

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 industry reports indicating 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, including 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 operation, incurring runtime overhead. Omitting these checks improves performance but permits undefined behavior when

invariants are violated. Higher-level languages such as Java enforce safety through garbage collection and runtime checks, but these mechanisms introduce runtime overhead that limits their applicability to resource-constrained and bare-metal systems programming[4].

Rust takes 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[5]. 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 at compile time?

II. BACKGROUND

A. State Management

State management ensures that operations occur only when preconditions are met[6]. A file handle, for instance, must transition through defined states (Figure 1): `create()` produces a Closed handle; `open()` transitions to Open; `read()` loads data into Readable; `close()` returns to Closed. Reading from a closed handle or closing an already-closed handle represents invalid state transitions that may cause undefined behavior in C, depending on the implementation. Conversely, failing to close a handle causes resource leaks.

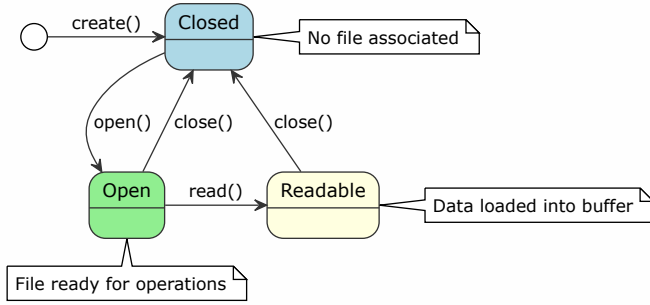


Fig. 1. File handle state machine. Transitions: `create()` produces `Closed`; `open()` transitions to `Open`; `read()` loads data into `Readable`; `close()` returns to `Closed`.

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 Tystate Pattern

The tystate pattern, introduced by Strom and Yemini[7], 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>`. This consumption is critical: once the handle transitions states, the original handle is invalidated, preventing operations on stale state—analogous to how Rust’s ownership prevents use-after-free at the memory level. 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 tystate-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[5]. Generic type parameters are instantiated through monomorphization—the process of generating specialized machine code for each concrete type used at compile time, rather than using runtime dispatch. Since `PhantomData<State>` is zero-sized, the state markers occupy no memory and produce no runtime code[8].

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

III. RELATED WORK

Strom and Yemini introduced tystate 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. Earlier work by Strom[9] established the foundations for compile-time security enforcement through type-based mechanisms.

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. While Rust lacks explicit tystate abstractions, its type system supports implementing them through generic parameters and `PhantomData`.

Prior empirical work comparing tystate performance to C at the assembly level remains limited. Existing literature focuses primarily on the theoretical foundations of tystate[7], [9], with minimal attention to empirical performance validation. This paper addresses that gap through assembly-level analysis of concrete implementations.

IV. METHODOLOGY

To isolate state management overhead from I/O latency, three variants of a minimal file handle abstraction were implemented. The handle stores only an integer data field, and operations simulate state transitions without actual file system calls. This synthetic approach enables direct comparison of state-checking overhead without confounding factors.

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 macOS ARM64 (`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:

```

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. The error return value of -1 represents one possible error-handling strategy; alternatives include setting errno or using out-parameters.

Each operation contains a conditional branch checking the current state. The compiled assembly includes 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<Closed> {
11     fn open(self) -> FileHandle<Open> {
12         FileHandle { data: self.data, _state:
13             ↪ PhantomData }
14     }
15
16 impl FileHandle<Open> {
17     fn read(self) -> FileHandle<Readable> {
18         FileHandle { data: 42, _state: PhantomData }
19     }
20 }
21
22 impl FileHandle<Readable> {
23     fn get_data(&self) -> i32 {
24         self.data
25     }
26 }

```

Listing 3: Rust typestate pattern with state transition methods

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 consume the handle by value (`self` rather than `&self`), preventing reuse after transition. The `get_data` accessor uses borrowing (`&self`) since reading does not change state.

V. RESULTS

The defensive C implementation produced the following assembly for `get_data` (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: ; Label (not an instruction)
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 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.

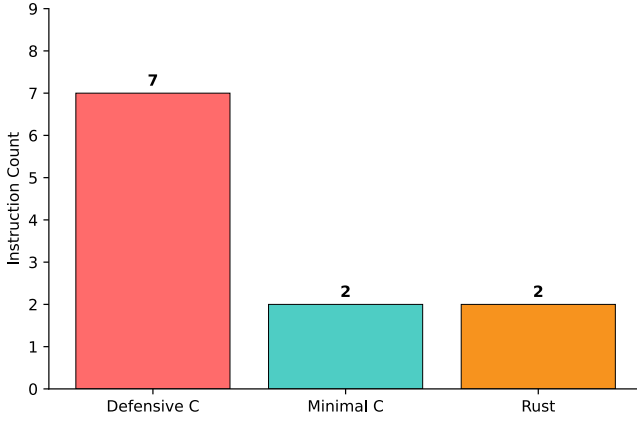


Fig. 2. Total instruction count comparison at -O2. Rust matches minimal C with 2 instructions, while defensive C requires 7 instructions due to state validation overhead.

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.

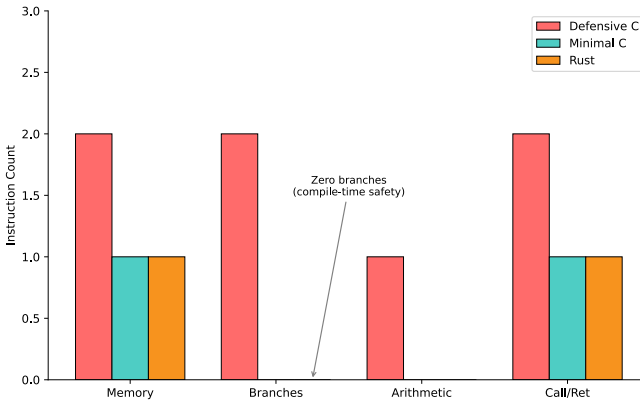


Fig. 3. Instruction category breakdown. The critical finding: Rust has zero conditional branch instructions, matching minimal C, while defensive C includes 2 branch instructions for runtime state validation.

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, the pattern remains consistent: Rust and minimal C produce identical assembly, while defensive C requires 3.5–5.5 \times the instructions due to state validation overhead.

In the Rust build, LLVM recognized that `open` and `close` have identical implementations (both produce a handle with the same data) 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 and does not affect the validity of the comparison.

VI. 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 the proportion of new memory-unsafe code decreased, memory safety vulnerabilities dropped from 76% in 2019 to 35% in 2022[10]. 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. Future work could investigate whether the typestate pattern applies to other vulnerability classes such as authorization checks, input validation, and command sanitization by encoding security state transitions in the type system.

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.

VII. 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 four functions, defensive C consistently requires $3.5\text{--}5.5\times$ 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 and warrant further investigation. Additionally, only -O2 optimization was tested; behavior at -O3 or with link-time optimization (LTO) remains unexplored. 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. Only a single state machine design was evaluated; more complex patterns warrant further investigation. The separate compilation methodology (cdylib) represents a conservative lower bound; whole-program optimization could yield further improvements.

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] 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>.
- [5] 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>.
- [6] 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>.
- [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] 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>.
- [9] 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>.
- [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>.