233. COMPARATIVE ANALYSIS OF SAFETY-PERFORMANCE TRADE-OFFS IN RUST AND C++ FOR EMBEDDED SYSTEMS

Kuzmitski M.S., Virbal A.E.
Belarusian State University of Informatics and Radioelectronics
Minsk, Republic of Belarus

Shaputsko A.V. – Lecturer, Master of Arts (Philology)

This paper compares Rust and C++ regarding safety and performance trade-offs. Rust's ownership model eliminates memory-safety vulnerabilities without performance overhead, demonstrating 50 % reduction in memory-safety bugs. Rust emerges as a secure programming standard while C++ remains relevant for legacy systems.

For decades, C and C++ have served as the foundation of systems programming due to their granular control over hardware and minimal runtime overhead. However, this manual approach to memory management has historically been the primary vector for cyber threats. According to the Microsoft Security Response Center, approximately 70 % of all security vulnerabilities addressed in annual updates are attributed to memory-safety issues [1]. The data is represented visually in Figure 1.

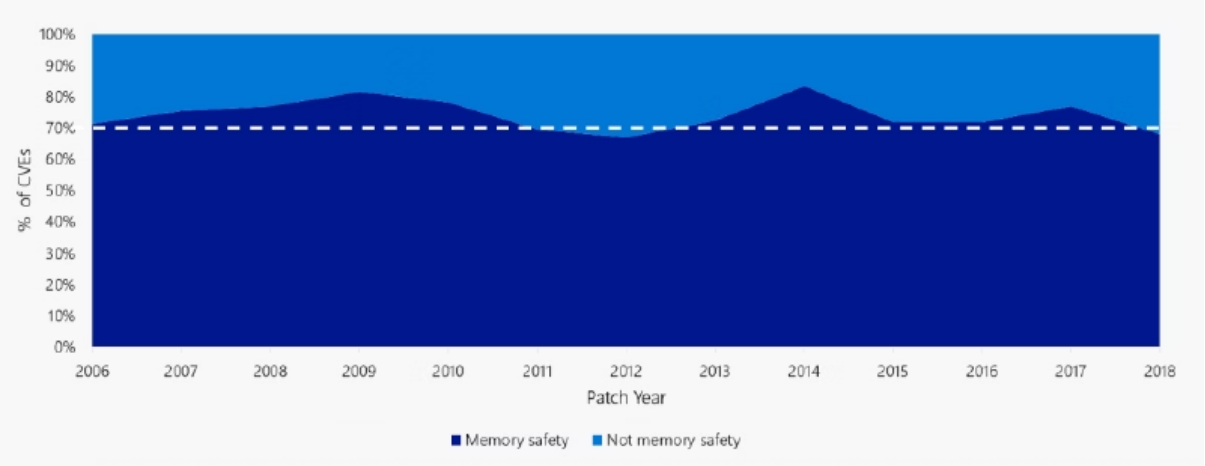

Figure 1 – Microsoft vulnerabilities graph

In C++, the responsibility for memory integrity rests entirely on the programmer, making large-scale embedded projects susceptible to human error, such as buffer overflows and use-after-free bugs.

Rust emerges as a compelling alternative, offering compile-time guarantees that eliminate these classes of errors through a strict borrow checker and ownership model, all without the need for a performance-heavy garbage collector.

The adoption of memory-safe languages has transitioned from a technical debate to a matter of national security. In 2022, the U.S. National Security Agency issued a report recommending organizations to transit away from memory-unsafe languages like C++ toward safer alternatives to protect critical infrastructure [2]. This directive was further solidified in 2024 by the U.S. Office of the National Cyber Director. Their technical report “Back to the Building Blocks” identifies the choice of programming language as a high-stakes architectural decision [3]. The ONCD emphasizes that using memory-safe languages like Rust provides a foundational security layer that significantly reduces the attack surface from the moment code is written [3].

The theoretical benefits of Rust are supported by empirical evidence from global technology leaders. Google reported that after integrating Rust into the Android Open-Source Project, there was a nearly 50 % reduction in memory-safety vulnerabilities in newly written components [4]. This data confirms that Rust effectively scales within complex, high-performance environments while drastically improving code reliability. Recent 2025 data shows memory-safety vulnerabilities in Android have dropped below 20 % of all vulnerabilities for the first time [5].

Furthermore, developer sentiment reflects a strong preference for Rust’s modern toolchain. Data from the Stack Overflow Developer Survey, the extract of which is presented in Figure 2, indicates that Rust has remained the most admired language for nearly a decade, consistently outperforming C++ in terms of developer satisfaction and perceived reliability [6].

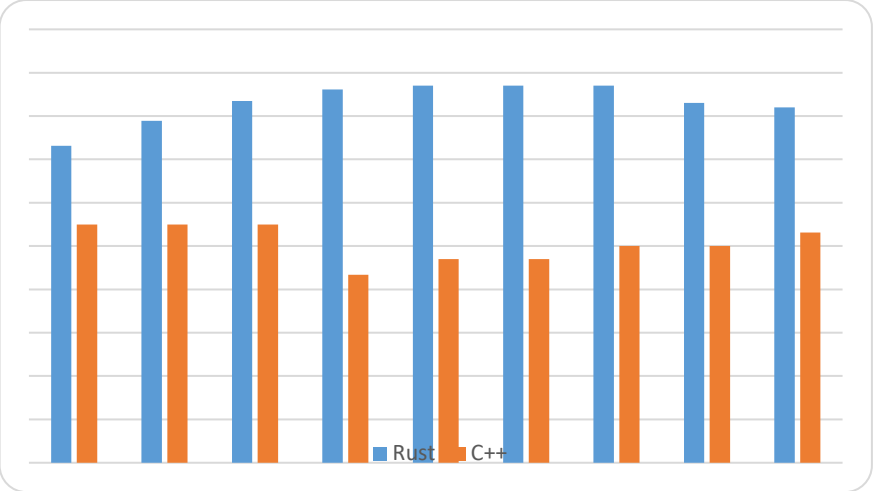

Figure 2 – Admiration level charts

A common concern in embedded development is binary size and execution latency, as these parameters directly influence startup time, energy consumption, and the feasibility of deploying software on microcontrollers with extremely limited resources. Since Rust utilizes the same LLVM compiler infrastructure as many modern C++ compilers, the generated machine code is highly optimized and benefits from decades of backend improvements. Although Rust may incur longer compilation times due to its rigorous safety checks and borrow-checker analysis, the resulting binaries consistently match the deterministic performance required for avionics, automotive, and IoT applications, where predictable timing behavior is essential.

Specific embedded constraints require deeper architectural consideration beyond general benchmarking. In resource-constrained environments, Rust's *no_std* profile allows developers to exclude the standard library entirely, linking directly against hardware registers while retaining type-level safety. This reduces the binary footprint to levels comparable with optimized C++ builds and addresses flash memory limitations common in microcontrollers. Additionally, Rust handles hardware interaction through explicit unsafe blocks, confining potential memory violations to well-defined, isolated sections rather than allowing them to permeate the entire codebase. This contrasts sharply with C++, where unsafe operations are implicit and ubiquitous, making large projects more dependent on manual discipline and extensive code review to maintain integrity.

Binary size concerns are further mitigated through link-time optimization (LTO) and panic strategies such as abort-on-panic, which remove unwinding tables and reduce code size without compromising reliability. These mechanisms ensure that Rust's safety guarantees do not undermine determinism, preserving the strict timing constraints required in embedded systems.

As a result, Rust provides a "safe by default" model where low-level access is explicitly opted into, reducing the attack surface while maintaining full control over hardware. This advantage is especially important in safety-certified domains like automotive ISO 26262 and industrial IEC 61508, where demonstrating the absence of undefined behavior is a regulatory requirement. Interoperability with existing C-based libraries through a well-defined FFI enables gradual adoption and mixed-language systems without rewriting legacy components.

Thus, the trade-off shifts from "safety versus performance" to "safety with guaranteed performance" fundamentally change the risk profile for embedded system architects. Rust represents a significant evolution in systems engineering by combining low-level efficiency with strong safety guarantees. It addresses long-standing sources of vulnerabilities. Supported by the NSA, the White House ONCD, Microsoft, and Google, Rust is increasingly positioned as a new standard for secure embedded development, while C++ remains relevant primarily for legacy maintenance and specialized domains where mature tooling remains decisive.

References:

1. Microsoft Security Response Center. A proactive approach to more secure code. – 2019. – URL: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (date of access: 23.03.2026).

2. National Security Agency. Software Memory Safety : Cybersecurity Information Sheet. – 2022. – URL: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/1/CSI_SOFTWARE_MEMORY_SAFETY.PDF (date of access: 23.03.2026).

3. Office of the National Cyber Director. Back to the Building Blocks: A Path Toward Secure and Measurable Software 2024. – URL: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf (date of access: 23.03.2026).

4. Google. Google Security Blog. Memory safe languages in Android 13. – 2022. – URL: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html (date of access: 23.03.2026).

5. Google. Google Security Blog. Rust in Android: move fast and fix things. – 2025. – URL: https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html (date of access: 23.03.2026).

6. Stack Overflow. Developer Survey Results. – 2025. – URL: https://survey.stackoverflow.co/ (date of access: 23.03.2026).