SafeTrans: LLM-assisted Transpilation from C to Rust

Muhammad Farrukh
Stony Brook University

Smeet Shah
Stony Brook University

Baris Coskun
Amazon Web Services

Michalis Polychronakis
Stony Brook University

ABSTRACT

Rust is a strong contender for a memory-safe alternative to C as a "systems" programming language, but porting the vast amount of existing C code to Rust is a daunting task. In this paper, we evaluate the potential of large language models (LLMs) to automate the transpilation of C code to idiomatic Rust, while ensuring that the generated code mitigates any memory-related vulnerabilities present in the original code. To that end, we present the design and implementation of *SafeTrans*, a framework that uses LLMs to i) transpile C code into Rust and ii) iteratively fix any compilation and runtime errors in the resulting code. A key novelty of our approach is the introduction of 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, i.e., whether potential vulnerabilities in the original C code have been properly addressed in the translated Rust code. We experimentally evaluated SafeTrans with six leading LLMs and a set of 2,653 C programs accompanied by comprehensive unit tests, which were used for validating the correctness of the translated code. Our results show that our iterative repair strategy improves the rate of successful translations from 54% to 80% for the best-performing LLM (GPT-4o), and that all types of identified vulnerabilities in the original C code are effectively mitigated in the translated Rust code.

CCS CONCEPTS

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

KEYWORDS

Transpilation, Rust, LLM, Memory Safety, Vulnerability Mitigation

1 INTRODUCTION

The exploitation of memory corruption vulnerabilities is among the leading causes of system compromise and malware infection. While there are several reasons behind the proliferation of exploitable bugs, the heavy reliance on C and C++, which do not guarantee memory or thread safety [1], plays a major role. Attempts to retrofit memory safety into unsafe languages like C/C++ face performance and compatibility challenges that have prevented their adoption [2, 3]. This suggests that *rewriting* existing code into memory-safe languages may be one of the most promising long-term strategies for addressing the problem of memory errors.

Modern memory-safe languages like Go and Rust offer compelling advantages as replacements for memory-unsafe languages. Their strict memory management, bounds checking, automatic memory allocation and deallocation, and other advanced features significantly reduce the risk of memory-related vulnerabilities. Despite the growing ecosystems of Go and Rust, operating systems, network services, and desktop software continue to be written mostly in memory-unsafe languages. Although encouraging progress is being made [4–8], the extensive developer familiarity with C/C++, the immense amount of existing code written in these languages, and their increased efficiency [9], impede large-scale software rewriting efforts.

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 [1, 10–13]. These techniques can scale to large programs and produce functionally equivalent code, but struggle with generating idiomatic code, which is harder to maintain. More importantly, the majority of the translated code is wrapped in `unsafe` blocks (Rust's way of allowing developers to bypass its safety guarantees), which defeats the purpose of translation from a security perspective. To address the limitations of rule-based methods, learning-based transpilers have been proposed, which convert code translation into a neural machine translation problem [14–17]. These techniques offer improvements over traditional program analysis methods, but bring new challenges, such as the high amount of resources required for training, and the scarcity of functionally equivalent program pairs in source and target languages like C and Rust, respectively.

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. [18] 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. [19] developed a repository-level benchmark (RustRepoTrans) for code translation evaluation, and developed a taxonomy of translation errors. LLM-transpiled code requires comprehensive methods to verify its correctness, so recent studies have addressed this concern using automated test case generation [20], fuzzing [21], multimodal specification [22], and formal verification [23]. Aside from correctness, LLMs also struggle with larger applications. Although

modern LLMs support large context windows, recent studies show that they often cannot attend to all parts of long inputs uniformly, which can impact performance on larger programs [24, 25]. Several studies have tackled this issue by dividing larger applications into smaller translation tasks [26–29]. Although these studies explain the causes of translation errors, it remains unclear how to leverage this knowledge to improve transpilation accuracy.

As a step towards bridging this gap and aiding the migration to memory-safe languages, in this paper we present *SafeTrans*, a framework for evaluating recent LLMs in their C-to-Rust code translation capabilities. SafeTrans uses LLMs to i) transpile C code into Rust and ii) iteratively fix any compilation and runtime errors in the resulting code. Successfully transpiled programs are then verified against the original C program's unit tests to ensure their functional correctness.

A key novelty of our approach is the use of an iterative basic repair phase, combined with an additional few-shot *guided repair* phase to improve the repair rate of translation errors. In the first repair phase, SafeTrans attempts to repair compilation errors by constructing prompts containing the faulty code and the corresponding compiler feedback. Any still unrepaired files undergo a second guided repair phase that adds contextual information about the specific errors encountered in the basic repair process. We analyzed the most frequent translation errors and assembled targeted instructions along with example code snippets to guide the LLMs towards the correct solution.

Another novel aspect of our work is the evaluation of the security implications of the transpilation process, i.e., whether potential vulnerabilities in the original C code have been properly addressed in the translated Rust code. Prior works have primarily focused on the correctness of the translated Rust programs, and do not investigate the security implications of the generated code [18, 19, 26, 29]. Our study investigates this aspect by identifying and categorizing various common types of vulnerabilities found in the original C programs, and analyzing whether these vulnerabilities have been neutralized in the translated Rust programs using inputs that trigger the respective bugs.

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., array out of bounds access, null-pointer dereference, use after free) in the 2,653 source C programs used in our evaluation, and demonstrate through experimental validation that these vulnerabilities are effectively neutralized in the translated Rust programs.

Our prototype and data set will be publicly available, and can be currently accessed anonymously through *this link*.

2 BACKGROUND AND RELATED WORK

Source-to-source code translation is a decades-old problem in the programming languages and software engineering communities [30], driven by the need to modernize applications, migrate legacy systems, and leverage the benefits of newer languages. Translation is achieved using a source-to-source compiler, also known as a *transcompiler* or *transpiler*, i.e., a program that converts between programming languages that operate at a similar level of abstraction [31]. Translating C programs to Rust has received significant attention due to Rust's memory safety and performance characteristics, which position it as a safer alternative to C.

2.1 Rule-based Code Translation

Conventional solutions to source-to-source translation have predominantly relied on rule-based methodologies. These approaches rely on static analysis to generate the abstract syntax tree and control flow graph of the code. Carefully crafted rules are then derived mostly manually to transcribe the source code into the target language. The development of such rule-based techniques is a tedious process that involves substantial manual human effort.

Existing solutions for automating the conversion of C code to Rust do not apply any of its borrowing and ownership features, resulting in overuse of Rust's `unsafe` keyword in the translated code. The most prominent tool in this category is C2Rust [10], which translates large-scale C programs to Rust using both predefined and custom rules. Despite its scalability, C2Rust produces non-idiomatic code with excessive use of `unsafe` blocks. Wrapping code in `unsafe` blocks bypasses Rust's safety checks, ultimately defeating the purpose of translating C programs to Rust (from a security perspective).

A recent study of C2Rust by Emre et al. [1] investigated the multiple 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 [11] 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 [12, 32, 33] focus on specific translation challenges, such as converting lock APIs and certain data types from C to Rust. CRustS [34] improves the translation of C2Rust further by auto-refactoring C2Rust output using code structure pattern matching and transformation without relying on compiler feedback. Rusty [35] is a preliminary study of a system for C-to-Rust code conversion via unstructured control specialization, which implements C-style syntactic sugar on top of Rust to eliminate the discrepancies between the two languages.

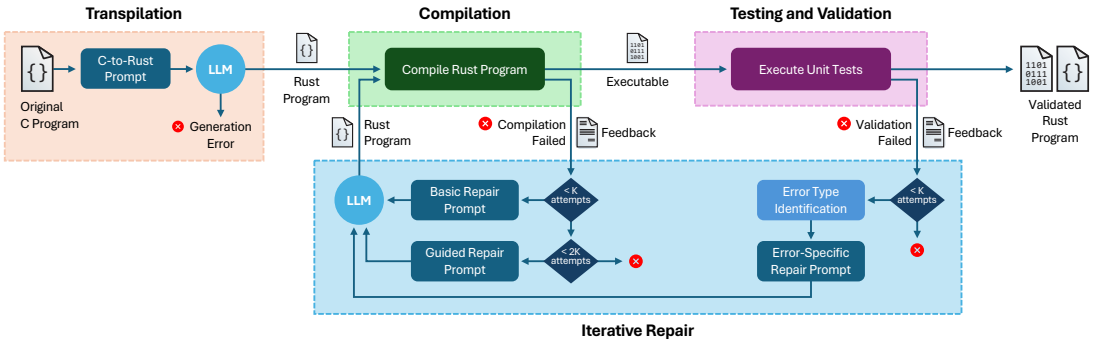

Figure 1: High-level architecture of SafeTrans' transpilation, compilation, repair, and validation pipeline.

2.2 LLM-based Code Translation

In a relatively short span, the fields of artificial intelligence and natural language processing have achieved impressive advances in generative AI, with a multitude of LLMs trained on various sources of data and for a wide range of applications [36–42]. In the field of computer programming, in particular, LLMs have shown great promise for code auto-completion, synthesizing code from natural language descriptions, summarizing and explaining existing code, and various other code-related tasks. By training models on immense code bases, such as source code files from public GitHub repositories, these code-specific LLMs can learn rich contextual representations that can be applied to various code-related tasks. This makes them an excellent tool for automated code translation [31].

Pan et al. [18] 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. Yang et al. [20] introduce UniTrans, a tool that generates unit tests using an LLM and iteratively incorporates feedback from these test cases to refine translation. However, UniTrans is limited to translating between C++, Java, and Python.

For C-to-Rust translation, several studies enhance LLM-based approaches with program analysis, formal verification, and fuzzing to improve translation accuracy. Yang et al. [23] propose VERT, which generates two Rust programs: an oracle Rust program lifted from source code using WebAssembly, and an LLM-translated Rust program from the same source. VERT then uses formal methods to rigorously test the translated Rust code against the oracle Rust program. Eniser et al. [21] developed FLOURINE, a framework that relies on fuzzing instead of test cases to verify the equivalence of translated Rust code. Similarly, Nitin et al. [22] introduce SPECTRA, which generates multi-modal program specifications from an LLM in a self-consistent manner, and provides them as input to the LLM along with the code to improve translation quality. However, these tools are only effective for C programs up to 600 LoC.

Some recent works focus on translating large-scale C programs to Rust. Nitin et al. [26] propose C2SaferRust, which leverages the C2Rust transpiler to generate an unsafe Rust version of an entire repository, acting as an intermediary to assist the LLM in translation. SYZYGY [27] also presents a method for translating entire repositories from C to Rust. Their approach segments the program into smaller translation units, uses an LLM to translate each unit, and verifies correctness through dynamic analysis and LLM-generated tests. Shiraishi et al. [28] adopt a similar methodology for handling large C programs, but focus solely on generating compilable Rust code without providing a means to verify functional correctness. Finally, a user study by Li et al. [43] demonstrates that human strategies for C-to-Rust translation differ from those used by automated tools, suggesting that future translators could benefit from incorporating human-like decision-making.

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 translated programs, our research addresses several fundamental questions specific to C-to-Rust translation. First, we identify the Rust features that LLMs struggle to comprehend, 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 concrete examples and contextual information that aid the repair process. Second, we evaluate the effectiveness of iterative basic and guided repair in rectifying faulty translations. Lastly, we examine which structural properties of Rust make programs resistant against common memory vulnerabilities present in C code. 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.

3 LLM-BASED CODE TRANSLATION

In this section we present the design of SafeTrans, our framework for automating the translation of C code into Rust, and the methodology we use to evaluate the fidelity of the translated code. 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 a large language model (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 phase of iterative repair is initiated.

Given some code written in the C programming language, translate it into equivalent Rust code that solves the exact same problem as the original code does. Ensure the following:
- Produce only safe Rust code.
- The translated Rust code can be compiled and executed with all the necessary imports.
- Output only the code without any additional explanation or comments.
- Wrap the code with ```rust
C code:
{C code}
Rust code:

Figure 2: Prompt template for C to Rust transpilation.

{Base prompt} // Base transpilation prompt from Figure 2

Rust code:
{Rust code} // Rust code with compilation errors

Executing your generated code gives the following errors because it is syntactically incorrect: {error messages}
Please suggest a corrected version of the complete code wrapped in ```rust

Figure 3: Prompt template for repairing compilation errors.

3.1 Transpilation from C to Rust

The first step in the translation pipeline is the initial attempt to generate a Rust translation of the original C program using an LLM. The input program is provided as part of the prompt, along with additional instructions. Figure 2 presents the format of our base transpilation prompt template.

The first instruction in the prompt ensures that the LLM produces safe Rust by avoiding the generation of `unsafe` code blocks in the translation. The second instruction helps to enforce the inclusion of all necessary dependencies in the resulting code. The last two instructions assist in extracting the translated code from the LLM's response. This is particularly useful for smaller open-source models, which often lack consistency in their output format. To mitigate this issue, we explicitly instruct the model to output only code, and wrap it within the defined tags.

We use this initial prompt to query the LLM, which can result in two scenarios. In the ideal case, the LLM generates a response that adheres to the required output format, allowing us to extract the Rust code based on the predefined tags. The extracted Rust code then advances through the pipeline. 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 descriptive content 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 Rust code from the previous phase is compiled using the `rustc` compiler. If the program is compiled successfully, the resulting binary is provided as input to the runtime testing and validation phase. Otherwise, SafeTrans attempts to automatically repair the compilation errors in the generated Rust code by initiating an iterative repair phase.

3.2.1 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 context about the issue. Recent studies [44] have demonstrated the effectiveness of this approach. Our repair prompt contains complete contextual information structured into three major parts, as illustrated in Figure 3. The base prompt section, which corresponds to the base translation template of Figure 2, helps the LLM retain contextual understanding and ensures that the repaired Rust code remains equivalent to the core functionality of the original C program. Next, we include the incorrect Rust translation generated by the LLM. Finally, we append the compilation error messages along with additional instructions about the required output format.

We adopt an iterative repair approach to resolve compilation errors, similar to Pan et al. [18]. 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).

3.2.2 Guided Repair. Even after several repair 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. The customized prompt includes specific guidance and concrete code examples tailored to the particular compilation errors encountered.

During our preliminary experiments, we performed an in-depth analysis of the frequency and distribution of compilation errors (based on their unique codes, as returned by `rustc`), and identified the most common translation errors (discussed in detail in Section 5). The error messages generated by `rustc` provide rich information about the root cause of the failure and potential solutions. 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. After the completion of the basic repair phase, if compilation still fails due to at least one of the most frequent errors, then SafeTrans initiates the guided repair phase, which is also performed iteratively (up to five times in our experiments).

The customized repair prompt is similar to the basic repair prompt (Figure 3), but is augmented with error-specific instructions and context. To maintain conciseness within the LLM's context

1. All variables are immutable by default. If the value of the variable needs to be changed, make sure you declare it as mutable using the 'mut' keyword. Example:

```
//Cause:
let x = 10;      // `x` is immutable by default
x = 20;          // Error: E0384, cannot assign to `x`
                 // because it is immutable
//Fix:
let mut x = 10; // `x` is now mutable
x = 20;          // This is allowed because `x` is mutable
```

2. Pattern-matched variables in match are immutable by default. Use mut in the pattern match to make the variable mutable in that scope. Example:

```
//Cause:
let opt = Some(10);
match opt {
    // Error: E0384, cannot assign to `x` because it is
    // immutable
    Some(x) => x += 1,
    None => (),
}
//Fix:
let opt = Some(10);
match opt {
    // Now `x` is mutable within this scope
    Some(mut x) => x += 1,
    None => (),
}
```

Figure 4: Example of guided repair instructions for error E0384 (*"Cannot assign to an immutable variable"*).

length constraints and minimize noise, we only include instructions relevant to the errors present in the current compilation output. Figure 4 shows an example of the error-specific instructions for error code E0384 (*"Cannot assign to an immutable variable"*) that are included in the prompt when this error is encountered. The instructions first explain the Rust property that the LLM-generated code violates, leading to error E0384. They then provide concrete examples of incorrect and corrected code snippets to help the LLM understand and resolve the issue effectively.

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.

First, we identify the type of failure, and based on the error type, we construct a dynamic prompt that incorporates the corresponding error feedback, as illustrated in Figure 5. We follow a structure similar to the compilation error repair prompts, but this phase

```
base prompt which contains original C code and output format
instructions
{base prompt}

Faulty Rust code
Rust code:
{rust code}

specific error messages according to error type
Executing your generated code gives the following {error type}
error:
{error message}
```

Figure 5: Prompt template used for dynamic repair.

Table 1: LLMs used in our experimental evaluation.

Model	Provider	Exact Version	Size	Context
GPT-4o	OpenAI	`gpt-4o`	N/A	128K
DeepSeek-V3	DeepSeek	`DeepSeek-V3`	671B	64K
Llama3	Meta	`llama-3-70b-Instruct`	70B	8K
DeepSeek-Coder	DeepSeek	`DeepSeek-Coder-V2-Lite-Instruct`	16B	128K
Qwen2.5-Coder	Alibaba	`qwen2.5-coder`	7B	128K
Codestral	Mistral AI	`Codestral-22B-v0.1`	22B	32K

handles multiple error types at the same time. While trying to repair the Rust program, it is possible for the LLM to introduce new compilation errors, which will lead to failed validation and the program will need to be repaired again. However, only basic repair will be used to repair intermediate compilation errors.

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

4.1 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`. For small open-source LLMs, we select `Qwen2.5-Coder`, `Codestral`, `DeepSeek-Coder`, and `Llama3`. 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. Table 1 provides a detailed overview of the selected LLMs. In our open-source LLM corpus, all the selected models are code-focused, except for `Llama3` and `DeepSeek-V3`, which serve as general-purpose models.

We access the SOTA LLMs through their respective public APIs. `Llama3` and `Codestral` are deployed on AWS (Amazon Web Services) Bedrock, while `Qwen2.5-Coder` and `DeepSeek-Coder` run on an `ml.g5.12xlarge` instance of AWS SageMaker AI.

4.2 LLM Hyperparameters

During the initial translation process, we configure the *temperature* (a hyperparameter that controls the randomness of output) to 0.2 for all models. We select a lower temperature value for base translation to keep the generated Rust translations deterministic. However, when repairing problematic translations, we increase this value to 0.6 to allow for more creative fixes. All other generation parameters, including *top-k* and *top-p*, which limit token selection to the most probable candidates based on fixed count and cumulative probability, respectively, are set to standard values as recommended by the developers of each language model.

4.3 Data Set Collection and Pre-Processing

For code translation tasks, prior studies have commonly utilized data sets such as CodeNet [45], AVATAR [46], and EvalPlus [47]. Our empirical study specifically targets the translation of C programs to Rust, which necessitates a data set with a substantial number of C programs. Having some strong evidence about the validity of the translation is also critical, and can be accomplished by respective test case inputs and outputs for runtime verification of functional correctness. Based on these requirements, CodeNet is the most suitable choice.

The CodeNet data set [45] comprises 4,053 competitive programming problems written in over 50 programming languages, with approximately 13 million code submissions. Each problem includes multiple solutions across different 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. We further eliminate duplicates and randomly sample one solution per problem. Additionally, we ensure that all selected problems have a comprehensive set of verified test cases.

Upon closer inspection, we observe that some solutions contain additional utility functions that are not called anywhere in the program. These "dead" functions may affect the accuracy of the translation, and therefore we remove them using the static analysis tool `tree-sitter` [48]. After completing all above filtering and preprocessing steps, we end up with a final set of 2,653 C programs, which comprise our evaluation data set.

To assess the quality of test cases accompanying the selected programs, we use the `gcov` tool [49], to measure the line coverage (defined as the ratio of executed lines to the total number of executable lines in a program) and function coverage (defined as the ratio of executed functions to the total number of executable functions in a program) for each test case. Higher line and function coverage typically indicates more comprehensive testing. However, it is important to note that high line and function coverages alone do not guarantee the detection of all potential defects. Figure 7 presents the distribution of line and function coverage rates across all selected C programs. A significant proportion of programs in the data set have comprehensive line coverage, with an average of 91.25% and 97.50% functional coverage. Notably, only approximately 2.27% of the programs have line coverage below 50%.

To understand the structural characteristics of the C programs in our evaluation data set, we again use `tree-sitter` to parse them and extract various code-related metrics, such as number of functions, number of pointers, number of structs, number of memory functions (e.g., `malloc`, `calloc`, and `free`), and lines of code (LoC). Figure 6 shows the cumulative distribution function of each of these metrics.

In terms of lines of code, the 80th percentile reaches approximately 100 LoC, indicating that the majority of programs in the data set are compact, except few programs larger than 200 LoC. In the number of functions distribution 80th percentile falls at approximately 5 functions, demonstrating that most programs in the data set employ a limited functional decomposition approach. The pointer usage distribution indicates that while a majority portion of programs use few or no pointers, there is a long tail of programs with more complex memory management needs, indicated by the gradually increasing curve beyond the 80th percentile.

Similarly, the distribution of struct declarations shows minimal use of complex data structures, approximately 90% of programs contain fewer than five `struct` definitions. In terms of dynamic memory allocation, over 90% of programs make fewer than ten calls to functions such as `malloc` or `free`, indicating that heap memory management is not widely employed.

These distributions suggest that while the majority of programs in the CodeNet data set are structurally simple, there remains a subset of larger and more complex programs (e.g., those exceeding 200 LoC). As the experimental evaluation results show (Section 5.4), these more complex programs pose additional challenges for automated translation.

5 RESULTS

In this section we present the results of our experimental evaluation, focusing on the following main research questions:

- **RQ1: Effectiveness of basic LLM-based C to Rust translation.** We evaluate in detail how recent LLMs perform on the task of translating C programs into Rust, and analyze the different failure conditions that are encountered.
- **RQ2: Analysis of compilation errors.** We explore the different types of compilation errors encountered in the transpiled Rust programs, and evaluate the effectiveness of simple iterative repair.
- **RQ3: Effectiveness of guided repair.** We investigate how compilation errors evolve and transition during repairing iterations, and evaluate the effectiveness of our guided repair strategy in fixing these errors.
- **RQ4: Improvement in overall successful translation rate.** We evaluate how our iterative repair strategies help in increasing the number of successfully translated programs.

5.1 RQ1: Basic Translation Success Rate

To assess the "out-of-the-box" performance of large language models (LLMs) in C to Rust translation, we adopt *computational accuracy (CA)*, proposed by Rozière et al. [31] and Szafraniec et al. [17], as our primary evaluation metric. CA is defined as the ratio of successfully

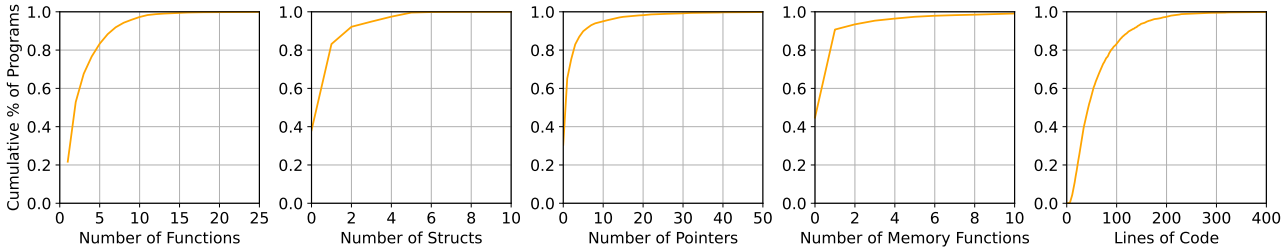

Figure 6: Cumulative distribution functions (CDFs) of structural code metrics for the C programs in the CodeNet data set.

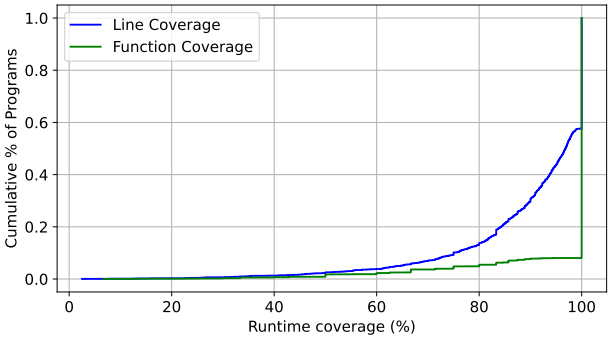

Figure 7: Runtime code coverage of the CodeNet test cases in terms of executed lines of code and functions.

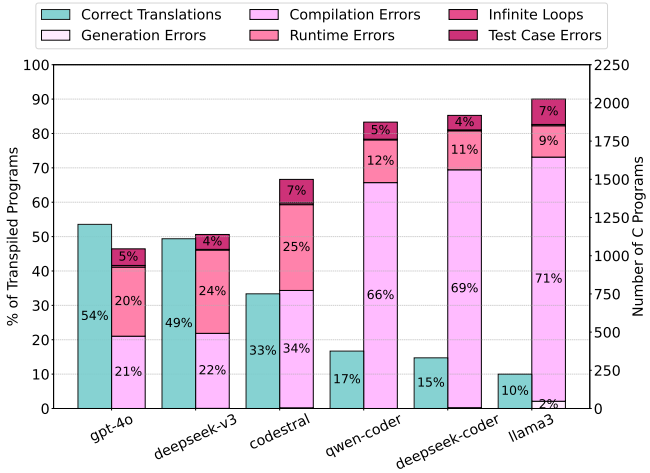

Figure 8: Percentage of correct Rust translations out of 2,653 C programs, and breakdown of the different types of failures for unsuccessful translations.

translated programs to the total number of translation samples. We prioritize CA over static evaluation metrics such as exact match, syntax match, and dataflow match [50], because CA evaluates the functional equivalence of translated programs by executing them with similar inputs. LLMs can achieve high scores on static metrics while demonstrating poor performance in computational accuracy, thereby revealing the limitations of such metrics in programming language translation tasks.

Figure 8 presents a comparative analysis of the six evaluated LLMs in their ability to perform base-level C-to-Rust translation (without any attempt to fix any errors). For each model, the left bar in the group represents the CA, while the right stacked bar illustrates a breakdown of the encountered translation errors. Following the categorization of Pan et al. [18], we classify these errors into five distinct types:

- *Generation Error*: The model either fails to generate a response according to required format, or the prompt exceeds the LLM's context length.
- *Compilation Error*: The transpiled Rust code fails to compile.
- *Runtime Error*: The Rust code is compiled successfully, but the execution of the program fails (e.g., panics).
- *Infinite Loop*: The program enters a non-terminating loop.
- *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). Although `DeepSeek-V3` is an open-source model, DeepSeek provides API access to its largest variant at a cost approximately 25 times lower than `gpt-4o`, while delivering comparable performance. In contrast to larger LLMs, smaller open-source models perform significantly worse, with significantly higher error rates.

On average, the smaller open-source code models in our study demonstrate over twice the CA of `Llama3`. Despite its 70B parameter size and substantial resource demands, `Llama3`'s general-purpose design results in limited performance on code-specific tasks, underscoring the importance of domain-specific training for programming language translation. Interestingly, despite having only seven billion parameters, `Qwen2.5-Coder` achieves performance on par with much larger competitors.

As evident from the stacked bars, compilation errors are the most frequent failure mode across all models. This highlights that syntactic correctness is a key challenge in LLM-based translation. The particularly high prevalence of compilation errors in smaller open-source models further emphasizes their limited understanding of Rust's syntax and compilation rules. For `gpt-4o` and `DeepSeek-V3`, runtime errors are equally frequent as compilation errors, which motivated us to also explore iterative repair techniques tailored to runtime errors.

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

Table 2: 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

programs result in a diverse range of compilation 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.

The heatmap of Figure 9 illustrates the distribution of the union of the top-10 most frequent Rust compilation errors per LLM encountered in the transpiled programs. Cells with darker color correspond to high relative contribution to the total number of compilation errors for a given LLM. A first observation is the 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`. This trend suggests that LLMs struggle with type inference and trait bounds when generating Rust code. We provide a detailed description of the compilation error codes in Appendix A.

We also observe model-specific patterns. For example, 18.9% of `DeepSeek-V3` translations fail due to E0428 (*"duplicate definition"*), while `Codestral` and `DeepSeek-Coder` struggle disproportionately (19.0-22.2%) with E0599 (*"method not found"*). Similarly, `Qwen2.5-Coder` and `Llama3` result in high rates of E0499 (*"lifetime issues"*) and E0502 (*"borrow conflicts"*), hinting at weaker handling of Rust's memory-safety constraints. The relatively lower variance in errors such as E0061 (*"invalid number of arguments in function call'*) implies that LLMs have captured a decent model of basic programming language constructs.

5.2.2 Iterative Compilation Repairing. SafeTrans employs an iterative compilation repair strategy to address compilation errors, as described in Section 3. To evaluate the effectiveness of this phase, we introduce two metrics:

- *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.
- *Pass Rate:* The percentage of repaired programs that run successfully and pass all test cases.

Table 2 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.

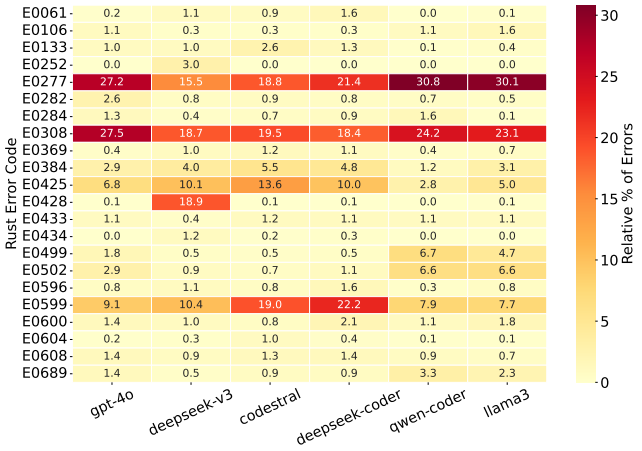

Figure 9: Distribution of the most common compilation errors for the transpiled Rust programs.

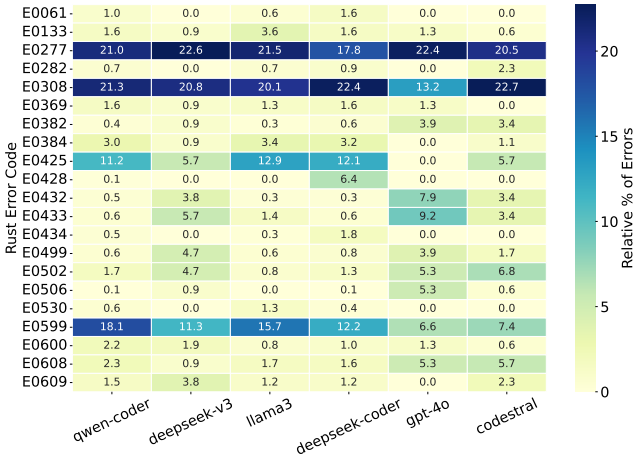

Figure 10: Distribution of the compilation errors that could not be fixed by the compilation repair phase.

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), demonstrating its strong ability to understand and benefit from compiler feedback. 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. As noted by Pan et al. [18], LLMs may introduce new errors while resolving existing ones, requiring multiple iterative passes to achieve fully correct and functional translations.

Table 3: Resolution rates (RR) for each error code across the six LLMs using Guided Compilation Repair. Each cell shows `RR% (remaining error count/initial error count)`.

Error Code	Codestral	DeepSeek-Coder	Qwen-Coder	LLaMA3	DeepSeek-V3	GPT-4o
E0277	48.57 (18/35)	34.80 (178/273)	43.11 (194/341)	64.73 (85/241)	79.17 (5/24)	82.35 (3/17)
E0308	60.53 (15/38)	34.20 (227/345)	31.01 (238/345)	67.41 (73/224)	81.82 (4/22)	50.00 (5/10)
E0425	90.00 (1/10)	56.68 (81/187)	49.44 (91/180)	89.36 (15/141)	83.33 (1/6)	0.00 (0/0)
E0599	69.23 (4/13)	67.20 (61/186)	70.10 (87/291)	83.80 (29/179)	75.00 (3/12)	80.00 (1/5)
E0384	100.00 (0/2)	77.55 (11/49)	46.94 (26/49)	74.29 (9/35)	100.00 (0/1)	0.00 (0/0)
E0282	100.00 (0/4)	66.67 (5/15)	81.82 (2/11)	100.00 (0/8)	0.00 (0/0)	0.00 (0/0)
E0502	41.67 (7/12)	10.53 (17/19)	40.74 (16/27)	55.56 (4/9)	60.00 (2/5)	75.00 (1/4)
E0499	33.33 (2/3)	25.00 (9/12)	22.22 (7/9)	83.33 (1/6)	60.00 (2/5)	100.00 (0/3)

5.3 RQ3: Effectiveness of Guided Repair

Even after iterative compilation repairing, certain compilation errors remain unresolved. Figure 10 shows the distribution of error types that persist after the completion of the compilation repair phase. It is evident that many of the most frequent errors before repairing 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:

- E0277: The type does not implement a required trait.
- E0308: Mismatched types encountered.
- E0425: Use of an undeclared name or identifier.
- E0599: Attempted call on a type that doesn't support it.
- E0384: Cannot assign to an immutable variable.
- E0282: Unable to infer enough type information.
- E0502: Cannot borrow as mutable because it is also borrowed as immutable.
- E0499: Cannot borrow as mutable more than once at a time.

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.

As an example, Figure 4 shows the guided instructions developed for error `E0384`. This error typically occurs when a new value is assigned to an immutable variable. Our analysis reveals that common patterns include reassigning struct instances and variables introduced through pattern matching. The instructions in Figure 4 are tailored to these patterns and provide examples to assist the LLM in resolving the error.

Table 3 summarizes the results on the effectiveness of guided repair in resolving the compilation errors remaining after the initial basic repair phase. To accurately evaluate its effectiveness, we introduce the *resolution rate* (RR) metric, which measures the percentage of targeted errors that are successfully repaired—instead of simply indicating whether a file is fixed or not. An RR of zero signifies that the target error was not present in the test set. Notably, `Llama3`, despite being a general-purpose LLM, achieves high resolution rates across nearly all targeted errors, suggesting that guided repair significantly enhances its ability to acquire the knowledge needed for effective correction. Errors such as `E0384`, `E0282`, and `E0425` consistently show high resolution rates across multiple models, indicating that these error types can be fixed easily when LLMs are provided with sufficient context and targeted feedback.

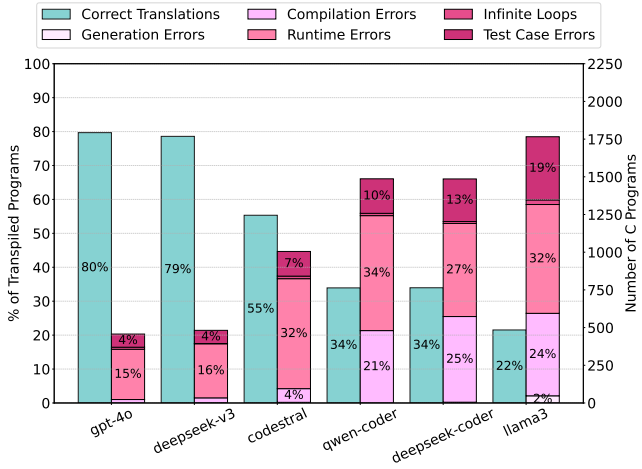

Figure 11: Final translation success rate and error breakdown for the full SafeTrans pipeline.

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 11 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`). Among the smaller open source code-LLMs, `Qwen2.5-Coder` and `DeepSeek-Coder` achieve the same CA performance (34%), despite `Qwen2.5-Coder` being smaller (7B) than `DeepSeek-Coder` (16B).

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 8). This indicates that larger LLMs are highly effective at debugging erroneous programs when provided with appropriate compiler 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` shows the largest reduction in compilation errors, indicating that even a general-purpose LLM can substantially benefit from our iterative repair strategies to enhance its code understanding capabilities. 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 iterative repair techniques, especially when coupled with targeted feedback, in improving translation success rate across diverse models.

The structural characteristics of source programs, such as lines of code (LoC), number of pointers, and number of functions, can significantly influence the translation outcome. Figure 12 presents an overview of how LOC and the number of pointers (#ptr) in a program affect the translation success rate. Regarding LoC, all LLMs exhibit steep cumulative distribution function (CDF) curves for correct translations, i.e., they can easily handle the vast majority of programs with up to 100 LoC. Notably, `gpt-4o`, `DeepSeek-V3`, and `Codestral` are able to generate correct translations for larger programs (i.e., those exceeding 100 LoC), whereas `Qwen2.5-Coder`, `DeepSeek-Coder`, and `Llama3` struggle with such programs. The majority of compilation errors occur in programs with more than 100 LoC. As the program size increases, LLMs face greater difficulty in maintaining syntactic correctness. A similar trend is seen in runtime and test case errors.

The pointer-based analysis in the bottom graphs further substantiates these findings, showing that translation success rates decline as pointer usage increases—a pattern consistently observed across all models. Notably, `DeepSeek-V3`'s slower-rising curve shows that it produces fewer compilation errors for pointer-heavy code. Overall, these results highlight that although recent LLMs achieve impressive code translation capabilities, their performance remains influenced by structural code characteristics such as code size and complexity.

6 VULNERABILTY MITIGATION

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 [51], we used the Efficient SMT-based Context-Bounded Model Checker (ESBMC) [52] formal verification tool to analyze the programs in our CodeNet data set and identify various types of vulnerabilities in them, such as illegal memory accesses and integer overflows. 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). ESBMC is an open-source tool that supports multiple programming languages, including C. 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 constitute *potential* vulnerabilities—determining whether they are indeed exploitable is outside the scope of this work. For the sake of brevity, we refer to them simply as *vulnerabilities* in the rest of this section, as the majority of these flaws are indeed critical (as discussed in Section 6.2).

Table 4: Categorization of the outcome of running ESBMC [52] verification on the C programs in our data set.

Type	Frequency	Percentage (%)
Verification Failed	1906	71.8
Verification Successful	332	12.5
Scan Error	415	15.6
Total	2653	100.0

The outcome of scanning a program with ESBMC can be categorized into one of the following three cases:

- *Verification Successful:* Indicates that no flaws have been found within the defined bounds.
- *Verification Failed:* ESBMC detected one or more flaws in the target program.
- *Scan Error:* During verification, ESBMC may crash or timeout. In such cases, we mark the file as having a Scan Error.

Table 4 shows the breakdown of ESBMC verification results on our evaluation data set. Approximately 70% of the programs contain some form of flaw that was not reported by the authors of CodeNet. We employ the same ESBMC configurations suggested by Tihanyi et al. [51], as our focus is on the behavior of potential vulnerabilities that are mitigated as the code is transpiled from C to Rust, rather than reporting all possible bugs in a C program. As pointed out by previous works [51, 52], ESBMC cannot produce false positives or false negatives, as each 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 5 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. Since our evaluation data set comprises competitive style programs that mostly involve data supplied through `stdin` or simple files, they commonly use simple `scanf()` calls, arrays, and use of arithmetic operations, and we thus observe many vulnerabilities associated to these operations. Overall, ESBMC identified a total of 10,375 vulnerabilities in the 2,653 C programs (~5 per program on average).

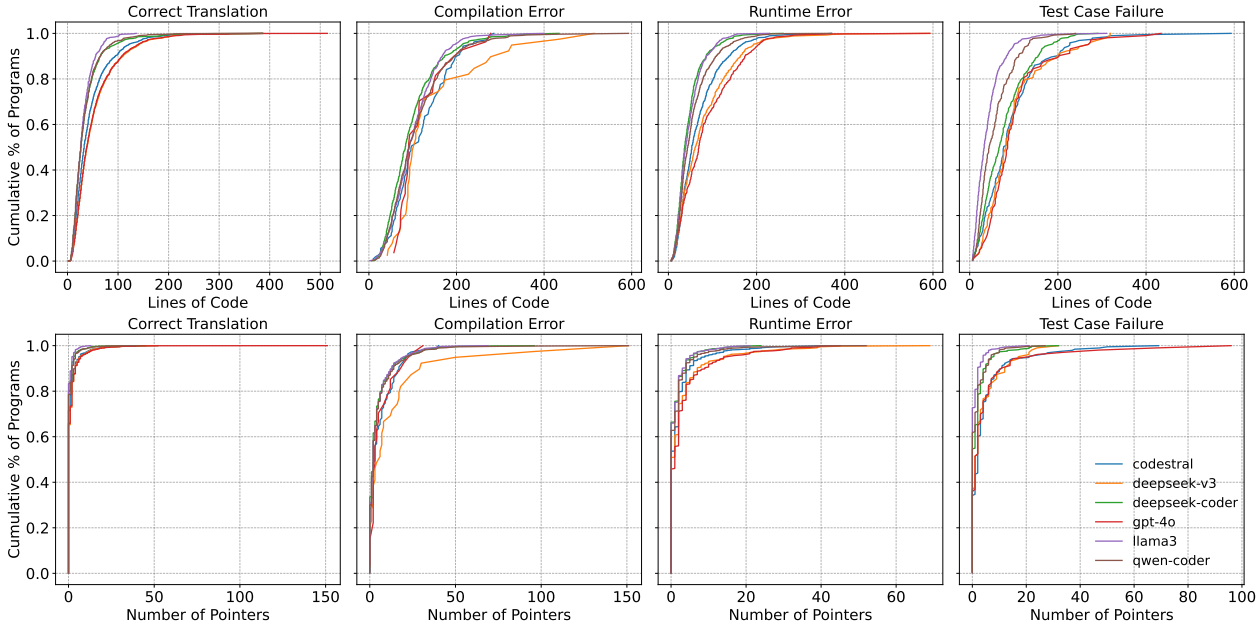

Figure 12: CDFs of code complexity metrics—lines of code (top) and number of pointers (bottom)—for successful and unsuccessful translation outcomes, as a percentage of the 2,653 C programs in our data set. As the complexity of the code increases, the rate of failed translations also increases.

Table 5: Distribution of the types of detected vulnerabilities in the original C programs, as provided by ESBMC [52].

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
Dereference Failure: Invalidated Dynamic Object	12
Dereference Failure: Invalid Pointer Freed	5
Dereference Failure: Misaligned Access to Data Object	4

6.2 Vulnerability Mitigation

For each identified flaw, ESBMC generates a "proof" of vulnerability in the form of an input that when fed into the program triggers the flaw. We first use these inputs to test the original C programs and verify that the identified vulnerabilities can indeed be triggered. We then run the corresponding successfully transpiled Rust programs with exactly the same inputs, and observe how they behave. In the following, we provide a detailed analysis of the most common types of potential vulnerabilities, and how they are mitigated in the translated Rust code.

6.2.1 Array Bounds Violation and Variable-Length Array Overflow. In the C implementations, the flexibility of using signed integers for array indexing and variable-length array (VLA) sizes introduces conditions of out-of-bounds array accesses and overflows of the array size beyond the available address space, potentially leading to undefined behavior or memory corruption. Rust mitigates these issues by preventing negative values from being used as array indices and sizes, enforcing the use of the unsigned type `usize`. Furthermore, Rust's dynamic memory allocation through heap-allocated vectors (`Vec<T>`) includes built-in bounds checking. In the following example code snippet from one of the test programs, providing the number -192 as input to the integer variable `n` causes the C program to access out-of-bounds memory. In the equivalent transpiled Rust code, the same negative input results in a safe panic rather than undefined behavior.

```c
// Original C program
int a[100010];
void f(int l, int r, int k) {
  ...
  if (a[r - 1] == k) {  // Array out-of-bounds access
    ...
  }
}
int main() {
  int n;
  scanf("%d", &n);
  ...
  f(0, n, 1);
  ...
}
```

```rust
// Translated Rust code
fn f(a: &mut [i32], l: usize, r: usize, k: i32) {
    ...
    if a[r - 1] == k {
        ...
    }
    ...
}
```

```rust
fn main() {
    ...
    let n: usize = input.trim().parse().unwrap();
    ...
    f(&mut a, 0, n, 1);
}
```

6.2.2 Arithmetic Overflow. In C, arithmetic operations can easily overflow (or underflow) if not explicitly handled by the programmer, as the language does not automatically check whether an arithmetic result exceeds the limits of the data type used. Rust, in contrast, emphasizes safety and robust error handling by utilizing stricter type systems and runtime checks. In the example below, if a user provides -2147483648 (`INT_MIN`) to the C program's variable `m`, the subtraction by 1 from `m` in the loop condition leads to a silent overflow, and the value wraps around to `INT_MAX`, resulting in undefined behavior. If the same input is provided to the transpiled Rust program, the execution results in a panic event that prevents the overflow. This is because the Rust program uses variables of type `usize` for loop conditions, which do not allow negative values or overflows.

```c
// Original C program
while (1) {
    scanf("%d%d", &n, &m);
    for (k = 0; k < m - 1; k++) {  // Arithmetic overflow
        ...
    }
}
```

```rust
// Translated Rust code
fn main() {
    ...
    loop {
        let line = lines.next().unwrap().unwrap();
        let mut parts = line.split_whitespace();
        let n: usize = parts.next().unwrap().parse().unwrap();
        let m: usize = parts.next().unwrap().parse().unwrap();
        ...
    }
}
```

6.2.3 Dereference Failures. During our analysis of memory-related vulnerabilities, we observe several common causes of such issues. First, the absence of validation when negative values are provided as sizes for memory allocations or array indices. Second, dynamic memory allocation functions like `malloc` may return `NULL` upon failure, and if unchecked, this can lead to `NULL` pointer dereferences. Lastly, manual memory management in C often results in use-after-free errors or dangling pointers, particularly when freed memory is accessed inadvertently.

In contrast, when these vulnerable C programs are translated into Rust, the original memory vulnerabilities are mitigated due to Rust's inherent safety guarantees. For complex data structures such as trees, Rust employs constructs like `Rc<RefCell<T>>` for shared ownership and interior mutability, and uses `Weak` pointers to represent non-owning references. This ensures that references to potentially deallocated nodes are handled safely, thus preventing use-after-free or invalid pointer dereferences.

Unlike C's `malloc`, which returns `NULL` on failure, Rust's `Vec` handles memory allocation more robustly by either allocating successfully or panicking in case of failure, thereby avoiding unsafe memory accesses. Finally, Rust extensively uses the `Option` type and explicit reference checks such as `Rc::ptr_eq` to manage special cases, including sentinel nodes or optional references. This ensures that potentially `NULL` or invalid pointers are explicitly handled, effectively eliminating accidental dereferences of uninitialized or invalid memory.

6.2.4 Rust Code Analysis. We observe that each major vulnerability class in the original C programs corresponds to specific compilation-time guarantees provided by Rust's type system and ownership model. For further validation, we used the automated Rust memory safety verification tools `Rudra` [53] and `RAPx` [54] on all transpiled programs. While `Rudra` detected no issues, `RAPx` reported potential memory violations (double-free, use-after-free, and memory leaks) for a fraction of the translated Rust programs of all LLMs (0.22% for `DeepSeek-Coder`, 0.18% for `Llama3`, 0.89% for `Qwen2.5-Coder`, 0.95% for `gpt-4o`, 0.89% for `Codestral`, and 1.44% for `DeepSeek-V3`).

After conducting thorough manual inspection of all these cases, we confirmed that these were all false positives, further supporting our safety claims. Despite explicit instructions to generate safe Rust code, all the evaluated models 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`). Nonetheless, `Rudra` and `RAPx` did not identify any memory issues in these programs. Given the relatively low frequency of such occurrences, the presence of `unsafe` blocks can be easily detected and managed, mitigating any associated security risks.

7 LIMITATIONS AND FUTURE WORK

Our evaluation data set is drawn from CodeNet, which comprises competitive-style programs from online coding platforms. Since we use the most recent LLMs, it is possible that our benchmark programs were included in the training process of these models. However, this potential data leakage between our evaluation data set and model training does not pose a major threat to the validity of our results. This is because these models are not designed specifically for code translation tasks, so they may contain independent program samples, but not their functionally equivalent code pairs in other languages (and specifically Rust).

To measure the functional correctness of the transpiled programs, we use the commonly accepted approach of test cases, which carries the inherent risk of considering a buggy translation as correct. We relied on the test cases provided by CodeNet to assess the functional correctness of translated programs, and our measurements show that these test cases achieve high line coverage. A limitation of our evaluation is based on relatively simple programs, with corresponding test cases mostly in the form of I/O pairs. As part of our future work, we plan to evaluate our approach on more complex programs that will require more comprehensive test suites—the main challenge such a study entails is that it requires a non-trivial experimental setup, and the collection of a large enough data set with enough such test cases per program.

For our guided repairing 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.

It is important to acknowledge that while Rust effectively mitigates many traditional memory vulnerabilities found in C, the language presents its own unique classes of potential bugs, particularly in the design and implementation of safe abstractions around `unsafe` code [55, 56] even in safe rust [57]. Future work might explore automated methods to refactor these `unsafe` implementations in Rust's translations into safe alternatives where possible.

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 how Rust's safety guarantees can mitigate the memory vulnerabilities present in the original C programs. We believe our findings highlight the potential of LLMs for automated transpilation to memory-safe languages and will encourage further research in this area.

REFERENCES

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

[2] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, pages 48–62, 2013.

[3] Scott A. Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 193–204, 2017.

[4] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in Rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[5] Experimenting with Rust in Chromium. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/rust-toolchain.md.

[6] Diane Hosfelt. Implications of rewriting a browser component in Rust. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/, 2019.

[7] Nelson Elhage. Supporting Linux kernel development in Rust. https://lwn.net/Articles/829858/, 2020.

[8] Jeff Vander Stoep and Stephen Hines. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html, 2021.

[9] Piyus Kedia, Manuel Costa, Matthew Parkinson, Kapil Vaswani, Dimitrios Vytiniotis, and Aaron Blankstein. Simple, fast, and safe manual memory management. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 233–247, 2017.

[10] Immunant. C2Rust. https://github.com/immunant/c2rust, 2022. Accessed: [Insert Date Here].

[11] 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.

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

[13] Jaemin Hong and Sukyoung Ryu. 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*, pages 40–52, 2024.

[14] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31, 2018.

[15] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[16] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021.

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

[18] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 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*, pages 1–13, 2024.

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

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

[21] 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.

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

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

[24] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 02 2024.

[25] 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.

[26] 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.

[27] 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.

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

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

[30] Paul F. Albrecht, Philip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg-Brückner. Source-to-source translation: Ada to Pascal and Pascal to Ada. *SIGPLAN Not.*, 15(11):183–193, Nov. 1980.

[31] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 20601–20611, 2020.

[32] 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.

[33] Jaemin Hong. Improving automatic C-to-Rust translation with static analysis. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 273–277, 2023.

[34] 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 44th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 354–355, 2022.

[35] Xiangjun Han, BaoJian Hua, Yang Wang, and Ziyao Zhang. RUSTY: Effective C to Rust conversion via unstructured control specialization. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, pages 760–761, 2022.

[36] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92:103917, 2025.

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

[38] Efe Bozkir, Süleyman Özdel, Ka Hei Carrie Lau, Mengdi Wang, Hong Gao, and Enkelejda Kasneci. 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*, pages 1–7, 2024.
[39] Yoonsang Kim, Zainab Aamir, Mithilesh Singh, Saeed Boorboor, Klaus Mueller, and Arie E. Kaufman. Explainable XR: Understanding user behaviors of XR environments using LLM-assisted analytics framework. *IEEE Transactions on Visualization and Computer Graphics*, 2025.
[40] Ranjan Sapkota, Shaina Raza, Maged Shoman, Achyut Paudel, and Manoj Karkee. Image, text, and speech data augmentation using multimodal LLMs for deep learning: A survey. *arXiv preprint arXiv:2501.18648*, 2025.
[41] Muhammad Muzammil, Abisheka Pitumpe, Xigao Li, Amir Rahmati, and Nick Nikiforakis. The Poorest Man in Babylon: A Longitudinal Study of Cryptocurrency Investment Scams. In *Proceedings of The Web Conference (WWW)*, 2025.
[42] Hamed Jelodar, Mohammad Meymani, and Roozbeh Razavi-Far. Large language models (LLMs) for source code analysis: applications, models and datasets. *arXiv preprint arXiv:2503.17502*, 2025.
[43] Ruishi Li, Bo Wang, Tianyu Li, Prateek Saxena, and Ashish Kundu. Translating C to Rust: Lessons from a user study. *arXiv preprint arXiv:2411.14174*, 2024.
[44] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. Fixing Rust compilation errors using LLMs. *arXiv preprint arXiv:2308.05177*, 2023.
[45] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.
[46] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for Java-Python program translation. *arXiv preprint arXiv:2108.11590*, 2021.
[47] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.
[48] Tree-sitter. https://github.com/tree-sitter/tree-sitter, 2023. Accessed: March 14, 2025.
[49] Free Software Foundation. gcov – a test coverage program. GNU Compiler Collection (GCC) documentation, 1986. Available at https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[50] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.
[51] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C Cordeiro, and Vasileios Mavroeidis. 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*, pages 33–43, 2023.
[52] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 888–891, 2018.
[53] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 84–99, 2021.
[54] Artisan-Lab. RAPx: Rust analysis platform. https://github.com/Artisan-Lab/RAPx, 2025. Accessed: 2025-05-13.
[55] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. A closer look at the security risks in the Rust ecosystem. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–30, 2023.
[56] Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. Friend or foe inside? exploring in-process isolation to maintain memory safety for unsafe rust. In *2023 IEEE Secure Development Conference (SecDev)*, pages 54–66. IEEE, 2023.
[57] Muhammad Hassnain and Caleb Stanford. Counterexamples in safe Rust. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pages 128–135, 2024.

A APPENDIX

Description of the most frequent Rust error codes encountered during our experimental evaluation.

- **E0061:** An invalid number of arguments was passed when calling a function.
- **E0106:** Missing lifetime specifier in a type.
- **E0133:** Use of unsafe code without an `unsafe` block.
- **E0252:** A name is defined multiple times in the same scope.
- **E0277:** A type does not implement a required trait.
- **E0282:** Type annotations needed because the compiler cannot infer the type.
- **E0284:** Overlapping implementations of a trait.
- **E0308:** Mismatched types.
- **E0369:** Binary operation cannot be applied to the given types.
- **E0382:** Use of moved value.
- **E0384:** Cannot assign twice to immutable variable.
- **E0425:** Cannot find value in this scope.
- **E0428:** Duplicate definitions with the same name.
- **E0432:** Unresolved import.
- **E0433:** Failed to resolve a path.
- **E0434:** Can't capture dynamic environment in a function item.
- **E0499:** Cannot borrow as mutable more than once at a time.
- **E0502:** Cannot borrow as mutable because it is also borrowed as immutable.
- **E0506:** Cannot assign to a variable that is borrowed.
- **E0530:** Use of `self` in a static method.
- **E0596:** Cannot borrow immutable item as mutable.
- **E0599:** No method found for the given type.
- **E0600:** Cannot call a non-function.
- **E0608:** Cannot index into a value of this type.
- **E0609:** Cannot access field of a primitive type.