Memory-Safe Programming Languages: A Focus on Rust

Larson Carter

June 30, 2025

Abstract

Memory-safety vulnerabilities remain a leading cause of software exploits. This paper introduces the core principles of memory-safe languages and explains how Rust's ownership and borrowing rules provide compile-time guarantees that prevent buffer overflows, use-after-free errors, and data races. We contrast Rust's approach with manual memory management in C/C++ and with the garbage collectors used by Java and Go. Real-world deployments and formal results demonstrate that Rust can deliver C-like performance while eliminating broad classes of security issues. We conclude with discussion of adoption trends and remaining challenges on the path toward a predominantly memory-safe software ecosystem.

Contents

1 Introduction: Memory Safety and Its Importance

Memory safety ensures that a program never reads from or writes to invalid locations in memory. Violations lead to buffer overflows, use-after-free errors, and other *undefined behaviours* that can be exploited by attackers. Industry post-mortems reveal that roughly 70 % of high-severity vulnerabilities in large C/C++ codebases stem from memory-safety bugs [1, 2]. Consequently, governments and vendors now advocate *memory-safe programming languages* that eliminate these classes of bugs by design [3].

Rust is a modern systems language that promises C-like performance *and* strong, compile-time memory-safety guarantees. This paper surveys the fundamentals of memory safety, examines Rust's approach in depth, and compares it with C/C++, Java, and Go. We highlight real-world deployments and the latest academic work that validate Rust's safety claims.

2 Memory Safety Issues in C/C++

C and C++ give developers manual control over allocation, de-allocation, and pointer arithmetic. While this yields maximal flexibility, it also means the compiler *cannot* enforce spatial or temporal safety. Listing 1 illustrates a classic double-free in C++.

Listing 1: Double-free in C++

```cpp
#include <cstdlib>
int main() {
    int* p = static_cast<int*>(std::malloc(sizeof(int)));
    *p = 42;
    int* q = p;
    std::free(p);
    std::free(q);           // undefined behaviour: double free
}
```

The program compiles but may crash or corrupt the heap at run-time. Static analysers or sanitizers can mitigate such bugs, yet they remain an *optional* and *best-effort* defence.

3 Garbage-Collected Languages: Java and Go

Languages like Java and Go achieve memory safety through a tracing garbage collector (GC) and run-time checks. The GC prevents temporal errors (e.g. use-after-free) by freeing objects only when unreachable; array bounds are checked at run-time to avoid spatial errors. This model is easy to program but incurs GC pauses and additional memory overhead, making Java/Go less suitable for hard real-time or very constrained systems.

4 Rust's Ownership and Borrowing Model

Rust removes the GC yet still guarantees safety by shifting checking to *compile time*.

4.1 Ownership

Every value has a single owner. When the owner goes out of scope, Rust inserts a deterministic `drop` call, freeing the allocation exactly once. Moving ownership invalidates the original handle, preventing double-free and use-after-free.

4.2 Borrowing and Lifetimes

Values can be *borrowed*. Immutable borrows (`&T`) allow many readers; a mutable borrow (`&mut T`) grants one writer and excludes readers. The *borrow checker* proves at compile time that:

- no dangling pointers exist (lifetimes),
- the aliasing rule (many readers *or* one writer) is upheld,
- data races are impossible in safe code.

Listing 2: Ownership prevents double-free

```
fn main() {
    let p = Box::new(42);
    let q = p;      // ownership moves; ‘p‘ is now invalid
    drop(q);        // memory freed once
    // drop(p);     // compile error: value moved
}
```

4.3 Unsafe Blocks

Rust permits escape hatches via the `unsafe` keyword for FFI and low-level manipulation. Such code is explicit and audited; the vast majority of production Rust ($<5\%$) remains safe [4].

5 Performance Comparison

Figure 1 summarises median run-time performance from Bugden & Alahmar's benchmark suite [5]. Rust matches C/C++ speed while Java and Go pay a GC tax.

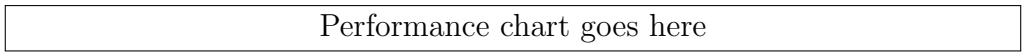

Figure 1: Normalised run-time performance (lower is better).

6 Real-World Deployments

Android (Google) Zero memory-safety vulnerabilities have been reported in new Rust components since 2019, halving Android's overall bug volume [2].

Firefox Stylo (Mozilla) Re-writing the CSS engine in Rust eliminated $\sim$74 % of prior C++ security bug classes while boosting performance [7].

Firecracker (AWS) A micro-VM monitor written in Rust boots VMs in 125 ms and underpins AWS Lambda [6].

Windows and Linux Kernels Both projects have accepted Rust subsystems or drivers to curb pervasive memory bugs [1].

7 Academic Validation

Rust's guarantees are not merely empirical. The *RustBelt* project gives a machine-checked proof of type-soundness and memory safety for a realistic subset of Rust, including user-defined `unsafe` abstractions [4].

8 Discussion and Future Work

Rust demonstrates that low-level performance and strong safety are not mutually exclusive. Adoption hurdles—primarily the learning curve and ecosystem maturity—are diminishing as tooling, libraries, and formal methods advance. Long-term, a migration from legacy C/C++ to memory-safe languages is forecast for critical infrastructure.

9 Conclusion

Memory-unsafe code accounts for the majority of serious software vulnerabilities. Garbage-collected languages address this but at a cost to control and predictability. Rust offers a third path: compile-time proofs of safety plus C-class performance. Empirical data and formal verification jointly confirm its effectiveness. For systems engineers and researchers alike, Rust represents a viable foundation for secure, high-performance software.

References

[1] Microsoft Security Response Center. A survey of memory safety issues, 2019.

[2] Jeff Vander Stoep and others. Memory safety in Android 13, Google Security Blog, 2022.

[3] Office of the National Cyber Director. Back to the Building Blocks: A Path Toward Memory-Safe Software, 2024.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. *RustBelt: Securing the Foundations of the Rust Programming Language*. POPL 2018.

[5] David Bugden and Firas Alahmar. Performance and safety comparison of six programming languages. Journal of Systems & Software, 2022.

[6] Agache et al. Firecracker: Lightweight virtualization for serverless applications, NSDI 2019.

[7] Mozilla Engineering Blog. Shaping up Stylo, 2017.