Is UNSAFE an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming

Mohan Cui
School of Computer Science,
Fudan University, China

Shuran Sun
School of Computer Science,
Fudan University, China

Hui Xu*
School of Computer Science,
Fudan University, China

Yangfan Zhou
School of Computer Science,
Fudan University, China

ABSTRACT

Rust is an emerging, strongly-typed programming language focusing on efficiency and memory safety. With increasing projects adopting Rust, knowing how to use Unsafe Rust is crucial for Rust security. We observed that the description of safety requirements needs to be unified in Unsafe Rust programming. Current unsafe API documents in the standard library exhibited variations, including inconsistency and insufficiency. To enhance Rust security, we suggest unsafe API documents to list systematic descriptions of safety requirements for users to follow.

In this paper, we conducted the first comprehensive empirical study on safety requirements across unsafe boundaries. We studied unsafe API documents in the standard library and defined 19 safety properties (SP). We then completed the data labeling on 416 unsafe APIs while analyzing their correlation to find interpretable results. To validate the practical usability and SP coverage, we categorized existing Rust CVEs until 2023-07-08 and performed a statistical analysis of std unsafe API usage toward the crates.io ecosystem. In addition, we conducted a user survey to gain insights into four aspects from experienced Rust programmers. We finally received 50 valid responses and confirmed our classification with statistical significance.

CCS CONCEPTS

• **Software and its engineering** → **General programming languages**; **Development frameworks and environments**.

KEYWORDS

Unsafe Rust, Safety Property, Rustdoc, CVE, User Survey, Undefined Behavior

ACM Reference Format:
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 *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639136

*Corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639136

1 INTRODUCTION

Rust is an emerging system programming language focusing on memory safety and efficiency [41]. It provides memory-safe guarantees via compile-time checks [18]; consequently, programmers must adhere to various syntactic constraints to satisfy the verification [76]. As a system programming language, it employs several zero-cost abstractions [32] to transform data without sacrificing performance [33] (*e.g.*, generic types). Although Rust has a steep learning curve [18], it has attracted many programmers due to its safety and efficiency [34]. Since 2016, Rust has been the most popular programming language in the open-source community [43–49], with many projects refactoring code in Rust [36, 37, 52].

As Rust continues to evolve, knowing how to use Unsafe Rust is essential for Rust security [5]. Safety isolation is one of the revolutionary innovations introduced by Rust [51]. It divides the portions the compiler can ensure safety into Safe Rust and adds the *unsafe* keyword as the superset [33]. The primary document defines Unsafe Rust as a keyword and a set of operations [50]. Any code with unsafe operations must be wrapped in an unsafe block. If not, programmers will trigger compilation errors. Without strict compiler checks in the unsafe scope, Rust developers may become insensitive to satisfying safety requirements, which is error-prone to causing undefined behavior (UB).

How does Rust document safety requirements for unsafe operations? We observed that most safety requirements are specified on a ***Safety*** label in Rust std. The Rust standard library [26] provides documents for unsafe APIs that are relatively comprehensive. As shown in Figure 1, we chose one API as the typical example. When calling `ManuallyDrop::take` [60], it has a safety requirement to be manually reviewed: Users cannot use this container again. Otherwise, it would trigger undefined behavior. Its implementation calls unsafe function `ptr::read` [64] (line 5), prompting us to think they may have analogous safety descriptions.

Unfortunately, Unsafe Rust does not provide developers with unified safety descriptions or systemic safety requirements. Recent research found that misusing several unsafe APIs may result in memory-safety issues [74], where overlapped owners can be created and cause double free [6, 11], such as `ManuallyDrop::take` and `*mut T::read`. Table 1 lists them with documents in Rust 1.70. Like `ManuallyDrop::take`, using a ***Safety*** label to start the safety

Table 1: A list of APIs with similar side effects and their document slices in Rust 1.70. These APIs accept mutable pointers as input and return a typed owner. In previous research, it has been reported that misuse of these APIs may result in double-free issues. The consistency and clarity of these documents need improvement: Related descriptions are not always located within the *Safety* section, and the description of the safety requirement (underlined) or side effect (bolded) is insufficient. We expect a specific error type to be defined.

Implemented Type	Unsafe Method	Safety Description Slices in API Documents.
impl<T: ?Sized> *mut T	**fn read(self) -> T**	read creates a bitwise copy of T, regardless of whether T is Copy. If T is not Copy, using both the returned value and the value at *src can **violate memory safety**. Note that assigning to *src counts as a use because it will attempt to drop the value at *src.
impl<T> ManuallyDrop<T>	**fn take(&mut ManuallyDrop<T>) -> T**	This function semantically moves out the contained value without preventing further usage, leaving the state of this container unchanged. It is your responsibility to ensure that this ManuallyDrop is not used again.
impl<T: ?Sized> Box<T>	**fn from_raw(*mut T) -> Self**	This function is unsafe because improper use may lead to **memory problems**. For example, a **double-free** may occur if the function is called twice on the same raw pointer.
impl<T: ?Sized> Rc<T>	**fn from_raw(*const T) -> Self**	The raw pointer must have been previously returned by a call to Rc<U>::into_raw where U must have the same size and alignment as T. The user of from_raw has to make sure a specific value of T is only dropped once.
impl CString	**fn from_raw(*mut c_char) -> Self**	This should only ever be called with a pointer that was earlier obtained by calling CString::into_raw. Other usage (e.g., trying to take ownership of a string that was allocated by foreign code) is likely to lead to **undefined behavior** or **allocator corruption**.
impl<T> Vec<T>	**fn from_raw_parts(*mut T, usize, usize) -> Self**	The ownership of ptr is effectively transferred to the Vec<T> which may then deallocate, reallocate or change the contents of memory pointed to by the pointer at will. Ensure that nothing else uses the pointer after calling this function.
impl String	**fn from_raw_parts(*mut u8, usize, usize) -> Self**	The ownership of buf is effectively transferred to the String which may then deallocate, reallocate or change the contents of memory pointed to by the pointer at will. Ensure that nothing else uses the pointer after calling this function.

Listing (1) Source code of ManuallyDrop::take in Rust std.

```
// impl<T> ManuallyDrop<T>
pub unsafe fn take(slot: &mut ManuallyDrop<T>) -> T {
    // SAFETY: we are reading from a reference, which is
    // guaranteed to be valid for reads.
    unsafe { ptr::read(&slot.value) }
}
```

Listing (2) Document of ManuallyDrop::take in Rust 1.70.

```
1  Takes the value from the ManuallyDrop<T> container out.
2  This method is primarily intended for moving out values in
       drop. Instead of using ManuallyDrop::drop to manually
       drop the value, you can use this method to take the
       value and use it however desired.
3  Whenever possible, it is preferable to use into_inner
       instead, which prevents duplicating the content of the
       ManuallyDrop<T>.
4  Safety
5  This function semantically moves out the contained value
       without preventing further usage, leaving the state of
       this container unchanged. It is your responsibility to
       ensure that this ManuallyDrop is not used again.
```

Figure 1: Example of an unsafe API in Rust std. Listing 1 provides the source code of an unsafe method of struct ManuallyDrop<T>. Listing 2 extracts the document in Rust 1.70. The document introduces the usage, functionality, and safety requirements to comply with by using a section labeled *Safety*.

description is intuitive. The majority of the listed APIs adhere to this criterion, such as implementations for `String`, `Vec<T>`, `CString`, and `Box<T>`. However, the `Rc<T>` lacks the ***Safety*** section, and the related issue caused by `read` is described in the outer section. At last, the texts of side effects exhibit differences: Only `Box<T>` explicitly states the potential double-free that may arise.

The unsafe API documents should systematically classify safety requirements for users to comply with. This paper comprehensively categorizes fine-grained safety requirements when crossing unsafe boundaries. In general, this paper seeks to address the following research questions (RQs):

- **RQ-1.** What finer-grained safety properties (requirements) should be satisfied across the Unsafe Rust boundary? (§3)
- **RQ-2.** Can those safety properties cover existing vulnerabilities caused by Unsafe Rust? (§4)
- **RQ-3.** How helpful are those safety properties for real-world Unsafe Rust programming? (§5)

For each RQ, our study introduces several sub-experiments. To answer RQ1, we extracted all public unsafe APIs within the Rust standard library [26] and manually audited the document. We categorized the safety requirements across the unsafe boundary as **Safety Properties** (SPs). We completed the data labeling for those APIs and then conducted a correlation analysis to find interpretable results. To answer RQ2, we examined all Rust CVEs [19] until 2023-07-08 and filtered through the root causes by misusing unsafe code, categorizing them according to Safety Properties to validate our classification. Then, we collected and analyzed the distribution of unsafe APIs within the crates.io [21] ecosystem. To answer RQ3, we surveyed experienced Rust developers. We provided participants with the definition of each SP and its minimal Proof of Concept

(PoC). We studied whether the developers acknowledged our categorization and whether each Safety Property was beneficial for unsafe programming.

Reviewing documents of unsafe APIs in Rust std, we performed an audit on 416 unsafe APIs. As a result, we identified and defined 19 safety properties (SP), categorized into two major categories: precondition and postcondition. Subsequently, we completed the SP labeling for all unsafe APIs, creating two datasets for correlation analysis. The results revealed six crucial SPs that users need to satisfy when dereferencing. Next, we classified the existing Rust CVEs based on safety properties, with 196 of 404 resulting from unsafe code. Notably, 86.73% of these errors were attributable to misuse of the standard library. Therefore, we conducted a statistical analysis of std unsafe API usage for the Rust ecosystem, which included 103,516 libraries on crates.io. Finally, we conducted user surveys targeting developers with over one year of Rust experience, having written over 5,000 lines of code and using over 1,000 lines of unsafe code. The evaluations for each SP were rated on four dimensions: precision, significance, usability, and frequency. We received 50 valid responses and conducted data analysis on them.

Our main contributions are listed as follows:

- We performed the first empirical study by learning unsafe API documents from the standard library to classify safety requirements across unsafe Rust boundaries.
- We classified 19 safety properties into two categories. All std-unsafe APIs were audited and labeled with safety properties. The labeled data were evaluated via correlation analysis, yielding interpretable results.
- We categorized all Rust CVEs based on safety properties, forming a collection of related issues that can serve as a benchmark. Unsafe API usage statistics are collected within crates.io to understand the usage frequency of unsafe APIs.
- We conducted an online survey and confirmed our categorization of safety properties with statistical significance.

2 BACKGROUND

2.1 Working with Unsafe Rust

Rust is subdivided into Safe Rust and Unsafe Rust, with Unsafe Rust being a superset [51]. Safe Rust ensures type and memory safety, preventing undefined behavior [5]. However, it lacks low-level controls over implementation details (*e.g.,* manual memory management). Unsafe Rust is an essential design feature to achieve low-level control at the system level [25]. It is employed if it has performance requirements or needs to interact with operating systems, hardware, or other programming languages.

unsafe **Keyword**. *unsafe* keyword can be used in declarations and code blocks. The first scenario indicates that the functions cannot be called in the safe code. Misuse may trigger undefined behavior. In code blocks, it signifies the scope that may violate safety guarantees without compiler-time checks, and the code requires manual auditing to ensure safety. This keyword acts as a railing, separating the safe and unsafe portions: All unsafe parts are encapsulated within this scope. The trust relationship between safe and unsafe parts is asymmetric [25]. When using an unsafe block, careful inspection is required to ensure that the data from the safe portion adheres to the contracts of the unsafe APIs. Conversely, when writing safe code, it is assumed that the unsafe code is correct and would not trigger undefined behavior.

unsafe **Operations**. Safe Rust and Unsafe Rust are designed for different scenarios. Safe Rust is a safe programming language designed for tasks that do not require low-level interactions. Contrariwise, Unsafe Rust fully leverages the capabilities of a systems-level programming language. Unlike languages such as C/C++, which are inherently unsafe, Unsafe Rust still requires adherence to certain contracts from the safe portion, such as ownership. The main differences in Unsafe Rust are that you can *1) Dereference raw pointers; 2) Call unsafe functions; 3) Implement unsafe traits; 4) Mutate static variables; and 5) Access fields of unions* [25]. These operations provide flexibility but come with the responsibility of the users to manually ensure correctness and safety.

Recent empirical research [76] suggests that Rust's safety mechanisms could be more learner-friendly. This study explored the learning challenges of its safety mechanisms by analyzing Stack Overflow comments and conducting user surveys, but it is restricted to Safe Rust. Instead, learning and utilizing Unsafe Rust is a prerequisite for advanced Rust developers.

2.2 Undefined Behavior in Rust

The undefined behavior in Rust is limited to include [23]:

- Dereferencing (using the * operator on) dangling or unaligned raw pointers.
- Breaking the pointer aliasing rules. References and boxes must not be dangling while they are alive.
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.
- Executing code compiled with platform features that the current thread of execution does not support.
- Producing invalid values, as explained in Table 2, even in private fields and locals.
- Mutating immutable data. All data inside a const item, reached through a shared reference or owned by an immutable binding, is immutable.
- Causing data races.
- Invoking undefined behavior via compiler intrinsics.
- Incorrect use of inline assembly.

This categorization is based on the side effects introduced by unsafe code. Since no formal model of Rust's semantics defines precisely what is and is not permitted in unsafe code [23], additional behavior may be deemed vulnerable. In this paper, we additionally introduce the following issues as program vulnerabilities if they are triggered by unsafe code:

- Causing a memory leak and exiting without calling destructors.
- Triggering an unreachable path then aborting (or panicking).
- Arithmetic overflow.

These undefined behavior and vulnerabilities serve as the basis for classifying safety requirements. Other errors fall outside the scope (*e.g.,* deadlocks and logic errors).

3 STUDYING UNSAFE DOCUMENTS IN STD

This section presents how we extract and define systematic safety requirements as **Safety Properties** (SP) from the existing unsafe

Table 2: Invalid value for Rust types, alone or as a field of a compound type, will trigger undefined behavior.

Rust Type	Invalid Value
!	Invalid for all values.
bool	Not 0 or 1 in bytes.
char	Outside [0x0, 0xD7FF] & [0xE000, 0x10FFFF].
str	Has uninitialized memory.
numeric i*/u*/f*	Reads from uninitialized memory.
enum	Has an invalid discriminant.
reference	Dangling, unaligned, or pointing to an invalid value.
raw pointer	Reads from uninitialized memory.
Box	Dangling, unaligned, or pointing to an invalid value.
fn pointer	NULL.
wide reference	Has invalid metadata. dyn Trait is invalid if it is not a pointer to a vtable for Trait that matches the actual dynamic trait the pointer or reference points to, and slice is invalid if the length is not a valid usize.
custom type	Has one of those custom invalid values.

documents in the standard library [26]. Our classification allows us to clarify the primary conditions and constraints necessary for Unsafe Rust, hence answering RQ1.

3.1 Preprocess on Rust Documents

Rustdoc [24] is the document system for the Rust programs, which enables the description of functionalities, requirements, expected results, and sample code snippets for APIs and crates. Intuitively, input requirements and side effects within an unsafe API must be explicitly specified in Rustdoc. We found that the document in the standard library is one of the most comprehensive resources for safety annotations within the Rust ecosystem. We thus audited documents of all public unsafe methods within the standard library as the knowledge base.

3.1.1 Design Goals. Table 1 reveals that even in the standard library: (i) the expression of the same safety requirement is not universally consistent; (ii) the enumeration of the safety requirements and side effects is not always sufficient. Thus, we manually categorize and define a series of finer-grained safety requirements as **Safety Properties** (SP), which need to satisfy the following design goals:

GOAL 3.1. ***Generality**: SP abstracts safety requirements not specific to one particular API's intricacies.*

GOAL 3.2. ***Unambiguous**: SP intends to adopt the existing terminology and explanations as much as feasible in Rust.*

GOAL 3.3. ***Nonoverlapping**: SP does not overlap, although they may be correlated.*

GOAL 3.4. ***Composability**: An Unsafe API's safety requirements can comprise several SPs.*

GOAL 3.5. ***Essentiality**: Failure to comply with any SP would cause undefined behavior or additional vulnerabilities.*

GOAL 3.6. ***Practicality**: SP is valuable and needs to be seriously considered in real-world programming scenarios.*

GOAL 3.7. ***Unilingual**: SP disregards the Foreign Function Interface (FFI) and the intrinsic requirements of other programming languages.*

By adhering to these principles, the extracted safety properties aim to provide a comprehensive and practical understanding of the safety considerations associated with Unsafe Rust. It maintains compatibility with Rust's existing terminology and avoids unnecessary complexities related to FFI.

3.1.2 Preprocessing. We noticed the redundancy in the standard library, such as std and core having an intersection. Thus we performed the following preprocessing for all unsafe APIs within std/core/alloc in Rust 1.70, including stable and nightly channels:

FILTER 3.1. *For the methods exposed by both* core *and* std*, we kept only one of them.*

FILTER 3.2. *For methods belonging to similar numeric types, we kept only one implementation.*

FILTER 3.3. *For compiler intrinsics, we retained only those with no stable counterpart.*

As a result, we obtained a collection of unsafe APIs comprising 416 unsafe methods, with 127 being folded as 11 unique APIs by Filter 3.2 (*e.g.,* unchecked_mul::<u8>/::<u16> [68, 69] are merged). By applying the preprocessing step, we aimed to streamline and consolidate an unsafe API collection for further analysis and investigation.

3.2 What Safety Properties Should We Satisfy?

A code audit of all API documents within the collection was conducted to determine what safety properties correspond with the design goals. As shown in Table 3, we divided all safety properties into two main categories with 19 subdivisions.

3.2.1 Working Procedure. There are three rounds of code auditing. For each API, we audited five sections: the functionality description, the safety description, the subchapters, the example code snippets, and the source code with its comments.

First Round: SP Construction. The first round constructs a comprehensive SP category for subsequent API labeling. Both the first and second authors participated fully in the entirety of the audit procedure. It has a sequential process: we maintain a queue of unsafe APIs that require auditing. For 30 APIs as a batch, the first author audits them and extracts unpresented SP, and then the second author verifies them, which forms an assembly line. The first author cannot begin a new batch unless the verification is completed.

Second Round: Initial Label. Following the achievement of SP categorization in the first round, the subsequent audit phase was dedicated to the task of labeling unsafe APIs. The first and second authors conducted the second round of auditing concurrently for all unsafe APIs. Both authors conducted parallel reviews until all APIs were completed, then resolved inconsistencies in SP labels. Because our classification method employs a referenced table as the ground truth, conflicts during the labeling process are slight.

Third Round: Correction Review. The process and conflict resolution in the third round of the audit are completely the same as in the second round. It serves solely for the secondary confirmation, correction, and supplementation of SP annotations.

Table 3: Safety properties learned from unsafe API documents in Rust standard library. Precondition and postcondition are two primary categories, and they have 19 subitems in total. We present the sum of the labeled unsafe API for each safety property and provide a detailed definition. Each safety requirement was also given a typical unsafe API as an example. Note that each SP may contain various sub-scenarios.

Safety Property (SP)	SUM	Definition and the safety requirement of each Safety Property.	Unsafe API Example
Precondition Safety Property			
Const-Numeric Bound	72	Relational operations allow for **compile-time** determination of the **constant** numerical boundaries on one side of an expression, including overflow check, index check, etc.	**impl**<**T: ?Sized**> *mut **T**::offset_from
Relative-Numeric Bound	114	Relational operations involve expressions where **neither side** is a constant numeric, including address boundary check, overlap check, size check, variable comparison, etc.	**trait Allocator**::grow
Encoding	16	Encoding format of the string, includes valid **UTF-8** string, valid **ASCII** string (in bytes), and valid **C-compatible** string (nul-terminated trailing with no `nul` bytes in the middle).	**impl String**::from_utf8_unchecked
Allocated	134	Value stored in the **allocated memory**, including data in the valid stack frame and allocated heap chunk, which cannot be `NULL` or dangling.	**impl**<**T: Sized**> **NonNull**<**T**>::new_unchecked
Initialized	59	Value that has been initialized can be divided into two scenarios: **fully initialized** and **partially initialized**. The initialized value must be **valid** at the given type (a.k.a. **typed**).	**impl**<**T**> **MaybeUninit**<**T**>::assume_init
Dereferencable	96	The memory range of the given size starting at the pointer must all be within the bounds of a **single allocated object**.	**impl**<**T: ?Sized**> *const **T**::as_ref
Aligned	67	Value is **properly aligned** via a specific **allocator** or the attribute `#[repr]`, including the alignment and the padding of one Rust type.	**impl**<**T: ?Sized**> *mut **T**::swap
Consistent Layout	110	Restriction on Type Layout, including 1) The **pointer's** type must be compatible with the **pointee's** type; 2) The **contained value** must be compatible with the generic parameter for the smart pointer; and 3) Two types are **safely transmutable**: The bits of one type can be **reinterpreted** as another type (bitwise move safely of one type into another).	**impl**<**T: ?Sized**> *mut **T**::read
Unreachable	9	Specific value will trigger **unreachable data flow**, such as **enumeration** index (**variance**), boolean value, closure and etc.	**impl**<**T**> **Option**<**T**>::unwrap_unchecked
Exotically Sized Type	24	Restrictions on Exotically Sized Types (EST), including **Dynamically Sized Types** (DST) that lack a statically known size, such as trait objects and slices; **Zero Sized Types** (ZST) that occupy no space.	**trait GlobalAlloc**::alloc
System IO	25	Variables related to the **system IO** depends on the **target platform**, including TCP sockets, handles, and file descriptors.	**trait FromRawFd**::from_raw_fd
Thread	3	Types that can be **transferred** across threads (**Send**) or types that can be safe to **share references** between threads (**Sync**), respectively.	**std**::**marker**::**Sync**
Postcondition Safety Property			
Dual Owner	31	Multiple owners (**overlapped objects**) that share the same memory in the ownership system by **retaking the owner** or **creating a bitwise copy**.	**impl**<**T: ?Sized**> **Box**<**T**>::from_raw
Aliasing & Mutating	30	Aliasing and mutating rules may be violated, including 1) The presence of **multiple** mutable references; 2) The **simultaneous** presence of mutable and shared references, and the memory the pointer points to cannot get mutated (**frozen**); 3) Mutating **immutable data** owned by an immutable binding.	**impl CStr**::from_ptr
Outliving	28	**Arbitrary lifetime** (unbounded) that becomes as big as context demands or **spawned thread**, may outlive the pointed memory.	**impl**<**T: ?Sized**> *const **T**::as_uninit_ref
Untyped	20	Value may not be in the **initialized state**, or the **byte pattern** represents an **invalid value** of its type.	**core**::**mem**::zeroed
Freed	17	Value may be manually **freed** or **released** by automated **`drop()`** instruction.	**impl**<**T: ?Sized**> **ManuallyDrop**<**T**>::drop
Leaked	13	Value may be **leaked** or **escaped** from the ownership system.	**impl**<**T: ?Sized**> *mut **T**::write
Pinned	5	Value may be **moved**, although it ought to be pinned.	**impl**<**P: Deref**> **Pin**<**P**>::new_unchecked

[1] `Send` [58] and `Sync` [59] are unsafe traits (markers) that are automatically implemented by the compiler when it determines that they are required. Therefore, they lack associated methods.
[2] The difference between `DualOwner` and `AliasingMutating` is that `DualOwner` only focuses on objects instead of pointers and references.

SP Creation (Round 1 Only). The SP collection is initially empty. The first author creates SPs, and others are allowed to modify them. A new SP is generated whenever a safety description cannot be classified under the existing scheme. Whenever one author suggests creating a new SP, the final decision is reached through a joint review. The name of SP is primarily extracted from the documents; the novel concepts we introduced are `DualOwner` and `ConsistentLayout`.

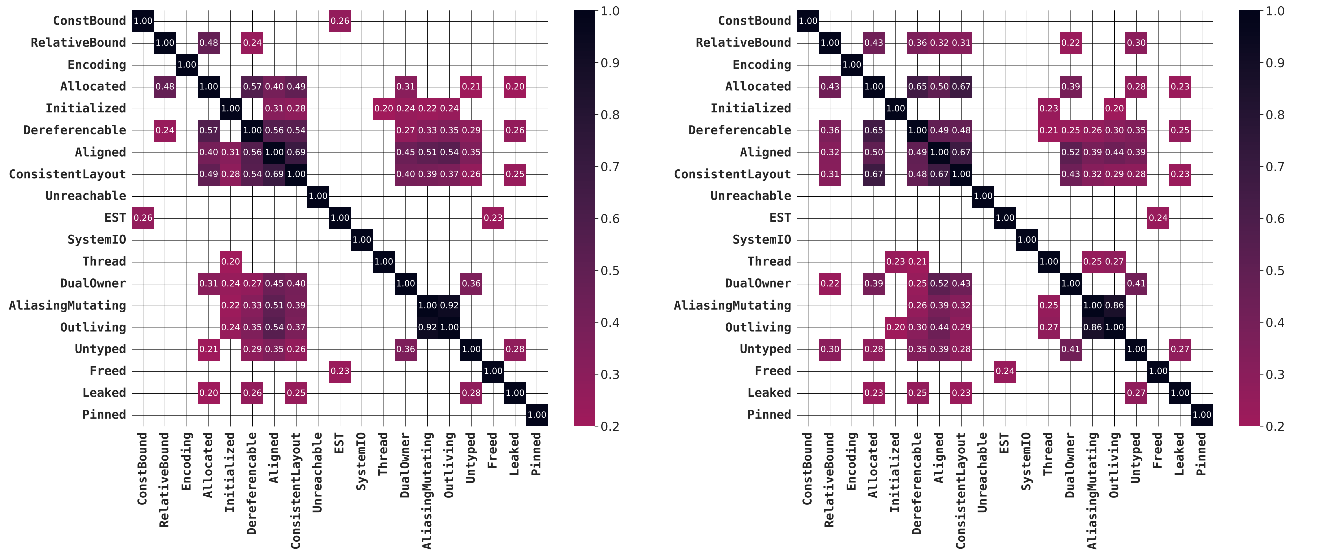
SP Modification (Round 1 Only). The author who considers that SPs can be modified can submit a request for SP modified approval, which is contingent on the agreement of a joint review. For example, integer overflow and static array index checks can be classified as const-numeric bounds, and we split `DualOwner` from `Aliasing` because the targeted types are different (objects and pointers).

Conflict Resolution. In cases of opinion conflicts during SP creation, SP modification, or SP labeling, a joint review is initiated. The joint review involves all four authors voting together: only all the authors vote in agreement for SP creation or SP modification, and more than three authors vote in agreement for labeling SP, which makes the action implemented. Otherwise, the original decision is retained.

3.2.2 *Categories.*

We have divided the safety properties into two categories based on the state of the function execution as in program testing [7], with no overlap between the sub-items.

Precondition Safety Property (PRE-SP). The precondition assumes that if the input values do not satisfy the safety requirements, the



(a) Correlation analysis results on the large dataset (original).

(b) Correlation analysis results on the small dataset (filtered).

Figure 2: Correlation matrices for both the large and small datasets. Each figure only includes the sections with weak correlation and above (correlation greater than 0.2).

function call will trigger undefined behavior or additional vulnerabilities in Section 2.2. Thus, any given API can be regarded as a black box for single-step execution [71] at the call site, regardless of its internal implementation. PRE-SP complies with the initial characteristic of unsafe function (*i.e.*, it cannot ensure safety for arbitrary inputs). In Table 3, we summarize 12 PRE-SPs. For example, `swap` [65] has the description *"Both x and y must be properly aligned."*, thus categorized into `Aligned`.

Postcondition Safety Property (POS-SP). The previous assumption leads to the deduction that the function can be safely called if the proper inputs are supplied. However, this assurance only concerns the current program point. POS-SP focuses on the potential safety issues that may arise from the subsequent operations, assuming that the input values satisfy all PRE-SPs needed. In Table 3, we finally summarize 7 POS-SPs. For example, `zeroed` [61] has the description *"There is no guarantee that an all-zero byte-pattern represents a valid value of some type T."*, thus categorized into `Untyped`.

The PRE-SP items are not overlapping within POS-SPs through this separation. It can be verified by a Rust design, where creating raw pointers is always safe, but dereferencing them is unsafe [23]. Similarly, we assume that PRE-SPs only affect the safety of function calls, while POS-SPs focus on the subsequent usage of inputs and return values. Furthermore, POS-SPs are only considered under the premise that all PRE-SPs are satisfied. Specifically, we merged `Aliasing` and `Mutating` based on the ground truth that all relevant APIs shared the same labels in these items. We empirically inferred that the primary side effect of breaking `Aliasing` rules is erroneously `Mutating` immutable data.

3.2.3 Corner Cases. The Rust standard library contains a list of unsafe APIs with no SP labels. We have provided an open-source resource that can be used for indexing [1]. Additional clarifications are given, which fall into the following categories:

- Several FFI functions with no safety requirements in the document (2), *e.g.,* `core::ffi::VaListImpl<'f>::arg` [54].
- The compilation procedure, containing code generation, intrinsic tagging, and LLVM (6), *e.g.,* `core::intrinsics::breakpoint` [55].
- Numerical conversions, but without an accurate, identifiable boundary (4), *e.g.,* `core::intrinsics::nearbyintf32` [56].
- Lacking any explanation of why they are considered unsafe (1), *e.g.,* `core::intrinsics::pref_align_of` [57].

There is no std API that has all SP labels. While it is theoretically feasible to be tagged with all SPs, the probability is exceedingly low because many SPs are connected with particular types. For example, `ConstBound` is associated with the numeric type, whereas `Encoding` pertains to strings. In practice, it is extremely challenging to introduce such complex inputs along with different types. But users can construct by hand an excessively intricate function that necessitates all safety requirements.

3.3 Correlation Analysis on Safety Properties

We obtained a valuable dataset after completing the labeling for unsafe API collection. Although one of our design goals focuses on nonoverlapping, it is still necessary to investigate potential correlations between different SPs. This notion is from the empirical

[1] https://github.com/Artisan-Lab/SafetyProperty

Table 4: SP pairs with correlation coefficients (CC) greater than 0.4 both in the large and small dataset correlation matrices.

SP1	SP2	AVG-CC	SP1	SP2	AVG-CC
Precondintion Safety Properties ONLY					
Allocated	RelativeBound	0.455	Allocated	Dereferencable	0.615
Allocated	ConsistantLayout	0.580	Allocated	Aligned	0.450
Dereferencable	Aligned	0.525	Dereferencable	ConsistantLayout	0.510
Aligned	ConsistantLayout	0.685			
Precondintion Safety Properties with Postcondition Safety Properties					
DualOwner	Aligned	0.485	DualOwner	ConsistantLayout	0.415
Outliving	Aligned	0.490	Outliving	AliasingMutating	0.895

intuition that data satisfying `Dereferenceable` should always meet `Allocated`.

3.3.1 Methodology. We conducted a correlation analysis based on two datasets, and their results demonstrate the anticipated differences. We will explain their characteristics first and then discuss the results of both datasets.

Large Dataset. The large dataset directly uses the original collection with labeled data, which includes the entire set of unsafe APIs. The labels of functionally related APIs may be similar. The intent of keeping a large dataset is to emphasize the quantity, as having adequate data can expose potential correlations.

Small Dataset. The small dataset is created by applying additional filter (Filter 3.4) to the large dataset. This is done to counteract the potential bias from excessive similar APIs. The small dataset intends to eliminate the redundancy of potentially irrelevant data and concentrate on diversity.

FILTER 3.4. *APIs must have the same labels and satisfy at least one of the following requirements:*

- *Implementations of the same method with different mutability.*
- *Implementations of the same trait for different types, including mono-morphizations in the trait or struct definitions.*
- *Functions with the same name implemented for different types within the same namespace.*
- *Encapsulation of intrinsic functions.*

3.3.2 Correlation Matrix. As depicted in Figure 2, we built correlation matrices for both the large and small datasets, retaining only the elements with moderate correlation and above (correlation coefficient > 0.20). The large dataset has a higher susceptibility to interference from redundant APIs. For example, there are 30 implementations of the trait `SliceIndex<[T]>` [67], all of which are labeled with `RelativeBound` and `Allocated` only. The correlation between them is thus higher in the large dataset, changing from 0.43 to 0.48. In the small dataset, the diverse functionality among APIs is more likely to result in the loss of pertinent data that could affect correlations. For example, the small dataset's correlation between `Aligned` and `AliasingMutating` decreases from 0.51 to 0.39. At last, `Encoding`, `Unreachable`, `SystemIO`, and `Pinned` achieve the best independence, as they have no substantial correlation with any other SPs in both matrices.

Case Study. Based on two diagrams in Figure 2, we extracted all the pairs with at least a moderate CC, as listed in Table 4. Among the six pairs with no POS-SPs, we empirically found that they are related to dereferencing operations. Although dereferencing was not considered a distinct item in the SP category, such operations are pervasive in the inner code of unsafe methods. We inferred knowledge about safety requirements for dereferencing that was not explicitly categorized: The first-class SP for a *valid pointer* with the highest priority is `Allocated`, followed by `Dereferencable`, `ConsistentLayout`, and `Aligned`. `RelativeBound` should be considered if it has pointer arithmetic. Even though `Initialized` is not stated in Table 4, we still view it as a prerequisite for dereferencing, as Rust's undefined behavior has explicit requirements for valid values of raw pointers. In this paper, we advocate for the safe usage of raw pointers by satisfying these 6 PRE-SPs.

4 VERIFYING REAL-WORLD UNSAFE CODE

This section presents the practical usability of our classification in real-world scenarios and the frequency of unsafe API usage in the Rust ecosystem. We classify existing CVEs [19] to validate SP coverage and conduct a statistical analysis of unsafe API usage on crates.io [21].

4.1 Classifying Existing Unsafe CVEs

4.1.1 Workflow. The workflow consists of two primary steps: Create a database of CVEs caused by misusing unsafe Rust and classify them into safety properties by manual code review.

CVE Set. We employed the CVE dataset from the CVE program (https://cve.mitre.org) and searched on the CVE list using the keyword ***"Rust"***. The results are sorted by CVE ID in chronological order (*i.e.,* submission date). The final CVE dataset ranged from CVE-2017-20004 [12] to CVE-2023-30624 [14]. We initially filtered CVE based on CVE descriptions, primarily retaining memory-safety issues. Unrelated CVEs were removed, such as leaking sensitive information. We filtered those CVEs triggered in the panic path because this study does not specifically work for panic safety [25]. Additionally, the retained CVEs cannot be located in a deprecated or yanked crate and should have a link to the source code to support a code audit.

CVE Audit. We performed a manual audit of error snippets that led to security issues. The first and second authors double-checked the results. We painstakingly investigated whether misusing unsafe code was the root cause of each CVE. Due to the short descriptions provided on the CVE website, we utilized various sources, including issues, pull requests, contributors (*e.g.,* RUSTSEC [27]), and fixed code, to pinpoint the source code related to each CVE. Any CVEs that did not satisfy the front criteria were removed from our dataset. Around 86.73% of the 196 CVEs in the final dataset were attributed to misusing unsafe APIs from the standard library. In contrast, the remaining CVEs were caused by dereferencing raw pointers, using non-std unsafe functions, or outside FFI. Based on the descriptions and code reviews, we further classified each CVE into the SPs it violated.

CVE Example. Figure 3 presents an example of a classified CVE derived from CVE-2021-45709 [2, 13]. Based on the explicit error description in its issue, we were able to locate the buggy source code and confirm incorrect usage of `from_raw_parts_mut` [66]. This API was annotated by the following SPs: ConstBound, RelativeBound, Allocated, Dereferencable, Aligned, ConsistantLayout,

Listing (3) Source code of CVE-2021-45709 in the crypto2 crate through 2021-10-08 for Rust.

```
#[inline]
fn xor_si512_inplace(a: &mut [u8], b: &[u32; Chacha20::
      STATE_LEN]) {
    unsafe {
        let d1 = core::slice::from_raw_parts_mut(a.
      as_mut_ptr() as *mut u32, Chacha20::STATE_LEN);
        for i in 0..Chacha20::STATE_LEN {
            d1[i] ^= b[i];
        }
    }
}
```

Listing (4) Description documented in RUSTSEC-2021-0121.

```
Description
The implementation does not enforce alignment requirements
on input slices while incorrectly assuming 4-byte alignment
through an unsafe call to std::slice::from_raw_parts_mut,
which breaks the contract and introduces undefined behavior.
```

Figure 3: Example of the CVE classification. The buggy source code and description of CVE-2021-45709 (in RUSTSEC). This CVE violates the safety requirement of `Aligned` and triggers UB when using the unsafe API slice::from_raw_parts_mut.

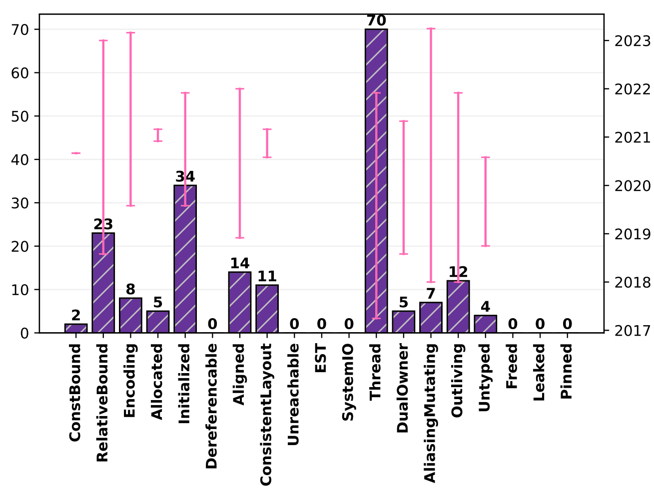


Figure 4: SP classification results and their duration on existing Rust CVEs. These CVEs are memory-safety issues resulting from misusing unsafe code and ignoring unrelated issues, such as leaking sensitive information.

AliasingMutating, and Outliving. This CVE violates the requirement of `Aligned`, leading to undefined behavior.

4.1.2 Results and Benchmark. We conducted a study on 404 CVE descriptions and performed a code review on the remaining 196 CVEs after filtering. We classified them based on SP categorization and analyzed their distribution. It has been manually verified that the causes of these CVEs do not exceed our SP classification. Finally,

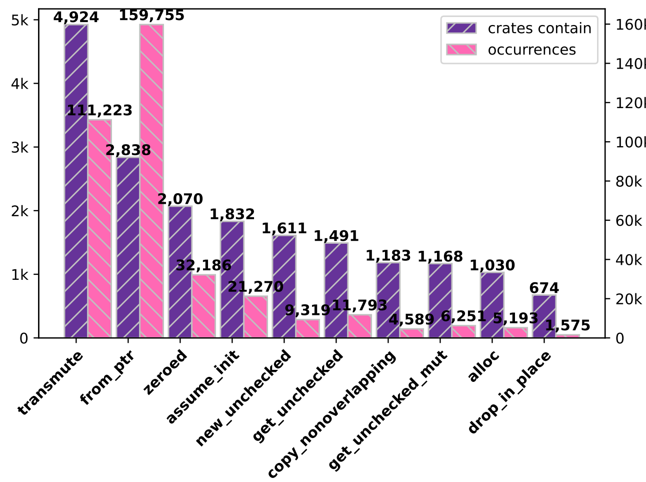


Figure 5: Statistics results on unfiltered strings (unsafe APIs) in the crates.io ecosystem. The top ten most frequently used strings across all repositories and their source code occurrences are sorted by the sum of crates.

we generated a benchmark encompassing the various SPs in the identified CVEs.

SP and Time-span Distribution. Figure 4 depicts the classification results. `RelativeBound`, `Initialized`, and `Thread` had a significant number of CVEs (at least 23). Following them, there are fewer CVEs associated with `Aligned`, `Outliving`, `ConsistentLayout`, and the other 6 SPs (ranging from 2 to 14). 7 SPs have no corresponding CVEs. Except for `ConsistentLayout`, `ConstBound`, and `Allocated`, the time span of CVEs for each SP ranges from as early as August 2019 to as late as December 2021 from a temporal perspective. This period contains approximately 91.84% of listed CVEs.

Case Study. The most CVEs were caused by `Thread` (70) violations. This is predominantly the result of user-defined types that unconditionally implement the `Send` [58]/`Sync` [59] traits or fail to ensure that `Send`/`Sync` implementations have the correct bounds. Such violations may result in data races and memory-safety issues across the thread boundaries. `Initialized` (37) was the second most common SP, and its typical scenarios are as follows: 1) Create an uninitialized buffer and pass it to the user-defined `Read` [64] implementation, allowing safe code to read uninitialized memory; 2) Increase buffer length without reserving memory, causing write-out-of-bound or dropping uninitialized memory issues; 3) Create an uninitialized `NonNull` pointer.

4.2 Statistics on crates.io Ecosystem

The findings from Section 4.1 indicate that 86.73% of the classified CVEs were caused by misusing unsafe APIs in std. This observation prompted us to conduct a statistical analysis of the usage of unsafe APIs within the Rust ecosystem.

4.2.1 Open-source Crates Database. As crates.io is the crate management platform in the Rust community, we used all its repositories to serve as a code database. To evaluate the frequency of unsafe

API usage, we matched regular expressions to source code without compilation. Using function name as the criterion, we merged identical unsafe APIs, creating a dictionary of 140 unique unsafe API strings. Then we removed 20 strings that have the same name as other safe functions in the std (*e.g.*, `add` [62, 63]). To improve the accuracy of the statistics, we only included per-file instances if the *unsafe* keyword was used in the source code.

4.2.2 Frequency Statistics. As of 2023-01-30, we mined all the latest crates from crates.io. The statistic indicates that 21,506 crates use Unsafe Rust among the 103,516 crates on crates.io (3,614 are yanked). For each string, we collected a statistical summary, including the number of crates in which the string appears and the total usage count of the string across all crates. The top ten most frequently used strings are listed in Figure 5, which depicts statistical results in two dimensions. We observed that the primary scenarios encompass type conversions (`transmute`), manual memory management (`zeroed`, `alloc`, `drop_in_place`), unsafe constructors (`new_unchecked`), deferred initialization (`assume_init`), unsafe indexing (`get_unchecked/mut`), unsafe referencing (`from_ptr`), and unsafe memory copies (`copy_nonoverlapping`). Note that the results presented above do not account for filtered strings (*e.g.*, `read`, `as_mut`, `from_raw`, etc.).

5 SURVEYING RUST PROGRAMMERS

In this section, we conducted an online survey on Goldendata [20] to evaluate the Safety Properties in precision, significance, usability, and frequency from the perspective of experienced Rust developers.

5.1 Methodology

5.1.1 Recruitment. We require participants to be at least 18 years old with a minimum of 1 year of experience in Rust programming and to have written at least 5,000 lines of Rust code, including over 1,000 lines of unsafe code. We posted our survey on the Rust-related community to recruit volunteers and emailed contributors from Rust-lang and the popular repositories on crates.io.

5.1.2 Procedure. We provide each defined SP with a representative unsafe API for participants on each page. Note that the relationship between API and SP is many-to-many. Hence, the given API only targets one SP. For each API, we further supply a triplet $(S,\ P_1,\ P_2)$, containing a document slice S for the current SP, a sound code snippet P_1, and a misused PoC P_2. P_1 and P_2 are carefully designed to be short and easy to debug. P_2 violates the safety requirements of the current SP, ensuring that all other SPs are satisfied. We link S to the online document for referencing, and both P_1 and P_2 can be redirected to the Rust Playground [22] for online execution. Furthermore, 18 out of 19 P_2 will trigger UB that can be captured by MIRI [9], making it easier for participants to understand issues.

Participants must read the definition and the triplet of each SP. Then we ask them four questions listed below:

- **Q1:** We asked participants to rank the accuracy of SP definitions. Can the safety requirements of each SP be explained in a concise and precise definition?
- **Q2:** We asked participants to rank the significance of each SP. Does violating each SP lead to unacceptable issues? Is it necessary to document such requirements explicitly in `Rustdoc`?
- **Q3:** We asked participants to rank the usability of each SP. Should users consider the context of SP satisfaction in real-world unsafe programming? Does adhering to each SP help write sound code?
- **Q4:** We asked participants to rank the frequency of each SP. How frequently do they encounter situations that require careful use of this SP? Is it often considered when crossing unsafe boundaries?

For each question, we have devised a $[-1, 0, 1]$ scoring value to represent negative, neutral, and positive responses. The range of the calculated total scores for each question in one SP is $[-50, 50]$. A higher score indicates a more favorable response, but these questions have no objectively correct answers. The responses may vary based on participants' coding experiences. To conduct a thorough evaluation of the feedback, we compute the mean and standard deviation of the total score distribution among all SPs for each individual question.

5.2 Survey Results

We distributed the survey between July 10 and July 25, 2023, and received 90 responses. After review by the first and second authors, it was determined that 50 were valid. The criteria for a valid response included excluding surveys with excessively short completion times (less than 15 minutes), the same pattern throughout the entire survey, and responses with unrelated comments.

Q1: Precision Ratings. The results of Q1 illustrate the endorsement of SP definitions, providing an in-depth appraisal of Goal 3.1 to 3.4. The score distribution is $[6, 25]$, with a mean of 19.3, a standard deviation of 4.8, and 16 SPs scoring greater than 15. We observed that SPs with brief descriptions tended to receive higher scores, such as `SystemIO` (24) and `Leaked` (25). Whereas SPs with subcategories had a lower score due to the comprehension threshold, such as `ConsistentLayout` (17) and `AliasingMutating` (14). Participants exhibited a comparatively negative attitude toward `Aligned` (6). They are also unclear about the newly introduced categories like `DualOwner` (12). In the open-ended comments, 4 participants emphasize the necessity of highlighting the side effects caused by violating `DualOwner`: *I hope the documentation to tell if it is valid to use twice on T: Copy types*; *add extra hint about panic safety.*

Q2: Significance Ratings. The results of Q2 indicate their perspectives on the safety issues caused by violating each SP, in accordance with Goal 3.5. The score distribution is $[15, 34]$, with a mean of 26.3, a standard deviation of 5.0, and 18 SPs scored greater than 20. We noticed that participants tend to be positive concerning memory safety. Rust developers devote particular attention to memory safety and evince a predictable sensitivity to the safety requirements of unsafe code. However, an exception existed, which is `Unreachable` (15). The majority of participants viewed `Unreachable` as an inconsequential problem and instinctively assumed that panic is always memory-safe, losing focus on potential threats to panic safety.

Q3: Usability Ratings. The results of Q3 represent the sensitivity to the context when crossing unsafe boundaries in real-world unsafe programming, thus addressing Goal 3.6. The score distribution is $[12, 28]$, with a mean of 21.4, a standard deviation of 4.2, and 15 SPs scored greater than 20, which are all similar to Q3. It can be explained by assuming that users may be less likely to check the requirements in real-world situations if they perceive one SP as

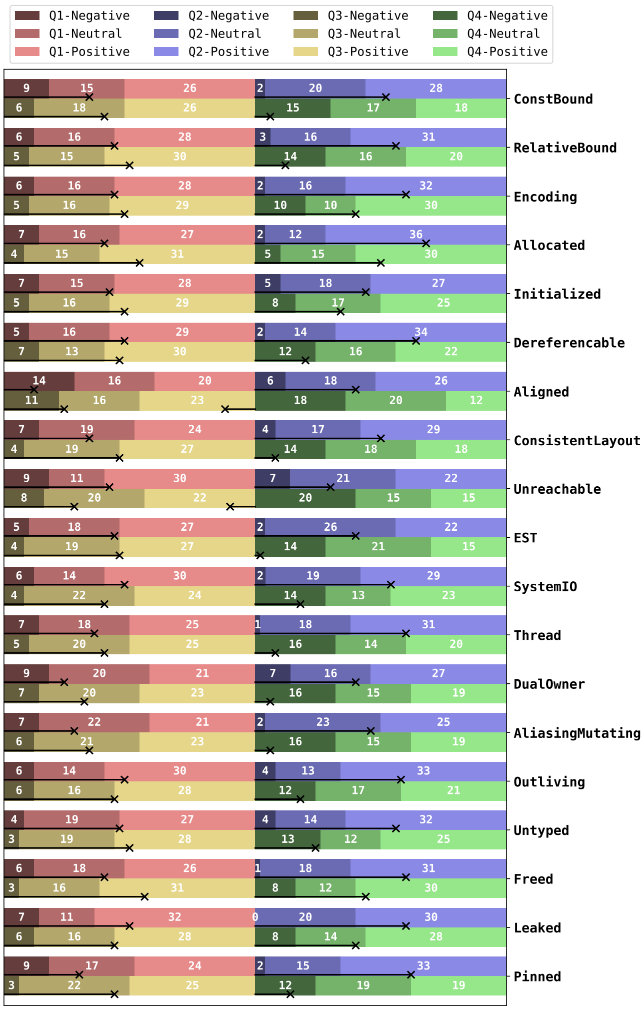


Figure 6: Survey results on Unsafe Rust programmers. Precision, significance, usability, and frequency are the four dimensions rated for each SP. Positivity, neutrality, and negativity are the options for each dimension.

insignificant. Conversely, if their programs are less affected by one SP in practice, they may perceive it as unimportant, which is consistent with their intuition. We observed that 89.4% of the Q2 scores are higher than Q3, indicating that participants may have a higher awareness of significance than programming habits in real-world practice.

Q4: Frequency Ratings. The results of Q4 reveal the frequency with which Rust developers encounter each SP in unsafe programming. The score distribution is $[-6, 25]$, with a mean of 8.6 and a standard deviation of 8.8. These responses further validated the results in Section 4.2, which measured the frequency of unsafe API usage in crates.io. We found significant discrepancies in the score distribution for this question. It shows that `Allocated` (25), `Freed` (22), `Leaked` (20), and `Initialized` (17) are encountered more frequently in Unsafe Rust. We infer that these SPs are tightly connected with common scenarios, including manual memory management and deferred initialization. As for `Encoding` (20), users may use it frequently to interact with C code in unsafe contexts. For SPs with lower or even negative scores, we suggest that the Rust developers and community may need to pay more attention to avoid misusing them.

The survey results confirmed our classification of safety properties with statistical significance. It is necessary to define a systemic classification, as experienced Rust programmers highly care about memory safety issues caused by unsafe code. At last, it also reveals a significant variation in the occurrence frequency of different SPs in real-world Rust programs.

6 THREATS TO VALIDITY

For internal validity threats, using std unsafe documents as our knowledge base might not provide exhaustive coverage for investigating the categories toward security requirements. We adopted a validation based on the CVE classification to address this limitation. Also, survey participants might not be representative enough; some might be malicious respondents, cheat on the programming experience, or send multiple submissions. To ensure internal validity, various measures were implemented. First, we utilized multiple recruitment channels, such as private email invitations. Second, we clearly outlined the mandatory requirements for programming experience. Third, the first and second authors manually verified all the responses. Fourth, we imposed restrictions on the number of submissions from the same IP address.

For external validity threats, due to the ongoing development of Rust, the programming style and usage of Unsafe Rust may evolve over time. Despite considering both stable and nightly channels, future updates may introduce new unsafe APIs, modify API descriptions and implementations, or even deprecate some unsafe APIs. We also acknowledge the existence of uncommon safety descriptions that cannot be classified based solely on the std unsafe documents or existing CVEs because they have not been documented in any std unsafe document or existing CVE.

At last, the process of SP construction is based on expert knowledge. Different programmers may hold divergent views regarding which sub-items of SPs should be merged or separated; thus, no definitive conclusion can be drawn. The core contribution of our work is to provide a classification and labeling method that extracts security properties. There may be other methods besides ours, but our research is the inaugural study to concentrate on the categorization of security requirements in Rust. We are optimistic that the results of our classification could benefit both the Rust and SE communities. Moreover, our method is open source, and available for all Rust developers to refer to.

7 RELATED WORK & IMPLICATIONS

Empirical Studies on Rust Security. Researchers have conducted empirical studies to understand how to use Unsafe Rust from real-world programs [5, 17, 50, 51, 75] and existing CVEs [74]. They summarize valuable bug patterns and provide insights into different aspects of safety guarantees. However, these studies do not extract safety requirements from the safety descriptions in the

Table 5: Open-source code analyzers that can detect specific SPs. Each static bug detection tool may not necessarily support all scenarios related to the corresponding SP.

Tool	Supported SPs of Each Static Analyzer
RUDRA [6]	`Thread`, `DualOwner`, `Initialized`
SAFEDROP [11]	`Allocated`, `DualOwner`, `Freed`, `Initialized`
MIRCHECKER [38]	`Const-Numeric Bound`, `Relative-Numeric Bound`, `DualOwner`
FFICHECKER [39]	`Allocated`, `Leaked` *(Rust/C FFI Only, based on LLVM)*
RCANARY [3]	`Leaked`

API documents. Several empirical studies focus on the Rust learning curve [1, 18] and the programming challenges introduced by compiler errors [76]. Researchers also leveraged Rust-related Stack Overflow data to understand real-world development problems [76]. However, we are more concerned with experienced system engineers who are proficient than Rust beginners. They need Unsafe Rust to achieve low-level control and better understand the safety requirements when crossing unsafe boundaries.

Bug detection methods in Rust. A lot of research has already been done on finding bugs in Rust programs using different methods, including formal verification [4, 10, 15, 28, 29, 31, 32, 35, 42, 73], symbolic execution [8, 40], model checking [6, 11, 38, 70, 72], interpreter [9], and fuzzing [16, 30]. We have discovered that some of the above prototypes analyze errors based on bug patterns corresponding to our SP categorization listed in Table 5. For example, a significant portion of CVEs on `Initialized` (85.3%) and `Thread` (92.9%) were discovered by RUDRA [6]. This observation suggests that analyzers designed for bug patterns may effectively identify vulnerabilities that violate specific SPs. Also, many bugs related to specific SP may not been discovered as in Figure 4.

Benefiting SE Community. CVE classification provides a set of CVE lists for each SP that can be used as a benchmark. This benchmark can be employed to evaluate the effectiveness of research prototypes or bug-detection tools designed for particular SPs, *e.g.,* `SafeDrop` and `Rudra`. As far as we know, the Rust community needs a unified, ground-truth-supported benchmark to support effectiveness comparisons based on safety issues. We advocate for setting such a benchmark to serve as a basis for the SE community.

8 FUTURE WORK

We have tagged Safety Property to each unsafe API, particularly building a true/false matrix for the Rust standard library. To promote this effort going forward, we are dedicated to reformulating the standard library as a basis for a more comprehensive unsafe document that is built upon the SP labels.

We utilize the unsafe API `Allocator::grow` [53] as a template in this section. The revised document needs to link "specific input or return value" with Safety Properties as one group. As shown in Listing 5, each safety requirement requires pairing parameter or return value with a specific SP as a summary, followed by detailed descriptions. Note that one SP can be mapped to multiple parameters, and one parameter may have multiple SP items.

After restructuring the unsafe document, we will integrate the revised SP document system into the rust-analyzer in the future. If the users invoke any unsafe API in Rust std, they will receive the corresponding safety requirements highlighted by rust-analyzer. As for now, this plugin is under the development process.

Listing 5: Revised document of Allocator::grow in Rust 1.70.

```
#SafetyProperty

ptr: Allocated
ptr must denote a block of memory currently allocated via this
      allocator.

ptr: Freed
If this returns Ok, then ownership of the memory block
      referenced by ptr has been transferred to this allocator.

old_layout: ConsistantLayout
old_layout must fit that block of memory.

new_layout & old_layout: RelativeBound
new_layout.size() must be greater than or equal to old_layout.
      size().
```

9 CONCLUSION

As Rust is a system programming language, Unsafe Rust is integral to achieving low-level control over implementation details. With increasing system software adopting Rust, understanding the safety requirements when crossing unsafe boundaries is crucial, particularly with well-defined categorization. To this end, we conducted the first comprehensive empirical study on safety requirements across the unsafe boundary. We focus on unsafe API documents in the standard library to infer safety properties, and then categorize unsafe APIs and existing CVEs. Additionally, we conducted a user survey to gain insights into four aspects of these safety properties from experienced Rust developers. Through these efforts, we aim to promote the standardization of systematic documents for Unsafe Rust in the Rust community.

ACKNOWLEDGMENTS

We thank anonymous ICSE reviewers for their valuable comments. We thank the engineers from Artisan Lab for their genuine help. This work is supported by the National Natural Science Foundation of China (No. 62372304).

REFERENCES

[1] Parastoo Abtahi and Griffin Dietz. 2020. Learning Rust: How Experienced Programmers Leverage Resources to Learn a New Programming Language. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–8. doi:10.1145/3334480.3383069.

[2] RUSTSEC-2021-0121: Non aligned u32 read in Chacha20 encryption and decryption. 2021. https://rustsec.org/advisories/RUSTSEC-2021-0121.html (Accessed on 01/11/2024).

[3] arXiv. 2023. rCanary: Detecting Memory Leaks Across Semi-automated Memory Management Boundary in Rust. In *arXiv*.

[4] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J Summers. 2022. The prusti project: Formal verification for rust. In *NASA Formal Methods Symposium*. Springer, 88–108. doi:10.1145/3547647.

[5] 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. doi:10.1145/3334480.3383069.

[6] 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. doi:10.1145/3477132.3483570.

[7] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. 2005. Symbolic execution with separation logic. In *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings 3*. Springer, 52–68. doi: 10.1007/11575467_5.

[8] MIRAI Contributors. 2024. MIRAI: Rust mid-level IR Abstract Interpreter. https://github.com/facebookexperimental/MIRAI (Accessed on 01/11/2024).

[9] Miri Contributors. 2024. Miri: An interpreter for Rust's mid-level intermediate representation. https://github.com/rust-lang/miri (Accessed on 01/11/2024).

[10] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2022. Modular information flow through ownership. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1–14. doi:10.1145/3519939.3523445.

[11] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–21. doi:10.1145/3542948.

[12] allowing for memory safety issues through race conditions. CVE-2017-20004: In the standard library in Rust before 1.19.0, there is a synchronization problem in the MutexGuard object. MutexGuards can be used across threads with any types. 2017. https://www.cve.org/CVERecord?id=CVE-2017-20004 (Accessed on 01/11/2024).

[13] allowing for memory safety issues through race conditions. CVE-2021-45709: In the standard library in Rust before 1.19.0, there is a synchronization problem in the MutexGuard object. MutexGuards can be used across threads with any types. 2021. https://www.cve.org/CVERecord?id=CVE-2021-45709 (Accessed on 01/11/2024).

[14] 7.0.1 CVE-2023-30624: Wasmtime is a standalone runtime for WebAssembly. Prior to versions 6.0.2, such as tables 8.0.1, Wasmtime's implementation of managing per-instance state, and contains LLVM-level undefined behavior. memories. 2023. https://www.cve.org/CVERecord?id=CVE-2023-30624 (Accessed on 01/11/2024).

[15] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29. doi:10.1145/3371102.

[16] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust typechecker using CLP (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 482–493. doi:10.1109/ASE.2015.65.

[17] 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*. 246–257. doi:10.1145/3377811.3380413.

[18] 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 *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, 597–616. https://www.usenix.org/conference/soups2021/presentation/fulton

[19] CVE Group. 2024. The CVE Program. https://www.cve.org (Accessed on 01/11/2024).

[20] Golendata Group. 2024. Golendata. https://www.jinshuju.net (Accessed on 01/11/2024).

[21] Rust Group. 2024. The Rust community's crate registry. https://crates.io (Accessed on 01/11/2024).

[22] Rust Group. 2024. The Rust Playground. https://play.rust-lang.org (Accessed on 01/11/2024).

[23] Rust Group. 2024. The Rust Reference. https://doc.rust-lang.org/reference (Accessed on 01/11/2024).

[24] Rust Group. 2024. The rustdoc book. https://doc.rust-lang.org/rustdoc (Accessed on 01/11/2024).

[25] Rust Group. 2024. The Rustonomicon. https://doc.rust-lang.org/nomicon (Accessed on 01/11/2024).

[26] Rust Group. 2024. TheRustStandardLibrary. https://doc.rust-lang.org/std (Accessed on 01/11/2024).

[27] Rust Secure Code Working Group. 2024. RUSTSEC: A vulnerability database for the Rust ecosystem. https://rustsec.org (Accessed on 01/11/2024).

[28] Florian Hahn. 2016. *Rust2Viper: Building a static verifier for Rust.* Master's thesis. doi:10.3929/ethz-a-010669150 .

[29] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 711–741. doi:10.1145/3547647.

[30] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust library fuzzing via API dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 581–592. doi:10.1109/ASE51524.2021.9678813.

[31] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32. doi:10.1145/3371109.

[32] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34. doi:10.1145/3158154.

[33] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.

[34] Nick Kolakowski. 2019. Fastest-Growing Programming Languages on GitHub. https://insights.dice.com/2019/11/11/10-github-programming-languages (Accessed on 01/11/2024).

[35] 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. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 286–315. doi:10.1145/3586037.

[36] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Shane Leonard, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The tock embedded operating system. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–2. doi:10.1145/3131672.3136988.

[37] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 234–251. doi:10.1145/3132747.3132786.

[38] 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. doi:10.1145/3460120.3484541.

[39] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2022. Detecting cross-language memory management issues in rust. In *European Symposium on Research in Computer Security*. Springer, 680–700. doi:10.1007/978-3-031-17143-7_33.

[40] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No panic! Verification of Rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 108–114. doi:10.1109/INDIN.2018.8471992.

[41] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104. doi:10.1145/2692956.2663188.

[42] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based verification for Rust programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 4 (2021), 1–54. doi:10.1145/3462205.

[43] Stack Overflow. 2016. Stack Overflow Developer Survey 2016. https://insights.stackoverflow.com/survey/2016#technology-most-loveddreaded-and-wanted (Accessed on 01/11/2024).

[44] Stack Overflow. 2017. Stack Overflow Developer Survey 2017. https://insights.stackoverflow.com/survey/2017#technology-most-loveddreaded-and-wanted (Accessed on 01/11/2024).

[45] Stack Overflow. 2018. Stack Overflow Developer Survey 2018. https://insights.stackoverflow.com/survey/2018#technology-most-loveddreaded-and-wanted (Accessed on 01/11/2024).

[46] Stack Overflow. 2019. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019#technology-most-loveddreaded-and-wanted (Accessed on 01/11/2024).

[47] Stack Overflow. 2020. Stack Overflow Developer Survey 2020. https://insights.stackoverflow.com/survey/2020#technology-most-loveddreaded-and-wanted (Accessed on 01/11/2024).

[48] Stack Overflow. 2021. Stack Overflow Developer Survey 2021. https://insights.stackoverflow.com/survey/2021#technology-most-loveddreaded-and-wanted (Accessed on 01/11/2024).

[49] Stack Overflow. 2022. Stack Overflow Developer Survey 2022. https://survey.stackoverflow.co/2022 (Accessed on 01/11/2024).

[50] Alex Ozdemir. 2019. Unsafe in Rust: Syntactic Patterns. (2019), (Accessed on 01/11/2024). https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax

[51] 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. doi:10.1145/3385412.3386036.

[52] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970. doi:10.1145/3373376.3378469.

[53] std::alloc::Allocator::grow. 2024. https://doc.rust-lang.org/std/alloc/trait.Allocator.html#method.grow (Accessed on 01/11/2024).

[54] std::ffi::VaListImpl::arg. 2024. https://doc.rust-lang.org/std/ffi/struct.VaListImpl.html#method.arg (Accessed on 01/11/2024).

[55] std::intrinsics::breakpoint. 2024. https://doc.rust-lang.org/std/intrinsics/fn.breakpoint.html (Accessed on 01/11/2024).

[56] std::intrinsics::nearbyintf32. 2024. https://doc.rust-lang.org/std/intrinsics/fn.nearbyintf32.html (Accessed on 01/11/2024).

[57] std::intrinsics::pref_align_of. 2024. https://doc.rust-lang.org/std/intrinsics/fn.pref_align_of.html (Accessed on 01/11/2024).

[58] std::marker::Send. 2024. https://doc.rust-lang.org/std/marker/trait.Send.html (Accessed on 01/11/2024).

[59] std::marker::Sync. 2024. https://doc.rust-lang.org/std/marker/trait.Sync.html (Accessed on 01/11/2024).

[60] std::mem::ManuallyDrop::take. 2024. https://doc.rust-lang.org/std/mem/struct.ManuallyDrop.html#method.take (Accessed on 01/11/2024).

[61] std::mem::zeroed. 2024. https://doc.rust-lang.org/std/mem/fn.zeroed.html (Accessed on 01/11/2024).

[62] std::ops::Add. 2024. https://doc.rust-lang.org/std/ops/trait.Add.html (Accessed on 01/11/2024).

[63] std::ptr::add. 2024. https://doc.rust-lang.org/std/primitive.pointer.html#method.add (Accessed on 01/11/2024).

[64] std::ptr::read. 2024. https://doc.rust-lang.org/std/ptr/fn.read.html (Accessed on 01/11/2024).

[65] std::ptr::swap. 2024. https://doc.rust-lang.org/std/ptr/fn.swap.html (Accessed on 01/11/2024).

[66] std::slice::from_raw_parts_mut. 2024. https://doc.rust-lang.org/std/slice/fn.from_raw_parts_mut.html (Accessed on 01/11/2024).

[67] std::slice::SliceIndex. 2024. https://doc.rust-lang.org/std/slice/trait.SliceIndex.html (Accessed on 01/11/2024).

[68] std::u16::unchecked_mul. 2024. https://doc.rust-lang.org/std/primitive.u16.html#method.unchecked_mul (Accessed on 01/11/2024).

[69] std::u8::unchecked_mul. 2024. https://doc.rust-lang.org/std/primitive.u8.html#method.unchecked_mul (Accessed on 01/11/2024).

[70] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: a bounded verifier for rust (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 75–80. doi:10.1109/ASE.2015.77.

[71] Gert-Jan Van Rootselaar and Bart Vermeulen. 1999. Silicon debug: scan chains alone are not enough. In *International Test Conference 1999. Proceedings (IEEE Cat. No. 99CH37034)*. IEEE, 892–902. doi:10.1109/TEST.1999.805821.

[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*. 321–330. doi:10.1145/3510457.3513031.

[73] Fabian Wolff, Aurel Bílỳ, Christoph Matheja, Peter Müller, and Alexander J Summers. 2021. Modular specification and verification of closures in Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29. doi:10.1145/3485522.

[74] 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 (TOSEM)* 31, 1 (2021), 1–25. doi:10.1145/3466642.

[75] Zeming Yu, Linhai Song, and Yiying Zhang. 2019. Fearless concurrency? understanding concurrent programming safety in real-world rust software. *arXiv preprint arXiv:1902.01906* (2019). arXiv:arXiv:1908.06849

[76] 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*. 1269–1281. doi:10.1145/3510003.3510164.