On the Dual Nature of Necessity in Use of Rust Unsafe Code

Yuchen Zhang
yzhan219@stevens.edu
Stevens Institute of Technology
USA

Ashish Kundu
ashkundu@cisco.com
Cisco Research
USA

Georgios Portokalidis
gportoka@stevens.edu
Stevens Institute of Technology
USA

Jun Xu
junxzm@cs.utah.edu
University of Utah
USA

ABSTRACT

Rust offers both safety guarantees and high performance. Thus, it has gained significant popularity in the industry. To extend its capability as a system programming language, Rust allows `unsafe` blocks where the execution has low-level controls but loses the safety guarantees. In principle, `unsafe` blocks should only be used when necessary. However, preliminary evidence shows a different situation. This paper aims to establish a deeper view of this matter and bring endeavors toward improvement.

We first present a study on the use of `unsafe` Rust in practice. We manually inspected 5,946 `unsafe` blocks from 140 popular libraries and applications, focusing on whether the use of `unsafe` code is **necessary** (precisely, whether they have safe alternatives). The study unveils hundreds of instances of unnecessary `unsafe` Rust code and provides a taxonomy together with detailed analyses. These results complement our understanding and offer insights for the community to make a change.

Following the study, we further summarize nine popular patterns of unnecessary `unsafe` blocks and design an IDE plugin to auto-suggest their safe alternatives. Applied to 140 buggy `unsafe` blocks from the RustSec Advisory Database, the plugin identifies and offers safe versions to remove the bug for 28.6% of all cases.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Rust Security, Unsafe Code, Software Engineering

ACM Reference Format:
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 (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3611643.3613878

1 INTRODUCTION

Rust is a young system programming language. It offers efficiency and safety concurrently [21], motivating many software vendors, including Microsoft [20], Meta [11], Google [17], and Amazon [12], to adopt it for developing their products. Rust achieves safety through a series of restrictive language designs (e.g., no raw pointers and strict lifetime checks), which has been shown to limit its capability for system software development in certain ways. For instance, its no-pointer policy prevents the use of GLibc functions. To this end, Rust allows `unsafe` code (i.e., code blocks enclosed by the `unsafe` keyword) where the safety restrictions are dropped and low-level controls are granted.

As language restrictions no longer work for `unsafe` Rust blocks, it becomes the developers' full responsibility to ensure their safety. Not surprisingly, this is not sustainable. The use of `unsafe` Rust in practice has led to various safety and security issues [9, 18]. For instance, the RustSec Advisory Database [14] includes hundreds of bugs due to misuse of `unsafe` Rust. To address the problem, the community has extended many efforts to understand the use of `unsafe` Rust. Astrauskas et. al. [1] and Evans et. al. [6] demystify **how** and **why** developers use `unsafe` Rust. Further, Boqin et. al. [13] unveil **what** bugs `unsafe` Rust can lead to. These results offer nice insights for bug-detection methods like dynamic testing [2, 15] and static analysis [3, 8], but they seem to present less direct benefits to the developers.

In this paper, we present endeavors to address `unsafe` Rust code in the development cycle. We focus on a question less covered by previous research: ***is the use of `unsafe` blocks necessary***? To answer this question, we first manually collected and examined 5,946 `unsafe` blocks from 140 popular libraries and applications, aiming to understand whether they can be re-implemented with only safe code. We find that `unsafe` blocks are indeed inevitable in many scenarios, such as ① interacting with foreign languages, such as calling GLibc functions, and ② bypassing language restrictions to implement specialized functionality, such as manipulating a linked list during iteration.

However, the remaining cases, which can be categorized into multiple groups (see section 4), represent unnecessary use of `unsafe` Rust. As we will show later in this section, these cases can often lead to bugs or even vulnerabilities. Based on our analysis, there are two presumable reasons why developers use unnecessary `unsafe` blocks. First, they intentionally bypass Rust restrictions for the goal of optimizations. For instance, Rust offers a safe function, `copy_within`, to copy data from one vector to another with built-in boundary checks. To avoid the boundary checks, developers may instead use `ptr::copy`, which can only be invoked in `unsafe` blocks. Second, the developers are not aware of the safe options. For example, they presume that mutable static variables—or global variables in general—must be accessed in `unsafe` blocks, without realizing that mutable static variables can be encapsulated within the `lazy_static!` macro and accessed in safe code.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613878

While our study brings helpful results, they still need to be comprehended and consumed before the developers can apply them to reduce unnecessary `unsafe` blocks. To further benefit developers, we generalize the safe alternatives for 9 popular groups of `unsafe` Rust summarized in Table 2 and integrate them as auto-suggestion rules into a VS Code plugin. For alternatives that may bring an execution slowdown, we also attach information about the performance overhead we empirically obtained. To understand the utility of our plugin, we perform an experiment on 140 `unsafe` blocks with bugs from the RustSec Advisory Database [14]. The result shows that our plugin can recognize and make correct suggestions for 28.6% of the buggy cases.

In summary, we make the following contributions:

- We perform an empirical study to understand the necessity of `unsafe` Rust code in practice, bringing insights toward reducing unnecessary use and related bugs.
- We design and develop a VS Code plugin to offer automated suggestions of safe alternatives for unnecessary `unsafe` blocks.
- Our study data and plugin are released at https://github.com/yzhang71/Rust-UnsafeToSafe.

2 BACKGROUND

2.1 Safe Rust

Rust, created in 2013, is a young system programming language. Compared to C/C++, Rust offers both memory safety and thread safety, incentivizing increasing adoptions by software vendors [4, 11, 12, 17]. The safety of rust is achieved through a combination of language and runtime features.

Raw Pointer Dereference: Rust still supports pointers but considers dereferences of raw pointers undefined behaviors. Its compiler will raise an error when encountering raw pointer dereferences. This prevents all safety issues caused by invalid pointers.

Boundary Checks: Out-of-bound (OOB) accesses are also undefined behaviors in Rust, which are prevented by compile-time and runtime checks. On stack accesses with known OOB indexes, Rust compiler will abort with an error. If the indexes are unknown, runtime checks will be inserted instead. Dynamically allocated data objects (*e.g.,* vectors) in Rust are represented as *fat pointers* that consist of a *pointer to the data*, the *length* (used space), and the *capacity* (allocated space). Run-time checks will be inserted on accesses to such objects to inspect if the index falls within the length.

Lifetime: *Liftime*, a new feature enforced in Rust, mandates that no references to data are valid if the data has not been created or has expired. The Rust compiler performs lifetime analysis and aborts when detecting any violations. Lifetime prevents temporal issues such as use-after-free and use-after-return.

Ownership and Borrowing: Rust further introduces the concept of *ownership*, requiring that each value has one and only one owner at any time. Once the owner goes out of scope, the value will be dropped. To enable necessary functionality such as passing objects by reference to another function, Rust supports *borrowing* of ownership that allows multiple references to the same value during the owner's lifetime. The borrowing carries a rule that *mutable references cannot co-exist with other mutable or immutable references.* Ownership and borrowing, together with lifetime, help prevent temporal issues and concurrency issues.

2.2 `Unsafe` Rust

Rust brings safety. However, its restrictions make certain low-level programming difficult. For instance, direct memory access is infeasible as raw pointer dereferences are not allowed. To this end, Rust introduces an extension called *`unsafe` Rust*. `Unsafe` Rust, one or more blocks annotated by the `unsafe` keyword, bypasses the restrictions we discussed above and interrupts the safety guarantees.

Raw Pointer Dereference: `Unsafe` code is allowed to dereference raw pointers as C does. All the aforementioned restrictions, including boundary checks, lifetime, ownership, and borrowing, do not apply to pointers dereferences. This becomes a primary reason for safety issues in Rust applications.

Others Flexibilities: Besides dereferencing raw pointers, `unsafe` Rust can perform many other operations prohibited in safe Rust.

- **Call `unsafe` functions:** In Rust, we can mark an entire function `unsafe` . Only `unsafe` code can call `unsafe` functions.
- **Access mutable static variables:** Global variables in Rust are declared as mutable and static objects outside of functions. Accesses to global variables are not permitted as that can lead to concurrent modifications by different threads and thus, data races. This restriction does not apply in the `unsafe` Rust code.
- **Implement `unsafe` traits:** A *trait* defines a collection of methods that a given type must implement. `Unsafe` traits indicate at least one of the methods is `unsafe` . `Unsafe` traits must be placed inside `unsafe` blocks.
- **Access union fields:** Unions are introduced in Rust RFC 1444 [7] to maintain compatibility with external C code. Direct access to union fields is only allowed in `unsafe` code.

3 STUDY SETUP

3.1 Dataset

To understand the necessity of `unsafe` Rust code in practice, we collect various real-world and publicly available Rust projects — usually termed as *crates* — from the wild. Previous studies [1, 6] focus on generic program analysis (e.g., counting the number of raw pointer dereferences), which can be automated with compiler-based tools. This enables them to include a large corpus of crates. In contrast, we aim to gain an in-depth functionality-level understanding of the `unsafe` code. In the example of raw pointer dereference, we need to unveil that the pointer is used for goals like mutating an element in a double-linked list. As such, our study mandates human intelligence and manual reasoning, which is only feasible to run on a selected set of crates.

Specifically, we collect the set of crates with 10k+ downloads on `crates.io` (the Rust package registry) [16] and the set of crates with 1k+ stars from `awesome-rust` (a repository of well-accepted Rust projects) [19]. We keep the crates that are included in both sets. This way, we ensure the representativeness of the dataset. For instance, our dataset includes many renowned applications developed by professional Rust development teams (e.g., `Servo`[10], a web browser engine developed by Mozilla Research). In total, we collect 140 Rust crates across 56 categories, consisting of 82 libraries (58.6%) and 58 applications (41.4%). In total, the crates include 5,946

Table 1: Distribution of `unsafe` Rust code.

Crates	Categories				
	`Raw Pointer`	`Unsafe Func`	`Mut Static Var`	`Unsafe Trait`	`Unions`
Applications	320	2470	57	10	12
Libraries	213	2834	15	1	14
Total	533	5304	72	11	26
Ratio	8.96%	89.20%	1.21%	0.19%	0.44%

instances of `unsafe` code. Our dataset is larger than those used by other manual studies of Rust (e.g., [13] only involves 10 crates).

3.2 Analysis

We run a three-step manual analysis of all instances of `unsafe` code in our dataset. ❶ First, we run a basic analysis to identify and categorize `unsafe` code based on their high-level goals described in subsection 2.2. ❷ The second step further refines the categories based on the functionality of each `unsafe` block. Consider dereferencing a raw pointer as an example. We first identify the declaration of the pointer and subsequently trace the control flow graph until the last use of the pointer. Combining the execution context, we then determine the functionality of using the raw pointer (e.g., bypassing borrow rules to implement a double-linked list) and group it into more fine-grained subcategories accordingly. ❸ In the last step, we evaluate whether the `unsafe` code in each subcategory is necessary to achieve the intended functionality. If the `unsafe` code can be replaced by a safe equivalent that preserves the same functionality, we conclude that the `unsafe` code is unnecessary and vice versa. For each unnecessary case we identify, we replace the `unsafe` code with the safe alternative and verify that the new code is compilable, remains runnable, and produces identical outcomes.

4 RESULTS AND DISCUSSION

We classify `unsafe` code into five categories shown in Table 1. About 9% of the `unsafe` instances involve dereferences of raw pointers, with the vast majority (approximately 90%) consisting of calling `unsafe` functions or methods. The remaining is distributed among manipulating mutable static variables (1.21%), implementing `unsafe` traits (0.19%), and accessing fields of unions (0.44%).

4.1 Dereferencing A Raw Pointer

We observe 533 instances of raw pointer dereference, which can be classified into three groups as per their functionality.

Interacting with Foreign Code [356 Instances]: Interoperability between programming languages is essential in cross-language programs. Rust's Foreign Function Interface (FFI) mechanism facilitates this by using raw pointers to cross the language boundary. Our study unveils three different ways in practice. ① Rust allows specifying C-style data layout by using the `#[repr(C)]` attribute. Data objects following `#[repr(C)]` can be passed to foreign C code and can only be accessed with raw pointers in Rust. ② Some Rust programs include functions that can be called by foreign C code. Arguments of such functions, if having complex types, must be represented as raw pointers and accessed via deference. ③ Rust programs often invoke functions developed in foreign languages, in particular GLibc. These functions can return raw pointers, which can only be accessed via pointer dereference.

► Unsafe Code is Necessary: To avoid incompatible data layouts between Rust and foreign languages, using raw memory is a desired way to support their interactions. Thus, raw pointers must be used and `unsafe` code to dereference the pointers is necessary.

```rust
pub fn make_mut_slice<T: Clone>(
 arc: &mut Arc<[T]>) -> &mut [T] {
 let mut_ref = unsafe {&mut *(arc as *mut Arc<[T]>)};
 match Arc::get_mut(mut_ref) {
  Some(x) => x,
  None => { *arc = arc.iter().cloned().collect();
  Arc::get_mut(arc).unwrap();} }
}
```

(a) Using `unsafe` block

```rust
pub fn make_mut_slice<T: Clone>(
 arc: &mut Arc<[T]>) -> &mut [T] {

 if  Arc::get_mut(arc).is_none() {
  *arc = arc.iter().cloned().collect();
 } // iter() immutablely borrows "arc"
 Arc::get_mut(arc).unwrap()
}
```

(b) No `unsafe` block

Figure 1: Example of using unnecessary raw pointers.

Bypassing Borrow Rules [13 Instances]: Raw pointers have been used to bypass the ownership restrictions of Rust. The necessity of `unsafe` code in this group depends on the intended functionality.

► Unsafe Code is Necessary: In 11 out of the 13 instances, raw pointers are used to implement data structures that require multiple mutable references. ① 6 cases implement mutable in-order traversal on trees where two mutable references to the same node are needed. ② The other 5 cases implement double-linked lists. They also require two mutable references, one for the previous node and one for the next. In these cases, the use of raw pointers is mandated by the data structures' functionality, which we deem necessary.

► Unsafe Code is Unnecessary: In two other instances, the use of raw pointers is unnecessary. Our "hypothesis" is that the developers might be unaware of how to satisfy Rust's ownership rules without using raw pointers. Figure 1 shows such an example. The developers create a raw pointer to derive a mutable reference to argument `arc`, which is later used for a condition check (see Figure 1a). The use of raw pointer is unneeded, as demonstrated in Figure 1b.

Interior Mutability [164 Instances]: Rust offers another way to opt out of the immutability guarantees via `UnsafeCell`. It enables *interior mutability*, which allows creating multiple mutable references inside `unsafe` blocks. Internally, `UnsafeCell` is an abstraction of a raw pointer to the data object. Thus, it inherits the safety issues of raw pointers.

► Unsafe Code is Unnecessary: In our study, we observe 164 use cases of `UnsafeCell`, which are not necessary. In those cases, `UnsafeCell` can be replaced by a well-encapsulated wrapper known as `RefCell`. `RefCell` provides a safe abstraction that stores the location of active borrows and dynamically checks the borrow rules, ensuring the immutability guarantees. Using `RefCell`, mutable references can also be created using the `borrow_mut()` interface, where `unsafe` blocks are not needed. Of course, `RefCell` carries extra runtime checks and thus, may lead to reduced performance.

4.2 Calling Foreign Functions

Calling `unsafe` function dominating the use of `unsafe` blocks covered by our study. In total, we observe 5,304 cases in this category, distributed across three major families.

Calling Foreign Functions [3,135 Instances]: The Rust ecosystem remains young. To reduce time-to-market, developers often reuse functions from projects developed in foreign languages (e.g., C). Those functions are termed **foreign functions**, which do not offer safety guarantees and can only be called inside `unsafe` blocks.

```rust
unsafe fn from_utf8_unchecked(v: &[u8]) {
  unsafe { mem::transmute(v) }
}
fn as_string<'a>(&'a self) -> &'a str {
  unsafe {
    let byte_array = transmute(slice);
    str::from_utf8_unchecked(byte_array)
  }
}
```

(a) Using unsafe function

```rust
pub const fn from_utf8(v: &[u8]) -> Result<&str> {
    match run_utf8_validation(v) {
        Ok(_) => {Ok(unsafe {from_utf8_unchecked(v)})}
        Err(err) => Err(err),
}
fn as_string<'a>(&'a self) -> &'a str {
    ...
    str::from_utf8(byte_array).unwrap()
}
```

(b) Not using unsafe function

Figure 2: Example for unnecessary use of unsafe function.

In our study, 446 cases are from GLibc, and most of the remaining are imported from other C/C++ projects. Interestingly, we also observe a few cases involving assembly instructions.

► **Unsafe Code is Necessary:** Using unsafe blocks to call foreign functions is mandated by Rust. Thus, we consider this necessary. If Rust verion for those functions (e.g., Rust-based Libc) appears in the future, those cases will become unnecessary.

Initializing Foreign Data Type [109 Instances]: The second family is also related to foreign code. Rust’s safety system requires all data objects to be initiated before use. However, certain data objects can get initialized by foreign code, which is invisible to Rust. This creates a dilemma: if we skip initialization in Rust, the compiler will complain; if we redo initialization in Rust, we may invalidate the execution of the foreign code. To address this issue, Rust offers special unsafe functions, such as `mem::zeroed()` and `assume_init()`, to mute the compiler on initialization issues related to target data objects. We see 109 such cases in our study.

► **Unsafe Code is Necessary:** We believe the use of unsafe functions in those cases is necessary. Otherwise, the initialization dilemma will persist. One may argue that developers should always initialize data objects before sending them to foreign code. This does not solve all the problems because many data objects are created in foreign code and returned to Rust.

Calling Internal Unsafe Function [2,060 Instances]: The last category involves internal functions that are explicitly marked as unsafe due to their lack of safety assurance. Unsafe functions can only be called inside unsafe blocks, spanning two major types according to our observation. First, 401 unsafe functions take raw pointers as some of their arguments. Their code needs to dereference the pointers, thus becoming unsafe. Second, the remaining 1,659 unsafe functions take non-pointer arguments, but they include operations without safety guarantees, such as copying memory without boundary checks.

► **Unsafe Code is Necessary:** In most of the cases (1,853 cases), the use of unsafe functions appears to be necessary. At least, we do not identify safe alternatives. For instance, functions with pointer arguments will certainly incur unsafe operations unless they do not use the arguments at all.

► **Unsafe Code is Unnecessary:** In the other 207 cases, the use of unsafe functions is unnecessary. Those functions all have alternatives where safety is ensured with extra checks. Figure 2a demonstrates such an example. With further analysis, we group the cases into 9 patterns, summarized in Table 2. Based on comments in those cases, one major reason why the developers use the unsafe version is performance consideration. Specifically, they use the unsafe version to avoid the extra checks in the safe alternative.

Table 2: Safe suggestions for common unsafe Rust code

	Code Example	Description

Creating a Vector	`buffer = Vec::with_capacity(len);`
	`unsafe {`
	`    buffer.set_len(len);`
	`}`
	`---> replace with --->`
	`let mut buffer = vec![0; len];`
		A popular way to create a new vector in Rust is using `with_capacity()` in conjunction with the unsafe `set_len()`. Vectors created this way are not initialized, often leading to the use of uninitialized memory. The same operation can be done with safe code “`vec![0; len]`”, which allocates and initializes the vector with zeros.

Resizing a Vector	`buffer.reserve(length);`
	`unsafe {`
	`   buffer.set_len(length);`
	`}`
	`---> replace with --->`
	`buffer.resize(length, 0);`
		To resize an existing vector, we can use `reserve()` together with the unsafe `set_len()`. Doing so will leave the newly added space, if the resizing enlarges the vector, uninitialized. The safe alternative is `resize(len, 0)`, which both resizes and initializes with zeros.

Copy within Vector	`unsafe {`
	`   ptr::copy(v[src], v[dst], cnt);`
	`}`
	`---> replace with --->`
	`v.copy_within(v[src]..cnt, v[dst]);`
		Copying a part of a vector to another location in the same vector can be done with unsafe function `ptr::copy()` without boundary checks. The same functionality can be completed with safe function `copy_within` with built-in boundary checks.

Copy Non-overlapping	`let src = vec![1,2,3];`
	`let mut dst = vec![0; 3];`
	`let cnt = 2;`
	`let beg = src[..].as_ptr();`
	`let end = dst[..].as_mut_ptr();`
	`unsafe {`
	`  ptr::copy_nonoverlapping(b, e, c);`
	`}`
	`---> replace with --->`
	`dst[..c].copy_from_slice(&src[..c]);`
		A variant of the above case is copying a part of a vector to another non-overlapping location. This is often done with unsafe function `ptr::copy_nonoverlapping()` where out-of-bound access can happen. The unsafe function have a safe alternative `copy_from_slice()`.

Create CString	`unsafe {`
	`  let s =`
	`    CString::from_vec_unchecked(vec);`
	`}`
	`---> replace with --->`
	`let s = CString::new(vec).unwrap();`
		To create a C-compatible string from a byte vector, developers often use unsafe function `from_vec_unchecked()` without checks for interior 0 bytes. The function has a safe alternative `CString::new()` that asserts on interior 0 bytes.

CString Size	`unsafe {`
	`  let l = libc::strlen(cstr.as_ptr());`
	`}`
	`---> replace with --->`
	`let l = cstr.as_bytes().len();`
		To get the length of a C-compatible string, Rust developers tend to use function `strlen()` from the “`libc`” crate, which is unsafe . This function has a safe version `cstring.as_bytes().len()`.

Getting Indexed Reference	`let mut vec = vec![0; len];`
	`unsafe {`
	` let elem = vec.get_unchecked(indx);`
	`}`
	`---> replace with --->`
	`let elem = vec.get(index).unwrap();`
	`# method to get mutable reference`
	`unsafe {`
	` let elem = vec.get_unchecked_mut(indx);`
	`}`
	`---> replace with --->`
	`let elem = vec.get_mut(index).unwrap();`
		To get a reference to an element inside a vector, developers often use the unsafe function `get_unchecked()` or `get_unchecked_mut()` to bypass the boundary checks. Such functions have safe alternatives `get().unwrap()` and `get_mut().unwrap()` to prevent potential out-of-bounds indexing.

Convert Bytes to String	`let mut bts:&[u8] = &[159,159,146,146];`
	`unsafe {`
	`    let str = from_utf8_unchecked(bts);`
	`}`
	`---> replace with --->`
	`let str = from_utf8(invalid).unwrap();;`
	`# method to get mutable string slice`
	`unsafe {`
	`  let s = from_utf8_unchecked_mut(bts);`
	`}`
	`---> replace with --->`
	`let s = from_utf8_mut(bts).unwrap();;`
		Converting a slice of bytes to a string slice can be done with unsafe function `from_utf8_unchecked()` or `from_utf8_unchecked_mut()` without checking whether the bytes are valid UTF-8. This can be replaced with safe versions `from_utf8().unwrap()` and `from_utf8_mut().unwrap()` to perform a validation check before returning results.

Cvt. u32 to char	`let u_32 = 0x2764;`
	`unsafe {`
	` let c = char::from_u32_unchecked(u_32);`
	`}`
	`---> replace with --->`
	`let c = char::from_u32(u_32).unwrap();`
		To obtain a `char` typed value from `u32`, many developer uses the unsafe function `from_u32_unchecked()` which ignores validity of the argument. The same functionality can be accomplished using a safe version `from_u32().unwrap()` that mitigates the potential undefined behavior.

4.3 Interacting With Mutable Static Variables

We identify 72 use cases of mutable static variables placed in unsafe blocks. In those cases, the variables share the same lifetime as the program and they can be accessed by any thread without synchronizations. Thus, concurrency issues like data races can arise.

Table 3: Results of unsafe pattern detection and suggestion for RustSec Advisory Database.

Datasets	Detected Unsafe Code Patterns									
	`Create Vector`	`Resize Vector`	`Copy Within Vector`	`Copy Non-Overlapping`	`Create CString`	`Get CString Len`	`Get Reference`	`Bytes to String`	`u32 to char`	`Total`
ADVISORY-DB	25	5	1	1	1	1	2	3	1	40
Ratio	62.5%	12.5%	2.5%	2.5%	2.5%	2.5%	5.0%	7.5%	2.5%	-

► **Unsafe Code is Unnecessary:** Using `unsafe` blocks for mutable static variables is not always necessary. For instance, by putting static mutable variables inside the '`lazy_static!`' macro, we can access them without `unsafe` blocks. Doing so offers more than merely removing the `unsafe` keyword: variables in'`lazy_static!`' are protected by thread-safe wrappers to avoid concurrency issues.

4.4 Defining `Unsafe` Traits

In Rust, a trait defines methods of a particular type and can share with other types, resembling `interface` in Java. Developers can use the `unsafe` keyword to implement an unsafe trait where one or more of its methods have variants with unverifiable safety. In our study, we find 11 `unsafe` instances belonging to this category. In those cases, the `unsafe` keyword is essentially a maker annotating that the implementation might be unsafe. Including or excluding it has no impact on the compiler and the actual safety. Thus, it becomes pointless to discuss its necessity.

4.5 Accessing Unions

To enable better compatibility with C, Rust introduces the `union` type in 2016. Unions are stored as a single contiguous block of memory, which are shared by overlapped fields. Each union access simply interprets the storage at the target filed's type where safety is not guaranteed. Thus, access to unions must be placed in `unsafe` blocks. In our study, we observe 26 `unsafe` blocks used to perform union access.

► **Unsafe Code is Necessary:** Using `unsafe` blocks is the only way to access unions. Thus, it is necessary.

5 APPLICATION OF STUDY RESULTS

Our study brings helpful results toward reducing unnecessary `unsafe` blocks. However, they still need to be comprehended and consumed by developers. To make our results directly usable in the development cycle, we generalize the safe alternatives for popular categories of `unsafe` Rust and integrate them as auto-suggestion rules into a VS Code plugin.

5.1 Safe Alternatives

Through further examination of each unnecessary `unsafe` instance, we identified 9 common patterns with generic safe alternatives. Details of the patterns are summarized in Table 2. These patterns cover 207 instances of `unsafe` code involved in the 140 projects we collected for our study.

5.2 Automated Suggestion

We have integrated the 9 patterns as auto-suggestion rules into the `rust-analyzer` extension of VS Code. In total, around 3.5K lines of Rust code are added. The plugin runs automatically to identify `unsafe` blocks when the developers edit their code. When a single `unsafe` block involves multiple independent expressions, we first group the expressions based on their dependency. This is done via backward slicing from the last expression in the block. All expressions included in the same slice are grouped as a unit. If more expressions remain, we repeat the slicing operation. We then conduct static analysis on the Abstract Syntax Tree (AST) to match each unit with our patterns developed in subsection 5.1. Once a pattern is identified, we concretize the safe version by replacing the abstractions with the actual objects and present the result.

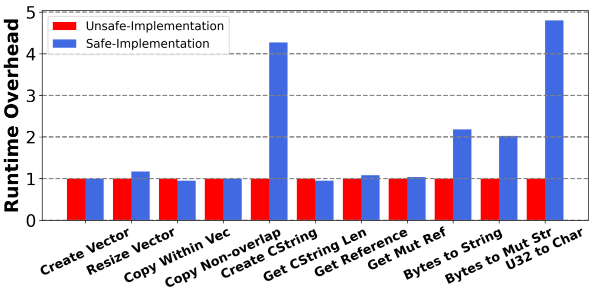

Figure 3: Performance comparison between `unsafe` and safe versions of the same functionality. We consider the `unsafe` version as the baseline. "Get Reference" and "Get Mut Ref" are two sub-categories of "Indexed Reference", depending on the requirement of mutability. "Bytes to Mut Str" and "Bytes to Str" are similar.

5.3 Evaluation

To evaluate the utility of our plugin, we conduct an evaluation on `unsafe` code involving bugs from the RustSec Advisory Database [14]. At the time of writing, the database contains 208 projects and 301 bugs. We utilize the Visual Studio Code IDE with our plugin installed and import all 208 projects from RustSec Advisory Database [14] that contain memory-related vulnerabilities (69.1% of the total 301 vulnerabilities). Among all 208 cases, 68 of them are not involved in `unsafe` code. We then manually examine each instance of the remaining 140 cases to determine if any of the `unsafe` function patterns outlined in subsection 5.1 are detected and if suggestions for safe alternatives are provided.

As shown in Table 3, we are able to detect and successfully provide safe suggestions for 40 cases. That is, our safe code suggestion can mitigate 28.6% of the bugs caused by `unsafe` code. Of all the identified `unsafe` code patterns, "Creating a Vector" is most prevalent, constituting 62.5% of the total detected cases. It is followed by "Resizing a Vector", "Convert Bytes to String", and "Getting Reference", accounting for 12.5%, 7.5%, and 5.0% of all cases, respectively. The remaining 68 bugs are semantic or logical issues that our plugin cannot handle, such as incorrect ordering of parameters for `unsafe` function `Vec::from_raw_parts` [5]. This experiment shows that our IDE plugin can help detect unnecessary `unsafe` Rust code, provide corresponding safe alternatives, and automatically mitigate the safety issues behind them.

Trade-off: The safe versions we suggest may introduce extra operations. For instance, the safe alternatives for "Resizing a Vector" and

"Convert Bytes to String" incur additional runtime initialization or checks, potentially reducing the performance. We further run experiments to understand the impact of the safe alternatives on runtime performance. For each instance matching one of our patterns, we create a standalone program wrapping the piece of target code and stress-test the program with random inputs. We repeat this after replacing the unsafe version with our safe alternatives. All our experiments are conducted on the same machine with Intel i7-8700@3.20GHz, 6 cores, 64GB RAM, and Ubuntu 18.04 LTS.

The evaluation results are shown in Figure 3. For 5 categories (or 7 sub-categories), the safe alternatives have negligible impact on the runtime performance. In the cases of 'Copy Within Vector" and "Getting CString Length", we even observe slight performance improvement. On "Creating a CString", "Convert U32 to Char", "Convert Bytes to String", and "Convert Bytes to Mutable String", the safe alternatives introduce significant overhead, ranging from `2.02x` to `4.27x`. However, these do not represent the generic situation, as we test the target code in an isolated way. Putting the target code—typically sparse—into an entire application, their impact on the runtime performance will be extremely diluted and invisible.

6 CONCLUSION

This paper presents an empirical study of how developers use `unsafe` Rust blocks in practice. It unveils, while the `unsafe` blocks are unavoidable due to reasons like an immature ecosystem, they are not necessary in many cases. Following the study, we further present an IDE tool that detects common patterns of unnecessary `unsafe` blocks and suggests safe alternatives. Through empirical evaluations, we show that our tool can help eliminate a significant amount of unnecessary `unsafe` blocks and the bugs they introduce.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their valuable feedback. This research was supported by the National Science Foundation (NSF) awards CNS-2213727 and OAC-2319880, as well as the DARPA award D21AP10116-00. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the US government, NSF, or DARPA.

REFERENCES

[1] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[2] Rust Fuzzing Authority. 2017. cargo-fuzz. https://github.com/rust-fuzz/cargo-fuzz.

[3] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. RUDRA: finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 84–99.

[4] Jansens Dana. 2013. The Use of Rust in the Chromium. https://security.googleblog.com/supporting-use-of-rust-in-chromium.html.

[5] Pinho Eduardo. 2016. safe-transmute-rs: A safeguarded transmute() for Rust. https://github.com/nabijaczleweli/safe-transmute-rs/pull/36.

[6] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust used safely by software developers?. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 246–257.

[7] Triplett Josh, Matsakis Niko, and Jung Ralf. 2015. Union: Provide native support for C-compatible unions. https://github.com/rust-lang/rfcs/blob/master/text/1444-union.md.

[8] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2183–2196.

[9] Member of RustSec. 2021. Out-of-bounds write in nix::unistd::getgrouplist. https://github.com/nix-rust/nix/issues/1541.

[10] Member of Servo. 2018. servo: a prototype web browser engine written in the Rust language. https://github.com/servo/servo.

[11] Meta. 2021. the Rust Foundation. https://developers.facebook.com/blog/post/2021/04/29/facebook-joins-rust-foundation/.

[12] Shane Miller and Carl Lerche. 2022. Sustainability with Rust. https://aws.amazon.com/blogs/opensource/sustainability-with-rust/.

[13] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying 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*. 763–779.

[14] Davidoff Sergey, Arcieri Tony, and Gaynor Alex. 2020. RustSec: is a repository of security advisories filed against Rust crates. https://github.com/rustsec/advisory-db.

[15] The Rust Team. 2015. Miri: An interpreter for Rust's mid-level intermediate representation. https://github.com/rust-lang/miri.

[16] The Crates.io Team. 2015. crates: Source code for the default Cargo registry. https://github.com/rust-lang/crates.io/.

[17] The Google Team. 2021. Android Gabeldorsche Bluetooth Stack. https://android.googlesource.com/.

[18] The Member of RustSec. 2021. Optional Deserialize implementations lacking validation. https://github.com/gz/rust-cpuid/issues/43.

[19] Bratt Thomas, Rubio Jesús, and Boegli Antoine. 2020. awesome-rust: A curated list of Rust code and resources. https://github.com/awesome-rust-com/awesome-rust.

[20] Paul Thurrott. 2023. Microsoft is Rewriting Parts of the Windows Kernel in Rust. https://www.thurrott.com/windows/282471/microsoft-is-rewriting-parts-of-the-windows-kernel-in-rust.

[21] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. 2022. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–6.