write unsafe, most participants wrote it from scratch and did not have any use cases where they reused previously written or third-party code. Six participants mentioned unsafe code fragments that they reused across different projects or wrote a macro that contained unsafe code. One person stated that they had copied code from the documentation.

We also found a few examples where unsafe code was used carelessly. Three participants knew that their unsafe code could lead to undefined behavior, e.g., one participant explained that their safe code layered above the unsafe code was not yet safe to use:

> "*There are multiple points where you could, in safe code, cause undefined behavior, and I've just basi-*

cally chosen to ignore it because I know I will solve that problem." - p05

Three participants stated that they sometimes wrote unsafe code without diligent care. Two reasons were projects with too many lines of unsafe code or projects that were perceived as less important or security-critical.

Considering the confidence in the written unsafe code, the responses were rather diverse. Twelve participants felt confident in their code or did not worry too much about its safety and security, e.g., because they stated to have an extensive test suite. The remaining participants were not confident or at least undecided. Reasons were unsafe code use cases that they found more complicated to write, changing security requirements, or lacking security experience:

> "*[B]ut then we have some new changes in Rust where something is, for example, not technically allowed [...]. That would mean that some of my unsafe code that I've written in the past might now be marked as wrong.*" - p26

For information on writing unsafe code securely, many participants referred to official documentation like crates' API documentation or the standard library's documentation. Twelve participants mentioned the Rustonomicon. Some had only consulted it once; others frequently used it as a reference guide for the secure use of unsafe code. Some participants relied on the Rust Book or blog posts: "*Well, a lot of blogs exist, but it's scattered information specifically for embedded.*" (p03) Many participants stated that they mostly did not look at guidelines, mainly as they drew from their experiences and used resources only for learning, or they had started writing Rust before elaborate guidelines even existed.

> RQ1: Common practices of deciding for and implementing software using unsafe Rust.
>
> - Most participants tried to avoid unsafe
> - Only a few participants reported special security policies to write unsafe
> - Developers trusted their common sense when using unsafe, project-level recommendations were rare
> - Participants reported a thoughtful process for writing unsafe, including the use of safe interfaces, good documentation and safety comments, and small unsafe code fragments
> - Unsafe was sometimes used without diligent care
> - To avoid writing unsafe, participants frequently reported using crates that encapsulate unsafe code
> - The confidence in their code varied among participants

5.4 Assessment of Unsafe Features, Limitations, and Related Security Risks

To explore RQ2 without intimidating participants, we asked them how they would explain unsafe code to an unfamiliar developer and asked follow-up questions as needed. Participants frequently compared writing unsafe code to programming in C or C++. Further, 19 participants explained that using unsafe code leaves the compiler unable to check all invariants and pre- or post-conditions for the unsafe fragments. As a consequence, developers have to do it themselves and be sure to know what they are doing. Several participants described this with statements similar to the one used in the official Rust book [14]: "Trust me, I know what I'm doing." One participant described unsafe code as an area where one has fine-grained control of the code's functionality and does not depend on compiler optimizations. Some participants compared unsafe code to an escape hatch in Rust without any barriers left. Ten participants had the misconception that inside unsafe Rust, all or at least some compiler checks would be turned off, whereas six participants explained that all rules which are enforced in safe code also apply to unsafe code. Eleven participants touched on or even stressed that unsafe Rust opens up some new features to work around the limitations of safe Rust and that unsafe code is a superset of safe Rust. Four participants pointed out that the requirements for not creating undefined behavior in unsafe code were stricter than in C++, as a tiny violation of Rust's ownership and borrowing model, concepts not present in C++, can lead to immediate undefined behavior.

For their regular interaction with unsafe code, 13 participants regarded the unsafe keyword as a marker of sorts, an area where critical bugs might occur and which they could better call out or monitor more closely. In this context, participants regularly mentioned that programmers were able to be more pedantic in unsafe code and focus on those areas, as they are more contained. The programmer can otherwise rely on Rust's guarantees for safe code. In contrast, three participants stated that unsafe code might become quite annoying, e.g., if it was impossible to limit unsafe code to small code fragments:

> "*Once you start running unsafe everywhere, it just starts to become annoying. So then we just say, 'Oh yes, this unsafe block count on the caller doing it right.' And this is not public-facing, this is all internal.*" - p05

Furthermore, 16 participants mentioned that unsafe code could be scary or hard to write or they might have a nagging feeling about it. The reasons were that it could be challenging to write unsafe code correctly and to consider all soundness issues or side effects. In this vein, one participant missed a knowledge base with more detailed information on undefined behavior of unsafe code:

> "*[T]here's still a lot of things that nobody really knows if what you're doing is sound. Sometimes, theoretically, it's not sound, but in practice, nobody has ever encountered an issue. So we do it anyway.*" - p25

Moreover, twelve participants explained that the difficulty of unsafe code depended on the context and specific use case.

The FFI was the most frequently given example of easy-to-write unsafe code, as mentioned by four participants. However, two participants regarded the correct usage of the FFI as difficult once it involved memory management, e.g., between different programming languages. Other reported challenges were the correct use of pointers, lifetime inferences, the creation of custom data structures, and the length of an unsafe block.

> RQ2: Developer assessment of unsafe Rust's features, limitations, and security risks.
>
> - Unsafe was often compared to writing code in C or C++
> - Some participants had the misconception that unsafe Rust turns off some/all compiler checks
> - Unsafe Rust was perceived as code fragments, where developers have to pay close attention to safety and security
> - Many participants perceived unsafe Rust as scary and error-prone
> - The assessment of unsafe strongly depended on the context and complexity of a code block

5.5 Security Code Reviewing and Testing Practices for Unsafe Rust

To answer RQ3 we asked the developers about their unsafe reviewing and tooling practices, the latter of which is strongly related to the used tooling options. As relevant for supply chain security, we also inquired about their management of (unsafe) crates they depended on.

5.5.1 Unsafe Code Reviewing

Regarding code reviews, 18 participants mentioned that they looked more thoroughly into unsafe code passages and checked all conditions and invariants for safety and security. They reported a wide range of security issues and bugs they specifically looked for in unsafe code. Recurring themes were the correct use of raw pointers and their referenced objects, also in connection with the FFI, avoiding uninitialized memory, e.g., probing the use of the `MaybeUninit` function, and preventing the occurrence of bugs due to a lax mutability use. One participant mentioned being skeptical if unsafe code was used arbitrarily:

> "*Normal code just doesn't use unsafe. [...] And if there was unsafe in there, I would take a closer look at why and would question the usage.*" - p22

Twelve participants emphasized checking that all invariants are upheld and eight participants indicated paying special attention to documentation, checking whether security comments exist and if they are actually correct. Only one participant mentioned that security comments might not be necessary to minimize the threshold of participation in their open source project. Another participant working on an open source project emphasized the importance to educate contributors and point them to the documentation, especially since they worked mostly with members from the open source community. "*It's not like we can do anything or block them from doing [pull requests].*" (p13)

Eight participants mentioned they would suggest a safe rewrite or rewrote the unsafe code to safe code themselves if possible. One participant paid no special attention to unsafe Rust:

> "*No. I understand you have to pay more attention when you use unsafe Rust code, but in reality, we actually don't do that very well.*" - p18

Twelve participants felt somewhat confident or confident in their code reviews for unsafe Rust. However, most of them realized that they might miss bugs during their code review. One participant mentioned that reviewing large pull requests of unsafe code was challenging and that they had become lazy while reviewing such code fragments in the past. Three participants stated rather low confidence in their review process for unsafe code, e.g., one participant stated:

> "*I always have a very dreadful gut feeling whenever I have to write anything like that or review anything like that, [...] the bugs are usually very subtle and it's difficult to catch them.*" - p13

5.5.2 Tests and Tooling

In this section, we shed light on the testing strategies for unsafe Rust and present security-relevant tools used by our participants. Since participants did not clearly distinguish testing and security-related tooling for safe and unsafe Rust, the following section does not only report measures for unsafe code but for Rust in general, unless otherwise noted.

We found that 18 participants reported testing their Rust code with Rust's built-in unit tests, while ten participants also mentioned integration tests. Only four participants mentioned writing documentation tests, which ensure that code examples inside the documentation are up-to-date. Moreover, one participant indicated that they relied on external security audits, while another participant mentioned penetration testing in their company. Two participants mentioned testing their code manually, e.g., by checking if permissions were set correctly or comparing the output of an algorithm to a less performant implementation using safe code. One participant stated to usually only test their code if they were unsure or knew that the code could have been error-prone. Furthermore, four participants explained that they did not test their projects for security at all:

> "*[N]o one of my project has a code coverage of more than like three percent. Even my trading code, which trades actual money, doesn't really have testing. What I do is that I manually test things and like only the things that I'm unsure about if it works or not.*" - p17

For unsafe code, participants often mentioned being more cautious and writing more tests to check all invariants. However, eight participants stated to test unsafe code less often than safe code. Prominent reasons were the FFI and interaction with low-level hardware, which made testing rather complex or even impossible. One participant tested unsafe code less often than safe code, as they felt that the complexity of their safe code was more challenging. Six participants described their testing strategies for safe and unsafe codes to be equal. One reason they gave was testing directed at the safe interface. Two participants raised concerns that if tests for unsafe code were run in debug mode, undefined behavior might slip through their test suite and cause problems in release mode, as there are fewer compiler optimizations in debug mode.

Regarding security-relevant tooling, seven participants mentioned using MIRI. While they liked the tool, participants remarked that MIRI did not support all unsafe operations, such as FFI or specific system calls:

> "*I feel like if you can run [the Rust project] with Miri because not everything works for Miri, then you have at least a higher chance of getting it right.*" - p26

Four participants used the profiling and debugging tool Valgrind for security purposes. Seven participants mentioned fuzz-like testing or the use of fuzzing tools, with some of them manually creating various randomized input patterns for their tests. One participant used the "Loom" tool [64] to test their code for concurrency issues, which also helped them find unsafe-code-related bugs.

Aside from a verification tool not being feasible for their project, some participants stated additional reasons for not using any. Some simply saw no benefit in the deployment for their project, while others perceived the tools as too complex in their usage or setup:

> " *One of the great things about Rust is that a lot of the tooling just works. [...] There's a high level of laziness when it comes to integrating new tooling. I don't really like to go jump through a ton of hoops to get something set up.*" - p25

15 participants used the Rust linter Clippy at least occasionally. For some participants, Clippy was a "must-have". Others used it only for larger milestones. Some were ambivalent towards the tool and annoyed by some lints. Four participants did not use Clippy as they perceived the signal-to-noise ratio as too low. One person explicitly stated that they were too lazy to configure Clippy.

5.5.3 Management of Unsafe Dependencies

We were interested in the participant's selection and verification methods for Rust dependencies, especially since 13 participants reported deliberately including crates that encapsulate unsafe Rust into their project.

Only a few participants reviewed crates for unsafe code before introducing them into their project as a dependency. Two participants mentioned that, if viable, they preferred crates that completely forbid unsafe. In contrast, 16 participants reported that they selected crates, including crates that might contain unsafe, based on their reputation. The number of downloads was mentioned most frequently in this context, followed by the authors' reputation. Meta-reviewing the crate by reading code samples or documentation was the second most-used strategy. Two participants reported that finding a lot of undocumented unsafe code would be an indication not to use the crate. The meta-reviewing strategy was also used as a fallback plan when a crate did not have enough reputation. Eight participants reported that they mostly inserted crates without any form of verification. One reason was time constraints. To aid in the selection of dependencies, one participant elaborated on more metrics for crates.io, e.g., the amount of unsafe code in each crate or security tags from large companies, like Google or Microsoft, who had verified a crate for security.

Three participants who found vulnerabilities in crates on which their projects depended solved the issue by forking or patching them themselves. One project addressed a security advisory in their dependencies by checking all possibly affected code with MIRI [17] and thus tried to safeguard it.

Few participants reported regularly checking their dependencies for updates. Two of them mentioned doing so via GitHub's Dependabot and one frequently used `cargo-update`. Another three participants checked their dependencies with `cargo-audit` or `cargo-deny`:

> "*Dependabot has recently started supporting Rust, and I [...] spent a lot [of time with] CVE reports on dependencies that we were using.* " - p14

> RQ3: Security code reviewing and testing practices around unsafe Rust used by developers.
>
> - Most participants reviewed unsafe more thoroughly than safe Rust
> - Participants reported checking the preservation of invariants and the integrity of the documentation for unsafe regions
> - Participants often tried to test unsafe more rigorously
> - However, testing unsafe was often reported as complex or not possible at all
> - The most widely used tool to test unsafe for undefined behavior was MIRI
> - A few participants reported the integration or use of security-related tools to be cumbersome or complicated
> - Participants often picked (unsafe) crates by their reputation or at most meta-verified them

5.6 Experiences with Security Incidents as the Result of Incorrect Use of Unsafe Rust

Most participants had no negative experience with severe unsafe bugs. This circumstance certainly contributed to the fact that participants generally assessed the safety approach of Rust as favorable. They were quite satisfied with the concept of unsafe in Rust, especially in contrast to other low-level programming languages.

However, several participants reported undefined behavior or bugs in unsafe code, but did not necessarily connect them with vulnerabilities. This included bugs that lead to program crashes or local denial of service:

> "*My understanding is that not all unsoundness is necessarily a vulnerability, right? There are unsoundnesses that are not large threats, depending on, of course on the threat model.*" - p08

Six participants gave examples for security-relevant bugs or vulnerabilities in unsafe code that were only found post-release. Mostly, however, the vulnerabilities were described as minor and non-exploitable. Some participants reported unsafe bugs to be very subtle. One participant reported a bug in unsafe code that had gone undiscovered for one to two years and was only found after a new compiler release. Another participant mentioned a bug that was introduced by a rather competent Rust programmer, and that was only found later:

> "*I found a piece of unsafe code that we removed for other reasons and written by my predecessor in the company. He was an extremely competent Rust developer, and basically an expert at writing unsafe, and he still missed that part.*" - p07

Overall, other than having to patch bugs and push new releases, no project suffered severe consequences of using unsafe Rust. While participants themselves made no serious experiences with the usage of unsafe, six participants recalled an incident in the context of a web framework concerning the lax usage of unsafe Rust. The crate used unsafe Rust extensively, leading to a debate and a backlash in the community, and finally to the developer quitting open source. As one participant put it "*there was only chaos, death, and destruction*" (p23).

As one reason that no serious vulnerabilities have yet been found in Rust code, two participants pointed out that it was easier to attack other non-Rust code, e.g., by exploiting a C library used through the FFI. Moreover, one participant reckoned that the community had not yet experienced severe security incidents as Rust is still used less than C++.

Participants did not only concentrate on unsafe code in their answers, eleven participants pointed out that security vulnerabilities can perfectly well occur in safe Rust, e.g., as memory leaks are not completely eliminated by Rust's guarantees or because of a flawed algorithm or logical errors. Consequently, one participant mentioned a potential security vulnerability that had been identified not in unsafe code but in the company's safe Rust code, and that could lead to being vulnerable to a denial-of-service attack.

> RQ4: Developer experiences with security incidences as the result of incorrect use of unsafe Rust.
>
> - Our participants did not experience severe security incidents
> - Most reported bugs in unsafe code were minor and did not introduce exploitable vulnerabilities
> - Participants did not necessarily connect undefined behavior or bugs with security vulnerabilities
> - Consequences were limited to additional effort for implementing fixes and publishing new releases
> - Participants generally liked Rust's approach to safety as well as the unsafe concept

6 Discussion

Below, we discuss our findings.

Use of Unsafe (RQ1). Most participants tried to avoid using unsafe code, utilizing it only when they felt they had no other option. When they had to use unsafe, most participants followed a cautious approach. Participants preferred including crates instead of re-implementing features from scratch and isolating their unsafe code, e.g., using interfaces.

Our findings suggest that the Rust community successfully communicates risks associated with unsafe code and dissuades developers from using unsafe, if possible. Moreover, the Rust community makes constant efforts to replace unsafe code with safe code.

Nevertheless, participants reported becoming increasingly inattentive or annoyed when unsafe code was frequently used since dealing with unsafe required more attention. Only a few Rust projects provided policies for writing unsafe code. Most participants relied on their common sense and the experiences of their peers. However, we found divergent opinions in detail, e.g., what should go inside an unsafe block or whether unsafe code is needed. Even though this trust system among peers seemed to work for most of our participants, this approach is not preferable from a security perspective. We consider this particularly important since some participants reported that unsafe bugs were subtle and difficult to find and reproduce. Some interviews indicated that experience might not always be sufficient to handle unsafe code. This aligns with previous research, finding security knowledge hard to apply or not helpful in preventing vulnerabilities [65]–[67]. Moreover, fostering an explicit security culture is a crucial element for motivating developers to program securely, as found through previous interviews [61], [68], case studies in companies [69], [70], and other means [63], [71].

Understanding and Misconceptions (RQ2). The general comprehension of unsafe code varied among our interviewees. Less than half of our participants mentioned that unsafe code is a superset of safe code, providing new features. Some developers believed that some or all compiler checks are turned off inside unsafe code. We did not find apparent differences in how our participants with different levels of unsafe knowledge approached writing unsafe code. Most reported being cautious, even if they had misconceptions about unsafe. While Qin *et al.* found that developers sometimes use unsafe code for labeling purposes [11], our interviews underline the importance of unsafe as a beacon in the participants understanding. They highlighted unsafe as confined code fragments where they knew to be careful.

We applaud that the Rust community conveys essential habits around unsafe code, such as limited use or isolation, and that developers tend to adopt those successfully. Despite this, based on our findings, not all developers understand the exact functionality of unsafe. Comprehension of unsafe is important, and communication should be improved, as the correct understanding can guide developers in their assessments regarding unsafe code. Misunderstanding the additional risks might induce developers to underestimate the need for strict and cautious handling regarding security guarantees. Moreover, as Rust is still in active development, the exact behavior of some unsafe edge cases might change over time. Undefined behavior, working today as intended by the developer, might lead to subtle bugs in the future [72].

Further, thinking that all compiler checks are turned off in unsafe could lead developers to believe that verification tools like MIRI are infeasible. As a result, they remain unaware of tooling options, as one participant stated about unsafe code: "*[I]t's an escape hatch. [...] [T]here's not much you can do [with tooling].*" (p01)

Appropriate communication about unsafe code is also important in the context of Rust's further adoption. If even developers that use Rust do not understand unsafe correctly, correctly assessing Rust's security benefits for developers not acquainted with Rust is likely to be harder.

Guidance and Accessibility (RQ1, RQ2). The Rust community constantly improves documentation and guidance for unsafe code, such as the Unsafe Code Guidelines Reference [73] or the Rustonomicon. While comprehensive documentation about unsafe is available, the knowledge does not seem to spread among developers sufficiently. Only half of our participants mentioned the standard reference Rustonomicon. Some suffered from unsafe Rust misconceptions. Hence, we recommend not only improving existing documentation, but also making it more easily accessible. Based on our work, in-depth documentation is partly rather scattered or, as one participant stated: "*a lot of hidden information [is] in GitHub issues.*" (p11). A big factor for the future is the packaging of this information so that developers are less likely to shy away and have it more bundled in one place. Because developers often only have little time, this information should be accompanied by quick-to-read roundups. This collection may also include concise and actionable guidelines for deciding for and writing unsafe code to establish common practices that individual projects could adopt without much effort.

Tooling and Usability (RQ3). Regarding applied testing and review practices, most participants tried to test and review unsafe code more thoroughly than safe code to check whether invariants were violated. This is in line with Evans *et al.*, who found participants to write more unit tests for unsafe Rust [12]. However, we found that some participants could only run insufficient tests because extensive testing, e.g., for the FFI or hardware interaction, was not applicable for them.

While our participants often knew of tools like MIRI, many did not use them. The integration and configuration of some external tools were mentioned to be too cumbersome, and participants preferred easier solutions. Despite research on general tool usage [74]–[76], the issue of usability is still underestimated by tool designers, although this can be a decisive criterion for tool success:

> "*There's a few fuzz testers, I use cargo fuzz. I don't really care [..] how good its performance is as long as it does the thing I want. And that's why I use cargo fuzz, because it's well documented and easy to set up.*" - p23

Security tools should reflect this knowledge and make their setup as simple as possible. This is even more important since previous work found that security is often only a secondary concern for developers [61], [63], [68], [69]. While for Rust, cargo successfully takes the first step towards simplifying the management and building of crates, for some of our participants, the burden of using security tools was still too high. We recommend deploying simplified onboarding assistants, such as configuration wizards. To cater to developers that are easily distracted in their workflow, e.g., by warnings or lints that are perceived as superfluous, tools could offer a minimal configuration, with only the most important lints or warnings turned on. Another approach for usage without any setup is the integration into browser interfaces like it is done with MIRI into the Rust playground, a web interface for the Rust compiler. This browser integration allows one to check unsafe code fragments without installing MIRI [77].

Dependency Selection (RQ3). Developers should be offered support not only with tool setup, but also with the selection of tooling and dependencies. A common practice among our participants was to utilize crates that already encapsulated unsafe code, so they would not need to write safety-critical code fragments themselves. However, many interviewees did not have time to verify the crates they included and trusted the crates or their authors more or less blindly based on their popularity. This is in line with Wermke *et al.* finding reputation and activity of repositories to be common selection criteria [49]. Therefore, we recommend trying to better assist developers in choosing the right crate, e.g., by providing easy-to-read metrics via crates.io. Examples of such metrics could be the presence or absence of unsafe code in a crate, or code reviews of unsafe code done by trusted entities. A more detailed elaboration of metrics does require further research. Initiatives to improve the open source software supply chain security such as the software bill of materials [78] or the OpenSSF scorecards [79] could be used to include such metrics. However, external tools such as `cargo-audit` were rarely used, while `cargo-vet` [80], which ensures that dependencies are audited, was not mentioned at all. Therefore, we suggest integrating such metrics as seamlessly as possible into the standard crate.io repository to increase adoption by developers.

Experiences and Future of Rust (RQ4). While we identified areas of improvement for handling unsafe Rust, none of our participants were affected by severe unsafe vulnerabilities. However, participants gave examples of unsafe code in their projects leading to very subtle undefined behavior that they had to patch. Some of these bugs already resided in their code bases for longer. This illustrates the need for diligent behavior around unsafe code. While participants reported a mostly cautious behavior around unsafe, many did not especially focus on vulnerabilities or exploitability inside their unsafe code but rather focused on invariants, as illustrated by one participant: "*But I'm not a security researcher. I care about correctness. If the code is correct, it is at least not insecure by being incorrect.*" (p07) Some participants believed that security vulnerabilities were a different kind of bug and treated them accordingly. This mindset is in line with previous work investigating the security of developers [61], [69]. It demonstrates that while many developers adopt Rust because of its security benefits, their general view on security does not necessarily change. This might result in developers being inclined to ignore or downplay security problems with unsafe Rust, especially when not having a security background. Some participants felt confident in their unsafe code, considering other attack vectors more critical. Assuming that Rust will gain further adoption and considering that Rust replaces languages like C/C++ for many security-critical projects, such thinking might lead to serious security incidents, especially since Alomar *et al.* found that security is often only added retrospectively [81].

While many participants reported that they could avoid unsafe code in the past, some use cases make unsafe code indispensable, e.g., using Rust's FFI. This is particularly important for software projects that move their code base from C/C++ to Rust. While tooling exists to support developers in transpiling C code to Rust, those do not eradicate the interaction of unsafe code. A better understanding of transforming existing code bases written in other languages to Rust has not yet been investigated and is subject of future work.

Comparison with Unsafe in other Languages. We found an overall cautious handling of Rust unsafe code. Participants were wary of unsafe, as bugs were considered subtle and sometimes hard to locate. Similar findings exist for Java.

In its internal class 'sun.misc.Unsafe' Java offers direct possibilities for interaction with low-level resources, such as allocating off-heap memory using the `allocateMemory()` method. Huang *et al.* discussed safety improvements for 'sun.misc.Unsafe'. Unsafe Java bugs were hard to locate and reproduce during testing and reviewing, which was a presumably common cause for despair [82]. Mastrangelo *et al.* analyzed the use of unsafe in Java applications and dependencies and found that unsafe Java is in fact used, yielding potentially dangerous security implications. The authors concluded that unsafe Java was most often used where the functionality was supposedly not otherwise available or to improve performance [83]. Oracle aims to reduce the spread of unsafe Java [84]. All the aforementioned coincides with our findings on Rust.

The Go programming language provides type and memory safety and includes an unsafe package as part of its standard library. Costa *et al.* found that Go projects tended to consolidate the use of unsafe into a few files. Moreover, the Go community is in a situation where breaking changes of unsafe Go could involve similar problematic aspects as we discussed for Rust. The exact functionality of unsafe Go could change, and, thus, could introduce a potential for subtle bugs and vulnerabilities. Regarding the documentation of unsafe Go, Costa *et al.* recommended more detailed official documentation for the most frequent unsafe Go use cases they found [85]. While we also recommend an improvement of the documentation for unsafe Rust, we argue that the situation is somewhat different for the Rust community. Our participants reported many guidelines and lots of documentation for unsafe Rust. However, some of them are hard to find or too complex. Hence, Rust seems to be more advanced in the development of unsafe documentation but would benefit from more easily accessible and usable documentation.

We propose to further investigate similarities with other programming languages. By comparing use cases, difficulties, and solutions, and exploring connections, we think developers and researchers can better collaborate and improve programming language security in many ways for and beyond Rust.

7 Conclusion

We conducted 26 interviews with experienced Rust developers to investigate their understanding and use of unsafe Rust and asked about their writing, reviewing, and testing approaches for unsafe code. Our results show that unsafe Rust was used rather cautiously. Many participants tried to avoid unsafe Rust whenever possible. They perceived it to be scary or error-prone. However, especially when interacting with hardware, operating systems, or other languages, developers needed to use unsafe code. Our participants tried to follow best practices such as creating safe interfaces or properly documenting their code. However, bad habits and external influences had impacts on their decisions and behavior. They often wrote unsafe code to the best of their belief and could not follow security policies or guidelines.

Acknowledgments

We would like to thank all interviewees for participating in our research. We also want to thank Marcel Fourné for their support and precious insights. Lastly, we want to thank the anonymous reviewers and our shepherd for their valuable feedback that helped us to improve our work. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

Availability

To improve the comprehensibility and replicability of our work, we provide a replication package including:

Interview guide. Our semi-structured interview guide consists of questions about Rust and unsafe code and is structured into seven thematic blocks.

Pre-survey. We collected demographic information through a pre-survey that participants completed prior to the interviews.

Codebook. The final codebook, created by three researchers, contains all the codes we used to label excerpts from the 26 interviews.

The replication package can be found at: https://doi.org/10.25835/gggv8xg7.

References

[1] *MITRE CVE*, https://www.cve.org/, Accessed: 2023.

[2] *NIST CVSS Severity Distribution Over Time*, https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time, Accessed: 2023.

[3] Cloudflare, *Incident report on memory leak caused by Cloudflare parser bug*, https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/, Feb. 2017.

[4] Google Project Zero, *gpg: heap buffer overflow in libgcrypt*, https://bugs.chromium.org/p/project-zero/issues/detail?id=2145, Jan. 2021.

[5] TechCrunch, *Apple says iOS 14.4 fixes three security bugs 'actively exploited' by hackers*, https://techcrunch.com/2021/01/26/apple-says-ios-14-4-fixes-three-security-bugs-under-active-attack/, Jan. 2021.

[6] Google Chrome Release Notes, *Heap buffer overflow in WebRTC*, https://chromereleases.googleblog.com/2022/07/stable-channel-update-for-desktop.html, Jul. 2022.

[7] *The Rust language*, https://www.rust-lang.org/, Accessed: 2023.

[8] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, 2021.

[9] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the Foundations of the Rust Programming Language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, 2017.

[10] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer, "RustBelt Meets Relaxed Memory," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–29, 2019.

[11] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "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*, Association for Computing Machinery, 2020.

[12] A. N. Evans, B. Campbell, and M. L. Soffa, "Is Rust Used Safely by Software Developers?" In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.

[13] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How Do Programmers Use Unsafe Rust?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[14] *The Rust Programming Language*, https://doc.rust-lang.org/stable/book/, Accessed: 2023.

[15] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, USENIX Association, 2021.

[16] *The Rustonomicon*, https://doc.rust-lang.org/nomicon/, Accessed: 2023.

[17] *MIRI*, https://github.com/rust-lang/miri, Accessed: 2023.

[18] *Valgrind*, https://github.com/jfrimmel/cargo-valgrind, Accessed: 2023.

[19] *Clippy*, https://github.com/rust-lang/rust-clippy, Accessed: 2023.

[20] *cargo_audit*, https://github.com/rustsec/rustsec, Accessed: 2023.

[21] *cargo_deny*, https://embarkstudios.github.io/cargo-deny/, Accessed: 2023.

[22] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked Borrows: An Aliasing Model for Rust," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.

[23] M. Lindner, J. Aparicius, and P. Lindgren, "No Panic! Verification of Rust Programs by Symbolic Execution," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, IEEE, 2018.

[24] Z. Li, J. Wang, M. Sun, and J. C.S. Lui, "MirChecker: Detecting Bugs in Rust Programs via Static Analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, Association for Computing Machinery, 2021.

[25] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, "Detecting Cross-Language Memory Management Issues in Rust," in *Computer Security – ESORICS 2022*, Springer Nature Switzerland, 2022.

[26] M. Cui, C. Chen, H. Xu, and Y. Zhou, "SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-Flow Analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, 2023.

[27] V. Astrauskas, A. Bílý, J. Fiala, *et al.*, "The Prusti Project: Formal Verification for Rust," in *NASA Formal Methods*, Springer International Publishing, 2022.

[28] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: A Bounded Verifier for Rust (N)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[29] B. Hua, W. Ouyang, C. Jiang, Q. Fan, and Z. Pan, "Rupair: Towards Automatic Buffer Overflow Detection and Rectification for Rust," in *Annual Computer Security Applications Conference*, Association for Computing Machinery, 2021.

[30] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to Safer Rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.

[31] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "In Rust We Trust – A Transpiler from Unsafe C to Safer Rust," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022.

[32] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, "Sandcrust: Automatic Sandboxing of Unsafe Components in Rust," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, Association for Computing Machinery, 2017.

[33] H. M. Almohri and D. Evans, "Fidelius Charm: Isolating Unsafe Rust Code," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, Association for Computing Machinery, 2018.

[34] P. Liu, G. Zhao, and J. Huang, "Securing Unsafe Rust Programs with XRust," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.

[35] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping Safe Rust Safe with Galeed," in *Annual Computer Security Applications Conference*, Association for Computing Machinery, 2021.

[36] A. Zeng and W. Crichton, "Identifying Barriers to Adoption for Rust through Online Discourse," in *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[37] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and Programming Challenges of Rust: A Mixed-Methods Study," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[38] P. Abtahi and G. Dietz, "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*, Association for Computing Machinery, 2020.

[39] M. Almeida, G. Cole, K. Du, *et al.*, "RustViz: Interactively Visualizing Ownership and Borrowing," in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2022.

[40] M. Coblenz, M. L. Mazurek, and M. Hicks, "Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[41] Z. Zhang, B. Qin, Y. Chen, L. Song, and Y. Zhang, "VRLifeTime – An IDE Tool to Avoid Concurrency and Memory Bugs in Rust," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, Association for Computing Machinery, 2020.

[42] K. Mindermann, P. Keck, and S. Wagner, "How Usable are Rust Cryptography APIs?" In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2018.

[43] K. Charmaz, *Constructing Grounded Theory*. Sage, 2014.

[44] A. Strauss and J. M. Corbin, *Grounded theory in practice*. Sage, 1997, p. 288.

[45] J. Corbin and A. Strauss, "Grounded theory research: Procedures, canons and evaluative criteria," *Qualitative Sociology*, vol. 19, no. 6, pp. 418–427, 1990.

[46] C. Urquhart, *Grounded theory for qualitative research: A practical guide*. Sage, 2012.

[47] M. Birks and J. Mills, *Grounded theory: A practical guide*. Sage, 2015.

[48] N. Huaman, B. von Skarczinski, C. Stransky, *et al.*, "A Large-Scale Interview Study on Information Security in and Attacks against Small and Medium-sized Enterprises," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, 2021.

[49] D. Wermke, N. Wöhler, J. H. Klemmer, M. Fourné, Y. Acar, and S. Fahl, "Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022.

[50] M. Gutfleisch, J. H. Klemmer, N. Busch, Y. Acar, M. A. Sasse, and S. Fahl, "How Does Usable Security (Not) End Up in Software Products? Results From a Qualitative Interview Study," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022.

[51] H. Beyer and K. Holtzblatt, *Contextual Design: Defining Customer-Centered Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[52] *Upwork*, https://www.upwork.com/, Accessed: 2022.

[53] D. Votipka, D. Abrokwa, and M. L. Mazurek, "Building and Validating a Scale for Secure Software Development Self-Efficacy," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, Association for Computing Machinery, 2020.

[54] H. Kaur, S. Amft, D. Votipka, Y. Acar, and S. Fahl, "Where to Recruit for Security Development Studies from: Comparing Six Software Developer Samples," in *31st USENIX Security Symposium (USENIX Security 22)*, USENIX Association, 2022.

[55] *Awesome Rust*, https://github.com/rust-unofficial/awesome-rust, Accessed: 2022.

[56] E. Kenneally and D. Dittrich, "The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research," *SSRN Electronic Journal*, 2012.

[57] C. Utz, S. Amft, M. Degeling, T. Holz, S. Fahl, and F. Schaub, "Privacy Rarely Considered: Exploring Considerations in the Adoption of Third-Party Services by Websites," *Proceedings on Privacy Enhancing Technologies*, vol. 1, pp. 5–28, 2023.

[58] Y. Acar, M. Backes, S. Fahl, *et al.*, "Comparing the Usability of Cryptographic APIs," in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017.

[59] P. L. Gorski, L. L. Iacono, D. Wermke, *et al.*, "Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse," in *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, USENIX Association, 2018.

[60] *GitHub Acceptable Use Policies*, https://docs.github.com/en/site-policy/acceptable-use-policies/github-acceptable-use-policies, Accessed: 2023.

[61] H. Assal and S. Chiasson, "Security in the Software Development Lifecycle.," in *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, USENIX Association, 2018.

[62] J. M. Haney, M. Theofanos, Y. Acar, and S. S. Prettyman, "" We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products.," in *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, USENIX Association, 2018.

[63] H. Assal and S. Chiasson, "'Think secure from the beginning' A Survey with Software Developers," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, Association for Computing Machinery, 2019.

[64] *Loom*, https://github.com/tokio-rs/loom, Accessed: 20223.

[65] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, "Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, 2020.

[66] D. S. Oliveira, T. Lin, M. S. Rahman, *et al.*, "API Blindspots: Why Experienced Developers Write Vulnerable Code," in *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, USENIX Association, 2018.

[67] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang, "It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots," in *Proceedings of the 30th Annual Computer Security Applications Conference*, Association for Computing Machinery, 2014.

[68] J. Xie, H. R. Lipford, and B. Chu, "Why do programmers make security errors?" In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2011.

[69] A. Poller, L. Kocksch, S. Türpe, F. A. Epp, and K. Kinder-Kurlanda, "Can Security Become a Routine? A Study of Organizational Change in an Agile Software Development Group," in *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, Association for Computing Machinery, 2017.

[70] A. Tuladhar, D. Lende, J. Ligatti, and X. Ou, "An Analysis of the Role of Situated Learning in Starting a Security Culture in a Software Company," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, USENIX Association, 2021.

[71] S. L. Kanniah and M. N. Mahrin, "A Review on Factors Influencing Implementation of Secure Software Development Practices," *International Journal of Computer and Systems Engineering*, vol. 10, no. 8, pp. 3032–3039, 2016.

[72] *The Rust Reference - Behavior considered undefined*, https://doc.rust-lang.org/reference/behavior-considered-undefined.html, Accessed: 2023.

[73] *Rust Unsafe Code Guidelines Reference*, https://rust-lang.github.io/unsafe-code-guidelines/introduction.html, Accessed: 2023.

[74] S. Xiao, J. Witschey, and E. Murphy-Hill, "Social influences on secure development tool adoption: Why security tools spread," in *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, Association for Computing Machinery, 2014.

[75] D. Botta, R. Werlinger, A. Gagné, *et al.*, "Towards Understanding IT Security Professionals and Their Tools," in *Proceedings of the 3rd Symposium on Usable Privacy and Security*, Association for Computing Machinery, 2007.

[76] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford, "Security During Application Development: An Application Security Expert Perspective," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, Association for Computing Machinery, 2018.

[77] *Rust Playground*, https://play.rust-lang.org, Accessed: 2023.

[78] *Software Bill of Materials (SBOM)*, https://www.cisa.gov/sbom, Accessed: 2023.

[79] *Scorecard*, https://securityscorecards.dev/, Accessed: 2023.

[80] *cargo-vet*, https://github.com/mozilla/cargo-vet, Accessed: 2023.

[81] N. Alomar, P. Wijesekera, E. Qiu, and S. Egelman, ""You've Got Your Nice List of Bugs, Now What?" Vulnerability Discovery and Management Processes in the Wild," in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, USENIX Association, 2020.

[82] S. Huang, J. Guo, S. Li, *et al.*, "SafeCheck: Safety Enhancement of Java Unsafe API," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[83] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at Your Own Risk: The Java Unsafe API in the Wild," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Association for Computing Machinery, 2015.

[84] Oracle Blog, *The Unsafe Class: Unsafe at Any Speed*, https://blogs.oracle.com/javamagazine/post/the-unsafe-class-unsafe-at-any-speed, May 2020.

[85] D. E. Costa, S. Mujahid, R. Abdalkareem, and E. Shihab, "Breaking Type Safety in Go: An Empirical Study on the Usage of the unsafe Package," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2277–2294, 2022.