

A Mixed Methods Study on the Implications of Unsafe Rust for Interoperation, Encapsulation, and Tooling

IAN MCCORMACK, Carnegie Mellon University, USA

TOMÁS DOUGAN, Brown University, USA

SAM ESTEP, Carnegie Mellon University, USA

HANAN HIBSHI, Carnegie Mellon University, USA and King Abdulaziz University, Saudi Arabia

JONATHAN ALDRICH, Carnegie Mellon University, USA

JOSHUA SUNSHINE, Carnegie Mellon University, USA

The Rust programming language restricts aliasing to provide static safety guarantees. However, in certain situations, developers need to bypass these guarantees by using a set of “unsafe” features. When these features are used incorrectly, they can reintroduce the types of safety issues that Rust was designed to prevent. We seek to understand how current development tools can be improved to better assist developers who find it necessary to interact with unsafe code. To that end, we study how developers reason about foreign function calls, the limitations of the tools that they currently use, their motivations for using unsafe code, and how they reason about encapsulating it. We conducted a mixed methods investigation consisting of semi-structured interviews with 19 developers, followed by a survey that reached an additional 160 developers. Our participants were motivated to use unsafe code when they perceived that there was no alternative, and most claimed that they would avoid using it. However, limited tooling support for foreign function calls made participants uncertain about their design choices, and certain foreign aliasing and concurrency patterns were difficult to encapsulate. To overcome these challenges, developers will need analysis tools that can find Rust-specific forms of undefined behavior within multilanguage applications.

CCS Concepts: • **Software and its engineering**;

1 Introduction

The Rust programming language has made rapid headway as a safe alternative for systems programming due to its static safety guarantees. However, it is still the case that critical systems are written in languages that do not provide inherent safety guarantees [25, 67]. Rust developers who interoperate with these applications must use a special subset of “unsafe” features, including calling foreign functions and accessing memory through raw “C-style” pointers. These features can only be used within a block or function marked with the `unsafe` keyword.

To minimize the risks of unsafe code, the Rust community advocates for keeping it minimal, well documented, and hidden beneath a safe interface [2]. However, these practices alone are not enough to ensure that Rust delivers on its promise of static safety. Errors in safe encapsulations of unsafe implementations have been a significant source of bugs and security vulnerabilities in the Rust ecosystem [49, 76, 82]. Applications that rely on interoperation or direct access to system resources may expose developers to sources of unsafety that cannot easily be minimized. Making

*This material is based upon work supported by the U.S. Department of Defense under Grant No. H98230-23-C-0274 and the National Science Foundation under Grant Nos. CCF-1901033, CCF-2447499, CCF-2339830, DGE1745016, and DGE2140739. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Department of Defense or the National Science Foundation.

Authors’ Contact Information: Ian McCormack, icmccorm@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; Tomás Dougan, tomas_dougan@alumni.brown.edu, Brown University, Pittsburgh, Pennsylvania, USA; Sam Estep, estep@cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; Hanan Hibshi, hhibshi@cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA and King Abdulaziz University, Jeddah, Saudi Arabia; Jonathan Aldrich, jonathan.aldrich@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; Joshua Sunshine, sunshine@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

matters worse, Rust’s static and dynamic semantics are still evolving [68, 74], so few tools are capable of fully testing or verifying these applications.

Interoperation is a common use case for unsafe code [2, 15, 16, 22], but most Rust-specific development tools lack support for foreign function calls. Safe interoperation will be crucial for Rust’s future, since both industry and regulatory stakeholders have advocated for adopting Rust as a replacement for existing systems languages [52, 75]. Automated translation tools are increasingly effective [33], but for the moment, interoperation is the most practical method for adopting Rust within existing, large-scale systems [52]. It can be challenging to use Rust correctly in this context, because it has significant differences from other languages [22, 42]. A detailed account of these challenges is missing, and we need this to be able to develop effective tooling for safe interoperation. To provide this perspective, we ask the following research questions:

RQ1 (Interoperation).

How do Rust developers reason about memory safety across foreign function boundaries?

RQ2 (Tooling).

What tools do Rust developers use when contributing to applications that include unsafe code, and how could tooling be improved?

Developers who interoperate with Rust will need to use a variety of unsafe features in addition to foreign function calls to be able to reconcile the differences between Rust and other languages. Having a broader understanding of what motivates Rust developers to use unsafe code in any capacity and how they reason about soundness will be helpful to identify if there are factors beyond these differences that impact the safety of multilanguage applications. However, most large-scale studies on these topics have analyzed patterns in source code instead of directly engaging with developers [2, 15]. Interviews and surveys have provided a missing qualitative perspective, but tend to have a limited focus on unsafe code [16], a small sample size [15], or different goals [22]. In addition, to serve as guidance for the design of future tools, we seek to generalize prior qualitative results to a broader population of developers. To that end, we ask the following additional research questions:

RQ3 (Motivations).

What are Rust developers’ motivations for using unsafe code?

RQ4 (Encapsulation).

How do Rust developers reason about encapsulating unsafe code?

Contribution. We provide the results of an exploratory, mixed method study that addresses the research questions above. We conducted semi-structured interviews with 19 developers and used the results to create a community survey that had 160 valid responses. Most of our participants used unsafe code to call foreign functions, and they described several aliasing and concurrency patterns that made interoperation difficult. Many developers had used Miri [29], a Rust interpreter, to find bugs, but most were deterred from using it again due to its slow performance and lack of support for key features. Participants prioritized keeping unsafe code minimal, encapsulated, and well documented, but rarely audited their dependencies and relied on ad hoc reasoning to justify their decisions. Expanded tooling support for multilanguage applications and improved documentation will be necessary to support developers in maintaining Rust’s safety guarantees across foreign function boundaries.

Outline. In [Section 2](#), we describe the role of unsafe code in the Rust ecosystem and the challenges associated with interoperating with other languages. We describe our methodology in [Section 3](#),

and we indicate threats to validity in [Section 4](#). We present our results in [Section 5](#) and compare our findings with prior work in [Section 6](#). We discuss the implications of our results in [Section 7](#) before concluding in [Section 8](#). Our replication package is available via Zenodo [\[41\]](#).

2 Background

Rust’s safety guarantees are based on the notion of ownership. Values with *copy* semantics do not have an owner and can be copied freely, while values with *move* semantics have a unique owner. Assignment transfers ownership from one alias to another, but ownership can also be *borrowed* by creating a reference. Each reference has a *lifetime*, which is the duration of the program where it is used. Rust’s *borrow checker* enforces that a value can have either a unique mutable reference or many immutable references, but not both simultaneously [\[11\]](#). A value’s type can implement *traits* that define how it behaves. Certain traits affect a value’s capability to be owned and borrowed. Values with the Copy trait have copy semantics, values that are Send can be moved between threads, and values that are Sync can be borrowed by multiple threads.

Rust’s aliasing rules are conservative and are not universally compatible with certain idioms of systems programming. When these restrictions become burdensome, developers can use the `unsafe` keyword to enable features that are not constrained by the borrow checker. Within an unsafe context, users can dereference raw pointers, execute inline assembly instructions, modify static mutable state, implement unsafe traits, and call unsafe functions [\[30\]](#). There are a variety of uses for these features [\[2\]](#). In certain situations, unsafe code can be necessary to implement design patterns that are fundamentally incompatible with the restrictions of safe references, such as shared mutable state. Foreign function calls, inline assembly, and intrinsics allow developers to leverage hardware acceleration and interact with external resources. These features may also improve performance by skipping certain run-time assertions, such as bounds checks.

However, if unsafe code is used inappropriately, it can reintroduce the same types of safety issues that Rust was designed to prevent, such as data races and accesses out-of-bounds. Rust’s aliasing restrictions also introduce new forms of *undefined behavior*. Safe references are constrained by the borrow checker, but they can also be cast as raw pointers, which can only be used in unsafe contexts. The Rust compiler is allowed to assume that programs which use these features are still following Rust’s aliasing rules. Programs that break these rules may be optimized incorrectly, introducing differences in behavior that can constitute security vulnerabilities.

Several best practices can help mitigate the risks of using unsafe code. When implementing an unsafe feature, developers are encouraged to follow the “interior unsafety” pattern by keeping it minimal, well documented, and hidden beneath a safe interface [\[49\]](#). Ideally, this interface should make it impossible for users to trigger undefined behavior from safe code. However, certain safety properties are difficult or impossible to encode into Rust’s type system without leveraging external verification tools. Miri [\[29\]](#), a Rust interpreter, is currently the only tool that can detect when unsafe operations lead to violations of Rust’s newest aliasing rules, so it is typically seen as the “de facto” [\[17\]](#) bug-finding tool for applications that use unsafe code. Currently, Miri can detect violations of two different models of Rust’s semantics. The Stacked Borrows [\[28\]](#) model was the first iteration of these rules; it is enabled by default. Developers can also switch to using the newer Tree Borrows [\[74\]](#) model by providing a configuration flag. Neither of the two models is canonical.

These practices are standard for the Rust community, but Rust is also increasingly being adopted in interoperation with preexisting large-scale C and C++ applications, such as Chromium [\[25\]](#) and the Linux Kernel [\[67\]](#). These applications will expose Rust components that may be “safe” in isolation to significant external sources of unsafety, which will not be as easy to minimize or document. In addition to these challenges, Miri does not fully support finding undefined behavior in this context. Miri can execute and trace certain foreign functions from shared libraries, but

it cannot detect aliasing violations that are triggered in foreign code, and its slow performance prevents it from being used at scale.

These obstacles will make it difficult for developers to avoid introducing undefined behavior in multilanguage Rust applications. Projects such as the DARPA’s TRACTOR program [75] and the Rust/C++ Interoperability Initiative [4] seek to provide new development tools to assist with these challenges. We need a broader understanding of how Rust developers use unsafe code to be able to design interventions that are effective in practice, since developers who use Rust in this context will necessarily engage with a variety of unsafe operations in order to reconcile differences in semantics across language boundaries.

3 Methodology

We investigated how Rust developers engage with unsafe code by adapting a sequential, exploratory mixed methods design [10], which consisted of three phases. We describe the first phase in [Section 3.1](#), where we used a screening survey to recruit developers for semi-structured interviews. In the second phase, after interviewing each eligible candidate, the first three authors conducted a thematic analysis [44] of the interview transcripts. We describe this process in [Section 3.2](#). We describe the third phase in [Section 3.3](#), where we used the results of our qualitative analysis to create a community survey. We chose this combination of methods because it allowed us to evaluate the generalizability of prior qualitative findings and to provide a missing qualitative perspective on interoperation and tool use.

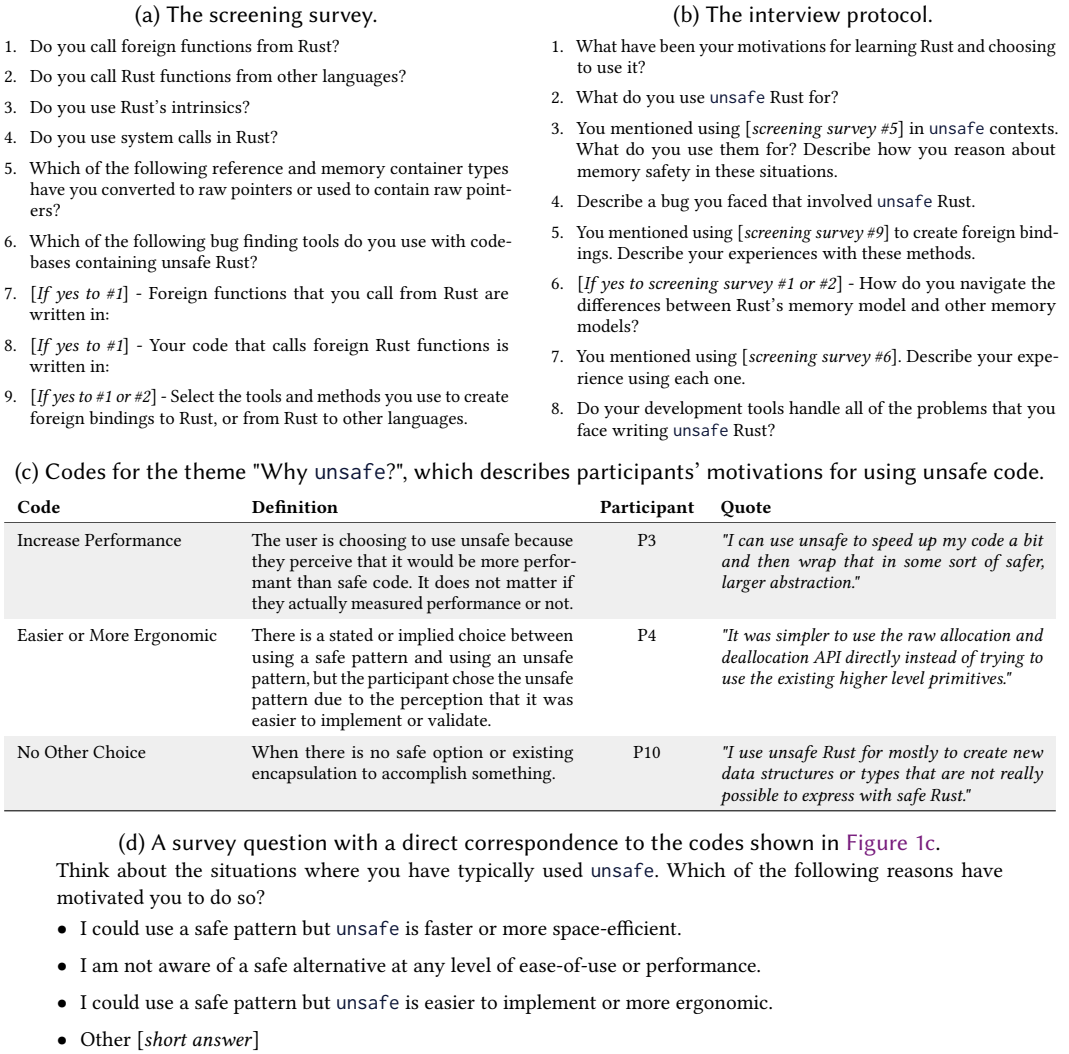
3.1 Interviews

On May 3rd, 2023, we posted a call to participate in semi-structured interviews on the Rust subreddit ([/r/rust](#)), the Rust Programming Language forums, and the #dark-arts channel of the Rust Community Discord server. We also contacted acquaintances with relevant experience. To be eligible to participate, candidates needed to have at least one year of experience using Rust and had to have “regularly written or edited Rust code within an unsafe block or function.” Eligible candidates completed a series of multiple choice and short answer questions about their use of unsafe features and development tools. We provide a sample of these questions in [Figure 1a](#). We also collected participants’ years of experience using Rust and we asked them to describe their affiliation. The first and third authors collaboratively coded each affiliation as one or more of the following categories: “Industry,” “Rust Team,” “Academia,” and “Open Source.”

We used participants’ responses to the screening survey to guide each interview. We provide a sample of our interview protocol in [Figure 1b](#). Certain questions were gated based on prior responses. If an candidate answered “No” to questions #1 and #2 of [Figure 1a](#), then they did not use Rust in multilanguage applications, so we did not ask them questions #7-9 of the screening survey or question #6 of the interview protocol, since each of these questions involve foreign functions.

The first author conducted all 19 interviews remotely over Zoom between May and July of 2023. Participants were not required to enable their camera. We used snowball sampling [48] to expand our pool of participants beyond the group that responded to our initial calls to participate. If a participant shared new information that we had not learned yet, then we prompted them to invite any acquaintances with relevant experience to complete our screening survey. We stopped recruiting new participants after we had reached saturation. Our concept of saturation corresponds to Saunders’ et al [69]’s definition of *data saturation* as a subjective judgement made during data collection before analysis. We knew that we had reached this point after we began to “hear the same comments again and again” [21] and had informally identified several themes from our interviews. Two themes were particularly notable at this stage; certain participants felt that Rust’s Box could be cumbersome due to its noalias semantics ([Section 5.2](#)), and that Miri was difficult to use in

Fig. 1. A sample of materials relevant to each stage of our methodology. This includes the screening survey, interview protocol, an example of a theme from our codebook, and a survey question connected to that theme. Notes [in brackets] are documentation and were not presented to participants.



multilanguage applications (Section 5.3). Our sample size (19) is comparable to other interview studies with Rust developers [16, 22]. We transcribed the audio recordings of the interviews using Whisper [51], and the first author reviewed every transcript to correct errors and label whether the utterances came from the interviewer or the interviewee.

3.2 Qualitative Analysis

We conducted an inductive, thematic analysis of our interview transcripts following best-practices by Miles, Huberman, and Saldaña [44]. To begin, the first three authors conducted open coding of

random samples of quotes taken from each interview. To create a sample, the first author picked random character indices within each of the interview transcripts and chose a snippet of one of the interviewees’ responses near each index. If all responses at that index had been sampled before, then the first author generated a new random index and repeated this process until a new quote was found. We coded three samples of quotes and met after each sample to resolve disagreements and record new codes within a shared codebook.

Next, the first and second authors used a card-sorting [5] exercise to identify higher-level themes among the codes that we had generated in the first phase. We created cards for each of the quotes randomly sampled in the first phase, as well as for a set of quotes that were highlighted by the first author when reviewing each transcript. We used the themes that we created from this process to organize our codebook, and we removed redundant or irrelevant codes. The first and second authors used this codebook to conduct closed coding of seven additional random samples of quotes, meeting after coding each sample to resolve disagreements and update the codebook. We used the same process as before to create samples, but instead of treating each interview transcript separately, we sampled quotes from a single document containing the concatenated text of every interview transcript. Table 1c shows one of the themes that we created during this process, as well as its constituent codes.

After this refinement stage, the first and second authors divided the set of transcripts between each other to code in full, independently, using ATLAS.ti Web [19]. The first author coded 11 transcripts, while the second author coded the remaining 8 transcripts. Throughout this process, both authors continued to update a shared codebook with additional codes and themes to describe new observations. After all transcripts had been coded by one of the two authors, the first author conducted an additional coding pass for every transcript and resolved each of their disagreements with the second author. The complete codebook and all coding decisions are included in our replication package.

3.3 Survey

We constructed the community survey after we had completed coding every interview, but before the first author had finished auditing coding decisions. Each of the first three authors independently assembled a list of survey questions, which the first author assembled into a single survey in Qualtrics [50]. Excluding consent and eligibility, the survey had 85 yes/no, multiple-choice, and Likert scale questions, which were grouped into 26 sections. To avoid priming participants to select certain responses, we randomly shuffled the order of multiple choice and yes/no options and randomly reversed the order of Likert scale items. Similar to our screening survey, certain questions were gated based on prior responses. We provide a complete copy of our survey, including its branching logic, in our Appendix [41].

Questions were either descriptive or exploratory. Figure 1d shows an example of a descriptive question. Each response option corresponds to the codes shown in Figure 1c for the theme “Why unsafe?”. Exploratory questions arose after we had finished conducting interviews, and they did not have a direct correspondence to the themes that we identified during qualitative analysis. For example, Höltervennhoff et al. [22]’s study was published concurrent to our investigation, and three of their participants mentioned using either `cargo-audit` or `cargo-deny` to detect vulnerabilities in their dependencies. None of our interview participants had mentioned using these tools, but we included a survey question asking if respondents had used them before. We also relaxed our eligibility criteria for the survey to include individuals who would have been eligible for Höltervennhoff et al. [22]’s study. Candidates were eligible to participate in their study if they had committed unsafe code to GitHub, but there was no indication if they had done so regularly. Our interview participants needed to have “regularly” engaged with unsafe code, but our survey

respondents were eligible if they had “written, edited, read, audited, or engaged with unsafe Rust code in any way.”

Two of our interview participants piloted an initial draft of the survey, and we incorporated their feedback into the final version. We distributed our community survey on September 22nd, 2023 to the same acquaintances and communities where we distributed our screening survey. We also published a link to our survey in the September 28th edition of the “This Week in Rust” newsletter and we advertised the survey during a presentation and poster sessions at an industry partners conference hosted by our institution. At the end of the survey, participants could provide a link to their profile on either GitHub or the Rust Programming Language Forum to have a chance to receive one of two \$250 gift cards. We randomly selected two participants who had provided a link to their profile with account activity prior to the start date of the survey.

We configured Qualtrics [50] to reject multiple responses from the same individual. We also enabled reCaptcha [37] and RelevantID [53] to detect possibly fraudulent submissions. RelevantID uses several forms of metadata, including location and IP information; none of this information was retained or visible to the investigators. In our analysis, we excluded responses that had a reCaptcha score of less than 0.5, indicating that the response was likely from a bot. For RelevantID, we excluded responses that had a fraud score greater than 0.3 or were marked as likely to be duplicate.

4 Ethics & Threats to Validity

Ethics. All procedures in this study were approved by our institution’s IRB. Before each intervention, participants were read or presented a script outlining the procedures of the study and they were asked to affirm consent. Participants were reminded to avoid sharing personally identifying information in their responses, and interview participants were asked to remain in a private space. The first author reviewed the contents of the replication package and redacted all direct (e.g. name, affiliation) and indirect (e.g. position, project) personal identifiers.

External Validity. Recruitment via snowball sampling and through online communities may bias our results toward certain areas of the Rust community. We attempted to maintain a diverse survey and interview population by sampling from multiple forums. However, both Reddit and “This Week in Rust” were overrepresented compared to the other communities where we distributed our survey. Our sample does not include developers who are not active in these communities. We cannot determine if our interview population has a similar bias, since we did not record where these participants had first learned about our study.

The “silent” developers excluded from these populations may have different experiences and perspectives that are not represented in the current findings. Rust’s unsafe “superpowers” are typically advertised as an advanced feature that developers “might run into every once in a while, but may not use every day” [55]. Developers who have engaged with unsafe code may more likely to have had more experience using Rust in any capacity than those who have not used unsafe code. We suspect that committed Rust experts are overrepresented in our interview population, since many were affiliated with prominent open source projects or had experience using the language within security-critical applications.

It is more difficult to evaluate the capabilities of our survey respondents. There was significant variance in the reported number of years of experience. However, on average, our survey population had nearly the same number of years of experience using Rust as our interview population, and more than a decade of experience within the field of software engineering. Most respondents reported behaviors that indicate non-trivial experience with Rust, such as using bug-finding or auditing tools and committing unsafe code to a published library. For this reason, we expect that

our current results reflect the “best-case scenario” for developers’ experiences using unsafe Rust. Any aspects of Rust development that our population found to be difficult seem likely to be even more challenging for the broader Rust community.

Additionally, the gift cards that we offered for compensation are most easily redeemable in North America, which may have dissuaded certain individuals from participating in either stage of our study. We did not collect location data from participants, so we cannot determine if our results are biased toward individuals from any particular area.

Construct Validity. Respondents were eligible to complete our survey if they had one year of experience and had “engaged” with unsafe code in any way. A subset may have been eligible without having significant practical experience with unsafe code. We mitigated this by gating questions based on whether participants had used certain features. When relevant, we compare results from the entire population with results from the subset who “regularly” wrote or edited unsafe code, which matches the eligibility criteria for our interviews. This criteria may have introduced self-selection bias among our interview participants, since individuals who frequently use unsafe code may be more likely to perceive that it is necessary for their purposes. We did not enable reCaptcha [37], RelevantID [53], or any of Qualtrics’ other safeguards when recruiting participants for interviews. These safeguards are subject to false positives [81], and may have flagged legitimate participants.

The validity of our results depends on participants’ ability to accurately self-report their behavior, and what developers perceive about their use of unsafe code may differ from how it is used in practice. We did not evaluate whether participants were able to reason correctly about unsafe code; only how they perceived their usage of these features. Höltervennhoff et al. [22] found that developers who use unsafe code have varying degrees of awareness about its implications for correctness; 10 of their interview participants indicated that unsafe contexts entirely disable Rust’s static safety restrictions, which is incorrect. Our qualitative results are also unavoidably influenced by the individual biases of each investigator. We tried to mitigate this by auditing coding decisions and meeting frequently to resolve disagreements. Although most of our interview protocol was written to be neutral toward unsafe code, certain questions may have biased participants toward providing responses that favored safe coding practices, such as asking participants to describe how they “reason about memory safety” and “navigate the differences” between Rust and other languages.

5 Results

We report the demographics of interview and survey participants in Section 5.1. Then, we describe how these developers reason about foreign functions in Section 5.2 (RQ1), and we discuss the tooling they use in Section 5.3 (RQ2). We describe developers’ motivations for using unsafe code in Section 5.4 (RQ3), and we report how they reason about encapsulation in Section 5.5 (RQ4). We summarize key results at the end of each section.

5.1 Demographics

Table 1 provides summary statistics on interview and survey participants. We invited all 42 eligible candidates to participate in interviews, and we interviewed 19 who responded to our invitation. Table 1a summarizes interview participants’ demographic information. There were 15 who had experience in industry, six in open source development, three in academia, and two were members of the Rust Team. They had 4.7 years of experience with Rust on average, and the most experienced participants had been using Rust for 10 years. We did not collect any additional demographic information from our interview participants.

Table 1. Demographics of interview and survey participants.

(a) Interview participants' affiliations ("Affil.") and years of experience using Rust ("Years") by ID.

ID	Affil.	Years	ID	Affil.	Years
1	I	2	11	I	5
2	I,R	10	12	I,A	6
3	I,O	5	13	I	10
4	I,O	8	14	R,O,I	7
5	I,O	2	15	I	1.5
6	A	3	16	I	3
7	I	6	17	A	4
8	I	5	18	I	3
9	O	2	19	I	4
10	O	2			

I = Industry R = Rust Team
A = Academia O = Open Source

(b) The mean (μ) and standard deviation (σ) of survey respondents' years of experience in the field of software engineering ("SE"), as well as using Rust, C and C++. Entries are grouped by where respondents indicated that they had first heard about the survey. Respondents could select multiple options.

Location	#	SE		C		C++		Rust	
		μ	σ	μ	σ	μ	σ	μ	σ
Rust Forums	16	17.3	10.1	7.5	6.8	5.6	6.2	4.5	2.4
Rust Discord	10	8.0	3.7	2.7	2.8	3.4	2.3	3.6	2.0
Reddit (/r/rust)	83	13.9	9.1	7.6	7.2	6.0	6.3	4.1	2.2
"This Week in Rust"	52	15.1	9.7	6.6	7.7	5.7	6.9	4.1	2.3
Other	3	23.7	15.8	17.3	14.6	9.0	5.6	3.7	1.5
All	160	14.5	9.5	7.2	7.5	5.8	6.3	4.1	2.2

We received 368 survey responses, of which 240 were complete, 203 met our eligibility criteria, and 160 passed each of our measures for fraud detection. Hereafter, when we refer to “respondents,” we include only the 160 that passed each of our checks. We collected several categories of demographic information from survey respondents. The majority were young, educated, identified as male, and had experience in either industry or open source development. Nearly 66% had completed at least a bachelor’s degree, and 30% had completed a graduate degree. Most of the respondents (83%) were under 40 years of age, and 56% were between the ages of 18 and 29. We followed Spiel et al.’s [71] guidelines for ethically surveying respondents’ gender; 82% identified as “Man”, 6% identified as “Non-binary”, 3% identified as “Woman” and 9% did not disclose their gender. Table 1b summarizes respondents’ years of experience with the field of software engineering, Rust, and other systems languages. On average, respondents had more than a decade of experience in software engineering, more than 5 years of experience with C and C++, and just over 4 years of experience using Rust.

Participants had a variety of roles within the Rust community and differing levels of engagement with unsafe code. Among survey respondents, 74% had committed unsafe code to a public GitHub repository, and 66% had contributed to a published crate. Our interview population had regularly written or edited unsafe code, but 47% of our survey population did not. Since “regularly” is subjective, we asked the questions shown in Table 2 to provide a more specific time-frame and to distinguish editing unsafe code from practices that do not require code contributions, such as auditing. Only 26% (2b) of survey respondents would write or edit unsafe code more than once a month, but 41% (2a) would *engage* with it at this frequency.

Table 2. How frequently survey respondents “engaged with” and wrote unsafe code. Percentages on the left include “Less than once a year” and “Yearly”, while those on the right include “Weekly” and “Daily”. The number of respondents in the sample is listed in parentheses on the righthand side.

ID	Question	Sample	Distribution
2a	How frequently do you engage with unsafe Rust code in any way?	All	23% 41% (160)
2b	How frequently do you write new Rust code or edit existing Rust code within an unsafe block or function?		31% 26% (160)

Less than once a year Yearly Monthly Weekly Daily

5.2 RQ1 - Interoperation

The majority of interview participants and 70% of survey respondents used unsafe code to call foreign functions. The five most popular languages that respondents interoperated with were C (90%), C++ (57%), JavaScript or TypeScript (22%), Assembly (21%), and Python (17%). With the exception of Assembly, the relative popularity of each language is similar to results from Fulton et al. [16]. Their survey included Assembly as part of the "Other" category, which only 7% of their respondents had selected. Although 72% of our respondents had converted part or all of an application written in another language to Rust, 25% of these respondents had not called foreign functions. This subset might have ported these applications wholesale instead of opting for an incremental approach.

Aliasing & Memory Management. Rust’s standard library provides several types that can be used to manage ownership of heap allocations. Each of these types can be converted to raw pointers when necessary, but similar to Rust’s safe references, they have distinct invariants that must be maintained to avoid undefined behavior. Participants frequently used these types to allocate memory for foreign function calls, converting them into raw pointers in the process. In particular, 54% of survey respondents who had called foreign functions also used one of Rust’s memory container types for this purpose. Rust’s thread-safe `Arc`—an atomic, reference-counted allocation—was used by 28% of these participants, while only 8% had used the non-atomic `Rc`. Rust’s `Box` provides unique ownership over a heap allocation, and it was used by 95% of these participants to allocate memory for foreign calls. One interview participant used `Box` extensively when porting an online multiplayer text-adventure game from C to Rust. The game’s logic was written in C, but all heap memory was allocated in Rust using `Box`. When a `Box` was unwrapped and exposed as a raw pointer to C, its address was added to a table. The table was checked and updated to prevent double-free errors.

Several interview participants mentioned having issues with this type, noting that when a `Box` is moved, any raw pointers to its contents become invalid. Participants with a background in C++ development had not expected `Box` to behave this way. They were accustomed to using `unique_ptr` from the C++ standard library. This has a similar role, but different semantics:

You give the raw pointer off to the interface, and then you relocate the unique pointer, but you still rely on that pointer having valid access tags. (P9)

Survey participants did not encounter this issue as frequently. Of the 51% of respondents had used `Box` in any unsafe context, only 17% had encountered issues when a pointer became invalid after a `Box` was moved. However, we did not determine if respondents had actually unwrapped a `Box` or otherwise retained a raw pointer to its contents, so this 17% could still represent a significant portion of the respondents who would have been able to encounter this problem. Our population is most representative of experienced Rust developers, so it is possible that the average Rust developer is even more likely to have misused Rust’s `Box` type in these contexts.

One participant observed several aliasing patterns in foreign libraries that were at odds with Rust’s restrictions. They struggled to encapsulate foreign functions that would read from a “source” pointer and write to a “destination” pointer. It was difficult to use safe references as parameters for the Rust encapsulations of these functions when the source and destination pointers needed to be aliases. The borrow checker would prevent the mutable destination reference from aliasing with the immutable source reference. Interview participants also noted that C allows unrestricted integer-to-pointer conversion, but Rust does not have an agreed-upon semantics for this behavior [26]. Another participant observed that C libraries would often use on “a rat’s nest of pointers” (P14) to encode self-referential structures, which were difficult to represent in Rust encapsulations.

To manage these differences, participants reviewed the documentation and implementation of foreign libraries to determine their safety requirements. However, one participant found that documentation was frequently missing, so they had to manually reason about these properties.

It's really difficult to figure out a lot of the properties of a given codebase...are you actually mutating something?...if I give you some input data, are you going to retain a pointer to that and keep operating on it later?...it's a very human issue of lack of documentation and lack of correctness within a codebase. (P11)

This pattern matches two of the aliasing violations observed by McCormack et al. [42], where "unique" mutable references were cast into raw pointers and stored into the heap by foreign functions.

The reverse of this situation occurred when participants exposed Rust libraries to languages with fewer safety guarantees. Properties that were implicitly enforced by Rust's type system became documentation in other languages.

In Rust, you can trivially say, I'm going to return to you a thing, which just borrows my memory, and then you just can't access me while you're using that ...this is not very easily mappable into a C API, or rather you could do it, but then you have to read all this documentation... (P13)

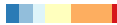
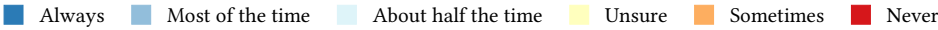
Some foreign APIs were structured in a way that made memory management easier. One participant found that "there [were] no complex lifetimes involved" (P11) in their encapsulation of an embedded library. Others observed that the properties they typically associated with a "well-crafted C and C++ API" (P15) made it easier to create a Rust encapsulation. For example, one participant observed that C APIs tend to expose pairs of initialization and cleanup functions for each type, which could be linked into the behavior of the Drop trait for its Rust encapsulation. This trait provides a cleanup function that is called to free the resources associated with an object when it leaves scope.

Concurrency. Interview participants typically used more sophisticated multithreading patterns in the Rust components of multilanguage applications. One participant used granular locking within the Rust codebase, while the C codebase had a singular global lock. For another participant, the C++ component of their application was structured along a single main thread, while the Rust component used multithreading freely—"we were throwing threads around all the time" (P2). It was easy for this participant to accidentally call into C++ from the wrong Rust thread, but they saw this as less challenging to manage than it would have been to implement the entire application in C++:

It was a validation of a belief...that this would have been impossible to do in pure C++...we had to hit this [problem] for the occasional times we had to call out to C++ from Rust...but if we were only writing C++ code... we would have been hitting this constantly. (P2)

However, Rust's approach to concurrency also made it difficult for one participant to encapsulate a foreign library. In Rust, a type can automatically derive the traits Send and Sync if all of its components implement these traits. However this participant observed that in C and C++, a single type can have methods and components with varying thread-safety properties: "Rust enforces thread safety based on types... C tends to do it based on functions" (P14). They found that it was more difficult to encapsulate foreign types that were only partially thread-safe.

Table 3. How often survey respondents would avoid passing abstract data types by value to foreign functions and converting foreign raw pointers into safe references. Percentages on the left include “Always” and “Most of the time”, while those on the right include “Sometimes” and “Never”. The number of respondents in the sample is listed in parentheses on the righthand side.

ID	Question	Sample	Distribution
3a	How often do you intentionally avoid passing Rust’s abstract data types (structs, enums) by value across FFI boundaries?	Used FFI	49%  24% (112)
3b	How often do you intentionally avoid converting raw pointers to memory allocated by FFI calls into safe references, such as &T or &mut T?		23%  43% (112)
			

Information-Hiding. Participants attempted to keep interoperation with foreign code as straightforward as possible. When declaring bindings, one participant preferred using primitive types and pointers instead of abstract data types like `Option` and `NonNull`, even though Rust allows these types to be implicitly cast into raw pointers [9]. They felt that these casts hid useful contextual information, so they preferred to use raw pointers explicitly. A few participants indicated that they had avoided passing structs by value across foreign boundaries in certain situations. One of these participants observed that certain Rust programs would behave incorrectly when structs were moved across a foreign boundary instead of copied, or if they had a destructor on the C++ side of the boundary. They typically avoided passing structs by value in either situation, and they recommended that “...if you’re doing FFI, the only things you should pass by value are copy types” (P2). Other interview participants also indicated that they felt comfortable passing structs by value as long as they had copy semantics and equivalent types on each side of the boundary.

In addition to limiting the use of certain types at boundaries, interview participants also attempted to minimize the interactions between each side of the foreign function boundary. They thought that a “very chatty API” (P7) would be difficult to encapsulate correctly. One participant described this complexity in terms of the heap object graph that a foreign library exposes to Rust. When the structure of the foreign heap was exposed to Rust in detail, safe encapsulations became difficult to use:

It’s hard to keep those objects around for very long unless you want all of your structs to end up with a bunch of lifetimes... you don’t want that... it’ll scare away new programmers for sure. (P14)

They typically avoided casting foreign pointers into Rust’s safe reference types since “applying lifetime semantics to foreign code” (P4) required difficult, manual reasoning. Another participant used opaque types to hide the underlying layout of foreign values. This design pattern is useful in situations where Rust needs to be able to reference a foreign value, but the value does not need to be accessed on the Rust side of the boundary. Instead of declaring the structure of its type on each side of the boundary, a value can be given a zero-size, “opaque” placeholder type in Rust.

We investigated these information-hiding practices using the survey questions shown in Table 3, where respondents indicated if they “intentionally avoid” passing structs by value or casting foreign pointers into Rust’s references. We added “Unsure” as a response option for these questions since they indirectly rely on participants having a notion of why they would “intentionally” avoid these

behaviors. Neither practice was consistently followed, but slightly more respondents would avoid passing abstract data types by value most or all of the time (49%, 3a), than would avoid casting foreign pointers into references (43%, 3b).

Key Findings - RQ1 (Interoperation).

How do Rust developers reason about memory safety across foreign function boundaries?

The majority of participants called foreign functions, which were typically written in C and C++. Certain foreign libraries used aliasing and concurrency patterns that conflicted with the expectations of Rust’s type system, which made them difficult to encapsulate. To make interoperation easier, participants minimized their interaction with foreign libraries, but they frequently used Rust’s memory containers as allocators, and few actively avoided casting foreign pointers into safe references. Developers identified safety properties and design patterns of foreign APIs that match the characteristics of libraries where McCormack et al. [42] detected aliasing violations, indicating that foreign function calls may be a significant unchecked source of undefined behavior in the Rust ecosystem.

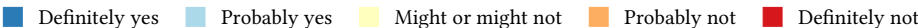
5.3 RQ2 - Tooling

Participants used a wide variety of development tools to assist with validating their design choices. We focused primarily on dynamic bug-finding tools and static tools for generating bindings to

Table 4. How often survey respondents audited their dependencies, used a debugger, wrote tests, and executed tests in Miri (if they had used Miri at least once). Percentages on the left include “Always” and “Most of the time”, while those on the right include “Sometimes” and “Never”. The number of respondents in each sample is included in parentheses on the righthand side.

ID	Question	Sample	Distribution
4a	How often do you write tests for Rust applications that use <code>unsafe</code> ?	All	61% 30% (160)
		Used Miri	72% 19% (98)
4b	How often do you run test cases in Miri?	Used Miri	38% 56% (98)
4c	How often do you audit your dependencies' use of <code>unsafe</code> ?	All	6% 88% (160)
		Used Auditing Tools	10% 80% (81)
Always Most of the time About half the time Sometimes Never			

Table 5. Whether or not survey respondents who had used foreign functions would trust the correctness of foreign function bindings written by hand and generated using a tool. Percentages on the left include “Definitely yes” and “Probably yes”, while those on the right include “Probably not” and “Definitely not”. The number of respondents the each sample is included in parentheses on the righthand side.

ID	Question	Sample	Distribution
5a	Do you trust FFI bindings that are written by hand?	Used FFI	45%  7% (112)
5b	Do you trust FFI bindings that are generated by a tool?		79%  2% (112)
			

foreign functions. However, certain participants also described their experiences using debuggers, verifiers, and auditing tools.

Dynamic Tools. Most participants used dynamic bug finding tools. Interview participants often mentioned Miri [29], AddressSanitizer [70], and Valgrind [45]. We prompted survey respondents to indicate which dynamic tools they had used from a list of options mentioned by our interview participants. Survey respondents most frequently used Miri (61%), Valgrind (44%), AddressSanitizer (26%), and cargo-fuzz [62] (26%). Each of the remaining tools listed in our survey was used by less than 15% of respondents. Three interview participants and 31% of survey respondents used some form of fuzzing tools. One interview participant leveraged industry sponsorship to fuzz a JIT compiler for hours-on-end. Another participant implemented a BTree data structure with optimizations for high performance computing, and they used libfuzzer [39] to perform differential testing [43] against the implementation provided by Rust’s standard library. Of the 26 developers interviewed by Höltervennhoff et al. [22] only seven had used Miri, seven used fuzzing tools, and three used Valgrind. Our results indicate that this is not representative of a broader trend in the Rust community.

The majority of survey respondents also wrote tests for applications that used unsafe code. As shown in Table 4, 61% (4a) of all participants wrote tests most of the time or always. This was equally true for the subset of participants who had used Miri; 72% would write test cases at least most of the time. However, 56% (4b) of survey respondents who had used Miri only sometimes used it to run their test cases, if at all. Höltervennhoff et al. [22] also had similar findings. The majority of their participants wrote unit tests for unsafe code, but a subset claimed that they would infrequently run these tests because foreign function calls or low-level hardware interaction made testing “impossible.” While notable, we cannot make any causal claims between Miri’s limitations and this lack of testing in our survey population. Additionally, we did not ask if survey respondents regularly ran their test cases. However, interview participants frequently mentioned that Miri’s slow performance and lack of support for foreign function calls prevented them from using it to find bugs in certain categories of applications, and survey respondents who had used Miri encountered similar issues. The majority (62%) were deterred from using it due to one or more of the following issues: lack of support for foreign functions (43%), slow performance (26%), or lack of support for inline assembly (19%). Developers interviewed by Höltervennhoff et al. [22] had also struggled with Miri’s lack of support for key features, but none had mentioned its performance.

Notably, one of our interview participant had neglected to use Miri since their application relied heavily on foreign function calls: “we’ve assumed that we are a JIT and Miri can’t run JIT code, so therefore it’s useless” (P13). However, after a vulnerability was discovered in their codebase, they decided it that it would be worthwhile to run Miri on the subset of their test cases that did not require foreign function calls. They discovered that Miri would have been able to detect the vulnerability earlier, since it was caused by a violation of Rust’s aliasing model. Another interview participant managed to circumvent Miri’s limitations by creating mocks of foreign functions, but this solution did not scale to programs with “50,000 lines of code” (P4). This participant and a few others were aware of the Krabcake project [31], which aims to extend Valgrind with Miri’s borrow tracking features. Each of these participants was already using Valgrind with Rust codebases, and they felt that Krabcake would be an ideal solution for overcoming Miri’s limitations.

Foreign Function Bindings. Interview participants encountered several issues with tools that generate bindings to foreign functions. Participants reported that bindgen [65] added significant runtime overhead during build times and was not as expressive as they desired. One participant observed that bindgen “did some weird stuff for anonymous unions and structs” (P17) but that this was easy to fix by modifying the tool’s configuration. However, another participant found it

necessary to switch from using `bindgen` to writing bindings by hand, because it was not capable of expanding the macros that they had used to define core data structures. Others felt that the bindings they wrote by hand were more idiomatic than the output of a tool. The majority (58%) of survey respondents who created bindings used multiple methods, while 28% only generated bindings and 14% only wrote them manually.

Interview participants who wrote bindings by hand viewed this process as error-prone, noting that bindings become a “*fiddly mess*” (P5) when you need to ensure that interfaces remain consistent on each side of the foreign boundary. These errors clearly do occur in practice, as 35% of survey respondents who had called foreign function had also encountered incorrect bindings. McCormack et al. [42] also found several instances of incorrect bindings in Rust libraries. Some were intentional—developers neglected to add return types for values they did not use—but others introduced aliasing violations or uninitialized reads. Survey respondents who had called foreign functions were somewhat more likely to trust generated bindings. As shown in Table 5, only 45% (5a) of respondents would at least “probably” trust handwritten bindings, compared to 79% (5b) who would trust generated bindings. However, developers were still more likely to trust handwritten bindings than to distrust them.

Debugging. Few interview participants reported using a debugger with Rust code. Only 10% of survey respondents used a debugger daily, 29% used one monthly, and 44% used one yearly at most, if at all. One interview participant had used both MSVC’s debugger and LLDB [38], and while both worked, neither matched the quality of the other parts of the Rust toolchain, “*where you get all the bells and whistles that you want, and things just work nicely together*” (P3). Mozilla’s RR debugger [46] was useful for one participant to resolve bugs found from fuzzing, but it was generally more common for participants to report using “*printf-style*” (P7) debugging.

Formal Methods. Few interview participants and only 10 survey respondents used tools that apply formal methods for static and dynamic verification. Interview participants mentioned using Kani [72], Prusti [3], and Crucible [18]. Kani was useful for one participant, who needed to prove that a particular routine would never execute more than twice for any input. Another participant who had “*tried a bunch of the tools*” in this category found that Prusti was the most complete, but they felt that it would be more effective if these tools were “*inherently parts of Rust*” so that they could “*hoist a lot of currently unsafe code into safe code*” (P11). Of the ten survey respondents who had used “formal methods tools,” five had used Kani, three had used Prusti, three had used Creusot [13], one had used Flux [35], and one had used Crucible. We provided a short-response option for this question to include any tools that were missing from our list.

Auditing. Only a few interview participants indicated that they had ever audited their dependencies. Höltervennhoff et al. [22] had similar findings; half of the developers that they interviewed had deliberately included dependencies with unsafe code, but the majority did not audit their dependencies. One of our participants would examine their dependencies if they were curious about how they functioned, and they actively avoided adding dependencies that used outdated versions of Rust or deprecated features. Another participant was in the process of having a third-party institution examine their codebase to determine if they could reduce their amount of unsafe code.

Survey respondents rarely audited their dependencies’ use of unsafe code. More than 80% (4c) of respondents only sometimes audited their dependencies’ use of unsafe code, if at all. It is possible that a subset of the participants who answered “Never” to this question did not have any dependencies to audit. However, 51% of our respondents had used automated auditing tools, compared to four interviewed by Höltervennhoff et al. [22]. Of this subset, 63% had used `cargo-audit` [63], 42% had

used cargo-update [60], 31% had used cargo-deny [58], 22% had used cargo-geiger [59], and only 7% had used cargo-vet [61].

Key Findings - RQ2 (Tooling).

What tools do Rust developers use when contributing to applications that include unsafe code, and how could tooling be improved?

Miri appears to be the de facto bug finding tool for the Rust ecosystem, since it was used by 61% of all respondents. However, 62% of the respondents who had used Miri before were deterred from using it again due to its performance and lack of support for key features—predominately, foreign function calls. Tools for generating foreign function bindings were often slow and required additional configuration, so a subset of developers still preferred writing bindings by hand. Participants infrequently used debuggers, and only 10 survey respondents had used formal methods tools. Survey respondents infrequently audited their dependencies, but 51% had used one or more auditing tools. Improvements to Miri may be the most effective way to provide stronger safety guarantees for the majority of our audience, since it was the most widely used tool in our survey.

5.4 RQ3 - Motivations

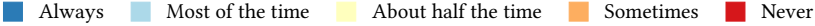
Participants used unsafe code because they perceived that it was more performant or ergonomic than safe alternatives, or that there were no safe alternatives at all. Here, we refer to these motivations as “Performance,” “Ergonomics,” and “Necessity,” which correspond to the codes “Increase Performance”, “Easier or More Ergonomic,” and “No Other Choice” from Figure 1c, respectively. We used the question shown in Figure 1d to measure the distribution of each of these motivations in our survey population. All but one participant chose to answer this question. Results are shown in Table 6. Necessity was the most common motivation, followed by performance and ergonomics.

Performance. Some interview participants and 47% of survey respondents indicated that they would use unsafe code when they perceived that it was faster or more space-efficient than an equivalent

Table 6. Survey respondents’ motivations for using unsafe code, as measured by the question shown in Figure 1d. All but one participant chose to answer this question. Each cell shows the percentage of participants who identified with both of the motivations for that row and column. All three motivations were selected by 7% of participants.

	Necessity	Performance	Ergonomics
Necessity	77%	92%	86%
Performance		47%	53%
Ergonomics			18%

Table 7. How frequently participants measured the impact of their changes when using unsafe code to improve performance. Percentages on the left include “Always” and “Most of the time”, while those on the right include “Sometimes” and “Never”. The number of respondents in the sample is included in parentheses on the righthand side.

ID	Question	Sample	Distribution
7a	When you choose to use unsafe because it performs faster or is more space efficient, how often do you measure the difference?	Used unsafe for performance	46%  36% (76)
			

safe API. Participants usually identified a safe alternative in these situations, but they felt that it would be too expensive to use: “so you could do it, right? But that would be some performance overhead” (P12). Most interview participants who attempted to use unsafe to improve performance had implemented small-scale optimizations in existing applications. These typically involved eliminating runtime checks, which were seen as unnecessary due to what they perceived were local or global invariants. For example, one participant used an unsafe API to consume a pointer to a string without checking if it contained valid UTF-8 characters. This pointer was provided by a foreign call to Python, and the participant thought that it would only ever provide text in a valid format. Another participant chose to store a heap allocation as a raw pointer in a static, mutable variable instead of using one of Rust’s safe static encapsulations, since they felt confident that the allocation would remain valid for the duration of the program.

However, a few interview participants used unsafe code to create entire components that were purpose-built to achieve performance. In each situation, performance was seen as a functional requirement for the application domain: “I write a serialization framework...and obviously a goal for it is to be extremely performant” (P3). A participant who took this approach found that it significantly increased the amount of unsafe code in their application, but they felt that this was a reasonable compromise to achieve greater performance. Survey respondents who reported using unsafe code to improve performance did not have a strong tendency toward either large-scale or small-scale use. We found that 26% only pursued “small-scale-optimizations in existing applications,” 20% only built new, “large-scale components,” and 42% used unsafe code for performance in either situation.

Participants did not consistently measure the performance impact of unsafe code. In some situations, they relied on their intuition to determine which design decisions would be best: “I’ll totally admit that...I think this is going to be a hot thing, so I’m going to kind of prematurely optimize” (P11). Survey respondents who used unsafe code to increase performance were also inconsistent about measuring its impact. As shown in Table 7, 46% of respondents reported measuring performance at least most of the time, while 36% only sometimes measured performance, if at all. In contrast, Höltervennhoff et al. [22] found that 6 participants would only use unsafe code to improve performance when the change was “noticeable.” We did not make a distinction between profiling and subjective measures of performance in our survey. However, one of our interview participants performed sophisticated profiling on three versions of a component, each of which had varying amounts of unsafe. The most unsafe-heavy version performed the best, but they opted for one with a moderate amount of unsafe code, balancing safety and performance concerns.

Ergonomics. Participants also chose to use unsafe code when they felt it would be easier or more ergonomic than using a safe API. This was the least common motivation cited by interview participants, and only 18% of survey respondents identified with this motivation. Höltervennhoff et al. [22] indicate that some of their participants would use unsafe code if it “saved them effort”, but it was unclear if this motivation was typical. Our participants usually stated or implied a choice between using safe or unsafe design patterns. For example, one participant chose to use only Rust’s unsafe allocator API rather than a mix of unsafe and safe allocation operations. They found that this approach was easier to reason about, since “if there’s too much abstraction, you lose the information that you need to make it actually sound” (P4). Several participants used the unsafe function `transmute` to perform unrestricted type conversion. Since types have differing invariants, `transmute` can only be used in unsafe contexts. One participant used transmutation to simplify the implementation of chess engine, which used three enumerations to represent the position of a chess piece. They found that it was easier to use integer conversion, bitwise operations, and transmutation to convert between these enumerations instead of safely matching on their values. Another participant used transmutation when working with a Rust encapsulation of a C font library.

Key Findings - RQ3 (Motivations).

What are Rust developers' motivations for using unsafe code?

Participants were most often motivated to use unsafe code because they felt that there was no safe alternative. In other situations, participants used unsafe code because it performed better or was more ergonomic. For performance, developers pursued both large-scale abstractions and small-scale optimizations, but they did not consistently measure the impact of their changes. Each of these motivations can influence a design decision simultaneously.

5.5 RQ4 - Encapsulation

Invariants. Participants reasoned about the soundness and correctness of their unsafe code in terms of invariants. When developing operating systems and embedded applications, these invariants were provided by hardware specifications. For example, one participant who developed an embedded application would write a “*magic value*” (P8) to RAM to indicate which operation had triggered a reboot. The microcontroller that they used did not erase its memory on reboot, so they could rely on the value being initialized. Some participants perceived hardware-level correctness properties as “*orthogonal to Rust*” (P1), since they were related to the properties of the language. However, other participants who developed embedded applications relied on third-party libraries to encode hardware requirements into the type system so that their programs would fail to compile if a device had been misconfigured.

Table 9. Survey questions relevant to RQ4. Percentages on the left include “Always” and “Most of the time”, while those on the right include “Sometimes” and “Never”. The number of respondents in each sample is included in parentheses on the righthand side.

ID	Question	Sample	Distribution
9a	When you use an unsafe API, how often do you look for documentation to ensure that you meet all of its requirements for safety and correctness?	Used unsafe APIs	85%  8% (152)
9b	When you expose a safe API for unsafe code, how often do you include runtime checks to ensure that its requirements for correctness and safety are met?	Exposed a safe API for unsafe code	58%  29% (141)
9c	When you expose an unsafe API to users, and it is their responsibility to ensure that certain requirements are met, how often do you document these requirements?	Exposed an unsafe API	90%  3% (68)
9d	When you expose an unsafe API to other users, and it is their responsibility to ensure that certain requirements are met, how often do you include runtime checks for these requirements?		18%  71% (68)
9e	How often do you refactor Rust applications to remove unsafe code?	All	6%  88% (160)
		Regularly wrote unsafe code	7%  85% (85)
9f	How often do you choose to avoid using an unsafe API when a safe alternative exists?	All	82%  10% (160)
		Used unsafe APIs	83%  10% (152)

 Always
 Most of the time
 About half the time
 Sometimes
 Never

Developers leveraged Rust’s type system to uphold invariants that they believed were necessary for soundness. In operating systems, these typically began as operation-specific refinement properties on primitive types, but they quickly built up into system-level invariants; “*it gets to a high-level really quickly*” (P3). Rust’s aliasing restrictions were a natural fit:

We only allow... someone who has access to one of these mapped memory regions to borrow it... That’s a great example of taking something that’s unsafe inherently and then kind of wrapping it up in a safe abstraction, using the power of Rust to do the borrow checking for us. (P12)

Another participant experimented with using ghost permissions to reason about pointer arithmetic. This method was popularized for Rust by Yanovski et al.’s GhostCell type [77]. Other participants relied on ad hoc reasoning to ensure that their programs were correct and free of undefined behavior. Some appealed to their prior experience: “[*in C++, you do this all the time*” (P15). One participant was accustomed to reasoning about aliasing rules in compiler development, so they felt confident that they were adhering to Rust’s restrictions in unsafe contexts. Other participants validated their decisions by auditing their code, but most had some level of implicit trust; “*that’s sort of another... I got to trust it type thing*” (P5). This aligns with Höltervennhoff et al. [22]—a majority of their participants reasoned about unsafe code in terms of contracts or invariants, but some reported using it carelessly, and most felt that it was difficult to write correctly due to its context-sensitivity.

Isolation. The majority of interview participants made a conscious effort to minimize and isolate unsafe code, which made it easier to document and reason about.

That’s generally the most important thing, just being self-contained, being well-isolated, being well-encapsulated... (P10)

Most survey respondents (61%) predicted that it would be at least somewhat easy for another developer at their skill level to understand a random unsafe block or function from their code. Interview participants who isolated their unsafe code had more confidence that its requirements were met, especially if they went beyond what Rust’s type system could verify. However, participants reported that unsafe code was prevalent in JIT compilers and operating systems. Domain-specific, non-local reasoning was necessary to understand any arbitrary unsafe snippet: “*you have to know the innards of the JIT-compiled function...so it’s just not encapsulated*” (P11). Though unsafe code may have been isolated, it is unclear if participants used it minimally. As shown in Table 9, more than 80% (9e) of survey respondents only sometimes refactored their code to remove unsafe features, if at all. This distribution was similar for the subset of respondents who had regularly written unsafe code.

Safe Interfaces. Most interview participants and 88% of survey respondents reported that they had exposed a safe API for unsafe code. This was also common for Höltervennhoff et al. [22], who found that 22 of their 26 participants mentioned safe interfacing. Our interview participants would create encapsulations where they perceived that the requirements for their unsafe code were satisfied by either Rust’s type system or what they reasoned to be local invariants.

However 43% of our survey respondents reported exposing unsafe APIs, relative to six of Höltervennhoff et al. [22]’s 26 interview participants. Our participants cited the same motivations for exposing unsafe APIs as they did for using unsafe code in any capacity. One participant exposed unsafe versions of safe API endpoints which had the same behavior, but they did not include runtime checks. Another interview participant exposed unsafe APIs so that users could circumvent

their safe encapsulations, in case they became difficult to use. Each of these practices aligns with another participant’s perception of a cultural value among Rust developers to make API surfaces as detailed as possible: “*Rust has a tendency to... make the API as complicated as it needs to be to fully represent what is actually happening behind the scenes*” (P1). Other participants indicated that they would only expose an unsafe API when it would be impossible to encapsulate without placing the burden of correctness on the user. Our survey respondents also identified with these motivations; 76% exposed unsafe APIs because it would be impossible to encapsulate them without preconditions, while 51% exposed unsafe APIs for performance and 13% did so for ergonomics. Höltervennhoff et al. [22] also found that participants would expose unsafe APIs by necessity or to improve performance, but it was unclear if ergonomics was also a motivation.

Regardless of motivation, 61% of respondents claimed that they would only expose unsafe APIs when they had requirements beyond what Rust’s type system could verify, and they typically documented these requirements. Results for Question 9c indicate that 90% of respondents who exposed unsafe APIs would document their safety requirements most of the time, if not always. However, respondents did not typically add any additional safeguards to prevent unsafe APIs from being used incorrectly, since 71% (9d) only sometimes inserted run-time checks within these APIs, if at all. This was more common when exposing safe APIs; 58% (9b) inserted checks at least most of the time when exposing a safe API for unsafe code. When using unsafe APIs, 85% (9a) of participants would usually look for documentation to inform their design decisions.

Interview participants perceived that the Rust community has a strong preference for safety, which affects tool design and development practices. Participants avoided exposing unsafe APIs in libraries partly because of the perception that other Rust developers would otherwise avoid using their crate: “*everyone wants a safe interface... no one really wants to go use the unsafe heavy ones*” (P9). One library author viewed safe encapsulation as an inevitable requirement. Even though exposing an unsafe trait places the burden of responsibility on users, they would still blame the library.

So in theory, the burden of proof is on them, but also if they implement something improperly, then they’re going to create a ticket because they’ll be like, oh, “I use your library and [this behavior is] a fault.” (P7)

An overwhelming majority of survey respondents also preferred safe APIs. More than 80% (9f) of respondents claimed that they would choose a safe API over an unsafe alternative most of the time, if not always—regardless of whether or not they had ever used an unsafe API before.

Uncertainty. Many interview participants who created safe APIs for unsafe code were uncertain if their encapsulations were sound in all possible situations.

No library author knows when they’re going to trigger [undefined behavior]... we do our best to give you a sound interface, but god knows if there’s a hole in it... (P9)

Höltervennhoff et al. [22] had similar findings; 16 of their 26 participants felt some degree of uncertainty about whether their unsafe code was correct. Our interview participants were most often uncertain when unsafe was pervasive or when they called foreign functions. When describing their uncertainty about encapsulation, our participants compared the “theory” of safety to the “practice” of how an API would be used. When these concepts were misaligned, errors occurred. One participant observed this mismatch with a Rust crate that encapsulated a foreign library. The library used a memory-mapped file, and the API made it possible to unmap the file while the user

Table 10. Survey questions relevant to RQ4. Percentages on the left include “Always” and “Most of the time”, while those on the right include “Sometimes” and “Never”. The number of respondents in each sample is included in parentheses on the righthand side.

ID	Question	Sample	Distribution				
10a	When you expose a safe API for unsafe code, how often are all of its requirements for correctness and safety satisfied by the properties of Rust’s type system?	Exposed a safe API for unsafe code	62%		18%	(141)	
10b	How often is the Rust community’s guidance and documentation adequate for you to know how to use unsafe correctly?	All	70%		16%	(160)	
Always Most of the time About half the time Unsure Sometimes Never							

retained a reference to it, leading to a use-after-free error. Another participant struggled with discrepancies between the interface and the implementation of Rust’s `Layout` API, which describes the size and alignment of an allocation. The design of `Layout` was changed to “*tweak the safety requirements*” (P14), which caused code that was previously correct to exhibit undefined behavior.

Interview participants also had a collective sense of uncertainty about Rust’s semantics. They perceived that Rust lacked a formal specification and that the guidelines for unsafe code were incomplete.

I think the biggest issue with Rust’s unsafe isn’t necessarily the tools, but just the spec—or, rather, the absence of the spec. If you write unsafe code today, you write it against the void. (P10)

A few interview participants encountered situations where the precise definition of undefined behavior was either unclear or overly restrictive. For example, one participant felt that it would be necessary to create a mutable reference to uninitialized memory. This is considered undefined behavior by Rust’s current semantics [14], but the participant was uncertain about how this pattern would be problematic in practice. Another participant wished to be able to take a reference to the first field of a product type and use it to create valid references to adjacent fields using integer offsets. This was considered undefined behavior under Rust’s Stacked Borrows [28] aliasing model, but it is accepted under the newer Tree Borrows model [74]. Additionally, multiple participants were uncertain about how to handle the case when a program panics while a value is being dropped.

Survey respondents were somewhat more certain about their decisions. As shown in Table 10, more than 60% (10a) of respondents who had exposed a safe API for unsafe code were certain at least most of the time, if not always, that its safety properties were satisfied by Rust’s type system. Additionally, 70% (10b) of respondents found that the Rust community’s documentation and guidance was adequate for them to know how to use unsafe code correctly at least most of the time. However, only 6% felt that it was always adequate, suggesting that improved documentation would be helpful for the overwhelming majority of respondents. Similar to the questions in Tables 5 and 8, we added “Unsure” as a response option for the questions shown in Table 10, since they rely on developers’ subjective notions of certainty and adequacy.

Key Findings - RQ4 (Encapsulation).

How do Rust developers reason about encapsulating unsafe code?

Interview participants derived requirements for unsafe code from external specifications and the inherent guarantees of the borrow checker. However, they also relied on ad hoc reasoning and tacit knowledge. Many were uncertain if their encapsulations were sound in all situations. Participants attempted to minimize and isolate unsafe code, and they avoided using unsafe APIs in favor of safer alternatives. However, this was not always possible in certain domain-specific contexts, such as JIT compilers and operating systems, where unsafe code was used pervasively. Survey respondents who exposed unsafe APIs documented their safety properties, and they relied on documentation to use unsafe APIs correctly. When our participants chose to expose unsafe APIs, they were motivated by performance, ergonomics, and necessity to the same extent as for any arbitrary use of unsafe code. Respondents rarely refactored unsafe code, even if they used it regularly.

6 Related Work

Here, we discuss relevant results from other studies of unsafe Rust code, comparing our results to quantitative findings where relevant. Most prior studies have examined unsafe code as it is written, either by surveying Rust’s package ecosystem [2, 15, 47] and standard library [12], or by examining vulnerabilities caused by unsafe code [42, 49, 76, 80, 82]. Other studies have collected blogs, articles, and forum posts to understand the implications of unsafe code for the Rust community. Zeng and Crichton [79] examined a set of articles and comments posted on Hacker News and the Rust subreddit to identify potential challenges to Rust’s adoption. Zhu et al. [83] examined posts on Stack Overflow to determine if developers frequently used unsafe code to resolve compilation errors. These studies provide meaningful contextual information that we can use to cross-validate our findings. However, none can fully answer our research questions.

Far fewer studies have used surveys or interviews to directly engage with developers who use unsafe code. As an extension to their survey of Rust libraries, Evans et al. [15] distributed a short survey on the Rust subreddit. They asked developers why they use unsafe code, which operations they use, and what steps they take to ensure that their code is correct; it received 20 responses. Zeng and Crichton [79] also interviewed three Rust contributors to provide additional context in their analysis. Fulton et al. [16] interviewed 16 developers who had advocated for using Rust in industry, followed by a survey that received 178 responses from many of the same communities as ours. Although they used a similar methodology, Fulton et al. [16] mainly focused on evaluating the benefits and drawbacks of Rust as a general-purpose systems programming language instead of examining Rust’s unsafe features. These studies provide evidence that can partially answer a subset of our research questions, but most do not share our direct focus on unsafe code or do not investigate challenges specific to tool use and interoperation.

Höltervennhoff et al. [22] are the only other group that we are aware of that has used qualitative methods to study how developers use unsafe code. Their study was published concurrent to our investigation; after we had completed our interviews but before distributing our survey. They interviewed 26 Rust developers who had experience using unsafe code. Except for three who piloted the interview protocol, all of their participants had contributed unsafe code to open source projects included in the unofficial “Awesome Rust,” list [8] of applications. We cover similar topics, but our research questions reflect different, complementary goals. They focused on understanding how developers reason about the security implications of using unsafe code, while we focused specifically on interoperation and tool use. We provide additional details on challenges associated with tool use and interoperation that are missing from their work, and we demonstrate that their results can generalize to a broader population of developers.

Here, we describe how our results compare with these prior studies for each of our research questions.

RQ1 - Interoperation. Interoperation has consistently been one of the most common use cases for unsafe code. In 2020, Astrauskas et al. [2] found that 44.6% of all publicly-exported unsafe functions in the Rust ecosystem were static bindings to foreign functions. Other surveys of the Rust ecosystem have had similar findings; Ozdemir [15] found that 30% of unsafe blocks were entirely dedicated to foreign function calls, and Evans et al. [15] found that 22.5% of unsafe function calls were to foreign functions. Zhang et al. [80] manually analyzed 5,946 unsafe blocks from 140 prominent libraries to determine which unsafe operations were necessary and which could have been avoided. The majority of all raw pointer dereferences and unsafe function calls were marked as necessary to interoperate with foreign libraries. Studies of developers have reached similar conclusions; 70% of Fulton et al. [16]’s survey respondents and all but one of Höltervennhoff et al. [22]’s interview participants had called foreign functions from Rust. Interoperation was also common among our participants; 70% of survey respondents had called foreign functions.

However, few studies have identified specific challenges associated with interoperation. Cui et al. [12] explicitly excluded foreign functions from their investigation of Rust’s safety properties. Xu et al. [76] found 12 examples of bugs and vulnerabilities that involved foreign function calls. They were related to application-specific invariants, data races, invalid alignment, and inconsistent layouts. However, the authors provided few details on the nature of these bugs, indicating that they were “straightforward” and not specific to Rust. Fulton et al. [16]’s participants had varying experiences with interoperation, but C++ was particularly difficult due to what one participant described as “rampant aliasing.” Höltervennhoff et al. [22] indicated that some participants struggled with foreign function calls that involved “memory management,” but that pure functions were easier to reason about. Our interview participants had similar experiences. Calling pure functions was typically straightforward, as long as the bindings were correct and ABI-compliant. When participants struggled to reason about foreign function calls, it was usually because these APIs had aliasing or concurrency patterns that conflicted with Rust’s idioms.

Even fewer studies have identified bugs that meaningfully involved Rust’s aliasing model. In 2023, a set of anonymous authors classified 22 bugs from five popular Rust encapsulations of foreign libraries [1]. Most of the errors they investigated involved incorrect handling of exceptions and heap objects that were improperly shared across foreign boundaries. They identified two bugs related to Rust’s aliasing restrictions, but they did not describe these bugs in terms of Stacked or Tree Borrows. McCormack et al. [42] extended Miri to interoperate with an LLVM interpreter and used this hybrid tool to conduct a large-scale evaluation of multilanguage libraries. In Rust, packages and libraries are referred to as “crates” and are published on crates.io. They found 46 bugs from 37 crates, and one of these crates was maintained by the Rust Project. This crate had an aliasing violation that occurred after a foreign API had copied a pointer derived from a “unique” mutable reference. This fits a pattern described by one of our interview participants, who was concerned that foreign APIs would “*retain a pointer*” from Rust and “*keep operating on it later*” (P11). Yu et al. [78] found additional aliasing violations with CapsLock, an instrumentation tool that uses custom RISC-V hardware capabilities. Their approach performs nearly twice as fast as Miri and has support for inline assembly, but its aliasing model is weaker than both Stacked and Tree Borrows, and the capabilities that it relies on have not been implemented in physical hardware yet. They evaluated their design on a variety of test cases that called foreign functions and found similar categories of aliasing violations as McCormack et al [42]. In particular, certain test cases were “mutating an owned object without borrowing the owner” [78].

RQ2 - Tooling. Rust is a relatively new programming language, so most research efforts have focused on developing new tools and not evaluating existing ones. Zeng and Crichton [79] hypothesized that a lack of promotion for tooling could prevent Rust from being widely adopted as a

systems language. However, their work was published prior to the creation of most Rust-specific bug-finding tools. Lack of promotion could explain why few of our participants had used formal methods tools. However, several of the tools listed in our survey, such as Verus [34], had recently been introduced, and few had support for unsafe code. Today, a variety of static verification tools for Rust are being used in industry, and we expect that studying their adoption would be a useful direction for future work.

Höltervennhoff et al. [22] are the only other group that we are aware of to examine how developers use tooling to assist with writing unsafe code. Certain development tools were somewhat underused among their interview population; only seven of their 26 participants had mentioned using Miri. Since 61% of our survey respondents had used Miri, this does not seem to indicate a broader trend within the Rust community. Likewise, few of their interview participants used automated auditing tools like cargo-audit, while 51% of our survey participants had used one or more of these tools. However, our respondents only needed to have used these tools once, while Höltervennhoff et al. [22] indicated that participants who audited their dependencies did so “regularly.” Höltervennhoff et al. [22] also indicated that the majority of their participants had used Clippy, and that some had mixed experiences. Although many of our interview participants had also used Clippy, our investigation was primarily concerned with bug-finding tools, and not linting tools, so we did not highlight it in our analysis or survey. We also cover debuggers, static verifiers, and tools that generate bindings to foreign functions, which were not mentioned in their study.

RQ3 - Motivations. Multiple studies have identified different use cases for unsafe code. Qin et al. [49] collected random samples of operations within unsafe contexts from Rust’s standard library and 10 popular third-party libraries. They identified three use cases for these operations; 42% were related to interoperation, 22% were to improve performance, and 14% involved sharing memory between threads. Astrauskas et al. [2] used “anecdotal examples” and snippets of source code to create a taxonomy of six purposes for using unsafe code. These included implementing data structures with complex sharing, overcoming the incompleteness of Rust’s type system, emphasizing contracts and invariants, calling foreign functions, using intrinsics or inline assembly, and improving performance. The most frequently used unsafe feature was calling unsafe functions. Fulton et al. [16] found that developers most frequently used unsafe code to call foreign functions, improve performance, and interact directly with hardware or the operating system.

Unlike these studies, we are concerned with “motivations,” which we define as feature-agnostic reasons for using unsafe code. This excludes several of the feature-specific topics mentioned in prior work, such as intrinsics [2] or sharing memory between threads [2]. We cannot reliably assign motivations to specific features without knowing more about the decision-making process behind each use case. Our perspective is similar to Höltervennhoff et al. [22], who identified necessity, performance, and efficiency as “motivations.” It is unclear whether this distinction was also an explicit consideration for their study. Zhang et al. [80] were able to classify unsafe operations as “necessary” by applying a strict definition: it had to be impossible to implement a safe alternative. In their evaluation of Rust libraries, they identified two reasons why developers chose to use unsafe code when it was not strictly necessary: to improve performance, and because they were unaware of a safer alternative. They did not identify whether developers were aware of a safe alternative but had chosen to use unsafe anyway, since it was easier. Their strict criteria also excludes certain context-specific perspectives on what it means for unsafe code to be necessary. For example, they classified all instances of `UnsafeCell` as unnecessary, since Rust’s safe `RefCell` can be used instead. However, it may be necessary to avoid the run-time cost of `RefCell` in performance-critical applications. Zhang et al. [80] account for this by evaluating the performance impact of several

safe alternatives to unsafe operations, but we expect that developers’ subjective experiences will also be helpful in understanding what motivates them to use unsafe code.

RQ4 - Encapsulation. Unsafe code is prevalent throughout the Rust ecosystem, but developers use it minimally and hide it under safe interfaces. In July of 2016, Ozdemir [47] found that 29% of all published crates used unsafe features, but most unsafe blocks were relatively small and tightly-scoped to include only unsafe operations. Evans et al. [15] observed similar patterns in September of 2018. The number of published crates had increased by more than ten times, but 29% still directly used unsafe code. Although 38% of crates did not directly use unsafe code, they either directly or indirectly depended on third-party crates that did use these features. In January of 2020, Astrauskas et al [2] found that 23.6% of third-party crates used unsafe features. Most instances of unsafe code were relatively straightforward and hidden beneath safe interfaces. The majority of unsafe functions called by libraries were standard (e.g. not closures, function pointers, or trait functions) and defined within the same crate or provided by Rust’s standard library. In particular, 34.7% of all crates had declared unsafe APIs, but they were not visible to users. This aligns with our participants’ experiences; 88% claimed that they would attempt to encapsulate unsafe operations beneath safe APIs, and 61% expected that it would be at least somewhat easy for another developer to understand a random unsafe block or function from their code.

Developers typically mark functions with the `unsafe` keyword to indicate that users need to uphold a particular safety property. The majority (61%) of the survey respondents who had exposed unsafe APIs only did so when they required an invariant beyond what Rust’s type system could provide. However, Astrauskas et al. [2] found that 36.1% of unsafe functions had completely safe implementations. On further investigation, many of these functions had been automatically generated or annotated as `unsafe` for “legacy reasons.” This could provide an alternative explanation beyond necessity or ease-of-use for why the remaining 39% of our respondents were motivated to expose unsafe APIs. While unsafe functions are relatively common, developers rarely use unsafe traits. Evans et al. [15] found that only slightly more than 1% of all crates declared an unsafe trait, and only 6% implemented an unsafe trait. Astrauskas et al. [2] had similar findings; only 2.5% of all traits were unsafe, and 40.4% of these declarations originated from five crates. Likewise, Zhang et al. [80] only identified 11 instances of unsafe trait implementations. Astrauskas et al. [2] suggested that this lack of use could be caused by a lack of guidance on when a trait should be labeled `unsafe`. One of our interview participants indicated that certain developers may have misconceptions on the role of unsafe traits. As mentioned earlier in Section 5.5, one participant who maintained a library that exposed an unsafe trait found that users who had implemented it incorrectly would blame the library, even though “*in theory, the burden of proof is on them*” (P7). Both the Rust Reference [55] and the Rustonomicon [66] have brief sections on unsafe traits, but additional resources could be helpful.

The safety properties for unsafe features are varied and complex. Cui et al. [12] manually reviewed 416 unsafe APIs in Rust’s standard library and identified 19 categories of preconditions and postconditions. Partly for this reason, discrepancies between the signatures of safe APIs and their interior unsafe implementations have been a significant source of bugs and vulnerabilities for the Rust ecosystem. Qin et al. [49] examined 170 bugs and vulnerabilities from ten high-profile Rust libraries. Memory safety errors typically occurred in safe contexts but were caused elsewhere by errors in interior unsafe implementations. Qin et al. [49] also identified 19 examples of incorrect encapsulations of unsafe code from both the third-party libraries in their sample and within Rust’s standard library. Xu et al. [76] exclusively studied registered vulnerabilities and identified similar errors in safe encapsulations. Zheng et al. [82] conducted the broadest empirical study of security issues in the Rust ecosystem to date. They examined 433 unique vulnerabilities that were

discovered between 2014 and 2022, identifying 300 affected repositories. Vulnerable components were significantly more likely to include unsafe blocks or functions. The overwhelming majority of all vulnerabilities affected safe functions.

For this reason, it is somewhat concerning that our participants were often uncertain about the correctness of their encapsulations. Höltervennhoff et al. [22] had similar findings; most of their participants relied on common sense and test coverage to evaluate the security of their code, and few had contributed to projects with formal policies for security or code review. However, many of their participants indicated that they would review unsafe code more carefully to ensure that it upheld all necessary safety invariants. Few participants had ever encountered security issues caused by unsafe code, and none of these issues were critical. Many of our interview participants had encountered memory or thread-safety bugs in unsafe implementations, but few described the security implications of these issues. Security was not a central focus of our study, so it is possible that our participants had reflected on the security implications of their decisions but neglected to mention this during interviews.

7 Discussion

Our goal in studying interoperation was to describe how developers navigate the differences between Rust and other languages (RQ1) and to determine how tooling could be improved to make this process easier (RQ2). Our interview participants found that interoperation was more or less difficult depending on how the characteristics of foreign APIs matched the inherent expectations of Rust’s aliasing model. When documentation was missing or certain design idioms conflicted with Rust’s aliasing rules, participants had no choice but to minimize their interaction with foreign function calls to reduce the perceived risk of triggering undefined behavior. At the time this survey was conducted, Miri was the only tool capable of finding violations of Rust’s aliasing model—it was both the most popular tool and the most lamented. Participants indicated that they had been deterred from using it due to its lack of support for foreign function calls and inline assembly, or its slow performance.

These challenges are not localized at foreign function boundaries. Developers need to reconcile the semantics of foreign APIs with safe, idiomatic Rust encapsulations, which often requires implementing a layer of interior-unsafe operations. Developers’ motivations for using these operations (RQ3) and their ability to reason about encapsulating unsafe code (RQ4) are inherently relevant to this process. Our population prioritized exposing safe encapsulations whenever possible. Necessity was the most predominant motivation for exposing unsafe APIs, and the overwhelming majority (9c) of participants indicated that they would always document any necessary preconditions and postconditions. However, factors that are orthogonal to correctness, like ergonomics and performance, influenced how participants used unsafe code within safe encapsulations. For example, one interview participant indicated that it had been easier to encapsulate a foreign API by transmuting a pointer to a long-lived allocation into a reference with a `'static` lifetime. The alternative would have been to derive a context-specific lifetime, and another participant indicated that this pattern could make an API more difficult to use. These motivations were less common but still significant—51% of participants who exposed unsafe APIs cited performance as a motivation.

We expect that Rust developers will continue to prioritize exposing safe APIs, at least in part due to community incentives. More than 80% (9f) of participants who had used an unsafe APIs before would prefer a safe alternative at least most of the time, if not always. As one interview participant put it: *“everyone wants a safe interface... no one really wants to go use the unsafe heavy ones”* (P9). This may explain why interview participants were willing to rely on ad hoc, application-specific reasoning to justify their design decisions in the face of domain-specific challenges and limitations in tooling. As a result, developers are left uncertain about their design decisions. Only 14% (8a)

of survey respondents who used unsafe code by necessity were always certain that it would be impossible to implement a safe alternative. Likewise, 62% (10a) of respondents who exposed safe APIs for unsafe code were certain “most of the time” that their interfaces were sound, but only 23% were always certain.

Participants suggested several potential solutions to this uncertainty. Developers who contributed to applications with a significant body of specialized unsafe features—mainly multilanguage applications, JIT compilers, and operating systems—would benefit from new dynamic analysis tools for finding Rust-specific aliasing bugs at-scale. Static analysis could also be helpful to generate and validate bindings and encapsulations for unsafe functions. However, Rust is also constantly evolving, and interview participants identified several topics that they were uncertain about due to Rust not having a formal, comprehensive specification. For this reason, we expect that our population would benefit from more comprehensive documentation on certain unstable language features.

7.1 Dynamic Tooling

Our participants also indicated that they needed new tools for finding Rust-specific forms of undefined behavior: “*a version of Miri that could go across that FFI boundary*” (P18), or “*...a unique pointer sanitizer*” (P1). Miri has experimental support for executing foreign functions from shared libraries. However, it is still several orders of magnitude slower than native execution, prohibiting it from being useful in the types of large-scale C and C++ applications where Rust is increasingly being adopted. We see two potential routes for implementing new tools in this category. The first would be to insert run-time checks for aliasing violations during compilation. This would require lowering a subset of Rust’s type information from the MIR level down into the intermediate representations used by these platforms, so that the instrumentation pass could distinguish between references and raw pointers. Yu et al. [78] have had success with this method in CapsLock, but their approach uses an aliasing model that is strictly weaker than Stacked and Tree Borrows. In ongoing work, we are creating BorrowSanitizer [40], which is implemented without hardware acceleration, allowing us to have parity with Miri’s aliasing models. Our approach uses many of the same APIs as other popular LLVM-based tools.

We expect that compile-time instrumentation tools will provide the combination of performance and compatibility that is necessary to support large-scale, multilanguage applications. However, participants who contributed to JIT compilers would also benefit from dynamic binary instrumentation. Tools in this category, such as Valgrind [45], can encode and instrument native assembly on-the-fly, which would be necessary to support JIT-compiled assembly. In 2023, the Krabcake project [31] proposed extending Valgrind to be capable of finding Stacked Borrows violations. Several participants had mentioned Krabcake and indicated that it would be a natural extension to their workflows, since they had already used Valgrind with Rust programs. It was also the second most popular dynamic bug-finding tool behind Miri; it was used by 44% of survey respondents. However, Krabcake is no longer actively maintained, and it is not yet capable of finding bugs in real-world programs. The Rust community should consider reactivating this project.

7.2 Static Tooling

Participants also indicated that static analysis tools could be useful for finding undefined behavior across foreign function boundaries. Several analyses could be implemented as extensions to binding generation tools at varying levels of sophistication. One participant wanted a tool that could tell “*whether your FFI conforms to your expectations...Do all my symbols match? Do all the types signatures match?*” (P5). McCormack et al. [42] were able to detect errors in foreign function bindings by comparing the size of types in the Rust declarations with the size of types in the LLVM definitions.

The authors implemented this as part of a dynamic analysis tool, but their method did not depend on dynamic information, so it could be factored out as a static analysis pass. Changes in Rust’s newest 2024 edition indicate that Rust will continue to evolve toward improving contract visibility at foreign APIs. Rust’s `extern` blocks now require `unsafe` annotations [57], indicating that developers are responsible for ensuring that foreign function bindings match their definitions. Another participant tried to avoid manually reasoning about the lifetime semantics of foreign APIs “*in hopes that in future, there will be a better method for doing this correctly*” (P4). In 2022, contributors to Clang proposed adding lifetime annotations to C++, which would be inferred through a whole-program static analysis pass [6]. The particular RFC has not had activity since August 2023, but the project was associated with Google’s Crubit analysis tool, which is still in development as of May 2025 [20].

Despite their uncertainty, few of our participants had tried using static verification tools. One possible explanation is that when we distributed the survey, only Verus [34] and Kani [72] had significant support for verifying unsafe code. Since then, both of these tools have added extensive support for unsafe features. Verus has been used to verify Vest [7], a parser combinator library, as well as several other security-critical application components [73]. Both Kani and Verifast [24] are being used to verify the entirety of Rust’s standard library as part of a joint initiative between Amazon and the Rust Foundation [32, 64]. We expect that these new tools will meet our participants’ needs for stronger guarantees of soundness. For example, two of our interview participants indicated that refinement typing could be useful—“*custom subtyping, so that you can specify I want an integer, but only between one and ten*” (P4). Flux [35], a refinement-type checker for Rust, was published concurrent to our study and has since been used to verify several real-world Rust applications [36, 54]. As verification continues to become standardized within the Rust community, it would be worthwhile for future studies to examine how these methods are being deployed in practice.

7.3 Unsafe Code Guidelines

The majority of survey respondents found that the Rust community’s guidance and documentation were helpful, but only 6% (10b) indicated that these resources were *always* enough to determine how to use unsafe code correctly. Interview participants also expressed uncertainty about Rust’s semantics. Some were uncertain about the interaction between Rust’s `panic!` macro and `Drop` implementations, with one participant indicating that “*real guidelines for library authors*” would be helpful in this area. This has been a subject of active discussion since at least 2021 [27], but has yet to be fully documented in community resources [23, 27]. We expect that other, similar topics lack formal documentation—instead, developers need to examine years-long issue threads within repositories for Rust’s compiler and Unsafe Code Guidelines [56].

We agree with Höltervennhoff et al. [22] that developers would benefit from having a dedicated resource summarizing points of contention about Rust’s semantics. However, Rust is still a relatively new systems programming language, and certain feature interactions have yet to be fully explored or agreed-upon to the point where writing adequate documentation would even be possible. Developers who implement design patterns that sit at the cutting-edge of Rust’s evolving semantics will inevitably be uncertain if their implementations are sound. Even so, it would be worthwhile for the Rust team to continue to review popular, long-standing issue threads and flag topics that have not been fully documented in the Rustonomicon [66] or the Rust Reference [55]. Meanwhile, developers who engage with unsafe code should periodically review relevant issues within the Unsafe Code Guidelines repository to ensure that they are using these features correctly and to have the opportunity to inform compiler developers about their use-cases. However, we still expect that improvements to Miri will have the greatest impact on our survey population, since it was

used by a majority of survey respondents, and it serves as somewhat of an executable specification for Rust.

8 Conclusion

The Rust programming language provides static safety guarantees. However, Rust is frequently used to interoperate with other languages. To call foreign functions, developers need to use Rust’s unsafe features, which bypass its safety restrictions. These features can introduce bugs and security vulnerabilities if they are used incorrectly. Few Rust-specific development tools provide comprehensive support for unsafe code, and even fewer support multilanguage applications. Developers will need new solutions for testing, bug finding, and verification in this context. We conducted a mixed methods, exploratory study to determine which interventions will be effective for developers who use unsafe code. We interviewed 19 developers who regularly wrote or edited unsafe code and created a survey that reached an additional 160 developers who engaged with unsafe code in some capacity. We examined the challenges associated with interoperation, limitations in tooling, developers’ motivations for using unsafe code, and how they reason about encapsulation.

The majority of participants used unsafe code to call foreign functions. Certain foreign libraries had aliasing and concurrency patterns that conflicted with the expectations of Rust’s type system, making them difficult to encapsulate. However, participants did not have a reliable method to determine if their encapsulations were sound. Miri, a Rust interpreter, was the most popular solution for detecting undefined behavior, but participants who had used it before were deterred from using it again because of its slow performance and lack of support for key features. Most participants were motivated to use unsafe code by necessity, but they also cited performance and ergonomics as motivations. The majority followed best practices, such as encapsulating and documenting unsafe code, but many were somewhat uncertain about their design patterns, and few felt that Rust’s documentation was always adequate for understanding how to use unsafe code correctly.

Our participants would benefit from new static and dynamic analysis tools to assist with interoperation and with finding Rust-specific forms of undefined behavior. Dynamic analysis tools could improve on Miri to provide better performance, support for inline assembly, and the capability to detect aliasing violations triggered by operations that span foreign function boundaries. The Rust community should also continue to invest in improving and expanding documentation on unsafe code, so that longstanding issue threads can become integrated into standard community resources. A useful direction for future work would be to synthesize the methods used by Astrauskas et al. [2] and Evans et al. [15], which provide different, complementary perspectives on the distribution of unsafe code. Future studies should consider the use of unsafe code in unpublished but publicly available crates, as well as the extent to which Rust libraries are exposed to unsafe operations through foreign function calls. The results of such a study would provide an excellent opportunity to triangulate the results described in this paper.

Acknowledgments

We thank the Rust Community for their engagement and support, as well as Jenny Liang and Courtney Miller for their feedback on our study design.

References

- [1] Anonymous. 2023. “Rewrite it in Rust” Considered Harmful?: Security Challenges at the C-Rust FFI. Retrieved 2026-02-17 from <https://goto.ucsd.edu/~rjhala/hotos-ffi.pdf>
- [2] Vytas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust?. In *Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (OOPSLA '20, Vol. 4)*. ACM, New York, NY, USA, Article 136, 27 pages. <https://doi.org/10.1145/3428204>

- [3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In *Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Athens, Greece) (OOPSLA '19, Vol. 3). ACM, New York, NY, USA, Article 147, 30 pages. <https://doi.org/10.1145/3360573>
- [4] Jon Bauman and The Rust Foundation. 2024. C++/Rust Interoperability Problem Statement. Retrieved 2026-02-17 from <https://github.com/rustfoundation/interop-initiative/>
- [5] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. <https://doi.org/10.1191/1478088706qp0630a>
- [6] Martin Brønne, Rosica Dejanovska, Gábor Horváth, Dmitri Gribenko, and Luca Versari. 2022. [RFC] Lifetime annotations for C++. Retrieved 2026-02-17 from <https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377>
- [7] Yi Cai, Zheng Yao, Pratap Singh, and Bryan Parno. 2026. Vest. Retrieved 2026-02-17 from <https://github.com/secure-foundations/vest>
- [8] Rust Community. 2026. Awesome Rust. Retrieved 2026-02-17 from <https://github.com/rust-unofficial/awesome-rust>
- [9] Rust Community. 2026. Options and pointers (“nullable” pointers). <https://doc.rust-lang.org/std/option/#options-and-pointers-nullable-pointers>.
- [10] J. W. Creswell, V. L. Plano Clark, M. L. Gutmann, and W. E. Hanson. 2003. Advanced Mixed Methods Research Designs. In *Handbook of Mixed Methods in Social and Behavioral Research*, A. Tashakkori and C. Teddlie (Eds.). Sage, Thousand Oaks, CA, Chapter 8, 209–240.
- [11] Will Crichton. 2020. The Usability of Ownership. In *Proceedings of the 1st Workshop on Human Aspects of Types and Reasoning Assistants* (Virtual Event) (HATRA '20). ACM, New York, NY, USA, 8 pages. <https://doi.org/10.48550/arXiv.2011.06171>
- [12] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. Is unsafe an Achilles’ Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 106, 13 pages. <https://doi.org/10.1145/3597503.3639136>
- [13] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry For the Deductive Verification Of Rust Programs. In *Proceedings of the 23rd International Conference on Formal Engineering Methods* (Madrid, Spain) (ICFEM '22). Springer-Verlag, Berlin, Heidelberg, 90–105. https://doi.org/10.1007/978-3-031-17244-1_6
- [14] Christopher Durham. 2022. Document current justification for not requiring recursive reference validity (in particular, &mut uninit not being immediate UB). <https://github.com/rust-lang/unsafe-code-guidelines/issues/346>.
- [15] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 246–257. <https://doi.org/10.1145/3377811.3380413>
- [16] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *Proceedings of the Seventeenth USENIX Conference on Usable Privacy and Security* (SOUPS'21). USENIX Association, USA, 20 pages. <https://www.usenix.org/conference/soups2021/presentation/fulton>
- [17] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Copenhagen, Denmark) (PLDI '24, Vol. 8). ACM, New York, NY, USA, Article 192, 25 pages. <https://doi.org/10.1145/3656422>
- [18] Galois. 2025. Crucible. Retrieved 2026-02-17 from <https://github.com/GaloisInc/crucible>
- [19] ATLAS.ti Scientific Software Development GmbH. 2024. ATLAS.ti Web. Retrieved 2026-02-17 from <https://atlasti.com/>
- [20] Google. 2025. Crubit: C++/Rust Bidirectional Interop Tool. Retrieved 2026-02-17 from <https://github.com/google/crubit>
- [21] Michael P. Grady. 1998. *Qualitative and Action Research: A Practitioner Handbook*. Phi Delta Kappa Educational Foundation, Bloomington. 26 pages.
- [22] Sandra Höltervennhoff, Philip Klöstermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. 2023. “I wouldn’t want my unsafe code to run my pacemaker”: An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust. In *32nd USENIX Security Symposium* (USENIX Security 23). USENIX Association, Anaheim, CA, 2509–2525. <https://www.usenix.org/conference/usenixsecurity23/presentation/holtervennhoff>
- [23] Eric Huss. 2025. Document panic-in-panic (“double panic”) behavior. Retrieved 2026-02-17 from <https://github.com/rust-lang/reference/issues/1712>
- [24] Bart Jacobs, Jan Smans, Frank Piessens, and Verifast Contributors. 2025. VeriFast. Retrieved 2026-02-17 from <https://github.com/verifast/verifast>
- [25] Dana Jansens. 2023. Supporting the Use of Rust in the Chromium Project. Retrieved 2026-02-17 from <https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html>
- [26] Ralf Jung. 2024. Rust Has Provenance. Retrieved 2026-02-17 from <https://github.com/rust-lang/rfcs/pull/3559>

- [27] Ralf Jung. 2025. Document when we *do* guarantee that drop runs. Retrieved 2026-02-17 from <https://github.com/rust-lang/nomicon/issues/135>
- [28] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (Cascais, Portugal) (POPL '19, Vol. 4)*. ACM, New York, NY, USA, Article 41, 32 pages. <https://doi.org/10.1145/3371109>
- [29] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.* 10, POPL, Article 48 (Jan. 2026), 29 pages. <https://doi.org/10.1145/3776690>
- [30] Steve Klabnik, Carol Nichols, and the Rust Community. 2026. The Rust Programming Language. Retrieved 2026-02-17 from <https://doc.rust-lang.org/book/>
- [31] Felix S. Klock and Bryan Garza. 2023. Krabcake: A Rust UB Detector. Rust Verification Workshop. Retrieved 2026-02-17 from <https://pnkfx.org/presentations/krabcake-rust-verification-2023-april.pdf>
- [32] Rahul Kumar. 2024. Verify the Safety of the Rust Standard Library. Retrieved 2026-02-17 from <https://aws.amazon.com/blogs/opensource/verify-the-safety-of-the-rust-standard-library/>
- [33] Per Larsen. 2018. C2Rust: Migrating Legacy Code to Rust. Retrieved 2026-02-17 from <https://www.youtube.com/watch?v=WEsR0Vv7jhg>
- [34] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs Using Linear Ghost Types. In *Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (Cascais, Portugal) (OOPSLA '23, Vol. 7)*. ACM, New York, NY, USA, Article 85, 30 pages. <https://doi.org/10.1145/3586037>
- [35] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation (Orlando, Florida) (PLDI '23, Vol. 7)*. ACM, New York, NY, USA, Article 169, 25 pages. <https://doi.org/10.1145/3591283>
- [36] Nico Lehmann, Cole Kurashige, Nikhil Akiti, Niroop Krishnakumar, and Ranjit Jhala. 2025. Generic Refinement Types. *Proc. ACM Program. Lang.* 9, POPL, Article 49 (Jan. 2025), 29 pages. <https://doi.org/10.1145/3704885>
- [37] Wei Liu. 2018. Introducing reCAPTCHA v3: the new way to stop bots. Retrieved 2026-02-17 from <https://developers.google.com/search/blog/2018/10/introducing-recaptcha-v3-new-way-to>
- [38] LLVM Project. 2025. LLDB. Retrieved 2026-02-17 from <https://lldb.llvm.org/>
- [39] LLVM Project. 2026. libFuzzer – a library for coverage-guided fuzz testing. Retrieved 2026-02-17 from <https://llvm.org/docs/LibFuzzer.html>
- [40] Ian McCormack, Oliver Braunsdorf, Johannes Kinder, Jonathan Aldrich, and Joshua Sunshine. 2025. BorrowSanitizer. Rust Verification Workshop. Retrieved 2026-02-17 from <https://borrowsanitizer.com/pdfs/rw2025.pdf>
- [41] Ian McCormack, Tomas Dougan, Sam Estep, Hanan Hibshi, Jonathan Aldrich, and Joshua Sunshine. 2026. A Mixed Methods Study on the Implications of Unsafe Rust for Interoperation, Encapsulation, and Tooling. <https://doi.org/10.5281/zenodo.18664571>
- [42] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. 2025. A Study of Undefined Behavior across Foreign Function Boundaries in Rust Libraries. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (Ottawa, Ontario, Canada) (ICSE '25)*. IEEE Press, Piscataway, NJ, USA, 2075–2086. <https://doi.org/10.1109/ICSE55347.2025.00167>
- [43] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technology Journal* 10, 1 (1998), 100–107.
- [44] Matthew B. Miles, A. Michael Huberman, and Johnny Saldaña. 2020. *Qualitative data analysis: a methods sourcebook* (4 ed.). SAGE, Los Angeles London New Delhi Singapore Washington DC Melbourne.
- [45] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (2007), 89–100. <https://doi.org/10.1145/1273442.1250746>
- [46] Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Santa Clara, CA, 377–389. <https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan>
- [47] Alex Ozdemir. 2016. Unsafe in Rust: Syntactic Patterns. Retrieved 2026-02-17 from <https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax/>
- [48] Michael Quinn Patton. 1990. *Qualitative evaluation and research methods*. SAGE Publications, inc, Thousand Oaks, CA, USA.
- [49] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiyang Zhang. 2020. 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 (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. <https://doi.org/10.1145/3385412.3386036>
- [50] Qualtrics. 2024. Qualtrics XM // The Leading Experience Management Software. Retrieved 2024-10-02 from <https://www.qualtrics.com>

- [51] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. 2023. Robust Speech Recognition via Large-Scale Weak Supervision. In *Proceedings of the 40th International Conference on Machine Learning (PLMR '23, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). Association for Computing Machinery, New York, NY, USA, 28492–28518. <https://proceedings.mlr.press/v202/radford23a.html>
- [52] Alex Rebert and Christoph Kern. 2024. *Secure by Design: Google's Perspective on Memory Safety*. Technical Report. Google Security Engineering. Retrieved 2026-02-17 from <https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/>
- [53] RelevantID. 2026. RelevantID: Enjoy a next-generation approach to ID validation. Retrieved 2024-10-02 from <https://www.imperium.com/relevantid/>
- [54] Vivien Rindisbacher. 2025. Vtock: Verifying the Tock Kernel. Retrieved 2026-02-17 from <https://www.youtube.com/watch?v=reU2rOjJAPY>
- [55] Rust Community. 2023. The Rust Reference. Retrieved 2026-02-17 from <https://doc.rust-lang.org/reference/>
- [56] Rust Community. 2023. Unsafe Code Guidelines Reference. <https://rust-lang.github.io/unsafe-code-guidelines/>.
- [57] Rust Community. 2024. The Rust Edition Guide - Unsafe extern blocks. <https://doc.rust-lang.org/edition-guide/rust-2024/unsafe-extern.html>.
- [58] Rust Community. 2025. cargo-deny. Retrieved 2026-02-17 from <https://embarkstudios.github.io/cargo-deny/>
- [59] Rust Community. 2025. cargo-geiger. Retrieved 2026-02-17 from <https://github.com/geiger-rs/cargo-geiger>
- [60] Rust Community. 2025. cargo-update. Retrieved 2026-02-17 from <https://crates.io/crates/cargo-update>
- [61] Rust Community. 2025. Cargo Vet. Retrieved 2026-02-17 from <https://mozilla.github.io/cargo-vet/index.html>
- [62] Rust Community. 2025. cargo fuzz. Retrieved 2026-02-17 from <https://github.com/rust-fuzz/cargo-fuzz>
- [63] Rust Community. 2025. RustSec: cargo audit. Retrieved 2026-02-17 from <https://github.com/RustSec/rustsec/tree/main/cargo-audit>
- [64] Rust Community. 2025. Verify Rust Standard Library Effort. Retrieved 2026-02-17 from <https://model-checking.github.io/verify-rust-std/intro.html>
- [65] Rust Community. 2026. The bindgen User Guide. Retrieved 2026-02-17 from <https://rust-lang.github.io/rust-bindgen/>
- [66] Rust Community. 2026. The Rustonomicon. Retrieved 2026-02-17 from <https://doc.rust-lang.org/nomicon/>
- [67] Rust for Linux. 2026. Rust for Linux - Rust kernel policy. Retrieved 2026-02-17 from <https://rust-for-linux.com/rust-kernel-policy>
- [68] Niko Matsakis Rémy Rakic. 2023. Polonius update. Retrieved 2026-02-17 from <https://blog.rust-lang.org/inside-rust/2023/10/06/polonius-update.html>
- [69] Benjamin Saunders, Julius Sim, Tom Kingstone, Shula Baker, Jackie Waterfield, Bernadette Bartlam, Heather Burroughs, and Clare Jinks. 2018. Saturation in qualitative research: exploring its conceptualization and operationalization. *Quality & Quantity* 52, 4 (2018), 1893–1907. <https://doi.org/10.1007/s11135-017-0574-8>
- [70] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) (*USENIX ATC '12*). USENIX Association, USA, 28. <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>
- [71] Katta Spiel, Oliver L. Haimson, and Danielle Lottridge. 2019. How to Do Better with Gender on Surveys: A Guide for HCI Researchers. *Interactions* 26, 4 (2019), 62–65. <https://doi.org/10.1145/3338283>
- [72] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying dynamic trait objects in rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) (*ICSE-SEIP '22*). Association for Computing Machinery, New York, NY, USA, 321–330. <https://doi.org/10.1145/3510457.3513031>
- [73] Verus Team. 2025. Verus — Projects. Retrieved 2026-02-17 from <https://verus-lang.github.io/verus/publications-and-projects/>
- [74] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. <https://doi.org/10.1145/3735592>
- [75] Dan Wallach. 2024. Translating All C to Rust (TRACTOR). Retrieved 2026-02-17 from <https://www.darpa.mil/program/translating-all-c-to-rust>
- [76] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Transactions on Software Engineering and Methodology* 31, 1, Article 3 (2021), 25 pages. <https://doi.org/10.1145/3466642>
- [77] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. In *Proceedings of the 26th ACM SIGPLAN International Conference on Functional Programming (Virtual)* (*ICFP '21*). ACM, New York, NY, USA, Article 92, 30 pages. <https://doi.org/10.1145/3473597>

- [78] Jason Z. Yu, Fangqi Han, Kaustab Choudhury, Trevor E. Carlson, and Prateek Saxena. 2025. Securing Mixed Rust with Hardware Capabilities. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) (CCS '25). Association for Computing Machinery, New York, NY, USA, 1544–1558. <https://doi.org/10.1145/3719027.3744861>
- [79] Anna Zeng and Will Crichton. 2018. Identifying Barriers to Adoption for Rust through Online Discourse. In *Proceedings of the 9th Workshop on Evaluation and Usability of Programming Languages and Tools* (Boston, MA, USA) (PLATEAU '18). Schloss Dagstuhl, Saarbrücken, Germany, 6 pages. arXiv:1901.01001 [cs.HC]
- [80] Yuchen Zhang, Ashish Kundu, Georgios Portokalidis, and Jun Xu. 2023. On the Dual Nature of Necessity in Use of Rust Unsafe Code. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) (ESEC/FSE 2023). Association for Computing Machinery, New York, NY, USA, 2032–2037. <https://doi.org/10.1145/3611643.3613878>
- [81] Ziyi Zhang, Shuofei Zhu, Jaron Mink, Aiping Xiong, Linhai Song, and Gang Wang. 2022. Beyond Bot Detection: Combating Fraudulent Online Survey Takers. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) (WWW '22). Association for Computing Machinery, New York, NY, USA, 699–709. <https://doi.org/10.1145/3485447.3512230>
- [82] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. 2023. A Closer Look at the Security Risks in the Rust Ecosystem. *ACM Transactions on Software Engineering and Methodology* 33, 2, Article 34 (2023), 30 pages. <https://doi.org/10.1145/3624738>
- [83] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 2022. Learning and Programming Challenges of Rust: A Mixed-Methods Study. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) (ICSE '22). Association for Computing Machinery, New York, NY, USA, 1269–1281. <https://doi.org/10.1145/3510003.3510164>