SafeTrans: LLM-assisted Transpilation from C to Rust

Muhammad Farrukh
Stony Brook University
New York, USA
mufarrukh@cs.stonybrook.edu

Baris Coskun*
Amazon Web Services
New York, USA
barisco@amazon.com

Tapti Palit
University of California, Davis
Davis, USA
tpalit@ucdavis.edu

Michalis Polychronakis
Stony Brook University
New York, USA
mikepo@cs.stonybrook.edu

Abstract

Rust is a strong contender for a memory-safe alternative to C as a "systems" language, but porting the vast amount of existing C code to Rust remains daunting. In this paper, we evaluate the potential of large language models (LLMs) to automate the transpilation of C code to idiomatic Rust. We present *SafeTrans*, a generic framework that leverages LLMs to i) transpile C code into Rust, and ii) iteratively repair compilation and runtime errors. A key novelty of our approach is a few-shot *guided repair* technique for translation errors, which provides contextual information and example code snippets for specific error types, guiding the LLM toward the correct solution. Another novel aspect of our work is the evaluation of the security implications of the transpilation process, showing how some vulnerability classes in C persist in the translated Rust code. SafeTrans was evaluated with six leading LLMs on 2,653 C programs and two real-world C projects. Our results show that iterative repair improves the rate of successful translations from 54% to 80% for the best-performing LLM (`gpt-4o`).

CCS Concepts

• **Security and privacy** → *Software security engineering*.

Keywords

Transpilation, Rust, LLM, Memory Safety, Vulnerability Mitigation

ACM Reference Format:
Muhammad Farrukh, Baris Coskun, Tapti Palit, and Michalis Polychronakis. 2026. SafeTrans: LLM-assisted Transpilation from C to Rust. In *1st Workshop on Code Translation, Transformation, and Modernization (ReCode '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3786180.3788317

1 Introduction

Many of the software systems used by enterprises and individual users rely on huge legacy C/C++ code bases. Migration to memory-safe languages will be a slow and tedious task if not (at least partially) automated. Rust is the strongest contender for a memory-safe "systems" language with acceptable runtime overhead, and major projects have started distributing some components written in Rust, or at least have introduced tooling support for integrating Rust code in existing C/C++ code bases (e.g., Firefox, Chrome, Linux, Windows). Automating the translation of *existing* C code into Rust, however, is challenging due to substantial syntactic and semantic disparities between the two languages, particularly concerning memory management and ownership.

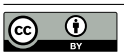
Initial attempts to this problem adopted rule-based translation approaches [8, 12, 14, 18, 44]. These methods scale to large programs and yield functionally equivalent code, but often fail to produce idiomatic and safe Rust, undermining their security benefits.

With the advent of large language models (LLMs), recent studies have focused on exploring their potential for automating code translation. LLMs can generate more idiomatic code than previous methods, but they come with their own set of challenges. Pan et al. [29] performed one of the first studies to understand the limitations of LLMs for code translation, and present a taxonomy of bugs introduced during the translation process. Similarly, Ou et al. [28] developed a repository-level benchmark for code translation evaluation, and developed a taxonomy of translation errors. LLM-transpiled code requires comprehensive methods to verify its correctness, an issue that recent studies attempt to address using various techniques [5, 9, 26, 41, 42]. Aside from correctness, LLMs also struggle with larger applications. Modern LLMs support large context windows, but recent studies show that they often cannot attend to all parts of long inputs uniformly, impacting their effectiveness [23, 24]. Several studies have tackled this issue by dividing larger applications into smaller translation tasks [27, 34, 35, 43]. Although these studies identify the causes of translation errors, how to use this knowledge to improve transpilation accuracy remains unclear.

Unlike the above recent works in automated transpilation using LLMs, which mostly focus on handling the translation of larger applications and developing additional approaches to verify the correctness of the translated programs, our research addresses several fundamental questions specific to C-to-Rust translation. First, we identify the Rust features that LLMs struggle to handle, which lead to the most frequent translation errors. We extend our analysis beyond building a simple taxonomy, by incorporating these findings to improve translation accuracy in the form of contextual information that aids the repair process.

Second, we evaluate the effectiveness of using this additional contextual information as part of a *guided* repair process to fix

*This work does not relate to Baris Coskun's position at Amazon.



ReCode '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2411-4/2026/04
https://doi.org/10.1145/3786180.3788317

faulty translations. This is achieved by analyzing the most frequent translation errors and assembling targeted instructions along with example code snippets to guide the LLMs towards the correct solution. Finally, we examine the effect of vulnerabilities present in C on the corresponding Rust translations. This targeted approach distinguishes our work from previous transpilation studies by specifically examining the language-level semantic challenges in translating from a memory-unsafe language to one with strict safety guarantees enforced at compilation time.

To this end, we developed *SafeTrans*, a framework for evaluating LLMs in their C-to-Rust code translation capabilities. We used SafeTrans to perform a total of 15,918 translations across 2,653 C programs and six LLMs. The combination of basic repair with few-shot guided repair achieves compilation repair success rates of up to 93.5% for `gpt-4o` and 89.8% for `DeepSeek-V3`. Guided repair effectively resolves challenging Rust compilation errors, such as trait implementation failures, with an average resolution rate of 58.7%, and borrow-checker violations with an average rate of 74.2% across all LLMs. Overall, our repair techniques collectively achieve substantial improvements in computational accuracy (CA), increasing the overall translation success rate from 54% to 80% for `gpt-4o`. Even smaller models, such as `Qwen2.5-Coder` and `DeepSeek-Coder`, nearly double their CA, highlighting the broad applicability and effectiveness of our approach.

In summary, we make the following main contributions:

- We present the design and implementation of SafeTrans, an end-to-end framework for comprehensively evaluating the C-to-Rust code transpilation capabilities of LLMs.
- We demonstrate that just providing compiler error messages and feedback is insufficient for repairing many types of faulty translations, and introduce a novel few-shot guided repair approach to improve the repair rate.
- We identify 10,375 vulnerabilities (e.g., out of bounds access, null-pointer dereference) in the 2,653 source C programs used in our evaluation, and demonstrate their behavior in the translated Rust programs.

Our prototype implementation and data set are publicly available through https://github.com/FarrukhCyber/SafeTrans.

2 Background and Related Work

Rule-based C-to-Rust Translation: Conventional solutions to source-to-source translation have predominantly relied on rule-based methodologies. The most prominent tool in this category is C2Rust [18], which translates C programs to Rust using both predefined and custom rules. Despite its scalability, C2Rust produces non-idiomatic code with excessive use of `unsafe` blocks.

A recent study of C2Rust by Emre et al. [8] investigates the underlying causes of unsafety. The authors propose a technique that relies on feedback from the `rustc` compiler to refactor a certain type of raw pointers into Rust references. Inspired by C2Rust, several studies have attempted to address its limitations. CROWN [44] improves upon C2Rust's output by converting raw pointers to references, but it is limited to mutable and non-array raw pointers. Similarly, other tools [11–13] focus on specific translation challenges, such as converting lock APIs and certain data types.

LLM-based C-to-Rust Translation: In recent years, LLMs have been used for a wide range of applications [6, 15, 19, 20, 25, 32, 40] including code generation. Pan et al. [29] conducted a comprehensive analysis of LLMs' translation capabilities across multiple language pairs, including C to Rust. Their study also provides a detailed taxonomy of LLM-based translation bugs but lacks in-depth insights into issues specific to C-to-Rust translation. For C-to-Rust translation, several studies enhance LLM-based approaches with program analysis, formal verification, and fuzzing to improve translation accuracy. VERT [41] uses WebAssembly to generate oracle Rust programs for formal equivalence testing. FLOURINE [9] verifies translated code via fuzzing, and SPECTRA [26] enhances translation using multimodal specifications. These tools, however, are limited to small programs (<600 LoC). Some recent works [5, 16, 27, 34, 35, 39] focus on translating large-scale C programs to Rust by decomposing the source code for incremental translation, while others employ LLM-powered agents for transpilation and testing [17, 22, 36] Finally, a user study by Li et al. [21] demonstrates that human strategies for C-to-Rust translation differ from those used by automated tools.

3 SafeTrans Overview

In this section, we present SafeTrans, our framework for automating C-to-Rust translation, along with the methodology used to evaluate translation fidelity. The overall translation process comprises four main steps, as illustrated in Figure 1: transpilation, compilation, repair, and validation. The original C program is first transpiled into Rust using an LLM. The resulting Rust program is then compiled, and if compilation fails, an iterative repair process is initiated to fix the errors using both generic and guided prompts. After successful compilation, the program is executed and validated against the original C program's test cases. In case of runtime or validation errors, another iterative repair cycle is initiated.

3.1 Transpilation from C to Rust

The first step in the translation pipeline is to generate an initial Rust version of the original C program using an LLM. The input program is included in the prompt, along with instructions for producing safe code and adhering to the required output format.

We use this initial prompt to query the LLM, which can result in two scenarios. Ideally, the LLM generates a response that adheres to the required output format, allowing us to extract the Rust code based on the predefined tags. In the second case, the LLM fails to generate a valid response, either because the prompt and the output exceed its context window, or because it produces incoherent output, without adhering to the required format. In such cases, we discard the response, a condition we refer to as a *Generation Error*.

3.2 Compilation and Repair:

In the compilation phase, the transpiled code is compiled using the `rustc` compiler. If the compilation is successful, the resulting binary is provided as input to the runtime testing and validation phase. Otherwise, SafeTrans attempts to automatically repair the compilation errors by initiating an iterative repair phase.

Basic Repair: In case of a compilation error, the `rustc` compiler provides detailed error messages which can be passed to the LLM along with the transpiled Rust program to provide additional

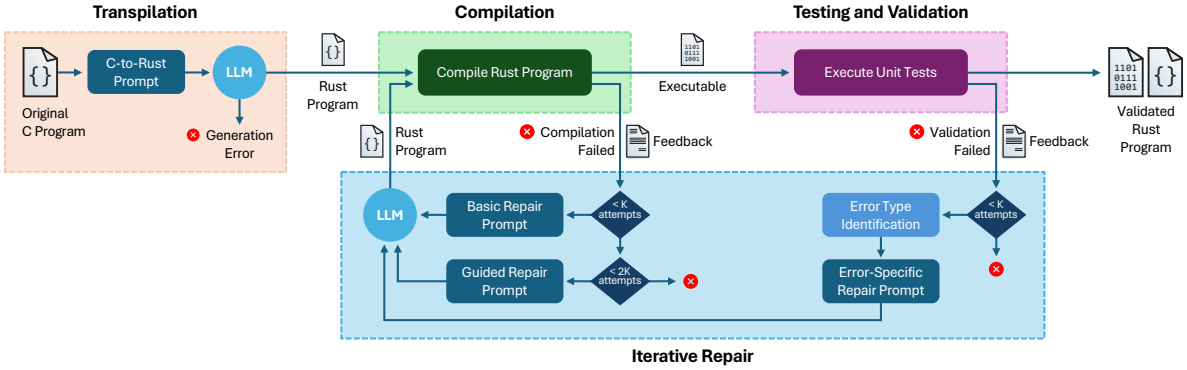


Figure 1: High-level architecture of SafeTrans' transpilation, compilation, repair, and validation pipeline.

context about the issue. Recent studies [7] have demonstrated the effectiveness of this approach. We adopt an iterative repair approach to resolve compilation errors similar to Pan et al. [29]. In each iteration, the incorrect Rust translation and the repair instructions within the prompt are updated based on the results of the previous iteration, while the base prompt remains unchanged. At the end of each iteration, the generated Rust program is compiled again, and if compilation fails, a new repair cycle is initiated. The phase continues until either the program is successfully compiled, or a predefined maximum number of iterations is reached (set to five in our experiments).

Guided Repair: Even after several iterations, some programs may still fail to compile. To improve the translation success rate, we introduce a new *guided repair* strategy, which uses few-shot learning by incorporating error-specific contextual information in the repair prompt. We selected the top eight most frequent errors, for which we developed tailored repair instructions, outlining key aspects to address along with common causes and fixes.

To maintain conciseness within the LLM's context length constraints and minimize noise, we only include instructions relevant to the errors present in the current compilation output. The instructions first explain the Rust property that was violated, leading to that error, followed by concrete examples of incorrect and corrected code snippets to assist the LLM in resolving the issue.

3.3 Runtime Testing and Validation

Successfully compiled Rust programs proceed to the runtime testing and validation phase, which executes the program with various test cases and validates the output against the expected results. We consider a C program as successfully translated if the generated Rust program passes all test cases. Erroneous outcomes of this dynamic analysis phase include *Runtime Error*, *Infinite Loop*, and *Test Case Error*. Our unit tests are based on the test cases available in the CodeNet data set used in our evaluation. If the translated Rust program fails to run properly or does not pass the test cases, it undergoes another round of iterative repair, this time with prompts tailored to the specific type of runtime error encountered.

In this phase, SafeTrans iteratively repairs the program until it passes all test cases or exceeds the maximum number of repair attempts (set to five in our tests). In each iteration, SafeTrans performs both compilation and runtime test checks, and if the validation fails, it queries the LLM with an appropriately updated (dynamically generated) prompt.

4 Experimental Setup

Large Language Model Selection: For our empirical study, we select a diverse set of LLMs, ranging from small open-source models to state-of-the-art (SOTA) LLMs. Among the SOTA LLMs, we include `gpt-4o` and `DeepSeek-V3` (671B). For small open-source LLMs, we select `Qwen2.5-Coder` (7B), `Codestral` (22B), `DeepSeek-Coder` (16B), and `Llama3` (70B). As a rapidly evolving field, LLMs are frequently updated, and new models are being released regularly. Due to time and cost reasons we could not include other recently released models, such as Claude and Gemini, but we tried to select a set of models that are representative of the spectrum of choices and capabilities in the current state of the art.

Data Set Collection and Pre-Processing: For code translation tasks, we use the CodeNet data set [30], and two real-world C libraries, `url_parser` [2] and `avl_tree` [1], which have been used in prior LLM-based translation studies. These two programs include comprehensive end-to-end test cases and represent challenging translation tasks. The CodeNet data set [30] comprises 4,053 competitive programming problems written in over 50 programming languages. Since our focus is on C, we filter the data set to retain only those problems that have an adequate number of solutions written in C.

After some preprocessing steps, our final evaluation data set contains 2,653 C programs. The test cases in our dataset achieve an average line coverage of approximately 87%.

5 Results

We present the results of our experimental evaluation, focusing on the following research questions: **RQ1:** How base LLMs perform on C-to-Rust translation task? **RQ2:** What compilation errors occur and how effective is iterative repair? **RQ3:** How effective is guided repair for persistent errors? **RQ4:** What is the overall improvement in translation success rate? **RQ5:** How does SafeTrans scale on complex real-world programs?

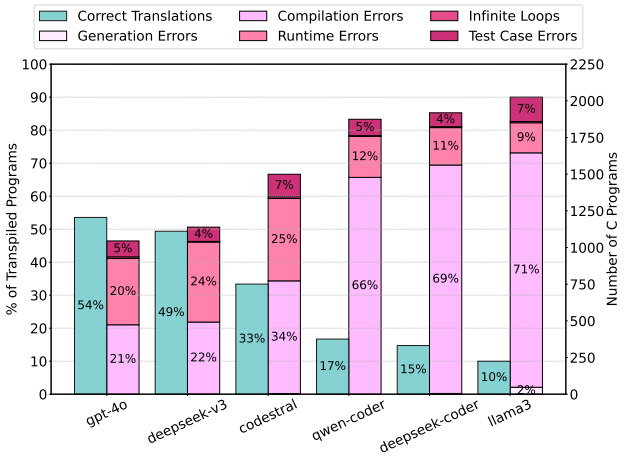


Figure 2: Percentage of correct Rust translations out of 2,653 C programs, and breakdown of the different types of failures for unsuccessful translations.

5.1 RQ1: Basic Translation Success Rate

To assess the "out-of-the-box" performance of LLMs in C to Rust translation, we adopt *computational accuracy (CA)*, proposed by Rozière et al. [31] and Szafraniec et al. [37], as our primary evaluation metric. CA is defined as the ratio of successfully translated programs to the total number of translation samples.

Figure 2 presents a comparative analysis of the six evaluated LLMs in their ability to perform basic C-to-Rust translation (without any attempt to fix any errors). For each model, the left bar in the group represents CA, while the right stacked bar illustrates a breakdown of the encountered translation errors. Following the categorization of Pan et al. [29], we classify these errors into five distinct types: ① *Generation Error*: The model either fails to generate a response according to required format, or the prompt exceeds the LLM's context length. ② *Compilation Error*: The transpiled Rust code fails to compile. ③ *Runtime Error*: The Rust code is compiled successfully, but the execution of the program fails (e.g., panics) ④ *Infinite Loop*: The program enters a non-terminating loop. ⑤ *Test Case Error*: Execution of at least one test case fails.

Among the evaluated models, `gpt-4o` and `DeepSeek-V3` achieve the highest base computational accuracy (54% and 49%, respectively). As evident from the stacked bars, compilation errors are the most frequent failure mode across all models.

5.2 RQ2: Compilation Error Analysis

5.2.1 Error Distribution. A high frequency of compilation errors demands an in-depth study of their root causes. The transpiled Rust programs result in a diverse range of errors, but we focus on the most frequent ones that comprise the vast majority of cases, since they represent the core features of Rust with which LLMs struggle. Additionally, we only consider the compilation errors for which `rustc` provides specific error codes, because this makes the categorization of errors easier for systematic analysis.

We observe consistent occurrence of some errors across all LLMs. Specifically, errors `E0277` (*"trait not implemented"*) and `E0308` (*"mismatched types"*) are the most prevalent, accounting for over 18% of all errors in most models, and up to 30% for `Qwen2.5-Coder` and `Llama3`, suggesting that LLMs struggle with type inference and trait bounds. Beyond these common errors, some models exhibit distinct error patterns. For example, 18.9% of `DeepSeek-V3` translations fail due to `E0428` (*"duplicate definition"*), while `Codestral` and `DeepSeek-Coder` struggle disproportionately (19–22%) with `E0599` (*"method not found"*). Similarly, `Qwen2.5-Coder` and `Llama3` result in high rates of *"lifetime issues"* and *"borrow conflicts."*

Table 1: Effectiveness of iterative repairing of failed Rust compilations.

LLM	Compilation Failures	Repaired	Repair Rate (%)	Pass Rate (%)
Qwen2.5-Coder	1743	1090	62.5	23.4
Llama3	1884	1096	58.1	14.1
DeepSeek-Coder	1836	1021	55.6	33.7
Codestral	906	778	85.8	28.9
GPT-4o	558	522	93.5	43.4
DeepSeek-V3	579	520	89.8	47.1

5.2.2 Iterative Compilation Repair. To evaluate the effectiveness of SafeTrans' repair phase, we introduce two metrics: i) *Repair Rate:* the percentage of transpiled programs that initially failed to compile, but then were successfully fixed during the repair phase, resulting in a compilable program; ii) *Pass Rate:* the percentage of repaired programs that run successfully and pass all test cases.

Table 1 provides a breakdown of the outcomes of the iterative compilation repair phase in terms of repair rate and pass rate for each LLM. The Compilation Failures column corresponds to the number of programs that failed to compile after the base translation. We observe that `gpt-4o` and `DeepSeek-V3` achieve high repair rates of 93.5% and 89.8%, respectively, outperforming the other LLMs. Interestingly, despite being significantly smaller, `Codestral` performs comparably to larger LLMs (only 7.7% and 4% lower than `gpt-4o` and `DeepSeek-V3` in terms of repair rate). The repair rate of `Llama3` (58.1%) is comparable to other smaller code-oriented LLMs, such as `Qwen2.5-Coder` (62.5%) and `DeepSeek-Coder` (55.6%), suggesting that it can effectively leverage repair prompts to fix faulty translations. The relatively lower pass rates across all LLMs compared to their repair rates suggest that while these models can interpret compiler feedback and fix syntactic issues, they often lose sight of the original program intent and functional equivalence.

5.3 RQ3: Effectiveness of Guided Repair

Even after iterative compilation repair, certain compilation errors remain. Many of the most frequent errors before repair continue to appear in abundance, which means that merely providing compiler feedback to the LLM is insufficient for resolving them. To better understand why LLMs struggle with these persistent errors, we selected the following ones (top-8) for further investigation: ① `E0277`: The type does not implement a required trait. ② `E0308`: Mismatched types encountered. ③ `E0425`: Use of an undeclared name or identifier. ④ `E0599`: Attempted call on a type that doesn't support it. ⑤ `E0384`: Cannot assign to an immutable variable. ⑥ `E0282`: Unable to infer enough type information. ⑦ `E0502`: Cannot borrow as mutable because it is also borrowed as immutable. ⑧ `E0499`: Cannot borrow as mutable more than once at a time.

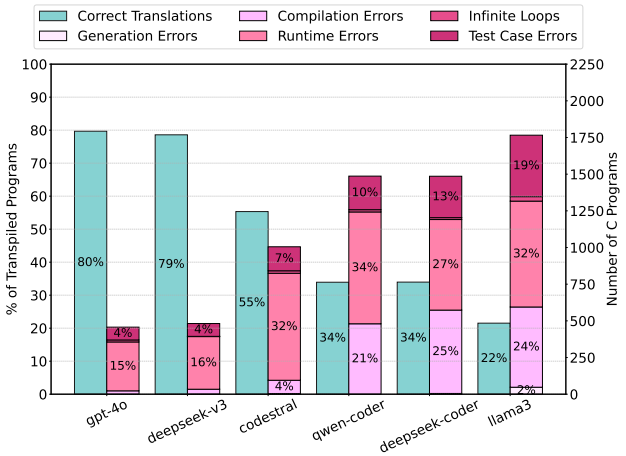


Figure 3: Final translation success rate and error breakdown for the full SafeTrans pipeline.

For each of these errors, we examine both their successful and failed repair cases to identify recurring patterns that lead to the error. Based on these observations, we develop guided instructions that describe the common causes and provide example fixes where applicable, which are used in our subsequent guided repair phase. For example, error `E0384` typically occurs when a new value is assigned to an immutable variable. Our analysis reveals that such common patterns include reassigning struct instances and variables introduced through pattern matching. The instructions in our guided repair are tailored to these patterns and provide examples to assist the LLM in resolving the error.

To evaluate the effectiveness of guided repair in resolving compilation errors, we define the *resolution rate* (RR) metric, which measures the percentage of targeted errors successfully repaired. Overall, guided repair proves highly effective across most error categories. `Llama3` achieves the highest average RR, resolving over 60% of common errors, followed closely by `DeepSeek-V3` and `Codestral`. Commonly encountered errors such as `E0384`, `E0282`, and `E0425` exhibit resolution rates above 80% across multiple models, showing that these issues can be fixed reliably when LLMs receive targeted feedback. Even challenging type inference errors like `E0277` ("trait not implemented") and `E0308` ("mismatched types") achieve an average resolution rate of approximately 59% and 55%, respectively, indicating that guided repair effectively improves model performance even on complex errors.

5.4 RQ4: Overall Translation Success Rate

After combining the basic and guided compilation repair approaches, in this section we evaluate the overall performance of the complete SafeTrans pipeline in successfully translating C programs into Rust. Figure 3 illustrates the substantial improvements achieved through the iterative repair techniques across all LLMs. The computational accuracy (CA) of both `gpt-4o` and `DeepSeek-V3` is increased by approximately 25% compared to their base CA, reaching 80% and 79%, respectively. Even for the underperforming models, CA is improved by roughly twice (+22% for `Codestral`, +17% for `Qwen2.5-Coder`, +19% for `DeepSeek-Coder` and +12% for `Llama3`).

For `gpt-4o` and `DeepSeek-V3`, we observe a drastic reduction across all error types, particularly in compilation errors, which drop below 2% (from 22–23% in Figure 2). This indicates that larger LLMs are highly effective at debugging erroneous programs when provided with appropriate feedback. Similarly, compilation errors for `Codestral`, `Qwen2.5-Coder`, `DeepSeek-Coder`, and `Llama3` drop by 30, 45, 44, and 47 percentage points, respectively. Among these, `Llama3` exhibits the largest reduction in compilation errors, indicating that even general-purpose LLMs can substantially benefit from our iterative repair strategies. For the smaller LLMs, we can see that there is an increase in non-compilation errors. This occurs because programs that previously failed to compile, once repaired, may generate new runtime or logic errors due to the hallucination tendencies of LLMs. However, the overall increase in CA shows the potential of targeted iterative repair techniques across models.

5.5 RQ5: Scalability on Complex Programs

To assess SafeTrans's scalability, we evaluated it on two real-world libraries, `url_parser` [2] and `avl_tree` [1], which have been used in prior studies [21, 27, 45]. While existing tools such as C2SaferRust, VERT, FLOURINE, and SACTOR either fail to compile these programs or produce translations with extensive unsafe code, SafeTrans successfully compiles both libraries while avoiding unsafe constructs. However, verification remains partial, as test-based feedback alone is insufficient to pinpoint semantic errors. A detailed comparison is provided in Appendix A.1 (Table 3).

6 Vulnerability Analysis

6.1 Identification of Potential Vulnerabilities

A major goal of our study is to assess the extent to which any vulnerabilities present in the original C code are effectively mitigated in the translated Rust code. Instead of planting bugs in existing programs or collecting a different data set of vulnerable programs, we observe that due to the nature of the CodeNet data set, its programs already contain numerous flaws that would pose security risks if they were to be used in production.

Inspired by the FormAI data set [38], we used the Efficient SMT-based Context-Bounded Model Checker (ESBMC) [10] formal verification tool to analyze the programs in our CodeNet data set and identify various types of vulnerabilities in them. ESBMC uses bounded model checking, which examines the correctness of a program by converting it into a finite state transition model and exploring possible states (up to a predefined boundary). It automatically verifies both predefined safety properties (e.g., out-of-bounds array access, illegal pointer dereferences, overflows) and user-defined program assertions. We should note that the flaws reported by ESBMC represent *potential* vulnerabilities—determining whether they are indeed exploitable is outside the scope of this work. For the sake of brevity, we refer to these flaws simply as *vulnerabilities* in the rest of this section.

The outcome of scanning a program with ESBMC can be categorized into one of the following three cases: i) *Verification Successful:* indicates that no flaws have been found within the defined bounds; ii) *Verification Failed:* ESBMC detected one or more flaws in the target program; iii) *Scan Error:* during verification, ESBMC may crash or timeout. Approximately 70% of the programs failed verification, i.e., they contain some form of flaw that was not reported by the authors of CodeNet. As pointed out by previous works [10, 38], ESBMC cannot produce false positives or false negatives, as each

Table 2: Distribution of the types of detected vulnerabilities in the original C programs, as provided by ESBMC [10].

Vulnerability Type	Instances
Buffer Overflow	3,258
Array Bounds Violated	2,951
Arithmetic Overflow	2,859
Dereference Failure: NULL Pointer	753
Division by Zero	196
VLA Array Size Overflows Address Space	175
Dereference Failure: Forgotten Memory	100
Dereference Failure: Invalid Pointer	47

identified issue is validated by counterexamples, and the fact that successful verification only occurs up to a predefined bound. This means that the possibility of some bugs hiding deep in the program still exists, but as we show, ESBMC still identifies plenty of potential vulnerabilities in the tested programs.

Table 2 shows the distribution of the most frequently reported types of vulnerabilities in the original C programs as reported by ESBMC. A single program can contain multiple vulnerabilities across multiple types. Overall, ESBMC identified a total of 10,375 vulnerabilities in the 2,653 programs (about five per program).

6.2 Vulnerability Mitigation

Despite explicit instructions to generate safe Rust code, all the evaluated models still produced very few translations containing some `unsafe` Rust code blocks (0.89% for `DeepSeek-Coder`, 1.25% for `Llama3`, 2.0% for `Qwen2.5-Coder`, 1.8% for `gpt-4o`, 3.4% for `Codestral`, and 4.4% for `DeepSeek-V3`). Moreover, in terms of average proportion of `unsafe` lines across all translated programs, the models exhibit roughly comparable behavior, with a mean unsafe ratio of approximately 34% overall.

For each identified flaw, ESBMC generates a "proof" of vulnerability in the form of program states that trigger the flaw. However, in some complex cases, these states do not show complete execution traces and therefore cannot be directly applied to the transpiled Rust programs. We employ a combination of automated and manual analysis to determine whether the same vulnerabilities in the original C programs can be triggered in the corresponding Rust translations. First, we used a combination of Rust memory-safety verification and undefined-behavior detection tools, including `Rudra` [4], `RAPx` [3], and `miri` [33]. None of these tools report any vulnerabilities in the transpiled Rust programs. Next, we perform manual analysis to identify any cases missed by automated checkers. Since scaling this manual analysis to all translations is not possible, we only focus on arithmetic overflow and null pointer dereference vulnerabilities.

Arithmetic Overflows: Rust provides built-in protections against arithmetic overflows in debug mode, but does not perform runtime checks in release mode by default. Although these checks can also be enabled in release mode, performing an overflow check for every arithmetic expression can introduce a significant performance overhead in computation-intensive programs. Rust allows developers to selectively use methods such as `checked_add` and `checked_sub` to handle overflows safely. Only 0.15% of correct translations across all models use these check functions. In our manual analysis of transpiled Rust programs corresponding to C programs with arithmetic overflow vulnerabilities, we found that such vulnerabilities often propagate to the Rust translations due to the absence of explicit `checked` functions.

Null Pointer Dereferences: We observe that the majority of the `unsafe` code in Rust translations arises from global variable usage rather than raw pointer usage (approximately 9.77% of all translated programs across models contain at least one raw pointer dereference). However, we still found cases where a null pointer dereference vulnerability in C propagates to the Rust translation due to incorrect handling of `unsafe` blocks. While translating a C implementation of B+ trees, `gpt-4o` introduces a bug by incorrectly translating a C for-loop to a Rust range expression which further leads to a segmentation fault.

```
// C: Shift elements right when inserting
for (i = leaf->num_keys; i > insertion_point; i--) {...}
```

```
// Rust: Empty range when insertion_point < num_keys
for i in ((*leaf).num_keys..insertion_point).rev() {...}
```

When inserting at position 0 with `num_keys = 1`, the range `(1..0).rev()` produces no iterations. Consequently, existing elements are not shifted, and `pointers[1]` remains uninitialized as `ptr::null_mut()`. The corruption occurs during range queries, which iterate over all keys and unconditionally dereference their associated pointers.

7 Limitations and Future Work

For our guided repair of compilation errors, we constructed contextual rules for only the top-eight errors we encountered, and incorporate them in the prompt during the repair process. As we have shown, this approach yields a high resolution rate for target errors. It is possible that after addressing the top errors, other compilation error types may prevent successful translation. Therefore, the set of custom instructions for guided repair prompts can be expanded to address additional errors, which would potentially increase further the success rate of compilation repair.

8 Conclusion

We presented SafeTrans, a framework that leverages LLMs to automate the transpilation of C code into Rust. Our approach combines basic repair and few-shot guided repair to address the inherent challenges in translating C code to *idiomatic* and *safe* Rust. Our extensive experimental evaluation results demonstrate that SafeTrans offers significant improvements in both computational accuracy (up to 25%) and compilation error repairing (up to 93.5%) compared to basic LLM transpilation. Our analysis of the security implications of the transpilation process highlights that LLM generated translations require further processing to mitigate source program vulnerabilities. We believe our findings highlight the potential of LLMs for automated transpilation to memory-safe languages and will encourage further research in this area.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was supported by the Office of Naval Research (ONR) through award N00014-24-1-2054, with additional support by Amazon through the Amazon Research Awards program.

References

[1] [n. d.]. An AVL Tree Implementation In C. https://github.com/xieqing/avl-tree/tree/master.

[2] [n. d.]. url.h. https://github.com/jwerle/url.h/tree/master.

[3] Artisan-Lab. 2025. RAPx: Rust Analysis Platform. https://github.com/Artisan-Lab/RAPx. https://github.com/Artisan-Lab/RAPx Accessed: 2025-05-13.

[4] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 84–99.

[5] Yubo Bai and Tapti Palit. 2025. RustAssure: Differential Symbolic Testing for LLM-Transpiled C-to-Rust Code. *arXiv preprint arXiv:2510.07604* (2025).

[6] Efe Bozkir, Süleyman Özdel, Ka Hei Carrie Lau, Mengdi Wang, Hong Gao, and Enkelejda Kasneci. 2024. Embedding large language models into extended reality: Opportunities and challenges for inclusion, engagement, and privacy. In *Proceedings of the 6th ACM Conference on Conversational User Interfaces*. 1–7.

[7] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing Rust compilation errors using LLMs. *arXiv preprint arXiv:2308.05177* (2023).

[8] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to Safer Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 121 (oct 2021).

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

[10] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. 2018. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 888–891.

[11] Jaemin Hong. 2023. Improving Automatic C-to-Rust Translation with Static Analysis. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. 273–277.

[12] Jaemin Hong and Sukyoung Ryu. 2023. Concrat: An automatic C-to-Rust lock API translator for concurrent programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 716–728.

[13] Jaemin Hong and Sukyoung Ryu. 2024. Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 716–740.

[14] Jaemin Hong and Sukyoung Ryu. 2024. To Tag, or Not to Tag: Translating C's Unions to Rust's Tagged Unions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 40–52.

[15] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. 2025. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces* 92 (2025), 103917. doi:10.1016/j.csi.2024.103917

[16] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2454–2476.

[17] Ali Reza Ibrahimzada, Brandon Paulsen, Reyhaneh Jabbarvand, Joey Dodds, and Daniel Kroening. 2025. MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair. *arXiv preprint arXiv:2509.16187* (2025).

[18] Immunant. 2022. C2Rust. https://github.com/immunant/c2rust.

[19] Hamed Jelodar, Mohammad Meymani, and Roozbeh Razavi-Far. 2025. Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets. *arXiv preprint arXiv:2503.17502* (2025).

[20] Yoonsang Kim, Zainab Aamir, Mithilesh Singh, Saeed Boorboor, Klaus Mueller, and Arie E. Kaufman. 2025. Explainable XR: Understanding User Behaviors of XR Environments using LLM-assisted Analytics Framework. *IEEE Transactions on Visualization and Computer Graphics* (2025).

[21] Ruishi Li, Bo Wang, Tianyu Li, Prateek Saxena, and Ashish Kundu. 2024. Translating C to Rust: Lessons from a user study. *arXiv preprint arXiv:2411.14174* (2024).

[22] Tianyu Li, Ruishi Li, Bo Wang, Brandon Paulsen, Umang Mathur, and Prateek Saxena. 2025. Adversarial Agent Collaboration for C to Rust Translation. *arXiv preprint arXiv:2510.03879* (2025).

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

[24] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (02 2024), 157–173. doi:10.1162/tacl_a_00638 arXiv:https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00638/2336043/tacl_a_00638.pdf

[25] Muhammad Muzammil, Abisheka Pitumpe, Xigao Li, Amir Rahmati, and Nick Nikiforakis. 2025. The Poorest Man in Babylon: A Longitudinal Study of Cryptocurrency Investment Scams. In *Proceedings of The Web Conference (WWW)*.

[26] Vikram Nitin, Rahul Krishna, and Baishakhi Ray. 2024. Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications. *arXiv preprint arXiv:2405.18574* (2024).

[27] Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. 2025. C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques. *arXiv preprint arXiv:2501.14257* (2025).

[28] Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xin Peng, and Zibin Zheng. 2024. Repository-level code translation benchmark targeting Rust. *arXiv preprint arXiv:2411.13990* (2024).

[29] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[30] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021).

[31] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 20601–20611.

[32] Ranjan Sapkota, Shaina Raza, Maged Shoman, Achyut Paudel, and Manoj Karkee. 2025. Image, Text, and Speech Data Augmentation using Multimodal LLMs for Deep Learning: A Survey. *arXiv preprint arXiv:2501.18648* (2025).

[33] Scott Olson. [n. d.]. Miri: an Interpreter for Rust's Mid-level Intermediate Representation. https://github.com/rust-lang/miri.

[34] Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A Seshia, and Koushik Sen. 2024. Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis. *arXiv preprint arXiv:2412.14234* (2024).

[35] Momoko Shiraishi and Takahiro Shinagawa. 2024. Context-aware Code Segmentation for C-to-Rust Translation using Large Language Models. *arXiv preprint arXiv:2409.10506* (2024).

[36] HoHyun Sim, Hyeonjoong Cho, Yeonghyeon Go, Zhoulai Fu, Ali Shokri, and Binoy Ravindran. 2025. Large Language Model-Powered Agent for C to Rust Code Translation. arXiv:2505.15858 [cs.PL] https://arxiv.org/abs/2505.15858

[37] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578* (2022).

[38] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C Cordeiro, and Vasileios Mavroeidis. 2023. The FormAI dataset: Generative AI in software security through the lens of formal verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*. 33–43.

[39] Chaofan Wang, Tingrui Yu, Chen Xie, Jie Wang, Dong Chen, Wenrui Zhang, Yuling Shi, Xiaodong Gu, and Beijun Shen. 2025. EvoC2Rust: A Skeleton-guided Framework for Project-Level C-to-Rust Translation. arXiv:2508.04295 [cs.SE] https://arxiv.org/abs/2508.04295

[40] HanXiang Xu, ShenAo Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and HaoYu Wang. 2024. Large language models for cyber security: A systematic literature review. *arXiv preprint arXiv:2405.04760* (2024).

[41] Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. VERT: Verified equivalent Rust transpilation with large language models as few-shot learners. *arXiv preprint arXiv:2404.18852* (2024).

[42] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1585–1608.

[43] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2024. Scalable, validated code translation of entire projects using large language models. *arXiv preprint arXiv:2412.08035* (2024).

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

[45] Tianyang Zhou, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. 2025. LLM-Driven Multi-step Translation from C to Rust using Static Analysis. *arXiv preprint arXiv:2503.12511* (2025).

A Appendix

A.1 Comparison with Prior C-to-Rust Translation Tools

Table 3 summarizes each tool's reported performance across three metrics: compilation success, complete code safety, and verification completeness. These include mostly prototype tools that require non-trivial setup, and in some cases manual intervention in appropriately formatting the source programs. Therefore, the presented results have been gathered directly from the respective publications describing each tool, rather than from independent experimentation.

Table 3: Comparison of C-to-Rust translation tools on the `url_parser` (UP) and `avl_tree` (AT) programs.

Tool	Model	Prog.	Comp.	Safe	Comments
C2SaferRust	gpt-4o-mini	UP	✓	×	Reports unsafe lines; no test coverage mentioned.
VERT	gpt-4	UP	×	–	Complete translation fails to compile.
	claude-2	AT	✓	–	Only subset of functions translated.
FLOURINE	claude-3.5	UP	×	–	Complete translation fails to compile.
SACTOR	gpt-4o	UP	✓	×	Partially idiomatic; fails verification.
	gpt-4o	AT	✓	×	Partially idiomatic; fails verification.
SafeTrans	gpt-4o	UP	✓	✓	Passes partial verification.
	gpt-4o	AT	✓	✓	Passes partial verification.