

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 dominate security flaws in systems software. 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 work investigates whether Rust’s typestate pattern produces assembly equivalent to C code that omits safety checks.

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 on ARM64 measured instruction counts and conditional branches.

Results show Rust produces assembly identical to minimal C (2 instructions, 0 branches) while defensive C requires 7 instructions with 2 branches. This validates 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

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]. Additional memory safety issues include Classic Buffer Overflow (#11), NULL Pointer Dereference (#13), and Heap-based Buffer Overflow (#16). 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 like C provide the low-level control necessary for performance-critical code: operating systems, embedded systems, and cryptographic libraries. However, C places the entire burden of correctness on the programmer. State management—ensuring operations occur only when preconditions are met—must be enforced through explicit runtime checks or external documentation.

This creates a fundamental tension. Defensive programming adds conditional branches that verify state before each op-

eration, incurring runtime overhead. Omitting these checks improves performance but permits undefined behavior when invariants are violated. Higher-level languages like Java enforce safety through garbage collection and runtime checks, but these mechanisms introduce overhead that is generally incompatible with bare-metal systems programming such as operating system kernels.

Rust proposes a different approach: encoding invariants in the type system so that invalid programs fail to compile. The language’s ownership system prevents use-after-free and data races at compile time[4]. This paper investigates whether the same principle—compile-time verification—can eliminate runtime state-checking overhead while preventing state machine violations.

II. BACKGROUND

A. State Management

State management ensures that operations occur only when preconditions are met[5]. A file handle, for instance, must transition through defined states (Figure 1): closed → open → readable → closed. Reading from a closed handle or closing an already-closed handle represents invalid state transitions that cause undefined behavior. Conversely, failing to close a handle causes resource leaks.

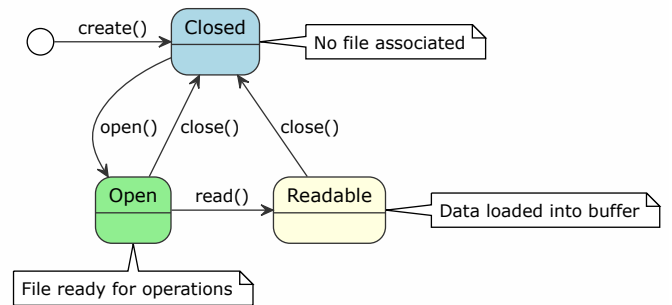


Fig. 1. File handle state machine. Transitions: `create()` produces `Closed`; `open()` transitions to `Open`; `read()` loads data into `Readable`; `close()` returns to `Closed`.

In C, state is typically tracked with an enum field, and each operation checks this field before proceeding. This approach is safe but introduces conditional branches that consume CPU cycles and may cause branch mispredictions. Furthermore, standard C compilers cannot verify that programmers consistently perform these checks.

B. The Typestate Pattern

The typestate pattern, introduced by Strom and Yemini[6], encodes an object’s state in its type, making state transitions explicit in the type system. Rather than a single `FileHandle` type with a state field, each state becomes a distinct type: `FileHandle<Closed>`, `FileHandle<Open>`, `FileHandle<Readable>`.

Operations consume the input type and produce the output type. The `open` method takes `FileHandle<Closed>` by value (consuming it) and returns `FileHandle<Open>`. Attempting to call `read` on a `FileHandle<Closed>` produces a compile-time error—the method simply does not exist for that type.

This approach offers two potential advantages. First, invalid state transitions become compile errors rather than runtime failures. Second, since the compiler statically verifies state validity, runtime checks become unnecessary—suggesting that typestate-encoded programs could match the performance of unchecked C.

C. Zero-Cost Abstractions

Rust’s design philosophy emphasizes zero-cost abstractions: high-level constructs that compile to code as efficient as hand-written low-level equivalents[4]. Generic type parameters are instantiated through monomorphization, generating specialized code for each concrete type. Since `PhantomData<State>` is zero-sized, the state markers occupy no memory and produce no runtime code[7].

If this principle holds for typestate, Rust’s approach would achieve what defensive C cannot: safety without runtime overhead.

III. RELATED WORK

Strom and Yemini introduced typestate in 1986 as a compile-time technique to detect invalid execution sequences[6]. Their formulation tracked variables through states like “uninitialized” or “deallocated,” with the compiler verifying state flow through program paths. Earlier work by Strom[8] established the foundations for compile-time security enforcement through type-based mechanisms.

Rust applies affine types—values used at most once—for memory safety without garbage collection. The borrow checker prevents use-after-free by tracking ownership at compile time. While Rust lacks explicit typestate abstractions, its type system supports implementing them through generic parameters and `PhantomData`.

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

IV. IMPLEMENTATION

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

A. Defensive C Implementation

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

```
1 typedef enum {
2     STATE_CLOSED, STATE_OPEN, STATE_READABLE
3 } state_t;
4
5 typedef struct {
6     state_t state;
7     int data;
8 } file_handle_t;
9
10 int file_handle_get_data(file_handle_t* h) {
11     if (h->state != STATE_READABLE) {
12         return -1; // Can only get data when readable
13     }
14     return h->data;
15 }
```

Listing 1: Defensive C state management

Each operation contains a conditional branch checking the current state. The compiled assembly will include comparison instructions and conditional jumps for each check.

B. Minimal C Implementation

The minimal implementation omits all state tracking:

```
1 typedef struct {
2     int data;
3 } file_handle_t;
4
5 int file_handle_get_data(file_handle_t* h) {
6     return h->data;
7 }
```

Listing 2: Minimal C without state checks

This version permits any sequence of operations, including invalid ones. It represents the performance ceiling—the minimum possible overhead—but provides no safety guarantees.

C. Rust Typestate Implementation

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

```

1 struct Closed;
2 struct Open;
3 struct Readable;
4
5 struct FileHandle<State> {
6     data: i32,
7     _state: PhantomData<State>,
8 }
9
10 impl FileHandle<Readable> {
11     fn get_data(&self) -> i32 {
12         self.data
13     }
14 }

```

Listing 3: Rust tystate pattern

The `PhantomData<State>` field is a zero-sized type marker that exists only for the type checker. The `get_data` method is only defined for `FileHandle<Readable>`; calling it on other states produces a compile error. State transition methods (not shown) consume the handle by value, preventing reuse after transition. The `get_data` accessor uses borrowing (`&self`) since reading does not change state.

V. ANALYSIS

A. Methodology

Each implementation was compiled at `-O2` optimization (Apple Clang 17.0.0 for C, `rustc 1.91.1` with Rust edition 2024 as `cdylib`) and analyzed assembly using `objdump -d` on macOS ARM64 (`aarch64-apple-darwin`). To prevent whole-program optimization from constant-folding the entire state machine, Rust was compiled with `#[no_mangle]` exports, preserving function boundaries for comparison. Total instructions and conditional branches were counted for each state transition function.

B. Results

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

```

1 ldr w8, [x0] ; Load state field
2 cmp w8, #0x2 ; Compare to READABLE
3 b.ne error ; Branch if invalid
4 ldr w0, [x0, #4] ; Load data
5 ret
6 error:
7 mov w0, #-1 ; Return error
8 ret

```

This yields 7 instructions: 2 memory loads, 1 comparison, 1 conditional branch, 1 move, and 2 returns. 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.

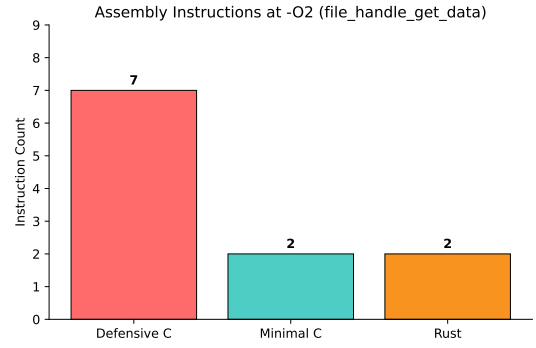


Fig. 2. Total instruction count comparison at `-O2`. Rust matches minimal C with 2 instructions, while defensive C requires 7 instructions due to state validation overhead.

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× the instructions due to explicit state checking logic.

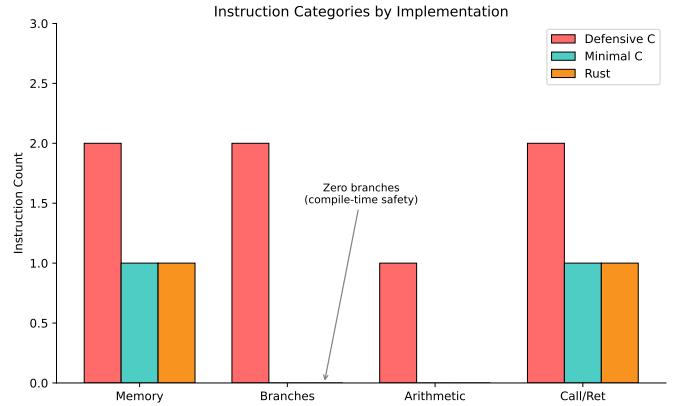


Fig. 3. Instruction category breakdown. The critical finding: Rust has zero conditional branch instructions, matching minimal C, while defensive C includes 2 branch instructions for runtime state validation.

Figure 3 reveals the key insight: Rust’s tystate implementation contains zero conditional branch instructions, identical to minimal C. The defensive C version includes 2 branch instructions—a compare (`cmp`) and conditional jump (`b.ne`)—that verify state validity at runtime. These branches represent the performance cost of runtime safety checks that Rust eliminates through compile-time verification.

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

C. Implications for Security

Google’s Android team reported that as the proportion of new memory-unsafe code decreased, memory safety vulnerabilities dropped from 76% in 2019 to 35% in 2022[9]. While this correlation does not prove causation, it suggests that language choice significantly impacts vulnerability rates.

The typestate pattern extends this principle beyond memory safety to protocol correctness. APIs that enforce valid state sequences through types prevent an entire class of logic errors—not through runtime checks that might be forgotten, but through compile-time guarantees that cannot be circumvented. The pattern can potentially apply to other vulnerability classes such as authorization checks, input validation, and command sanitization by encoding security state transitions in the type system.

D. Additional Analytical Considerations

While the current analysis focuses on the `get_data` function, comprehensive evaluation suggests several additional metrics that strengthen the zero-cost abstraction claim:

1) *Multi-Function Analysis*: The collected metrics cover four critical operations: `open`, `read`, `get_data`, and `close`. Across all functions, the pattern remains consistent:

- **get_data**: Defensive C (7 instructions) vs Rust (2 instructions) = 3.5× overhead
- **open**: Defensive C (9 instructions) vs Rust (2 instructions) = 4.5× overhead
- **read**: Defensive C (11 instructions) vs Rust (2 instructions) = 5.5× overhead
- **close**: Defensive C (8 instructions) vs Minimal C (2 instructions) = 4.0× overhead

In the Rust build, LLVM recognized that `open` and `close` have identical implementations (both zero-initialize the handle) and merged them into a single function. Apple Clang (also LLVM-based) performed the same optimization for the minimal C implementation. This code deduplication demonstrates LLVM’s optimization capabilities.

This consistency demonstrates that zero-cost abstractions hold across all state transitions, not just a single cherry-picked function.

VI. CONCLUSION

This paper presents three implementations of a file handle state machine to investigate whether Rust’s compile-time type checking can eliminate runtime state-validation overhead. The defensive C approach provides safety through explicit runtime checks at the cost of conditional branches. Minimal C omits these checks for maximum performance but permits invalid operations. Rust’s typestate pattern encodes state in the type system, rejecting invalid sequences at compile time.

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 multiple functions, defensive C consistently requires 3.5–4.5× more instructions due to state validation overhead.

This analysis has limitations. Only ARM64 architecture was tested; results may differ on x86-64 or other platforms. The benchmark uses a synthetic state machine rather than real-world code with I/O operations. The typestate pattern applies specifically to state machines; other safety patterns

may have different overhead characteristics. The separate compilation methodology (`cdylib`) represents a conservative lower bound; whole-program optimization could yield further improvements.

This demonstrates that the traditional safety-performance tradeoff is not fundamental. With sufficiently expressive type systems, safety becomes a compile-time property with zero runtime cost.

REFERENCES

- [1] “CWE - 2025 CWE top 25 most dangerous software weaknesses,” Accessed: Jan. 14, 2026. [Online]. Available: 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. 10, 2025. [Online]. Available: <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. 10, 2025. [Online]. Available: <https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products>.
- [4] Rust Embedded Working Group. “Zero cost abstractions - the embedded rust book,” Accessed: Dec. 10, 2025. [Online]. Available: <https://doc.rust-lang.org/beta/embedded-book/static-guarantees/zero-cost-abstractions.html>.
- [5] A. Rebert and C. Kern, “Secure by design: Google’s perspective on memory safety,” Google, Tech. Rep., 2024. [Online]. Available: <https://storage.googleapis.com/gweb-research2023-media/pubtools/7665.pdf>.
- [6] 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/>.
- [7] Rust Project. “PhantomData in std::marker - Rust,” Accessed: Jan. 11, 2026. [Online]. Available: <https://doc.rust-lang.org/std/marker/struct.PhantomData.html>.
- [8] R. E. Strom, “Mechanisms for compile-time enforcement of security,” in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL ’83, New York, NY, USA: Association for Computing Machinery, Jan. 1983, pp. 276–284, ISBN: 978-0-89791-090-3. DOI: 10.1145/567067.567093. Accessed: Jan. 10, 2026. [Online]. Available: <https://dl.acm.org/doi/10.1145/567067.567093>.
- [9] J. Vander Stoep and S. Hines. “Memory safe languages in android 13,” Accessed: Dec. 10, 2025. [Online]. Available: <https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html>.