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

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, Boin 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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.1145/3611643.3613878

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
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: Lifetime, 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
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. ▶ First, we run a basic analysis to identify and categorize `unsafe` code based on their high-level goals described in subsection 2.2. ▶ 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. ▶ 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 dereferencing, 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. ▶ 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. ▶ 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 dereference. ▶ Rust programs often invoke functions developed in foreign languages, in particular GLib. 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.

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. ▶ cases implement mutable in-order traversal on trees where two mutable references to the same node are needed. ▶ 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 unnecessary, 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 dominates 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.
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 version 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 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::transmute()` and `mem::transmute_copy()`, to mutate 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

<table>
<thead>
<tr>
<th>Code Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsafe { mem::transmute(v) };</code></td>
<td>To create a C-compatible string from a byte vector, developers often use <code>unsafe</code> functions without checks for interior 0 bytes. The function has a safe alternative <code>CString::new()</code> that asserts on interior 0 bytes.</td>
</tr>
</tbody>
</table>
| `unsafe { ptr::copy(vsrc, vdest, cnt); }`                                   | 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 has a safe alternative `copy_from_slice()`.
| `unsafe { mem::transmute::<u8, *const u8>(v) };`                           | 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()` without boundary checks. The same functionality can be completed with safe function `copy_within` with built-in boundary checks. |
| `unsafe { from_utf8_unchecked(v) };`                                        | A popular way to create a new vector in Rust is using `with_capacity()` in conjunction with the `unsafe` `set_len()` function. 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. |
| `unsafe { from_utf8(v).unwrap(); }`                                        | To resize an existing vector, we can use `resize`() 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. |
| `unsafe { mem::transmute::<u8, char>(v) };`                                | To obtain a typed value from a type. Developers often use `unsafe` function `get_unchecked()` or `get_unchecked_mut()` to bypass the boundary checks. Such functions have safe alternatives `get()` and `get_mut()` that prevent potential out-of-bounds indexing. |
| `unsafe { from_utf8(v).unwrap(); }`                                        | Converting a slice of bytes to a string slice can be done with `unsafe` function `from_utf8_unchecked()` or `from_utf8_unchecked()` without checking whether the bytes are valid UTF-8. This can be replaced with safe versions `from_utf8()` and `from_utf8().unwrap()` to perform a validation check before returning results. |

### 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.
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 file’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.

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.

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.

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

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Create Vector</th>
<th>Resize Vector</th>
<th>Copy Within Vector</th>
<th>Copy Non-Overslapping</th>
<th>Create String</th>
<th>Get String Len</th>
<th>Get Reference</th>
<th>Bytes to String</th>
<th>u32 to char</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratio</td>
<td>62.5%</td>
<td>12.5%</td>
<td>25%</td>
<td>2.5%</td>
<td>7.5%</td>
<td>1%</td>
<td>4%</td>
<td>1%</td>
<td>1%</td>
<td>40%</td>
</tr>
</tbody>
</table>

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.
“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