

Do Rust's compile-time type checks produce equivalent assembly to C while preventing state management errors?

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—use-after-free, buffer overflows, out-of-bounds write—are at the top of known exploited vulnerabilities[1]. System programming languages like C, which allow for low-level memory control, are needed to write performance critical code. State is the current condition of an object that determines what operations are valid. C cannot enforce valid state transitions, such as reading an open file, at compile-time. Operations need to be checked explicitly at runtime if they are in a valid state before running. Higher-level languages like Java automate detecting potentially unsafe operations, preventing errors but degrading performance. Therefore, developers have to make a tradeoff between safety and speed.

Rust has the ability to move certain behaviors to compile-time execution or analysis. This approach claims to be able to verify state validity during compilation, which could lead to preventing errors that runtime checks would detect while achieving comparable C performance. This work tests whether Rust's compile-time guarantees produce assembly with equivalent performance to C code that omits all safety checks.

Programs must track state explicitly or implicitly. A file handle must be opened before reading; once closed, reads must fail. Verifying these transitions—the core of a state machine—requires verification logic. This work implements three versions of a file handle state machine, isolating state management from I/O overhead:

- 1) **Defensive C**: Uses an enum to track state, with explicit validation checks before each operation. Safe, but includes runtime conditional branches.
- 2) **Minimal C**: Tracks state with an enum, but omits all validation. Fast but permits invalid operations to compile.
- 3) **Rust**: Encodes each state as a distinct type, making invalid transitions not compile.

The resulting assembly will be compared using total instruction count, conditional branches, and state-tracking overhead.

The deliverables are side-by-side assembly comparisons showing whether Rust eliminates defensive checks present in safe C and an analysis with the above mentioned assembly. The assembly comparison will demonstrate whether encoding state in the type system eliminates defensive branches while matching minimal C's instruction count.

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 2024

CWE Top 10 Known Exploited Vulnerabilities list, memory corruption issues such as out-of-bounds writes, use-after-free, and buffer overflows occupy the highest ranks[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].

The Heartbleed vulnerability (CVE-2014-0160) exemplifies the consequences of memory unsafety. A missing bounds check in OpenSSL's heartbeat extension allowed attackers to read up to 64KB of server memory per request, potentially exposing private keys and user credentials[4]. The fix was straightforward—adding a bounds check—but the damage from this single missing validation affected millions of systems[5].

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 operation, 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 their overhead makes them unsuitable for systems programming.

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[6]. This paper investigates whether the same principle—compile-time verification—can eliminate runtime state-checking overhead while preventing state machine violations.

II. BACKGROUND

A. Memory Safety and State Management

Google's research defines memory safety bugs as arising "when a program allows statements to execute that read or write memory, when the program is in a state where the memory access constitutes undefined behavior"[7]. When such statements are reachable under adversarial control, they often represent exploitable vulnerabilities.

State management is closely related. A file handle, for instance, must transition through defined states: closed → open → readable → closed. Reading from a closed handle or closing an already-closed handle represents invalid state transitions that may cause undefined behavior or resource leaks.

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, the compiler cannot verify that programmers consistently perform these checks.

B. The Typestate Pattern

The typestate pattern encodes an object's state in its type, making state transitions explicit in the type system[6]. 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[6]. The type parameters used in typestate patterns are erased during compilation. A `PhantomData<State>` field occupies zero bytes; it exists only to satisfy the type checker.

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

III. IMPLEMENTATION

To isolate state management from I/O overhead, we implement a minimal file handle abstraction in three variants. 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 read_file(file_handle_t* h) {
11     if (h->state != STATE_OPEN) {
12         return -1; // Invalid state
13     }
14     h->state = STATE_READABLE;
15     h->data = 42;
16     return h->data;
17 }
```

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 read_file(file_handle_t* h) {
6     h->data = 42;
7     return h->data;
8 }
```

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<Open> {
11     fn read(self) -> FileHandle<Readable> {
12         FileHandle {
13             data: 42,
14             _state: PhantomData,
15         }
16     }
17 }
```

Listing 3: Rust typestate pattern

The `PhantomData<State>` field is a zero-sized type marker that exists only for the type checker. The `read` method is only defined for `FileHandle<Open>`; calling it on other states produces a compile error. Crucially, consuming `self` by value prevents reuse of the old handle after transition.

IV. ANALYSIS

A. Compile-Time Guarantees

The Rust implementation rejects invalid state sequences at compile time. Attempting to compile:

```
1 let f = FileHandle::<Closed>::new();
2 let f = f.read(); // Error: no method `read`
3 // for FileHandle<Closed>
```

produces an error because `read` is not implemented for `FileHandle<Closed>`. The defensive C version compiles equivalent code without complaint, deferring the error to runtime.

B. Expected Assembly Characteristics

Based on Rust's zero-cost abstraction principle, we expect the following characteristics in optimized assembly:

The defensive C implementation should contain comparison instructions (`cmp`) and conditional jumps (`je`, `jne`) for each state check. It should also include instructions to load and store the state field.

The minimal C implementation should contain only the essential operations: memory allocation, data assignment, and deallocation. No comparisons or conditional branches for state validation.

The Rust implementation, if zero-cost abstractions hold, should produce assembly equivalent to minimal C. The `PhantomData` fields should be completely erased, and no state-checking code should appear since validity is guaranteed at compile time.

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[8]. 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.

V. CONCLUSION

This work 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.

If Rust's zero-cost abstraction principle holds—and preliminary analysis suggests it should—the typestate implementation will produce assembly equivalent to minimal C while providing guarantees stronger than defensive C. This would demonstrate that the traditional safety-performance tradeoff is not fundamental: with sufficiently expressive type systems, safety can be a compile-time property with no runtime cost.

Future work will provide detailed assembly comparisons across multiple optimization levels and compiler versions, quantifying instruction counts and branch frequencies to validate these expectations empirically.

REFERENCES

- [1] “CWE - 2024 CWE top 10 KEV weaknesses,” Accessed: Dec. 10, 2025. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_kev_list.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. “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] D. A. Wheeler. “The heartbleed bug: How to protect yourself,” Accessed: Dec. 10, 2025. [Online]. Available: <https://web.archive.org/web/20170202064748/https://www.dwheeler.com/essays/heartbleed.html>.
- [5] V. Teague. “How the heartbleed bug reveals a flaw in online security,” Accessed: Dec. 10, 2025. [Online]. Available: <https://web.archive.org/web/20140417090409/http://theconversation.com/how-the-heartbleed-bug-reveals-a-flaw-in-online-security-25536>.
- [6] Rust Embedded Working Group. “Zero cost abstractions,” Accessed: Dec. 10, 2025. [Online]. Available: <https://doc.rust-lang.org/beta/embedded-book/static-guarantees/zero-cost-abstractions.html>.
- [7] Google, “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>.
- [8] 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>.