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*—component, formatting, style, styling, insert.**

I. Introduction

This document is a model and instructions for LaTeX. Please observe the conference page limits. For more information about how to become an IEEE Conference author or how to write your paper, please visit the IEEE Conference Author Center website: https://conferences.ieeeauthorcenter.ieee.org/. See.

References

[1] "CWE - 2024 CWE top 10 KEV weaknesses," CWE - 2024 CWE Top 10 KEV Weaknesses, Accessed: Dec. 10, 2025. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html.