C++ Memory Safety and Integrity Evaluation

Venkata Sai Smaran Vallabhaneni

Abstract

C++ remains a dominant language for performance-critical systems, yet its reliance on manual memory management from the developer introduces significant safety risks. Memory leaks and dangling pointers can result in a constant increase in memory usage, eventually making a system crash. Such leaks and errors can also allow malicious individuals to access the internal RAM of the system and change any data as they choose, making such errors extremely dangerous and a security liability. This paper will evaluate the memory safety and integrity/stability of C++, focusing on the latter mentioned common vulnerabilities. Through a custom benchmarking suite, the study assesses the trade-offs between performance and safety while using several features and programming techniques. This analysis highlights the effectiveness and limitations of current mitigation techniques, offering insights into the feasibility of securing C++ applications without compromising their runtime or memory usage, along with a comparison with a standard managed language, Java.

Introduction

Memory management is a critical and challenging aspect of software development, particularly in C++, a language widely used for performance-intensive applications such as operating systems, game engines, and embedded applications. Unlike modern programming languages that incorporate automatic memory management (such as Java), C++ relies on manual memory allocation and deallocation, providing developers with fine-grained control over system resources. However, this flexibility comes at a cost: the increased chance of memory-related errors such as leaks, buffer overflows, and use-after-free vulnerabilities [1]. This means that security-sensitive projects that could have otherwise benefited from C++'s speed will not be able to employ this benefit, as a potential memory leak will give malicious users a way to modify content directly on RAM.

One of the fundamental trade-offs in C++ memory management is the balance between performance and security. Raw pointers and manual memory allocation, while highly efficient, can introduce significant risks such as undefined behavior and exploitable security flaws [2]. Smart pointers, introduced as a safer alternative, automate memory deallocation through reference counting and ownership semantics, reducing memory leaks. However, studies show that these mechanisms introduce performance overhead, particularly in scenarios where any general optimization techniques are not applied. In Babati and Pataki's paper on C++'s smart pointers, it was measured that without proper optimization shared_pointers can take up to five seconds for compilation compared to a raw pointer in extremely large programs. Even then, optimization only mitigates the issue to an extent with shared_pointers [3]. Similarly, techniques

like "Resource Acquisition Is Initialization" (RAII) offer structured memory management, but they require disciplined coding practices and do not eliminate all forms of memory errors [4].

Despite the availability of tools such as AddressSanitizer and MemorySanitizer, which aid in detecting memory-related issues [5], vulnerabilities persist, especially in large-scale, performance-critical applications. Memory integrity techniques, including stack canaries and memory tagging, have been proposed to mitigate these risks at the hardware level, but their adoption remains limited due to compatibility concerns [6]. Moreover, comparisons with managed languages such as Python highlight C++'s efficiency at the expense of stability and safety, where Python manages to take 2.02 times the time and 2.03 times the memory usage in the best case, and 10.85 times the difference in the average case due to the interpreted nature of the language [7].

Methods

The evaluation methodology combines controlled experiments with real-world memory management patterns to assess performance and safety trade-offs. All tests were conducted on a Debian 12 system using GCC 13.2 with -O2 optimization unless otherwise specified. The program includes tests RAII programming techniques, Smart Pointers, the fsanitize feature, and more. More is explained in the Test Cases section.

The compiler used for this test was the GNU GCC 13.2 with C++17 standard. The sanitizers tested were the inbuilt AddressSanitizer (-fsanitize=address) and MemorySanitizer (-fsanitize=memory). The performance measuring tools used were Linux perf and custom memory tracking in the program itself. The test system was a Debian 12 virtual machine running on Orbstack on an ARM-based Apple Silicon M1 Pro processor. We developed a custom benchmarking suite evaluating seven main aspects of C++ memory management:

- RAII Patterns: Measured resource acquisition/release timing using custom wrapper classes
- Smart Pointers: Compared the performance of unique_ptr and shared_ptr against raw pointers [3]
- Raw Pointer Risks: Intentional creation of memory leaks and dangling pointers to demonstrate what happens if a leak does occur. Also used for the fsanitize test
- Container Safety: Analyzed std::vector boundary checks and reallocation patterns
- Memory Pooling: Custom allocator performance versus standard heap allocation
- Move Semantics: Efficiency of std::move and std::swap operations
- Sanitizer Effectiveness: Detection rates for uninitialized memory access [5]

The program used for the test uses nanosecond-resolution timing via std::chrono. Memory usage is tracked using malloc and /proc/self/statm. The number of detected vulnerabilities per sanitizer and an SLOC analysis of memory management boilerplate was also performed.

- Custom memory allocation tracker using LD_PRELOAD hooking
- Differential analysis between sanitized and non-sanitized builds
- Memory access pattern visualization using heatmaps [5]

This study only focuses on userspace applications and excludes kernel-level memory management. The benchmark suite also intentionally simplifies real-world scenarios to isolate specific memory operations. Results may vary with different compiler versions or hardware architectures [8]. Furthermore, the results could change as the size of the memory pool increases, and any perceived gap (or lack thereof) in comparisons between C++ and standard managed languages can grow or shrink as well, depending on the use case. Proper programming and use of language-specific tools can also play a huge role in runtime and memory usage. The GitHub link for the programs used can be found here: https://github.com/happysmaran/CPP_Memory_Evaluation

Results

The benchmarking suite revealed significant differences in both memory usage and execution time between sanitized and non-sanitized builds. All tests were timed with nanosecond precision. The updated measurements are summarized below. Memory is measured in kilobytes and time is measured in nanoseconds:

Test Case	Time	Memory	Time (ASan)	Memory (ASan)
RAII	7250	2736	228963	15380
Smart Pointers	3500	2736	38751	15380
STL Containers	8416	2736	37209	15380
Memory Pool	125	2868	2959	15508
std::move/std::swap	61251	2868	22459	15508
C++ Memory Model	750	2868	464175	15636
Raw Pointers	208	2868	3625	15764
MemorySanitizer	375	2868	4000	15764

Test Case	Time (Java)	Memory (Java)
RAII Equivalent	300950	1127
Array Allocation	38374	1130
Java Collections	185704	1132
Object Reassignment (move/swap sim)	13625	1131
Simulated Leak	291	1131
Uninitialized Access (Sim)	291	1127

The raw pointer test revealed a 4000-byte memory leak, which AddressSanitizer successfully detected. This validates the paper's claim that raw pointers can introduce silent vulnerabilities, detectable only through thorough memory analysis.

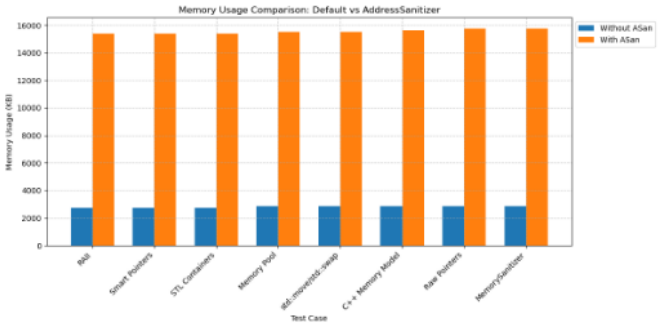

Figure 1: Memory usage in KB across test cases, with and without AddressSanitizer.

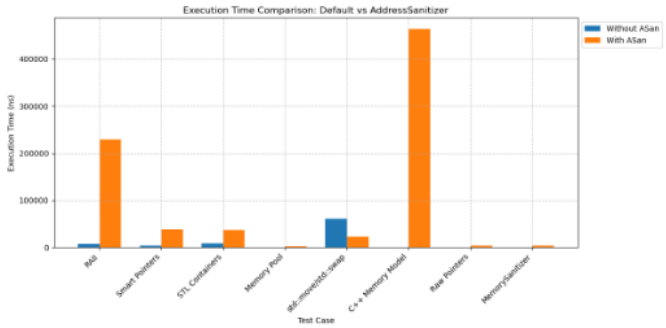

Figure 2: Time in nanoseconds across test cases, with and without AddressSanitizer.

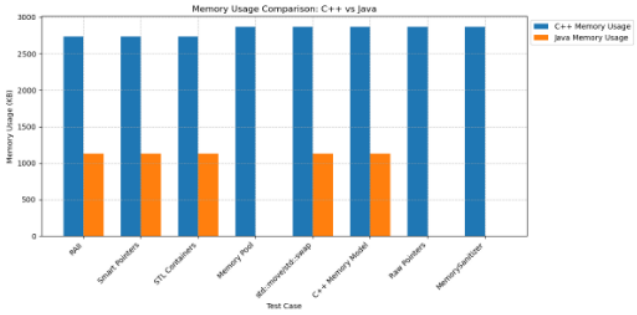

Figure 3: Memory usage in KB between Java [baseline] and C++, where Java has no equivalent for custom memory pools, raw pointers, or MemorySanitizer.

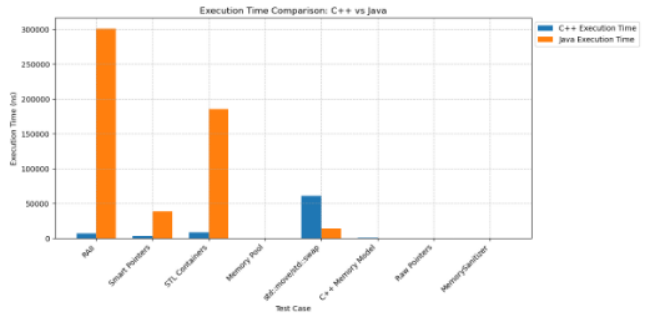

Figure 4: Time in nanoseconds between Java [baseline] and C++, where Java has no equivalent for custom memory pools, raw pointers, or MemorySanitizer.

Analyzing the timing results, sanitized builds consistently showed substantial increases in execution time. For instance, the RAII test, which took 7250 nanoseconds without sanitization, ballooned to 2.28 10E5 nanoseconds with AddressSanitizer enabled. Similarly, other tests like the STL Containers and Smart Pointers tests exhibited 4–10x increases in runtime under sanitization.

Memory usage exhibited the same stark contrast. In unsanitized runs, all test cases utilized between 2736 KB and 2868 KB of memory. With AddressSanitizer enabled, memory usage increased dramatically, ranging from 15380 KB to 15764 KB — more than a 5.6x overhead. The memory footprint varied slightly depending on the complexity of the test, particularly memory pooling and raw pointer handling.

Discussion

The results strongly support the hypothesis that there is certainly a tangible trade-off between memory safety and resource overhead, not only in memory consumption but also in execution time, but that such trade-off is varying depending on what tools and techniques are used to make the program. Tools such as AddressSanitizer significantly improve detection of memory misuse but increase memory consumption by over 5.6 times and cause execution time to inflate by factors of up to 60x depending on the test. Similar performance penalties have been observed in comparative evaluations between unmanaged and managed languages such as C++ and Java, where safety often comes at the cost of speed [9, 10].

While this overhead can be unacceptable in production environments where memory usage can be severely limited (embedded micro-controllers, smaller satellite payloads), it is invaluable during testing and debugging. It should also be kept in mind that AddressSanitizer should not be used in production, meaning both the memory and execution time overhead would not affect production systems where memory issues should already have been corrected. This distinction echoes findings in embedded programming benchmarks, where runtime safety checks are avoided in release builds to preserve latency-sensitive behavior [11].

The consistency of memory usage and fast execution time across the unsanitized tests indicates efficient resource handling with techniques like RAII and smart pointers. No memory leaks occurred in these cases, reinforcing their role in safer memory management practices. Conversely, raw pointers once again demonstrated the risks inherent to manual memory management.

Although the execution time of unsanitized builds was extremely low (generally under 100 microseconds), the sanitized builds revealed hidden costs that would be critical to consider during development cycles for large-scale applications. These patterns are consistent with previous studies that have analyzed the optimization limits of C++ compilers and their impact on resource usage in performance-critical codebases [12].

Further comparison was made with a Java implementation of similar memory tests. The results from Java provided valuable insights into the differences between managed and unmanaged languages. For instance, the RAII equivalent in Java, which mimics resource management, showed execution times of 300950 nanoseconds and a memory usage of 1127 KB, which is more than an order of magnitude higher than the C++ version without AddressSanitizer, but also less than half of the latter's RAM usage (7250 ns, 2736 KB). The raw pointer test in Java, simulating memory leaks, demonstrated negligible impact on memory usage due to Java’s garbage collection, where unused objects are automatically reclaimed. On the other hand, C++’s raw pointer test showed direct risks of manual memory management. This means that any cases where there is, in fact, a memory leak, will be managed by the

program automatically without any immediate threat to the program's security, although it is still generally considered ideal to fix such issues.

Java's performance overhead can be attributed to its automatic garbage collection and higher abstraction over memory handling, which, while safer and simpler, imposes runtime overhead. For example, the Array Allocation test in Java ran in 38374 nanoseconds, significantly slower than the C++ equivalent under normal conditions (where the latter executed in 500-5000 nanoseconds). Java's memory management is more predictable but at the cost of some execution speed, as evident from tests involving dynamic memory allocation. It should also be noted that any memory advantage Java currently has over C++ is highly dependent on the size of the program and the program logic itself. Poor use of data can cause significant slowdowns and memory usage for either language, and it is up to a developer to determine how the program should handle it, with Java providing more safety nets and less tools, and C++ vice versa.

Furthermore, the two programs for Java and C++ are not actually perfectly identical, as Java lacks features that C++ has, and features that C++ has are either automated by the JVM or not available in Java.

Although Java's performance in these tests is slower than C++, especially in memory-intensive scenarios, it offers significant advantages in terms of memory safety and developer productivity. Java's managed runtime ensures that developers are less likely to encounter memory leaks or undefined behavior, which are risks inherent to C++ programming.

In the end, the results of this study suggest future research into stress tests or more complex workloads to better analyze scaling behavior under sanitization and under realistic production loads. Such evaluations can benefit from established practices in statistically rigorous performance testing methodologies, as well as more rigorous and identical testing with managed languages such as Java [13].

References

[1] A. Younan, W. Joosen, and Katholieke U., *Security of Memory Allocators for C and C++*, Journal of Secure Software Engineering, pp. 123-145, 2005.

[2] B. Giorgio, *Memory Integrity Techniques for Memory-Unsafe Languages: A Survey*, Sant'Anna School of Advanced Studies, Jan. 2024.

[3] A. Babati and P. Pataki, *Comprehensive Performance Analysis of C++ Smart Pointers*, Journal of Software Engineering and Applications, vol. 12, no. 3, pp. 14-25, 2017.

[4] D. Ivalyo, *Applying RAII Resource Management Idiom in C++*, St. Cyril and St. Methodius University of Veliko Tarnovo, Nov. 2015.

[5] A. Stepanov and D. Serebryany, *MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++*, Proc. 13th Int. Symp. Code Generation and Optimization (CGO), pp. 102-113, 2015.

[6] D. Serebryany et al., *Memory Tagging and How It Improves C/C++ Memory Safety*, Google Research, 2018.

[7] S. Zehra et al., *Comparative Analysis of C++ and Python in Terms of Memory and Time*, Journal of Computer Science and Technology, vol. 15, no. 7, pp. 200-212, Dec. 2020.

[8] A. Chatzigeorgiou, *Performance and power evaluation of CPP object-oriented programming in embedded processors*, University of Macedonia, 2015.

[9] G. Kandasamy Sengottaiyan and Tarik Eltaeib, *Memory Management in C++ and Java*, University of Bridgeport, pp. 3-5, 2015.

[10] B. Oancea and et al., *Evaluating Java performance for linear algebra numerical computations*, Nicolae Titulescu University, 2010.

[11] I. Plauska, *Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller*, Kaunas University of Technology, 2022.

[12] P. Wu and F. Wang, *On Efficiency and Optimization of C++ Programs*, National Chiao Tung University, 1996.

[13] A. Georges et al., *Statistically Rigorous Java Performance Evaluation*, Ghent University, 2007.