UDC 004.451.36

A REVOLUTION IN MEMORY SAFETY: A COMPARATIVE STUDY OF RUST, GO, AND MODERN C++ WITH CONTEMPORARY MEMORY PROTECTION MECHANISMS

Ustimchuk R. V., Shubaba M. I.

Belarusian State University of Informatics and Radioelectronics, Minsk, Republic of Belarus

Alekseev V.F. – PhD, assistant professor, associate professor of the department of ICSD,
Lasarenko A.M. – Senior Lecturer at the Department of Foreign Languages

Annotation. This paper examines the problem of memory safety in systems programming and analyzes the evolution of approaches to addressing it in modern programming languages. A comparative study of memory management models and protection mechanisms implemented in Rust, Go, and modern C++ is conducted in terms of reliability, performance, and vulnerability prevention.

Keywords. Memory safety, systems programming, Rust, Go, C++, memory management, software vulnerabilities

***Introduction*.** In recent decades, software security has become one of the key tasks in the field of systems programming, since a significant portion of critical vulnerabilities is directly related to memory management errors [1,2]. Buffer overflows, use-after-free, double freeing of resources, and data races remain the main causes of failures and compromise of software systems, including operating systems, network services, and embedded platforms [1,2,3].

Traditional systems programming languages such as C and C++ provide the developer with a high level of control over resources; however, they place full responsibility for correct memory handling on the developer, which significantly increases the likelihood of errors in complex and large-scale projects [2,3]. In this regard, a stable trend has formed in the software industry toward transitioning to languages and technologies that ensure memory safety at the level of the language, compiler, and standard library [1,3].

The purpose of this work is to analyze modern approaches to ensuring memory safety and to conduct a comparative study of Rust, Go, and modern C++ in terms of the protection mechanisms used, their effectiveness, and their impact on the performance of software systems.

Main part. Systems programming languages such as C and C++ historically provided the developer with full control over memory, which made it possible to achieve maximum performance and precise resource management. However, this approach places all responsibility for correct memory handling on the programmer. Even with a high level of qualification, the probability of errors remains significant, especially in large and long-lived codebases [2,3].

In recent years, a steady shift has formed in the software industry toward the use of programming languages that provide built-in guarantees of memory safety [1,3]. This process is often characterized as a revolution in the field of memory safety, since it is aimed at eliminating entire classes of vulnerabilities at the level of the language, compiler, and standard library, rather than through external analysis and testing tools [1,2].

One of the most radical approaches is the model implemented in the Rust language. Memory safety here is ensured through a strict system of ownership, borrowing, and lifetimes of objects, verified by the compiler at the stage of transforming the program's source code into machine code or bytecode understandable to the processor [3,4,7]. Each value has a single owner, and borrowing rules prevent simultaneous unsafe access to data. As a result, errors such as use-after-free, double freeing of memory, and data races are impossible in safe Rust code [3,7].

A fundamentally important feature of Rust is the transfer of memory management correctness checks from the execution stage to the compilation stage [3,4]. This makes it possible to achieve high reliability without using a garbage collector and the associated overhead. When low-level operations are required, the language allows the use of unsafe blocks.

```
unsafe{
    //unsafe code block.
}
```

Figure 1 – Designation of an unsafe code block in Rust

However, they are explicitly marked and isolated, which reduces the risk of error propagation throughout the entire codebase [3,7].

The Go language implements an alternative approach based on automatic memory management using a garbage collector [8].

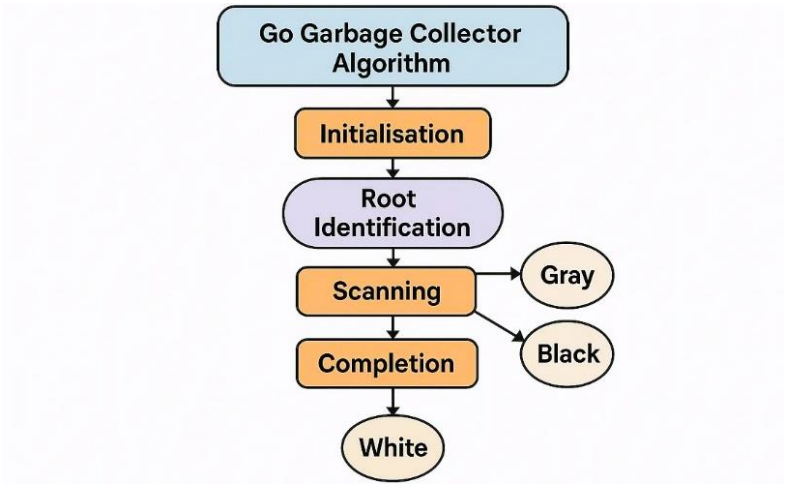

Figure 2 – Garbage collector algorithm in Golang

The developer is freed from the need to manually manage the lifetime of objects, which eliminates a number of errors typical for C and C++. Additional checks of array bounds and references are performed during program execution, which increases the overall robustness of applications [8].

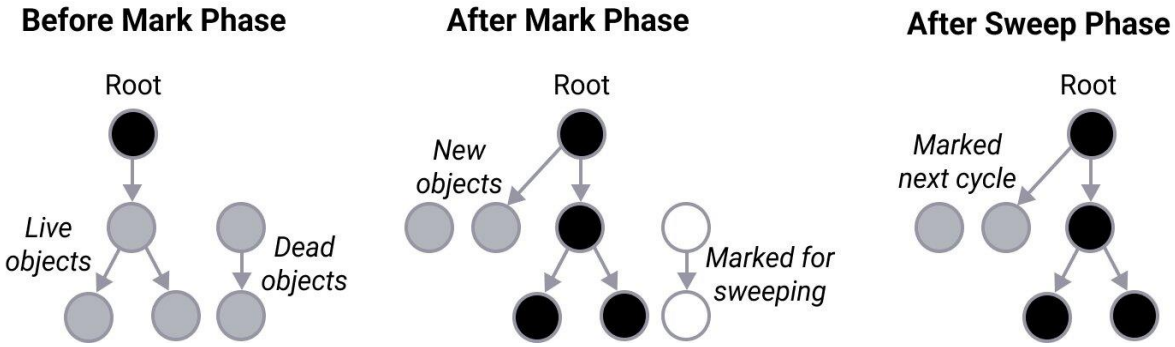

Figure 3 – Principle of garbage collector operation in Golang

The principle of the garbage collector operation is as follows: Mark: locating and marking all reachable objects from a set (for example, in the heap).

Sweep: traversing all objects in the heap, then erasing unreachable ones and returning them to the pool of free memory [8].

Despite the increase in safety, the use of a garbage collector in Go leads to runtime overhead and potentially unpredictable pauses, which may be critical for systems with strict latency requirements [6,8].

In addition, automatic memory management does not prevent logical data races; responsibility for synchronization still remains with the developer [8].

Modern C++ occupies an intermediate position between the classical low-level approach and languages with strict memory safety guarantees.

Mechanisms such as the RAII idiom, smart pointers, standard library containers, and a strict type system have been introduced into the language [2,7]. External static and dynamic analysis tools are also used, including sanitizers and formal analyzers [2].

However, the key limitation of C++ remains the absence of mandatory memory safety guarantees [2,7].

The use of protective mechanisms depends on developer discipline and project practices. This leads to the fact that even modern C++ projects remain vulnerable to classic memory management errors, especially when working with legacy code [2,3].

A comparative analysis of practical cases shows that transitioning to languages with guaranteed memory safety can significantly reduce the number of critical vulnerabilities.

Large technology companies, including Google, Microsoft, Cloudflare, and Discord, report a reduction in the number of security incidents after introducing Rust into system components [1,3,5,6].

From the performance perspective, Rust demonstrates results comparable to C++, and in a number of scenarios provides more stable latency due to the absence of a garbage collector [4,6,7]. Go, in turn, shows sufficient performance for most server and network applications; however, it is inferior to Rust and C++ in real-time systems and high-frequency workloads [6,8].

Conclusion. Thus, modern programming languages represent different philosophies for solving the problem of memory safety. Rust is focused on formal guarantees and error prevention at the compilation stage [3,4], Go – on simplifying development and automating resource management [8], and C++ – on maximum control and flexibility provided that the developer has high qualifications [2,7].

The revolution in the field of memory safety reflects a general shift in the industry toward preventing vulnerabilities at early stages of the software life cycle [1,2].

The choice of programming language should be determined by requirements for security, performance, and scalability, as well as acceptable trade-offs between control and reliability [7,8].

References

1. Rebert, A. Safer with Google: Advancing Memory Safety [Electronic resource] / A. Rebert, C. Carruth, J. Engel, A. Qin // Google Security Blog. – 2024. – Access mode: https://security.googleblog.com/2024/10/safer-with-google-advancing-memory.html. – Access date: 13.01.2026.

2. Gregory, J. Can memory-safe programming languages kill 70% of security bugs? [Electronic resource] / J. Gregory // IBM News – Security. – 2023. – Access mode: https://www.ibm.com/security/news/security-bugs. – Access date: 22.01.2026.

3. MSRC Team. Why Rust for safe systems programming [Electronic resource] // Microsoft Security Response Center Blog. – 2019. – Access mode: https://msrc.microsoft.com/blog/2019/07/why-rust-for-safe-systems-programming/. – Access date: 26.01.2026.

4. Yadav, A. Rust vs. C++ Performance: Analyzing Safe and Unsafe Implementations in System Programming [Electronic resource] / A. Yadav // ResearchGate. – 2025. – Access mode: https://www.researchgate.net/publication/388221456_Rust_vs_C_Performance_Analyzing_Safe_and_Unsafe_Implementations_in_System_Programming. – Access date: 03.02.2026.

5. Wu, Y. How we built Pingora, the proxy that connects Cloudflare to the Internet [Electronic resource] / Y. Wu, A. Hauck // Cloudflare Blog. – 2022. – Access mode: https://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/. – Access date: 10.02.2026.

6. Howarth, J. Why Discord is switching from Go to Rust [Electronic resource] / J. Howarth // Discord Engineering Blog. – 2020. – Access mode: https://discord.com/blog/why-discord-is-switching-from-go-to-rust. – Access date: 15.02.2026.

7. Apriorit. Rust vs C++ Comparison [Electronic resource] // Apriorit – Technical Blog. – 2025. – Access mode: https://www.apriorit.com/tech-blog/rust-vs-cpp-comparison. – Access date: 15.02.2026.

8. JetBrains. Rust vs Go: Which one to choose in 2025 [Electronic resource] // The RustRover Blog (JetBrains). – 2025. – Access mode: https://blog.jetbrains.com/rust/2025/06/rust-vs-go-which-one-to-choose-in-2025/. – Access date: 12.02.2026.