SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification

Tianyang Zhou[1], Ziyi Zhang[2], Haowen Lin[1], Somesh Jha[2,3], Mihai Christodorescu[3], Kirill Levchenko[1], Varun Chandrasekaran[1]

[1]University of Illinois Urbana-Champaign, [2]University of Wisconsin–Madison, [3]Google

tz64@illinois.edu, ziyi.zhang2@wisc.edu, haowenl3@illinois.edu, jha@cs.wisc.edu, christodorescu@google.com, klevchen@illinois.edu, varunc@illinois.edu

Abstract

Translating software written in C to Rust has significant benefits in improving memory safety. However, manual translation is cumbersome, error-prone, and often produces unidiomatic code. Large language models (LLMs) have demonstrated promise in producing idiomatic translations, but offer no correctness guarantees. We propose SACTOR, an LLM-driven C-to-Rust translation tool that employs a two-step process: an initial "unidiomatic" translation to preserve interface, followed by an "idiomatic" refinement to align with Rust standards. To validate correctness of our function-wise incremental translation that mixes C and Rust, we use end-to-end testing via the foreign function interface. We evaluate SACTOR on 200 programs from two public datasets and on two more complex scenarios (a 50-sample subset of CRust-Bench and the `libogg` library), comparing multiple LLMs. Across datasets, SACTOR delivers high end-to-end correctness and produces safe, idiomatic Rust with up to 7× fewer Clippy warnings; On CRust-Bench, SACTOR achieves an average (across samples) of 85% unidiomatic and 52% idiomatic success, and on `libogg` it attains full unidiomatic and up to 78% idiomatic coverage on GPT-5.

1 Introduction

C is widely used due to its ability to directly manipulate memory and hardware (Love, 2013). However, manual memory management leads to vulnerabilities such as buffer overflows, dangling pointers, and memory leaks (Fan et al., 2020). Rust addresses these issues by enforcing memory safety through a strict ownership model without garbage collection (Matsakis and Klock, 2014), and has been adopted in projects like the Linux kernel[1] and Mozilla Firefox. *Translating legacy C code into idiomatic Rust improves safety and maintainability, but manual translation is error-prone, slow, and requires expertise in both languages.* Beyond type-system benefits, recent evidence shows that the choice of target language significantly affects the vulnerability landscape of translated code: migrating from C to memory-safe languages like Rust can substantially reduce vulnerability classes such as buffer overflows (Wu et al., 2025).

Automatic tools such as C2Rust (Immunant, 2020) generate Rust by analyzing C ASTs, but rule-based or static approaches (Zhang et al., 2023; Immunant, 2020; Emre et al., 2021; Hong and Ryu, 2024; Ling et al., 2022) typically yield unidiomatic code with heavy use of `unsafe`. Given semantic differences between C and Rust, idiomatic translations are crucial for compiler-enforced safety, readability, and maintainability.

Large language models (LLMs) show potential for capturing syntax and semantics (Pan et al., 2023), but they hallucinate and often generate incorrect or unsafe code (Perry et al., 2023). In C-to-Rust translation, naive prompting produces unsafe or semantically misaligned outputs. Prior work has explored prompting strategies (Shetty et al., 2024; Nitin et al., 2025; Shiraishi and Shinagawa, 2024) and verification methods such as fuzzing and symbolic execution (Yang et al., 2024; Eniser et al., 2024). While these improve correctness, they struggle with complex programs and rarely yield idiomatic Rust. For example, Vert (Yang et al., 2024) fails on programs with complex data structures, and C2SaferRust (Nitin et al., 2025) still produces Rust with numerous `unsafe` blocks. Crucially, many of these pipelines translate or refine the entire program at once, limiting both error localization and idiomatic redesign. SACTOR instead translates in dependency order at the function level, verifying each fragment via FFI before proceeding, which enables targeted repair and bottom-up API formation (§ 3).

In this paper, we introduce SACTOR, a

[1]https://github.com/Rust-for-Linux/linux

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 646–673
July 2-7, 2026

structure-aware, LLM-driven C-to-Rust translator (Figure 1). SACTOR follows a two-stage pipeline:

1. **C → Unidiomatic Rust:** Interface-preserving translation that may use `unsafe` for low-level operations.
2. **Unidiomatic → Idiomatic Rust:** Behaviorally-equivalent translation that refines to Rust idioms, eliminating `unsafe` and migrating C API patterns to Rust equivalents.

Static analysis of C code (pointer semantics, dependencies) guides both stages. To verify correctness, we embed the translated Rust with the original C via the Foreign Function Interface (FFI), enabling end-to-end testing on both stages and accept a stage when all end-to-end tests can pass. This decomposition separates syntax from semantics, simplifies the LLM task, and ensures more idiomatic, memory-safe Rust[2]. An example of SACTOR translation process is in Appendix F. Prompts that used in the pipeline are in Appendix N.

LLM orchestration. SACTOR places the LLM inside a neuro-symbolic feedback loop. Static analysis and a machine-readable interface specification guide prompting; compiler diagnostics and end-to-end tests provide structured feedback. In the idiomatic verification phase, a rule-based harness generator with an LLM fallback completes the feedback loop. This design first ensures semantic correctness in unidiomatic Rust, then refines it into idiomatic Rust, with both stages verifiable in a unified two-step process.

Our contributions are as follows:

- **Method:** An LLM-orchestrated, structure-aware two-phase pipeline that separates semantic preservation from idiomatic refinement, guided by static analysis (§ 4)
- **Verification:** SACTOR verifies both unidiomatic and idiomatic translations via FFI-based testing. During idiomatic verification, it uses a co-produced interface specification to synthesize C/Rust harnesses with an LLM fallback for missing patterns; compiler and test feedback are structured into targeted prompt repairs (§ 4.3).
- **Evaluation:** Across two datasets (200 programs) and five LLMs, SACTOR reaches 93% / 84% end-to-end correctness (DeepSeek-R1) and improves idiomaticity (§ 6.2). On CRust-Bench (50 samples), unidiomatic translation averages 85% *function-level success rate* across all samples (82% aggregated across functions), with 32/50 samples fully translated; idiomatic success is computed on those 32 samples and averages 52% (43% aggregated; 8/32 fully idiomatic). On `libogg` (77 functions), the *function-level success rate* is 100% for unidiomatic and 53% and 78% for idiomatic across GPT-4o and GPT-5, respectively (§ 6.3).
- **Diagnostics:** We analyze efficiency, feedback, temperature sensitivity, and failure cases: GPT-4o is the most token-efficient, compilation/testing feedback boosts weaker models by 17%, temperature has little effect, and reasoning models like DeepSeek-R1 excel on complex bugs such as format-string and array errors (Appendix H).

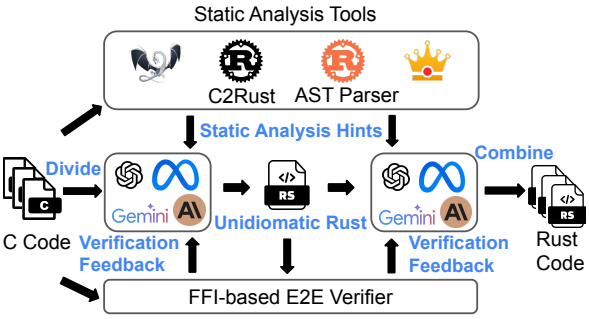


Figure 1: Overview of the SACTOR methodology.

2 Background

Primer on C and Rust: C is a low-level language that provides direct access to memory and hardware through pointers and abstracts machine-level instructions (TIOBE, 2025). While this makes it efficient, it suffers from memory vulnerabilities (MITRE, 2006a,b,d,c). Rust, in contrast, provides memory safety without additional performance penalty, *and* has the same ability to access low-level hardware as C; it enforces strict compile-time memory safety through *ownership*, *borrowing*, and *lifetimes* to eliminate memory vulnerabilities (Matsakis and Klock, 2014; Jung et al., 2017).

Challenges in Code Translation: Despite its advantages, and since Rust is relatively new, many widely used system-level programs remain in C. It is desirable to translate such programs to Rust, but the process is challenging due to fundamental language differences. Figure 5 in Appendix A shows an example of a simple C program and its Rust equivalent to illustrate the differences between two languages in terms of memory management and error handling. While Rust permits `unsafe` blocks for C-like pointer operations, their use is

[2]SACTOR code is available at https://github.com/qsdrqs/sactor and datasets are available at https://github.com/qsdrqs/sactor-datasets

discouraged due to the absence of compiler guarantees and their non-idiomatic nature for further maintenance[3].

3 Related Work

LLMs for C-to-Rust Translation: Vert (Yang et al., 2024) combines LLM-generated candidates with fuzz testing and symbolic execution to ensure equivalence, but this strict verification struggles with scalability and complex C features. Flourine (Eniser et al., 2024) incorporates error feedback and fuzzing, using data type serialization to mitigate mismatches, yet serialization issues still account for nearly half of errors. Shiraishi and Shinagawa (2024) decompose C programs into sub-tasks (e.g., macros) and translate them with predefined Rust idioms, but evaluate only compilation success without functional correctness. Shetty et al. (2024) employ dynamic analysis to capture runtime behavior as translation guidance, but coverage limits hinder generalization across execution paths. Nitin et al. (2025) refine C2Rust outputs with LLMs to reduce unidiomatic constructs (`unsafe`, `libc`), but remain constrained by C2Rust's preprocessing, which strips comments and directives (§ 4.2) and reduces context for idiomatic translation.

Non-LLM Approaches for C-to-Rust Translation: C2Rust (Immunant, 2020) translates by converting C ASTs into Rust ASTs and applying rule-based transformations. While syntactically correct, the results are structural translations that rely heavily on `unsafe` blocks and explicit type conversions, yielding low readability. Crown (Zhang et al., 2023) introduces static ownership tracking to reduce pointer usage in generated Rust code. Hong and Ryu (2024) focus on handling return values in translation, while Ling et al. (2022) rely on rules and heuristics. Although these methods reduce some `unsafe` usage compared to C2Rust, the resulting code remains largely unidiomatic.

Positioning of SACTOR: Many prior LLM-based pipelines, including C2SaferRust, translate or refine the program as a whole and rely on post-hoc LLM repair to improve code quality. Because the repair stage operates on complete programs, it offers limited error localization and often yields only modest idiomaticity gains; indeed, Figure 4 shows that C2SaferRust can produce more Clippy warnings than SACTOR's unidiomatic output alone. In contrast, SACTOR translates code in dependency order at the function level, verifying each fragment via FFI-based end-to-end tests before moving to the next. This modular, bottom-up design provides precise fault attribution (a failing test identifies exactly the fragment under translation), enables incremental API formation (later functions build on already-verified Rust interfaces), and creates a stronger starting point for idiomatic refinement.

4 SACTOR Methodology

We propose SACTOR, an LLM-driven C-to-Rust translation tool using a two-step translation methodology. As Rust and C differ substantially in semantics (§ 2), SACTOR augments the LLM with static-analysis-derived "hints" that capture semantic information in the C code. At a high level, SACTOR first uses its `C Parser` to split the input into fragments (types, globals, functions) and computes a dependency order over them. It then translates the program bottom-up: each fragment is translated by the LLM, compiled, and linked back into the remaining C program via FFI for validation: the project's end-to-end tests are run on this mixed C/Rust build, and only when all tests pass is the translation accepted and the system proceeds to the next fragment. If a fragment fails verification, structured feedback (compiler errors or execution traces) is returned to the LLM for repair, up to a fixed attempt budget. Once all fragments have been translated into unidiomatic Rust, the same bottom-up, fragment-level procedure is applied for idiomatic refinement under the same verification harness. The four stages of this pipeline are detailed below.

4.1 Task Division

We begin by dividing the program into smaller parts that can be processed by the LLM independently. This enables the LLM to focus on a narrower scope for each translation task and ensures the program fits within its context window. This strategy is supported by studies showing that LLM performance degrades on long-context understanding and generation tasks (Liu et al., 2024; Li et al., 2024). By breaking the program into smaller pieces, we can mitigate these limitations and improve performance on each individual task. To facilitate task division and extract relevant language information – such as

[3] Other differences include string representation, pointer usage, array handling, reference lifetimes, and error propagation. A non-exhaustive summary appears in Appendix A.

definitions, declarations, and dependencies – from C code, we developed a *static analysis tool* called `C Parser` based on `libclang` (a library that provides a C compiler interface, allowing access to semantic information of the code).

Our `C Parser` analyzes the input program and splits the program into fragments consisting of a single type, global variable, or function definition. This step also extracts semantic dependencies between each part (e.g., a function definition depending on a prior type definition). We then process each program fragment in dependency order: all dependencies of a code fragment are processed before the fragment. Concretely, `C Parser` constructs a directed dependency graph whose nodes are types, global variables, and functions, and whose edges point from each item to the items it directly depends on. We compute a translation order by repeatedly selecting items whose dependencies have already been processed. If the dependency graph contains a cycle, SACTOR currently treats this as an unsupported case and terminates with an explicit error. In addition, to support real-world C projects, SACTOR makes use of the C project compile commands generated by the `make` tool and performs preprocessing on the C source files. In Appendix B, we provide more details on how we preprocess source files and divide programs.

4.2 Translation

To ensure that each program fragment is translated only *after* its dependencies have been processed, we begin by translating data types, as they form the foundational elements for functions. This is followed by global variables and functions. We divide the translation process into two steps.

Step 1. Unidiomatic Rust Translation: We aim to produce interface equivalent Rust code from the original C code, which allows the use of `unsafe` blocks to do pointer manipulations and C standard library functions while keeping the same interface as original C code. For data type translation, we leverage information from C2Rust (Immunant, 2020) to help the conversion. While C2Rust provides reliable data type translation, *it struggles with function translation* due to its compiler-based approach, which omits source-level details like comments, macros, and other elements. These omissions significantly reduce the readability and usability of the generated Rust code. Thus, we use C2Rust only for data type translation, and use an LLM to translate global variables and functions. For functions, we rely on our `C Parser` to automatically extract dependencies (e.g., function signatures, data types, and global variables) and reference the corresponding Rust code. This approach guides the LLM to accurately translate functions by leveraging the previously translated components and directly reusing or invoking them as needed.

Step 2. Idiomatic Rust Translation: The goal of this step is to refine unidiomatic Rust into idiomatic Rust by removing `unsafe` blocks and following Rust idioms. This stage focuses on rewriting behavioral-equivalent but low-level constructs into type-safe abstractions while preserving behavior verified in the previous step. Handling pointers from C code is a key challenge, as they are considered unsafe in Rust. Unsafe pointers should be replaced with Rust types such as references, arrays, or owned types. To address this, we use Crown (Zhang et al., 2023) to facilitate the translation by analyzing pointer mutability, fatness (e.g., arrays), and ownership. This information provided by Crown helps the LLM assign appropriate Rust types to pointers. Owned pointers are translated to `Box`, while borrowed pointers use references or smart pointers. Crown assists in translating data types like `struct` and `union`, which are processed first as they are often dependencies for functions. For function translations, Crown analyzes parameters and return pointers, while local variable pointers are inferred by the LLM. Dependencies are extracted using our `C Parser` to guide accurate function translation. The idiomatic code is produced together with an interface transformation specification, forms the input to the verification step in § 4.3.

4.3 Verification

To verify the equivalence between source and target languages, prior work has relied on symbolic execution and fuzz testing, are impractical for real-world C-to-Rust translation (details in Appendix D). We instead validate correctness through *soft equivalence*: ensuring functional equivalence of the entire program via end-to-end (E2E) tests. This avoids the complexity of generating specific inputs or constraints for individual functions and is well-suited for real-world programs where such E2E tests are often available and reusable. Correctness confidence in this framework depends on the code coverage of the E2E tests: the broader the coverage, stronger the assurance of equivalence. Importantly, SACTOR's fragment-level FFI-based testing re-

alizes a form of *compositional verification* (Anshumaan et al., 2025): each translated fragment is verified independently against the full test suite before being committed, so local guarantees accumulate into whole-program assurance without requiring monolithic equivalence proofs.

Verifying Unidiomatic Rust Code. This is straightforward, as it is semantically equivalent to the original C code and maintains compatible function signatures and data types, which ensures a consistent Application Binary Interface (ABI) between the two languages and enabling direct use of the FFI for cross-language linking. The verification process involves two main steps: First, the unidiomatic Rust code is compiled using the Rust compiler to check for successful compilation. Then, the original C code is recompiled with the Rust translation linked as a shared library. This setup ensures that when the C code calls the target function, it invokes the Rust translation instead. To verify correctness, *E2E tests are run on the entire program*, comparing the outputs of the original C code and the unidiomatic Rust translation. If all tests pass, the target function is considered verified.

Verifying Idiomatic Rust Code. Idiomatic Rust diverges from the original C program in both types and function signatures, producing an *ABI mismatch* that prevents direct linking into the C build. We therefore verify it via a *synthesized, C-compatible test harness together with E2E tests*.

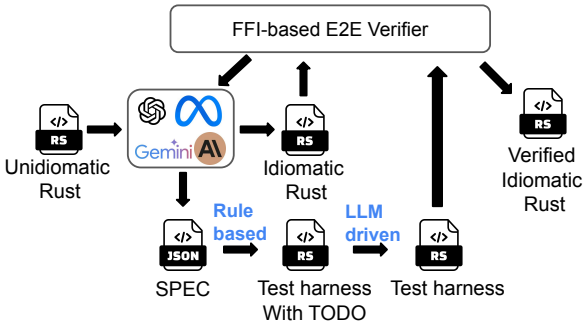


Figure 2: Spec-driven harness generation and verification loop. The idiomatic translator co-produces idiomatic Rust and a machine-readable SPEC.

During idiomatic translation, SACTOR co-produces a small, machine-readable *specification* (SPEC) for each function/struct. The SPEC captures, in a compact form, how C-facing values map to idiomatic Rust, including the expected pointer shape (`slice/cstring/ref`), where lengths come from (a sibling field or a constant), and basic nullability and return conventions; it also allows marking fields that should be compared in self-checks. A rule-based generator consumes the SPEC to synthesize a C-compatible harness that bridges from the C ABI to idiomatic code and backwards. Figure 2 shows the schematic, and Table 13 summarizes current supported patterns; Appendix L presents a detailed exposition of the SPEC-driven harness generation technique (rules and design choices), and Appendix E provides a concrete example of the generated harness. For structs, the SPEC defines bidirectional converters between the C-facing and idiomatic layouts, validated by a lightweight roundtrip test that checks the fields marked as comparable for consistency after conversion. When the SPEC includes a pattern the generator does not yet implement (e.g., aliasing/offset views or unsupported pointer kinds or types), we emit a localized TODO and use an LLM guided by the SPEC to fill only the missing conversions. Finally, we compile the idiomatic crate and the generated harness, link them into the original C build via FFI, and run the program's existing E2E tests; passing tests validate the idiomatic translation under the coverage of those tests, while failures trigger the feedback procedure in § 4.3.

Feedback Mechanism. For failures, we feed structured signals back to translation: compiler errors guide fixes for build breaks; for E2E failures we use the Rust procedural macro to automatically instrument the target to log salient inputs/outputs, re-run tests, and return the traces to the translator for refinement. Algorithm 2 in Appendix C gives the detailed procedure.

4.4 Code Combination

By translating and verifying all functions and data types, we integrate them into a unified Rust codebase. We first collect the translated Rust code from each subtask and remove duplicate definitions and other redundancies required only for standalone compilation. The cleaned code is then organized into a well-structured Rust implementation of the original C program. Finally, we run end-to-end tests on the combined program to verify the correctness of the final Rust output. If all tests pass, the translation is considered successful.

5 Experimental Setup

5.1 Datasets Used

For the selection of datasets for evaluation, we consider the following criteria:

- *Sufficient Number:* The dataset should contain a substantial number of C programs to ensure a robust evaluation of the approach's performance across a diverse set of examples.
- *Presence of Non-Trivial C Features:* The dataset should include C programs with advanced features such as multiple functions, `structs`, and other non-trivial constructs as it enables the evaluation to assess the approach's ability to handle complex features of C.
- *Availability of E2E Tests:* The dataset should either include E2E tests or make it easy to generate them as this is essential for accurately evaluating the correctness of the translated code.

Based on the above criteria, we evaluate on two widely used program suites in the translation literature: TransCoder-IR (Szafraniec et al., 2023) and Project CodeNet (Puri et al., 2021). Complete details for these datasets are in Appendix G. For TransCoder-IR and CodeNet, we randomly sample 100 C programs from each (for CodeNet, among programs with external inputs) to ensure computational feasibility while maintaining statistical significance.

To better reflect the language features of real-world C codebases and allow test reuse (§ 6.3), we also evaluate on two targets: (i) a 50-sample subset of CRust-Bench (Khatry et al., 2025) and (ii) the `libogg` multimedia container library (Xiph, 2025). In CRust-Bench, we exclude entries outside our pipeline's scope (e.g., circular dependencies or compiler-specific intrinsics). `libogg` is a real-world C project of about 2,000 lines of code with 77 functions involving non-trivial `structs`, `buffers`, and pointer manipulation. Both benchmarks reuse their upstream end-to-end tests to verify the translated code.

5.2 Evaluation Metrics

Success Rate: At each translation stage (unidiomatic or idiomatic), the fraction of programs whose translation is produced and passes the E2E tests. To enable the LLMs to utilize feedback from previous failed attempts, we allow the LLM to make up to 6 attempts for each translation process.
Idiomaticity: To evaluate the idiomaticity of the translated code, we use three metrics:

- *Lint Alert Count* is measured by running Rust-Clippy (The Rust Project Contributors), a tool that provides lints on unidiomatic Rust (including improper use of `unsafe` code and other common style issues). By collecting the warnings and errors generated by Rust-Clippy for the translated code, we can assess its idiomaticity: fewer alerts indicate more idiomaticity. Previous translation works (Yang et al., 2024; Eniser et al., 2024) have also used Rust-Clippy.
- *Unsafe Code Fraction*, inspired by Shiraishi and Shinagawa (2024), is defined as the ratio of tokens inside `unsafe` code blocks or functions to total tokens for a single program. High usage of `unsafe` is considered unidiomatic, as it bypasses compiler safety checks, introduces potential memory safety issues and reduces code readability.
- *Unsafe Free Fraction* indicates the percentage of translated programs in a dataset that do not contain any `unsafe` code. Since `unsafe` code represents potential points where the compiler cannot guarantee safety, this metric helps determine the fraction of results that can be achieved without relying on `unsafe` code.

5.3 LLMs Used

We evaluate 6 models across different experiments. Table 1 shows our configurations for different LLMs in evaluation. On the two datasets (TransCoder-IR and CodeNet) we use four non-reasoning models—GPT-4o, Claude 3.5 Sonnet, Gemini 2.0 Flash, and Llama 3.3 70B, and one reasoning model DeepSeek-R1. For real-world codebases, we run GPT-4o on CRust-Bench and run both GPT-4o and GPT-5 on `libogg`.

Model	Version	Temperature
GPT-4o	gpt-4o-2024-08-06	0
Claude 3.5 Sonnet	claude-3-5-sonnet-20241022	0
Gemini 2.0 Flash	gemini-2.0-flash-exp	0
Llama 3.3 Instruct 70B	Llama 3.3 Instruct 70B[1]	0
DeepSeek-R1	DeepSeek-R1 671B[2]	0
GPT-5	gpt-5-2025-08-07	default

[1] https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct
[2] https://huggingface.co/deepseek-ai/DeepSeek-R1

Table 1: Configurations of Different LLMs in Evaluation. All other hyperparameters (e.g., Top-P, Top-K) use provider defaults. As GPT-5 does not support temperature setting, we use its default temperature.

6 Evaluation

Through our evaluation, we answer: (1) How successful is SACTOR in generating idiomatic Rust code using different LLM models?; (2) How idiomatic is the Rust code produced by SACTOR

compared to existing approaches?; and (3) How well does SACTOR generalize to real-world C codebases?

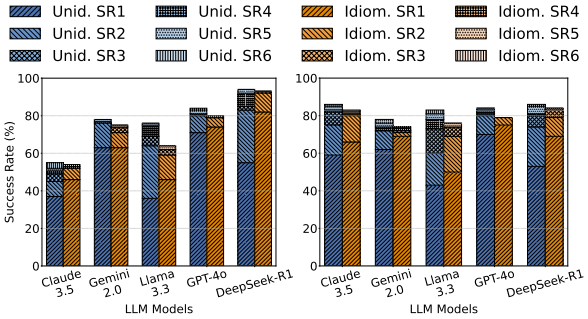
Our results show that: (1) DeepSeek-R1 achieves the highest success rates (93%) with SACTOR on TransCoder-IR and also reaches the highest success rates (84%) on Project CodeNet (§ 6.1), while failure reasons vary between datasets and models (Appendix H); (2) SACTOR's idiomatic translation outperforms prior baselines on idiomaticity and safe-code generation, producing Rust code with fewer Clippy warnings and 100% unsafe-free translations (§ 6.2); and (3) For real-world codebases (§ 6.3), SACTOR attains strong unidiomatic success and moderate idiomatic success: on CRust-Bench, unidiomatic averages 85% across 50 samples (82% aggregated across 966 functions; 32/50 fully translated) and idiomatic averages 52% across 32 samples that fully translated into unidiomatic Rust (43% aggregated across 580 functions; 8/32 fully translated); on `libogg` unidiomatic reaches 100% and idiomatic spans 53% and 78% for GPT-4o and GPT-5, respectively. Failures concentrate at ABI/type boundaries and harness synthesis (pointer/slice shape, length sources, lifetime or mutability), with additional cases from unsupported features and borrow/ownership pitfalls. Overall, improving the model itself alleviates a subset of failure modes; for instance, on `libogg`, GPT-5 cuts idiomatic compile-error failures from six to one relative to GPT-4o. For a fixed model, strengthening the framework and interface rules also improves outcomes but remains limited when confronted with previously unseen patterns.

We also evaluate the computational cost of SACTOR (Appendix I), the impact of the feedback mechanism (Appendix J), and temperature settings (Appendix K) . GPT-4o and Gemini 2.0 achieve the best cost-performance balance, while Llama 3.3 consumes the most tokens among non-reasoning models. DeepSeek-R1 uses 3-7 $\times$ more tokens than others. The feedback mechanism boosts Llama 3.3's success rate by 17%, but has little effect on GPT-4o, suggesting it benefits lower-performing models more. Temperature has minimal impact.

6.1 Success Rate Evaluation

We evaluate the success rate (as defined in § 5.2) for the two datasets on different models. For idiomatic translation, we also plot how many attempts are needed.

(1) TransCoder-IR (Figure 3a): DeepSeek-R1 achieves the highest success rate (SR) in both unidiomatic (94%) and idiomatic (93%) steps, only 1% drops in the idiomatic translation step, demonstrating strong consistency in code translation. GPT-4o follows with 84% in the unidiomatic step and 80% in the idiomatic step. Gemini 2.0 comes next with 78% and 75%, respectively. Claude 3.5 struggles in the unidiomatic step (55%) but does not show substantial degradation when converting unidiomatic Rust to idiomatic Rust (54%, only a 1% drop), but it is still the worst model compared to the others. Llama 3.3 performs well in the unidiomatic step (76%) but drops significantly in the idiomatic step (64%), and requiring more attempts for correctness.



(a) TransCoder-IR SR (b) CodeNet SR

Figure 3: Success rates (SR) across different LLM models for the TransCoder-IR and CodeNet datasets. SR 1-6 represent the number of attempts made to achieve a successful translation. Unid. and Idiom. denote unidiomatic and idiomatic translation steps, respectively.

(2) Project CodeNet (Figure 3b): DeepSeek-R1 again leads with 86% in the unidiomatic step and 84% in the idiomatic step, showing only a 2% drop in the idiomatic translation step. Claude 3.5 follows closely with 86% success rate in the unidiomatic step and 83% in the idiomatic step. GPT-4o performs consistently well in the unidiomatic step (84%) but drops to 79% in the idiomatic step, indicating a 5% drop between the two steps. Gemini 2.0 follows with 78% in the unidiomatic step and 74% in the idiomatic step, showing consistent performance between two datasets. Llama 3.3 still exhibits significant drops (83% to 76%) in both steps and finishes last in the idiomatic step.

The results demonstrates that DeepSeek-R1's SRs remain high and consistent–94%/93% (unidiomatic/idiomatic) on TransCoder-IR versus 86%/84% on CodeNet–while other models exhibit notable performance drops when moving to TransCoder-IR. This suggests that models with reasoning capabilities may be better for handling com-

plex code logic and data manipulation.

6.2 Measuring Idiomaticity

We compare our approach with four baselines: C2Rust (Immunant, 2020), Crown (Zhang et al., 2023), C2SaferRust (Nitin et al., 2025) and Vert (Yang et al., 2024). Of these baselines, C2Rust is the most versatile[4], supporting most C programs, while Crown is also broad but lacks support for some language features. C2SaferRust focuses on refining the unsafe code produced by C2Rust, allowing it to handle a wide range of C programs. In contrast, Vert targets a specific subset of simpler C programs. We assess the idiomaticity of Rust code generated by C2Rust, Crown, and C2SaferRust on both datasets. Since Vert produced Rust code only for TransCoder-IR, we evaluate it solely on this dataset. All the experiments are conducted using GPT-4o as the LLM for baselines and our approach, with max 6 attempts per translation.

Results: Figure 4 presents the lint alert count (sum up of Clippy warnings and errors count for a single program) across all approaches. C2Rust consistently exhibits high Clippy issues, and Crown shows little improvement over C2Rust, indicating both struggle to generate idiomatic Rust. C2SaferRust reduces Clippy issues, but it still retains a significant number of warnings and errors. Notably, even the unidiomatic output of SACTOR surpasses all of these 3. This underscores the advantage of LLMs over rule-based methods. While Vert improves idiomaticity, SACTOR's idiomatic phase yields fewer Clippy issues, outperforming some existing LLM-based approaches.

Table 2 summarizes unsafe code statistics. Unsafe-Free indicates the percentage of programs without unsafe code, while Avg. Unsafe represents the average proportion of unsafe code across all translations. C2Rust and Crown generate unsafe code in all programs with a high average unsafe percentage. C2SaferRust has the ability to reduce unsafe code and able to generate unsafe-free programs in some cases (45.6% in TransCoder-IR), but cannot sufficiently reduce the unsafe uses in the CodeNet dataset. Vert has a higher success rate than SACTOR but occasionally introduces unsafe code. SACTOR's unidiomatic phase retains C semantics, leading to a high unsafe percentage. However, its idiomatic phase eliminates all unsafe code, achieving a 100% Unsafe-Free rate.

[4]Versatility refers to an approach's applicability to diverse C programs.

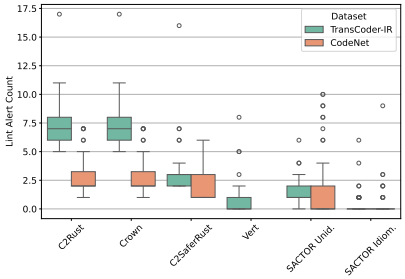


Figure 4: Total clippy issues (warnings + errors) across different method.

METHOD	DATASET	SR (%)	UF (%)	AU (%)
C2Rust	TransCoder-IR	100	0	100
	CodeNet	100	0	75.9
Crown	TransCoder-IR	100	0	100
	CodeNet	100	0	75.9
C2SaferRust	TransCoder-IR	90	45.6	10.8
	CodeNet	93	0	75.8
Vert	TransCoder-IR	92	95.7	1.6
SACTOR (Unid.)	TransCoder-IR	84	3.6	91.7
	CodeNet	84	1.1	42.7
SACTOR (Idiom.)	TransCoder-IR	80	**100**	**0**
	CodeNet	79	**100**	**0**

Table 2: Unsafe code statistics. UF denotes "Unsafe Free" and AU denotes "Avg. Unsafe."

6.3 Real-world Code-bases

To evaluate SACTOR's performance on two real-world code-bases, we run the translation process up to three times per sample, with SACTOR attempts to translate each function, struct and global variable at most six attempts in each run. For `libogg`, we also experiment with both GPT-4o and GPT-5 to compare their performance.

CRust-Bench. At the function level, SACTOR reaches a mean per-sample unidiomatic success rate of 85.15% (788 of 966 functions, 81.57% aggregated), with 32 samples fully translated. Restricting idiomatic evaluation to these 32 samples, the mean per-sample rate is 51.85% (249 of 580, 42.93% aggregated), and 8 samples are fully translated under the idiomatic setting. Table 3 summarizes stage-level outcomes.

Observations and failure modes. We organize failures into five main categories. *(1) Interface/name drift:* Symbol casing or exact-name mismatches (e.g., `CamelCase` vs. `snake_case`). *(2) Semantic mapping errors:* Mistakes in translating C constructs to idiomatic Rust (e.g., pointer-of-pointer vs. `Vec`, shape drift, lifetime or mutability issues). *(3) C-specific features:* Incomplete han-

Stage	Samples Eval.	Per-sample SR (func.)	Aggregated SR (func.)	Full SR	Avg. Lint / Function
Unidi.	50	85.15%	788 / 966 (81.57%)	32 / 50 (64.00%)	2.96
Idiom.	32†	51.85%	249 / 580 (42.93%)	8 / 32 (25.00%)	0.28

Table 3: CRust-Bench function-level translation results. Success rate (SR) is averaged per-sample; † idiomatic stage is evaluated only on samples whose unidiomatic pass fully translated all functions.

Step (model)	SR (%)	Avg. lint / Function	Avg. attempt
Unid. (GPT-4o)	100	1.45	1.52
Idiom. (GPT-4o)	53	0.28	2.00
Unid. (GPT-5)	100	1.45	1.04
Idiom. (GPT-5)	78	0.23	1.25

Table 4: Evaluation of SACTOR's function translation on libogg. "Unid."/"Idiom." denotes unidiomatic/idiomatic translation. "SR" is the success rate of translating functions. "Avg. lint"/"Avg. attempt" is the average lint alert count/average number of attempts, for functions that both LLM models succeed in translating.

dling some features like function pointers and C variadics. *(4) Borrowing and resource-model violations:* Compile-time borrow-checker errors in idiomatic Rust bodies (e.g., overlapping borrows in updates). *(5) Harness/runtime faults:* Faulty test harnesses translation (e.g. buffer mis-sizing, out-of-bounds access). Other minor cases include *unsupported intrinsics* (SIMD) and *global-state divergence* (shadowed globals). Table 14 (in Appendix M.1) summarizes each sample's outcome and its primary cause.

Idiomaticity. Unidiomatic outputs average 50.14 Clippy alerts per sample (2.96 per function) with a 97.86% unsafe fraction; idiomatic outputs drop to 2.27 alerts per sample (0.28 per function) and reach a 100% unsafe-free rate.

Libogg. The unidiomatic and idiomatic translations of all structs and global variables are successful with each LLM model. For functions, the result is summarized in Table 4. SACTOR succeeds in all functions' unidiomatic translations. For idiomatic translations, SACTOR's success rate is 53% and SACTOR takes 2.00 attempts on average to produce a correct translation with GPT-4o. For GPT-5, the performance is significantly better with a success rate of 78% and average number of attempts of 1.25.

Observations and failure modes. The most significant reasons for failed idiomatic translations include: *(1)* failure to pass tests due to mistakes in translating pointer manipulation and heap memory management; *(2)* compile errors in translated functions, especially arising from violation of Rust safety rules on lifetimes, borrowing and mutability; *(3)* failure to generate compilable test harnesses for data types with pointers and arrays. GPT-5 performs significantly better than GPT-4o. For example, GPT-5 only have one failure caused by a compile error in the translated function, in contrast to six compile error failures with GPT-4o, which shows the progress of GPT-5 in understanding Rust grammar and fixing compile errors. More details can be found in Appendix M.2.

Idiomaticity. SACTOR's unidiomatic translations cause lint alerts largely due to the use of `unsafe` code while idiomatic translations lead to very few lint alerts, *i.e.*, fewer than 0.3 alerts per function on average (Table 4). With each model, the unidiomatic translations are all in `unsafe` code but the idiomatic translations are all in `safe` code. As a result, the idiomatic translations have an avg. unsafe fraction of 0% and unsafe-free fraction of 100%. The unidiomatic translations are the opposite.

7 Conclusions

Translating C to Rust enhances memory safety but remains error-prone and often unidiomatic. While LLMs improve translation, they still lack correctness guarantees and struggle with semantic gaps. SACTOR addresses these through a two-stage pipeline: preserving ABI interface first, then refining to idiomatic Rust. Guided by static analysis and validated via FFI-based testing, SACTOR achieves high correctness and idiomaticity across multiple benchmarks, surpassing prior tools on idiomaticity and safe-code generation. On `libogg`, GPT-5 further lifts idiomatic success from 53% to 78% over GPT-4o, suggesting that SACTOR's effectiveness will grow with LLM capability. Remaining challenges include stronger correctness assurance, richer C-feature coverage, and improved scalability and efficiency (see §7).

Limitations

While SACTOR is effective in producing correct, idiomatic Rust, several limitations remain:

- **Model variance.** Translation quality depends on the underlying LLM. Stronger models meaningfully reduce failures: on `libogg`, GPT-5 raises idiomatic success from 53% to 78% and cuts compile-error failures from six to one relative to GPT-4o (§ 6.3). Older or weaker models, however, remain less accurate and less stable.
- **Unsupported C features.** Complex macros, pervasive function pointers, global state, C variadics and inline assembly are only partially handled, limiting applicability to such codebases (see § 6.3).
- **Harness generation stability.** The rule-based generator with LLM fallback can still emit incomplete or brittle adapters on complex patterns (e.g., unusual pointer shapes or length expressions), causing otherwise-correct translations to fail verification. Hardening rules and reducing reliance on the fallback should improve robustness and reproducibility.
- **Cost and latency.** Multi-stage prompting, compilation, and test loops incur non-trivial token and time costs, which matter for large-scale migrations.
- **Test coverage dependence.** Our soft-equivalence checks rely on existing end-to-end tests, which cannot guarantee full semantic equivalence (Anshumaan et al., 2025); shallow or incomplete coverage can miss subtle semantic errors. Integrating fuzzing, test generation, or formal verification could raise coverage and catch corner cases.
- **Static analysis precision.** Current analysis may under-specify aliasing, ownership, and pointer shapes in challenging code, leading to adapter/spec errors. Stronger analyses could improve mapping and reduce retries.

Acknowledgments

We thank the anonymous reviewers and the area chair for their valuable feedback on this paper. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590131. Approved for public release; distribution is unlimited.

References

Divyam Anshumaan, Tej Chajed, Varun Chandrasekaran, Alvin Cheung, Sarthak Choudhary, Adwait Godbole, Somesh Jha, Nils Palumbo, Elizabeth Polgreen, Sanjit A Seshia, and Tianyang Zhou. 2025. Llm-based code translation needs formal compositional reasoning. (UCB/EECS-2025-174).

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39.

P David Coward. 1988. Symbolic execution systems—a review. *Software Engineering Journal*, 3(6):229–239.

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating c to safer rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29.

Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards translating real-world code with llms: A study of translating to rust. *arXiv preprint arXiv:2405.11514*.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512.

Jaemin Hong and Sukyoung Ryu. 2024. Don't write, but return: Replacing output parameters with algebraic data types in c-to-rust translation. *Proceedings of the ACM on Programming Languages*, 8(PLDI):716–740.

Immunant. 2020. C2rust.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34.

Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. 2025. Crust-bench: A comprehensive benchmark for c-to-safe-rust transpilation. *arXiv preprint arXiv:2504.15254*.

James C King. 1976. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060*.

Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218.

Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. 2022. In rust we trust: a transpiler from unsafe c to safer rust. In *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, pages 354–355.

Xiang Liu, Peijie Dong, Xuming Hu, and Xiaowen Chu. 2024. Longgenbench: Long-context generation benchmark. *arXiv preprint arXiv:2410.04199*.

Robert Love. 2013. *Linux system programming: talking directly to the kernel and C library*. " O'Reilly Media, Inc.".

Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104.

Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44.

MITRE. 2006a. CWE-121: Stack-based Buffer Overflow.

MITRE. 2006b. CWE-122: Heap-based Buffer Overflow.

MITRE. 2006c. CWE-401: Missing Release of Memory after Effective Lifetime.

MITRE. 2006d. CWE-416: Use After Free.

Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. 2025. C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques. *arXiv preprint arXiv:2501.14257*.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the effectiveness of large language models in code translation. *CoRR*.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2785–2799.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks.

Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A Seshia, and Koushik Sen. 2024. Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis. *arXiv preprint arXiv:2412.14234*.

Momoko Shiraishi and Takahiro Shinagawa. 2024. Context-aware code segmentation for c-to-rust translation using large language models. *arXiv preprint arXiv:2409.10506*.

Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2023. Code translation with compiler representations. *ICLR*.

The Rust Project Contributors. Clippy.

TIOBE. 2025. TIOBE Index for January 2025.

Qilong Wu, Taoran Li, Tianyang Zhou, and Varun Chandrasekaran. 2025. Sok: Understanding (new) security issues across ai4code use cases. *arXiv preprint arXiv:2512.18456*.

Xiph. 2025. Xiph/ogg: Reference implementation of the Ogg media container.

Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. Vert: Verified equivalent rust transpilation with large language models as few-shot learners. *arXiv preprint arXiv:2404.18852*.

Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership guided c to rust translation. In *International Conference on Computer Aided Verification*, pages 459–482. Springer.

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36.

A Differences Between C and Rust

A.1 Code Snippets

Here is a code example to demonstrate the differences between C and Rust. The example shows a simple C program and its equivalent Rust program. The `create_sequence` function takes an integer `n` as input and returns an array with a sequence of integers. In C, the function needs to allocate memory for the array using `malloc` and will return the pointer to the allocated memory as an array. If the size is invalid, or the allocation fails, the function will return `NULL`. The caller of the function is responsible for freeing the memory using `free` when it is done with the array to prevent memory leaks.

C Code:

```c
int* create_sequence(int n) {
    if (n <= 0) {
        return NULL;
    }
    int* arr = malloc(n * sizeof(int));
    if (!arr) {
        return NULL;
    }
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }
    return arr;
}

int* sequence = create_sequence(5);
if (sequence == NULL) {
    ...
}
...
free(sequence); // Need to free the memory when done
```

Rust Code:

```rust
fn create_sequence(n: i32) -> Option<Vec<i32>> {
    if n <= 0 {
        return None;
    }
    let mut arr = Vec::with_capacity(n as usize);
    for i in 0..n {
        arr.push(i);
    }
    Some(arr)
}
match create_sequence(5) {
    Some(sequence) => {
        ... // Does not need to free the memory
    }
    None => {
        ...
    }
}
```

Figure 5: Example of a simple C program and its equivalent Rust program, both hand-written for illustration.

A.2 Tabular Summary

Here, we present a non-exhaustive list of differences between C and Rust in Table 5, highlighting the key features that make translating code from C to Rust challenging. While the list is not comprehensive, it provides insights into the fundamental distinctions between the two languages, which can help developers understand the challenges of migrating C code to Rust.

B Preprocessing and Task Division

B.1 Preprocessing of C Files

To support real-world C projects, SACTOR parses the compile commands generated by the `make` tool, extracting relevant flags for preprocessing, parsing, compilation, linking, and third-party tools' use.

C source files usually contain preprocessing directives, such as `#include`, `#define`, `#ifdef`, `#endif`, *etc.*, which we need to resolve before parsing C files. For `#include`, we copy and expand non-system headers recursively while keeping `#include` of system headers intact, because included non-system headers contain project-specific definitions such as structs and enums that the LLM has not known while system headers' contents are known to the LLM and expanding them would unnecessarily introduce too much noise. For other directives, we pass relevant C project compile flags to the C preprocessor from GCC to resolve them.

B.2 Algorithm for Task Division

The task division algorithm is used to determine the order in which the items should be translated. The algorithm is shown in Algorithm 1.

In the algorithm, L_i is the list of items to be translated, and $dep(a)$ is a function that returns the dependencies of item a. The algorithm returns a list L_{sorted} that contains the items in the order in which they should be translated. It begins by collecting all items (e.g., functions, structs) to be translated and their respective dependencies (in both functions and data types). Items with no unresolved dependencies are pushed into the translation order list first, and other items will remove them from their dependencies list. This process continues until all items are pushed into the list. If no item is added to L_{sorted} during an iteration (i.e., every remaining item has an unprocessed dependency), the graph contains a cycle. SACTOR currently treats this as unsupported and terminates with an explicit error.

C Feedback and Repair Procedure

Algorithm 2 describes the translation and repair loop applied to each function or data structures. Both the unidiomatic and idiomatic stages use the same procedure; they differ only in the prompt context supplied to the LLM (the idiomatic stage additionally provides Crown pointer analysis hints and the interface specification) and the verification harness (the idiomatic stage uses a SPEC-driven harness to bridge ABI differences).

FEATURE	C	RUST
MEMORY MANAGEMENT	Manual (through `malloc/free`)	Automatic (through ownership and borrowing)
POINTERS	Raw pointers like `*p`	Safe references like `&p/&mut p`, `Box` and `Rc`
LIFETIME MANAGEMENT	Manual freeing of memory	Lifetime annotations and borrow checker
ERROR HANDLING	Error codes and manual checks	Explicit handling with `Result` and `Option` types
NULL SAFETY	Null pointers allowed (e.g., `NULL`)	No null pointers; uses `Option` for nullable values
CONCURRENCY	No built-in protections for data races	Enforces safe concurrency with ownership rules
TYPE CONVERSION	Implicit conversions allowed and common	Strongly typed; no implicit conversions
STANDARD LIBRARY	C stand library with direct system calls	Rust standard library with utilities for strings, collections, and I/O
LANGUAGE FEATURES	Procedure-oriented with minimal abstractions	Modern features like pattern matching, generics, and traits

Table 5: Key Differences Between C and Rust

Algorithm 1 Translation Task Order Determination

Require: L_i: List of items to be translated
Require: $dep(a)$: Function to get dependencies of item a
Ensure: L_{sorted}: List of groups resolving dependencies

$L_{sorted} \leftarrow \emptyset$ ▷ Empty list
$L_{processed} \leftarrow \emptyset$
while $|L_{sorted}| < |L_i|$ **do**
  $progress \leftarrow$ **false**
  for $a \in L_i$ **do**
    if $a \notin L_{processed}$ and $dep(a) \subseteq L_{processed}$ **then**
      $L_{sorted} \leftarrow L_{sorted} + a$ ▷ Add to sorted list
      $L_{processed} \leftarrow L_{processed} \cup \{a\}$
      $progress \leftarrow$ **true**
    end if
  end for
  if $\neg progress$ **then**
    abort ▷ Unsupported: cyclic dependency detected
  end if
end while
return L_{sorted}

Subroutine details.

- COMPILE invokes the Rust compiler on the translated fragment and returns the compiler diagnostics (error spans and messages) on failure.
- FFI_LINK compiles the Rust translation as a shared library and links it into the original C build so that calls to the target function are dispatched to the Rust implementation.
- RUN_E2E_TESTS executes the project's existing end-to-end tests on the mixed C/Rust binary. The translated function is annotated with the `trace_fn` procedural macro, which automatically logs each invocation's arguments and return value. If the test times out, the attempt is treated as a failure.
- EXTRACT_TRACES parses the test output for execution traces (delimited by `Entering function` / `Exiting function` markers). When traces are present, they provide the LLM with concrete input/output evidence; otherwise, the raw test stderr is used as feedback.
- LLM_REPAIR re-queries the LLM with the failed translation and the structured feedback, asking it to analyze the error and produce a corrected version.

D Equivalence Testing Details in Prior Literature

D.1 Symbolic Execution-Based Equivalence

Symbolic execution explores all potential execution paths of a program by using symbolic inputs to generate constraints (King, 1976; Baldoni et al.,

Algorithm 2 Fragment Translation with Feedback

Require: f: source fragment (function or data structure) to translate
Require: ctx: translation context (dependencies, type hints, Crown output for idiomatic)
Require: K: maximum number of attempts
Ensure: Verified Rust translation of f, or FAIL

$translation \leftarrow$ LLM_TRANSLATE(f, ctx) ▷ Initial LLM translation
for $attempt = 1$ **to** K **do**
  $(compileOk,\ errors) \leftarrow$ COMPILE$(translation)$
  if $\neg\ compileOk$ **then**
    $translation \leftarrow$ LLM_REPAIR$(f,\ ctx,\ translation,\ errors)$
    continue
  end if
  $binary \leftarrow$ FFI_LINK$(translation,$ original C build$)$ ▷ Link Rust into C via FFI
  $(testOk,\ output) \leftarrow$ RUN_E2E_TESTS$(binary)$ ▷ `trace_fn` macro instruments the target
  if $testOk$ **then**
    return $translation$ ▷ Verified
  end if
  $feedback \leftarrow$ EXTRACT_TRACES$(output)$ ▷ Parse execution traces from test output
  $translation \leftarrow$ LLM_REPAIR$(f,\ ctx,\ translation,\ feedback)$
end for
return FAIL

2018; Coward, 1988). While theoretically powerful, this method is impractical for verifying C-to-Rust equivalence due to differences in language features. For instance, Rust's RAII (Resource Acquisition Is Initialization) pattern automatically inserts destructors for memory management, while C relies on explicit `malloc` and `free` calls. These differences cause mismatches in compiled code, making it difficult for symbolic execution engines to prove equivalence. Additionally, Rust's compiler adds safety checks (e.g., array boundary checks), which further complicate equivalence verification.

D.2 Fuzz Testing-Based Equivalence

Fuzz testing generates random or mutated inputs to test whether program outputs match expected results (Zhu et al., 2022; Miller et al., 1990; Liang et al., 2018). While more practical than symbolic execution, fuzz testing faces challenges in constructing meaningful inputs for real-world programs. For example, testing a URL parsing function requires generating valid URLs with specific formats, which is non-trivial. For large C programs, this difficulty scales, making it infeasible to produce high-quality test cases for every translated Rust function.

E An Example of the Test Harness

Here, we provide an example of the test harness used to verify the correctness of the translated code in Figure 6, which is used to verify the idiomatic Rust code. In this example, the `concat_str_idiomatic` function is the idiomatic translation we are testing, while the `concat_str_c` function is the test harness function that can be linked back to the original C code. where a string and an integer are passed as input, and an owned string is returned. Input strings are converted from C's `char*` to Rust's `&str`, and output strings are converted from Rust's `String` back to C's `char*`.

```rust
fn concat_str_idiomatic(orig: &str, num: i32) -> String {
    format!("{}{}", orig, num)
}

fn concat_str(orig: *const c_char, num: c_int) -> *const c_char {
    // convert input
    let orig_str = CStr::from_ptr(orig)
        .to_str()
        .expect("Invalid UTF-8 string");
    // call target function
    let out = concat_str_idiomatic(orig_str, num as i32);
    // convert output
    let out_str = CString::new(out).unwrap();
    // `into_raw` transfers ownership to the caller
    out_str.into_raw()
}
```

Figure 6: Test harness used for verifying `concat_str` translation

F An Example of SACTOR Translation Process

To demonstrate the translation process of SACTOR, we present a straightforward example of

translating a C function to Rust. The C program includes an `atoi` function that converts a string to an integer, and a `main` function that parses command-line arguments and calls the `atoi` function. The C code is shown in Figure 7a.

We assume that there are numerous end-to-end tests for the C code, allowing SACTOR to use them for verifying the correctness of the translated Rust code.

First, the divider will divide the C code into two parts: the `atoi` function and the `main` function, and determine the translation order is first `atoi` and then `main`, as `atoi` is the dependency of `main` and the `atoi` function is a pure function.

Next, SACTOR proceeds with the unidiomatic translation, converting both functions into unidiomatic Rust code. This generated code will keep the semantics of the original C code while using Rust syntax. Once the translation is complete, the unidiomatic verifier executes the end-to-end tests to ensure the correctness of the translated function. If the verifier passes all tests, SACTOR considers the unidiomatic translation accurate and progresses to the next function. If any test fails, SACTOR will retry the translation process using the feedback information collected from the verifier, as described in § 4.3. After translating all sections of the C code, SACTOR will combine the unidiomatic Rust code segments to form the final unidiomatic Rust code. The unidiomatic Rust code is shown in Figure 7b.

Then, the SACTOR will start the idiomatic translation process and translate the unidiomatic Rust code into idiomatic Rust code. The idiomatic translator requests the LLM to adapt the C semantics into idiomatic Rust, eliminating any unsafe and non-idiomatic constructs, as detailed in § 4.2. Based on the same order, the SACTOR will translate two functions accordingly, and using the idiomatic verifier to verify and provide the feedback to the LLM if the verification fails. After all parts of the Rust code are translated into idiomatic Rust, verified, and combined, the SACTOR will produces the final idiomatic Rust code. The idiomatic Rust code is shown in Figure 7c, representing the final output of SACTOR.

```c
#include <stdio.h>

int atoi(char *str) {
    int result = 0;
    int sign = 1;

    while (*str == ' ' || *str == '\t' || *str == '\n' ||
           *str == '\r' || *str == '\v' || *str == '\f') {
        str++;
    }

    if (*str == '+' || *str == '-') {
        if (*str == '-') {
            sign = -1;
        }
        str++;
    }

    while (*str >= '0' && *str <= '9') {
        result = result * 10 + (*str - '0');
        str++;
    }

    return sign * result;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }

    int value = atoi(argv[1]);
    printf("Parsed integer: %d\n", value);
    return 0;
}
```

(a) C implementation of `atoi`

```rust
use libc::c_char;
use std::env;
use std::ffi::CString;
use std::process;
pub unsafe fn atoi(str: *const c_char) -> i32 {
    let mut result: i32 = 0;
    let mut sign: i32 = 1;
    let mut ptr = str;
    while *ptr == ' ' as c_char
        || *ptr == '\t' as c_char
        || *ptr == '\n' as c_char
        || *ptr == '\r' as c_char
        || *ptr == '\x0B' as c_char
        || *ptr == '\x0C' as c_char
    {
        ptr = ptr.add(1);
    }
    if *ptr == '+' as c_char || *ptr == '-' as c_char {
        if *ptr == '-' as c_char {
            sign = -1;
        }
        ptr = ptr.add(1);
    }
    while *ptr >= '0' as c_char && *ptr <= '9' as c_char {
        let digit = (*ptr - '0' as c_char) as i32;
        if let Some(new_result) = result.checked_mul(10).and_then(
            |r| r.checked_add(digit),
        ) {
            result = new_result;
        } else {
            return if sign == 1 { i32::MAX } else { i32::MIN };
        }
        ptr = ptr.add(1);
    }
    sign * result
}
pub fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} <number>", args[0]);
        process::exit(1);
    }
    let c_str = match CString::new(args[1].as_str()) {
        Ok(cstring) => cstring,
        Err(_) => {
            eprintln!("Failed to create CString from input");
            process::exit(1);
        }
    };
    let value = unsafe { atoi(c_str.as_ptr() as *const c_char) };
    println!("Parsed integer: {}", value);
}
```

(b) Unidiomatic Rust translation from C

```rust
use std::env;
use std::process;
pub fn atoi(input: &str) -> i32 {
    let mut result: i32 = 0;
    let mut sign: i32 = 1;
    let mut chars = input.chars().peekable();
    while let Some(&c) = chars.peek() {
        if c.is_whitespace() {
            chars.next();
        } else {
            break;
        }
    }
    if let Some(&c) = chars.peek() {
        if c == '+' || c == '-' {
            if c == '-' {
                sign = -1;
            }
            chars.next();
        }
    }
    for c in chars {
        if let Some(digit) = c.to_digit(10) {
            if let Some(new_result) = result
                .checked_mul(10)
                .and_then(|r| r.checked_add(digit as i32))
            {
                result = new_result;
            } else {
                return if sign == 1 { i32::MAX } else { i32::MIN };
            }
        } else {
            break;
        }
    }
    sign * result
}
pub fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} <number>", args[0]);
        process::exit(1);
    }
    let input = &args[1];
    let value = atoi(input);
    println!("Parsed integer: {}", value);
}
```

(c) Idiomatic Rust translation from unidiomatic Rust

Figure 7: SACTOR translation process for `atoi` program

G Dataset Details

G.1 TransCoder-IR Dataset (Szafraniec et al., 2023)

The TransCoder-IR dataset is used to evaluate the TransCoder-IR model and consists of solutions

Dataset	Size	Preprocessing	E2E Tests	Coverage (Line/Func)
TransCoder-IR (Szafraniec et al., 2023)	100	Removed buggy programs (compilation/memory errors) and entries with existing Rust	Present	97.97% / 99.5%
Project CodeNet (Puri et al., 2021)	100	Filtered for external-input programs (`argc`/`argv`); auto-generated tests	Generated	94.37% / 100%
CRust-Bench (Khatry et al., 2025)	50	Excluded unsupported patterns; combine code of each sample to a single `lib.c`	Present	76.18% / 80.98%
libogg (Xiph, 2025)	1	None. Each component of the library is contained within a single C file.	Present	83.3% / 75.3%

Table 6: Summary of datasets and real-world code-bases used for evaluation; coverage audited with `gcov` on the tests exercised in our pipeline.

to coding challenges in various programming languages. For evaluation, we focus on the 698 C programs available in this dataset. First, we filter out programs that already have corresponding Rust code. Several C programs in the dataset contain bugs, which are removed by checking their ability to compile. We then use *valgrind* to identify and discard programs with memory errors during the end-to-end tests. Finally, we select 100 programs with the most lines of code for our experiments.

G.2 Project CodeNet (Puri et al., 2021)

Project CodeNet is a large-scale dataset for code understanding and translation, containing 14 million code samples in over 50 programming languages collected from online judge websites. From this dataset, which includes more than 750,000 C programs, we target only those that accept external input. Specifically, we filter programs using `argc` and `argv`, which process input from the command line. As the end-to-end tests are not available for this dataset, we develop the SACTOR test generator to automatically generate end-to-end tests for these programs based on the source code. For evaluation, we select 200 programs and refine the dataset to include 100 programs that successfully generate end-to-end tests.

G.3 CRust-Bench (Khatry et al., 2025)

CRust-Bench is a repository-level benchmark for C-to-safe-Rust transpilation. It collects 100 real-world C repositories (the CBench suite) and pairs each with a manually written, safe Rust interface and a set of tests that assert functional correctness. By evaluating full repositories rather than isolated functions, CRust-Bench surfaces challenges common in practice, such as complex, pointer-rich APIs. In our evaluation, we use a 50-sample subset in CRust-Bench, which exclude entries that are out of scope for our pipeline (e.g., circular type or function dependencies and compiler-specific intrinsics that do not map cleanly). For each selected sample, we reuse the upstream end-to-end tests and relink them so that calls exercise our translated code; build environments and link flags follow the sample's configuration.

G.4 libogg (Xiph, 2025)

`libogg` is the reference implementation of the Ogg multimedia container. Ogg is a stream-oriented format that frames, timestamps, and multiplexes compressed media bitstreams (e.g., audio/video) into a robust, seekable stream. The `libogg` distribution contains only the Ogg container library (codecs such as Vorbis or Theora are hosted separately). In our case study, the codebase comprises roughly 2,041 lines of code (excluding tests), six struct definitions, three global variables, and 77 exported functions. We use the project's upstream tests and build scripts. This single-project evaluation complements the CRust-Bench subset by focusing on non-trivial structs, buffers, and pointer manipulation in a real-world C library.

H Failure Analysis in Evaluating SACTOR

Here, we analyze the failure cases of SACTOR in translating C code to Rust that we conducted in Section 6.1. as cases where SACTOR fails offer valuable insights into areas that require refinement. For each failure case in the two datasets, we conduct an analysis to determine the primary cause of translation failure. This process involves leveraging DeepSeek-R1 to identify potential reasons (prompts available in Appendix N.5), followed by manual verification to ensure correctness. We only focus on the translation process from C to unidiomatic Rust because: (1) it is the most challenging step, and (2) it can better reflect the model's ability to fit the syntactic and semantic differences between the two languages. Table 7 summarize the categories of failure reasons, and Figure 8a and 8b illustrate failure reasons (FRs) across models.

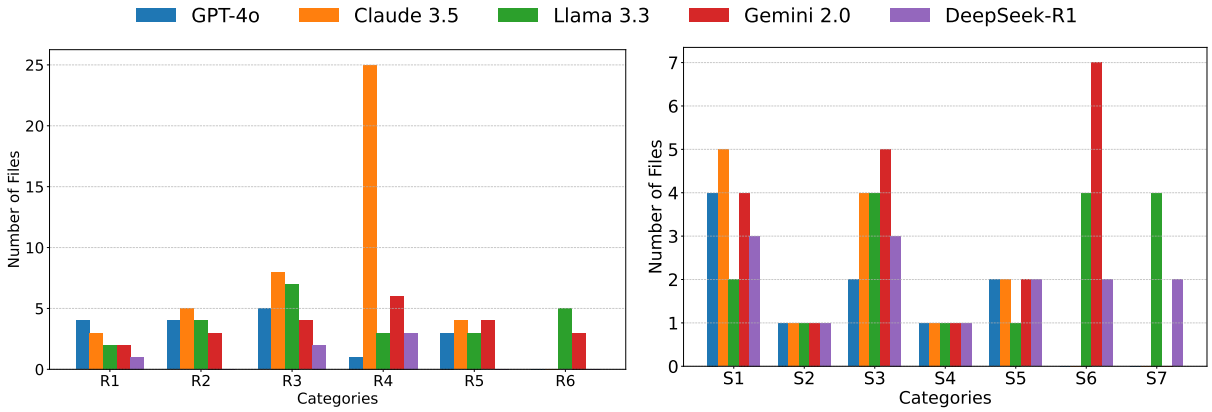
(a) TransCoder-IR

CATEGORY	DESCRIPTION
R1	Memory safety violations in array operations due to improper bounds checking
R2	Mismatched data type translations
R3	Incorrect array sizing and memory layout translations
R4	Incorrect string representation conversion between C and Rust
R5	Failure to handle C's undefined behavior with Rust's safety mechanisms
R6	Use of C-specific functions in Rust without proper Rust wrappers

(b) Project CodeNet

CATEGORY	DESCRIPTION
S1	Improper translation of command-line argument handling or attempt to fix wrong handling
S2	Function naming mismatches between C and Rust
S3	Format string directive mistranslation causing output inconsistencies
S4	Original code contains random number generation
S5	SACTOR unable to translate mutable global state variables
S6	Mismatched data type translations
S7	Incorrect control flow or loop boundary condition translations

Table 7: Failure reason categories for translating TransCoder-IR and Project CodeNet datasets.



(a) TransCoder-IR (b) Project CodeNet

Figure 8: Failure reasons across different LLM models for both datasets.

(1) TransCoder-IR (Table 7a, Figure 8a): Based on the analysis, we observe that different models exhibit varying failure reasons. Claude 3.5 shows a particularly high incidence of string representation conversion errors (R4), with 25 out of 45 total failures in the unidiomatic translation step. In contrast, GPT-4o has only 1 out of 17 failures in this category. Llama 3.3 demonstrates consistent challenges with both R3 (incorrect array sizing and memory layout translations) and R6 (using C-specific functions without proper Rust wrappers), with 10 files for each category. GPT-4o shows a more balanced distribution of errors, with its highest count in R3. All models except GPT-4o struggle with string handling (R4) to varying degrees, suggesting this is one of the most challenging aspects of the translation process. For R6 (use of C-specific functions in Rust), which primarily is a compilation failure, only Llama 3.3 and Gemini 2.0 consistently fail to resolve the issue in some cases, while all other models can successfully handle the compilation errors through feedback and avoid failure in this category. DeepSeek-R1 has the fewest overall errors across categories, with failures only in R1 (1 file), R3 (2 files), and R4 (3 files), while completely avoiding errors in R2, R5, and R6.

(2) Project CodeNet (Table 7b, Figure 8b): Similar to the TransCoder-IR dataset, we also observe that different models in Project CodeNet demonstrate varying failure reasons. C-to-Rust code translation challenges in the CodeNet dataset. Most notably, S6 (mismatched data type translations) presents a significant barrier for Llama 3.3 and Gemini 2.0 (7 files each), while GPT-4o and Claude 3.5 completely avoid this issue. Input argument handling (S1) and format string mistranslations (S3) emerge as common challenges across all models in CodeNet, suggesting fundamental difficulties in translating these language features regardless of model architecture. Only Llama 3.3 and DeepSeek-R1 encounter control flow translation failures (S7), with 2 files each. S4 (random number generation)

and S5 (mutable global state variables) are unable to be translated by SACTOR because the current SACTOR implementation does not support these features.

Compared to the results in TransCoder-IR, string representation conversion (R4 in TransCoder-IR, S3 in CodeNet) remains a consistent challenge across both datasets for all models, though the issue is significantly more severe in TransCoder-IR, particularly for Claude 3.5 (24 files). This also suggests that reasoning models like DeepSeek-R1 are better at handling complex code logic and string/array manipulation, as they exhibit fewer failures in these areas, demonstrating the potential of reasoning models to address complex translation tasks.

I SACTOR Cost Analysis

LLM	Dataset	Tokens	Avg. Queries
Claude 3.5	TransCoder-IR	4595.33	5.15
	CodeNet	3080.28	3.15
Gemini 2.0	TransCoder-IR	3343.12	4.24
	CodeNet	2209.38	2.39
Llama 3.3	TransCoder-IR	4622.80	5.39
	CodeNet	4456.84	3.80
GPT-4o	TransCoder-IR	2651.21	4.24
	CodeNet	2565.36	2.95
DeepSeek-R1	TransCoder-IR	17895.52	4.77
	CodeNet	13592.61	3.11

Table 8: Average Cost Comparison of Different LLMs Across Two Datasets. The color intensity represents the relative cost of each metric for each dataset.

Here, we conduct a cost analysis of SACTOR for experiments in § 6.1 to evaluate the efficiency of different LLMs in generating idiomatic Rust code. To evaluate the cost of our approach, we measure (1) *Total LLM Queries* as the number of total LLM queries made during translation and verification for a single test case in each dataset, and (2) *Total Token Count* as the total number of tokens processed by the LLM for a single test case in each dataset. To ensure a fair comparison across models, we use the same tokenizer (`tiktoken`) and encoding (`o200k_base`).

In order to better understand costs, we only analyze programs that successfully generate idiomatic Rust code, excluding failed attempts (as they always reach the maximum retry limit and do not contribute meaningfully to the cost analysis). We evaluate the combined cost of both translation phases to assess overall efficiency. Table 8 compares the average cost of different LLMs across two datasets, measured in token usage and query count per successful idiomatic Rust translation as mentioned in § 5.2.

Results: Gemini 2.0 and GPT-4o are the most efficient models, requiring the fewest tokens and queries. GPT-4o maintains a low token cost (2651.21 on TransCoder-IR, 2565.36 on CodeNet) with 4.24 and 2.95 average queries, respectively. Gemini 2.0 is similarly efficient, especially on CodeNet, with the lowest token usage (2209.38) and requiring only 2.39 queries on average. Claude 3.5, despite its strong performance on CodeNet, incurs higher costs on TransCoder-IR (4595.33 tokens, 5.15 queries), likely due to additional translation steps. Llama 3.3 is the least efficient in non-thinking model (GPT-4o, Claude 3.5, Gemini 2.0), consuming the most tokens (4622.80 and 4456.84, respectively) and requiring the highest number of queries (5.39 and 3.80, respectively), indicating significant resource demands.

As a reasoning model, DeepSeek-R1 consumes significantly more tokens (17,895.52 vs. 13,592.61) than non-reasoning models–5-7 times higher than GPT-4o–despite having a similar average query count (4.77 vs. 3.11) for generating idiomatic Rust code. This high token usage comes from the "reasoning process" required before code generation.

J Ablation Study on SACTOR Designs

This appendix reports additional ablations that evaluate key design choices in SACTOR. All experiments in this section use GPT-4o with the same configuration as Table 1.

J.1 Feedback Mechanism

To evaluate the effectiveness of the feedback mechanism proposed in § 4.3, we conduct an ablation study by removing the mechanism and comparing the model's performance with and without it. We consider two experimental groups: (1) with the feedback mechanism enabled, and (2) without the feedback mechanism. In the latter setting, if any part of the translation fails, the system simply restarts the translation attempt using the original prompt, without providing any feedback from the failure.

We use the same dataset and evaluation metrics described in § 5, and focus our evaluation on only two models: GPT-4o and Llama 3.3 70B. We choose these models because GPT-4o demonstrated one of the highest performance and Llama 3.3 70B

the lowest in our earlier experiments. By comparing the success rates between the two groups, we assess whether the feedback mechanism improves translation performance across models of different capabilities.

The results are shown in Figure 9.

(1) TransCoder-IR (Figure 9a): Incorporating the feedback mechanism increased the number of successful translations for Llama 3.3 70B from 57 to 76 in the unidiomatic setting and from 46 to 64 in the idiomatic setting. In contrast, GPT-4o performed slightly worse with feedback, decreasing from 87 to 84 (unidiomatic) and from 83 to 80 (idiomatic).

(2) Project CodeNet (Figure 9b): A similar trend is observed where Llama 3.3 70B improved from 62 to 83 (unidiomatic) and from 59 to 76 (idiomatic), corresponding to gains of 21 and 17 percentage points, respectively. GPT-4o, however, showed only marginal improvements: from 82 to 84 in the unidiomatic setting and from 77 to 79 in the idiomatic setting.

These results suggest that the feedback mechanism is particularly effective for lower-capability models like Llama 3.3, substantially improving their translation success rates. In contrast, higher-capability models such as GPT-4o already perform near optimal with simple random sampling, leaving little space for improvement. This indicates that the feedback mechanism is more beneficial for models with lower capabilities, as they can leverage the feedback to enhance their overall performance.

J.2 Plain LLM Translation vs. SACTOR

We compare SACTOR against a trivial baseline where GPT-4o directly translates each CRust-Bench sample from C to Rust in a single step. We reuse the same end-to-end (E2E) test harness as SACTOR, and give the trivial baseline more budget: up to 10 repair attempts with compiler/test feedback (vs. 6 attempts in SACTOR). We study two prompts: (i) a minimal one ("translate the following C code to Rust"); and (ii) an interface-preserving one that explicitly asks the model to preserve pointer arithmetic, memory layout, and integer type semantics (thereby encouraging `unsafe`). We report *function success* as the fraction of functions whose Rust translation passes all tests, and *sample success* as the fraction of samples where all translated functions pass.

Results on `CRust-Bench`. Even with 10 attempts and an "encourage `unsafe`" prompt, the trivial baseline reaches only 21.43% function success and 40.00% sample success. Its sample-level performance exceeds SACTOR's idiomatic stage (40.00% vs. 25.00%) because preserving C-style pointer logic in `unsafe` Rust is substantially easier than performing an idiomatic rewrite. However, SACTOR achieves much higher function-level correctness and produces significantly more idiomatic code (e.g., 0.28 vs. 1.90 average Clippy alerts per function).

Results on `libogg`. Under the same E2E tests and attempt budget as SACTOR, both trivial prompts fail to produce any test-passing translations, whereas SACTOR achieves 100% unidiomatic and 53% idiomatic success with GPT-4o (Table 4). This indicates that plain one-shot translation collapses on pointer-heavy libraries, while SACTOR remains effective.

J.3 Effect of Crown in the Idiomatic Stage

We ablate Crown's contribution to idiomatic translation (§ 4.2) on `libogg`, using the same setup as § 6.3 and keeping all other components unchanged. Table 10 reports idiomatic function success with and without Crown.

Results and Representative failure patterns. Turning off Crown reduces idiomatic success from 41 to 34 functions. The failures are structured. Two representative patterns are:

```
// Without Crown (shape lost):
pub struct OggPackBuffer { pub ptr:
    usize }

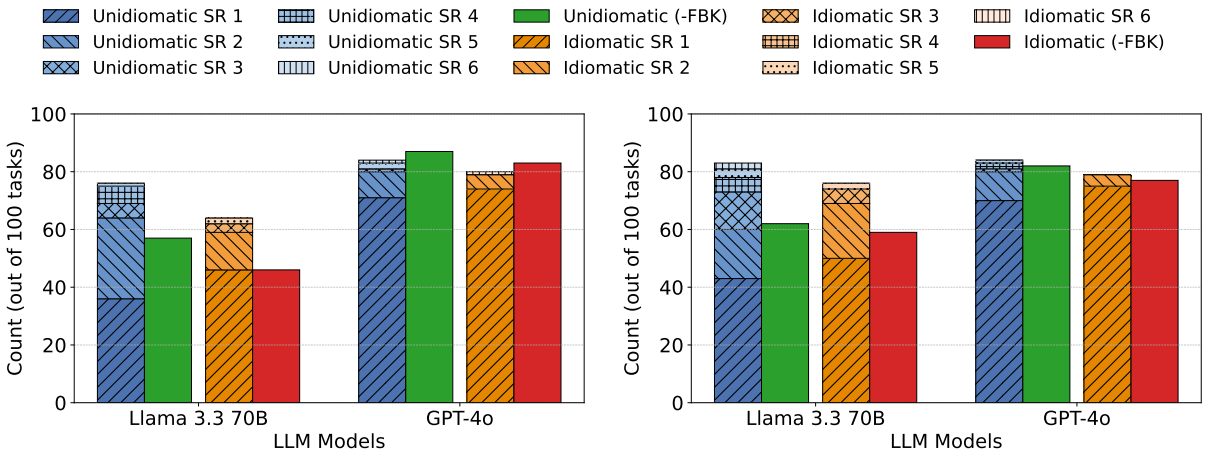
// With Crown (shape preserved):
pub struct OggPackBuffer { pub ptr: Vec<
    u8> }

// Without Crown (ownership
    misclassified as owned):
pub struct OggIovec { pub iov_base: Vec<
    u8> }

// With Crown (ownership made explicit):
pub struct OggIovec<'a> { pub iov_base:
    &'a [u8] }
```

Once a buffer pointer is collapsed into a scalar index, the harness cannot reconstruct a valid C-facing view of the struct, so pointer arithmetic and buffer access fail together. Similarly, if a non-owning pointer (e.g., `unsigned char *iov_base`) is misclassified as owned storage (`Vec<u8>`), Rust ends up "owning" memory that C actually controls, making safe round-tripping infeasible without inventing allocation/free rules that do not exist.

Interpretation. These failures do not indicate model weakness but an *information-theoretic lim-*



(a) TransCoder-IR With/Without Feedback

(b) CodeNet With/Without Feedback

Figure 9: Ablation study on the feedback mechanism. The success rates of the models with and without the feedback (marked as *-FBK*) mechanism are shown for both TransCoder-IR and CodeNet datasets.

METHOD	MAX ATT.	FUNCTION SUCCESS	SAMPLE SUCCESS	AVG. CLIPPY / FUNC.
SACTOR unidiomatic	6	**788/966 (81.57%)**	32/50 (64.00%)	2.96
SACTOR idiomatic†	6	**249/580 (42.93%)**	8/32 (25.00%)	0.28
Trivial (1-step)	10	77/966 (7.97%)	12/50 (24.00%)	1.60
Trivial (1-step, encourage `unsafe`)	10	207/966 (21.43%)	20/50 (40.00%)	1.90

Table 9: Plain LLM translation vs. SACTOR on CRust-Bench (GPT-4o). The trivial baselines directly translate each sample in one step with up to 10 repair attempts. † The idiomatic stage is evaluated only on samples whose unidiomatic stage fully translated all functions.

CONFIGURATION	# IDIOM. SF	IDIOM. SR	REL. DROP
SACTOR	41	53%	–
SACTOR w/o Crown	34	44%	**17%**

Table 10: Ablating Crown on `libogg` (GPT-4o). SF stands for successful functions, and SR stands for success rate. Removing Crown reduces idiomatic success by 17%.

itation: local C syntax does not encode pointer fatness or ownership. For a declaration such as `char *iov_base`, both `Vec<u8>` and `&mut u8` are locally plausible. Even an *idealized oracle model* cannot uniquely infer the correct Rust type without global information about ownership and fatness. Crown supplies these semantics via whole-program static analysis; removing it makes idiomatic translation of pointer-heavy code underdetermined and explains the observed drop.

J.4 Prompting about `unsafe` in Stage 1

We ablate the stage-1 (unidiomatic translation) prompt line that says "the model may use `unsafe` if needed." All experiments in this subsection are conducted on `libogg`, using exactly the same setup as in § 6.3.

J.4.1 Removing "may use `unsafe` if needed"

We compare the original stage-1 prompt with a variant that deletes this line, keeping everything else unchanged.

Two observations follow. *(1) Overall unsafety hardly changes:* the unsafe fraction drops only from 99.99% to 98.55%. *(2) The safety profile becomes worse:* `clippy::not_unsafe_ptr_arg_deref` jumps from 1 to 146. That is, the model keeps APIs safe-looking but dereferences raw pointer arguments inside function bodies, pushing unsafety from explicit `unsafe fn` signatures into hidden dereferences inside safe-looking public functions.

J.4.2 Replacing With "AVOID using `unsafe`"

We replace "may use `unsafe` if needed" with a stronger directive: "AVOID using `unsafe` whenever possible".

Under "AVOID `unsafe`", the model often attempts premature "safe Rust" rewrites of pointer-heavy C code (changing buffer layouts, index arithmetic, and integer types), which increases logic and type

PROMPT VARIANT	UNIDI. SR	CLIPPY TOTAL	missing_safety_doc	not_unsafe_ptr_arg_deref	UNSAFE FRACTION
Baseline stage 1 (may use unsafe)	100%	108	76	1	8704/8705 (99.99%)
Remove "may use unsafe"	100%	**224**	37	**146**	8100/8219 (98.55%)

Table 11: Removing explicit permission to use unsafe in stage 1 on libogg (GPT-4o).

PROMPT VARIANT	PASSED/TOTAL	SR	REL. DROP
Baseline stage 1	77/77	100%	–
Replace with "AVOID unsafe"	66/77	85%	**15%**

Table 12: Discouraging unsafe in stage 1 harms unidiomatic success on libogg (GPT-4o).

errors and breaks translations. Together, these two prompt variants show that discouraging unsafe in stage 1 harms correctness and produces a worse safety profile, supporting our design choice: allow necessary unsafe in the syntactic first stage, then systematically remove it in the idiomatic refinement stage.

K SACTOR Performance with Different Temperatures

In § 6, all the experiments are conducted with the temperature set to default values, as explained on Section 5.3. To investigate how temperature affects the performance of SACTOR, we conduct additional experiments with different temperature settings (0.0, 0.5, 1.0) for GPT-4o on both TransCoder-IR and Project CodeNet datasets, as shown in Figure 10. Through some preliminary experiments and discussions on OpenAI's community forum [5], we find that setting the temperature more than 1 will likely to generate more random and less relevant outputs, which is not suitable for our task.

(1) TransCoder-IR (Figure 10a): Setting the decoder to a deterministic temperature of $t = 0$ resulted in 83 successful translations (83%), while both $t = 0.5$ and $t = 1.0$ yielded 80 successes (80%) each. This represents a slightly improvement with 3 additional correct predictions under the deterministic setting.

(2) Project CodeNet (Figure 10b): Temperature does not have a significant impact: the model produced 79, 81, and 79 successful outputs at $t = 0$, $t = 0.5$, and $t = 1.0$ respectively (79–81%), which does not indicate any outstanding trend in performance across the temperature settings.

The results on both datasets suggests that lowering temperature to zero can offer a slight boost in reliability some of the cases, but it does not significantly affect the overall performance of SACTOR.

[5] https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api/172683

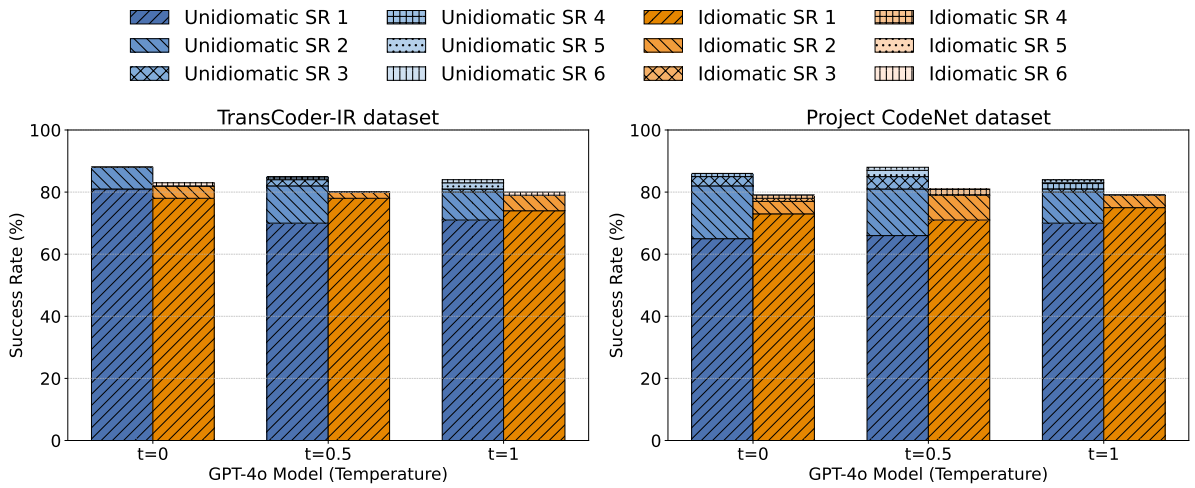
L Spec-driven Harness Rules

This section describes the details of the SPEC-driven harness generator used in the idiomatic verification phase. Table 13 summarizes the SPEC patterns our rule-based generator currently supports.

Harness construction details. The generator consumes a per-item SPEC (JSON) produced alongside idiomatic code and synthesizes: (i) a C-compatible shim that matches the original ABI, and (ii) idiomatic adapters that convert to/from Rust types. Pointer shapes (scalar, cstring, slice, ref) determine how memory is borrowed or owned; length sources come from sibling fields or constants; nullability and ownership hints select Option<> or strict checks. Return values are mapped back to U form, writing lengths when needed. This bridging resolves the ABI mismatch introduced by idiomatic function signatures.

Struct mappings and self-check. For structs, the SPEC defines bidirectional converters between unidiomatic and idiomatic layouts. We validate adapter consistency with a minimal roundtrip: Unidiomatic → Idiomatic(1) → Unidiomatic → Idiomatic(2). The self-check compares Idiomatic(1) and Idiomatic(2) field-by-field according to compare hints: by_value requires exact equality on scalar fields; by_slice compares slice contents using the SPEC-recorded length source; skip omits fields that are aliasing views or externally owned to avoid false positives. Seed unidiomatic values are synthesized by an LLM guided by the SPEC so that nullability, ownership, and length sources are populated consistently.

Fallback and verification loop. When a SPEC uses patterns not yet implemented (e.g., pointer kinds outside cstring/slice/ref; nontrivial len_from expressions; string args whose spec.kind ≠ cstring), the generator emits a localized TODO that is completed by an LLM using the same SPEC as guidance; the resulting harness is then validated as usual. End-to-end tests



(a) Success Rate on TransCoder-IR

(b) Success Rate on Project CodeNet

Figure 10: Success Rate of SACTOR with different temperature settings for GPT-4o on TransCoder-IR and Project CodeNet datasets.

run against the linked harness and idiomatic crate; passing tests provide confidence under their coverage, while failures trigger the paper's feedback procedure for regeneration and refinement.

SPEC rule reference

This section explains the rule families the SPEC uses to describe how unidiomatic, C-facing values become idiomatic Rust and back. The schema has two top-level forms: a struct description and a function description. Both are expressed as small collections of field mappings from the unidiomatic side to idiomatic paths; a function return is just another mapping whose idiomatic path is the special name `ret`. This uniform treatment keeps the generator simple and makes the SPEC readable by humans and machines alike.

Pointer handling is captured by a compact notion of shape. A field is either a scalar or one of three pointer shapes: a byte string that follows C conventions, a slice that pairs a pointer with a length, or a single-object reference. Slices record where their length comes from (either a sibling field or a constant). Each pointer also carries a null policy that distinguishes admissible NULL from forbidden NULL, which in turn selects idiomatic options versus strict checks in the generated adapters.

Two lightweight hints influence how the harness allocates and how the roundtrip self-check behaves. An ownership hint (owning vs transient) signals whether the idiomatic side should materialize owned data or borrow it for the duration of the call. A comparison hint (by value, by slice, or skip) declares how roundtrip checks should assert equality, so that aliasing views or externally owned buffers can be skipped without producing spurious failures.

Finally, the schema enforces well-formedness and defines a safe escape hatch. Invalid combinations are rejected early by validation. Patterns that are valid but not yet implemented by the generator, such as complex dotted paths or unusual pointer views, are localized and handed to the LLM fallback described earlier; the SPEC itself remains the single source of truth for the intended mapping.

M Real-world Codebase Evaluation Details

M.1 CRust-Bench Per-sample Outcomes

Table 14 lists, for each of the 50 samples, the function-level translation status and a concise failure analysis. Status is reported as per-sample function-level percentages in separate columns for the unidiomatic (Unid.) and idiomatic (Id.) stages.

M.2 `libogg` Outcomes

(1) Using GPT-4o. 36 functions cannot be translated idiomatically. nine of the translation failures are caused by translated functions not passing the test cases of `libogg`. Six failures are due to compile errors in the translations, five of which result from the LLM violating Rust's safety rules on lifetime, borrow, and mutability. For example, the translation of function `_os_lacing_expand` fails because the translation sets the value of a function

Category	SPEC keys	U (C/ABI) → I (Rust)	Notes / Status
Scalars	`shape: "scalar"`	scalar → scalar	Common libc types are cast with `as` when needed; default compare is by value in roundtrip selftest.
C string	`ptr.kind: "cstring"`, `ptr.null`	`*const/*mut c_char` → `String` / `&str` / `Option<String>`	NULL handling via `ptr.null` or `Option<>`; uses `CStr`/`CString` with lossless fallback. Return strings are converted back to `*mut c_char`.
Slices	`ptr.kind: "slice"`, `len_from|len_const`	`*const/*mut T` + length → `Vec<T>`, `&[T]`, or `Option<...>`	Requires a length source; empty or NULL produces `None` or empty according to spec; writes back length on I→U when a paired length field exists.
Single-element ref	`ptr.kind: "ref"`	`*const/*mut T` → `Box<T>` / `Option<Box<T»`	For struct `T`, generator calls auto struct converters `C T_to_T_mut`/`T_to_C T_mut`.
Derived length path	idiomatic path ending with `.len`	`len` field ↔ `vec.len`	Recognizes idiomatic `data.len` and reuses the same U-side length field on roundtrip.
Nullability	`ptr.null: nullable|forbidden`	C pointers → field with/without `Option`	`nullable` maps to `Option<>` or tolerant empty handling.
&mut struct params	ownership: transient	`*mut CStruct` → `&mut Struct` or `Option<&mut Struct>`	Copies back mutated values after the call using generated struct converters.
Return mapping	Field with `i_field.name = "ret"`	idiomatic return → U output(s)	Scalars: direct or via `*mut T`. Strings: to `*mut c_char`. Slices: pointer + length writeback. Structs: via struct converters.
Comparison hints	`compare: by_value|by_slice|skip`	selftest behavior	Optional per-field checks after U→I1→U→I2 roundtrip, and compare with I1 and I2
Unsupported paths	All SPEC key pairs other than supported paths	fallback	Generator emits localized TODOs for LLM completion; schema validation rejects malformed SPECs.

Table 13: SPEC-driven harness coverage. U denotes the unidiomatic C-facing representation; I denotes the idiomatic Rust side.

parameter to a reference to the function's local variable `vec`, leading to an error "`vec` does not live long enough." Two failures are due to SACTOR being unable to generate compilable test harnesses. If a function calls another function that SACTOR cannot translate, then the caller function cannot be translated either. This is the reason why the remaining 13 translations fail.

(2) Using GPT-5. 17 functions cannot be translated idiomatically. Among them, three are because the generated functions cannot pass the test cases and three are due to failure to generate compilable test harnesses. Only one is caused by a compile error in the translated function, which shows the progress of GPT-5 in understanding Rust grammar and fixing compile errors. The remaining failures result from the callee functions of those functions being untranslatable.

Sample	Unid. (%)	Id. (%)	Failure reason	Category
2DPartInt	100.0%	100.0%	–	–
42-Kocaeli-Printf	75.0%	–	C variadics require unstable c_variadic; unresolved va_list import blocks build.	Unidiomatic compile (C varargs/unstable feature)
CircularBuffer	100.0%	54.6%	CamelCase-to-snake_case renaming breaks signature lookup; later run panics under no-unwind context.	Idiomatic compile (symbol/name mapping)
FastHamming	100.0%	60.0%	Output buffer sized to input length in harness; bounds-check panic at runtime.	Harness runtime (buffer/length)
Holdem-Odds	100.0%	6.9%	Off-by-one rank yields out-of-bounds bucket index; SIGSEGV under tests.	Runtime fault (boundary/indexing)
Linear-Algebra-C	100.0%	44.8%	Pointer vs reference semantics mismatch (nullable C pointers vs Rust references); harness compile errors.	Harness compile (pointer/ref semantics)
NandC	100.0%	100.0%	–	–
Phills_DHT	75.0%	–	Shadowed global hash_table keeps dht_is_initialised() false; assertion in tests.	Runtime fault (global state divergence)
Simple-Sparsehash	100.0%	40.0%	CamelCase-to-snake_case renaming causes signature/type mismatches; harness does not compile.	Idiomatic compile (symbol/name mapping)
SimpleXML	83.3%	–	Missing ParseState and CamelCase-to-snake_case renaming breaks signatures; unidiomatic stalls.	Idiomatic compile (symbol/name mapping)
aes128-SIMD	85.7%	–	Array-shape mismatch (expects 4x4 refs; passes row pointer); plus intrinsics/typedef noise.	Unidiomatic compile (array shape; intrinsics/types)
amp	80.0%	–	Returned C string from amp_decode_arg is not NULL-terminated; strcmp reads past allocation and trips invalid read under tests.	Runtime fault (C string NULL termination)
approxidate	85.7%	–	match_alpha references anonymous enum C2RustUnnamed that is never defined, causing cascaded missing-type errors across retries.	Unidiomatic compile (types/aliases)
avalanche	100.0%	75.0%	Capturing closure passed where fn pointer required; FILE*/Rust File bridging mis-modeled; compile fails.	Harness runtime (I/O/resource model mismatch)
bhshell	88.2%	–	Many parser errors (enum lacks PartialEq, missing consts, u64 to usize drift, duplicates).	Unidiomatic compile (types/aliases)
bitset	100.0%	50.0%	Treats bit count as byte count in converter; overreads and panics under tests.	Harness runtime (buffer/length)
bostree	52.4%	–	Function-pointer typedefs and pointer-shape drift break callback bridging.	Unidiomatic compile (function-pointer types/deps)
btree-map	100.0%	26.3%	Trace/instrumentation proc macro requires Debug on opaque C type node; harness compilation fails for get_node_count.	Harness compile (instrumentation bound)
c-aces	100.0%	3.9%	Struct converter mismatch (Vec<CMatrix2D> vs Vec<Matrix2D>) in generated harness; compile fails after retries.	Harness compile (struct converter/shape)
c-string	100.0%	29.4%	Size vs capacity mismatch in StringT constructor; empty buffer returned, C asserts.	Runtime fault (size/capacity mismatch)
carrays	100.0%	68.5%	Trace macro imposes Debug on generic T and callback; harness fails to compile (e.g., gca_lsearch).	Harness compile (instrumentation bound)
cfsm	50.0%	–	Missing typedefs for C function-pointer callbacks; harness lacks nullable extern signatures, compile fails.	Unidiomatic compile (function-pointer types/deps)
chtrie	100.0%	0.0%	Pointer-of-pointers vs Vec adapter mismatch for struct chtrie	Harness compile (struct converter/shape)
cissy	100.0%	19.1%	Anonymous C types that c2rust renamed cannot be fetched correctly as a dependency	Unidiomatic compile (types/aliases)
clog	31.6%	–	Variadic logging APIs and duplicate globals; unresolved vfprintf/c_variadic; compile fails.	Unidiomatic compile (C varargs/unstable feature)
cset	100.0%	25.0%	Translator renames XXH_readLE64 to xxh_read_le64; SPEC/harness require exact C name; exhausts six attempts.	Idiomatic compile (symbol/name mapping)
csyncmers	66.7%	–	Unsigned underflow in compute_closed_syncmers (i - S + 1 without guard) triggers overflow panic; prior __uint128_t typedef issues.	Runtime fault (arithmetic underflow)
dict	17.7%	–	Fn-pointer fields modeled non-optional (need Option<extern "C" fn>); plus va_list requires nightly c_variadic; compile fails.	Unidiomatic compile (function-pointer types/deps)
emlang	16.3%	–	Anonymous-union alias (C2RustUnnamed) misuse; duplicate program_new; assertion bridging (__assert_fail) mis-modeled.	Unidiomatic compile (types/aliases)
expr	33.3%	–	Missing C2RustUnnamed alias; C varargs in trace_eval; strncmp len type mismatch.	Unidiomatic compile (types/aliases)
file2str	100.0%	100.0%	–	–
fs_c	100.0%	60.0%	Idiomatic I/O wrappers mismatch C expectations (closed fd/OwnedFd abort; Err(NotFound) leads to C-side segfault).	Harness runtime (I/O/resource model mismatch)
geofence	100.0%	100.0%	–	–
gfc	100.0%	54.6%	Converter overread + ownership misuse; later compile errors.	Harness runtime (converter/ownership)
gorilla-paper-encode	100.0%	9.1%	Missing adapters + lifetimes (Cbitwriter_s/Cbitreader_s vs BitWriter/BitReader<'a>).	Harness compile (lifetimes/struct adapters)
hydra	100.0%	50.0%	Borrow overlap in list update; name mapping for FindCommand.	Idiomatic compile (borrow/lifetime; symbol mapping)
inversion_list	17.0%	–	C allows NULL comparator/function pointers; wrapper unwraps and panics.	Runtime fault (function-pointer nullability)
jccc	88.7%	–	Missing C2RustUnnamed alias and duplicate Expression/Lexer types; compile fails.	Unidiomatic compile (types/aliases)
leftpad	100.0%	100.0%	–	–
lib2bit	100.0%	13.6%	Non-clonable std::fs::File in harness (C FILE* vs Rust File I/O handle mismatch)	Harness runtime (I/O/resource model mismatch)
libbase122	100.0%	37.5%	Reader cursor/buffer not preserved across calls; writer shape mismatch; tests fail.	Harness runtime (converter/ownership)
libbeaufort	100.0%	66.7%	Returns reference to temporary tableau; matrix parameter shape drift (char** vs Vec<Option<String»); compile fails.	Idiomatic compile (borrow/lifetime)
libwecan	100.0%	100.0%	–	–
morton	100.0%	100.0%	–	–
murmurhash_c	100.0%	100.0%	–	–
razz_simulation	33.3%	–	Type-name drift; node shape; ptr/ref API mismatch.	Harness compile (type/name drift; API mismatch)
rhbloom	100.0%	33.3%	Pointer/ref misuse; bit-length as bytes; overreads/panics.	Harness runtime (pointer/ref; length units)

Sample	Unid. (%)	Id. (%)	Failure reason	Category
`totp`	77.8%	–	Anonymous C types that c2rust renamed cannot be fetched correctly as a dependency; plus duplicate helpers (`pack32`/`unpack64`/`hmac_sha1`); compile fails.	Unidiomatic compile (types/aliases)
`utf8`	100.0%	30.8%	NULL deref + unchecked indices; SIGSEGV in tests.	Runtime fault (NULL deref/out-of-bounds)
`vec`	100.0%	0.0%	Idiomatic rewrite uses a bounds-checked copy; out-of-range panic under tests.	Runtime fault (boundary/indexing)

Table 14: CRust-Bench per-sample outcomes (function-level). Translation Status columns report per-sample function-level success rates for unidiomatic (Unid.) and idiomatic (Id.) stages.

N Examples of Prompts Used in SACTOR

The following prompts are used to guide the LLM in C-to-Rust translation and verification tasks. The prompts may slightly vary to accommodate different translation task, as SACTOR leverages static analysis to fetch the necessary information for the LLM.

N.1 Unidiomatic Translation

Figure 11 shows the prompt for translating unidiomatic C code to Rust.

```
Translate the following C function to Rust. Try to
     keep the **equivalence** as much as possible.
`libc` will be included as the **only** dependency
     you can use. To keep the equivalence, you can
     use `unsafe` if you want.
The function is:
```c
{C_FUNCTION}
```

// Specific for main function
The function is the `main` function, which is the
     entry point of the program. The function
     signature should be: `pub fn main() -> ()`.
For `return 0;`, you can directly `return;` in Rust
     or ignore it if it's the last statement.
For other return values, you can use `std::process::
     exit()` to return the value.
For `argc` and `argv`, you can use `std::env::args()
     ` to get the arguments.

The function uses some of the following stdio file
     descriptors: stdin. Which will be included as
```rust
extern "C" {
    static mut stdin: *mut libc::FILE;
}

```
You should **NOT** include them in your translation,
      as the system will automatically include them.

The function uses the following functions, which are
      already translated as (you should **NOT**
     include them in your translation, as the system
      will automatically include them):
```rust
{DEPENDENCIES}
```

Output the translated function into this format (
     wrap with the following tags):
----FUNCTION----
```rust
// Your translated function here
```
----END FUNCTION----
```

Figure 11: Unidiomatic Translation Prompt

N.2 Unidiomatic Translation with Feedback

Figure 12 shows the prompt for translating unidiomatic C code to Rust with feedback from the previous incorrect translation and error message.

```
Translate the following C function to Rust. Try to
     keep the **equivalence** as much as possible.
`libc` will be included as the **only** dependency
     you can use. To keep the equivalence, you can
     use `unsafe` if you want.
The function is:
```c
{C_FUNCTION}

```

// Specific for main function
The function is the `main` function, which is the
     entry point of the program. The function
     signature should be: `pub fn main() -> ()`.
For `return 0;`, you can directly `return;` in Rust
     or ignore it if it's the last statement.
For other return values, you can use `std::process::
     exit()` to return the value.
For `argc` and `argv`, you can use `std::env::args()
     ` to get the arguments.

The function uses some of the following stdio file
     descriptors: stdin. Which will be included as
```rust
extern "C" {
    static mut stdin: *mut libc::FILE;
}

```
You should **NOT** include them in your translation,
      as the system will automatically include them.

The function uses the following functions, which are
      already translated as (you should **NOT**
     include them in your translation, as the system
      will automatically include them):
```rust
fn atoi (str : * const c_char) -> c_int;
```

Output the translated function into this format (
     wrap with the following tags):
----FUNCTION----
```rust
// Your translated function here
```
----END FUNCTION----

Lastly, the function is translated as:
```rust
{COUNTER_EXAMPLE}
```
It failed to compile with the following error
     message:
```
{ERROR_MESSAGE}
```
Analyzing the error messages, think about the
     possible reasons, and try to avoid this error.
```

Figure 12: Unidiomatic Translation with Feedback Prompt

N.3 Idiomatic Translation

Figure 13 shows the prompt for translating unidiomatic Rust code to idiomatic Rust. Crown is used to hint the LLM about the ownership, mutability, and fatness of pointers.

```
Translate the following unidiomatic Rust function
     into idiomatic Rust. Try to remove all the `
     unsafe` blocks and only use the safe Rust code
     or use the `unsafe` blocks only when necessary.
Before translating, analyze the unsafe blocks one by
      one and how to convert them into safe Rust
     code.
**libc may not be provided in the idiomatic code, so
      try to avoid using libc functions and types,
     and avoid using `std::ffi` module.**
```rust
{RUST_FUNCTION}
```

"Crown" is a pointer analysis tool that can help to
     identify the ownership, mutability and fatness
     of pointers. Following are the possible
     annotations for pointers:
```
fatness:
    - `Ptr`: Single pointer
    - `Arr`: Pointer is an array
mutability:
    - `Mut`: Mutable pointer
    - `Imm`: Immutable pointer
ownership:
    - `Owning`: Owns the pointer
    - `Transient`: Not owns the pointer
````

The following is the output of Crown for this
     function:
```
{CROWN_RESULT}
```
Analyze the Crown output firstly, then translate the
      pointers in function arguments and return
```

```
    values with the help of the Crown output.
Try to avoid using pointers in the function
    arguments and return values if possible.

Output the translated function into this format (
    wrap with the following tags):
----FUNCTION----
```rust
// Your translated function here
```
----END FUNCTION----

Also output a minimal JSON spec that maps the
    unidiomatic Rust layout to the idiomatic Rust
    for the function arguments and return value.
Full JSON Schema for the SPEC (do not output the
    schema; output only an instance that conforms
    to it):
```json
{_schema_text}
```
----SPEC----
```json
{{
  "function_name": "{function.name}",
  "fields": [
    {{
      "u_field": {{
        "name": "...",
        "type": "...",
        "shape": "scalar" | {{"ptr": {{"kind": "
    slice|cstring|ref", "len_from": "?", "len_const
    ": 1}}}}
      }},
      "i_field": {{
        "name": "...",
        "type": "..."
      }}
    }}
  ]
}}
```
----END SPEC----
Few-shot examples (each with unidiomatic Rust
    signature, idiomatic Rust signature, and the
    SPEC):
Example F1 (slice arg):
Unidiomatic Rust:
```rust
pub unsafe extern "C" fn sum(xs: *const i32, n:
    usize) -> i32;
```
Idiomatic Rust:
```rust
pub fn sum(xs: &[i32]) -> i32;
```
----SPEC----
```json
{{
  "function_name": "sum",
  "fields": [
    {{ "u_field": {{"name": "xs", "type": "*const
    i32", "shape": {{ "ptr": {{ "kind": "slice", "
    len_from": "n" }} }} }},
       "i_field": {{"name": "xs", "type": "&[i32]"
    }} }},
    {{ "u_field": {{"name": "n",  "type": "usize",
        "shape": "scalar" }},
       "i_field": {{"name": "xs.len", "type": "usize
    " }} }}
  ]
}}
```
----END SPEC----

Example F2 (ref out):
Unidiomatic Rust:
```rust
pub unsafe extern "C" fn get_value(out_value: *mut
    i32);
```
Idiomatic Rust:
```rust
pub fn get_value() -> i32;
```
----SPEC----
```json
{{
  "function_name": "get_value",
  "fields": [
    {{ "u_field": {{"name": "out_value", "type": "*
    mut i32", "shape": {{ "ptr": {{ "kind": "ref"
    }} }} }},
       "i_field": {{"name": "ret", "type": "i32" }}
    }}
  ]
}}
```
----END SPEC----
Example F3 (nullable cstring maps to Option):
Unidiomatic Rust:
```rust
pub unsafe extern "C" fn set_name(name: *const libc
    ::c_char);
```
Idiomatic Rust:
```rust
pub fn set_name(name: Option<&str>);
```
----SPEC----
```json
{{
  "function_name": "set_name",
  "fields": [
    {{ "u_field": {{"name": "name", "type": "*const
    c_char", "shape": {{ "ptr": {{ "kind": "cstring
    ", "null": "nullable" }} }} }},
       "i_field": {{"name": "name", "type": "Option
    <&str>" }} }}
  ]
}}
```
----END SPEC----
```

Figure 13: Idiomatic Translation Prompt

N.4 Idiomatic Verification

Idiomatic verification is the process of verifying the correctness of the translated idiomatic Rust code by generating a test harness. The prompt for idiomatic verification is shown in Figure 14.

```
We have an initial spec-driven harness with TODOs.
    Finish all TODOs and ensure it compiles.
Idiomatic signature:
```rust
pub fn compute_idiomatic(
    x: i32,
    name: &str,
    data: &[u8],
    meta: HashMap<String, String>,
) -> i32;;
```
Unidiomatic signature:
```rust
pub unsafe extern "C" fn compute(x: i32, name: *
    const libc::c_char, data: *const u8, len: usize
    , meta: *const libc::c_char) -> i32;;
```

Current harness:
```rust
pub unsafe extern "C" fn compute(x: i32, name: *
    const libc::c_char, data: *const u8, len: usize
    , meta: *const libc::c_char) -> i32
{
    // Arg 'name': borrowed C string at name
    let name_str = if !name.is_null() {
        unsafe { std::ffi::CStr::from_ptr(name) }.
    to_string_lossy().into_owned()
    } else {
        String::new()
    };
    // Arg 'data': slice from data with len len as
    usize
    let data_len = len as usize;
    let data_len_non_null = if data.is_null() { 0 }
    else { data_len };
    let data: &[u8] = if data_len_non_null == 0 {
        &[]
    } else {
        unsafe { std::slice::from_raw_parts(data as
    *const u8, data_len_non_null) }
    };
    // TODO: param meta of type HashMap < String ,
    String >: unsupported mapping
    let __ret = compute_idiomatic(x, &name_str, data
    , /* TODO param meta */);
    return __ret;
}
```
Output only the final function in this format:
----FUNCTION----
```rust
// Your translated function here
```
----END FUNCTION----
```

Figure 14: Idiomatic Verification Prompt

N.5 Failure Reason Analysis

Figure 15 shows the prompt for analyzing the reasons for the failure of the translation.

```
Given the following C code:
```c
{original_code}
```
The following code is generated by a tool that
     translates C code to Rust code. The tool has a
     bug that causes it to generate incorrect Rust
     code. The bug is related to the following error
      message:
```json
{json_data}
```
Please analyze the error message and provide a
     reason why the tool generated incorrect Rust
     code.

1. Append a new reason to the list of reasons.
2. Select a reason from the list of reasons that
     best describes the error message.

Please provide a reason why the tool generated
     incorrect Rust code **FUNDAMENTALLY**.

List of reasons:
{all_current_reasons}

Please provide the analysis output in the following
     format:
```json
{
    "action": "append", // or "select" to select a
     reason from the list of reasons
    "reason": "Format string differences between C
     and Rust", // the reason for the error message,
      if action is "append"
    "selection": 1 // the index of the reason from
     the list of reasons, if action is "select"
    // "reason" and "selection" are mutually
     exclusive, you should only provide one of them
}
```

Please **make sure** to provide a general reason
     that can be applied to multiple cases, not a
     specific reason that only applies to the
     current case.
Please provide a reason why the tool generated
     incorrect Rust code **FUNDAMENTALLY** (NOTE
     that the reason of first failure is always NOT
     the fundamental reason).
```

Figure 15: Failure Reason Analysis Prompt