MANTECH Publications
International Journal of Programming Languages and Internet of Things
Volume 1, Issue 2, July-December 2025

Memory Safe Language Design & Verification

Varun Sharma[1], Anup Das[2], Meenal Verma[3]
Lecturer[1], Students[2, 3]
Department of Computer Science Engineering
Nalanda Institute of Technology
***Email**: Varuns574@yahoo.com[1], anupdas_99@gmail.com[2]*

ABSTRACT

Memory related bugs are among the most critical sources of software vulnerabilities in modern computing systems. Issues like buffer overflows, dangling pointers, null dereferencing, and data races frequently result in system crashes, unpredictable behavior, and serious security threats. Traditional programming languages such as C and C++ offer high performance but provide little or no built-in protection against unsafe memory operations. Recently, memory safe languages such as Rust, Go, Swift, and managed languages like Java have gained significant attention because they enforce safety guarantees at compile time or runtime. Along with language design, formal verification and static analysis methods are also being adopted to ensure memory correctness. This paper presents a detailed review of memory safe language design principles and verification techniques used to eliminate memory errors. It discusses ownership models, borrow checking, garbage collection, type systems, and formal methods for proving memory safety. A comparative analysis of popular memory safe languages is also presented. The paper also highlights how verification tools and proof assistants are used for checking correctness. Some limitations and future research directions are discussed as well.

KEYWORDS: *Memory safety, Rust, ownership model, formal verification, static analysis, garbage collection, type systems, borrow checker, safe programming languages.*

INTRODUCTION

Memory safety is a fundamental requirement for reliable software systems. Many software vulnerabilities reported every year are due to improper memory handling. Buffer overflow, use-after-free, double free, and pointer arithmetic errors are common in low-level system programs. These problems are very difficult to debug and often go unnoticed until exploited.

Traditional languages like C and C++ give programmers full control over memory, but this control comes with responsibility and risk. Modern computing environments such as cloud platforms, embedded systems, and IoT devices require higher reliability and security. Because of this, memory safe language design has become an important research and development area.

Memory safe languages enforce constraints that prevent illegal memory access either at compile time or runtime. Verification techniques further strengthen this by mathematically proving that programs cannot violate memory rules. This paper explores how language design and verification work together to ensure memory safety.

COMMON MEMORY ERRORS IN SOFTWARE SYSTEMS

Memory errors are among the most frequent reasons for software crashes and security vulnerabilities, especially in systems written in low-level languages where manual memory handling is required. These errors often remain hidden during development and appear only under rare runtime conditions, which makes them very difficult to detect and debug. Understanding these errors is the first step toward designing memory safe languages and verification mechanisms.

Buffer Overflow

A buffer overflow occurs when a program writes more data into a memory buffer than it was allocated to hold. This extra data overwrites adjacent memory locations, which may contain important program data, control information, or return addresses.

Example scenario:

An array of size 10 is declared, but the program writes 15 elements into it.

Impacts:

- Program crash or unpredictable behavior

- Security vulnerabilities (code injection, privilege escalation)
- Corruption of adjacent data structures

Buffer overflows are widely exploited in cyber-attacks and are one of the main reasons memory safety is emphasized in modern language design.

Dangling Pointer (Use-After-Free)

A dangling pointer refers to a pointer that still references a memory location after that memory has been freed. Accessing this memory leads to undefined behavior because the memory may be reassigned for another purpose.

Example scenario:

Memory is allocated, used, freed, and then accessed again through the old pointer.

Impacts:

- Random crashes
- Silent data corruption
- Difficult-to-trace bugs

This error is common in C/C++ programs due to manual memory deallocation using free() or delete.

Null Pointer Dereferencing

This occurs when a program attempts to access memory through a pointer that has not been initialized or is explicitly set to NULL.

Example scenario:

A pointer is declared but never assigned a valid address before use.

Impacts:

- Immediate program termination (segmentation fault)
- System instability in embedded or real-time systems

Modern languages avoid this by disallowing null references or by using safe constructs like Option types (in Rust).

Memory Leak

A memory leak happens when a program allocates memory but fails to release it after use. Over time, this leads to exhaustion of available memory resources.

Example scenario:

A function allocates memory repeatedly inside a loop but never frees it.

Impacts:

- Gradual performance degradation
- Application or system crash due to memory exhaustion
- Serious issue in long-running servers and embedded devices

Garbage collection and ownership models are designed mainly to eliminate memory leaks.

Table 1: Several Types of Memory errors

Error Type	Description	Impact
Buffer Overflow	Writing beyond allocated memory	Security breach, crashes
Dangling Pointer	Accessing freed memory	Undefined behavior
Null Pointer Deref.	Using uninitialized pointer	Program termination
Memory Leak	Memory not released after use	Resource exhaustion
Data Race	Concurrent unsafe memory access	Corrupted data

PRINCIPLES OF MEMORY SAFE LANGUAGE DESIGN

Memory safe languages are not created by adding a few runtime checks; they are built from the ground up with design rules that prevent unsafe memory behavior by default. These principles are embedded into the compiler, type system, and runtime model of the language so that programmers are guided toward writing safe code naturally. The following core principles are commonly seen in modern memory safe languages.

Strong Type System

A strong and expressive type system is the first line of defense against memory misuse. In unsafe languages, pointers can often be cast freely between types, leading to incorrect memory

interpretation and corruption.

Memory safe languages restrict such operations by:

- Disallowing arbitrary pointer casting
- Enforcing type correctness at compile time
- Differentiating between value types and reference types
- Preventing implicit conversions that may cause memory misinterpretation

For example, in Rust and Swift, you cannot simply treat a block of memory as a different type without explicit and safe conversion. This ensures that memory is always accessed in a valid and predictable manner.

Advanced type systems also encode ownership, mutability, and lifetimes into types, making memory rules part of the language syntax itself.

Automatic Memory Management

Manual allocation and deallocation are major sources of memory bugs. To address this, many memory safe languages introduce automatic memory management.

This is achieved using:

- **Garbage Collection (GC):** Used in Java, Go, and C#. The runtime automatically identifies unused memory and reclaims it.
- **Automatic Reference Counting (ARC):** Used in Swift. Memory is released when reference count becomes zero.

Benefits include:

- Prevention of memory leaks
- Elimination of dangling pointers
- Reduced developer burden

However, GC introduces runtime overhead and non-deterministic timing, which is not ideal for real-time systems. This limitation led to alternative approaches like Rust's ownership model.

Ownership and Borrowing

Rust introduced a unique concept where memory safety is guaranteed without garbage collection through **ownership rules**.

Key ideas:

- Every value in memory has a single owner.
- When the owner goes out of scope, memory is automatically freed.
- References to memory are called *borrows*.
- Only one mutable reference OR multiple immutable references are allowed at a time.

This prevents:

- Use-after-free errors
- Data races
- Double free errors

The compiler checks these rules at compile time through the **borrow checker**, eliminating runtime cost while ensuring strict memory safety.

Immutable by Default

Immutability is a powerful concept that prevents accidental modification of memory. In many memory safe languages, variables are immutable unless explicitly declared mutable.

Advantages:

- Prevents unintended side effects
- Simplifies reasoning about program behavior
- Avoids race conditions in concurrent programs
- Encourages functional programming practices

For example, in Rust and functional languages, immutability is default behavior. This design forces programmers to consciously allow mutation only when necessary, reducing bugs.

Bound Checking

Out-of-bounds memory access is one of the most common causes of vulnerabilities. Memory

safe languages incorporate bound checking either at compile time or runtime.

This includes:

- Checking array and slice indices before access
- Preventing pointer arithmetic beyond allocated memory
- Using safe container abstractions instead of raw arrays

Although runtime bound checks may introduce small overhead, modern compilers optimize many of these checks away when safety can be proven statically.

RUST: A CASE STUDY IN MEMORY SAFETY

Rust has emerged as a prominent system programming language that provides **memory safety without using a garbage collector**. Unlike traditional low-level languages such as C and C++, Rust prevents common memory errors at compile time through strict rules embedded into its type system and compiler.

This makes Rust suitable for performance-critical systems like operating systems, embedded software, browsers, and cloud infrastructure where both safety and speed are required.

Rust's design is centered around a few powerful concepts that together eliminate issues such as dangling pointers, data races, double frees, and null pointer dereferencing before the program even runs.

Ownership Rules

Ownership is the foundation of Rust's memory model.

The main rules are:

1. Every value in Rust has a single owner.
2. When the owner goes out of scope, the value is automatically dropped (memory freed).
3. Ownership can be transferred (moved) but not duplicated implicitly.

This ensures that memory is freed exactly once and prevents both memory leaks and double free errors.

Example idea:

If a variable holding heap memory is passed to another function, the ownership moves to that function. The original variable can no longer access it.

This simple rule removes many classes of memory errors found in C/C++.

Borrow Checker

The borrow checker is a compile-time component of Rust that ensures references to memory follow strict safety rules.

Borrowing allows temporary access to data without transferring ownership.

Rules enforced:

- You can have **either** one mutable reference **or** multiple immutable references at a time.
- References must always be valid (no dangling references).

This prevents:

- Data races in concurrent code
- Use-after-free errors
- Simultaneous unsafe modifications

The borrow checker is often considered difficult for beginners, but it guarantees memory correctness without runtime overhead.

Lifetimes

Lifetimes in Rust describe how long references are valid. They are checked at compile time to ensure no reference outlives the data it points to.

- This prevents dangling pointers automatically.
- In many cases, lifetimes are inferred by the compiler. Only complex scenarios require explicit lifetime annotations.
- Lifetimes make memory relationships explicit and verifiable before execution.

No Null Pointers

Rust does not allow null pointers in safe code. Instead, it uses the Option<T> type to represent

the presence or absence of a value.

This forces programmers to explicitly handle the case where data may be absent.

As a result:

- Null pointer dereferencing is impossible in safe Rust
- Error handling becomes more structured and reliable

Safe Concurrency

Concurrency is another area where Rust provides strong safety guarantees. Data races are impossible in safe Rust due to ownership and borrowing rules.

Rust achieves this by:

- Preventing shared mutable state without synchronization
- Enforcing thread safety through type system traits like Send and Sync
- Encouraging message passing instead of shared memory

This makes Rust highly suitable for multi-threaded and parallel systems.

MANAGED LANGUAGES AND GARBAGE COLLECTION

Languages such as Java, C#, and Go rely on garbage collectors.

Advantages:

- Automatic memory cleanup
- Easier development
- Reduced pointer errors

Disadvantages:

- Runtime overhead
- Non-deterministic memory release
- Not suitable for real-time systems

Table 2: Comparison of Memory Safe Languages

Feature	Rust	Go	Java	Swift
Garbage Collection	No	Yes	Yes	ARC
Compile-time Safety	High	Medium	Medium	High
Ownership Model	Yes	No	No	Partial
Concurrency Safety	Strong	Moderate	Moderate	Strong
Performance	Very High	High	Medium	High

VERIFICATION TECHNIQUES FOR MEMORY SAFETY

Verification plays important role in proving that a program follows memory safety rules.

Static Analysis

Tools analyze source code without execution to detect potential errors.

Model Checking

System behavior is checked against formal specifications.

Formal Proof Assistants

Tools like Coq, Isabelle help in mathematically proving correctness.

Symbolic Execution

Explores different program paths to find memory violations.

ROLE OF TYPE SYSTEMS IN VERIFICATION

Advanced type systems like dependent types and affine types encode memory rules into language itself. Rust uses affine types to enforce ownership. Such types reduces need for runtime checks.

MEMORY SAFETY IN CONCURRENT SYSTEMS

Concurrency introduces new memory challenges like race conditions. Rust and Swift use compile time checks to prevent data races. Go uses channels to communicate instead of shared memory.

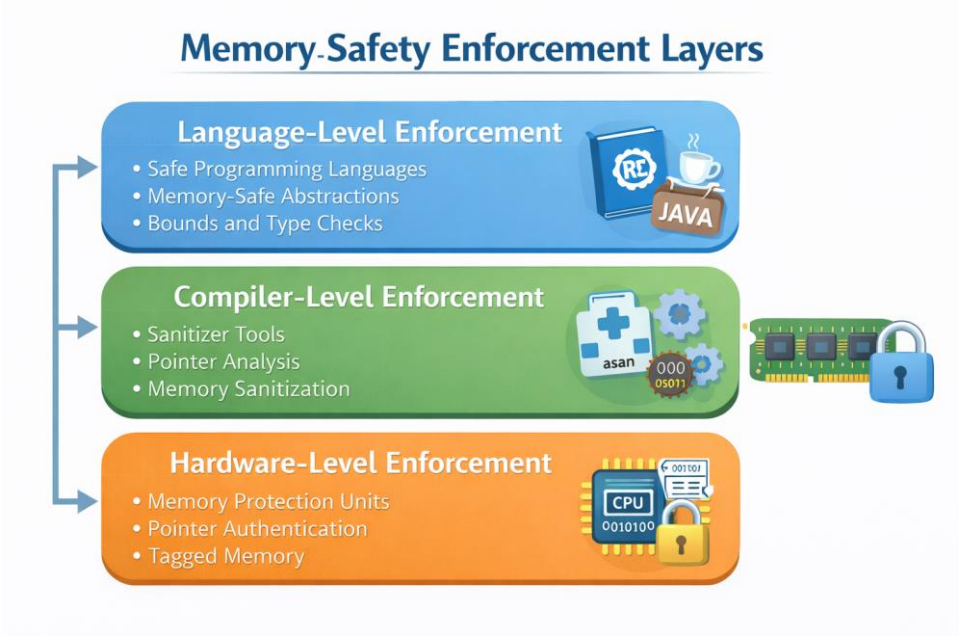

Figure 1: Memory Safety Enforcement Layers

APPLICATIONS OF MEMORY SAFE LANGUAGES

- Operating systems (Rust based kernels)
- Web browsers (Firefox components)
- Embedded and IoT systems
- Blockchain nodes
- Cloud infrastructure software

LIMITATIONS and Challenges

- Learning curve of Rust ownership model
- Garbage collector overhead
- Legacy code written in unsafe languages
- Interoperability issues with C/C++

FUTURE RESEARCH DIRECTIONS

- Hybrid models combining GC and ownership
- Verified compilers for memory safe languages
- Automated migration tools for legacy code
- Memory safety for AI and ML frameworks

CONCLUSION

Memory safety is essential for secure and reliable software. Language design combined with verification techniques provides strong guarantee against common memory errors. Rust shows that compile-time safety without garbage collection is possible. Managed languages still play important role due to ease of development. Verification tools further enhances confidence in software correctness. In future, memory safe languages will dominate system programming and critical applications.

REFERENCES

1. M. Miller, “Trends in Memory Safety Vulnerabilities,” IEEE Security, 2019.
2. N. Matsakis and F. Klock, “The Rust Language,” ACM SIGAda, 2014.
3. J. Gosling, “The Java Language Specification,” Oracle, 2015.
4. R. Pike, “Go at Google: Language Design,” Google Tech Report, 2012.
5. B. Pierce, *Types and Programming Languages*, MIT Press, 2002.
6. X. Leroy, “Formal Verification of Compilers,” CACM, 2009.
7. G. Klein et al., “seL4: Formal Verification of OS Kernel,” SOSP, 2009.
8. A. Donaldson, “Static Analysis for Memory Safety,” IEEE Software, 2018.
9. The Rust Foundation, “Rust Ownership Model Documentation,” 2021.
10. D. Grossman, “Type Systems for Safe Memory Management,” University of Washington, 2003.