IAES Journal of Electrical and Electronics Engineering
Vol. 11, No. 2, December 2024, pp. 38~42
ISSN: 2338-7432, DOI: 10.11591/jeee.v11i2.pp38-42

The rise of rust and memory-safe programming: a review of secure software development, performance, and systems reliability

Laksana Talenta Ahmad[1], Fathur Rahmawanto[2]
[1]Embedded Systems and Power Electronics Research Group, Yogyakarta, Indonesia
[2]Institute of Advanced Engineering and Science, Yogyakarta, Indonesia

Article Info

Article history:

Received Apr 23, 2022
Revised Nov 26, 2022
Accepted Mar 24, 2023

Keywords:

Buffer overflow
Embedded systems
Memory safety
Rust
Secure software development
Software reliability
Systems programming

ABSTRACT

The increasing complexity of software infrastructure has intensified concerns about memory-related vulnerabilities, particularly in applications developed using C and C++. Rust has emerged as a systems programming language that seeks to combine low-level performance with stronger memory and concurrency guarantees. This review examines the principles underlying Rust's ownership, borrowing, and lifetime mechanisms, and evaluates their implications for secure software development. Particular attention is given to buffer overflows, use-after-free errors, data races, and other forms of memory corruption that frequently affect conventional systems software. The review also compares Rust with C and C++ in terms of memory management, performance, development complexity, and practical implementation. Evidence from operating systems, embedded platforms, microcontrollers, and software verification research is examined to assess Rust's suitability for cloud computing, Internet of Things applications, and reliability-sensitive systems. The findings indicate that Rust can substantially reduce memory-safety risks in safe code without requiring garbage collection, although unsafe blocks, foreign-function interfaces, and third-party dependencies remain important security boundaries. The review concludes by identifying migration strategies, verification requirements, and research priorities for dependable Rust-based software ecosystems.

This is an open access article under the CC BY-SA license.

1. INTRODUCTION

Modern software systems increasingly depend on complex interactions between operating systems, network services, embedded controllers, and cloud infrastructure. These environments require programming languages that can deliver computational efficiency while maintaining strong protection against software defects and security attacks. C and C++ have historically dominated systems programming because they provide direct access to memory, hardware resources, and low-level operating system interfaces. However, their flexibility also places substantial responsibility on developers to manage memory allocation, pointer operations, and object lifetimes correctly. Memory-related programming errors therefore remain an important concern in the development of dependable software [1]–[3].

Rust has attracted attention as an alternative because it combines systems-level control with compile-time enforcement of important safety properties. Its ownership and borrowing mechanisms are designed to prevent invalid memory access and unsafe aliasing before a program is executed. Unlike garbage-collected languages, Rust normally manages memory through deterministic ownership rules without requiring a continuously operating garbage collector. This design is particularly relevant to applications where predictable resource management and low-level hardware interaction are important. Nevertheless, the practical effectiveness of Rust depends on implementation discipline, library quality, and the management of code that operates outside its safe abstractions [4]–[6].

The central research problem addressed in this review is the tension between the performance advantages of traditional systems languages and the security risks associated with manual memory management. Existing approaches such as static analysis, runtime sanitizers, and defensive programming can reduce these risks, but they do not provide the same compile-time guarantees as Rust's safe programming model. At the same time, adopting Rust introduces challenges involving developer expertise, legacy integration, embedded hardware support, and verification of unsafe components. Therefore, this review evaluates Rust as a practical approach to improving software security and reliability rather than assuming that the language automatically eliminates every vulnerability. Its contribution is a comparative synthesis of language-level protection, performance evidence, and deployment considerations across modern computing environments.

2. RUST MEMORY-SAFETY ARCHITECTURE

Rust's principal safety mechanism is its ownership system, which assigns each allocated value a clearly defined owner responsible for its lifetime. When ownership is transferred, the previous owner can no longer access the transferred value through the original binding. Borrowing allows functions and other program components to access data without necessarily transferring ownership. The compiler checks borrowing rules and lifetimes to prevent references from remaining valid after the underlying data has been destroyed [1], [4]. These mechanisms provide a foundation for deterministic memory management without depending on a garbage collector.

The borrowing model also restricts unsafe combinations of shared access and modification. Multiple immutable references can access the same value, whereas mutable access requires exclusive borrowing under the relevant rules. This restriction reduces the possibility of data races and invalid pointer aliasing in safe Rust programs. Formal research has examined the theoretical foundations of these mechanisms and demonstrated how ownership-based reasoning can support memory-safe abstractions [1], [4], [5]. Nevertheless, the guarantees apply to code that respects Rust's safety model, rather than automatically validating every external library or low-level operation.

Rust also provides an unsafe mechanism for operations that require capabilities beyond the guarantees of its safe type system. Such operations are necessary for hardware access, foreign-function interfaces, operating system development, and certain optimized data structures. However, unsafe code transfers additional responsibility to developers because the compiler cannot establish every required memory invariant. Empirical investigations have identified vulnerabilities associated with unsafe blocks, unsound library abstractions, and incorrect implementations of low-level interfaces [3], [6], [7]. Consequently, a reliable Rust project should minimize unsafe code and subject every unsafe abstraction to explicit review and testing.

3. COMPARISON OF RUST, C, AND C++

Rust, C, and C++ provide different approaches to memory management, software abstraction, and system-level performance. C offers direct pointer manipulation and manual allocation, making it suitable for compact firmware, kernels, and hardware-oriented applications. C++ adds object-oriented and generic programming facilities, together with resource-management techniques such as RAII and smart pointers. Rust integrates ownership and borrowing rules into its type system, allowing many invalid memory operations to be rejected during compilation. These differences influence not only vulnerability exposure but also development practices and the complexity of maintaining large software projects [2], [8], [9].

Table 1 illustrates that Rust's main distinction is not simply its syntax or programming paradigm, but the enforcement of memory-related constraints during compilation. C and C++ can implement safe systems when developers apply disciplined allocation, pointer validation, and object-lifetime management. However, these practices rely heavily on developer consistency and external verification tools. Rust moves part of this responsibility into the language and compiler, although unsafe code and interoperability boundaries still require careful engineering. Therefore, Rust should be viewed as a stronger default safety framework rather than an unconditional replacement for all C and C++ applications [2], [8], [10].

Performance remains an important consideration when selecting a systems programming language. Rust does not require garbage collection for ordinary ownership-based memory management, which helps preserve predictable execution and resource control. However, benchmarking studies have reported that performance differences depend on workload characteristics, compiler optimization, bounds checks, data structures, and implementation choices. Zhang et al. identified measurable overhead in several Rust benchmarks relative to C, while other comparisons have found broadly competitive execution performance under different workloads [8], [9]. These findings indicate that performance should be evaluated using representative application benchmarks rather than general assumptions about one language being universally faster.

Table 1. Comparative characteristics of Rust, C, and C++

Evaluation aspect	C	C++	Rust
Memory management	Primarily manual	Manual and RAII-based abstractions	Ownership-based and deterministic
Pointer safety	Largely developer-managed	Depends on abstraction and usage	Checked references in safe code
Buffer-bound protection	Requires explicit checks or tools	Requires explicit checks or abstractions	Bounds-checked safe indexing
Use-after-free prevention	Developer responsibility	Reduced through RAII, not eliminated	Prevented in safe code by ownership rules
Data-race protection	Limited language-level protection	Limited language-level protection	Compile-time restrictions for safe concurrency
Runtime overhead	Generally low	Generally low	Generally low; some checks and abstractions have costs
Hardware access	Mature and extensive	Mature and extensive	Available through embedded and unsafe abstractions
Learning complexity	Moderate	High	High, particularly ownership and lifetimes
Legacy ecosystem	Extensive	Extensive	Growing, with interoperability requirements

4. PREVENTION OF BUFFER OVERFLOW AND MEMORY CORRUPTION

Buffer overflow occurs when a program writes beyond the allocated boundary of a memory region. Such errors can corrupt adjacent data, alter program execution, or create opportunities for unauthorized code execution. Use-after-free, double-free, dangling references, and invalid pointer arithmetic represent additional memory-corruption risks commonly associated with manual memory management. Rust reduces these risks by enforcing bounds checks on safe indexing and by restricting references to valid ownership and lifetime relationships [2]–[4].

The prevention mechanism is particularly important because memory corruption can occur even when individual software components appear functionally correct. In C and C++, developers must ensure that pointer arithmetic, buffer lengths, and allocation sizes remain valid across all execution paths. Rust's safe abstractions make many such invalid operations difficult or impossible to express without explicitly entering unsafe code. Research on translating C programs into safer Rust demonstrates that migration can improve memory-safety properties, although the process requires careful treatment of pointer-heavy code and legacy interfaces [10], [11]. This suggests that incremental migration and targeted replacement of high-risk components can be more practical than rewriting an entire software system.

Rust's protection should not be interpreted as complete immunity to memory corruption. Unsafe blocks, incorrect foreign-function interfaces, unsound generic abstractions, and defects in trusted libraries can undermine the expected guarantees. Studies of real-world Rust vulnerabilities have documented memory-safety problems that arise from these boundaries and from mistakes in the implementation of unsafe components [3], [6], [7]. Effective prevention therefore requires code review, dependency auditing, fuzz testing, static analysis, and clear documentation of safety assumptions. The combination of language-level guarantees and engineering controls provides a more realistic foundation for secure software development.

5. OPERATING SYSTEMS AND EMBEDDED SYSTEMS

Operating systems are among the most important application domains for Rust because kernels and device drivers frequently manipulate memory, interrupts, hardware registers, and concurrent processes. These operations demand low-level control, but defects in privileged software can affect the reliability of an entire computing platform. Research on Rust-based kernel development has demonstrated that useful operating system abstractions can be implemented while limiting the amount of unsafe code that must be trusted [12], [13]. Formal verification research further provides methods for reasoning about Rust programs and the correctness of their low-level abstractions [14], [15].

Embedded systems introduce additional constraints because microcontrollers commonly operate with limited memory, processing capacity, and energy resources. The Tock operating system demonstrates how Rust can support memory protection, fault isolation, and multiprogramming on resource-constrained microcontrollers [12]. Earlier development experience also identified practical challenges involving event-driven programming, hardware resource sharing, and asynchronous operations [16]. These findings show that the adoption of Rust in embedded platforms requires suitable abstractions for interrupts, peripheral access, and real-time execution rather than relying exclusively on general-purpose software libraries.

Performance evaluation on the ESP32 microcontroller provides further evidence of Rust's relevance to IoT development. Plauska *et al.* [17] compared C/C++, Rust, TinyGo, and MicroPython using signal-processing and security-related workloads, including FFT, CRC, SHA, and digital filters. Their results showed that C/C++ was generally fastest, while Rust and TinyGo remained competitive in several tested workloads. A separate study of a Rust-based embedded operating system investigated low-latency execution and reported interrupt handling at approximately 60 microseconds under its experimental configuration [18]. These results support the technical feasibility of Rust for selected embedded applications, although platform support and real-time behavior must still be evaluated for each target device.

6. ADOPTION IN CLOUD COMPUTING, IOT, AND CRITICAL SYSTEMS

Cloud computing provides an important opportunity for Rust because network services, infrastructure agents, virtualization components, and distributed applications require both performance and dependable resource management. Rust's ownership model can reduce memory-related defects in components that handle concurrent requests, network buffers, and long-running processes. Its lack of mandatory garbage collection can also be useful where predictable resource behavior is desirable. However, cloud reliability depends on more than programming-language safety, including isolation architecture, authentication, access control, dependency management, and operational monitoring. Consequently, Rust should be considered one component of a broader cloud security strategy rather than a standalone security solution.

In IoT systems, Rust offers potential advantages for sensor gateways, communication stacks, embedded controllers, and edge-processing applications. These devices frequently operate with limited resources while interacting with networks and physical environments that may expose them to security threats. The combination of compile-time memory checks and efficient native execution is therefore attractive for firmware that must remain responsive and maintainable. Nevertheless, hardware abstraction libraries, peripheral drivers, and platform-specific support remain important adoption factors. The ESP32 performance study and Tock research demonstrate that Rust is technically applicable to constrained platforms, while also identifying the need for careful evaluation of execution performance and development maturity [12], [13], [16]–[18].

Critical systems require an even more rigorous assessment because software defects may affect industrial control, transportation, energy infrastructure, and other safety-sensitive operations. Rust can reduce selected memory-related failure modes, but memory safety alone does not prove functional correctness, fault tolerance, or compliance with safety standards. Developers must still validate timing behavior, concurrency assumptions, hardware interactions, and failure-recovery mechanisms. Formal verification tools such as Aeneas illustrate how Rust programs can be analyzed for functional properties, while foundational research emphasizes the importance of proving the soundness of low-level abstractions [1], [14], [15]. Therefore, Rust adoption in critical systems should be accompanied by systematic verification, traceable requirements, and domain-specific certification activities.

7. RESEARCH CHALLENGES AND FUTURE DIRECTIONS

Despite its advantages, Rust adoption faces several technical and organizational challenges. Developers accustomed to C and C++ may require substantial training to understand ownership, borrowing, lifetimes, and the design of safe abstractions. Existing systems may also depend on large legacy codebases, proprietary libraries, and hardware interfaces that cannot be replaced immediately. Research on C-to-Rust translation and Rust safety analysis indicates that migration requires attention to semantic equivalence, unsafe code, and the verification of external interfaces [3], [6], [7], [10], [11]. Future development should therefore prioritize incremental migration strategies, improved static analysis, and better tools for auditing unsafe code.

Another important research direction is the systematic evaluation of Rust in real-time and safety-critical environments. Performance comparisons should include execution time, memory consumption, energy use, binary size, and worst-case response time under realistic workloads. Embedded and cloud applications also require stronger evidence about dependency security, interoperability, and long-term maintenance. Further studies should examine how Rust-based components behave under hardware faults, adversarial inputs, and resource exhaustion. These efforts would help establish more reliable engineering guidelines for adopting Rust across heterogeneous computing platforms [19].

8. CONCLUSION

Rust represents a significant development in systems programming because it integrates compile-time memory-safety mechanisms with low-level resource control. Compared with C and C++, its ownership

and borrowing model can prevent many common memory errors, including use-after-free, invalid references, and unsafe data sharing in safe code. Existing research demonstrates its applicability to operating systems, embedded platforms, microcontrollers, and performance-sensitive software. Nevertheless, Rust does not eliminate all security risks because unsafe blocks, external interfaces, third-party dependencies, and logical defects remain potential sources of failure. Its adoption in cloud computing, IoT, and critical systems should therefore be supported by rigorous testing, dependency governance, performance evaluation, and formal verification where appropriate. Overall, Rust offers a promising foundation for more secure and reliable software development, particularly when its language-level guarantees are combined with disciplined engineering and system-level assurance.

REFERENCES

[1] R. Jung, J. H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the rust programming language," in *Proceedings of the ACM on Programming Languages*, Jan. 2018, pp. 1–34. doi: 10.1145/3158154.

[2] R. Jung, J. H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in rust," *Communications of the ACM*, vol. 64, no. 4, pp. 144–152, Apr. 2021, doi: 10.1145/3418295.

[3] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? An in-depth study with all rust CVEs," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–25, Jan. 2022, doi: 10.1145/3466642.

[4] R. Jung, H. H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: An aliasing model for rust," in *Proceedings of the ACM on Programming Languages*, Jan. 2020, pp. 1–32. doi: 10.1145/3371109.

[5] J. Yanovski, H. H. Dang, R. Jung, and D. Dreyer, "GhostCell: separating permissions from data in rust," in *Proceedings of the ACM on Programming Languages*, Aug. 2021, pp. 1–30. doi: 10.1145/3473597.

[6] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA: ACM, Jun. 2020, pp. 763–779. doi: 10.1145/3385412.3386036.

[7] P. Liu, G. Zhao, and J. Huang, "Securing unsafe rust programs with xrust," in *Proceedings - International Conference on Software Engineering*, New York, NY, USA: ACM, Jun. 2020, pp. 234–245. doi: 10.1145/3377811.3380325.

[8] Y. Zhang, Y. Zhang, G. Portokalidis, and J. Xu, "Towards understanding the runtime performance of rust," in *ACM International Conference Proceeding Series*, New York, NY, USA: ACM, Oct. 2022, pp. 1–6. doi: 10.1145/3551349.3559494.

[9] W. Bugden and A. Alahmar, "The safety and performance of prominent programming languages," *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 5, pp. 713–744, May 2022, doi: 10.1142/S0218194022500231.

[10] N. Popescu, Z. Xu, S. Apostolakis, D. I. August, and A. Levy, "Safer at any speed: Automatic context-aware safety enhancement for Rust," in *Proceedings of the ACM on Programming Languages*, Oct. 2021, pp. 1–23. doi: 10.1145/3485480.

[11] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to safer rust," in *Proceedings of the ACM on Programming Languages*, Oct. 2021, pp. 1–29. doi: 10.1145/3485498.

[12] A. Levy *et al.*, "Multiprogramming a 64 kB computer safely and efficiently," in *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*, New York, NY, USA: ACM, Oct. 2017, pp. 234–251. doi: 10.1145/3132747.3132786.

[13] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The case for writing a kernel in rust," in *Proceedings of the 8th Asia-Pacific Workshop on Systems, APSys 2017*, New York, NY, USA: ACM, Sep. 2017, pp. 1–7. doi: 10.1145/3124680.3124717.

[14] X. Yin *et al.*, "SafeOSL: Ensuring memory safety of C via ownership-based intermediate language," *Software - Practice and Experience*, vol. 52, no. 5, pp. 1114–1142, May 2022, doi: 10.1002/spe.3057.

[15] S. Ho and J. Protzenko, "Aeneas: Rust verification by functional translation," in *Proceedings of the ACM on Programming Languages*, Aug. 2022, pp. 711–741. doi: 10.1145/3547647.

[16] A. Levy *et al.*, "Ownership is theft: Experiences building an embedded OS in rust," in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015*, New York, NY, USA: ACM, Oct. 2015, pp. 21–26. doi: 10.1145/2818302.2818306.

[17] I. Plauska, A. Liutkevičius, and A. Janavičiūtė, "Performance evaluation of C/C++, MicroPython, rust and TinyGo programming languages on ESP32 microcontroller," *Electronics (Switzerland)*, vol. 12, no. 1, p. 143, Dec. 2023, doi: 10.3390/electronics12010143.

[18] I. Culic, A. Vochescu, and A. Radovici, "A low-latency optimization of a rust-based secure operating system for embedded devices," *Sensors*, vol. 22, no. 22, p. 8700, Nov. 2022, doi: 10.3390/s22228700.

[19] K. A. S. Oey, C. Kwee, W. Chua, G. Z. Nabiilah, and Rojali, "Rust's memory safety model: An evaluation of its effectiveness in preventing common vulnerabilities," *Procedia Computer Science*, vol. 227, pp. 119–127, 2023, doi: 10.1016/j.procs.2023.10.509.