# A Study on the Use of Unsafe Mode in Rust Programming Language

**Abbas Alshuraymi** *alsh7875@vandals.uidaho.edu*
*Computer Science Department*
*University of Idaho*
*Moscow, Idaho, 83844, USA*

**Jia Song** *jsong@uidaho.edu*
*Computer Science Department*
*University of Idaho*
*Moscow, Idaho, 83844, USA*

**Abstract**

Rust is a modern systems programming language that prioritizes safety and performance. Its key innovation, the ownership and borrowing system, ensures memory safety by preventing common errors such as dangling pointers, data races, and use-after-free bugs. Rust's type system further enhances reliability by catching logic errors at compile time. The Rust compiler enforces memory safety through its ownership and borrowing system, performing a set of strict checks to guarantee the security and safety of the program. However, specific scenarios necessitate the use of Unsafe Rust, which bypasses some of the safety checks. This is particularly relevant for performance optimizations, interfacing with other languages, and implementing complex data structures. In this paper, we have conducted a literature review on the use of Unsafe Rust, exploring why and how programmers are utilizing it. The result indicates that while unsafe Rust is widely used, it is often encapsulated to minimize risks. However, there are still many vulnerabilities in Rust that are caused by using unsafe Rust. So, this paper also suggests some future research directions to help with the safer use of unsafe Rust.

**Keywords:** Unsafe Rust, Rust Safety, Compiler Check, Rust Programming Language, Unsafe Block.

## 1. INTRODUCTION

Rust is a system programming language designed for performance, safety, and concurrency. Developed by Mozilla Research and first released in 2010, Rust is notable for its strong emphasis on memory safety without relying on a garbage collector (Jespersen et al., 2015). Central to Rust’s design is its ownership system, which ensures memory safety by managing how memory is accessed and freed. This system, combined with borrowing and lifetimes, helps manage references to data, ensuring these references remain valid and preventing data races. Rust also features powerful pattern matching through match expressions, enabling concise and readable handling of complex conditions. The language’s focus on concurrency allows developers to write concurrent programs safely, leveraging the ownership model to avoid data races. Additionally, Rust’s package manager and build system, Cargo, simplifies dependency management and project building.

The Rust compiler performs a set of strict checks to guarantee the security and safety of the program. However, unsafe Rust is a subset of the language that allows developers to bypass some of Rust’s safety guarantees and perform operations that the compiler does not check. This can be necessary in certain situations, such as interfacing with C libraries or system calls, where unsafe code is required to handle the interactions properly. Performance-critical sections of code might also leverage unsafe optimizations unattainable within safe Rust's constraints. Direct memory manipulation is another area where unsafe Rust is beneficial, enabling the implementation of data structures or algorithms needing precise memory control. Additionally, inline assembly, which is used for specific processor instructions or optimizations, often necessitates the use of unsafe code. Finally, creating multiple mutable references to a single piece of data, which is prohibited in safe Rust, can be achieved using unsafe Rust when such aliasing is necessary.

While unsafe Rust provides powerful functionalities, it comes with significant security risks because it bypasses Rust's safety checks. Unsafe code can lead to memory safety violations, such as null pointer dereferencing, buffer overflows, and use-after-free errors. These violations can cause a program to crash or exhibit unpredictable behavior. Unsafe Rust can also introduce data races, which occur when two threads access or modify the same memory location concurrently. Additionally, unsafe operations can result in undefined behavior, where the program's actions become unpredictable and can vary across different platforms or compiler versions. This unpredictability makes debugging extremely difficult and the code harder to maintain. Unsafe code can also introduce security vulnerabilities. Hence, it may allow attackers to exploit flaws and execute arbitrary code.

In this research, our goal is to understand why and how unsafe Rust is used. To answer this question, we first study what unsafe Rust can provide. It shows that unsafe Rust allows developers to perform low-level memory manipulation, interface directly with C libraries, optimize performance-critical sections of code, use inline assembly, and create mutable aliases. These operations bypass Rust's stringent safety checks, providing a level of control and efficiency that can be crucial in system programming, embedded development, and performance-intensive applications. Next, we review recent studies on the use of unsafe Rust. This involves analyzing academic papers, industry reports, and community discussions to understand the motivations, benefits, and issues associated with employing unsafe Rust. The review focused on identifying common patterns, specific use cases, and the experiences of developers who have integrated unsafe Rust into their projects. Later, we focused on 4 larger studies about the unsafe Rust and summarized them. The findings from the literature review show that unsafe Rust is used extensively, but is necessary in certain situations where performance and control are crucial. For example, programmers use unsafe Rust to get detailed control over memory, work with other programming languages, and make critical parts of their code faster. However, using unsafe Rust comes with big trade-offs in terms of safety and security. The findings highlight that while unsafe Rust can provide strong advantages, it needs to be managed carefully to avoid security risks and is usually only used when its benefits are greater than its potential problems. We also proposed some future work that may help with using unsafe Rust safely.

## 2. BACKGROUND

### 2.1 Rust Programming Language Overview

Graydon Hoare, a Mozilla employee, created Rust out of his frustrations with the limitations of languages used in Firefox development. Firefox's core, written in C++, has various memory-related errors – dangling pointers, buffer overflows, data races, use-after-free, and so on (Jespersen et al., 2015). These bugs were difficult to track down and caused crashes or unpredictable behavior. This frustrating experience demonstrated the need for a new approach to build systems-level software, where safety is the top priority, but fine-tuned control for performance is still necessary. Rust's primary goal was to achieve memory safety without the performance costs of traditional garbage collection. It aimed to offer the low-level control found in languages like C/C++ alongside the peace of mind provided by safer, higher-level languages. Additionally, Rust was designed to be a practical programming language, ready to handle the complexities of real-world projects. Its development was informed by lessons learned from a wide range of existing languages (Data Source15, 2023).

Mozilla's 2009 sponsorship of Rust provided resources that helped the rapid development and increased the language's visibility. Rust embraced an open model from the outset (Data

Source13, 2021). Language evolution was guided by a transparent RFC (Request for Comments) process, giving the community a voice in shaping Rust's direction. For example, the foundational decisions for Rust's memory management, syntax, and tooling were forged collaboratively (Data Source14, 2020). In addition, an effort has been made to make Rust easier to learn and use without compromising its core strengths. For instance, comprehensive documentation exists, smoothing the learning curve and making Rust more accessible to developers of all levels. Moreover, it has a friendly community and great tools that make development smoother. Rust's built-in tool, Cargo, handles packages and project tasks for programmers. In addition, Rust's compiler is famous for its exceptionally helpful error messages that not only explain the problem but often guide towards solutions, fostering a strong learning environment (Klabnik & Nichols, 2023).

Rust's innovation lies in its ownership and borrowing system. This compiler-enforced mechanism ensures that at any given moment, a piece of data has either a single, well-defined owner responsible for its cleanup or has been temporarily loaned out for read-only (`&`) or read-write (`&mut`) access. With the compiler rigorously enforcing these rules at compile time, this approach eliminates errors such as dangling pointers, data races, and use-after-free, which are common vulnerabilities in other programming languages, such as C/C++. Rust's dedication to safety and performance carries into the realm of concurrency with its `Send` and `Sync` traits (Klabnik & Nichols, 2023). These traits ensure that data can be safely shared between threads, preventing common concurrency-related bugs.

Rust goes beyond preventing crashes; its rich type system proactively catches potential logic errors at compile time (Klabnik & Nichols, 2023).Expressive traits allow for elegant code organization and reusability. Most importantly, Rust's zero-cost abstractions mean you can use high-level constructs like iterators, pattern matching, and its `Option` and `Result` types for robust error handling without sacrificing performance. Moreover, Rust's robust cross-platform support empowers you to build software for a wide range of operating systems and hardware architectures with confidence. The Rust compiler understands detailed platform specifications and can generate code tailored to specific targets. Its standard library offers both cross-platform functionality and the flexibility to adapt to platform-specific needs. Tools like **rustup** and **Cargo** simplify managing different target environments and build processes (Klabnik & Nichols, 2023). This cross-compatibility means that programmers can write their Rust code once and deploy it across diverse systems, from desktop computers to embedded devices, ultimately expanding the reach of your software and reducing development overhead. Then, in 2018, Rust introduced `async` and `await` to streamline asynchronous code, making Rust ideal for network-heavy and concurrent applications. This feature significantly broadened Rust's appeal to developers.

Overall, Rust lets programmers write safe and easy-to-maintain code without giving up performance. The programming language has had significant success and has ranked as the most loved programming language for six consecutive years on StackOverflow's developer survey (Ho & Protzenko, 2022). In addition, Rust's safety and performance drew the attention of tech giants like Microsoft, Amazon, Discord, and Cloudflare, who adopted it for critical systems (Data Source16, 2022).

## 2.2. Safety Mechanisms

Rust is a modern systems programming language designed to empower developers to build simultaneously fast, reliable, and easy-to-maintain software. It provides the low-level control necessary for critical systems such as operating systems, embedded devices, and Real-time operating systems while offering robust safety features that prevent common errors and streamline development (Klabnik & Nichols, 2023).

### 2.2.1. Ownership in Rust

Rust's safety mechanism targets the root cause of memory-related errors like dangling pointers, double frees, use-after-free, and data races, which are common vulnerabilities in C and C++.

Rust's key innovation is proactively detecting memory-related errors during compilation, where the compiler strictly enforces its ownership and borrowing system. The ownership rule in Rust states that each piece of data has a single owner. When ownership is transferred (e.g., by passing a value to a function), the original owner can no longer use that data. Alternatively, data can be temporarily borrowed by references. References can either be read-only (`&`) or read-write (`&mut`), but the compiler strictly enforces that there can only be one mutable reference at a time. In Rust, variables are non-mutable by default (unless the `mutable` keyword is used), and they are bound to their content (they own it or they have ownership of it) (Blanco-Cuaresma & Bolmont, 2016). When that variable goes out of scope (its lifetime ends), Rust automatically cleans up the value. This ownership rule ensures that every value has a clear owner responsible for cleanup, and references are strictly managed to prevent unexpected data modification (Anderson et al., 2016). Therefore, this system eliminates errors like dangling pointers, data races, double-free, and use-after-free bugs. While this might feel slightly more complex to Rust programmers, the payoff is worth the extra work: successfully compiled Rust code offers a strong guarantee of memory safety.

**Ownership Move.** In Rust, every value has a single owner. When you assign values to a variable, such as`let x = y;`,ownership often moves. This means responsibility for managing and deallocating the data associated with that value transfers from the old owner (`y`) to the new owner (`x`). Moves are the default for most data types because they enhance efficiency and provide clarity about which part of your code is responsible for the data's lifetime.

```
 1  fn main() {
 2   let hello_message = String::from("How are you");
 3  // String created, hello_message owns it
 4
 5  let another_message = hello_message;
 6  //Ownership moves to another_message
 7
 8  // This would now be an error:
 9  // println!("{}", hello_message);
10  // hello_message is no longer valid!
11
12   println!("{}", another_message);
13   //This Works as another_message is now the owner
14  }
15
```

**FIGURE 1:** An example of Ownership Move in Rust.

An example of an ownership move is shown in Figure 1. On line 2, the assignment statement assigns the string "How are you" to variable `hello_message`, which means `hello_message` owns the string. Then, on line 5, the ownership of the string moves to the variable `another_message`. On lines 9 and 12, the code tries to print the string using `hello_message`and `another_message`. However, if the code on line 9 is uncommented, then an error message will be given because `hello_message`no longer owns the string "How are you" and hence can not be used to access it. On the other hand, no compiler error message will be generated when executing the code on line 12 because `another_message` is the owner of the string.

The advantages of ownership move include (1) Avoiding value copies: When Ownership moves, Rust often just transfers a pointer to the data rather than copying the entire contents of the data. This is especially important for larger data structures where copying would be time-consuming and memory-intensive. (2) Enabling optimizations: Since the compiler knows who owns a value and when it's invalidated, it can automatically insert memory deallocation code at the right point. This keeps the code clean while allowing for behind-the-scenes optimizations.

**Ownership Borrowing.** Sometimes, we need temporary access to data without taking permanent ownership. Rust facilitates this through its borrowing system. It uses references `&` for immutable (read-only) access, allowing reading the value without the ability to change it. For modification, `&mut` (mutable reference) must be used, which grants exclusive read and write capabilities. The compiler performs a check on it – if there's a read-only reference, no changes are allowed, and only one mutable reference is allowed at any given time. This strictness is the key to preventing data races and unexpected mutations, making Rust code inherently safe.

Rust's borrowing system also introduces flexibility while guaranteeing safety. It allows temporary access to data without the need to take full ownership. This is ideal for situations where you need to work with information without assuming permanent responsibility for it. By allowing read-only or exclusive mutable access, Rust safeguards data integrity. This means you can have confidence that data remains consistent and reliable, especially when multiple parts of the code interact with it.

**Mutable and Shared References.** Mutable references `(&mut)`grant exclusive read and write access to the underlying data. This is important for safe modification. For example, using a global mutable static variable for the cache introduces potential problems, especially in multi-threaded environments where it prevents data races – the hazard of multiple threads unpredictably modifying the same memory. The shared reference (`&`) allows multiple readers to observe the data simultaneously. It provides a guarantee that the data they point to will remain unchanged. A mutable reference acts as a temporary shield, preventing any conflicting access and letting you change data safely. Shared references (`&`) enable you to safely pass data to functions without fearing the function might change its value.

## 2.3 Unsafe Rust

Rust is a programming language known for its focus on memory safety, zero-cost abstractions, and concurrency control. However, it has a unique feature called unsafe Rust. Unsafe Rust allows developers to intentionally bypass the strict safety guarantees that Rust usually enforces. While Rust's primary goal is to eliminate common programming errors through rigorous compiler checks, there are scenarios where developers might need to bypass some of the language's safety checks, such as when interacting with libraries written in other programming languages. When a block of code or a function is marked as unsafe in Rust, it means that the compiler won't enforce some of its usual safety checks within that scope. By using the `unsafe` keyword, programmers essentially assume full responsibility for the code's safety within that block, placing full responsibility for the code's safety in the programmers' hands. Therefore, the programmer must be absolutely certain of the code's safety and accept the increased risk in exchange for potential performance gains. In this exploration, we will delve into five specific reasons for using unsafe Rust.

**2.3.1. Dereferencing a Raw Pointer:** Unsafe Rust allows developers to perform low-level memory operations, such as dereferencing raw pointers. Raw pointers are essentially memory addresses without any associated metadata or lifetime information, offering direct access to memory locations. While inherently risky, raw pointers can be powerful tools when used with extreme care. Programmers may resort to this when interacting with low-level interfaces, optimizing critical sections of code where the overhead of Rust's safety checks becomes prohibitive, requiring direct manipulation of memory addresses, or interfacing with low-level APIs or external libraries that use raw pointers.

In Rust, raw pointers are denoted by `*const T` (immutable) or `*mut T` (mutable), where `T` is a data type. Unlike references (`&T` and `&mut T`), raw pointers lack the guarantees provided by

Rust’s borrow checker and compiler checks. They can point to arbitrary memory locations, potentially leading to undefined behavior if not handled carefully.

```
1  fn main() {
2    let num = 42;
3    let raw_ptr = &num as *const i32;
4    unsafe {
5      // Danger! This points to invalid memory
6      println!(”Raw␣pointer␣value:␣{}”,*raw_ptr);
7    }
8  }
```

**FIGURE 2:** An example of dereferencing a raw pointer in Rust.

See Figure 2 for an example of dereferencing a raw pointer in Rust. This code demonstrates the creation and usage of a raw pointer in Rust and how unsafe blocks are used to dereference raw pointers. On line 2 of the code, it creates an integer variable `num` with a value of 42. On line 3, `raw_ptr` takes a reference to the variable `num` and`*const i32` casts this reference to a raw pointer of type `*const i32` (a raw pointer to a constant i32). Then, on line 6, the raw pointer `raw_ptr` is dereferenced using `*raw_ptr` to access the value it points to. This must be done in an unsafe block (lines 4-7); otherwise, dereferencing a raw pointer is not allowed in Rust.

Dereferencing a raw pointer poses a significant risk due to their lack of safeguards compared to Rust’s safer alternatives. One major issue is the potential for dereferencing an invalid pointer. An invalid pointer may point to uninitialized memory, deallocated memory, or simply an incorrect location, leading to crashes or unpredictable results upon dereferencing it. Furthermore, working with raw pointers inside unsafe blocks eliminates the protection of Rust’s borrow checker. This can lead to data races, where concurrent access or aliasing (multiple pointers referencing the same memory) can result in unexpected memory modifications and corrupted data. In addition, mistakes in offset calculations or neglecting to release allocated memory can lead to memory that remains in use, even if the program no longer needs it.

### 2.3.2. Calling an Unsafe Function or Method

Rust allows the definition of unsafe functions or methods, which can only be called within an unsafe block. Unsafe functions or methods are the functions or methods marked with the `unsafe` keyword. An unsafe block allows programmers to use unsafe features, such as dereferencing raw pointers or calling unsafe functions, within a clearly defined, restricted section of your code. It provides a way to encapsulate unsafe behavior and limits its scope. It is crucial to remember that the compiler often performs fewer or even no runtime safety checks on code within unsafe functions and blocks. For example, in certain performance-critical situations where Rust’s safety checks introduce unacceptable overhead, unsafe code might be employed to bypass the check (Data Source17, 2023). Therefore, when using an unsafe block, ensuring the safety of the encapsulated operations becomes entirely the programmer’s responsibility.

Any function declared with an `unsafe` keyword must be called from within an unsafe block. This mechanism signals that the programmer is aware of the potential security risks involved. The unsafe keyword acts as a barrier, preventing accidental calls and ensuring that the use of unsafe functions is deliberate and intentional.

```
1   unsafe fn potentially_dangerous_operation
2   (input: &str) -> usize {
3       // Dereferencing a raw pointer
4       //without bounds checks
5       let raw_ptr = input.as_ptr() as *mut u8;
6       let value_at_offset = *(raw_ptr.add(10));
7       // Reading directly from memory... danger!
8       // Converting a reference into a raw pointer
9       let ptr = input.as_ptr();
10      let some_value = *ptr;
11      //'input' might be gone by now
12
13      // Transmuting between incompatible types
14      let bytes = input.as_bytes();
15      let fake_int_ptr = bytes.as_ptr() as *const i32;
16      let fake_int = *fake_int_ptr;
17      //Likely misinterpreting memory
18
19      input.len() * 2  //
20  }
21  fn main() {
22      let my_string = "Hello";
23      unsafe {
24        let result = potentially_dangerous_operation(my_string);
25        println!("Result: {}", result);
26      }
27  }
28
```

**FIGURE 3:** An Example of an Unsafe Function.

Figure 3 shows an example of an unsafe function (lines 1 to 20). On line 1, the `unsafe` keyword indicates that the function contains operations that the programmer does not want the Rust compiler to check their security. This code demonstrates several unsafe operations in Rust, such as dereferencing raw pointers (on line 6 and line 10) and transmuting between incompatible types can lead to misinterpreting memory (on lines 14-16). Because this unsafe function contains potentially dangerous operations, the function can only be called from an unsafe block. In this example, the function is called on line 24, which is inside an unsafe block (lines 23-26).

When an unsafe Rust code is needed, programmers need to decide whether to use unsafe functions or unsafe blocks. Unsafe functions are best for encapsulating unsafe operations where you want to define precise safety requirements. This approach offers modularity and helps guide others who might use the unsafe function by clearly outlining its preconditions. Use unsafe blocks for smaller, isolated snippets of code where creating an entire unsafe function might be unnecessary. When programming, it is important to document any unsafe function, explicitly specifying the conditions required for its safe use.

### 2.3.3. Accessing or Modifying a Mutable Static Variable

Unsafe Rust allows the creation and modification of mutable static variables, which have a single memory location and are accessible across different threads in a program. While generally discouraged due to potential concurrency issues, the use of the global mutable state might be necessary in specific scenarios, such as when designing low-level concurrency patterns or interacting with certain system-level resources. If a mutable static variable must be used, it is imperative to employ appropriate synchronization mechanisms (such as mutexes, locks, or atomic operations) to prevent data races and ensure thread safety.

A static variable in Rust has a static lifetime, meaning it persists for the entire duration of the program's execution. It has a fixed memory location, unlike regular variables that lose their values between function calls. Static variables often serve as global constants or values that need to be shared across different parts of the codebase. When declaring a static variable with the `mut` keyword, it becomes mutable (modifiable). However, it is crucial to exercise extreme caution when working with mutable static variables. They can introduce the potential for data races if multiple threads attempt to modify them simultaneously, leading to undefined behavior, unpredictable program execution, or even crashes. Therefore, careful synchronization and adherence to Rust's concurrency guidelines are essential when dealing with mutable static variables.

Figure 4 illustrates the use of a mutable static variable in Rust. This code demonstrates how a global counter (`OPERATIONS_COUNT`, declared on line 1) can be used to track the number of times the `perform_operation` function is called. On line 5, the global static variable `OPERATIONS_COUNT` is incremented by 1 within an unsafe block (lines 4-6). This is necessary because modifying a mutable static variable is considered an unsafe. Similarly, in the `main` function, another unsafe block (lines 14-17) is used to read and print the value of `OPERATIONS_COUNT`. Again, this is required because accessing a mutable static variable outside of an unsafe block would trigger a compiler error message.

```
 1  //Global Counter
 2  static mut OPERATIONS_COUNT: u32 = 0;
 3  fn perform_operation() {
 4    unsafe {
 5      OPERATIONS_COUNT += 1;
 6    }
 7  }
 8
 9  fn main() {
10    for _ in 0..10 {
11      perform_operation();
12    }
13
14    unsafe {
15      println!("Total_operations_performed:{}",
16      OPERATIONS_COUNT);
17    }
18  }
```

**FIGURE 4:** An example of accessing or modifying a mutable static variable.

While generally discouraged, there are specific situations where altering a static variable within a program might seem beneficial. Examples include caching computationally expensive results, tracking counts across the program, or storing global configuration values that can change at runtime. However, it's important to remember that mutable static variables introduce significant security risks.

### 2.3.4. Implementing an Unsafe Trait

Traits in Rust act like interfaces in other languages, providing a way to define a blueprint of shared behaviors that different types can adopt. They are essential in Rust for several reasons: (1) Polymorphism, a key benefit of traits, allows programmers to write code that works with multiple types without needing to worry about the specifics of their implementations. This flexibility means a function can accept any type that implements a certain trait. (2) Traits also promote code reusability; by defining shared functionality in a trait, programmers can have multiple structs implement those behaviors, eliminating repetition. Furthermore, traits can be implemented for types from external libraries, making it easy to integrate those types within a code. (3) Rust offers unsafe traits that can provide more low-level control or performance optimization. Unsafe traits in Rust enable the definition of traits with associated unsafe methods. This can be valuable when dealing with specialized scenarios requiring manual resource management or other unsafe operations. Programmers may opt for this when working on advanced abstractions that require manual control over certain aspects of the code.

```
1   trait Shape {
2       fn area(&self) -> f64; // Method signature
3   }
4
5   struct Rectangle {
6       width: f64,
7       height: f64,
8   }
9
10  impl Shape for Rectangle {
11  // Implementing the Shape trait for Rectangle
12      fn area(&self) -> f64 {
13          self.width * self.height;
14      }
15  }
16
17  fn main() {
18      let rect = Rectangle {width:5.0, height:10.0};
19      println!("Area␣of␣rectangle:{}", rect.area());
20  }
21
22
```

**FIGURE 5:** An example of trait in Rust.

Figure 5 presents a code example of using trait in Rust. This code demonstrates how to define and implement a trait in Rust. Lines 1-3 define a Shape trait with a method area that must be implemented by any type that wants to implement this trait. Then on lines 5-8, we defined a Rectangle struct with width and height as its fields. Then, the Shape trait is implemented for Rectangle by defining the area method (lines 10-15). This method computes the area by multiplying the rectangle's width and height. In the main function, an instance of Rectangle is created with a width of 5.0 and a height of 10.0 and the area method is called on this instance, and the resulting area is printed to the console, displaying the area of the rectangle.

### 2.3.5. Manual Memory Management

Rust's ownership model is the key to its memory safety guarantees. By default, Rust relies on the concepts of ownership and borrowing. When a variable goes out of scope, its memory is automatically freed (Matsakis & Klock, 2014). This system effectively prevents common memory-related issues like memory leaks and dangling pointers. However, while Rust's automated memory management is excellent for most use cases, there are certain situations where manual control might be necessary. These include interacting with C libraries to manage memory, implementing specialized data structures where fine-grained control over memory layout is needed, or in certain scenarios where the program needs to gain additional efficiency by bypassing Rust's checks. According to Balasubramanian et al., many techniques for improving the performance and reliability of systems hinge on the ability to automatically manipulate program state in memory within unsafe Rust (Balasubramanian et al., 2017).

While Rust's ownership model is sufficient in most cases, sometimes you might need direct control over memory allocation and deallocation on the heap. Rust's standard library provides functions like `alloc`, `alloc zeroed`, and `dealloc`(within the `std::alloc` module) for these tasks. These functions return raw pointers and allow programmers to manage memory explicitly. However, these functions are sometimes too restrictive in terms of managing memory; therefore, the use of unsafe Rust to bypass the checks might be necessary. For example, when designing specialized data structures, interacting directly with hardware, or attempting to improve the performance of a program (Liu et al., 2020). However, it's crucial to emphasize that the programmer assumes full responsibility for ensuring correct and safe manual memory management in these situations.

```
 1
 2  use std::alloc::{alloc, dealloc, Layout};
 3  fn main() {
 4    unsafe {
 5      // Allocate enough space for 5 integers
 6      let layout =Layout::array::<i32>(5).unwrap();
 7      let ptr = alloc(layout) as *mut i32;
 8
 9      //Write values into the allocated memory
10      for i in 0..5 {
11          *ptr.add(i) = i as i32;
12      }
13      // Read the values
14      for i in 0..5 {
15          println!(”Index␣{}:{}”,i,*ptr.add(i));
16      }
17      // Crucial: Explicitly deallocate
18      //to avoid memory leaks
19      dealloc(ptr as *mut u8, layout);
20    }
21  }
```

**FIGURE 6:** An example of Manual Memory Management in Rust.

As shown in Figure 6, an example of manual memory management in Rust is presented. This code demonstrates manual memory management in Rust using the `std::alloc` module (on line 2),allocating memory for an array of 5 i32 integers (lines 6 and 7),writing values into the allocated memory by using the `for` loop on lines 10 to 12. The values are printed from the allocated memory on lines 14-16. On line 19, deallocating the memory to avoid memory leaks.

The use of unsafe blocks (lines 4-20) is necessary for these operations because manual memory management bypasses Rust's safety checks. This code illustrates the importance of handling memory allocation and deallocation carefully to ensure memory safety and prevent leaks.

## 3. RELATED WORK

As Rust gains adoption in critical systems, Qin et al. wanted to understand the real-world safety issues that arise in Rust code to improve programming practices and tools (Qin et al., 2020). Qin et al. conducted the first empirical study of Rust by manually inspecting 850 unsafe code usages and 170 bugs in five open-source Rust projects, five widely used Rust libraries, two online security databases, and the Rust standard library. The study answers three important questions: what memory-safety issues real Rust programs have, what concurrency bugs Rust programmers make, and how and why programmers write unsafe code.

The study explores how unsafe code is utilized, modified, and used. Throughout the examined projects, the authors claim that unsafe Rust is used often and usually for legitimate reasons like improving performance or reusing existing code. However, the authors found that the programmers also make efforts to minimize the use of unsafe code whenever possible. Furthermore, they observed that the practice of interior unsafe is commonly used by programmers as a means to encapsulate unsafe code. In Rust, interior unsafe refers to the practice of using unsafe code within a safe abstraction to perform operations that are not checked by the compiler's usual safety guarantees. This allows developers to encapsulate potentially dangerous code in a way that presents a safe interface to the outside, leveraging the power of unsafe while maintaining overall program safety (Qin et al., 2020).

This study investigates memory-safety issues in real-world Rust applications through bug analysis in selected projects and databases (Qin et al., 2020). The key finding is that all memory-safety bugs involve unsafe code. Surprisingly, most of these bugs also incorporate elements of safe code, highlighting the risk of errors when programmers write safe code without sufficient awareness of connected unsafe code. Additionally, when used with unsafe code, misunderstandings of the Rust lifetimes contribute to many memory-safety issues.

Their study also analyzes concurrency bugs within Rust code, specifically looking at non-blocking and blocking bugs (Qin et al., 2020).Blocking bugs are the bugs that are severe enough to halt the program's execution and stop it from functioning. Non-blocking bugs are the bugs that cause problems or unexpected behavior in a program, but they don't completely prevent the program from continuing to execute. The authors made a surprising discovery: non-blocking bugs can arise in both safe and unsafe Rust code, while all the blocking bugs examined were found exclusively within safe code sections. This finding highlights the potential for concurrency issues even within Rust's typically safer code environments.

In their research, Qin et al. also proposed specific design principles and methods for creating effective Rust bug detectors focused on lifetime issues. They developed two static bug detectors, a use-after-free detector and a double-lock detector, both of which revealed a total of 10 bugs that had not been previously discovered (Qin et al., 2020).

Another study, conducted by Astrauskas et al., delved into real-world applications of unsafe Rust (Astrauskas et al., 2020). The paper offers valuable insights into why programmers choose to employ unsafe Rust. It also provides an empirical analysis of how unsafe Rust appears in actual codebases. In addition, the research aims to verify the Rust hypothesis suggesting that unsafe Rust is used sparingly, is easy to review, and is hidden behind safe interfaces within projects. The study looked at how often unsafe code appears, its size, if it uses other libraries, and what kinds of functions it calls. The authors designed comprehensive methods, including automatic code analysis with information from the Rust compiler and human code reviews, to explore this question. Findings show that while the Rust hypothesis holds partially true—unsafe Rust tends to be uncomplicated and well-contained—it is used extensively in interacting with other programming languages (Astrauskas et al., 2020).

In the third large study we studied, the authors aimed to investigate how software developers are utilizing unsafe Rust in real-world Rust libraries and applications (Evans et al., 2020). Although Rust promotes safety properties like memory safety and no data race, the authors aimed to assess the extent to which these guarantees hold in practice when unsafe Rust is involved. By analyzing the usage of unsafe Rust within Rust libraries, they sought to identify potential risks and challenges to Rust's safety claims, ultimately aiming to recommend changes that could enhance the safety awareness and practices of Rust developers.

The Authors indicate that the use of unsafe Rust may lead to various issues, including(Evans et al., 2020):

- Memory safety issues - Bypassing Rust's memory management system can lead to memory leaks, dangling pointers, and buffer overflows. These can cause crashes, security vulnerabilities, and other unexpected behavior.
- Undefined behavior - Unsafe code can trigger undefined behavior, causing program crashes or unpredictable results. This makes it difficult to reason about the behavior of unsafe code and can lead to hard-to-find bugs.
- Security vulnerabilities - Improper use of unsafe code can introduce security vulnerabilities like buffer overflows, which attackers can exploit.
- Increased development complexity - Using unsafe code makes reasoning about program behavior more challenging, potentially leading to bugs. Unsafe code can also be difficult to test and maintain.

Evans et al. also developed a tool to analyze Rust code and build call graphs that track how safety guarantees might propagate. Safety guarantees in Rust refer to the language's features that help prevent the following issues (Evans et al., 2020): (1) Memory Errors - Errors like dangling pointers, buffer overflows, and memory leaks. These can cause crashes, security vulnerabilities, and other unexpected behavior. (2) Data Races - Issues that occur when multiple threads access the same memory location concurrently without proper synchronization. And (3) Undefined Behavior - The program's behavior is not specified by the Rust language, leading to unpredictable results.

In the context of the study, the authors' tool tracks how these safety guarantees propagate through a program's code. They found that while not every library uses the `unsafe` keyword directly, over half of Rust codebases become indirectly unsafe due to dependencies on unsafe code elsewhere. This suggests that even seemingly safe Rust code may not be fully protected from the bugs. Rust is designed to prevent, highlighting the need for careful use and better tracking of Unsafe Rust (Evans et al., 2020).

In another paper, the authors studied how Rust developers utilize the unsafe code feature and the necessity of its use (Zhang et al., 2023). The authors analyzed thousands of unsafe blocks from popular Rust projects, finding that while unsafe code is often essential (like interacting with foreign languages), there are also many cases where safe alternatives exist that the Rust code could be refactored to eliminate the unsafe keyword while maintaining the same functionality (Zhang et al., 2023). These unnecessary uses often stem from performance optimizations or a lack of awareness of safe Rust constructs. To help developers, the study further identifies common patterns of unnecessary unsafe usage and developed a VS Code plugin that suggests safe alternatives. When the plugin detects unnecessary unsafe code, the plugin directly suggests code transformations that eliminate the need for the unsafe keyword while preserving functionality. By guiding developers towards safer Rust practices, the plugin would contribute to a reduction in memory-related bugs and vulnerabilities in Rust projects. This plugin demonstrates the potential for improving Rust code safety, and evaluations showed that many safe suggestions have minimal performance impact. While some safe alternatives may have slight performance overhead, the impact is generally minimal (Zhang et al., 2023). The effectiveness of the plugin was tested on 140 unsafe blocks from the RustSec Advisory Database, successfully suggesting safe replacements in 28.6% of cases, thus potentially reducing the number of bugs and vulnerabilities associated with unsafe code (Zhang et al., 2023).

## 4. DISCUSSION

Programmers use unsafe Rust primarily due to the need for flexibility and control over low-level operations that are not readily available within the language's strict safety rules. The study conducted on the use of unsafe Rust programs identified several key reasons for this:

1. Performance and flexibility: Unsafe code is often used to optimize performance or to interact directly with hardware or system components where safe abstractions of the language may introduce overhead or be insufficient. For example, unsafe operations might involve direct memory management or bypassing certain compiler checks that enforce safety but can limit direct control over hardware or execution logic.

2. Reuse of existing code: Another common reason for using unsafe code is the need to interface or integrate with existing libraries, particularly those written in other languages like C or C++. This is necessary to leverage existing functionality without rewriting substantial amounts of code in a safe manner.

3. Limitations of safe abstractions: While languages like Rust aim to provide robust safety guarantees, these can sometimes be too restrictive, preventing valid low-level operations necessary for certain system programming tasks. Unsafe code blocks allow developers to perform these operations by opting out of some of the language's safety guarantees.

4. Necessity in systems programming: In system programming, where direct interaction with system resources is frequent, unsafe code is often a necessity. It allows programmers to write highly efficient and low-level code that interacts directly with operating system kernels or hardware devices.

The use of unsafe code, while necessary in many scenarios, does introduce risks, particularly related to memory safety and security vulnerabilities. As such, its use is typically surrounded by strong recommendations to limit the scope of unsafe operations and to ensure that unsafe code is well-reviewed and tested. The potential issues associated with the use of Unsafe Rust are highlighted below:

- **Compiler Guarantees**: When developers use Unsafe Rust, the compiler is unable to guarantee memory safety, which is one of Rust's primary benefits. Without a compiler check, the unsafe Rust code may lead to software vulnerabilities that are difficult to detect.

- **Propagation of Unsafeness:** The study found that even if a small part of a Rust library uses Unsafe Rust, the unsafeness can propagate through the library's call chain. This means that applications depending on such libraries might also be compromised, even if they do not directly use Unsafe Rust.

- **Prevalence in Popular Crates:** Unsafe Rust is more commonly used in popular Rust libraries (crates), which are widely used across the Rust ecosystem. This increases the risk that a large number of applications built on these popular crates may inadvertently include unsafe code.

- **Difficulty in Auditing:** Auditing Rust libraries for Unsafe Rust usage is challenging, especially since it requires a thorough examination of all dependent libraries. This complicates security reviews and can lead to overlooked vulnerabilities.

- **False Sense of Security:** Developers might have a false sense of security due to Rust's inherent safety features, not realizing that the use of unsafe Rust can introduce safety risks. This complacency can lead to less rigorous testing and code review practices.

- **Potential for Memory-Safety Bugs:** While Rust aims to eliminate common memory-safety issues found in languages like C and C++, the use of Unsafe Rust reintroduces the potential for these types of bugs, such as buffer overflows, use-after-free errors, and race conditions.

- **Impact on Rust's Ecosystem:** The widespread use of Unsafe Rust, especially in foundational or heavily used libraries, could have a broader impact on the perceived reliability and safety of the Rust ecosystem as a whole.

### 4.1 Future Directions
Make the unsafe code more visible to the programmers. Unsafe Rust is used in a significant portion of Rust libraries and projects. While less than 30% of Rust libraries explicitly use the unsafe keyword, more than half of all libraries cannot be fully statically checked for safety because they depend on other libraries that use Unsafe Rust (Evans et al., 2020). The propagation of unsafeness through library dependencies challenges Rust's claim as a memory-safe language, as even libraries that do not directly use unsafe can be affected by the unsafe code in their dependencies (Evans et al., 2020). To enhance the visibility and management of Unsafe Rust, one possible solution is that some modifications to the Rust compiler and the central repository interfaces can be applied to better inform developers about the presence and implications of unsafe code in their projects.

Improve the IDE plugins to help programmers find safer alternatives when they use unsafe Rust. Implementation of new plugins and tools that can provide automated suggestions for safe coding practices where feasible. This tool could leverage static analysis techniques to detect patterns where unsafe code could be replaced with safe Rust constructs. For instance, it could identify instances where code unnecessarily uses unsafe for global mutable states and recommend encapsulating such states in thread-safe constructs using Rust's concurrency primitives.

Use advanced code analysis frameworks adoption of advanced code analysis frameworks that could help in encapsulating unsafe code safely and detecting bugs in a proactive manner.

## 4. CONCLUSION

In conclusion, Rust’s growing adoption in systems programming indicates a fundamental shift toward safer and more secure software development. The language’s unique ownership and borrowing system, coupled with a strong type system and efficient abstractions, empowers developers to create reliable and high-performance code. By design, Rust prevents many common errors that often plague systems-level programming in languages like C and C++.

This paper primarily focuses on the use of unsafe Rust. When the `unsafe` keyword is used to define a block of code or a function, the compiler abandons its usual safety checks, which means the programmers are responsible for the safety and security of the unsafe code. The use of unsafe code can help in certain cases, such as improving the performance of the code or interacting with other programming languages. However, the use of unsafe Rust requires careful consideration and thorough review to make sure there are no security issues associated with the unsafe Rust. Studying existing research on the use of unsafe Rust shows that programmers extensively use unsafe Rust, and the unsafeness of such code often goes unnoticed by the programmers due to a large number of unsafe codes existing in the Rust libraries that are inherited by the written program. Based on the identified issues, we suggest some future directions to help with the use of unsafe Rust.

## 5. REFERENCES

Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J., & Sapin, S. (2016). Engineering the servo web browser engine using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 81–89).

Astrauskas, V., Matheja, C., Poli, F., Müller, P., & Summers, A. J. (2020). How do programmers use unsafe Rust? *Proceedings of the ACM on Programming Languages, OOPSLA*, 1–27.

Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., & Ryzhyk, L. (2017). System programming in Rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (pp. 156–161).

Blanco-Cuaresma, S., & Bolmont, E. (2016). What can the programming language Rust do for astrophysics? *Proceedings of the International Astronomical Union, 12*(S325), 341–344.

Data Source13. (2021). Mozilla welcomes the Rust Foundation. Retrieved from https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/

Data Source14. (2020). Laying the foundation for Rust's future. Retrieved from https://blog.rust-lang.org/2020/08/18/laying-the-foundation-for-rusts-future.html

Data Source15. (2023). Rust: World's fastest-growing programming language. Retrieved from https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/

Data Source16. (2022). Sustainability with Rust. Retrieved from https://aws.amazon.com/blogs/opensource/sustainability-with-rust/

Data Source17. (2023). Understand unsafe Rust. Retrieved from https://rustmagazine.org/issue-3/understand-unsafe-rust/

Evans, A. N., Campbell, B., & Soffa, M. L. (2020). Is Rust used safely by software developers? *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 246–257).

Ho, S., & Protzenko, J. (2022). Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages, ICFP*, 711–741.

Jespersen, T. B. L., Munksgaard, P., & Larsen, K. F. (2015). Session types for Rust. *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (pp. 13–22).

Klabnik, S., & Nichols, C. (2023). *The Rust programming language*. No Starch Press.

Liu, P., Zhao, G., & Huang, J. (2020). Securing unsafe Rust programs with XRust. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 234–245).

Matsakis, N. D., & Klock, F. S. (2014). The Rust language. *ACM SIGAda Ada Letters, 34*(3), 103–104.

Qin, B., Chen, Y., Yu, Z., Song, L., & Zhang, Y. (2020). Understanding memory and thread safety practices and issues in real-world Rust programs. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 763–779).

Sam, G., Cameron, N., & Potanin, A. (2017). Automated refactoring of Rust programs. In *Proceedings of the Australasian Computer Science Week Multiconference* (pp. 1–9).

Zhang, Y., Kundu, A., Portokalidis, G., & Xu, J. (2023). On the dual nature of necessity in use of Rust unsafe code. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 2032–2037).