# Zero-Cost State Management: Comparing Rust Typestate and C Runtime Checks at the Assembly Level

Daniel Borgs

*Faculty of Computer Science and Business Information Systems*
*Technical University of Applied Sciences Würzburg-Schweinfurt*
Würzburg, Germany
daniel.borgs@study.thws.de

*Abstract*—**Memory safety vulnerabilities constitute the majority of security flaws in systems software, with Microsoft reporting rates of approximately 70%. C requires explicit runtime checks for state validation, introducing conditional branches and overhead. Rust claims zero-cost abstractions: compile-time verification without runtime penalty. This paper investigates whether Rust's typestate pattern produces assembly identical to unsafe C code.**

**Three file handle state machine implementations were compared: defensive C with runtime validation, minimal C without checks, and Rust with compile-time typestate enforcement. Assembly analysis at -O2 optimization on ARM64 measured instruction counts and conditional branches.**

**Results demonstrate that Rust produces assembly identical to minimal C (2 instructions, 0 branches) while defensive C requires 7 instructions with 2 branches. These findings validate that compile-time type checking eliminates runtime overhead while preventing invalid state transitions.**

*Index Terms*—**memory safety, typestate, Rust, C, zero-cost abstractions, compile-time verification, ARM64**

## I. Introduction

Memory safety vulnerabilities remain the dominant class of security flaws in systems software. According to the 2025 CWE Top 25 Most Dangerous Software Weaknesses, memory corruption vulnerabilities rank highly: Out-of-bounds Write (CWE-787, #5), Use After Free (CWE-416, #7), and Out-of-bounds Read (CWE-125, #8)[1]. Microsoft reports that approximately 70% of their security vulnerabilities are memory safety issues[2]. The U.S. Cybersecurity and Infrastructure Security Agency (CISA) has issued guidance urging software manufacturers to adopt memory-safe languages[3].

Systems programming languages such as C provide the low-level control necessary for performance-critical code. However, C places the entire burden of correctness on the programmer. State management—ensuring operations occur only when preconditions are met[4]—must be enforced through explicit runtime checks. In C, state is typically tracked with an enum field, and each operation checks this field before proceeding, introducing conditional branches with associated overhead.

This creates a fundamental tension: defensive programming incurs runtime overhead, while omitting checks permits undefined behavior. Higher-level languages enforce safety through garbage collection and runtime checks, but these mechanisms limit their applicability to resource-constrained systems[5].

Rust takes a different approach: encoding invariants in the type system so that invalid programs fail to compile. Rust's design philosophy emphasizes *zero-cost abstractions*—high-level constructs that compile to code as efficient as hand-written low-level equivalents[6]. Generic type parameters are instantiated through monomorphization, generating specialized machine code for each concrete type at compile time rather than using runtime dispatch. This paper hypothesizes that Rust's typestate pattern produces assembly identical to unchecked C while providing compile-time safety guarantees. The central research question is: Can compile-time state verification eliminate runtime overhead while preventing invalid state transitions? The complete source code for all implementations, build scripts, and generated assembly are available at https://github.com/failpark/zero_cost/.

## II. Background

Strom and Yemini introduced typestate in 1986 as a compile-time technique to detect invalid execution sequences[7]. Their formulation tracked variables through states like "uninitialized" or "deallocated," with the compiler verifying state flow through program paths. The typestate pattern encodes an object's state in its type: rather than a single `FileHandle` type with a state field, each state becomes a distinct type (`FileHandle<Closed>`, `FileHandle<Open>`, etc.). Operations consume the input type and produce the output type—the `open` method takes `FileHandle<Closed>` by value and returns `FileHandle<Open>`, invalidating the original handle. Attempting to call `read` on a `FileHandle<Closed>` produces a compile-time error; the method does not exist for that type.

Rust applies affine types, meaning values can be used at most once, enabling memory safety without garbage collection. The borrow checker prevents use-after-free by tracking ownership at compile time[8]. While Rust lacks explicit typestate abstractions, its type system supports implementing them

through generic parameters and zero-sized `PhantomData` markers that occupy no memory[9].

Prior empirical work comparing typestate performance to C at the assembly level remains limited. This paper addresses that gap through assembly-level analysis of concrete implementations.

## III. METHODOLOGY

To isolate state management overhead from I/O latency, three variants of a minimal file handle abstraction were implemented (Figure 1). The handle stores only an integer `data` field, and operations simulate state transitions without actual file system calls.
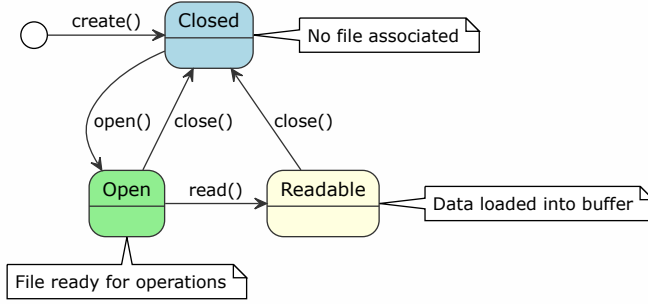


Fig. 1. File handle state machine with four states and their transitions.

Each implementation was compiled at -O2 optimization level: Apple Clang 17.0.0 for C, and rustc 1.91.1 with Rust edition 2024 for Rust. Assembly was extracted using `objdump -d` on an Apple M3 chip (`aarch64-apple-darwin`).

Rust was compiled as a `cdylib` (C-compatible dynamic library) to prevent whole-program optimization from constant-folding the entire state machine. Functions were marked with `#[no_mangle]` to preserve function boundaries for comparison. The C implementations were compiled as shared libraries with equivalent settings (`-shared -fPIC`).

This study is limited to ARM64 architecture; results on other platforms such as x86-64 may differ.

Instructions and conditional branches were counted manually for each function, with assembly listings verified against compiler output. Four state transition functions were analyzed: `open`, `read`, `get_data`, and `close`. These four functions represent all state transitions in the file handle state machine, providing complete coverage of the pattern's behavior.

### A. Defensive C Implementation

The defensive implementation tracks state explicitly with an enum and validates preconditions before each operation:

```
int file_handle_get_data(file_handle_t* h) {
  if (h->state != STATE_READABLE) return -1;
  return h->data;
}
```

Listing 1. Defensive C state management with runtime validation.

Each operation contains a conditional branch, producing comparison and jump instructions.

### B. Minimal C Implementation

The minimal implementation omits all state tracking:

```
int file_handle_get_data(file_handle_t* h) {
  return h->data;
}
```

Listing 2. Minimal C without state checks

This version represents the performance ceiling but permits invalid operations.

### C. Rust Typestate Implementation

The Rust implementation encodes each state as a zero-sized type:

```
struct Closed; struct Open; struct Readable;

struct FileHandle<State> {
  data: i32,
  _state: PhantomData<State>,
}

impl FileHandle<Open> {
  fn read(self) -> FileHandle<Readable> {
    FileHandle { data: 42, _state: PhantomData }
  }
}

impl FileHandle<Readable> {
  fn get_data(&self) -> i32 { self.data }
}
```

Listing 3. Rust typestate pattern. Methods consume `self` by value for state transitions.

The `PhantomData<State>` field is zero-sized and exists only for type checking. The `get_data` method is only defined for `FileHandle<Readable>`; calling it on other states produces a compile error.

## IV. RESULTS

The defensive C implementation produced the following assembly for `get_data` (comments added for clarity; original `objdump` output contains only instructions):

```
ldr  w8, [x0]        ; Load state field
cmp  w8, #0x2        ; Compare to READABLE
b.ne error           ; Branch if invalid
ldr  w0, [x0, #4]    ; Load data
ret
error:               ; Label (not an instruction)
mov  w0, #-1         ; Return error
ret
```

This yields 7 instructions: 2 memory loads, 1 comparison, 1 conditional branch, 1 move, and 2 returns. The label `error:` marks a branch target and is not counted as an instruction. The 7-instruction count includes the error handling path; the happy path (valid state) requires 5 instructions, but both paths must be present in the compiled binary.

The minimal C and Rust implementations both produced identical assembly:

```
1  ldr  w0, [x0]        ; Load data directly
2  ret
```

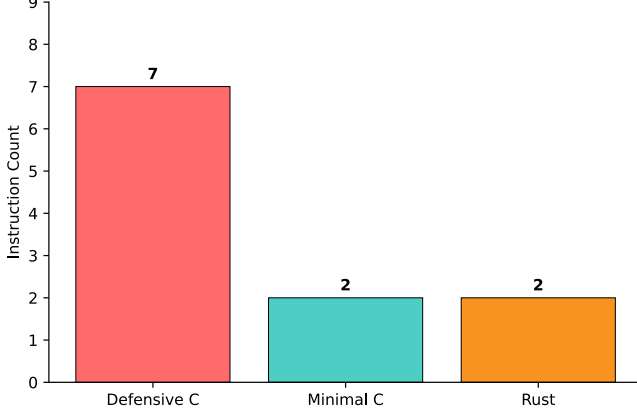This yields 2 instructions: 1 memory operation and 1 return, with zero conditional branches.



Fig. 2. Total instruction count comparison at -O2. Rust matches minimal C with 2 instructions, while defensive C requires 7 instructions due to state validation overhead.

Figure 2 demonstrates that Rust achieves the same instruction count as minimal C—both implementations compile to exactly 2 instructions. The defensive C implementation requires $3.5\times$ the instructions due to explicit state checking logic.
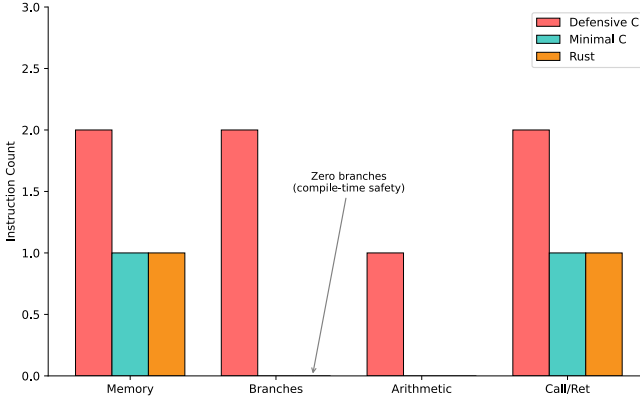


Fig. 3. Instruction category breakdown. The critical finding: Rust has zero conditional branch instructions, matching minimal C, while defensive C includes 2 branch instructions for runtime state validation.

Figure 3 reveals the key insight: Rust's typestate implementation contains zero conditional branch instructions, identical to minimal C. The defensive C version includes a compare instruction (`cmp`) and one conditional branch (`b.ne`) that verify state validity at runtime. These instructions represent the performance cost of runtime safety checks that Rust eliminates through compile-time verification.

## A. Multi-Function Analysis

To verify that results generalize beyond a single function, all four state transition operations were analyzed using the same methodology. Table I presents the complete results.

TABLE I
INSTRUCTION COUNTS ACROSS ALL STATE TRANSITION FUNCTIONS AT -O2 OPTIMIZATION.

| Function | Defensive C | Minimal C / Rust | Overhead |
|---|---|---|---|
| `get_data` | 7 | 2 | $3.5\times$ |
| `open` | 9 | 2 | $4.5\times$ |
| `read` | 11 | 2 | $5.5\times$ |
| `close` | 8 | 2 | $4.0\times$ |

Across all functions, Rust and minimal C produce byte-identical assembly, while defensive C requires $3.5$–$5.5\times$ the instructions. Complete assembly listings for all functions are available in the project repository for independent verification.

## V. DISCUSSION

The empirical results validate the zero-cost abstraction principle: Rust's type-level state encoding produces assembly identical to minimal C while providing compile-time safety guarantees equivalent to defensive C's runtime checks.

### A. Implications for Security

Google's Android team reported that as memory-unsafe code decreased, memory safety vulnerabilities dropped from 76% in 2019 to 35% in 2022[10]. The typestate pattern extends this principle beyond memory safety to protocol correctness — APIs that enforce valid state sequences through types prevent logic errors through compile-time guarantees, not runtime checks that might be forgotten.

### B. Generalizability

The consistency across all four analyzed functions strengthens confidence that the zero-cost property is not an artifact of a single cherry-picked example. The identical assembly output for Rust and minimal C across different state transitions suggests that the typestate pattern reliably compiles away at optimization levels typical of production builds.

However, several factors limit generalizability. First, only a single state machine pattern was analyzed; more complex state machines with additional fields or nested states may exhibit different behavior. Second, the synthetic benchmark isolates state management from real-world concerns such as I/O, error propagation, and concurrent access. Third, LLVM's optimization behavior may differ across versions or with different optimization flags.

## VI. CONCLUSION

This paper compares three file handle implementations: defensive C with runtime validation, minimal C without checks, and Rust with compile-time typestate enforcement. The empirical results validate the zero-cost abstraction principle: Rust's typestate implementation produces assembly identical to minimal C (2 instructions, 0 branches) while providing

compile-time guarantees equivalent to defensive C's runtime checks (7 instructions, 2 branches). Across four functions, defensive C consistently requires 3.5–5.5× more instructions.

These findings suggest that the traditional safety-performance tradeoff is not fundamental. With sufficiently expressive type systems, safety can become a compile-time property with zero runtime cost.

REFERENCES

[1] "CWE - 2025 CWE top 25 most dangerous software weaknesses," Accessed: Dec. 31, 2025. [Online]. Available: https://web.archive.org/web/20251231122313/ https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html.

[2] Microsoft Security Response Center. "We need a safer systems programming language," Accessed: Dec. 5, 2025. [Online]. Available: https://web.archive.org/web/20251205123558/https://www.microsoft.com/en-us/msrc/blog/2019/07/we-need-a-safer-systems-programming-language.

[3] Cybersecurity and Infrastructure Security Agency. "The urgent need for memory safety in software products," Accessed: Dec. 23, 2025. [Online]. Available: https://web.archive.org/web/20251223080013/https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products.

[4] A. Rebert and C. Kern, "Secure by design: Google's perspective on memory safety," Google, Tech. Rep., Mar. 2024. Accessed: Dec. 31, 2025. [Online]. Available: https://web.archive.org/web/20250831141723/ https://storage.googleapis.com/gweb-research2023-media/pubtools/7665.pdf.

[5] Office of the National Cyber Director, "Back to the building blocks: A path toward secure and measurable software," The White House, Tech. Rep., Feb. 2024. [Online]. Available: https://web.archive.org/web/20250120134421/https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[6] Rust Embedded Working Group. "Zero cost abstractions - the embedded rust book," Accessed: Dec. 30, 2025. [Online]. Available: https://web.archive.org/web/20251130072449/https://doc.rust-lang.org/beta/embedded-book/static-guarantees/zero-cost-abstractions.html.

[7] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, Jan. 1986, ISSN: 1939-3520. DOI: 10.1109/TSE.1986.6312929. Accessed: Jan. 10, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/6312929/.

[8] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," in *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 3, 2019, pp. 1–30. DOI: 10.1145/3360573.

[9] Rust Project. "PhantomData in std::marker - Rust," Accessed: Jan. 6, 2026. [Online]. Available: https://web.archive.org/web/20260106142924/https://doc.rust-lang.org/std/marker/struct.PhantomData.html.

[10] J. Vander Stoep and S. Hines. "Memory safe languages in android 13," Accessed: Dec. 24, 2025. [Online]. Available: https://web.archive.org/web/20251224110719/https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html.