Proceedings of the 35th European Safety and Reliability & the 33rd Society for Risk Analysis Europe Conference
Edited by Eirik Bjorheim Abrahamsen, Terje Aven, Frederic Bouder, Roger Flage, Marja Ylönen
 Published by Research Publishing, Singapore.
doi: 10.3850/978-981-94-3281-3_ESREL-SRA-E2025-P7174-cd

Enhancing Software Safety Through Programming Languages: A Study of Rust

Thor Myklebust

SESS, SINTEF Digital, Norway. E-mail: thor.myklebust@sintef.no

Christian Askeland

Technology - SW Engineering, Autronica Fire and Security, Norway. E-mail: christian.askeland@carrier.com

Espen Helle

MatCyb, SINTEF Digital, Norway. E-mail: espen.helle@sintef.no

Ensuring software safety has become a paramount concern in modern software development, with the choice of programming language playing a crucial role. This paper investigates the role of Rust, a systems programming language, in enhancing software safety, with a specific focus on embedded development and microcontrollers in the context of developing a modern fire alarm system. Through a combination of literature review, evaluation of safety standards, three discussion meetings with software engineers, and practical experimentation, we explore the unique features of Rust that contribute to safer software development practices. The study includes an in-depth analysis of Rust's ownership model and concurrency mechanisms, comparing them with traditional languages like C and C++. Furthermore, we conduct interviews with software engineers to gather insights into their experiences with Rust, particularly its adoption challenges, benefits, and implications for transitioning from C++ and Python to Rust. Additionally, we present a practical experiment involving code development in Rust, specifically tailored to a modern fire alarm system, to demonstrate its effectiveness in ensuring safety and reliability in critical embedded applications. The findings of this study provide valuable insights into the role of programming languages, particularly Rust, in advancing software safety. They also offer practical guidance for software developers aiming to leverage safer alternatives in their projects, especially in the specialized domain of embedded systems and microcontroller-based safety-critical applications.

Keywords: RUST, Safety, C, C++, Programming language, IEC 61508.

1. Introduction

Rust is emerging as a key programming language in safety-critical domains, offering unique features that address the challenges of developing reliable software for embedded systems and microcontrollers. The increasing complexity of embedded systems and microcontroller-based applications has elevated the importance of ensuring software safety in safety-critical domains such as automotive, aerospace, and industrial automation. Failures in these systems can result in severe consequences, including loss of life, financial damage, or compromised data integrity. As software grows in complexity, traditional programming languages like C and C++ struggle to adequately mitigate common issues such as memory leaks, null pointer dereferences, and race conditions.

RUST's memory safety, concurrency model, and strong&static type system minimize errors like memory leaks, null pointers, and data races, making it a safer alternative to traditional languages such as C and C++. By enforcing ownership, borrowing, and lifetime rules, Rust ensures predictable memory usage and thread safety while eliminating many runtime errors.

In the context of functional safety standards like IEC 61508 [IEC 61508 2010], Rust is gaining recognition for its potential. Certified Rust compilers, such as Ferrocene

[Ferrocene 2025] and HighTec's AURIX Rust Compiler [AURIX 2025], provide compliance with safety standards like SIL-4 (Safety Integrity Level) and ASIL-D, (Automotive SIL) while tools like svd2rust and standardized Hardware Abstraction Layers (HAL) facilitate efficient development for embedded systems. The launch of the Safety-Critical Rust Consortium [Rust Foundation. 2024] underscores the language's growing adoption, fostering its responsible use in safety-critical applications. This paper explores Rust's features, tools, and ecosystem, highlighting its suitability for advancing software safety in critical systems.

2. Background

In this paper, we draw upon a variety of official resources, industrial insights, and foundational materials to provide accurate and up-to-date information on tools, frameworks, and ongoing initiatives. This approach is essential as traditional academic publications often lag behind the rapid advancements in technology and industry practices, particularly in emerging areas such as programming language ecosystems and safety-critical software development. When starting to develop in Rust for small microcontrollers, it is recommended to read through the "The Embedded Rust Book" [Rust Embedded Working Group. 2025]. This book explains relevant topics and provides example code that can be run on physical hardware or using an QEMU (Quick emulator) [QEMU2025] an open-source machine emulator that allows developers to run programs designed for one computer architecture on a different one. It is an excellent resource for experienced embedded developers coming from the C or C++ domain. But it assumes the reader has a basic understanding of the Rust language.

In the context and scope of this paper only a few relevant papers have been published. Two of them are presented below. According to [Okutan et al.], translating C++ to Rust is challenging due to differences in syntax, ecosystem, and memory safety. Large Language Models (LLMs) using Retrieval-Augmented Generation (RAG) offer a promising solution for source code translation. Cpp2Rust, an LLM-based transpiler leveraging OpenAI's ChatGPT-4 and a dataset of C++ and Rust snippets from LeetCode, facilitates structured translation. Initial evaluations show that 84% of transpiled solutions without compilation errors were accepted on LeetCode, demonstrating the potential of AI-driven tools in automating language migration. This aligns with efforts to enhance software safety, where Rust's ownership model and memory safety provide advantages over C++. By applying advanced AI techniques, Cpp2Rust helps developers transition C++ codebases to Rust while ensuring correctness and efficiency. According to [Xia et al.], Rust provides strong security guarantees, its unsafe feature introduces memory vulnerabilities due to the absence of compile-time and run-time checks. Diagnosing these vulnerabilities is particularly challenging due to complex interactions between safe and unsafe code. To address this, Rustcheck was developed as the first dynamic program analysis framework aimed at enhancing memory safety in Rust. By leveraging static instrumentation, Rustcheck dynamically detects memory issues stemming from improper use of unsafe Rust. In an evaluation involving 56 real-world CVEs, Rustcheck successfully detected all 65 memory vulnerabilities while maintaining a low runtime overhead of 3.30% on average. These results highlight the effectiveness of Rustcheck in improving the security of Rust programs while preserving performance.

The different functional safety standards include requirements for programming languages. Below we have described the requirements for the generic std IEC 61508:2010. IEC 61508-3 [IEC 61508-3:2010] and IEC 61508-7 [IEC 61508-7:2010]: These standards include generic requirements for programming languages in part 3 and guidelines in part 7. The challenge for some companies has been that RUST is not listed in the guidelines in part 7 while several others are listed in the guidelines. The acronyms in Table 1 below are: Systematic Capability (SC): measure (expressed on a scale of SC 1 to SC 4) of the confidence that the systematic capability of an element meets the requirements of the specified SIL, in respect of the specified element safety function, when the element is applied in accordance with the instructions specified in the safety manual for compliant items. Highly recommended (HR): the technique or measure is highly recommended at this systematic capability level. Passive recommendation (PR) the technique

or measure has no recommendation for or against being used. Recommended (R): the technique or measure is recommended at this systematic capability level. Not recommended (NR): the technique or measure is positively not recommended at this systematic capability level. If this technique or measure is used, then the rationale behind using it should be detailed.

Programming language	SC2
3 Java	NR
4 Java with subset (including either no garbage collection or garbage collection which will not cause the application code to stop for a significant period of time). See Annex G for guidance on use of object oriented facilities.	R
9 C	PR
10 C with subset and coding standard, and use of static analysis tools	HR
11 C++ (see Annex G for guidance on use of object oriented facilities)	PR
12 C++ with subset and coding standard, and use of static analysis tools (see Annex G for guidance on use of object oriented facilities)	HR
13 Assembler	R
14 Assembler with subset and coding standard	R
15 New RUST	HR

Fig. 1. Extract of Table C.1 Programming languages listed in IEC 61508-7, where RUST is added for SIL-2.

The [ISO/IEC 24772-1:2024] standard provides a comprehensive language-independent catalogue of programming vulnerabilities to help software developers, engineers, and organizations identify, mitigate, and avoid common security and safety risks in software development. The document highlights vulnerabilities that can arise due to incomplete specifications, undefined behaviors, and implementation-dependent constructs in programming languages, which may lead to security breaches or system failures. It addresses general vulnerability issues such as predictable execution, type system failures, and buffer overflows. Additionally, it examines programming language vulnerabilities, including memory safety issues, unchecked array indexing, pointer arithmetic flaws, and concurrency-related issues, as well as application vulnerabilities such as path traversal, code injection, resource exhaustion, and improper access control. The standard also provides best practices for designing safer programming languages and writing secure software. It serves as a reference for developers, programming language designers, and safety and security practitioners, supporting compliance with functional safety and cybersecurity standards. This makes it particularly relevant for software in safety-critical and security-sensitive domains, including automotive, aerospace, industrial automation, and the process industry.

Tool validation and compliance are critical for meeting functional safety standards such as IEC 61508. Compilers are classified as T3 (most stringent), which directly contribute to safety-related systems, plays a key role in ensuring the reliability of these systems. Recent draft of the standard, including the removal of certain certification requirements and the introduction of Annex H for tool confidence, reflect the evolving practices in tool validation. RUST compilers have emerged as a strong choice for safety-critical applications due to their emphasis on memory and thread safety and performance and should be evaluated as T3 according to IEC 61508. Additionally, early integration of validated tools into the software development lifecycle is vital for reducing risks and ensuring compliance.

The Safety-Critical Rust Consortium, launched in 2024, aims to ensure Rust's responsible adoption in safety-critical software.

2.1. *Meeting with SAE J1020*

We had a meeting with SAE JA1020 "Recommendations for the Rust Programming Language in Safety-Related Systems" [SAE International. 2025. JA1020] December 2, 2024, discussed the status of the SAE Guide. Due to many abstentions, another vote is planned for January 2025, with publication expected in February 2025. Certified tools like Ferrocene and ADAcore can assist in SIL 2 compliance, while the use of open-source Rust libraries remains unaddressed. AutoSAR documentation now supports Rust, with successful integrations by Vector and HighTec for automotive safety

applications. Elektrobit introduced a Linux OS extension for ASIL B/SIL 2 compliance [Elektrobit 2025], and the DARPA TRACTOR [DARPA 2025] project plans to transition US government C code to Rust for enhanced safety. While core crates will be SIL-compliant, further expansions like Embassy are possible. Unsafe C-based modules could be sandboxed using WebAssembly. The Safety-Critical Rust Consortium, recently established, shows promise for future developments. ARM64 and QNX were also discussed briefly.

Copy from the SAE JA1020 Abstract: "*This document describes best practices for the use of Rust in Safety-Related Systems development. The scope will not include repetition of existing guidelines but will summarize and point to them; if existing guidelines differ from this document, these will be noted. Objectives of this task force will be to: 1. Evaluate the Rust ecosystem to identify a Safer subset of Rust. 2. Develop guidelines with respect to the Rust subset for: a. Integrating Rust into automotive and aerospace safety-related applications b. Avoiding programming mistakes and failures that could lead to hazards, and c. Increasing confidence in its use in the automotive and aerospace industry 3. Document evidence to support the guidelines, and to 4. Provide general recommendations for the use of Rust to support safety and cybersecurity*".

3. RUST programming for safety

Rust is becoming a popular choice for safety-critical applications due to its memory safety, robust concurrency model, and minimal runtime overhead. In the safety domains, where reliability also is paramount, Rust's innovative features stand out. Certified Rust compilers, such as Ferrocene (SIL-4 and ASIL-D compliant), make Rust highly suitable for embedded systems and high-assurance software. Tools like AdaCore's GNAT Pro for Rust [AdaCore 2025] and HighTec's AURIX Rust Compiler which is ISO 26262 [ISO 26262:2010] compliant for Infineon processors) [Infenion 2025] further expand its applicability in industries like automotive and aerospace.

Rust's safety lies in its unique ownership and its related features model and type system, which eliminate common bugs like memory leaks, null pointers, and data races. Each value in Rust has a single owner, and when the owner goes out of scope, the memory is automatically deallocated. Borrowing rules allow safe references to data without risking invalid or simultaneous mutable access. The compiler tracks how long references are valid using lifetimes, which prevents dangling pointers or use-after-free errors. Although these rules require a learning curve, they ensure predictable and safe memory usage. This deterministic runtime behaviour is essential for real-time systems and sets Rust apart from garbage-collected languages like Java or manually managed languages like C and C++. Rust supports flexible concurrency models that prioritize both performance and safety. Through its unique ownership and borrowing rules, Rust ensures that race conditions are prevented at compile time, even when working with native operating system threads. This safety is achieved using traits like Send and Sync, which play a key role in managing data across threads. The Send trait ensures that ownership of a type can be safely transferred to another thread, while the Sync trait guarantees that a type can be safely shared between threads, provided it is immutable or properly synchronized. Rust's async/await model enables lightweight task switching, making it ideal for I/O-bound operations without using threads. For compute-heavy workloads, libraries like Rayon distribute tasks efficiently across multiple CPU cores. Additionally, Rust encourages safe inter-thread communication through message-passing via channels, transferring ownership of data to prevent conflicts. Shared-state synchronization is supported using mutexes and atomic types, which integrate seamlessly with Rust's safety guarantees.

Rust's ecosystem includes tools designed to streamline development. Rust Analyzer provides smart IDE integration for efficient coding, Clippy

identifies inefficiencies and maintains code quality, and Rustdoc generates documentation automatically from comments. While the ecosystem offers many powerful libraries, certified options remain limited. Developers must carefully evaluate libraries before using them in critical systems, especially when compliance with standards like ISO 26262 is required. Rust's growing adoption in safety-critical domains demonstrates its potential to improve software reliability and safety.

Google's experience with Rust [The Register 2024] shows that Rust is a more productive and safer alternative to C++. Developers using Rust are reportedly twice as productive as those using C++, thanks to its modern features and superior memory safety, which help reduce debugging time.

Rust outperforms C++ in terms of memory and thread safety, while maintaining almost comparable performance with C++. These safety guarantees are archived at compile-time rather than relying on a garbage collector as many other memory-safe languages do.

Despite these advantages, challenges such as a steep learning curve and a smaller ecosystem remain. Rust also matches or exceeds Go in certain aspects, combining simplicity with high performance. While promising, further research is needed to validate these findings across various environments.

4. Case studies

A preliminary study was carried out at Autronica, a manufacturer of fire detection systems. Rust was used in several prototypes to evaluate its usability in the given environment. A team of developers began a greenfield project to build a next-generation fire detection system. While leveraging know-how and components from earlier products, many parts were developed from scratch using new hardware and software. They decided to experiment with technologies like Rust. The team already had experience in C, C++, Python, and JavaScript. Key components were selected for experimental development to test features, not for production, and applications were written for PC and embedded platforms. Devices used a mix of these platforms, interconnected via UDP/IP (User Datagram Protocol/Internet Protocol) and TCP/IP (Transmission Control Protocol/IP), and were designed to meet certifications like EN54 (Fire Detection) [European Standard. EN 54] and SIL-2. The preliminary study aimed to achieve two goals: gain experience with Rust, especially in safety applications, and validate key concepts such as domain objects, a black channel protocol, and core architecture. Final technology decisions would be based on comparing this work with the team's prior experience. The tools and libraries used in the experiment are shown below.

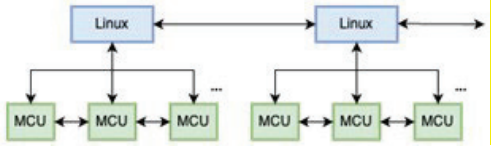

Fig. 2. Experimental setup. A collection of microcontrollers was connected over UDP to a Linux box which in turn were connected to each other over TCP. The same safety protocol was used for all connections.

The MCUs (Micro Controller Units) were STM32H745 microcontrollers [STMicroelectronics 2025] running Embassy [Embassy RS 2025], using Rust's no-std environment with minimal crates. PCs ran Linux (or other desktop OSs during development), with Rust applications using Tokio [Tokio 2025] and Tonic [Tonio 2025]. The safety protocol used Protocol Buffers with nanoPB/Prost for MCUs and Tonic/Prost for Linux. Async/await was employed due to the i/o-limited nature, requiring few threads. Certified libraries will replace some, like Embassy, in the final product. Parallel Python applications were created for Linux as drop-in replacements for Rust versions for verification and testing.

Results and Analysis: Prototype applications included AutroPoll, a safety protocol for state distribution with redundancy and robustness, based on Protocol Buffers, operating over UDP or gRPC/TCP (Google Remote Procedure Call/TCP) with long polling and pagination. Clients and servers for AutroPoll were implemented on all platforms. A decision engine

tested fire detection input-to-output processing scalability using buttons and LEDs for input/output on MCUs. MCU firmware avoided dynamic heap allocation for safety, using preallocated memory (e.g., heapless:Vec). The C library nanoPB was integrated into Rust, demonstrating compatibility between C and Rust in applications. Over 12–18 months, three developers adopted Rust gradually, becoming confident in using its features. Testing (unit, module, integration) revealed no race conditions or memory errors—unlike C/C++, where debugging often involves such issues. Debugging instead focused on functionality, such as module interoperability and behaviour correctness. Developers noted that once Rust code compiled, major bug categories were eliminated, streamlining development.

Another company, specializing in ultrasound sensing systems, is exploring Rust for safety-critical applications. Their experience highlights Rust's memory safety and compile-time checks, alongside current limitations. Focused on IEC 61508 certification with the certification body Exida, they use Ferrous Systems' certified Rust compiler and are transitioning to bare-metal on the Cortex-M7 processor for efficiency. Bare metal refers to running software directly on hardware without an operating system. It gives complete control over the system but requires you to manage hardware resources, such as memory and peripherals, manually. It is often used in embedded systems for efficiency and real-time performance. Challenges persist for applications requiring SIL above 2, with certified libraries like libcore still in development. They avoid uncertified components and separate safety-critical and non-safety-critical functions across processor cores using black channel communication, eliminating the need for an operating system. While Rust's safety-critical ecosystem matures, their collaboration with certification bodies and modular, deterministic designs demonstrates their commitment to modern tools and certification challenges.

5. Challenges and considerations in adopting RUST

Transition from C++ to Rust: While transitioning from C++ to Rust, developers encounter both similarities and challenges. Familiar elements include a strong type system, rich libraries, polymorphic behaviour through traits, and generics. However, Rust's strictness introduces new concepts like lifetimes and ownership, which enforce better practices but may initially feel restrictive. C++ developers, particularly those using older compilers (C++14 or earlier), often rely on basic C-like features, leading to low-quality code—a practice Rust inherently prevents. Newer C++ versions include concepts closer to Rust, easing the learning curve slightly. The ownership model in Rust, including lifetimes and borrowing, may seem complex to C++ users accustomed to memory management with smart pointers. Error handling through the Result type and the "?" operator differs from exceptions but feels more like an improvement, especially for those familiar with std::expected. Traits for function signatures are extensively used in Rust, including Sync and Send, which have conceptual parallels in C++. Rust lacks C++'s header files for separating class interfaces and implementations, requiring C++ developers to adapt to module and trait-based code organization. In safety environments, Rust's compiler enforces many rules and guidelines, reducing the need for manual reviews while still requiring oversight.
Rust's ability to integrate C/C++ libraries aids in reusing existing code during large-scale migrations. Alternatives like porting code are emerging, with initiatives like DARPA's TRACTOR [DARPA 2025] project offering promising, though ambitious, solutions.

Developer Experience and Adoption Barriers: Initially, Rust may feel constraining compared to C++, but developers often come to appreciate its enforced correctness. The main barriers to adoption include transitioning the team to a new language and establishing a safety-compliant environment. While time and motivation can resolve team transitions, achieving compliance for compilers, libraries, and processes is more challenging. However, Rust's ecosystem is steadily evolving to meet these needs.

6. Conclusion

Rust offers a powerful solution for safety-critical systems, combining memory safety, concurrency guarantees, and efficient bare-metal support. While its strict rules and limited certified libraries present challenges, Rust's growing ecosystem and alignment with safety standards make it a

compelling choice for robust, safe and reliable software development.

6.1. *Key findings*

Rust ensures memory and thread safety by eliminating null pointers, memory leaks, and data races through its ownership, borrowing, and lifetime rules. Certified compilers like Ferrocene and HighTec's AURIX comply with IEC 61508 and ISO 26262, though most libraries remain uncertified. Tools such as svd2rust, PACs, and HAL streamline hardware access and enhance portability. Rust supports safe concurrency with threading, async/await, and Rayon for multicore parallelism, all with compiler guarantees. Its "no_std" feature enables bare-metal development, static memory allocation, and RTOS (Real-Time Operating System) integration like FreeRTOS. While Rust's strict rules present a learning curve, they prevent critical errors and ensure robust, safe code.

RUST and the development lifecycle: Some manufacturers [Myklebust et al., 2025] first develop a prototype in Python and later use RUST for the deployable version of the produc.

6.2. *Future work*

Future work should focus on expanding certified libraries, improving tooling for safety-critical systems, enhancing support for bare-metal and RTOS environments, and streamlining Rust's adoption through better training resources. A deeper investigation into structured training programs is needed, particularly in how Rust's safety features are taught compared to C and C++ in safety-critical domains. This includes evaluating existing training materials, identifying gaps, and developing industry-specific courses that emphasize Rust's advantages in preventing memory safety issues and concurrency errors. Advancing Rust's integration with safety standards like the new edition of IEC 61508, the soon-to-be-issued SAE JA1020, and real-time systems will further strengthen its role in safety-critical domains.

6.2. *Acknowledgement*

This work has been supported by two Norwegian Research Council projects: "Integrated Service & Safety Platform," project number 309406. We extend our gratitude to Espen Albrektsen at Sonair for his invaluable contributions. Special thanks to Christof Petig, the leader of SAE JA1020, for his guidance and support of this work.

References

IEC. 2010. IEC 61508:2010 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. International Electrotechnical Commission.

IEC. 2010. IEC 61508-3:2010 Functional Safety – Software Requirements. International Electrotechnical Commission.

IEC. 2010. IEC 61508-7:2010 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems – Part 7: Overview of Techniques and Measures. International Electrotechnical Commission.

ISO. 2018. ISO 26262:2018 Road Vehicles – Functional Safety. 2nd ed. International Organization for Standardization.

European Standard. EN 54 Fire Detection and Fire Alarm Systems Series.

Ferrocene. 2025. 'Ferrocene - Rust for Functional Safety.' Accessed January 31, 2025. https://ferrocene.dev

HighTec. 2025. 'Rust Compiler for Safety-Critical Applications.' Accessed January 31, 2025. https://hightec-rt.com/rust .

Rust Foundation. 2024. 'Announcing the Safety-Critical Rust Consortium.' March 31, 2024. https://rustfoundation.org/media/announcing-the-safety-critical-rust-consortium/.

Rust Embedded Working Group. 2025. The Embedded Rust Book. Accessed January 31, 2025. https://docs.rust-embedded.org/book/.

QEMU. 2025. 'About QEMU.' Accessed January 31, 2025. https://www.qemu.org/docs/master/about/index.html.

SAE International. 2025. JA1020: Guidelines for Software Development in Safety-Critical Systems. Accessed February 17, 2025. https://www.sae.org/standards/content/ja1020/.

Elektrobit. 2025. 'EB Corbos Linux for Safety Applications.' Accessed January 31, 2025. https://www.elektrobit.com/products/ecu/eb-corbos/linux-for-safety-applications/.

DARPA. 2025. 'Translating All C to Rust.' Accessed January 31, 2025. https://www.darpa.mil/research/programs/translating-all-c-to-rust.

AdaCore. 2025. 'GNAT Pro Rust.' Accessed January 31, 2025. https://www.adacore.com/gnatpro-rust.

Embassy RS. 2025. Embassy Embedded Framework for Rust. GitHub. Accessed January 31, 2025. https://github.com/embassy-rs/embassy.

Infineon. 2025. 'Microcontroller Solutions.' Accessed January 31, 2025. https://www.infineon.com/cms/en/product/microcontroller/.

The Register. 2024. 'Google's Rust Push for Safety-Critical Software.' March 31, 2024. Accessed January 31, 2025. https://www.theregister.com/2024/03/31/rust_google_c/.

STMicroelectronics. 2025. 'STM32H745/755 Microcontrollers.' Accessed January 31, 2025. https://www.st.com/en/microcontrollers-microprocessors/stm32h745-755.html.

Tokio. 2025. 'Tokio - An Asynchronous Runtime for Rust.' Accessed January 31, 2025. https://tokio.rs/.

Tonic. 2025. 'Tonic - A Rust Implementation of gRPC.' Accessed January 31, 2025. https://docs.rs/tonic/latest/tonic/.

Myklebust, Thor, Tor Stålhane, and Dorthea Mathilde Kristin Vatn. The AI Act and The Agile Safety Plan. Cham, Switzerland: Springer Nature, 2025. https://doi.org/10.1007/978-3-031-80504-2

ISO/IEC 24772-1:2024 Programming languages — Avoiding vulnerabilities in programming languages Part 1: Language-independent catalogue of vulnerabilities.

A. Okutan, S. Merten, C. C. Michael and B. Ryjikov, "Leveraging RAG-LLM to Translate C++ to Rust," 2024 International Conference on Assured Autonomy (ICAA), Nashville, TN, USA, 2024, pp. 102-105, doi: 10.1109/ICAA64256.2024.00024.

L. Xia, Y. Wu and B. Hua, "Rustcheck: Safety Enhancement of Unsafe Rust via Dynamic Program Analysis," 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), Chiang Mai, Thailand, 2023, pp. 871-872, doi: 10.1109/QRS-C60940.2023.00102.