How Safe is C to Rust Translation Using LLMs?

Kenneth Fulton
Dept. of CEMS
Texas A&M University - San Antonio
San Antonio, US
kfult01@jaguar.tamu.edu

Joshua Ibrom
Dept. of CEMS
Texas A&M University - San Antonio
San Antonio, US
jibro01@jaguar.tamu.edu

***Abstract*—The increasing adoption of Rust has prompted efforts to rewrite insecure C codebases, traditionally used for low-level applications, into safer Rust equivalents. While manual C-to-Rust translation is labor-intensive and error-prone, Large Language Models (LLMs) offer a promising automated alternative, as explored in initiatives like DARPA's TRanslating All C TO Rust (TRACTOR) project [1]. This study investigates the safety and reliability of LLM-based C-to-Rust translation, focusing on ChatGPT's ability to handle C code with memory safety errors. Results reveal persistent challenges, including missing crate dependencies, runtime panics from memory or index errors, and warnings for unused imports or deprecated methods. Despite adding common crates to mitigate dependency issues, LLM translation succeeded in less than half of small programs, aligning with prior findings that only 11% of C pointers convert to safe Rust references [2]. These outcomes emphasize the need for hybrid approaches combining other C to Rust conversion tools and manual refinement to achieve safe Rust code.**

***Index Terms*—programming language translation, ai safety**

I. Background

Traditional systems-level programming of embedded software, operating systems, and other low-level applications has been done with language such as C so that memory and other system components may be managed by the programmer as needed. With this form of programming also comes the potential for pitfalls that leave applications "open to exploits, crashes, or corruption" [3]. With the increasing popularity of Rust, a systems-level programming language promising safety from the common C / C++ pitfalls, there is now the question of how we might perform some re-write of critical system software from their potentially insecure C versions to ones written in Rust [3], [4].

One such method of performing a re-write of some codebase is to attempt to hand-roll a new version of that software by manually translating line-by-line given C code to Rust code—a method that is time consuming, expensive, and may lead to there being missing or incorrectly-implemented blocks of code. Another such approach that has now become far more feasible with the prevalence and advanced reasoning abilities of *large language models* (LLMs) is to use those LLMs to perform automatic translation with human verification.

A. Research Questions

1) Can LLMs resolve memory safety errors when converting C code to Rust?
2) How efficiently can LLMs translate C code to Rust?
3) How can you prove that Rust code is safe?

II. Related Work

Rust's rich type system and ownership model guarantee memory and thread safety, delivering high-performance applications with reduced risks of exploits, crashes, or corruption. Leveraging these benefits, the Defense Advanced Research Projects Agency (DARPA) has initiated the TRanslating All C TO Rust (TRACTOR) project, using Large Language Models (LLMs) to convert C codebases into Rust, aiming to produce secure, high-quality Rust code matching the expertise of skilled Rust developers [1]. However, translating unsafe C code to safe Rust remains challenging, as static analysis struggles to convert most C pointers to safe Rust references (only $\sim$11%) and LLM-based auto-translation succeeds in less than half of small C programs as found by Li et al. [5], necessitating advanced techniques like static analysis, LLM repair, and manual intervention to preserve C code properties in well-typed Rust programs [6]. Some common C-to-Rust conversion tools include C2Rust for transpilation, LLMs for experimental translation, Rust Analyzer for refactoring and of course manual rewriting. Each approach alone faces challenges like pointer aliasing and limited automation, but hybrid strategies may help achieve safe, idiomatic Rust code. In this study, we evaluate the performance of ChatGPT, a widely used LLM for C-to-Rust conversion, focusing on its ability to handle C code containing memory safety issues.

III. Proposed Approach

In this experiment, we created C code with memory errors and collected samples from a GitHub dataset [7]. A Python script was used to translate each `.c` file into Rust using two prompting strategies: the first instructed the LLM to "Convert the following C code to Rust, providing only the complete Rust code in plaintext," (hereinafter referred to as the *Simple Prompt*) while the second added "while preserving all memory safety errors, if feasible" (hereinafter referred to as the *Unsafe Prompt*). The resulting `.rs` files were stored in an outputs directory, and a testing Python script executed each Rust file, recording whether it ran successfully, encountered compilation errors, or produced runtime errors (See Figure 1).

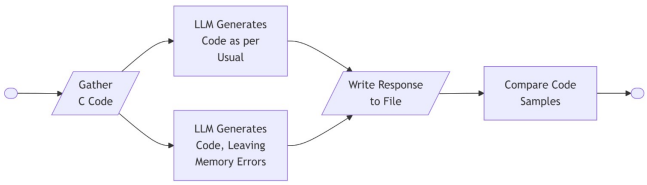

Fig. 1: Experimental Approach

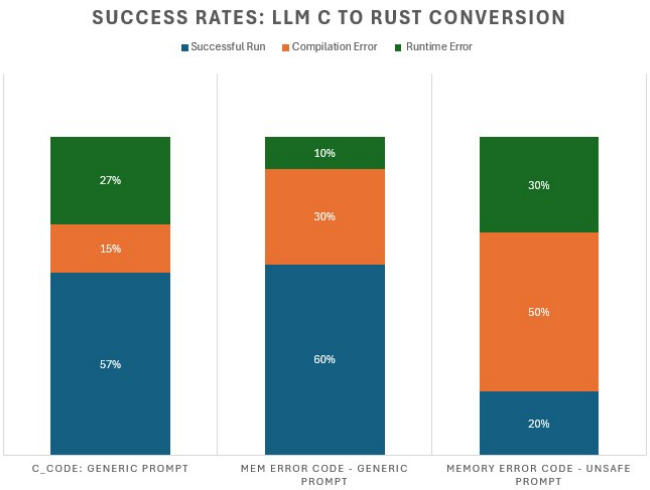

Fig. 2: Rust Execution Success Rates

IV. Experiment Outcomes and Observations

Several recurring issues were observed during the C-to-Rust conversion process. Common problems included mismatched types, particularly in function arguments and return values, as well as missing dependencies, where files attempted to use crates such as `libc`, `chrono`, and `users` without proper linkage or inclusion in the `Cargo.toml` file. This was circumvented in part by adding a list of common crates to the `Cargo.toml` file referenced by the python testing script. These included `serde`, `tokio`, `anyhow`, `thiserror`, `lazy_static`, `log`, `once_cell`, `rayon`, `derive_builder`, `libc`, `users` and `chrono`. Additionally, runtime panics frequently occurred due to memory-related errors, index out-of-bounds violations, or thread-related issues. Furthermore, many generated Rust files triggered warnings related to unused imports, unused variables, or the use of deprecated methods.

In some instances, such as Rust code samples which were generated using the Unsafe Prompt, these errors were to be expected. For example, the code shown in Listing 1 generated using the Unsafe Prompt contains an assignment to an index greater than the defined capacity of the `id_sequence` array which then results in a compilation error—an issue that a C compiler likely would not catch, leading to undefined behavior. Through this example, we might also observe how the LLM caught the issue and expanded the size of the array to accommodate the extra element.

Original C Code:

```c
/**
 * Source:
 ↪  https://cwe.mitre.org/data/definitions/787.html
 */

int main() {
  int id_sequence[3];

  /* Populate the id array. */

  id_sequence[0] = 123;
  id_sequence[1] = 234;
  id_sequence[2] = 345;
  id_sequence[3] = 456;
}
```

Rust Code Generated using the Safe Prompt:

```rust
fn main() {
    let mut id_sequence: [i32; 4] = [0; 4]; //
    ↪  Change size to 4 to accommodate the last
    ↪  index

    // Populate the id array
    id_sequence[0] = 123;
    id_sequence[1] = 234;
    id_sequence[2] = 345;
    id_sequence[3] = 456;
}
```

Rust Code Generated using the Unsafe Prompt:

```rust
fn main() {
    let mut id_sequence = [0; 3];

    // Populate the id array.
    id_sequence[0] = 123;
    id_sequence[1] = 234;
    id_sequence[2] = 345;
    id_sequence[3] = 456; // This will cause a panic
    ↪  in Rust due to out-of-bounds access
}
```

Source Code 1: C Code and its Translated Rust Code

Other samples of generated Rust code, even for those which were generated using the Safe Prompt, did not fit within the scope of being idempotent, leaving there to be the potential of memory errors are C strings are still used in place of Rust string slices and copies.

V. Conclusion

From our research we have been able to find that there does exist the potential of using LLMs to translate code from C, a language which in modern times has been referred to as “unsafe”, to Rust which has appeared to be elevated to a status of being the new standard for developing “safe” applications. Large language models do appear to understand the context of more basic C code and are able to translate that code with relative safety even when not strictly prompted to do so. However, other methods of translation must still be considered, such as requesting that the code be more idiomatic or providing some entirely different representation of the C code to the LLM, such as an abstract syntax tree (AST) or intermediate representation (IR). Moreover, further research must be determine methods to verifiably prove that a given sample of Rust code is “safe”, at least within a general margin of safety. There is much promise in cross-programming

language translation and therefore there is also still more work to be done to guarantee safer methods of refactoring and rebuilding potentially unsafe legacy codebases.

REFERENCES

[1] DARPA, "Translating all c to rust," 2025, accessed: 2025-04-30. [Online]. Available: https://www.darpa.mil/research/programs/translating-all-c-to-rust

[2] M. Emre, P. Boyland, A. Parekh, R. Schroeder, K. Dewey, and B. Hardekopf, "Aliasing limits on translating c to safe rust," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 551–579, 2023.

[3] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018.

[4] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.

[5] R. Li, B. Wang, T. Li, P. Saxena, and A. Kundu, "Translating c to rust: Lessons from a user study," *arXiv preprint arXiv:2411.14174*, 2024.

[6] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, "Towards translating real-world code with llms: A study of translating to rust," *arXiv preprint arXiv:2405.11514*, 2024.

[7] G. Thakur, "beginners-c-program-examples," https://github.com/gouravthakur39/beginners-C-program-examples, 2025, accessed: 2025-04-28.