embeddedworld
Exhibition&Conference

C, Rust, C-rusted and MISRA for Safe and Secure Embedded Software

Roberto Bagnara
University of Parma, Italy
Email: name.surname@unipr.it

Nicola Vetrini, Luca Ciucci, Abramo Bagnara
BUGSENG, Italy
Email: name.surname@bugseng.com

Federico Serafini
BUGSENG, Italy
Email: name.surname@bugseng.com
Ca' Foscari University of Venice, Italy
Email: name.surname@unive.it

***Abstract*—C has long been the dominant programming language for embedded systems due to its efficiency, portability, and close-to-hardware capabilities. However, C's low-level memory management and absence of strong safety guarantees expose it to common vulnerabilities such as out-of-bounds accesses, null or invalid pointer dereferencing and memory leaks. To mitigate risks associated with C's flexibility and potential for misuse, the MISRA guidelines became a de facto standard in all sectors where safety and security are crucial. Nonetheless, the embedded systems community, following a trend common to the entire IT world, has been exploring alternatives like Rust. Rust's design inherently reduces the likelihood of common programming errors seen in C, making it an appealing choice for safety- and security-critical embedded software. However, transitioning from C to Rust is not without challenges and hence proposals, such as C-rusted, that can provide a gradual migration path with the same guarantees of Rust but in standard C, are particularly interesting. This presentation features a comparative analysis of C, Rust, C-rusted and the MISRA guidelines (including the potential for a possible MISRA Rust coding standard), with a focus on their implications for embedded software safety and security. We discuss the respective strengths, limitations and use cases, offering insights into how organizations can choose and apply these tools and methodologies based on specific project requirements.**

I. Introduction

The need for a much more secure software infrastructure has become increasingly acute over the years. Vulnerabilities in software continue to pose serious risks, leading to security breaches, financial losses, and threats to human safety, especially in industries such as automotive, aerospace, railways, and medical devices.

Recognizing these challenges, governmental and regulatory bodies have intensified efforts to improve software security. In particular, reports from the U.S. *Cybersecurity and Infrastructure Security Agency* (CISA) [1], [2] and the *White House Office of the National Cyber Director* (ONCD) [2] have accelerated the push towards adopting memory-safe programming languages as a fundamental cybersecurity strategy.

Among memory-safe languages, Rust has emerged as a strong candidate. First introduced 15 years ago, Rust enforces memory safety and concurrency guarantees through its ownership model, preventing many classes of vulnerabilities that have plagued C and C++. However, while Rust is a valuable tool, a full transition from C to Rust is neither practical nor cost-effective in the short term. In fact:

1) The cost of transition: migrating entire legacy systems to Rust would incur prohibitive costs in terms of redevelopment, revalidation, and retraining personnel.
2) Rust is *not free from undefined behavior*: while Rust eliminates many common C vulnerabilities, it still has its own form of undefined behavior, particularly in *unsafe Rust*, which is often required for low-level operations.
3) Many of the tools available for C/C++ development are not (yet) available for Rust. In particular, no Rust compiler exists for many architectures in wide use in embedded systems.

While Rust presents a compelling solution for improving software safety and security, it is important to temper the prevailing enthusiasm with a pragmatic and balanced perspective. The notion that Rust is a universal remedy capable of eliminating all safety and security issues is overly simplistic. Instead, a more measured and strategic approach should be taken — one that acknowledges Rust's strengths while also recognizing the value of incremental enhancements within the existing C ecosystem. Indeed, the current hype around Rust has underestimated at least three factors:

1) The role of the MISRA guidelines: MISRA C, established 25 years ago, provides well-defined safety and security coding guidelines for C. Its long-standing presence and broad adoption across safety-critical industries have allowed organizations to leverage the existing C ecosystem while achieving significant improvements in software safety. The misconception about the MISRA guidelines being only applicable to embedded system — something that is completely false since the publication of MISRA C:2012 [3] — is probably the main cause why their role has been underestimated.
2) Rust needs guidelines as well: besides its cases of undefined behavior, the language has plenty of complex constructs and features that may easily confuse programmers.
3) The case for C-rusted: C-rusted [4], [5] is a natural evolution that enhances the C language by introducing concepts from Rust, such as ownership, exclusivity, and static verification of memory safety properties. This approach enables industries to adopt modern safety features

without discarding their existing investments in C-based software and tooling.

A well-considered transition plan, rather than an outright replacement of legacy systems, is key to ensuring both feasibility and long-term success.

Given these considerations, we believe a pragmatic approach to improving cybersecurity could involve the integrated adoption of MISRA C, C-rusted, and Rust. This paper explores how these technologies can be used together to achieve an incremental and cost-effective transition towards memory safety and stronger cybersecurity. Additionally, we recognize that Rust, like C, requires its own set of coding guidelines for safety-critical applications. Thus, we present a mapping of MISRA C guidelines to Rust, identifying which rules are applicable, in adapted form, and where new guidelines are needed to ensure that Rust meets the safety and security requirements of industries traditionally dominated by C. By adopting a multi-tiered strategy, industries can preserve their existing investments while gradually incorporating memory-safe programming practices, leading to safer and more resilient software infrastructures.

The plan of the paper is as follows: the next section introduces the topic of memory safety and the role of Rust in the current debate, it recalls the pros and cons of the C programming languages as well as the mitigation provided by its MISRA C subset, and it introduces C-rusted; Section III reminds us that Rust too has undefined and erroneous behaviors and that, moreover, it is only defined by its implementation; Section IV provides the elements of a possible Rust coding standard based on MISRA C:2025 showing that around 35% of the MISRA C guidelines may sensibly be adapted to *safe Rust* whereas around 58% of them, suitably modified, are applicable to *unsafe/FFI Rust*; Section V explores some of the synergies between C-rusted and MISRA C showing how the constraints of C-rusted helps in achieving MISRA compliance and the other way around; Section VI discusses the possible integration of MISRA C, C-rusted, and Rust — subsetted along the lines proposed in this paper — in safety-critical industries; Section VII concludes.

Note: Section IV assumes the reader has familiarity with both Rust [6] and MISRA C:2025 [7]: we recommend having a copy of the latter handy while reading that section. The paper has been written so that a reader can skip the more technical part of Section IV without losing comprehension of the rest of the paper.

II. The Quest for Memory-Safe and Efficient Programming Languages

Historically, programmers have always sought for languages and tools that, on the one hand, were able to produce fast and memory-efficient executables, while at the same time being easy to reason about and program in order to adapt to an ever-changing landscape of needs and requests by the industry. These two goals have long been viewed as — and, until recently, they truly were — difficult to reconcile. As a result, the tradeoff has often been resolved by prioritizing efficiency, frequently at the expense of safety and security.

Recently, there has been a strong acceleration towards memory-safe languages. This movement has resulted in two publications that have had, and continue having, a deep impact on the industry at large [1], [2].

The first publication is a report titled "The Case for Memory Safe Roadmaps: Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously" and is a collaborative effort by the U.S. *Cybersecurity and Infrastructure Security Agency* (CISA), *National Security Agency* (NSA), *Federal Bureau of Investigation* (FBI), and the cybersecurity authorities of Australia, Canada, the United Kingdom, and New Zealand [1]. It addresses the pervasive issue of memory safety vulnerabilities in software development and advocates for a strategic shift towards memory-safe programming practices. The report acknowledges that memory safety vulnerabilities constitute a significant portion of disclosed software vulnerabilities, leading to frequent security updates and patches. Despite substantial investments in mitigation efforts, these vulnerabilities persist, imposing ongoing costs on both manufacturers and users. The report thus advocates transitioning to *Memory Safe Programming Languages* (MSLs), as they can effectively eliminate significant classes of memory safety vulnerabilities, reducing the need for continuous mitigation efforts. The idea is that, in the long term, MSLs will yield significant benefits by enhancing product security and reducing associated costs over time. Therefore this is a call to action to manufacturers and senior executives, who are urged to prioritize the adoption of MSLs in their development processes as they could be held responsible for the security impacts of their products on customers. An appendix provides an overview of some of the programming languages that the report considers to be MSLs.

The second influential publication is titled "Back to the Building Blocks: A Path Toward Secure and Measurable Software" and was released in February 2024 by the *White House Office of the National Cyber Director* (ONCD). It outlines strategies to enhance software security and measurability [2]. In particular, it emphasizes the need for systemic changes in software development to reduce vulnerabilities and improve cybersecurity quality by means of the adoption of:

- *Memory Safe Languages* that inherently prevent memory-related vulnerabilities, reducing common security issues.
- *Memory Safe Hardware*, that is, hardware architectures designed to enforce memory safety at the hardware level, thereby complementing software efforts.
- *Formal Methods*, that is, mathematical techniques, such as abstract interpretation [8] and model checking [9], to prove the correctness of algorithms and systems, ensuring they function as intended without vulnerabilities.

The report also highlights the difficulties in quantifying software security and quality, which impede the ability to assess and manage risks effectively: developing reliable metrics can inform stakeholders, guide improvements, and shift market forces towards prioritizing cybersecurity. The report concludes

that, by focusing on these foundational aspects, the software industry can and should move toward a more secure and resilient digital environment.

A. The Success of and Challenges of C

C has long been the go-to language for low-level and systems programming tasks, where performance is as critical as the ergonomics of the programming language. This is largely due to a number of well-known factors:

- it is a relatively simple language, that can be learned at a superficial level in a matter of days by a person already familiar with programming;
- the investments done by many industries led to the growth of an ecosystem of tools encompassing almost any task a programmer could ever think of;
- as a consequence, almost any chip in existence can be programmed in C with the help of a compiler;
- due to its relative closeness to assembly (some say that C is a cross-platform assembly), its abstractions have near-zero cost in most cases, and an expert programmer can estimate the *Worst Case Execution Time* (WCET) of an algorithm written in C just by looking at the source code;
- by relying on the assumed absence of *undefined behavior* in a program, an optimizing compiler can achieve impressive performance with many programs.

However, these advantages come with a cost, which is memory (un)safety: C places most of the burden on the programmer to reliably manage allocated memory, not go beyond the bounds of aggregate objects, initialize automatic storage data before accessing it and many other traps that, even for very experienced programmers, are easy to fall into.

These issues are well-known: indeed, C criticism for the ease with which memory handling programming mistakes are committed dates back to shortly after the language was made available to the public [10].

There have been countless attempts over the years to try to mitigate or sidestep these important concerns. Most of these did not get any traction due to several reasons:

- Some aim to tackle the issue by modifying the language syntax and/or semantics, often requiring programmers to use a new compiler or preprocessor in order to translate such programs into executables; while it could be argued that the new C dialect offers many benefits for safety, it is indeed a different language, not supported by existing tooling. One such example is the *Cyclone* language [11], which requires its own `cyclone` compiler.
- Some employ formal methods in order to prove the correctness of certain pieces of code. Frequently this is done via the definition of a system of annotations for the program often based on first-order logic or related formalisms, with the help of a proof assistant or some static code analysis tool that checks the coherence of the specification with the source code. This approach has the benefit of being able to prove that certain properties hold for a program. It has been shown to be too complicated for programmers to reason about in non-trivial cases and thus undesirable because it reduces development speed and is still prone to human error. One such example is the *ANSI/ISO C Specification Language* (ACSL) [12] and its counterpart ACSL++ for C++.
- Compiler-specific annotations such as `__nullable` and `__nonnull` in LLVM Clang[1] only address specific aspects, such as whether a pointer is expected to be potentially `NULL` or not. However, their implementations can vary between compilers and while they provide some useful additional safety checks, they cannot be blindly trusted when developing safety-critical software. Compiler implementers typically do not provide any formal soundness or completeness guarantees. For instance, what they report might depend in a non-trivial way on the selected optimization level [13].

None of the proposed solutions to improve safety when programming in C, which fall in one or more of the approaches outlined above, while useful, were able to significantly impact the traditional development processes used within the C community. Their usage remains a niche at best, as traditional static or dynamic analysis techniques remain the norm rather than the exception.

In recent years, one of the more compelling proposals for a safer general-purpose programming language that has been a strong candidate to replace C and C++ in some application domains is the Rust programming language.

B. The Emergence and the Promises of Rust

Rust is a programming language designed for safety, concurrency, and performance. It was created by Graydon Hoare, first as a personal project and later officially sponsored by Mozilla. The first public release was in 2012, and its first stable release was in 2015. The language, which is now developed by the *Rust Foundation*, is evolving rapidly, with a new release every six weeks. The key features of Rust are:

Memory safety without garbage collection: Rust enforces strict memory safety rules at compile time. These are based on an *ownership model* with *borrow checking* to manage memory and prevent issues like use-after-free, null pointer dereferencing, and memory leaks (without the need for a garbage collector).

Concurrency without data races: The ownership system ensures that mutable state is accessed safely across multiple threads.

Zero-cost abstractions: Rust provides high-level abstractions which, thanks to compiler optimization, have a performance comparable to C and C++. For instance, high-level iterator abstractions are as fast as and safer than manual array indexing.

Strong typing and pattern matching: Rust has a strongly typed, statically checked type system, which prevents many common programming errors. For example, there

[1] https://clang.llvm.org/docs/analyzer/developer-docs/nullability.html, last accessed on March 7, 2025.

are very few implicit type conversions. The extensive use of pattern matching and enums has the potential of improving code clarity and correctness. Pattern matching allows eliminating long *if-else* chains (and it works with enums, structs, tuples, and even arrays) and ensures exhaustiveness checking — the compiler forces the programmer to handle all cases.

Interoperability with C: Rust supports an *FFI* (*Foreign Function Interface*) for seamless integration with C codebases, which can be used to replace parts of C/C++ projects incrementally.

While syntactically and semantically Rust shares a lot more similarities with multi-paradigm object-oriented programming languages such as C++ and Java, the core model of object lifetime management is radically different and aims to solve many of the issues of memory safety that C and C++ still pose, even with the help of powerful tooling to help the programmer spot the most blatant mistakes.

In the last few years, Rust has seen an ever-growing pool of open-source and proprietary projects adopting its use. Among the best-known open-source projects are the Linux kernel[2] and the Zephyr RTOS.[3]

C. MISRA C: A Safe and Secure Subset of C

MISRA C is a set of coding guidelines for the C programming language, which has been continuously developed by MISRA over the past quarter century to improve software safety, security, reliability, and maintainability, particularly in (but not limited to) embedded systems. MISRA C is widely used in all industries, such as automotive, railways, aerospace and medical devices where safety and security are paramount. The first edition of MISRA C was published in 1998 [14] and the version that is current at the time of publishing this paper is MISRA C:2025 [7].

The MISRA C guidelines restrict unsafe or ambiguous constructs in the C programming language, with a strong focus on safety and security. They prescribe and help eradicate all undefined and unspecified behaviors as well as minimize the risks associated with implementation-defined behavior in C, thereby preventing run-time errors such as buffer overflows, pointer misuse, and data races. Guidelines are categorized as *mandatory*, *required*, and *advisory*, depending on their severity and necessity. For further details on MISRA C we refer the interested reader to the official document [7] and to [15], [16], [17], [18], [19].

D. C-rusted: The Guarantees of Rust in ISO C

C-rusted is an innovative approach that enhances the C programming language by introducing annotations to express ownership, exclusivity, and shareability of resources, as well as dynamic properties of objects and their evolution during execution [4], [5]. These annotations enable static analysis tools to validate code, ensuring that annotated sections are free from a wide range of logical, security, and run-time errors. Importantly, annotated C-rusted code remains compatible with standard ISO C compilers, allowing developers incrementally adopting these safety features without overhauling existing codebases.

[2]https://rust-for-linux.com/, last accessed on March 7, 2025.

[3]https://github.com/zephyrproject-rtos/zephyr-lang-rust, last accessed on March 7, 2025.

```
1  T *find(T *vec, size_t n, T elem) {
2    for (size_t i = 0; i < n; ++i)
3      if (vec[i] == elem)
4        return &vec[i];
5    return NULL;
6  }
7
8  void foo(T * e_opt() vec, size_t n, T elem) {
9    T *ptr = find(vec, n, elem);
10 }
```

Fig. 1: C-rusted: Explicit-only optional pointers prevent null pointer dereference

```
1  void do_things(T * e_hown() p);
2
3  void foo(size_t n) {
4    T *ptr = malloc(n * sizeof(T));
5    if (ptr == NULL) return;
6    do_things(ptr);
7    free(ptr);
8  }
```

Fig. 2: C-rusted: Ownership transfer prevents double free

The following examples illustrate how C-rusted improves memory safety and security while maintaining full compatibility with existing C compilers and toolchains.

1) Handling Null Pointers with Optional Annotations: In C-rusted, pointers in function signatures can be null only if they are annotated as *optional* by means of `e_opt()` annotations. This ensures that null pointers are properly tracked and flagged when misused. Consider the example in Figure 1: here, the *C-rusted Analyzer* reports two warnings:

Line 5: The function `find()` returns `NULL`, but its return type is not explicitly marked as optional.

Line 9: The optional pointer `vec` is passed to `find()`, but `find()` does not expect an optional pointer, introducing a potential null-pointer dereference.

By enforcing proper use of optional pointers, C-rusted helps eliminate null pointer dereferencing errors at compile time, a major source of vulnerabilities in traditional C.

2) Ownership and Memory Management: C-rusted introduces owning references using the `e_hown()` annotation. This allows enforcing strict ownership rules similar to Rust, preventing double frees, memory leaks, and use-after-free errors. The example in Figure 2 illustrates the notion of *ownership transfer*. In this case:

Line 4: Ownership of the allocated memory is transferred from `malloc()` to `ptr`.

Line 6 Ownership is moved to the `do_things()` function, which is responsible for releasing the memory.

```
1  void add2(T *r, const T *a, const T *b) {
2    *r = *a + *b;
3  }
4
5  void add3(T *r, const T *a, const T *b, const T *c) {
6    add2(r, a, b); add2(r, r, c);
7  }
8
9  void do_math(void) {
10   T x = 2, y = 2, z = 2;
11   add3(&z, &x, &y, &z);
12   printf("%d\n", z);
13 }
```

Fig. 3: C-rusted: Implicit shared and exclusive references prevent aliasing issues

```
1  void bar(T *q, size_t n);
2  void foo(size_t n) {
3    T *ptr = calloc(n, sizeof(T));
4    if (ptr == NULL) return;
5    T * e_excl() r = ptr;
6    free(ptr);
7    bar(r, n);
8  }
```

Fig. 4: C-rusted: Exclusive reference prevents use-after-free

Line 7 The *C-rusted Analyzer* detects an error here: `ptr` has lost ownership, and calling `free(ptr)` could cause a double free vulnerability.

By applying systematic ownership tracking, C-rusted prevents memory safety violations without runtime overheads.

3) Preventing Aliasing Issues with Exclusive and Shared References: C-rusted introduces *exclusive* and *shared* references, inspired by Rust's borrowing model. This helps eliminate aliasing issues and ensures that a resource is either *mutably borrowed* (exclusive reference) or *immutably borrowed* (shared reference, read-only). The exclusive or shared nature of a reference can be forced by the programmer using annotations, but it is also inferred by the static analyzer looking at the `restrict` and `const` qualifiers: if an unannotated pointer is restrict-qualified, then it is an exclusive reference; otherwise, if the pointer is not restrict-qualified, then it is a shared reference if the pointee is const-qualified; otherwise, it is an exclusive reference. Consider the example in Figure 3: at first glance, this may seem correct, but calling `add3(&z, &x, &y, &z)` introduces aliasing issues, where `r` is both modified and used as input in `add2()`. The *C-rusted Analyzer* flags this at compile time, at lines 6 and 11, ensuring that potential data races and unexpected behavior are prevented.

4) Usability and Borrowing Rules: C-rusted introduces usability constraints that enforce correct lifetimes of references. The example in Figure 4 demonstrates an incorrect usage pattern. In fact:

Line 5: `ptr` is *paused* because `r` is created as an exclusive reference via the `e_excl()` annotation.

Line 6: There is an error here: `ptr` is used in `free(ptr)` before `r` has finished being used. The *C-rusted Analyzer* correctly flags this, ensuring that `ptr` is only freed when no active references exist.

III. Rust (Un)Safety

The Rust programming language can be divided into two parts: *safe Rust* and *unsafe Rust*, with the former being a subset of the latter. In fact, there are some operations that are permitted only in unsafe Rust: dereferencing raw pointers, call an unsafe function or method, access or modify a mutable static variable, implement an unsafe trait and access fields of a `union`. These operations are necessary in order to perform the following tasks:

- do system-level programming, where low-level details about memory and data representation must be accessible and directly manipulated by the programmer;
- performance optimizations, to avoid run-time checks that can be automatically inserted by the Rust compiler;
- interface with foreign languages, such as C or assembly, used to implement lower layers of operating systems and libraries;
- build abstractions, since all high-level abstraction offered by Rust need to be implemented with efficient and low-level operations.

Unsafe Rust is therefore essential to build real-world applications [20], but it has some drawbacks: while for the safe subset the compiler enforces rules such as ownership and borrow checking to ensure strong memory safety guarantees, for unsafe Rust the correctness is under the programmers responsibility (similar to C and C++). An incorrect use of unsafe code can trigger undefined behaviors that can lead to safety and security issues. This is the reason why `unsafe` blocks should be introduced with care, trying to minimize their number and complexity. For instance, the approach taken by the *Rust for Linux* project, as mentioned in section II-B, is to define Rust wrappers for the kernel C APIs for various subsystems. These wrappers need to be defined using unsafe blocks, but other software that depends on those wrappers, such as a driver for a Wi-Fi card, can be written in safe Rust.

A. Rust Undefined Behavior

While the vast majority of Rust functionality is well-documented, despite the absence of a formal specification for the language, there are some constraints on what a valid Rust program can do and what kind of behaviors are defined, even if erroneous, or undefined. The Rust Reference lists a series of behaviors considered undefined;[4] it is noted in the document itself, however, that such a list is not meant to be exhaustive or fixed in any way: Rust's semantics is not fully defined, therefore some behaviors may be undefined, yet not appear in this list. Such behaviors are:

RUB.01 (Data race) Two or more threads concurrently access a memory location, where at least one access is a write and at least one of them is unsynchronized.

[4] https://doc.rust-lang.org/stable/reference/behavior-considered-undefined.html, last accessed on March 7, 2025.

RUB.02 (Illegal memory access) Accessing (loading from or storing to) a place[5] that is dangling or based on a misaligned pointer.

RUB.03 (In-bounds pointer arithmetic violation) Performing a place projection (field expression, tuple indexing or array/slide indexing) that violates the requirements of in-bounds pointer arithmetic.[6]

RUB.04 (Breaking the pointer aliasing rules) Breaking any `noalias` constraint of the LLVM IR (*Intermediate Representation*) on pointers (akin to C's `restrict` qualifier).

RUB.05 (Mutating immutable bytes) Performing a memory write of one or more bytes that overlap with bytes that are directly or implicitly immutable according to the language semantics.

RUB.06 (Undefined behavior via compiler intrinsics) Invoking undefined behavior via compiler intrinsics.

RUB.07 (Depending on unsupported platform features) Executing code compiled with a platform feature that the current platform does not support, unless this is explicitly documented to be safe.[7]

RUB.08 (Wrong ABI) Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.

RUB.09 (Producing an invalid value) Reading from a place or assigning an invalid value to it (passing a parameter to a function counts as an assignment, similar to MISRA's definition of assignment in C).

RUB.10 (Incorrect use of inline assembly) The Rust Reference [21] has specific guidance on the rules that programmers must follow when writing inline assembly;[8] failing to comply with all the constraints results in undefined behavior.

RUB.11 (Invalid pointer reinterpretation) In a context that is evaluated at compile time, the behavior is undefined if a pointer is reinterpreted or transmuted (e.g., via `std::mem::transmute`) into a non-pointer type, such as an integer.

It is very important to note that, as is the case in C and C++, Rust undefined behavior affects the entire program. For example, if a Rust program calls a C function that exhibits undefined behavior of C, this means that the entire program has undefined behavior, including the Rust code. And the other way around: undefined behavior in Rust can cause adverse affects on code executed by any FFI calls to other languages. It is up to the programmer to ensure that there are no occurrences of undefined behavior in a program, whether or not such undefined behaviors are in unsafe blocks, outside them, or due to the interaction between safe and unsafe code.

B. Erroneous Behavior

Rust gives a precise definition of *unsafe operation*: "those that can potentially violate the memory-safety guarantees of Rust's static semantics" [21]. This implies that behaviors not directly impacting such memory-safety guarantees are not considered to be *unsafe* and, consequently, they do not trigger compile-time errors by the Rust compiler. Nonetheless, there are several behaviors that, while not being *unsafe* in that sense, are typically considered erroneous. These are: deadlocks, memory or resource leaks, exiting without calling destructors, exposing randomized base addresses through pointer leaks, integer overflows, logic errors.[9]

C. Implementation-Defined Behavior

While the expression "implementation-defined behavior" does not occur in the Rust literature, the entire language is implementation-defined, meaning that there exists basically only one implementation of the language, in the form of the `rustc` compiler. Indeed, Rust changed considerably over the years, and it keeps changing with a new release every six weeks. This is a problem, as functional safety standards such as ISO 26262 prescribe the use of safe subsets of standardized programming languages used with qualifiable translation toolchains (see, e.g., ISO 26262 Part 8 and RTCA DO-330). In fact,one of the key challenges in adopting Rust for functional safety applications lies in the rapid pace of its evolution and the monopoly of its implementation. While Rust boasts a strong ownership model that prevents entire classes of memory safety vulnerabilities, it diverges significantly from the principles of standardized and stable languages as required by all functional safety standards.

Unlike C and C++, which have well-established ISO standards, currently there is no formal specification that defines the language independently of its implementation, meaning that the behavior of Rust is effectively dictated by the decisions made in `rustc`'s development. Even Ferrocene, a qualified Rust compiler toolchain, is based on a specific version of the `rustc` compiler with accompanying documentation, additional tests, and a language specification that is supposed to correspond to the language implemented by that specific version of the `rustc` compiler. This lack of an independent and truly stable specification raises concerns in safety-critical domains, where predictability and determinism are paramount.

Rust follows a six-week release cycle, introducing frequent changes and new features that can alter the behavior of previously compiled code. While Rust guarantees backward compatibility for stable features, this rapid iteration presents significant challenges for long-term maintenance, regression testing, and certification:

[5] A place in Rust is defined by a *place expression* (https://doc.rust-lang.org/reference/expressions.html#place-expressions-and-value-expressions, last accessed on March 7, 2025) and roughly represents a memory location in the source code. A concrete representation is `rustc` MIR's `Place` (https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/struct.Place.html, last accessed on March 7, 2025).

[6] https://doc.rust-lang.org/stable/std/primitive.pointer.html, last accessed on March 7, 2025.

[7] https://doc.rust-lang.org/reference/attributes/codegen.html#the-target_feature-attribute, last accessed on March 7, 2025.

[8] https://doc.rust-lang.org/stable/reference/inline-assembly.html, last accessed on March 7, 2025.

[9] https://doc.rust-lang.org/stable/reference/behavior-not-considered-unsafe.html, last accessed on March 7, 2025.

- Functional safety projects require long lifecycles (often spanning decades in automotive, aerospace and railways), whereas Rust's ecosystem is designed around continuous evolution and breaking changes in unstable features.
- Toolchain qualification is an essential part of functional safety compliance (e.g., ISO 26262 Part 8, RTCA DO-330), requiring that compilers be stable, verifiable, and free of unintended changes. Rust's frequent updates make it challenging to qualify the toolchain under such standards.
- Safety-critical software must be predictable, but rapid updates increase the risk of toolchain-induced variability, where seemingly minor compiler updates may affect code generation, optimization strategies, or static analysis results.

If Rust is to be adopted in safety-critical domains, steps must be taken to align it with functional safety requirements, such as:

1) Defining a stable, safety-focused subset of Rust that can be used in long-term projects.
2) Establishing a formal Rust specification independent of its implementation in `rustc`.
3) Providing long-term support (LTS) versions of Rust with extended maintenance guarantees.
4) Developing methods to qualify Rust toolchains under the prescriptions of functional safety standards.

Until these concerns are addressed, Rust's rapid evolution remains a barrier to its direct adoption in functional safety domains, reinforcing the need for alternative, qualified approaches like MISRA C and C-rusted, which provide stable and certifiable pathways to safer software development.

The next section is a first step toward addressing the first of the above concerns.

IV. Elements for a Rust Coding Standard

As we have seen in previous sections, one of the things that Rust misses, if it has to qualify as a stable foundation for safety-critical applications, is a coding standard defining a suitable safety-focused subset. Here we explore the possibility of defining such a subset along the lines of MISRA C/C++, that is, with a coding standard constituted by a set of coding guidelines. Moreover, given that Rust has similarities with C and C++, MISRA C and MISRA C++ are obvious candidates as starting points for a Rust coding standard.

Since Rust is a complex, multi-paradigm programming language with a lot of features, many aspects are at play when looking at specific issues that may arise in a program. This consideration would likely lead to the formulation of guidelines more akin to those seen in MISRA C++:2023 [22], rather than those in MISRA C:2025 [7]. On the other hand, Rust can be interfaced with C++ only in a restricted set of cases via third-party libraries such as `bindgen`.[10] While one could avoid using the STL and manually call constructors and destructors, that approach does not scale well. In contrast, the built-in FFI support for interfacing with C is much better integrated into the language, and likely of greater importance in embedded contexts: this is why, in this paper, we will draw the elements for a possible Rust coding standard from MISRA C:2025.

One of the most important decisions to be made when considering a coding standard for Rust is the treatment of unsafe Rust: since unsafe Rust and the use of FFI are, as we have seen before, key enablers to developing useful real-world software, especially for embedded systems, their handling requires a lot more care, as there are fewer correctness guarantees given by the language. We will thus distinguish between guidelines that are applicable only to unsafe Rust and the FFI from those that are applicable to the entire Rust language.

In the next section we will discuss the use of guidelines inspired by MISRA C:2025 for the prevention of undefined and erroneous behavior, as well as the prevention of confusion on the part of developers and code reviewer. We will then present a synoptic table with a mapping of all MISRA C:2025 guidelines to Rust.

We will use different fonts to render the guideline *category*: the identifiers of *mandatory* guidelines are written in boldface (e.g., **Rule 9.1**), in italics if *advisory* (e.g., *Dir 4.4*), or in normal font if *required* (e.g., Rule 5.3).

A. Coverage of Rust Undefined Behaviors

One of the main objectives of the MISRA C guidelines in is the prevention of the many undefined or critical unspecified behaviors of in C. Here we discuss how selected MISRA C guidelines related to undefined behavior of C can be adapted to avoid undefined behavior of Rust as recalled in Section III-A.

RUB-01 (Data race) Concurrent execution of different parts of a program can lead to a data race. While Rust offers some protection against data races thanks to its ownership system, not all data races can be prevented by compile-time checks, thus MISRA C guidelines dealing with such aspects are partially applicable to safe Rust and fully applicable to unsafe blocks.

RUB-02 (Illegal memory access) This can arise either from incorrect manipulation of misaligned pointers or from dereferencing dangling pointers, which is particularly relevant when interfacing with C code via FFI. For both cases, MISRA C guidelines are applicable.

RUB-03 (In-bounds pointer arithmetic violation) Violating the rules of in-bounds pointer arithmetic can happen when dealing with `unsafe` functions or methods that directly manipulate pointers, such as `pointer::offset`. Therefore, most of the rules from MISRA C Series 18 dealing with pointers are applicable to unsafe Rust.

RUB-04 (Breaking the pointer aliasing rules) Rust and C have different viewpoints with respect to aliasing. In particular, in C aliasing is seen as an opt-out feature via the `restrict` qualifier, while in Rust aliasing can be realized via `UnsafeCell<U>` via the pattern known

[10] https://rust-lang.github.io/rust-bindgen/cpp.html, last accessed on March 7, 2025.

as *interior mutability*. As a result, while MISRA C discourages the use of `restrict` (Rule 8.14), there is no rule other than Rule 1.3 (There shall be no occurrence of undefined or critical unspecified behaviour) that directly addresses RUB.04, therefore a coding standard for Rust will need to consider this aspect.

RUB-05 (Mutating immutable bytes) While Rust's borrow checker and immutability by default greatly help in reducing the likelihood of this undefined behavior, care must still be taken when interfacing with C code that may implicitly require the pointed to data not to be modified despite the lack of constness. It is also possible in Rust to convert from a `const` pointer to a `mut` pointer, thereby allowing such behavior to take place.

RUB-06 (Undefined behavior via compiler intrinsics) As compiler intrinsics are language extensions, prevention corresponds to the enforcement of guidelines similar to MISRA C *Dir 1.2*.

RUB-07 (Depending on unsupported platform features) Features only available on certain platforms are an implementation-defined aspect, therefore their prevention corresponds to the enforcement of MISRA C Dir 1.1.

RUB-08 (Wrong ABI) Determining which ABI should be used when calling a function defined in foreign code is an implementation-defined aspect, therefore enforcing MISRA C Dir 1.1 ensures that appropriate measures are taken to avoid this behavior.

RUB-09 (Producing an invalid value) While MISRA C Rule 1.1 explicitly mentions the syntax and constraints of the C language, therefore making the rule still applicable to Rust when interfacing with C code, an equivalent guideline can be useful for safe Rust, as invalid values can be produced.

RUB-10 (Incorrect use of inline assembly) Usage of inline assembly is regulated by MISRA C directives prescribing proper encapsulation and documentation, as the exact constraints are an implementation-defined aspect. These directives are equally valid for Rust.

RUB-11 (Invalid pointer reinterpretation) Rules like those of Series 11 of MISRA C, such as Rule 11.4 (A conversion should not be performed between a pointer to object and an integer type), can be used to prevent most instances of invalid pointer reinterpretation. For the remaining instances, the use of unions can be limited via the enforcement of guidelines like MISRA C *Rule 19.2* (The `union` keyword should not be used) and *Rule 19.3* (A union member shall not be read unless it has been previously set).

B. Coverage of Rust Erroneous Behaviors

Many of the Rust erroneous behaviors recalled in Section III-B may arise in C programs as well and can be mitigated by slight adaptations of existing MISRA C guidelines.

1) Deadlocks: MISRA C addresses deadlocks through Dir 5.2 (There shall be no deadlocks between threads). Being a directive, compliance is not just a matter of modifying the code to being compliant, but also promotes establishing proper documentation that outlines strategies to prevent deadlocks (e.g., a global ordering of resources). These strategies can then be validated with the help of a static analysis tool.

2) Memory and resource leaks: Resource leaks are covered by *Dir 4.13* (Functions which are designed to provide operations on a resource should be called in an appropriate sequence). In Rust this also requires that destructors are correctly implemented and triggered. Additionally, Dir 4.12 (Dynamic memory allocation shall not be used) and Rule 21.3 (The memory allocation and deallocation functions of `<stdlib.h>` shall not be used) discourages the use of manual heap management.

3) Exiting without calling destructors: Not triggering destructors is a potential cause of resource leaks and thus this case is also partially covered by *Dir 4.13*.

4) Integer overflows: This is addressed by MISRA C under Dir 4.1 (Run-time failures shall be minimized). Rust provides some built-in protections through debug assertions in non-optimized builds[11], but explicit enforcement of overflow handling remains the responsibility of the programmer.

5) Logic errors: While Dir 4.1 also partially addresses logic errors by requiring run-time failures to be minimized, general logical correctness is beyond what MISRA C and the Rust compiler can enforce. For instance, as noted by the Rust Reference,[12] inappropriate trait implementations can introduce erroneous or unpredictable behavior.

6) Exposing randomized base addresses through pointer leaks: Exposing randomized addresses is not covered by any MISRA C guideline, though Rule 11.4 (A conversion should not be performed between a pointer to object and an integer type) prevents some instances of pointer reinterpretation that could lead to such situations.

C. Applicability of the MISRA C Essential Type System

The *Essential Type System* of MISRA C addresses the pitfalls and dark corners of C's rather weak type system, whereby implicit conversions can easily fool developers and reviewers.

Rust possesses a much stronger type system, inspired by those of functional and object-oriented programming languages, which mitigates by default many of the issues that the MISRA C guidelines based on the essential type system aim to address. More specifically, conversions in Rust can either occur explicitly (via the `as` operator and the unsafe `std::mem::transmute` function) or implicitly via *coercions*, which are a set of rules that apply at specific coercion sites,[13] and do not relate with MISRA C essential types.

In general, Rust's type system imposes strict constraints on the allowed explicit conversions and specifies what kind

[11] https://doc.rust-lang.org/stable/reference/expressions/operator-expr.html#overflow, last accessed on March 7, 2025.

[12] https://doc.rust-lang.org/reference/behavior-not-considered-unsafe.html#logic-errors, last accessed on March 7, 2025.

[13] https://doc.rust-lang.org/stable/reference/type-coercions.html#coercion-sites, last accessed on March 7, 2025.

```rust
// compile error: cannot add `u32` to `i32`
let _ = 1i32 + 3u32;

// Rule 10.5 violation
assert_eq!(-1i8 as u8, 255u8);
assert_eq!(1234u16 as u8, 210u8);
assert_eq!(42.9f32 as i32, 42);

// Rule 10.5 violation
let c: char = unsafe {
    std::mem::transmute(u32::MAX)
};

#[repr(u8)]
enum Enum { A, B }

// Rule 10.5 violation (leads to UB)
let e: Enum = unsafe {
    std::mem::transmute(5u8)
};
```

Fig. 5: Explicit conversions between arithmetic types that can lead to unexpected or undefined behavior

of transformations are applied to the values.[14] This avoids many pitfalls addressed by the MISRA C rules in Series 10. However, the allowed explicit conversions are subject to possibly erroneous manipulations (e.g., unintended truncation and casting invalid Unicode scalar values[15], that is, integer values outside the ranges $[0x0000, 0xD7FF]$ and $[0xE000, 0x10FFFF]$ to `char`), and thus other some of the rules in that series are applicable to Rust. For example, *Rule 10.5* (The value of an expression should not be cast to an inappropriate essential type) and Rule 10.8 (The value of a composite expression shall not be cast to a different essential type category or a wider essential type) are applicable to the allowed `as` conversions and to the unsafe `std::mem::transmute` since they could lead to unexpected or undefined behavior: a few representative situations are shown in Figure 5.

D. Coverage of Developer Confusion

Rust's strong type system, combined with its syntax and semantics, provide protection against some common programming patterns that often lead to developer confusion in C. However, many aspects enforced by the MISRA guidelines remain relevant to Rust.

Rust supports nested block comments, which mitigates some of the risks associated with comment misuse in C. However, guidelines such as *Dir 4.4* (Sections of code should not be "commented out") and Rule 3.1 (The character sequences `/*` and `//` shall not be used within a comment) remain relevant. In fact, misplaced or improperly nested comments may still lead to unintended code inclusion or exclusion in Rust, which these rules will prevent.

[14] https://doc.rust-lang.org/reference/expressions/operator-expr.html#type-cast-expressions, last accessed on March 7, 2025.

[15] https://www.unicode.org/glossary/#unicode_scalar_value, last accessed on March 7, 2025.

Typographical issues remain a potential source of confusion, particularly when identifiers and literals appear similar: *Dir 4.5* (Identifiers in the same name space with overlapping visibility should be typographically unambiguous) helps mitigating these risks by enforcing a clear distinction between identifiers.

A related concern is *shadowing*, where an identifier is redefined within the same or an inner block, which makes the outer definition not referenceable. This can lead to developer confusion because someone may write code in an inner block on the assumption that the outer definition is being used, likely leading to the introduction of subtle defects. As Rust is subject to the same issue, Rule 5.3 is clearly applicable (An identifier declared in an inner scope shall not hide an identifier declared in an outer scope).

Rust implements floating-point arithmetic according to the IEEE-754 standard [23], thus guidelines concerning floating-point arithmetic such as Dir 4.15 (Evaluation of floating-point expressions shall not lead to the undetected generation of infinities and NaNs) can be applied to Rust, thereby reducing chances of errors and loss of precision in numerical computations.

Another area of concern is the management of resources. Many Rust APIs use the type system to enforce proper sequencing of resource allocations and deallocations via the RAII idiom. *Dir 4.13* (Functions which are designed to provide operations on a resource should be called in an appropriate sequence) is still applicable, since it is possible to use unsafe functions (via the `core` crate or FFI) or omitting some calls to destructors; in combination with the RAII idiom, these may cause missed deallocations. Moreover, *Dir 4.13* regards all sorts of operations on all kinds of resources, including those that, being completely user-defined, cannot be controlled by the Rust compiler. Additionally, many MISRA C guidelines from Series 21 and 22, concerning standard libraries and resources, respectively, are directly applicable in the context of Rust's FFI capabilities.

Macros are also a source of potential confusion. While Rust possesses a more structured macro system compared to C, many aspects covered by the MISRA rules remain open. For example *Dir 4.9* (A function should be used in preference to a function-like macro where they are interchangeable) discourages the use of function-like macros because they do not perform type checks, they could potentially have multiple side effects, unnecessary multiple expression evaluation and debugging intricacies. Similar to C, the behavior of macros is also context-dependent despite their "partial hygiene" [24].

Rust provides *lints* to detect unused or unreachable code (e.g. the `dead_code` and `unused_must_use`) but these are not strictly enforced by the language. MISRA rules addressing dead code (Rule 2.2), error handling (Dir 4.7), and function return value usage (Rule 17.7) are all fully applicable to safe and unsafe Rust.

Finally, several MISRA C naming rules concerning declarations and pointer qualifiers, particularly those concerning external linkage and FFI interactions, are applicable to Rust with only minor changes to align them with Rust's semantics

while preserving their original intent: an example is *Rule 8.7* (Functions and objects should not be defined with external linkage if they are referenced in only one translation unit).

E. A Mapping of MISRA C:2025 Guidelines to Rust

This section summarizes the work done at BUGSENG and in the *MISRA C Working Group* on the applicability of the MISRA C:2025 guidelines [7] to Rust. Work on this mapping started independently at BUGSENG in mid 2024 as part of the work on C-rusted [4], [5]. At the beginning of 2025 forces were joined with the *MISRA C Working Group*, which has been conducting parallel work along the same lines. This joint work culminated in the publication of MISRA C:2025 Addendum 6 [25] and of the tables presented in this section, namely: Table I, for the mapping itself, and Table II, for some statistics on the mapping.

All efforts were made to avoid contradiction in the material that is common between Table I and [25]. Nonetheless, the reader should take into account that:

1) The additional fields and the different notes in Table I are intentional and reflect the needs of this paper and the personal views of its authors.
2) The official version of the mapping is the one given in MISRA C:2025 Addendum 6 [25] and not the one given in the present paper.

Table I has the following columns:

Guideline: The MISRA C guideline identifier.

Rationale A summary of the rationale for the MISRA C guideline. While for the official text we refer the reader to MISRA C:2025 Addendum 6 [25], here is a concise summary of the meaning of the acronyms used in the table:

- **UB:** The MISRA C guideline aims to prevent the insurgence of one or more C *Undefined* or *Unspecified Behaviors*.
- **IDB:** The MISRA C guideline is concerned with one or more C *Implementation-Defined Behaviors*.
- **CQ:** The MISRA C guideline is concerned with *Code Quality* [26].
- **DC:** The MISRA C guideline aims at preventing possible *Developer Confusion*.

Many guidelines have multiple rationales: this is a consequence of MISRA intent of maintaining a reasonable number of guidelines that are as simple as possible.

Applicability: A guideline might be either fully applicable to Rust (`y`), partly applicable (`p`) if only part of the rationale for the MISRA C guideline is applicable to Rust, or not applicable (`n`). Furthermore, the applicability field may differ between safe and unsafe Rust due to the fact that many guarantees about memory safety that are enforced in the safe subset of Rust are under the programmer's responsibility in `unsafe` blocks. Given that uses of the C FFI must be made inside an `unsafe` block, and many MISRA C guidelines are directed at preventing misuse of C Standard Library functions, the applicability classification has been conceived by taking into account the use of those functions via FFI.

Rust UB: This column contains the Rust undefined behaviors, as described in Section III-A, which application of the MISRA C guideline helps preventing. Note that Rule 1.3 (There shall be no occurrence of undefined or critical unspecified behaviour) is interpreted, as in MISRA C, as a catch-all rule that encompasses all undefined or critical unspecified behaviors not explicitly mentioned in other guidelines. As such, it has been mapped to all Rust UBs.

Notes: These are similar in spirit, but often more detailed than than the *Comment* given in MISRA C:2025 Addendum 6, and their main purpose is to explain the reasoning behind the classification of a guideline as applicable, not applicable or partially applicable to Rust. While some of these notes might deserve a more comprehensive explanation, we strove to find a balance between conciseness, completeness, and understandability.

V. C-rusted and MISRA C

There are important synergies between C-rusted and MISRA C. On the one hand, to deliver its own guarantees C-rusted partly relies on some of the guarantees provided by MISRA C, such as the absence of pointer conversions whose behavior is not fully defined. On the other hand, the safety guarantees provided by C-rusted result in compliance with, or justified deviation from, important — and, in some cases, challenging — MISRA C guidelines.

Before going into these topics, note that C-rusted, like Rust, has *unsafe blocks*: this is a portion of code for which correctness is under the programmer responsibility and upon which the *C-rusted Analyzer* is not required to give any warnings. Nonetheless, as C-rusted code is entirely within standard ISO C, code in unsafe blocks can and should be checked for MISRA compliance.

A. C-rusted for MISRA C

1) Dynamic Memory Allocation: Use of dynamic memory allocation (however implemented) is discouraged by MISRA C Dir 4.12, while memory allocation and deallocation functions of `<stdlib.h>` are explicitly targeted by Rule 21.3. Projects that deviate those in order to use dynamic memory allocation from `<stdlib.h>` are still subject to Rule 22.1 (All resources obtained dynamically by means of Standard Library functions shall be explicitly released) and **Rule 22.2** (A block of memory shall only be freed if it was allocated by means of a Standard Library function).

(continued after Tables I and II)

TABLE I: Applicability of MISRA C:2025 Guidelines to Rust

Guideline	Rationale	Applicability unsafe/FFI	Applicability safe	Rust UB	Notes
				RUB-07	
Dir 1.1	IDB	y	y	RUB-08 RUB-10	As noted in section III-C, given the absence of a full specification and a single main implementation of the language, a lot of aspects are implementation-defined.
Dir 1.2	IDB	y	y	RUB-06	Any usage of experimental or unstable features via the `#[target_feature(...)]` macro should be documented, as well as compiler intrinsics.
Dir 2.1	UB, CQ, DC	y	y		
Dir 3.1	CQ	y	y		Requirements traceability tools for Rust are not as mature as those for C.
Dir 4.1	UB, CQ	y	y	RUB-01	Run-time failures in Rust are often in the form of panics.
Dir 4.2	IDB, CQ	y	n	RUB-10	
Dir 4.3	DC, CQ	y	n	RUB-10	
Dir 4.4	DC	y	y		The `#[cfg(...)]` attribute allows for conditional compilation, as a counterpart to C preprocessor directives `#if` and `#ifdef`.
Dir 4.5	DC	y	y		
Dir 4.6	DC	n	n		Primitive types already fulfill this requirement.
Dir 4.7	DC	y	y		Recoverable errors in Rust should use the `Option`, `Result` types.
Dir 4.8	DC	n	n		Robust encapsulation and information hiding in Rust is accomplished via the module system.
Dir 4.9	DC, CQ	y	y		Many of the considerations for C macros are true for both forms of Rust macros, and especially for procedural macros [24].
Dir 4.10	UB, DC	n	n		Rust does not have header files.
Dir 4.11	UB, IDB	y	y		This directive applies to all libraries.
Dir 4.12	UB, CQ	y	y		
Dir 4.13	UB, DC	y	y		Many APIs from the Rust Standard Library use the type system to enforce ordering; nevertheless, this applies to all libraries.

continued on next page

continued from previous page

Guideline	Rationale	Applicability unsafe/FFI	Applicability safe	Rust UB	Notes
Dir 4.14	UB, CQ	y	y		
Dir 4.15	UB, IDB, DC	y	y		Rust implements IEEE-754 [23].
Dir 5.1	UB	y	p	RUB-01	Not all safe Rust types are race-free. A notable exceptions is `std::rc::Rc`. See section *Send and Sync* of the Rustonomicon [27].
Dir 5.2	UB	y	y		In section IV-B an overview of the issues and remediations against deadlock in Rust is given.
Dir 5.3	UB, DC	y	y		
Rule 1.1	UB, IDB	y	p	RUB-09 RUB-01 RUB-02 RUB-03 RUB-04 RUB-05	See the discussion at III-C.
Rule 1.3	UB, IDB	y	y	RUB-06 RUB-07 RUB-08 RUB-09 RUB-10 RUB-11	Rust has a list of behaviors considered undefined, but any occurrence of Undefined Behavior not covered by other rules is covered by this one.
Rule 1.4	UB, DC	n	n		Specific to C versioning.
Rule 1.5	UB, IDB, DC	y	y		Applies to deprecated APIs, marked with the `#[deprecated(...)]` attribute.
Rule 2.1	DC	y	y		The `rustc` compiler and other standard Rust tooling have partial support to detect unused code.
Rule 2.2	DC	y	y		The `rustc` compiler and other standard Rust tooling have partial support to detect dead code.
Rule 2.3	DC	y	y		The `rustc` compiler and other standard Rust tooling have partial support to detect unused code.
Rule 2.4	DC	n	n		Rust does not have a separate tag name space, unlike C, therefore this guideline does not apply to Rust.
Rule 2.5	DC	y	y		The `rustc` compiler and other standard Rust tooling have partial support to detect unused code.
Rule 2.6	DC	y	y		The `rustc` compiler and other standard Rust tooling have partial support to detect unused code.
Rule 2.7	DC	y	y		The `rustc` compiler and other standard Rust tooling have partial support to detect unused code.
Rule 2.8	DC	y	y		The `rustc` compiler and other standard Rust tooling have partial support to detect unused code.
Rule 3.1	DC	y	y		Nested block comments are fully supported, but mixing them with line comments might still be a source of developer confusion.
Rule 3.2	DC	n	n		Rust does not have line splicing.
Rule 4.1	DC, IDB	n	n		
Rule 4.2	DC	n	n		Rust does not have trigraphs.
Rule 5.1	UB, IDB, DC	y	p		Rust does not impose limits on the number of significant characters, except in `extern "C"` blocks, but long identifiers harm readability.
Rule 5.2	UB, IDB, CQ	y	y		Rust does not impose limits on the number of significant characters, except in `extern "C"` blocks, but long identifiers harm readability.
Rule 5.3	DC	y	y		In Rust, the rule should also apply to macro identifiers.
Rule 5.4	UB, IDB, DC	n	n		Can be handled by extending Rule 5.3 to also encompass macro identifiers.
Rule 5.5	UB, IDB, DC	p	p		In Rust macros and functions use different syntax, but it is still good practice to avoid reusing the same identifier.

continued on next page

continued from previous page

Guideline	Rationale	Applicability unsafe/FFI	Applicability safe	Rust UB	Notes
Rule 5.6	DC	y	y		The Rust module system makes name conflicts less likely, but this can still be applied to type aliases.
Rule 5.7	UB, DC	n	n		Rust does not have a separate tag name space.
Rule 5.8	DC	y	y		While the Rust module system makes name conflicts less likely, applying the rule still helps avoid confusion.
Rule 5.9	DC	y	y		
Rule 5.10	UB, DC	y	p		Applicable when interfacing with C. Weak keywords can be used as identifiers, but they are discouraged.
Rule 6.1	UB, IDB	n	n		Bit-fields are not part of the language and only provided by external libraries.
Rule 6.2	DC	n	n		Bit-fields are not part of the language and only provided by external libraries.
Rule 6.3	IDB	n	n		Bit-fields are not part of the language and only provided by external libraries.
Rule 7.1	DC	y	y		Rust octals have a distinct prefix from decimals.
Rule 7.2	DC	y	y		This is not a suffix allowed by default but can be enabled. Note that standard Rust numeric literals can have suffixes indicating size and signedness.
Rule 7.3	DC	n	n		Dealt with by standard Rust suffixes with explicit size.
Rule 7.4	UB	n	n		Rust string literals have immutable type `&str`.
Rule 7.5	UB	n	n		Rust does not have these type of macros, because numeric literals can have appropriate suffixes.
Rule 7.6	DC	n	n		Rust does not have these types of macros, because numeric literals can have appropriate suffixes.
Rule 8.1	DC	n	n		Rust promotes type inference, but has no dangerous implicit conversions.
Rule 8.2	UB, DC	n	n		Rust syntax avoids the issue.
Rule 8.3	UB, DC	y	n		An `extern "C"` declaration shall have a type compatible with the C declaration.
Rule 8.4	UB	n	n		
Rule 8.5	DC	y	n		May affect `extern "C"` declarations.
Rule 8.6	UB	y	n		May affect `extern "C"` declarations.
Rule 8.7	DC	y	y		Rust handles visibility of data types and functions via its module system; using `pub` on items referenced in only one module should be avoided.
Rule 8.8	DC	n	n		
Rule 8.9	DC	y	y		
Rule 8.10	UB, DC	n	n		Rust syntax avoids the issue.
Rule 8.11	DC	n	n		Rust syntax requires array sizes to be known at compile time, while slices have a size that can be queried at runtime.
Rule 8.12	DC	n	n		
Rule 8.13	DC	y	y		`mut` should be avoided unless necessary.
Rule 8.14	UB	n	n		Rust has no equivalent to the `restrict` qualifier in C.
Rule 8.15	UB	y	n	RUB-02	May affect `extern "C"` declarations.
Rule 8.16	DC	n	n		Zero-sized types have implicitly alignment 0, but the attribute cannot be explicitly specified.
Rule 8.17	DC	p	p		Rust has attributes to specify the aligment of a type, rather than individual objects, therefore the rule is partially applicable. The attribute syntax allows multiple alignment attributes on a type.

continued on next page

continued from previous page

Guideline	Rationale	Applicability unsafe/FFI	Applicability safe	Rust UB	Notes
Rule 8.18	UB, DC	n	n		
Rule 8.19	UB, DC	n	n		
Rule 9.1	UB	y	y		For safe Rust, this is prevented by the compiler, but it is still applicable; for `unsafe` it can be possible to access uninitialized data.
Rule 9.2	UB, CQ, DC	n	n		Rust syntax avoids the issue.
Rule 9.3	UB	n	n		Rust syntax avoids the issue.
Rule 9.4	DC	y	y		The constraint is enforced by the compiler, but it is still applicable.
Rule 9.5	IDB, DC	n	n		Rust does not have designated initializers.
Rule 9.6	DC	n	n		Rust does not have chained designators.
Rule 9.7	UB	y	n	RUB-01	Can cause Undefined Behavior when interfacing with C,
Rule 10.1	UB, IDB, DC	n	n		See the discussion at section IV-C.
Rule 10.2	DC	n	n		See the discussion at section IV-C.
Rule 10.3	UB, IDB	n	n		See the discussion at section IV-C.
Rule 10.4	IDB	n	n		See the discussion at section IV-C.
Rule 10.5	DC	y	p		See the discussion at section IV-C.
Rule 10.6	DC	n	n		See the discussion at section IV-C.
Rule 10.7	DC	n	n		See the discussion at section IV-C.
Rule 10.8	DC	y	p		See the discussion at section IV-C.
Rule 11.1	UB, IDB	y	p		Converting a function pointer to a raw pointer is safe via the `as` operation, but there exists also the unsafe `std::mem::transmute`.
Rule 11.2	UB	y	n	RUB-02	Relevant when interfacing with C code, as well as transmuting pointers to traits, generics or enums.
Rule 11.3	UB	y	y	RUB-02	In general, when converting between pointers in Rust extreme care must be taken.
Rule 11.4	UB, IDB	y	y	RUB-02	In general, when converting between pointers and addresses in Rust extreme care must be taken.
Rule 11.5	UB	y	n	RUB-02	Relevant when interfacing with C code.
Rule 11.6	UB, IDB	y	n		Relevant when interfacing with C code.
Rule 11.8	UB	y	p	RUB-05	Rust does not have the `_Atomic` or `volatile` qualifiers, but a `const` raw pointer can be converted to a `mut` pointer.
Rule 11.9	DC	n	n		Rust does not have a null pointer constant, though there are ways to obtain pointers with address 0.
Rule 11.10	UB	n	n		
Rule 11.11	DC	y	y		This constraint is enforced by the compiler as it results in a type error, but nevertheless the constraint is applicable.
Rule 12.1	DC	y	y		Lower priority, can be subjected to recategorization.
Rule 12.2	UB, DC	y	p		The behavior in Rust is fully defined. Depending on how the code is compiled, if overflow occurs at runtime it can either call `panic` or wrap around according to two's complement semantics.
Rule 12.3	DC	n	n		Rust does not have a comma operator.
Rule 12.4	DC	y	n		Rust by default does not allow constant expressions to wrap around, but even if the default behavior is changed via `#[allow(arithmetic_overflow)]`, the resulting operation is still defined.

continued on next page

continued from previous page

Guideline	Rationale	Applicability unsafe/FFI	Applicability safe	Rust UB	Notes
Rule 12.5	DC	n	n		
Rule 12.6	UB	n	n		
Rule 13.1	UB	y	y		
Rule 13.2	UB	n	n		Rust has a stricter order of evaluation than C, which prevents the issue.
Rule 13.3	UB, DC	n	n		Rust does not have unary increment or decrement operators.
Rule 13.4	UB, DC	n	n		In Rust the return value of the assignment operator has type `unit`, therefore its result cannot be used inside an arithmetic expression. Addiditonaly, the order of evaluation is strict in Rust.
Rule 13.5	DC	y	y		
Rule 13.6	UB, DC	n	n		Rust has `mem::size_of` for statically sized types and `mem::sizeof_val` for dyamically sized types, therefore the operator is not applied to expressions, but rather to types.
Rule 14.1	DC	y	p		Only applicable to `while` loops. In general Rust promotes the use of iterators for determinate iteration (e.g., `for ... in`).
Rule 14.2	DC	n	n		Rust has the `for ... in` construct for determinate iteration.
Rule 14.3	DC	y	y		
Rule 14.4	DC	y	y		The compiler enforces this constraint; nevertheless, the constraint is applicable.
Rule 15.1	DC	n	n		Rust does not have `goto` statements.
Rule 15.2	DC	n	n		Rust does not have `goto` statements.
Rule 15.3	DC	n	n		Rust does not have `goto` statements.
Rule 15.4	DC	y	y		Only the part concerning the `break` statement is applicable, but the rationale is still fully applicable.
Rule 15.5	DC	y	y		Disapplied in MISRA C:2025, but the rationale is still valid.
Rule 15.6	DC	n	n		Rust does not allow non-compound expressions in this context.
Rule 15.7	DC	y	y		
Rule 16.1	DC	n	n		Rust `match` expression has a stricter syntax and enforces a lot of the constraints.
Rule 16.2	DC	n	n		Rust `match` expression has a stricter syntax and enforces a lot of the constraints.
Rule 16.3	DC	n	n		Rust `match` expression has a stricter syntax and enforces a lot of the constraints.
Rule 16.4	DC	n	n		Enforced by `match` constraints.
Rule 16.5	DC	n	n		A catch-all pattern causes subsequent patterns to be unreachable and diagnosed as such.
Rule 16.6	DC	n	n		Rust `match` expression has a stricter syntax and enforces a lot of the constraints.
Rule 16.7	DC	n	n		Rust `match` expression has a stricter syntax and enforces a lot of the constraints.
Rule 17.1	UB	n	n		Rust `match` expression has a stricter syntax and enforces a lot of the constraints.
Rule 17.2	UB, DC	y	y		
Rule 17.3	UB	n	n		
Rule 17.4	UB	n	n		Rust allows to return values without using the `return` keyword, and the corresponding condition is enforced by the compiler.

continued on next page

continued from previous page

Guideline	Rationale	Applicability unsafe/FFI	Applicability safe	Rust UB	Notes
Rule 17.5	UB, DC	n	n		
Rule 17.7	DC	y	y		`must_use` can help indicate where this is important, but does not affect applicability.
Rule 17.8	DC	y	y		Parameters are not modifiable by default, unless declared `mut`, but nothing prevents this to be done.
Rule 17.9	UB	y	n		Rust expresses divergence with a separate type, the `!` (Never) type.
Rule 17.10	DC	n	n		Condensed into the semantics of the `!` (Never) type.
Rule 17.11	DC	y	y		Rust does not enforce a diverging function to have the `!` (Never) type, therefore the rule is applicable.
Rule 17.12	DC	n	n		
Rule 17.13	UB	n	n		Rust syntax prevents a function type to have qualifiers.
Rule 18.1	UB	y	n	RUB-03	Using `unsafe`, a pointer beyond the bounds of an array can be obtained and dereferenced,
Rule 18.2	UB	y	n	RUB-03	Subtracting pointers belonging to different arrays can be done in `unsafe` blocks.
Rule 18.3	UB	y	y	RUB-03	Rust raw pointers implement the `PartialOrd` trait, therefore they can be operands to the comparison operators.
Rule 18.4	DC	y	n	RUB-03	Applies to use of the `unsafe` API.
Rule 18.5	DC	y	y		
Rule 18.6	UB	y	n	RUB-02	In unsafe Rust dangling pointers are a concern of the developer.
Rule 18.7	UB, DC	n	n		Rust does not have flexible array members.
Rule 18.8	UB, DC	n	n		Rust does not have VLAs.
Rule 18.9	UB	n	n		Rust does not have the concept of *temporary lifetime*.
Rule 18.10	UB	n	n		Rust does not have VLAs.
Rule 19.1	UB	y	n		Only possible in `unsafe` code.
Rule 19.2	UB, DC	y	y		In Rust a `union` can be read only in `unsafe` blocks, but can be written in safe blocks.
Rule 19.3	UB	y	y	RUB-09	Applicable to both safe and unsafe due to the fact that writing to a `union` is safe, while reading from it must be placed inside an `unsafe` block, therefore the constraint is only for unsafe blocks, but checking that the rule is complied with will involve both safe and unsafe code.
Rule 20.1	UB	n	n		Rule specific to the C preprocessor.
Rule 20.2	UB	n	n		Rule specific to the C preprocessor.
Rule 20.3	UB	n	n		Rule specific to the C preprocessor.
Rule 20.4	UB	p	p		Raw identifiers can allow to use certain reserved keywords as identifiers (weak keywords), but the visual marker `r#` makes this explicit.
Rule 20.5	DC	n	n		Rule specific to the C preprocessor.
Rule 20.6	UB	n	n		Rule specific to the C preprocessor.
Rule 20.7	DC	p	p		Only applicable to procedural macros: macros introduced by `macro_rules` do not have this issue.
Rule 20.8	DC	n	n		Rule specific to the C preprocessor.
Rule 20.9	DC	n	n		Rule specific to the C preprocessor.
Rule 20.10	UB	n	n		Rule specific to the C preprocessor.
Rule 20.11	UB	n	n		Rule specific to the C preprocessor.
Rule 20.12	DC	n	n		Rule specific to the C preprocessor.

continued on next page

continued from previous page

Guideline	Rationale	Applicability unsafe/FFI	Applicability safe	Rust UB	Notes
Rule 20.13	DC	n	n		Rule specific to the C preprocessor.
Rule 20.14	DC	n	n		Rule specific to the C preprocessor.
Rule 20.15	UB	n	n		Rule specific to the C preprocessor.
Rule 21.3	UB, IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.4	UB	y	n		only accessible through unsafe `extern "C"`.
Rule 21.5	UB, IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.6	UB, IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.7	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.8	UB, IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.9	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.10	UB, IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.11	UB	n	n		As this is handled with macros in C, this is not accessible from Rust via the FFI.
Rule 21.12	UB, IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.13	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.14	DC	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.15	DC	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.16	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.17	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.18	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.19	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.20	IDB, DC	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.21	UB, IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.22	UB	n	n		As this is handled with macros in C, this is not accessible from Rust via the FFI.
Rule 21.23	DC	n	n		As this is handled with macros in C, this is not accessible from Rust via the FFI.
Rule 21.24	CQ	y	n		Only accessible through unsafe `extern "C"`.
Rule 21.25	UB	y	y		Rust has the `std::sync::atomic::Ordering` enum with the `SeqCst` variant, and its atomic operations model heavily borrows from that of C++20 atomics.
Rule 21.26	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.1	UB, CQ	y	n		Applies to resources acquired through FFI only.
Rule 22.2	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.3	UB, IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.4	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.5	IDB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.6	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.7	DC	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.8	DC	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.9	DC	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.10	DC	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.11	UB	y	p		Rust has `std::thread::spawn` to create threads; it returns an owned `JoinHandle` that can be used to join the thread. If the join handle is dropped, the thread is implicitly detached, which leads to a resource leak. Rust ownership rules prevent a thread to be joined more than once, but unsafe code should be careful not violate the rule.

continued on next page

continued from previous page

Guideline	Rationale	Applicability unsafe/FFI	Applicability safe	Rust UB	Notes
Rule 22.12	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.13	UB, DC	y	y	RUB-02	
Rule 22.14	UB	y	p		Rust thread synchronization primitives cannot be used while uninitialized, but it is still a good practice to create them before creating the threads that use them.
Rule 22.15	UB	y	p		Rust does not have explicit thread synchronization objects destruction APIs, but it is stil possible to cause a panic if they are misused.
Rule 22.16	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.17	UB	y	n		Only accessible through unsafe `extern "C"`.
Rule 22.18	UB	y	y		Rust does not specify the behaviour of a non-recursive `Mutex` after calling the `lock()` method while holding the lock. The only guarantee is that the call will not return (either panic or deadlock is possible).
Rule 22.19	UB	y	y	RUB-05	Rust has the `CondVar` type. Associating different mutexes to the same condition variable may cause a panic in the `wait()` method.
Rule 22.20	UB	y	p		Rust has the `std::thread::LocalKey` type and the `thread_local!` macro. While there is no undefined behaviour tied to accessing the value before it has been set, the APIs for the use of thread-local storage are subjected to panics upon several conditions, which is generally undesirable.
Rule 23.1	DC	n	n		Generic selections are a C-specific feature. Rust has generics and traits that form a much wider and complex area of the language, whose issues are best addressed by Rust-specific guidelines.
Rule 23.2	DC	n	n		See the note on Rule 23.1.
Rule 23.3	DC	n	n		See the note on Rule 23.1.
Rule 23.4	DC	n	n		See the note on Rule 23.1.
Rule 23.5	DC	n	n		See the note on Rule 23.1.
Rule 23.6	DC	n	n		See the note on Rule 23.1.
Rule 23.7	DC	n	n		See the note on Rule 23.1.
Rule 23.8	DC	n	n		See the note on Rule 23.1.

TABLE II: MISRA C:2025-to-Rust mapping summary

Applicability	**unsafe/FFI**	**safe**
yes	125	61
partial	4	18
no	94	144
Total	223	223

A program that sticks to the ownership model of C-rusted has, by construction, guarantees about the absence of memory leaks, double free, invalid free and use after free. This implies compliance with Rule 22.1 and **Rule 22.2**, and it provides a valid argument to deviate Dir 4.12 and Rule 21.3 for the aspects related to memory management errors.[16]

2) Temporal Memory Safety: Temporal memory safety is about accessing objects within their lifetimes. Dangling pointers threaten memory safety, since they can be misused to accessed objects after the expiration of their lifetimes. MISRA C Rule 18.6 and Rule 18.9 are there to prevent this from happening.

C-rusted defines different kinds of resources with different *lifespans* to precisely express when a resource is released in relation to the entire program (extending the C Standards notions of *storage duration* and *lifetime*). Constraints related to the ownership model and borrowing, combined with the concept of lifespan of resources, allow the *C-rusted Analyzer* checking that every time a reference is used, it refers to a resource whose lifespan has not ended, thus ensuring temporal memory safety. As a result, enforcing the constraints imposed by C-rusted leads to compliance with Rule 18.9 and the formulation of a safe deviation for Rule 18.6, since no dangling pointer can be used without raising a warning of the analyzer.

3) Spatial Memory Safety: Spatial memory safety is about accessing objects within the intended boundaries. Several MISRA C guidelines are there to prevent buffer overflow/underflow: Rule 17.5, Rule 18.1, **Rule 21.17**, and **Rule 21.18**.

C-rusted includes forward, bidirectional and string iterators by explicitly associating to a reference the information about the boundaries of the referred resource, such as size, the count of elements and the references to the beginning element and/or to the past-the-end element. This allows the programmer iterating in a controlled and safe way with the assistance of the *C-rusted Analyzer*: if no warnings are given, this ensures compliance with the aforementioned rules for code not inside unsafe blocks. If manual iteration or pointer arithmetic is needed, this will have to be contained in unsafe blocks and compliance with such rules will have to be demonstrated, or other means of ensuring safety will have to be applied.

4) Resource Acquisition and Release: The C-rusted ownership model is applicable to any kind of system- or user-defined resource: it is based on `e_own()` annotations and it ensures each resource is correctly acquired and then released. As an example, consider the `fopen()` and `fclose()` functions of `<stdio.h>`: the former is interpreted in C-rusted as if returning an optional owning reference to a `FILE` object; the latter is considered the release function for such ownership.[17]

[16]Note that there are other factors to be considered before allowing the use of dynamic memory in safety critical systems, such as the execution time of allocators/deallocators and dealing with out-of-storage run-time failures [17].

[17]C Standard library and the POSIX library have been annotated once and for all, so that the *C-rusted Analyzer* is able to correctly interpret the behavior of their functions (the same can be done for any frequently used library).

A program compliant to the C-rusted ownership model is therefore compliant with MISRA C Rule 22.1 (All resources obtained dynamically by means of Standard Library functions shall be explicitly released), **Rule 22.6** (The value of a pointer to a `FILE` shall not be used after the associated stream has been closed), and compliant with *Dir 4.13* (Functions which are designed to provide operations on a resource should be called in an appropriate sequence) for the part related to resource allocation and deallocation.

5) Initialization: **Rule 9.1** (The value of an object with automatic storage duration shall not be read before it has been set) and **Rule 9.7** (Atomic objects shall be appropriately initialized before being accessed) are MISRA C's mandatory guidelines regarding object initialization.

C-rusted manages uninitialized memory in a way similar to the one used for optional pointers: every formal parameter is considered as initialized by default, meaning that, whenever possibly uninitialized memory is passed to or returned from a function, this shall be made explicit via a `e_uninit()` annotation. This requires also the support for initialization functions: the `e_init()` annotation is used to annotate formal parameters to specify that the function will completely initialize the referred resource (i.e., it will write it completely and not read any part of it without prior writing). In the case of atomic objects, according to the amplification of **Rule 9.7**, the *C-rusted Analyzer* considers the object as initialized only if directly initialized in its declaration or after a call to `atomic_init()`. Effective and complete tracking of the initialization state of resources throughout the program avoids any possible read from uninitialized memory locations, thereby ensuring compliance with both **Rule 9.1** and **Rule 9.7**.

6) Error Handling: C-rusted provides `e_err()` annotations to explicitly define whether a function returns error information through its return value, a parameter, or a global variable. The *C-rusted Analyzer* checks whether such error information is handled after every function call and raise an error if this is not the case, or if the function does not set the error information in at least one of its exit paths. A disciplined programmer, by following the discipline of proper function annotation, can use this as an argument to claim compliance with Dir 4.7 (If a function returns error information, then that error information shall be tested) and partial compliance with Rule 17.7 (The value returned by a function having non-void return type shall be used).

7) Restricted Pointers: Use of the C's `restrict` type qualifier can result in performance improvements and more precision in static analysis. Nonetheless, a misuse of such qualifier caused by unwanted pointer aliasing is undefined behavior: this is why use of `restrict` is discouraged by MISRA C Rule 8.14.

C-rusted distinguishes between exclusive and shared references. As the name suggests, an exclusive reference grants exclusive access to a resource, meaning that the *C-rusted Analyzer* ensures that no more than one usable exclusive reference to the same resource can exist at any given time. Moreover, the existence of a usable exclusive reference is incompatible

with the existence of any other usable reference to the same resource. This is a valid argument to deviate Rule 8.14 when implicit or explicit exclusive references are `restrict`-qualified: all the constraints related to the `restrict` qualifier are, in fact, implicitly satisfied by the borrowing model.

B. MISRA C for C-rusted

While the most infamous and dangerous undefined behaviors of C are avoided in C-rusted, other undefined and critical unspecified behaviors are not covered. The reason is that, C-rusted has been designed to leverage on MISRA C while being 100% compatible with it.

In C-rusted, code not inside unsafe blocks shall be MISRA compliant under the *C-rusted Guideline Re-categorization Plan*. A *Guideline Re-categorization Plan* (GRP) [26] is an assignment to each MISRA C guidelines of a possibly new category as follows:

- A *required* guideline can be re-categorized as *required* or *mandatory*;
- A *advisory* guideline can be re-categorized as *advisory*, *required*, *mandatory*, or *disapplied*, the latter meaning that the guideline needs not be checked.

In the C-rusted GRP, most MISRA C guidelines regarding undefined, or critical unspecified, or implementation-defined behavior are re-categorized as *mandatory*. In addition, *Dir 1.2* (The use of language extensions should be minimized), is also re-categorized as *mandatory*. Consequently:

- Possibly non-definite pointer conversions can only be done in unsafe blocks, as the rules in MISRA C Series 11 (*Pointer type conversions*) from Rule 11.1 to Rule 11.8 are mandatory in safe C-rusted code.
- Language extensions, which includes assembly code expressed alongside C code in whatever form, can only be used in unsafe blocks.

Rule 8.13 (A pointer should point to a `const`-qualified type whenever possible) is also re-categorized as *required*: this enforces const correctness and, in turn, is exploited by the type inference of C-rusted to infer the exclusive or shared nature of a reference without the need of annotations.

In general, compliance with the (re-categorized) MISRA C guidelines simplifies the job of any static analyzer, including the *C-rusted Analyzer*.

VI. Integration of MISRA C, C-rusted, and Rust in Safety-Critical Industries

Industries like automotive, aerospace, railways, and medical require high-assurance software due to strict safety, security, and reliability requirements. Each industry has legacy C codebases, but also needs modern, memory-safe solutions. They also need to comply with functional safety and cybersecurity standards. This is where MISRA C, C-rusted, and Rust may work together to ensure safe and secure software development. We review some of the challenges and possible solutions adopting all these technologies in the following sections, each one devoted to a specific industry sector.

A. Automotive Industry

Standards: ISO 26262 (*Road Vehicles — Functional Safety*) [28].

Challenges:

- Real-time constraints, power efficiency, and strict safety requirements.
- Extensive use of C due to hardware constraints and legacy software.
- Growing demand for memory safety and security hardening, especially in autonomous driving.

Possible integration: MISRA C enforces safe coding guidelines for C to comply with ISO 26262; C-rusted introduces ownership, exclusivity, and advanced static analysis to improve memory safety in existing MISRA C codebases; Rust can be used for safety-critical components, particularly in ADAS (*Advanced Driver Assistance Systems*) and in next-gen automotive firmware.

Example integration: ECU (*Electronic Control Unit*) where: existing ECUs rely on MISRA C for compliance with ISO 26262; C-rusted is used to refactor and validate safety-critical functions while ensuring compliance; Rust is introduced for new safety-critical modules, such as sensor fusion or AI-based decision-making in autonomous systems.

B. Aerospace Industry

Standards: RTCA DO-178C (*Software Considerations in Airborne Systems*) [29].

Challenges:

- High certification costs (DO-178C requires extensive testing and verification).
- Legacy avionics software in C must be maintained for decades.
- Security threats to avionics systems require better memory-safe approaches.

Possible integration: MISRA C enforces structured C development for DO-178C compliance; C-rusted allows safe refactoring of avionics software while maintaining certification; Rust can be used in new secure avionics modules, flight control software, and mission-critical embedded systems.

Example integration: *Flight Control Systems* where: existing flight software follows MISRA C; C-rusted enables safer updates to navigation, fault-tolerant systems; Rust may be adopted for mission-critical firmware in newer designs (see, e.g., [30]).

C. Railway Signaling and Control Systems

Standards: CENELEC EN 50128, EN 50657, EN 50716 (*Software for Railway Control and Protection Systems*). [31], [32], [33].

Challenges:

- Railway control systems extremely safety-critical.
- Long-term software maintenance (railway systems last 30+ years).

- Standards compliance mandating extensive software verification.

Possible integration: MISRA C ensures safe coding practices in railway control software; C-rusted may be used to validate memory safety in existing control system firmware; Rust may be applied to newer train automation, AI-based monitoring, and security-critical modules.

Example integration: *Railway Signaling Systems* where: existing train control systems strongly rely on MISRA C; C-rusted improves safety verification in legacy railway firmware; Rust is used in AI-driven predictive maintenance (e.g., detecting rail faults).

D. Medical Devices and Healthcare Software

Standards: IEC 62304 (*Medical Device Software*) [34]; FDA's *General Principles of Software Validation* [35].

Challenges:

- Memory safety bugs in medical devices can be life-threatening (see, e.g., [36]).
- Medical software must comply with functional safety and security standards. Legacy C-based firmware needs security improvements.
- Standards compliance mandating extensive software verification.

Possible integration: MISRA C ensures safe, secure and reliable firmware for medical devices; C-rusted enables safer static analysis and verification for C-based medical software; Rust may be used in secure patient monitoring, medical imaging, and robotic surgery systems.

Example integration: *Pacemakers and Medical Implants* where: legacy pacemaker software follows the MISRA C guidelines; C-rusted validates memory safety in critical heart-monitoring firmware; Rust is used for AI-based diagnostics and real-time imaging.

Summarizing, we envisage a gradual and practical transition. Since many critical industries still rely on C, transitioning entirely to Rust is not feasible, but we believe a hybrid approach works. As regulatory bodies are increasingly pushing for memory-safe languages, more industries will incrementally move towards Rust, but C will persist in legacy systems and very performance-sensitive applications. In this scenario, C-rusted may act as a bridge, allowing industries enhancing safety in C without massive rewrites.

VII. Conclusion

The increasing demand for secure and reliable software has made memory safety a critical concern in industries such as automotive, aerospace, railways, and medical devices. Governmental and regulatory bodies have recognized this urgency, advocating for the adoption of memory-safe programming languages as a means to strengthen cybersecurity. While Rust has been positioned as a leading solution due to its strong memory safety guarantees, a full-scale migration from C to Rust is neither practical nor economically viable for many industries that rely on extensive C-based ecosystems.

A pragmatic approach to improving software security must therefore involve a gradual and integrated transition that balances memory safety with existing investments in software infrastructure. MISRA C has long provided a foundation for safe C programming, enabling safety-critical industries to improve software reliability while maintaining compatibility with legacy systems. C-rusted extends this approach by incorporating modern verification techniques inspired by Rust, allowing for stronger static analysis and improved memory safety within existing C codebases. Meanwhile, Rust can be strategically introduced for new developments where its ownership model and zero-cost abstractions offer significant security and performance advantages.

By adopting an integrated strategy that leverages MISRA C, C-rusted, and Rust, industries can achieve meaningful cybersecurity improvements without disrupting existing workflows. This approach ensures that memory safety is enhanced incrementally, reducing costs while maximizing security benefits. Furthermore, just as MISRA C has guided the safe use of C, it is essential to define a corresponding set of guidelines for Rust, ensuring that its use in safety-critical domains aligns with best practices and regulatory requirements.

Ultimately, the transition to more secure software must be driven by practical considerations rather than hype. A well-balanced methodology — grounded in proven safety standards, incremental adoption, and rigorous security verification — offers the most effective path forward. By combining MISRA C for legacy safety, C-rusted for enhanced C verification, and Rust for forward-looking security, industries can achieve safer, more resilient, and future-proof software architectures without sacrificing the value of decades-long investments in C-based systems.

Acknowledgments

We wish to thank the following members of the *MISRA C Working Group*, and their employers, for their significant contribution to the joint work that led to the creation of Table I and of MISRA C:2025 Addendum 6 [25]: Andrew Banks (LDRA Ltd and Intuitive Consulting) and Alex Celeste (Perforce). Thanks are also due to other members of the *MISRA C Working Group*, and their employers, for their contribution to the review phase of the documents, namely: Jill Britton (Perforce), Douglas Deslauriers (Vector Informatik GmbH), Daniel Kästner (AbsInt Angewandte Informatik GmbH), Gerlinde Kettl (Vitesco Technologies GmbH), Gavin McCall (Codethink Ltd), Chris Miller (GE Aerospace), Chris Tapp (Keylevel Consultants), and David Ward (HORIBA MIRA Limited). The MISRA C Guideline headlines are reproduced with permission of *The MISRA Consortium Limited*. Last but not least, we are grateful to Lavinia Battaglia and Patricia M. Hill, both of BUGSENG, for their help improving the present manuscript.

References

[1] "The Case for Memory Safe Roadmaps — Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously," U.S. Cybersecurity and Infrastructure Security Agency, U.S. National

Security Agency, U.S. Federal Bureau of Investigation, Australian Signals Directorate's Australian Cyber Security Centre, Canadian Centre for Cyber Security, U.K. National Cyber Security Centre, New Zealand National Cyber Security Centre, Computer Emergency Response Team New Zealand, Tech. Rep. 508c, Dec. 2023.

[2] White House Office of the National Cyber Director. (2024, Feb.) Back to the building blocks: A path toward secure and measurable software. Previously available at https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/e/e6/Back_to_the_Building_Blocks_-_A_Path_Toward_Secure_and_Measurable_Software.pdf

[3] MISRA, *MISRA C:2012 — Guidelines for the use of the C language in critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Mar. 2013.

[4] R. Bagnara, A. Bagnara, and F. Serafini, "C-rusted: The advantages of Rust, in C, without the disadvantages," Report `2302.05331 [cs.PL]`, 2023, available at http://arxiv.org/. [Online]. Available: https://arxiv.org/abs/2302.05331

[5] R. Bagnara, A. Bagnara, N. Vetrini, and F. Serafini, "C-rusted: A formally verifiable flavor of C for the development of safe and secure systems," in *embedded world Conference 2024 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2024, pp. 427–438.

[6] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2022, online version available at https://doc.rust-lang.org/book/ and maintained at https://github.com/rust-lang/book, last accessed on March 9, 2024.

[7] MISRA, *MISRA C:2025 — Guidelines for the use of the C language in critical systems*. Norwich, Norfolk, NR3 1RU, UK: The MISRA Consortium Limited, Mar. 2025.

[8] P. Cousot, *Principles of Abstract Interpretation*. MIT Press, 2021.

[9] D. Kroening, P. Schrammel, and M. Tautschnig, "CBMC: The C Bounded Model Checker," Report `arXiv:2302.02384 [cs.SE]`, 2023, available at http://arxiv.org/.

[10] D. M. Ritchie, *The Development of the C Programming Language*. New York, NY, USA: Association for Computing Machinery, 1996, pp. 671–698. [Online]. Available: https://doi.org/10.1145/234286.1057834

[11] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, C. S. Ellis, Ed. Monterey, CA: USENIX Association, Jun. 2002, pp. 275–288. [Online]. Available: http://www.usenix.org/publications/library/proceedings/usenix02/jim.html

[12] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. (2024, Nov.) ANSI/ISO C specification language. CEA-List, Inria. Version 1.21. [Online]. Available: https://github.com/acsl-language/acsl

[13] R. Bagnara, A. Bagnara, P. M. Hill, and N. Vetrini, *Software Verification Done Right: Introduction to Static Analysis*, 1st ed. Parma, Italy: BUGSENG, 2025. [Online]. Available: https://doi.org/10.979.12210/84894

[14] Motor Industry Software Reliability Association, *MISRA-C:1998 — Guidelines for the use of the C language in vehicle based software*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Jul. 1998.

[15] R. Bagnara, A. Bagnara, and P. M. Hill, "The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software," in *Static Analysis: Proceedings of the 25th International Symposium (SAS 2018)*, ser. Lecture Notes in Computer Science, A. Podelski, Ed., vol. 11002. Freiburg, Germany: Springer International Publishing, 2018, pp. 5–23.

[16] ——, "The MISRA C coding standard: A key enabler for the development of safety- and security-critical embedded software," in *embedded world Conference 2019 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2019, pp. 543–553.

[17] ——, "A rationale-based classification of MISRA C guidelines," in *embedded world Conference 2022 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2022, pp. 440–451.

[18] ——, "Coding guidelines and undecidability," in *embedded world Conference 2023 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2023, pp. 488–499.

[19] R. Bagnara, S. Stabellini, N. Vetrini, A. Bagnara, S. Ballarin, P. M. Hill, and F. Serafini, "Bringing existing code into MISRA compliance: Challenges and solutions," in *embedded world Conference 2024 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2024, pp. 327–338.

[20] H. Zhang. (2023, Apr.) Comprehensive understanding of unsafe Rust. [Online]. Available: https://rustmagazine.org/issue-3/understand-unsafe-rust

[21] The Rust Team. (2025) The Rust reference. [Online]. Available: https://doc.rust-lang.org/stable/reference/

[22] MISRA, *MISRA C++:2023 — Guidelines for the use of C++17 in critical systems*. Norwich, Norfolk, NR3 1RU, UK: The MISRA Consortium Limited, Oct. 2023.

[23] IEEE, *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society, Jul. 2019, IEEE Std 754-2019 (Revision of IEEE Std 754-2008).

[24] D. Keep, A. Burka, A. Gaynor, and D. van Berkel. (2016) The little book of Rust macros. [Online]. Available: https://danielkeep.github.io/tlborm/book/mbe-min-hygiene.html

[25] MISRA, *MISRA C:2025 Addendum 6 — Applicability of MISRA C:2025 to the Rust Language*. Norwich, Norfolk, NR3 1RU, UK: The MISRA Consortium Limited, Mar. 2025.

[26] ——, *MISRA Compliance:2020 — Achieving compliance with MISRA Coding Guidelines*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2020.

[27] The Rust Team. (2025) The Rustonomicon. [Online]. Available: https://doc.rust-lang.org/stable/nomicon/

[28] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Dec. 2018.

[29] RTCA, SC-205, *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Dec. 2011.

[30] NASA. (2021, Sep.) Rust in cFS: Prevent bugs with memory-safe programming. Goddard Space Flight Center. Completed technology project page. [Online]. Available: https://techport.nasa.gov/projects/96767

[31] CENELEC, *EN 50128:2011/A2:2020: Railway applications — Communication, signalling and processing systems — Software for railway control and protection systems*. Brussels, Belgium: CENELEC, Aug. 2020, amendment A2 to EN 50128:2011.

[32] ——, *EN 50657:2017/A1:2023: Railway applications — Rolling stock applications — Software on Board Rolling Stock*. Brussels, Belgium: CENELEC, Nov. 2023, amendment A1 to EN 50657:2017.

[33] ——, *EN 50716:2023: Railway Applications — Requirements for software development*. Brussels, Belgium: CENELEC, Nov. 2023.

[34] IEC, *IEC 62304:2006/Amd 1:2015: Medical device software — Software life cycle processes — Amendment 1*. Geneva, Switzerland: IEC, Jun. 2015.

[35] U.S. Department Of Health and Human Services, Food and Drug Administration, Center for Devices and Radiological Health, Center for Biologics Evaluation and Research, *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*. CDRH, CBER, Jan. 2002.

[36] C. M. Mejía-Granda, J. L. Fernández-Alemán, J. M. Carrillo-de-Gea, and J. A. García-Berná, "Security vulnerabilities in healthcare: an analysis of medical devices and software," *Medical & Biological Engineering & Computing*, vol. 62, pp. 257–273, 2024.