LLM-Driven Multi-step Translation from C to Rust using Static Analysis

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Translating software written in C to Rust has significant benefits in improving memory safety while maintaining high performance. 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 as they lack the ability to capture the semantic differences between the source and target languages. We propose SACTOR, an LLM-driven C-to-Rust translation tool that employs a two-step process: an initial "unidiomatic" translation to preserve semantics, followed by an "idiomatic" refinement to align with Rust standards. SACTOR leverages static analysis of the C source to handle pointer semantics and dependency resolution. To validate the correctness of step-wise translation, we use end-to-end testing via the foreign function interface. We evaluate the translation of 200 programs from two datasets and two case studies, comparing the performance of GPT-4o, Claude 3.5 Sonnet, Gemini 2.0 Flash, Llama 3.3 70B and DeepSeek-R1 in SACTOR. Our results demonstrate that SACTOR achieves high correctness and enhanced idiomaticity, with the best-performing model (DeepSeek-R1) reaching 93% and 84% correctness (on each dataset, respectively), while generating more idiomatic, Rust-compliant code, reducing `Clippy` lint alerts by up to 7×, and producing unsafe-free translations on both datasets compared to existing methods.

1 Introduction

The C language is the most widely used language in system-level programming, because of its ability to directly manipulate memory and raw hardware [1]. However, its manual memory management has led to memory-related bugs, such as buffer overflows, dangling pointers, and memory leaks, which have been the root cause of many security vulnerabilities [2]. To address these shortcomings, Rust has emerged as a promising alternative; it is a modern system-programming language that offers memory safety without relying on garbage collection by imposing a strict object ownership model [3]. Rust is adopted by projects like the Linux kernel in developing new hardware drivers[1] and Mozilla Firefox. *Translating legacy C code to Rust – particularly into idiomatic Rust rather than a direct, unidiomatic mapping – can improve safety and performance, but doing so manually is time-intensive, error-prone, and requires expertise in both languages.*

Traditional automatic translation tools such as C2Rust [4] analyze the abstract syntax tree (AST) of C code to generate syntactically equivalent Rust code. However, these rule-based or statically analyzed approaches [5, 4, 6, 7, 8] often fail to produce "idiomatic" Rust: code that follows the Rust programming model and conventions with the least use of `unsafe` blocks. *Due to fundamental differences*

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

in semantics and memory management between C and Rust, idiomatic translations are crucial to compiler-enforced memory safety, as well as improving code readability and maintainability.

While large language models (LLMs) demonstrate the ability to "understand" syntax and code semantics [9], they are prone to hallucinations and often generate incorrect or semantically misaligned code [10]. In C-to-Rust translation, naive LLMs lacking structure awareness frequently produce unsafe or code. Prior work has explored translation quality through various prompting strategies [11, 12, 13] or ensure correctness via verification techniques, such as fuzz testing and symbolic execution [14, 15], to validate LLM-generated code. *While these methods improve semantic alignment or enforce correctness, they often struggle with complex code and fall short of producing idiomatic, safety-compliant Rust.* For example, Vert [14] struggles to verify programs involving data structures through symbolic execution, and C2SaferRust [12] still produces Rust code with substantial use of `unsafe`.

In this paper, we introduce Structure-Aware C-to-Rust Translator (or SACTOR), an LLM-driven zero-shot translation tool. An overview of SACTOR is presented in Figure 4. We divide the translation process into two stages:

1. **C to Unidiomatic Rust** – This phase performs a syntactic translation, preserving C-like semantics with `unsafe` blocks to handle low-level memory operations.
2. **Unidiomatic Rust to Idiomatic Rust** – This phase refines the translation to Rust's semantic standards, eliminating `unsafe` blocks to guarantee memory safety.

Throughout both stages, we leverage static analysis to extract pointer semantics and dependency information, guiding the translation. To efficiently verify LLM-generated code, we embed the translated Rust alongside the original C implementation via Foreign Function Interface (FFI), enabling end-to-end testing of the complete program for correctness. The two-phase strategy separates syntax from semantics, allowing the LLM to focus on a simplified task at each step and improving translation correctness, while static analysis further enhances this process by supplying precise program semantics, helping the LLM generate idiomatic, memory-safe Rust. These steps combined significantly improve translation quality, achieves high-level correctness, and provides a broader applicability to different types of C programs.

In summary, we make the following contributions:

- We present SACTOR[2], a tool that integrates information extracted from static analysis into a two-step translation pipeline. SACTOR reliably produces idiomatic, `unsafe`-free Rust from C (§ 4). A full example translation is provided in Appendix G.
- We introduce a novel verification approach that embeds the translated Rust code alongside the original C implementation using FFI. This enables practical, scalable, and efficient end-to-end testing for LLM-generated code (§ 4.3).
- We evaluate SACTOR on a diverse set of C programs across two datasets and five LLMs. Our best-performing model, DeepSeek-R1, achieves a 93% and 84% success rate on each dataset, respectively (§ 6.1). Notably, SACTOR generates more idiomatic, `unsafe`-free Rust compared to four existing methods (§ 6.2). These results demonstrate SACTOR's effectiveness.
- Finally, we present a comprehensive cost and diagnostic analysis:
    1. *Token Efficiency:* GPT-4o is the most token-efficient, while DeepSeek-R1 consumes 5–7$\times$ more tokens. The average number of queries varies less, with Llama 3.3 taking only $\sim$1.4$\times$ more queries than Gemini 2.0 (Appendix K).
    2. *Feedback Improvement:* Incorporating compilation and testing feedback significantly boosts the success rate of weaker models such as Llama 3.3 by around 17% (Appendix L).
    3. *Temperature Sensitivity:* Temperature adjustments have a negligible effect, with only slight improvements observed at lower values (Appendix M).
    4. *Failure Analysis:* Reasoning models like DeepSeek-R1 excel at resolving complex issues such as format string and array manipulation errors (Appendix J).

2 Background

Primer on C and Rust: C is one of the most widely used programming languages [16]. It is a low-level language that provides direct access to memory and hardware through pointers and abstracts

[2]SACTOR code: `https://anonymous.4open.science/r/sactor-C8D6`; datasets: `https://anonymous.4open.science/r/sactor-datasets-DE21`.

the machine-level instructions. While this makes it efficient, it suffers from memory vulnerabilities such as buffer overflow [17, 18], dangling pointers [19], and memory leaks [20]. Rust, in contrast, is a modern programming language that provides memory safety without additional performance penalty, and has the same ability to access low-level hardware as C. Instead, Rust enforces strict compile-time memory safety through *ownership*, *borrowing*, and *lifetimes*—mechanisms designed and undergoing formal verification to eliminate memory vulnerabilities [3, 21].

Challenges in Code Translation: Despite its advantages, Rust is a relatively new language; many widely-used system-level programs remain in C, which lacks memory safety and is prone to vulnerabilities. It's desirable to translate such programs to Rust and gain security and reliability benefits, but the process is challenging due to fundamental language differences. Figure 5 in Appendix C shows an example of a simple C program and its Rust equivalent to illustrate the differences of two languages in terms of memory management and error handling. While Rust permits `unsafe` blocks for C-like pointer operations, their use is discouraged due to the absence of compiler guarantees and their non-idiomatic nature for further maintenance.

Other differences include string representation, pointer usage, array handling, reference lifetimes, and error propagation. A non-exhaustive summary appears in Appendix C. These gaps make translation non-trivial: an effective translator must (1) understand the semantics of both languages, (2) reason over memory management differences, and (3) handle language-specific constructs. In addition, a practical correctness verification approach for the translated code is also essential.

3 Related Work

LLMs for C-to-Rust Translation: Vert [14] uses LLMs to generate translation candidates and applies fuzz testing and symbolic execution to verify equivalence. While this ensures syntactic and semantic correctness, this verification method is too strict and struggles with scalability and complex C features. Flourine [15] guides LLMs using error feedback and verifies correctness via fuzz testing, while employing data type serialization to address type mismatches; however, serialization issues still account for half of the translation errors. Shiraishi and Shinagawa [13] segment C code into smaller sub-tasks and guide LLMs to translate specific constructs (e.g., macros) using predefined Rust idioms, but only evaluate compilation success without verifying functional correctness. Shetty et al. [11] use dynamic analysis to analyze the runtime behavior of the C code, and uses this information to guide LLM translation. However, dynamic analysis is limited by input coverage, making it challenging to generalize across all execution paths. Nitin et al. [12] refine C2Rust outputs using LLMs to reduce unidiomatic Rust (`unsafe`, `libc`), but the output still rely heavily on them. The overall translation quality is restricted by C2Rust, which removes comments and preprocessor directives (§ 4.2.1) and makes the LLM harder to interpret the code and produce a more idiomatic translation.

Non-LLM Approaches for C-to-Rust Translation: C2Rust [4] translates C to Rust by converting the C abstract syntax tree (AST) into a Rust AST, followed by a rule-based transformation into Rust code. Although it ensures syntactic accuracy, the output remains a purely structural translation that relies heavily on `unsafe` blocks and suffers from low readability due to direct construct mapping and explicit type conversions. Crown [5] is a tool that can analyze the pointer information in C code based on its static ownership tracking. It uses this information to generate Rust code with the corresponding semantics and reduce the pointer usage in the translated code. Hong and Ryu [7] proposed an approach to handle the return value of functions in C-to-Rust translation. Ling et al. [8] translate Rust based on a set of predefined rules and heuristics. Although these approaches reduce the use of `unsafe` blocks compared to C2Rust, the translated code still heavily relies on `unsafe` blocks and remains largely unidiomatic.

4 SACTOR Methodology

We propose SACTOR, an LLM-driven C-to-Rust translation tool using a two-step translation methodology. The overview of SACTOR's methodology is shown in Figure 4 in Appendix B. Recall that the semantics of Rust is different from that of C (§ 2). To assist the LLM, and capture more nuanced semantic information, we utilize a suite of static analysis tools to provide additional information in the form of hints to the LLM. We outline SACTOR's four main stages 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 [22, 23]; 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 definitions, declarations, and dependencies–from C code, *we developed a static analysis tool called* `C Parser` based on `libclang`. `libclang` is 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. In Appendix D, we provide more details on how we divide the program.

4.2 Code 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.

4.2.1 Unidiomatic Rust Translation

We aim to produce semantically equivalent Rust code from the original C code, allowing the use of `unsafe` blocks and C standard library functions. For data type translation, we leverage C2Rust [4] to convert C code into Rust. 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. For other language primitives like global variables, we extract them via static analysis and map them to their Rust equivalents using C2Rust-derived hints, preserving their mutability and visibility semantics.

4.2.2 Idiomatic Rust Translation

The goal of this step is to refine unidiomatic Rust into idiomatic Rust by removing `unsafe` blocks and following Rust idioms. 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 [5] 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.

4.3 Verification

To verify the equivalence between source and target languages, prior work has relied on symbolic execution and fuzz testing. However, these methods are impractical for real-world C-to-Rust translation (details in Appendix E). We propose a *new approach* to validate the correctness of translated Rust code by focusing on *soft equivalence*–ensuring functional equivalence of the entire program based on success on end-to-end (E2E) tests. This approach 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 under this framework depends on the code coverage of accompanying E2E tests: the broader the coverage, the stronger the assurance of equivalence.

4.3.1 Verifying Unidiomatic Rust Code

Verifying the unidiomatic Rust code is straightforward, as it is semantically equivalent to the original C code and maintains compatible function signatures and data types. This compatibility ensures a consistent Application Binary Interface (ABI) between the two languages, 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.

4.3.2 Verifying Idiomatic Rust Code

The verification of idiomatic Rust code is more complex, as the idiomatic Rust code does not align with the original C code in data types or function definitions. Direct linking between the original C code and the idiomatic Rust code is therefore infeasible. To address this, we use the LLM to create a *test harness* for each target function. The test harness mirrors the function definition of the unidiomatic Rust version, allowing it to accept C data types as input and return C data types as output. Within the harness, inputs are converted from C data types to their corresponding Rust types before calling the target function. The output is then converted back from Rust types to C types to maintain type consistency. Due to space constraints, Figure 6 in Appendix F presents an example of such a harness. To ensure reliability, all harnesses are automatically synthesized and validated through compilation and end-to-end testing. If any test fails, we regenerate both the test harness and the target function, and repeat the verification process until all tests pass. Once the harness is constructed, it can be linked to the original C code, as in § 4.3.1. E2E tests are then executed for the entire program, enabling verification of the idiomatic Rust code even when it diverges from the original C code in data types and function definitions.

What does the harness do? The harness relies on type conversions to bridge the semantic gap between C and Rust representations. We categorize these conversions into two cases:

- *Basic types:* Simple conversions like `int` to `i32`, `float` to `f32`, and `char*` to `String` or `&str`, which are straightforward for the LLM to handle.
- *Custom data structures:* These conversions are more complex due to layout differences between C and Rust structures. For such cases, the LLM firstly generates two conversion functions for each structure–one for converting from C-to-Rust and another for converting from Rust-to-C. These functions are tailored to specific data structures and are invoked by the test harness.

4.3.3 Feedback Mechanism

For code that fails the verification process, *feedback is provided to the translation process to help the LLM correct the issues.* If the Rust code fails to compile, the compiler errors are directly passed back to the translation process to guide the LLM in fixing the translation. For E2E test failures, a Rust procedural macro is employed to insert debugging code into the target function during compile time. This debugging code logs the inputs and outputs of the target function when a test fails. The E2E tests are then re-executed, and the collected information is fed back to the translation process.

4.4 Code Combination

By translating and verifying all functions and data types, we can integrate them into a unified Rust code-base. We first gather the translated Rust code from each sub-task, then eliminate duplicate `use` statements and other redundancies needed for individual sub-task compilation. Next, we organize the code into a well-structured, idiomatic implementation of the original C program. Once combined, the entire program is subjected to E2E tests to verify the correctness of the final Rust code. If all tests pass, the translation process is deemed successful.

5 Experimental Setup

5.1 Datasets Used

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

- *Absence of Corresponding Rust Code:* The dataset must not include equivalent Rust code to prevent LLMs from simply memorizing the dataset instead of performing genuine translation.
- *Sufficient Number of C Programs:* 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 aspects, we select two datasets TransCoder-IR [24] and Project CodeNet [25], which are widely used in the code translation domain, as well as two real-world projects: *avl-tree* and *urlparser*. For evaluation purposes, we randomly sample 100 C programs from each of the TransCoder-IR and Project CodeNet (with input arguments) datasets to ensure computational feasibility while maintaining statistical significance. The *avl-tree* and *urlparser* projects are real-world C projects with around 1000 and 500 lines of code, respectively. The complete details of these datasets are presented in Appendix H.

5.2 Evaluation Metrics

Success Rate: This is defined as the ratio of the number of programs that can (a) successfully be translated to Rust and (b) successfully pass the E2E tests for both unidiomatic and idiomatic translation phases to the total number if programs. 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 (each using feedback from "all" previous attempts). After 6 attempts, we treat this test case as failure.

Idiomaticity: To evaluate the idiomaticity of the translated code, we use three metrics:

- *Lint Alert Count* is measured by running Rust-Clippy [26], a tool that provides lints on unidiomatic Rust code (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 idiomatic translation. Previous works [14, 15] have also used Rust-Clippy for similar evaluations.
- *Unsafe Code Fraction*, inspired by Shiraishi and Shinagawa [13], 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 in Rust, 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 our approach using GPT-4o (OpenAI), Claude 3.5 Sonnet (Anthropic), Gemini 2.0 Flash (Google), DeepSeek-R1 (DeepSeek) and Llama 3.3 70B Instruct (Meta)–a fine-tuned variant of Llama 3.3 70B optimized for text generation. Except for DeepSeek-R1, all models are non-reasoning models, i.e., directly generate output without chain-of-thought reasoning. LLM configurations are detailed in Appendix I.

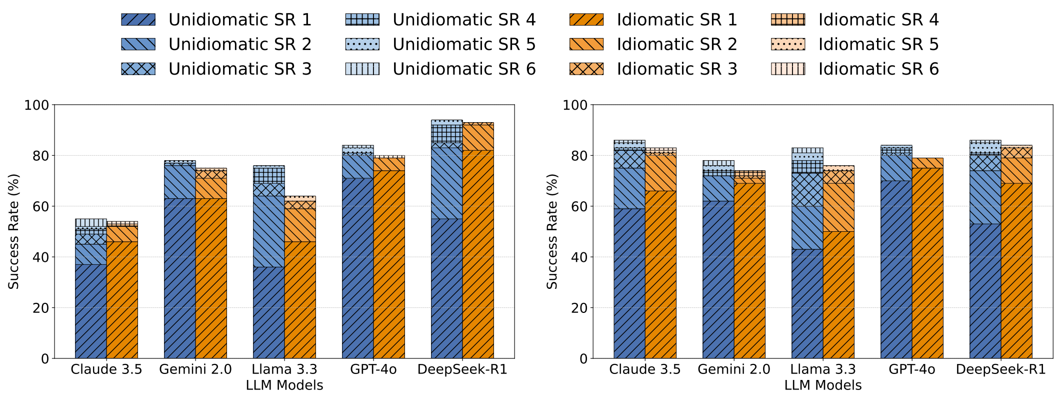
6 Evaluation

Through our evaluation, we aim to answer the following research questions: (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 complex code-bases?

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 J); (2) SACTOR's idiomatic translation results outperforms all previous baselines, producing Rust code with fewer Clippy warnings and 100% unsafe-free translations (§ 6.2); and (3) SACTOR successfully generates the verified unidiomatic translation results on complex code-bases like `avl_tree` and `urlparser`, but has difficulties verifying idiomatic translations when handling function pointers and complex data conversions (§ 6.3).

We also evaluate the computational cost of SACTOR (Appendix K), the impact of the feedback mechanism (Appendix L), and temperature settings (Appendix M) . 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



(a) TransCoder-IR SR

(b) CodeNet SR

Figure 1: 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.

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

(1) TransCoder-IR (Figure 1a): DeepSeek-R1 achieves the highest success rate (SR) in both unidiomatic (94%) and idiomatic (93%) steps, only 1% drop 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.

(2) Project CodeNet (Figure 1b): 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 comes next with 78% in the unidiomatic step and 74% in the idiomatic step, showing a consistent performance between two datasets. Llama 3.3 still exhibits a significant drops (83% to 76%) in both steps and comes to the last place 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 complex code logic and data manipulation.

6.2 Measuring Idiomaticity

We compare our approach with four baselines: C2Rust [4], Crown [5], C2SaferRust [12] and Vert [14]. Of these baselines, C2Rust is the most versatile[3], 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.

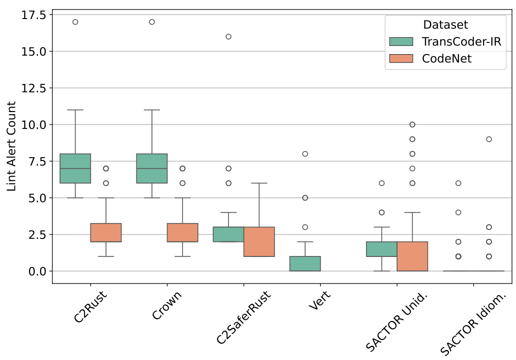


Figure 2: Total Clippy Issues (Warnings + Errors) Across Different Method

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 1: Unsafe Code Statistics. UF denotes Unsafe Free and AU denotes Avg. Unsafe

Results: Figure 2 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.

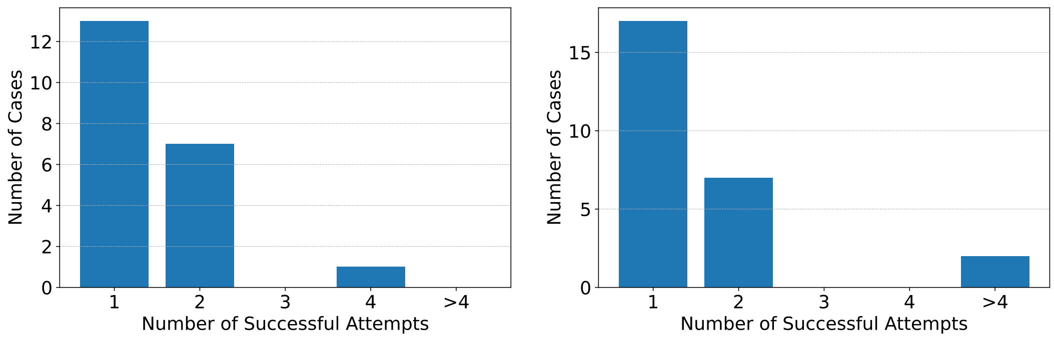
Table 1 summarizes unsafe code statistics. Unsafe-Free indicates the percentage of programs without unsafe code, while Avg. Unsafe represents the average proportion of unsafe code across all translations. C2Rust and Crown generate unsafe code in all programs with a high average unsafe percentage. C2SaferRust 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.

6.3 Complex Code-bases

We evaluate the generalization of our approach to complex code-bases by two case studies: the `avl_tree` and the `urlparser`; more details are in Appendix H. For both two case studies, we use the GPT-4o model to generate Rust code from the C code. In summary, SACTOR successfully translates the entire C project into unidiomatic Rust, achieving 10/23 function and 4/5 data type conversions. Failures stem from the LLM's difficulty in generating correct test harnesses for verification.

Case Study 1: `avl_tree`. The `avl_tree` project consists of two parts: `avl_data`, which provides helper functions for data management, and `avl_bf`, which implements the AVL tree. We successfully

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



(a) `avl_tree` Attempts Distribution

(b) `urlparser` Attempts Distribution

Figure 3: Attempts Distribution for Case Studies in Unidiomatic Translation

generate idiomatic Rust for `avl_data`, passing verification tests. However, for `avl_bf`, we only produce unidiomatic Rust, though it still passes verification.

Figure 3a shows the generation attempts. Failures stem from Rust compilation errors, which are used as feedback to refine subsequent attempts. Within 4 attempts per function, we achieve idiomatic Rust for `avl_data` and unidiomatic Rust for `avl_bf`.

The key challenge in generating idiomatic Rust for `avl_bf` lies in handling function pointers with void types:

```
int (*compare)(const void *, const void *);
```

In unidiomatic Rust, this translates directly using raw pointers:

```
Option<unsafe extern "C" fn(*const c_void, *const c_void) -> c_int>
```

For idiomatic Rust, raw pointers should be replaced with generics:

```
Option<Box<dyn Fn(&T, &T) -> i32>>
```

where `T` represents the AVL tree's data type. However, since our verification tests rely on FFI compatibility, maintaining the C interface requires raw pointers. Rust's generics are determined at compile time, making it impossible to seamlessly convert them to raw pointers. This constraint prevents our approach from producing idiomatic Rust for `avl_bf`.

Case Study 2: `urlparser`. The `urlparser` project is a simple C-based URL parser. We successfully generate unidiomatic Rust code that passes verification, as shown in Figure 3b.

For idiomatic Rust, we successfully translate 10 of 23 functions and 4 of 5 data types (enums, structures, and constants). However, the key challenge for the remaining functions is constructing a correct harness function to verify the generated code. As discussed in § 4.3.2, idiomatic Rust functions require a harness to convert C data structures to Rust and transfer outputs back. While the translated Rust code may be correct, the inability to generate a proper harness prevents verification. Without verification, we cannot confirm the correctness of these functions.

7 Conclusion

Translating C to Rust enhances memory safety but is error-prone and often unidiomatic. While LLMs improve translation, they lack correctness guarantees and struggle with semantic differences. SACTOR tackles this with a two-step approach, preserving semantics first, then refining for Rust conventions. Leveraging static analysis and FFI-based validation, it outperforms existing methods, with DeepSeek-R1 reaching 93% and 84% success rates on TransCoder-IR and Project CodeNet, respectively. However, key challenges remain: ensuring correctness, handling complex C features, scaling, interoperability, and efficiency (see Appendix A for details). Some examples of prompts used in SACTOR are shown in Appendix N.

References

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

[2] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 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, 2020.

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

[4] Immunant. C2rust, 2020. URL `https://c2rust.com/`.

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

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

[7] Jaemin Hong and Sukyoung Ryu. 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, 2024.

[8] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. 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, 2022.

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

[10] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 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, 2023.

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

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

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

[14] Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. Vert: Verified equivalent rust transpilation with few-shot learning. *arXiv preprint arXiv:2404.18852*, 2024.

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

[16] TIOBE. TIOBE Index for January 2025, 2025. URL `https://www.tiobe.com/tiobe-index/`.

[17] MITRE. CWE-121: Stack Buffer Overflow, 2006. URL `https://cwe.mitre.org/data/definitions/121.html`.

[18] MITRE. CWE-122: Heap Buffer Overflow, 2006. URL `https://cwe.mitre.org/data/definitions/122.html`.

[19] MITRE. CWE-416: Use After Free, 2006. URL https://cwe.mitre.org/data/definitions/416.html.

[20] MITRE. CWE-401: Missing Release of Memory after Effective Lifetime, 2006. URL https://cwe.mitre.org/data/definitions/401.html.

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

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

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

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

[25] 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. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

[26] The Rust Team. Clippy, 2016. URL https://github.com/rust-lang/rust-clippy.

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

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

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

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

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

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

Appendix

Note: For better formatting, each appendix section is on a new page.

A Limitations

While SACTOR has proven effective in producing correct, idiomatic Rust translations, it has several limitations: our soft-equivalence checks depend on existing end-to-end tests, so incomplete or shallow coverage can allow subtle semantic errors to go undetected. Integrating automated test-generation or fuzzing tools could fill coverage gaps and catch subtle semantic mismatches; Also, the translation quality hinges on the underlying LLM—although GPT-4o and DeepSeek-R1 perform well, other models may yield significantly lower accuracy; Our current implementation does not support certain C features–such as complex macros, pervasive function pointers, global variables, and inline assembly—which restricts SACTOR's applicability to those codebases (see § 6.3); Incorporating more advanced static analysis tools that capable of extracting more precise information from such constructs could further enhance SACTOR's translation capabilities.

B SACTOR Overview Figure

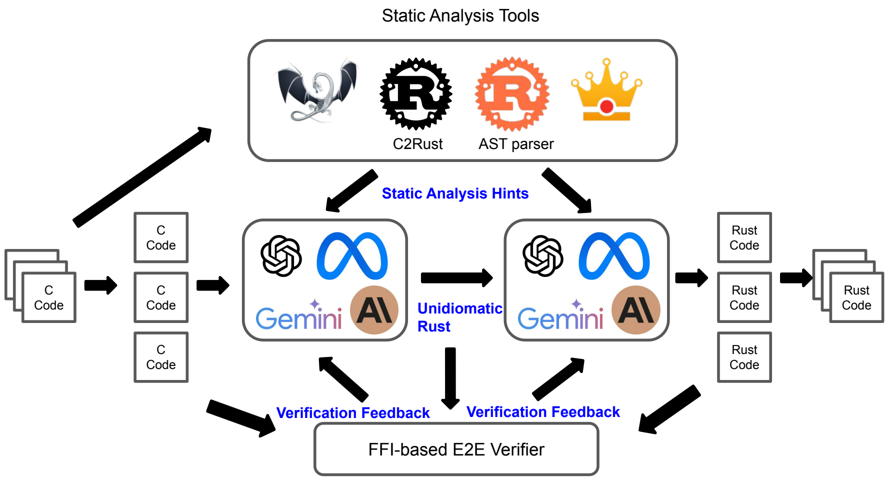


Figure 4: Overview of the SACTOR methodology.

C Differences Between C and Rust

C.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.

C.2 Tabular Summary

Here, we present a non-exhaustive list of differences between C and Rust in Table 2, 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.

Table 2: Key Differences Between C and Rust

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

D 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.

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
while $|L_{sorted}| < |L_i|$ **do**
  $L_{processed} \leftarrow \emptyset$
  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
      $L_{processed} \leftarrow L_{processed} \cup a$
    end if
  end for
  if $L_{processed} = \emptyset$ **then**
    $L_{circular} \leftarrow DFS(L_i, dep)$ ▷ Circular dependencies
    $L_{sorted} \leftarrow L_{sorted} + L_{circular}$ ▷ Add a group to sorted list
  end if
end while
return L_{sorted}

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. $DFS(L_i, dep)$ is a depth-first search function that returns a list of items involved in a circular dependency. 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, or circular dependencies are detected. If circular dependencies are detected, we resolve them through a depth-first search strategy, ensuring that all items involved in a circular dependency are grouped together and handled as a single unit.

E Equivalence Testing Details in Prior Literature

E.1 Symbolic Execution-Based Equivalence

Symbolic execution explores all potential execution paths of a program by using symbolic inputs to generate constraints [27, 28, 29]. 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.

E.2 Fuzz Testing-Based Equivalence

Fuzz testing generates random or mutated inputs to test whether program outputs match expected results [30, 31, 32]. 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.

F 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*`.

```
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

G 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.

```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

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 § 4.3.1. 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 § 4.2.2. 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.

H Dataset Details

Table 3: Summary of datasets and real-world projects used for evaluation.

DATASET	SIZE	PREPROCESSING	E2E TESTS	CORRESPONDING RUST
TRANSCODER-IR [24]	100	Removed buggy programs (compilation and memory errors) and programs that have Rust translation	Present	Absent
PROJECT CODENET [25]	100	Filtered for programs with external input (`argc`/`argv`)	Absent	Absent
REAL-WORLD PROJECTS	2	Extend macros, combine the whole project to a single file	Present (Limited)	Absent

H.1 TransCoder-IR Dataset [24]

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

H.2 Project CodeNet [25]

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.

H.3 Real-World Projects

For § 6.3, we use two real-world projects for evaluation: *avl-tree*[4] and *urlparser*[5]. Both projects are written in C and have some non-trivial C code features. While earlier works may have used different versions, we selected these projects because they include end-to-end tests, enabling us to evaluate the correctness of the translated code. To make the projects fit the input requirements, we use the *cpp* preprocessor to expand macros and combine the entire project into a single file. The *avl-tree* project contains 12 end-to-end tests to test the different functionalities of the AVL tree implementation. The *urlparser* project contains 3 end-to-end tests to test the different functionalities of the URL, we manually create 7 additional end-to-end tests to test the different functionalities of the URL parser.

[4] `https://github.com/xieqing/avl-tree`

[5] `https://github.com/jwerle/url.h`

I LLM Configurations

Table 4 shows our configurations for different LLMs in evaluation. All other hyper-parameters, like Top-P or Top-K, are set as the model's default values.

Table 4: Configurations of Different LLMs in Evaluation

Model	Version	Temperature	Hosting Platform
GPT-4o	gpt-4o-latest (As of 2024-12)	1	AzureOpenAI API
Claude 3.5 Sonnet	claude-3-5-sonnet-20241022	1	Anthropic API
Gemini 2.0 Flash	gemini-2.0-flash-exp	default	Google Cloud API
Llama 3.3 Instruct 70B	Llama 3.3 Instruct 70B Q4	0.8	4xH100 GPU
DeepSeek-R1	DeepSeek-R1 671B	1	DeepSeek API

J Failure Analysis in Evaluating SACTOR

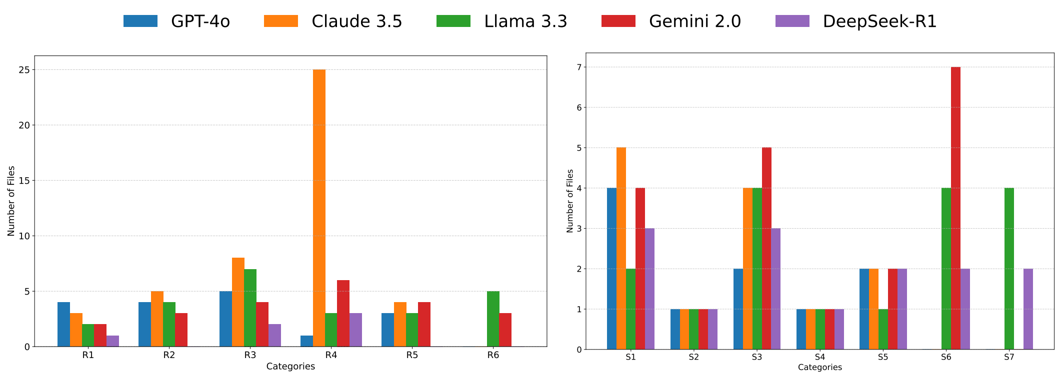
Table 5: Failure reason categories for translating TransCoder-IR and Project CodeNet datasets.

(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



(a) TransCoder-IR

(b) Project CodeNet

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

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 5 summarize the categories of failure reasons, and Figure 8a and 8b illustrate failure reasons (FRs) across models.

(1) TransCoder-IR (Table 5a, 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 5b, 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.

K SACTOR Cost Analysis

Table 6: Average Cost Comparison of Different LLMs Across Two Datasets. The color intensity represents the relative cost of each metric for each dataset.

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

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 6 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.

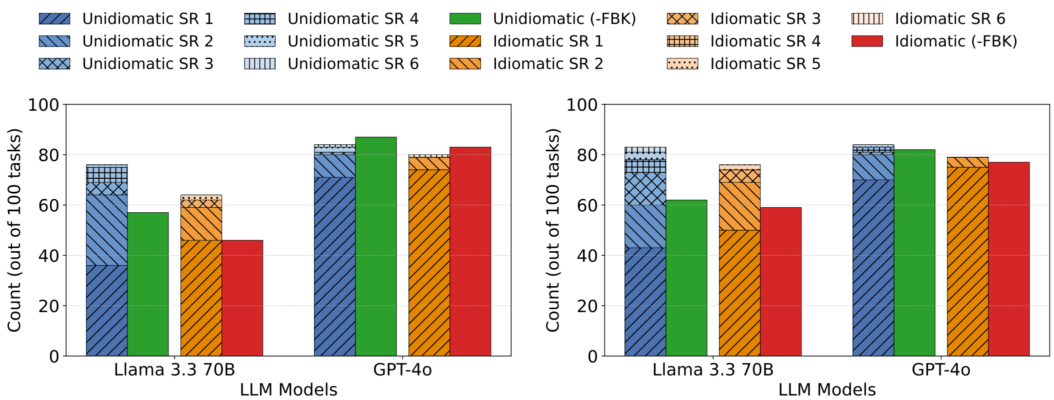
As a reasoning model, DeepSeek-R1 consumes significantly more tokens (17,895.52 vs. 13,592.61) than non-reasoning models–5-7 times higher than GPT-4o–despite having a similar average query count (4.77 vs. 3.11) for generating idiomatic Rust code. This high token usage comes from the "reasoning process"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

L Ablation Study on the Feedback Mechanism

To evaluate the effectiveness of the feedback mechanism proposed in § 4.3.3, 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.



(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.

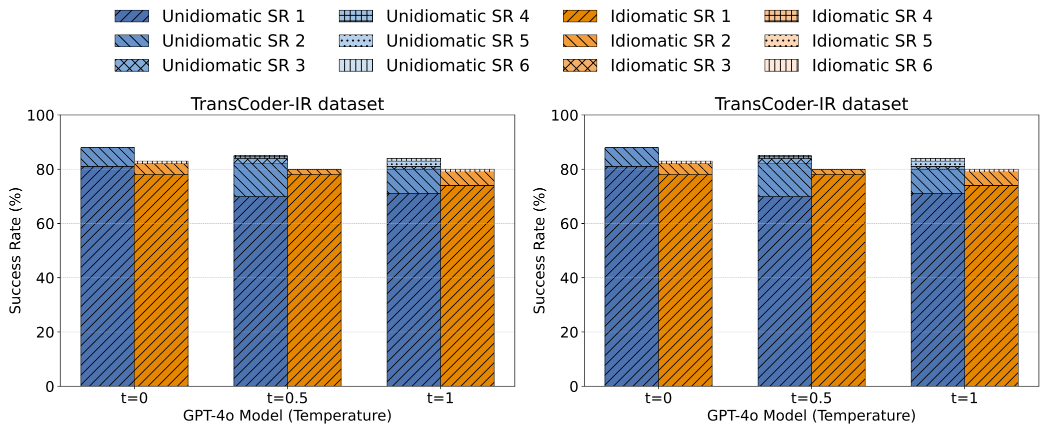
(1) TransCoder-IR (Figure 9a): Incorporating the feedback mechanism increased the number of successful translations for Llama 3.3 70B from 57 to 76 in the unidiomatic setting and from 46 to 64 in the idiomatic setting. In contrast, GPT-4o performed slightly worse with feedback, decreasing from 87 to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

M SACTOR Performance with Different Temperatures

In § 6, all the experiments are conducted with the temperature set to default values, as explained on Appendix I. 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 [6], 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.



(a) Success Rate on TransCoder-IR

(b) Success Rate on TransCoder-IR

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

(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.

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

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----
```

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.

```
This is the idiomatic translation of Rust code from C, the function signature is
```rust
{FUNCTION_NAME}_idiomatic
```
This is the unidiomatic translation of Rust code from C, the function signature is
```rust
{FUNCTION_DEPENDENCIES}
```
Generate the harness for the function atoi_idiomatic with the following code pattern so that
     it can be tested:
Finish all the TODOs.
You should **NOT** add any dummy implementation of the function or structs, as it will be
     provided by the verifier:
```rust
// TODO: add necessary `use`s here
// Don't add the definitions of any other functions and structs, they will be provided by the
     system

fn {FUNCTION_NAME} ({UNIDIOMATIC_ARGS}) -> {UNIDIOMATIC_RETURN} {
    // TODO: Add code here to Convert the input to the idiomatic format
    let result = fn {FUNCTION_NAME}_idiomatic({IDIOMATIC_ARGS}) -> {IDIOMATIC_RETURN}; // Call
      the idiomatic function
    // TODO: Add code here to Convert the result back to the original format
    //
}
```
remove all the TODOs and replace them with the necessary code.

Output the translated function into this format (wrap with the following tags):
----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

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and Section 1 clearly enumerate the tool's contributions in bullet list and match the empirical results reported later.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: We discuss the limitations of our work in Section A.

Guidelines:

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: The work is empirical; it presents no new theorems or formal proofs.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Source code and evaluation datasets are publicly released in the footnote of the paper.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Code and datasets with run instructions are linked in the paper.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).

- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: § 5 gives datasets, metrics, model versions, temperatures, and Appendix I lists all hyper-parameters.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: In Appendix M, we show the results under different temperature settings, and there is no significant difference between them. The results are also consistent across different datasets.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide the compute resources in Appendix I.

   Guidelines:

   - The answer NA means that the paper does not include experiments.

- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We reviewed the NeurIPS Code of Ethics and found no conflicts;

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: For positive societal impact, we have discussed in § 1 and § 7. This work is about code translation between two different languages and we don't expect any negative societal impact.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: We do not release any high-risk pretrained model.

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All the datasets and code used in this paper are publicly available and properly cited in the paper.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: We release SACTOR-datasets in the paper and provide the documentation in the footnote.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: No human subjects were involved in this research.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: No human subjects were involved in this research.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: We detail the usage of LLMs in § 5, and the LLMs are the core component of our method.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.