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 provides zero-cost abstractions: compile-time verification without runtime penalty. This paper aims to investigate whether Rust's typestate pattern produces assembly identical to unsafe C code.**

Three file handle state machine implementations were compared:

1) **defensive C with runtime validation,**
2) **minimal C without checks,**
3) **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, assembly analysis**

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]—requires explicit runtime checks in C. 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. Many 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. Can compile-time state verification eliminate runtime overhead while preventing invalid state transitions?

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—needing no memory allocated—`PhantomData` markers [9].

Typestate analysis, sometimes called *protocol analysis*, defines valid sequences of operations that can be performed upon an instance of a given type [10]. DeLine and Fähndrich demonstrated how typestate can enforce high-level protocols in low-level software, describing resource management protocols that specify operations must be performed in a certain order [11]. Garcia et al. formalized typestate-oriented programming, showing how a typestate checker can statically ensure that an object method is only called when the object is in a state for which the operation is well defined [12]. However, prior research has focused primarily on formalization and type theory rather than performance characteristics. Existing empirical work on typestate languages has examined usability [13] rather than runtime overhead. To the author's knowledge, no prior study has compared the assembly output of typestate implementations against equivalent C code at the instruction level. This paper tries to address that gap through assembly-level analysis of concrete implementations.

III. Methodology

The complete source code for all implementations, build scripts, and generated assembly is available at https://github.com/failpark/zero_cost/. 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.

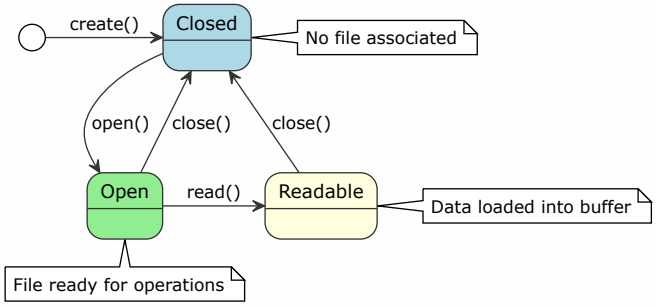

Fig. 1. File handle state machine with four states and their transitions. Author's figure. Generated using PlantUML.

Each implementation was compiled at -O2 optimization level—the industry standard [14]—: 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:

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

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

B. Minimal C Implementation

The minimal implementation omits all state tracking:

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

This version represents optimal performance but permits invalid operations.

C. Rust Typestate Implementation

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

```rust
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 }
}
```

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:` is a symbolic marker, not an executed instruction and thus not counted. 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 instruction-equivalent assembly:

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

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

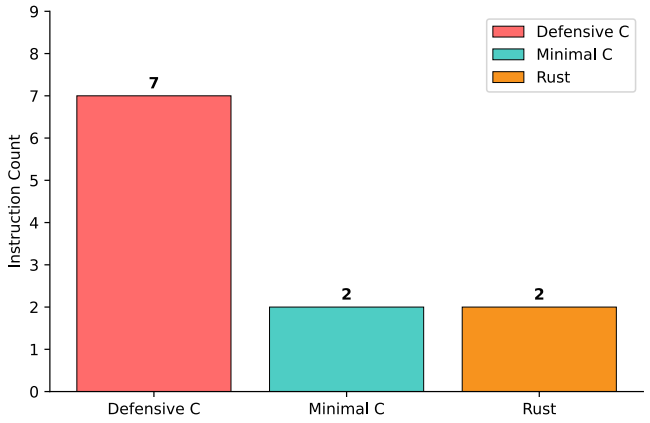

Fig. 2. Total instruction count comparison. Author's figure. Generated using matplotlib.

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.

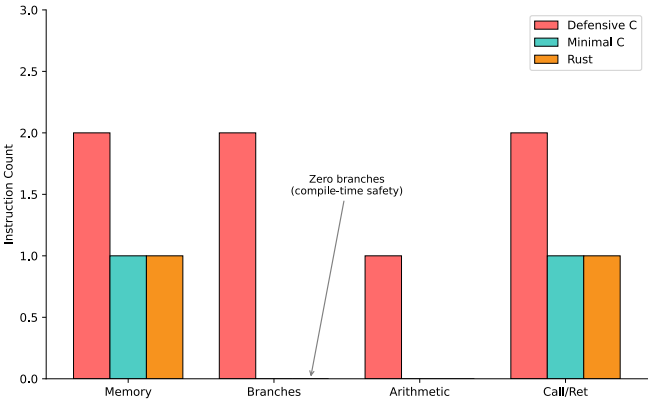

Fig. 3. Instruction category breakdown. Author's figure. Generated using matplotlib.

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×
`open`	9	2	4.5×
`read`	11	2	5.5×
`close`	8	2	4.0×

Across all functions, Rust and minimal C produce instruction-identical assembly, while defensive C requires 3.5–5.5× 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 [15].

The typestate pattern extends this principle beyond memory safety to protocol correctness. Strom and Yemini's original formulation ensured that "the sequence of operations on each variable obeys a finite state grammar associated with that variable's type" [7]. More recent work has shown how typestate can enforce that operations must be performed in a certain order and that certain operations must be performed before accessing a given data object [11]. 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, LLVM's optimization behavior may differ across versions or

with different optimization flags. Second, the synthetic benchmark isolates state management from real-world concerns such as I/O, error propagation, and concurrent access. Third, only a single state machine pattern was analyzed; more complex state machines with additional fields or nested states may exhibit different behavior. Most importantly, the typestate pattern may increase type complexity and may impact compile times or code readability.

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 for certain patterns, the traditional safety-performance tradeoff can be avoided. With sufficiently expressive type systems, safety can become a compile-time property with zero runtime cost. Future work should examine x86-64 platforms, more complex state machines, and runtime performance under real workloads.

References

[1] "CWE - 2025 CWE top 25 most dangerous software weaknesses," Accessed: Jan. 23, 2026. [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: Jan. 23, 2026. [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: Jan. 23, 2026. [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: Jan. 23, 2026. [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: Jan. 23, 2026. [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. 23, 2026. [Online]. Available: https://web.archive.org/web/20260106142924/https://doc.rust-lang.org/std/marker/struct.PhantomData.html.

[10] R. DeLine and M. Fähndrich, "Typestates for objects," in *ECOOP 2004 – Object-Oriented Programming, 18th European Conference*, M. Odersky, Ed., ser. Lecture Notes in Computer Science, vol. 3086, Oslo, Norway: Springer, 2004, pp. 465–490. DOI: 10.1007/978-3-540-24851-4_21.

[11] R. DeLine and M. Fähndrich, "Enforcing high-level protocols in low-level software," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, USA: ACM, Jun. 2001, pp. 59–69. DOI: 10.1145/378795.378811.

[12] R. Garcia, É. Tanter, R. Wolff, and J. Aldrich, "Foundations of typestate-oriented programming," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 4, pp. 1–44, Oct. 2014. DOI: 10.1145/2629609.

[13] M. Coblenz, J. Aldrich, B. A. Myers, and J. Sunshine, "Can advanced type systems be usable? An empirical study of ownership, assets, and typestate in Obsidian," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428200.

[14] R. H. Enterprise. "Chapter 15. building code with gcc — developer guide — red hat enterprise linux — 7 — red hat documentation," Accessed: Jan. 23, 2026. [Online]. Available: https://web.archive.org/web/20260114140435/https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/developer_guide/gcc-compiling-code.

[15] J. Vander Stoep and S. Hines. "Memory safe languages in android 13," Accessed: Jan. 23, 2026. [Online]. Available: https://web.archive.org/web/20251224110719/https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html.