Raw Pointer Rewriting with LLMs for Translating C to Safer Rust

Yifei Gao[1], **Chengpeng Wang**[1], **Pengxiang Huang**[2],
Xuwei Liu[1], **Mingwei Zheng**[1], **Xiangyu Zhang**[1]
[1]Purdue University [2]University of Rochester
{gao749, wang6590, liu2598, zheng618, xyzhang}@purdue.edu
pengxiang.huang@rochester.edu

Abstract

There has been a growing interest in translating C code to Rust due to Rust's robust memory and thread safety guarantees. Tools such as C2Rust enable syntax-guided transpilation from C to semantically equivalent Rust code. However, the resulting Rust programs often rely heavily on unsafe constructs, particularly raw pointers, which undermines Rust's safety guarantees. This paper aims to improve the memory safety of Rust programs generated by C2Rust by eliminating raw pointers. Specifically, we propose a raw pointer rewriting technique that lifts raw pointers in individual functions to appropriate Rust data structures. Technically, PR^2 employs decision-tree-based prompting to guide the pointer lifting process. It also leverages code change analysis to guide the repair of errors introduced during rewriting, effectively addressing errors encountered during compilation and test case execution. We implement PR^2 and evaluate it using `gpt-4o-mini` on 28 real-world C projects. It is shown that PR^2 successfully eliminates 18.57% of local raw pointers across these projects, significantly enhancing the safety of the translated Rust code. On average, PR^2 completes the transformation of a project in 5.02 hours, at a cost of $1.13. Our code is available at https://github.com/bhcsayx/PR2.

1 Introduction

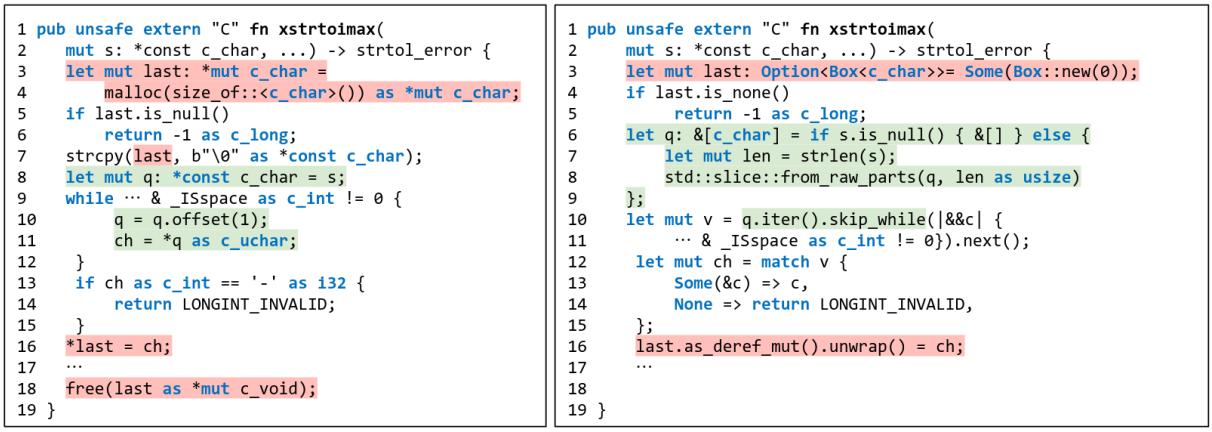
C has long been a dominant language for system and application software, powering critical infrastructure like operating systems, embedded devices, and security libraries. However, its lack of built-in memory (van Oorschot, 2021, 2023; Lord, 2023) and type safety (Yang et al., 2018; Turner, 2014; Avots et al., 2005) has led to significant vulnerabilities. A notable example is the Heartbleed bug, an out-of-bounds read in OpenSSL caused by improper memory handling, which led to large-scale privacy leaks, highlighting the risks of C language lacking strict compile-time checks for resource management.

Driven by the need to eliminate dangerous memory handling issues, many developers have turned to Rust, a systems programming language that enforces strict memory and type safety guarantees. Rust's ownership model imposes tight constraints on resource allocation and deallocation (Matsakis and Klock, 2014), while its borrow checker prevents various memory errors at compile time. These safety features allow Rust to deliver performance on par with C, while significantly reducing memory-related vulnerabilities. Consequently, many projects (e.g., `s2n-tls`) are migrating legacy C codebases to Rust, enhancing the security and longevity of critical software.

Unfortunately, translating C code to Rust is challenging. Tools like C2Rust (Galois and Immunant, 2025) can achieve syntax-level transpilation but often rely on raw pointers and unsafe functions, which bypasses Rust's safety mechanisms, still introducing memory risks in the Rust code. To generate safer Rust code, existing studies (Emre et al., 2021, 2023; Zhang et al., 2023; Hong and Ryu, 2024) have explored transforming raw pointers into references or other data structures. For example, Laertes (Emre et al., 2021) and its enhanced version (Emre et al., 2023) use pointer analysis to reason about value lifetimes, enabling conversion to references, while a recent study eliminates output parameters by introducing additional return values (Hong and Ryu, 2024). However, these static analysis techniques are rule-based, limiting their ability to handle all raw pointers, as will be shown in the motivating example.

Recent advances in large language models (LLMs) have impacted programming, showing strong abilities in code comprehension (Cui et al., 2024; Nam et al., 2024; Zheng et al., 2025) and transformation (Yang et al., 2024b; Yuan et al., 2024; Yin et al., 2024). Particularly, these mod-

Findings of the Association for Computational Linguistics: ACL 2026, pages 35360–35373



```
1 pub unsafe extern "C" fn xstrtoimax(
2     mut s: *const c_char, ...) -> strtol_error {
3     let mut last: *mut c_char =
4         malloc(size_of::<c_char>()) as *mut c_char;
5     if last.is_null()
6         return -1 as c_long;
7     strcpy(last, b"\0" as *const c_char);
8     let mut q: *const c_char = s;
9     while ··· & _ISspace as c_int != 0 {
10         q = q.offset(1);
11         ch = *q as c_uchar;
12     }
13     if ch as c_int == '-' as i32 {
14         return LONGINT_INVALID;
15     }
16     *last = ch;
17     ···
18     free(last as *mut c_void);
19 }
```

(a) Before rewrite

```
1 pub unsafe extern "C" fn xstrtoimax(
2     mut s: *const c_char, ...) -> strtol_error {
3     let mut last: Option<Box<c_char>>= Some(Box::new(0));
4     if last.is_none()
5         return -1 as c_long;
6     let q: &[c_char] = if s.is_null() { &[] } else {
7         let mut len = strlen(s);
8         std::slice::from_raw_parts(q, len as usize)
9     };
10    let mut v = q.iter().skip_while(|&&c| {
11         ··· & _ISspace as c_int != 0}).next();
12     let mut ch = match v {
13         Some(&c) => c,
14         None => return LONGINT_INVALID,
15     };
16     last.as_deref_mut().unwrap() = ch;
17     ···
18
19 }
```

(b) After rewrite

Figure 1: A motivating example of eliminating raw pointers in the tool C2Rust generated Rust program by rewriting them with Rust data structures, namely a `Option` of a `Box` and a slice. (a) The program generated by C2Rust from a C program; (b) A safer Rust program after raw pointer rewriting.

els can help translate C code into safer Rust by understanding high-level memory usage intentions hard to obtain through static analysis, guiding the conversion of raw pointers to safer structures like slices. For example, an LLM can recognize one raw pointer used to traverse a memory region and convert it to a slice in Figure 1(a). Additionally, LLMs have shown promise in program repair, fixing bugs from compilation errors or test failures, providing a degree of correctness assurance for generated Rust code (Xia and Zhang, 2024; Bouzenia et al., 2024; Kulsum et al., 2024).

Based on the above insights, we propose PR^2, a novel C-to-Rust transpilation technique that combines C2Rust for syntax-level translation with LLMs to eliminate raw pointers, eventually producing safer Rust code. With the guidance of a decision tree, PR^2 prompts LLMs for the *pointer lifting*, which replaces the low-level raw pointers with safer Rust constructs, such as `Option`, `Vec`, `Box`, slices, and references. Besides, PR^2 iteratively repairs the statements that use the newly introduced Rust data structures if the rewrite is wrong, ensuring the generated Rust code can pass the memory safety check during the compilation and passes the test cases at runtime. Particularly, we perform the *code change analysis* in the repair stage, which helps direct the LLMs to focus on the statements most likely responsible for errors. While PR^2 doesn't eliminate all raw pointers, it significantly reduces reliance on them and thus improves memory safety.

In summary, the contributions of this work include: (1) We introduce the raw pointer rewriting technique systematically eliminating raw pointers in Rust programs translated from C, facilitating translation to safer Rust. (2) We develop decision tree-based prompting to lift raw pointers into appropriate Rust data structures, and employ code change analysis to guide program repair in response to compilation errors and test case failures. (3) We implement PR^2 and evaluate it on 28 real-world programs ranging from 437 to 136k LOC. Utilizing `gpt-4o-mini`, it eliminates 2,255 local raw pointers, 3.46 times more than existing techniques, while preserving program semantics in all manually sampled cases, with an average time cost of 5.02 hours and an average financial cost of $1.13 per project.

2 Related Work

Rust's compile-time memory safety has motivated extensive work on automating C-to-Rust translation, broadly divided into two categories. Specifically, the first line of research directly translates C into unsafe but functionally equivalent Rust. Corrode (Sharp, 2025) was an early attempt focusing on module-level semantic preservation. C2Rust (Galois and Immunant, 2025) remains the most advanced industrial-grade transpiler, offering automated refactoring and cross-checking. Citrus (Citrus-rs, 2025) provides syntax-based translation for manual refactoring despite limited semantic preservation. Recent LLM-based approaches (Yang et al., 2024a; Shetty et al., 2024) generate multiple candidate translations and use test cases to select correct outputs. The second category begins with a semantics-preserving Rust baseline and incrementally enhances memory safety by rewriting raw pointers. Laertes (Emre et al., 2021) optimistically rewrites pointers into references, while its enhanced version (Emre et al.,

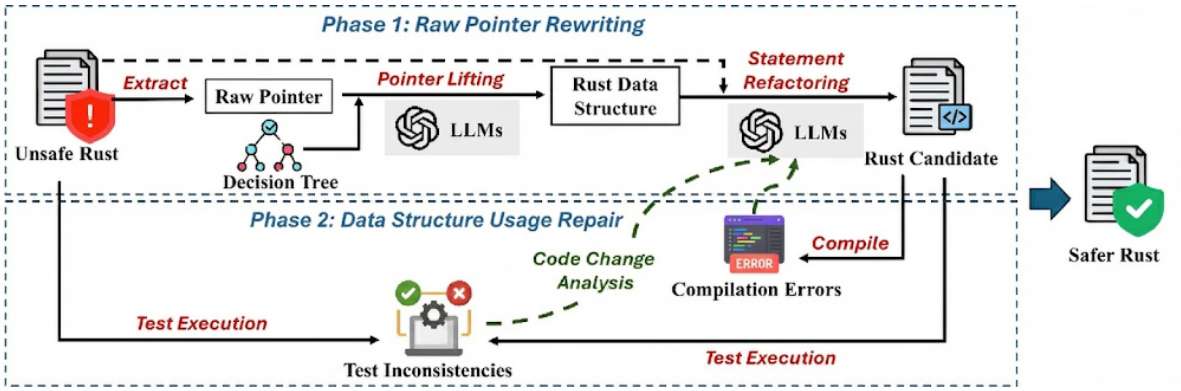


Figure 2: The workflow of PR^2

2023) applies more precise pointer analysis, albeit risking dynamic semantic changes. NOPCRAT (Hong and Ryu, 2024) and TYMCRAT (Hong and Ryu, 2025) focus on refining function parameters into safe constructs like `Option` and `Result`. Although the semantic equivalence can be guaranteed, existing techniques only support rewriting a restrictive form of raw pointers, leaving numerous raw pointers not rewritten in the Rust code.

With the rapid development of generative AI, LLMs have greatly advanced cross-language code translation by enabling functionality-preserving conversions. However, LLM-generated translations often suffer from syntax errors and semantic inconsistencies, limiting their practical reliability. To mitigate this, many approaches incorporate test cases to detect errors and iteratively refine outputs. For example, UNITRANS (Yang et al., 2024b) generates and executes test cases to validate translations across languages, using feedback for refinement. COTRAN (Jana et al., 2024) goes further by fine-tuning an LLM with reinforcement learning for Java–Python translation, using compiler feedback and symbolic execution to ensure semantic equivalence. Despite these advances, C-to-Rust translation poses unique challenges. Rust's strict ownership, borrowing, and lifetime rules require fundamental changes in memory management and pointer handling, unlike C++, Java, or Python. A recent work (Hong and Ryu, 2025) tackles signature-level type migration by converting C function signatures into idiomatic Rust, but it is limited to interfaces and still reports compiler and semantic errors, highlighting the insufficiency of LLMs alone for Rust's strict type system.

3 Approach

This section presents technical details of PR^2, of which the workflow is shown in Figure 2. Specifically, PR^2 attempts to eliminate raw pointers in single functions by using Rust data structures (Phase 1) and repairs the usage of the introduced Rust data structures (Phase 2) to ensure that the generated Rust code can be compiled successfully and pass the test cases. If the data structure usage repair fails a specific number of times, denoted by N, PR^2 gives up replacing the raw pointer. In our implementation, we set N to five by default. The details of raw pointer rewriting and data structure usage repair are demonstrated in Section 3.1 and Section 3.2, respectively.

3.1 Raw Pointer Rewriting

This section presents the details of rewriting raw pointers using Rust data structures. Specifically, we introduce two important stages, i.e., *pointer lifting* and *statement refactoring*, in Section 3.1.1 and Section 3.1.2, respectively.

3.1.1 Pointer Lifting

Existing techniques (Emre et al., 2021, 2023; Zhang et al., 2023; Hong and Ryu, 2024) primarily focus on replacing raw pointers with references based on the results of pointer analysis. However, due to challenges in analyzing the semantics of complex program constructs, such as loops and library function calls, these approaches often fail to identify opportunities to refactor raw pointers into more expressive Rust data structures, such as `Vec` and slices. This limitation degrades the overall effectiveness of these methods, resulting in many raw pointers being retained in the translated Rust programs.

To enable the rewriting of raw pointers using a wider range of Rust data structures, we leverage the LLMs' capability to understand program semantics, that is, *the LLMs can act as a flexible program analyzer when guided by a well-crafted prompt*. To this

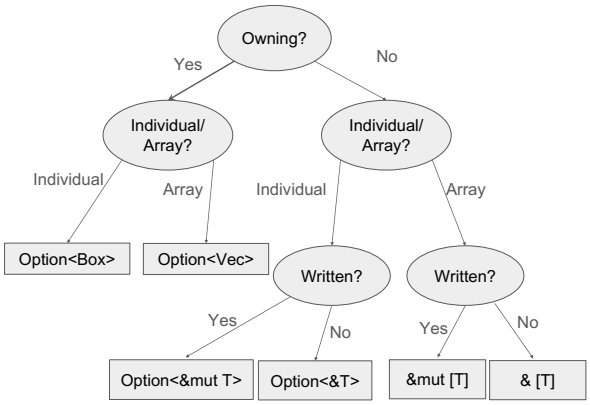


Figure 3: The decision tree used for the pointer lifting

Analysis: Here is a Rust function translated from C code and contains a raw pointer:
Function: [FUNC] **Raw pointer:** [POINTER]
You need to conduct the following three analyses upon the raw pointer:
1) Analyze whether the raw pointer owns the memory.
2) Analyze the shape of the memory, i.e., whether it is a singleton or array.
3) Analyze memory read/write operations using the raw pointer.

Based on the analysis results, please follow the decision tree [DECISION TREE LOGIC] to determine an appropriate data structure to replace the raw pointer.

Expected Output Format: Return ONLY one of the following six candidates: `Option<Box<TYPE>>`, `Option<Vec<TYPE>>`, `Option<&mut TYPE>`, `Option<&TYPE>`, `&mut [TYPE]`, `&[TYPE]`

Figure 4: The prompt template used in the pointer lifting

end, we introduce a decision tree-based prompting strategy that directs the LLMs to perform semantic analysis on raw pointers. This analysis helps the LLMs identify suitable safe Rust data structures for replacement. As illustrated in Figure 3, we enforce the LLMs to reason about three key properties of each raw pointer: *ownership*, *the shape of the memory buffer* (i.e., singleton or array), and *memory access* (i.e., read/write behavior). These properties collectively determine whether and how a raw pointer can be replaced with a specific data structure. Concretely, the ownership property helps determine whether the pointer can be refactored into a reference, `Box`, or `Vec`; the shape property indicates whether the pointer corresponds to a single value or a collection (e.g., `Vec` or slice); and the memory-access property informs whether the resulting data structure should be mutable or immutable. Since raw pointers can be null, we conservatively wrap all inferred data structures in `Option`. Following this decision tree, we can instantiate the prompt template shown in Figure 4 to generate a concrete prompt that guides the LLMs in lifting the raw pointer to an appropriate Rust data structure. As an example, the LLMs can correctly infer the non-owning, array-like, and immutable properties of pointer `q` (Line 8 in Figure 1(a)) and determine

Refactor: Please rewrite `**Raw pointer**` in `**Function**` below using given `**Data structure**` and refactor relevant statements using the constructors and other APIs offered by the data structure

Function: [FUNC] **Raw pointer:** [POINTER] **Data structure:** [Data structure]

Here are two important hints:
1) You may need to utilize the constructors of the data structure to replace the declaration of the raw pointer for data structure initialization.
2) You need to conduct the **alias analysis** to identify all the use-sites of the raw pointers and refactor the statements.

Here are some examples: [EXAMPLES]

Figure 5: The prompt template for statement refactoring

the data structure as an immutable slice.

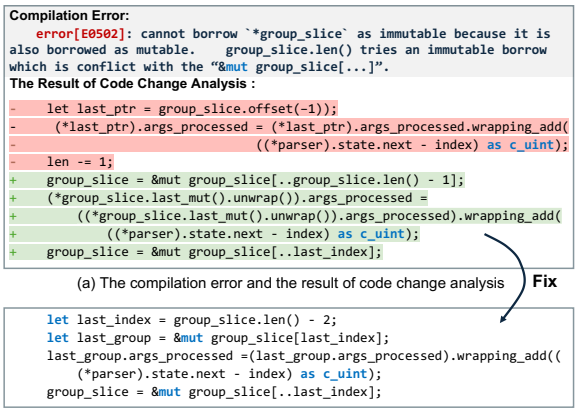
In contrast to rule-based symbolic static analysis approaches, our decision tree-based prompting strategy enables the LLMs to leverage its semantic understanding and creativity to identify more expressive patterns of raw pointer manipulation, such as those involving loops and library function calls, which are typically difficult for traditional symbolic analysis to handle. Moreover, our approach is easily extensible: by designing more sophisticated decision trees and corresponding prompts, we can support more complex data structures. In this work, we focus on sequential data structures for non-singleton values. As one of the future works, we plan to extend PR^2 to support additional collection types, such as sets and maps.

3.1.2 Statement Refactoring

Based on the result of pointer lifting, we proceed to replace the raw pointer with the suggested data structure. This process involves refactoring both the declaration of the raw pointer and all statements that make use of it. As illustrated by the prompt template in Figure 5, we provide two key hints to the LLMs. First, we instruct the LLMs to replace the declaration of the raw pointer with a constructor of the suggested data structure. Second, we enforce the LLMs to reason about the aliasing facts of the raw pointer in order to locate all potential use-sites, and refactor them using the appropriate APIs provided by the data structure. Notably, the statement refactoring leverages two core capabilities of the LLMs, including analyzing semantic properties of the program (e.g., alias analysis) and its ability to generate code (e.g., rewriting statements accordingly), as shown in Figure 1(b).

3.2 Data Structure Usage Repair

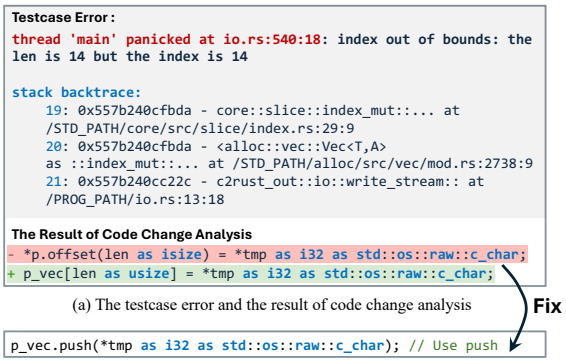
This section presents data structure usage repair, which consists of *compilation error fixing* (Section 3.2.1) and *testcase error fixing* (Section 3.2.2).



(a) The compilation error and the result of code change analysis

(b) The result of compilation error fixing

Figure 6: An example of compilation error fixing



(a) The testcase error and the result of code change analysis

(b) The result of testcase error fixing

Figure 7: An example of testcase error fixing

3.2.1 Compilation Error Fixing

As shown in Section 3.1, only the code within a single function is modified during the raw pointer rewriting. This design implies that any resulting compilation error must be introduced by the modified statements within that function, enabling us to narrow down the scope of fixing to the function for the error fixing. However, a single function may contain many raw pointers, most of which are unrelated to the repair task. What is even worse, the function itself can be quite large (e.g., over one hundred lines), with the majority of statements not interacting with the memory addressed by the raw pointers. These unrelated raw pointers and statements introduce irrelevant tokens as the *noise* into the repair context, which poses a significant challenge for the LLMs, often leading to hallucinations and repair failures.

To enable more informed repairs using the LLMs, we filter out irrelevant tokens from the repair context. Technically, we introduce *code change analysis*, which computes the difference between the function before and after the raw pointer rewriting. The resulting added, deleted, or modified statements are the most likely sources of compilation errors. By feeding code changes along with error messages into the LLMs, we can effectively guide the LLMs to focus its *attention* on the most relevant statements, improving its ability to fix compilation errors. Due to space limit, we do not show the prompt template in the paper as the design is very standard based on the code change analysis. Figure 6 shows an example of using code change analysis to fix a conflicting immutable (`group_slice.len()`) and mutable (`&mut group_slice`) borrows error. Specifically, Rust compiler allows only one mutable reference to a variable at any time, thus calling the API `len` to the slice will raise an error as it needs to hold an immutable reference. To resolve this error, the fix calls the API `len` before the mutable slice is created, and stores the length value in a new variable, avoiding the interleaving of references.

3.2.2 Testcase Error Fixing

Once we obtain a Rust program with all compilation errors fixed, we proceed to run the program's test cases to uncover potential violations of the original input-output relationship. The overall design of testcase error fixing closely follows that of compilation error fixing described in Section 3.2.1. Figure 7 shows a prompt template and an example of testcase error fixing. It is important to note that passing all test cases which the original Rust program pass is a necessary condition for a valid translation, although it does not theoretically guarantee semantic equivalence. In practice, the validity of our approach depends on the quality of the available testcases. While many real-world projects include comprehensive test suites, there may be corner cases not captured by the tests.

4 Evaluation

We implemented PR^2 as a prototype with 1,072 lines of Rust and 1,187 lines of Python. It uses `tree-sitter` to extract raw pointers, struct definitions, and global variables from translated Rust code. Pointer lifting, statement refactoring, and repair phases are powered by `gpt-4o-mini`, chosen for its low cost (i.e., \$0.150/M input tokens and \$0.600/M output tokens (OpenAI, 2025)), making it suitable for iterative transformation. To speed up compilation during error fixing, we use `cargo check`, which performs type and borrow checking without linking. We allow up to five repair attempts per pointer (i.e., the value of N in Section 3 is five) and set the temperature to 0.0 for greedy decoding.

Table 1: The main statistics of PR^2

ID	#ERP	#AF	RT(s)	CFT(s)	TFT(s)	TotalT(s)	#InTok	#OutTok	#P	Cost($)
1	8	5	188.81	45.52	10.65	244.98	46,068	13,336	39	0.015
2	10	9	305.84	123.32	20.46	449.61	81,371	27,257	89	0.029
3	6	4	747.44	118.85	215.55	1,081.84	222,652	72,066	94	0.077
4	6	5	1,263.03	161.50	45.86	1,470.39	318,859	57,769	125	0.082
5	22	10	1,182.65	164.52	11.70	1,358.88	268,129	70,788	146	0.083
6	11	10	1,150.23	264.11	349.21	1,763.55	392,429	87,263	252	0.111
7	17	12	1,595.18	451.75	66.10	2,113.03	427,183	127,902	187	0.141
8	18	15	1,809.05	537.59	31.19	2,377.83	436,328	123,892	253	0.140
9	17	13	2,930.73	316.33	436.54	3,683.60	928,716	120,543	278	0.212
10	16	11	2,839.98	330.11	14.17	3,184.26	610,139	112,067	206	0.159
11	15	12	3,365.71	693.44	158.15	4,217.30	1,102,037	147,435	322	0.254
12	12	12	4,873.78	491.10	75.06	5,439.94	2,010,145	160,203	363	0.398
13	38	27	4,743.65	793.61	238.83	5,776.08	1,504,725	264,773	511	0.385
14	53	43	8,951.85	2,658.11	1,037.37	12,647.33	14,008,537	418,288	920	2.352
15	80	61	8,293.31	2,281.44	862.65	11,437.40	9,699,548	519,250	1,137	1.766
16	87	70	12,463.53	650.76	319.79	13,434.08	2,651,673	234,328	1,004	0.538
17	60	47	6,510.92	582.86	586.44	7,680.21	579,508	136,921	531	0.169
18	99	80	16,218.48	4,631.28	622.08	21,471.84	4,546,460	952,967	1,980	1.254
19	73	58	18,867.08	2,041.97	1,963.26	22,872.31	4,831,868	356,541	1,490	0.939
20	249	132	16,791.45	2,422.30	12,365.05	31,578.80	5,352,697	881,196	1,541	1.332
21	157	113	20,378.32	4,229.75	2,410.53	27,018.61	8,878,129	861,532	1,882	1.849
22	194	158	19,006.08	8,199.58	2,640.55	29,846.22	10,299,083	960,924	2,039	2.121
23	105	83	12,430.74	2,192.71	8,888.03	23,511.47	60,813,477	609,506	1,594	9.488
24	84	68	29,406.70	1,595.97	6,008.53	37,011.20	9,507,522	442,569	2,225	1.692
25	158	130	16,422.75	5,805.27	6,840.01	29,068.04	2,625,267	606,173	2,128	0.757
26	175	152	41,537.59	3,229.05	38,712.37	83,479.01	17,347,685	662,482	3,114	3.000
27	138	125	36,488.11	4,170.67	20,440.27	61,099.05	3,322,335	616,631	2,570	0.868
28	347	257	51,822.16	5,775.61	2,837.01	60,434.78	5,344,925	973,554	3,621	1.386

We evaluate the effectiveness and efficiency of PR^2 by answering the four research questions:

- **RQ1.** How effectively and efficiently does PR^2 eliminate raw pointers in the Rust programs generated by C2RUST?
- **RQ2.** How does PR^2 compare with existing techniques that transform C to safer Rust programs?
- **RQ3.** Which kinds of data structures are introduced by PR^2 to replace raw pointers?
- **RQ4.** How do the decision tree-based prompting and code change analysis contribute to the performance of PR^2?

Benchmark. We adopt C projects from the evaluation set used in NOPCRAT (Hong and Ryu, 2024), comprising widely used real-world applications also used in prior C-to-Rust studies (Emre et al., 2021, 2023). Of the original 55 programs, 26 lack test cases and 1 contains raw pointers only in function parameters, which are outside our scope. We thus select the remaining 28 projects with test cases as our benchmark, which are listed by Table 4 in Appendix A.1. Overall, the benchmark spans programs from a few hundred to 136K LOC and 17 to 2,731 raw pointers, providing a diverse set for evaluating the effectiveness and scalability of PR^2.

4.1 RQ1: Effectiveness and Efficiency

Setup and Metrics. To quantify the effectiveness of PR^2, we introduce the number of eliminated raw pointers as our main metric. Meanwhile, we collect the number of functions refactored during this process. The eliminated raw pointers in these functions can improve the memory safety of the function usage. To quantify the efficiency of PR^2, we measure the time cost, input/output token costs, prompting rounds, and financial cost. Notably, we do not include the time overhead introduced by C2RUST, as it is negligible compared to the overhead incurred during the rewriting and repair phases of PR^2.

Results. Table 1 presents detailed evaluation results of PR^2. The column **#ERP** shows the number of eliminated raw pointers. The column **#AF** indicates the number of affected functions. On average, PR^2 removes 18.57% of raw pointers and transforms 13.05% of functions into safer code. Notably, it eliminates 34.3% of pointers in `grep-3.11` (ID = 20), demonstrating the effectiveness of PR^2 when analyzing large projects.

The columns **RT**, **CFT**, and **TFT** represent time spent on raw pointer rewriting, compiler error fixing, and test case error fixing, respectively, with total time in **TotalT**. On average, PR^2 takes 5.02 hours per project, with the longest taking 28

hours. The bottleneck in time is rewriting, averaging 74.26% of total time. Additionally, the columns **InTok**, **OutTok**, and **#P** represent input/output tokens and prompt rounds during rewriting and repair. The column **Cost** reflects total prompting expense. Each successful rewrite requires 14 prompt rounds. Across all 28 benchmarks, the total cost is $31.60, averaging just $1.13 per project, highlighting the high efficiency of PR^2.

4.2 RQ2: Comparison with Baselines

Setup and Metrics. We compare our approach with LAERTES (Emre et al., 2021), an automated tool that aims to convert raw pointers into safe references. Specifically, LAERTES applies an optimistic rewriting strategy and iteratively addresses lifetime inference and borrow-checking errors to achieve a successful compilation. Another recent work, NOPCRAT (Hong and Ryu, 2024), focuses on rewriting raw pointers in the parameters that propagate output values as side effects. However, due to the difference in scope, the sets of raw pointers eliminated by PR^2 and NOPCRAT are disjoint thus we do not include it in our comparative evaluation.

Results. Table 2 shows the comparison results of PR^2 and the baselines. The column **ID** lists program identifiers (as in Table 4). In the column **LAERTES**, the sub-column $\boldsymbol{E_L}$ denotes the number of raw pointers eliminated by LAERTES, while the sub-column $\boldsymbol{E_{L \setminus P}}$ indicates the number of the raw pointers LAERTES eliminates and PR^2 does not eliminate. The sub-column $T(s)$ indicates the execution time of LAERTES. Similarly, the column **PR^2** includes the sub-column $\boldsymbol{E_P}$, which indicates the number of pointers eliminated by PR^2, and $\boldsymbol{E_{P \setminus L}}$, which indicates the number of raw pointers eliminated by PR^2 and not eliminated by LAERTES. Overall, PR^2 removes 2,255 raw pointers, compared to 506 by LAERTES, and outperforms it on most projects. The size of the project `buffer` (ID = 2) is small, enabling LAERTES to infer lifetimes effectively and efficiently. Notably, PR^2 uniquely eliminates 2,160 raw pointers (95.79% of its total), indicating that most rewrites are beyond the reach of the baseline LAERTES. In contrast, LAERTES only uniquely handles 438 raw pointers. Overall, PR^2 eliminates 3.46× more raw pointers than LAERTES, demonstrating its effectiveness.

In terms of efficiency, LAERTES processes all 28 projects in 3,732.85 seconds (133.32 seconds per project), while PR^2 averages 5.02 hours per project due to prompt-related overhead. However, with ongoing LLM performance improvements, PR^2 is expected to become significantly faster over time.

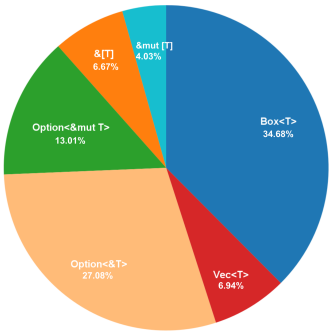


Figure 8: The proportions of different data structures

4.3 RQ3: Categorization of Data Structures

Setup and Metrics. To show the diversity of Rust data structures utilized by PR^2, we categorize Rust data structures used by PR^2 to uniquely eliminate rewritten raw pointers. This analysis is important because, unlike LAERTES, which primarily converts raw pointers to references, PR^2 supports a broader spectrum of Rust data structures, thereby offering more improvements on memory safety.

Results. Figure 8 shows the distribution of Rust data structures used in pointer rewriting. Overall, singleton data structures, e.g., `Box`, `Option<&T>`, and `Option<&mut T>`, account for the majority of rewritten raw pointers (74.8%). Meanwhile, a non-trivial portion (17.6%) of raw pointers are successfully rewritten into sequential data structures, including `Vec`, `&[T]`, and `&mut [T]`. In terms of ownership, 41.6% of raw pointers are rewritten into `Box` and `Vec` which ensures that the lifetimes of the wrapped values are properly bounded, thereby preventing memory leaks. Lastly, 33.8% of the raw pointers are converted into read-only structures (e.g., `Option<&T>`, `&[T]`), removing unnecessary mutability. We present a case demonstrating PR^2's capability of leveraging Rust inherent APIs. We provide a case study of data structure usage within the generated Rust code in Appendix A.2.

4.4 RQ4: Ablation Study

Setup and Metrics. To quantify how the technical designs of PR^2 contribute to the rewriting results, we introduce two ablations: NT and NCC, which disable decision tree-based prompting and code change analysis in the pointer lifting and data structure usage repair, respectively. Specifically, the ablation NT enforces the LLM to directly

Table 2: The statistics of comparison with baseline LAERTES and two ablations NT and NCC

Id	LAERTES			PR^2		NT		NCC	
	E_L	$E_{L\setminus P}$	T(s)	E_P	$E_{P\setminus L}$	E_T	$E_{T\setminus L}$	E_C	$E_{C\setminus L}$
1	2	2	1.15	8	8	3	3	11	10
2	27	17	2.41	10	0	8	2	12	1
3	0	0	1.74	6	6	1	1	2	2
4	5	2	1.44	6	3	5	3	4	2
5	3	2	4.35	22	21	7	7	10	10
6	1	0	5.64	11	10	7	7	7	7
7	0	0	1.73	17	17	6	6	9	9
8	4	2	4.11	18	15	11	9	16	13
9	2	2	7.11	17	17	7	6	0	0
10	12	11	7.74	16	15	8	6	8	7
11	3	3	25.55	15	15	7	7	7	7
12	14	12	5.54	12	10	2	2	4	3
13	5	3	4.25	38	35	8	7	15	15
14	9	6	121.46	53	48	36	33	40	35
15	14	12	46.22	80	77	45	41	61	56
16	7	7	15.08	87	86	56	55	68	67
17	12	12	42.42	60	58	54	52	62	60
18	2	1	90.6	99	98	0	0	0	0
19	28	28	18.2	73	73	91	85	71	69
20	52	39	45.38	249	227	80	75	68	65
21	42	40	47.66	157	155	96	95	95	93
22	43	36	95.03	194	187	108	102	124	118
23	37	24	35.06	105	88	0	0	0	0
24	10	10	74.99	84	84	0	0	41	41
25	58	53	713.13	158	153	142	132	177	168
26	37	37	164.86	175	170	187	179	169	164
27	40	40	1619.28	138	138	95	94	106	104
28	37	37	530.72	347	346	138	138	393	392

choose a data structure from the six data structures shown in the leaf nodes in Figure 3, without conducting semantic analyses. The ablation NCC feeds the functions before and after raw pointer rewriting to the LLM along with error messages, but without any code change information. We use the number of raw pointers eliminated under different ablations as the primary evaluation metric.

Results. Table 2 reports results for ablations NT and NCC. Under the column **NT**, the sub-columns $\boldsymbol{E_T}$ and $\boldsymbol{E_{T\setminus L}}$ denote the total and uniquely eliminated pointers, respectively; **NCC** shows the same for its setting. As shown, NT eliminates 1,208 pointers (46.43% fewer than the full PR^2), and NCC eliminates 1,580 (29.93% fewer). Removing the decision tree prompt causes a 46.43% drop in unique eliminations; omitting code change analysis leads to a 29.93% drop. These results confirm the importance of both components—especially the decision tree—in enhancing rewrite quality. Even in reduced forms, both variants outperform LAERTES, highlighting the robustness of LLM-driven rewriting. Appendix A.3 provides a case study that shows an example handled by full-featured PR^2 but not addressed by its ablation NT.

4.5 Discussion

User Study. To assess semantic correctness, we recruited ten Rust-experienced volunteers to review outputs from PR^2. Each independently evaluated 20 randomly selected raw pointer eliminations (around 1% of all cases), using `gpt-4o-mini` explanations and Rust documentation to judge semantic equivalence. Reviews took about 45 minutes each, and all volunteers confirmed the rewritten code preserved original semantics. These results suggest PR^2 is a practical tool for C-to-Rust translation. While it lacks formal guarantees, it achieves strong empirical outcomes. With modest human effort, users can semi-automatically produce safer, idiomatic Rust code.

Threats to Validity. The primary threat to internal validity stems from the stochastic nature of LLMs, which can yield nondeterministic outputs and affect reproducibility. As noted in Section 4.4, two ablation settings slightly outperform the full-featured PR^2 on `tar-1.34` and `glpk-5.0`. To address this, we fix the temperature to 0 for deterministic behavior. External validity depends on test case quality, which influences both the elimination number and correctness. Our test-driven approach reduces manual effort, and the user study confirms that PR^2 preserves semantic equivalence. Since broader test coverage could further improve rewriting quality, we calculated the coverage of test suites in our benchmark projects, observing 62.84% on average. We then focused on rewritten pointers in functions not covered by existing test suites and leveraged LLMs to synthesize test cases and run them, in 59 total cases we found only 2 incorrect pointer rewrites, corresponding to a 96.61% correctness rate among inspected cases, which further supports the correctness of our transformation pipeline. Construct validity relates to LLM choice. We use `gpt-4o-mini`, a cost-effective and widely available model, and validate our findings with `Claude-3.5-Sonnet` and `Deepseek-v3.2` on the top 10 projects in Table 1 with results demonstrated in Table 3, where E_{Cl}, E_{Dp} correspond to pointers eliminated by `Claude-3.5-Sonnet` and `Deepseek-v3.2`, and $E_{Cl} \setminus E_l$, $E_{Dp} \setminus E_l$ correspond to pointers uniquely eliminated by `Claude-3.5-Sonnet` and `Deepseek-v3.2`. Compared with `gpt-4o-mini`, `Claude-3.5-Sonnet` and `Deepseek-v3.2` demonstrate similar performance in most projects and significantly better performance in some projects (1, 2, 8, 10). As LLMs advance and inference costs drop, the effectiveness and efficiency of PR^2 will continue to improve.

Id	E_{Cl}	$E_{Cl} \setminus E_L$	E_{Dp}	$E_{Dp} \setminus E_L$
1	14	12	14	12
2	11	1	29	1
3	4	4	10	10
4	6	5	6	5
5	24	23	20	17
6	20	19	24	23
7	18	18	18	18
8	42	41	34	33
9	22	22	27	26
10	32	31	21	18

Table 3: Additional LLM Pointer Elimination Results

5 Conclusion

This paper presents PR^2, a novel approach for transforming C programs into safer Rust programs by eliminating raw pointers using LLMs. Building on syntax-guided C-to-Rust translation, we introduce a raw pointer rewriting technique that targets raw pointers within single functions and replaces them with appropriate Rust data structures. Specifically, PR^2 employs LLMs to lift raw pointers to Rust data structures and iteratively repairs the transformed program by focusing on the changed code. Our evaluation on 28 real-world C programs demonstrates that PR^2, powered by `gpt-4o-mini`, eliminates a total of 2,255 raw pointers, with an average processing time of 5.02 hours and a cost of $1.13 per project. We hope that PR^2 provides valuable insights for advancing program translation techniques aimed at improving system reliability.

6 Limitations

Although PR^2 effectively refactors specific raw pointers using Rust data structures, it has several drawbacks. First, it performs raw pointer rewriting within individual functions, keeping type signatures unchanged, as shown in Section 3.1.2. While this conservative scope simplifies repairs, it misses chances to eliminate raw-pointer parameters, unlike NOPCRAT (Hong and Ryu, 2024). Future work could enable more expressive rewriting by modifying function signatures across call chains. Second, PR^2 lacks a theoretical correctness guarantee; without strong or sufficient test coverage, raw pointer elimination can break functional equivalence despite passing compilation and tests. Enhancements could involve synthesizing unit tests using LLMs for differential testing (Zhou et al., 2025; Zhang et al., 2025; Zheng et al.; Rao et al., 2024) and applying verification techniques like relational verification (Unno et al., 2021; Churchill et al., 2019; Zheng et al., 2024). Finally, the rewritten code may have low readability and maintainability, since current efforts focus mainly on raw pointer elimination. We plan to use LLMs in a post-processing phase to further refine and improve the quality of the refactored Rust code.

Acknowledgement

We are grateful to the Center for AI Safety for providing computational resources. This work was funded in part by the National Science Foundation Awards SHF-1901242, SHF-1910300, Proto-OKN 2333736, IIS2416835, DARPA VSPELLS - HR001120S0058, ONR N00014-23-1-2081, and Amazon. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

Dzintars Avots, Michael Dalton, V Benjamin Livshits, and Monica S Lam. 2005. Improving software security with a c pointer analysis. In *Proceedings of the 27th international conference on Software engineering*, pages 332–341.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*.

Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1027–1040. ACM.

Citrus-rs. 2025. Citrus: Convert c to rust.

Jielun Cui, Yutong Zhao, Chong Yu, Jiaqi Huang, Yuanyuan Wu, and Yu Zhao. 2024. Code comprehension: Review and large language models exploration. In *2024 IEEE 4th International Conference on Software Engineering and Artificial Intelligence (SEAI)*, pages 183–187. IEEE.

Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2023. Aliasing limits on translating c to safe rust. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):551–579.

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

Galois and Immunant. 2025. C2rust demonstration.

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

Jaemin Hong and Sukyoung Ryu. 2025. Type-migrating c-to-rust translation using a large language model. *Empirical Software Engineering*, 30(1):3.

Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution. In *ECAI 2024*, pages 4011–4018. IOS Press.

Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. 2024. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pages 103–111.

Bob Lord. 2023. The urgent need for memory safety in software products. *Cybersecurity & Infrastructure Security Agency*.

Nicholas D Matsakis and Felix S Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, pages 103–104.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

OpenAI. 2025. Api pricing.

Nikitha Rao, Elizabeth Gilbert, Tahina Ramananandro, Nikhil Swamy, Claire Le Goues, and Sarah Fakhoury. 2024. Diffspec: Differential testing with llms using natural language specifications and code artifacts. *arXiv preprint arXiv:2410.04249*.

Jamey Sharp. 2025. Corrode: Automatic semantics-preserving translation from c to rust.

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*.

Stephen Turner. 2014. Security vulnerabilities of the top ten programming languages: C, java, c++, objective-c, c#, php, visual basic, python, perl, and ruby. *Journal of Technology Research*, 5:1.

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-based relational verification. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 742–766. Springer.

Paul C van Oorschot. 2021. Toward unseating the unsafe c programming language. *IEEE Security & Privacy*, 19(02):4–6.

Paul C van Oorschot. 2023. Memory errors and memory safety: C as a case study. *IEEE Security & Privacy*, 21(2):70–76.

Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 819–831, New York, NY, USA. Association for Computing Machinery.

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

Gao Yang and 1 others. 2018. The source and exploitation of the program vulnerability. In *2018 3rd Joint International Information Technology, Mechanical and Electronic Engineering Conference (JIMEC 2018)*, pages 89–94. Atlantis Press.

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

Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472*.

Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. 2024. Transagent: An llm-based multi-agent system for code translation. *arXiv preprint arXiv:2409.19894*.

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

Yixuan Zhang, Ningyu He, Jianting Gao, Shangtong Cao, Kaibo Liu, Haoyu Wang, Yun Ma, Gang Huang, and Xuanzhe Liu. 2025. Drwasi: Llm-assisted differential testing for webassembly system interface implementations. *ACM Transactions on Software Engineering and Methodology*.

Mingwei Zheng, Qingkai Shi, Xuwei Liu, Xiangzhe Xu, Le Yu, Congyu Liu, Guannan Wei, and Xiangyu Zhang. 2024. Pardiff: Practical static differential analysis of network protocol parsers. In *Proc. ACM Program. Lang.*, OOPSLA '24, pages 1208–1234. ACM.

Mingwei Zheng, Danning Xie, Qingkai Shi, Chengpeng Wang, and Xiangyu Zhang. Validating network

protocol parsers with traceable rfc document interpretation. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2025.

Mingwei Zheng, Danning Xie, and Xiangyu Zhang. 2025. Large language models for validating network protocol parsers. *arXiv preprint arXiv:2504.13515*.

Shiyao Zhou, Jincheng Wang, He Ye, Hao Zhou, Claire Le Goues, and Xiapu Luo. 2025. Lwdiff: An llm-assisted differential testing framework for webassembly runtimes. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 769–769. IEEE Computer Society.

Table 4: The statistics of benchmark programs

ID	Project Name	C Size	Rust Size	#F	#RP
1	quadtree	437	1,180	31	17
2	buffer	452	1,111	40	30
3	genann	690	2,325	27	30
4	libtree	1,412	2,617	32	41
5	urlparser	440	1,380	21	52
6	ed-1.19	2,584	5,462	137	80
7	mcsim-6.2.0	20,033	35,558	506	80
8	hello-2.12.1	37,192	10,372	176	88
9	indent-2.2.13	20,927	15,066	128	100
10	pth-2.0.7	8,797	12,649	235	102
11	gzip-1.12	53,751	21,463	244	122
12	pexec-1.0rc8	5,684	12,027	158	156
13	units-2.22	7,253	11,188	144	185
14	patch-2.7.6	54,350	114,076	601	361
15	cpio-2.14	71,819	84,012	707	431
16	cflow-1.7	45,223	25,467	480	476
17	enscript-1.6.6	38,424	78,159	259	523
18	libosip2-5.3.1	18,848	34,195	726	628
19	rcs-5.10.1	63,254	35,811	496	664
20	grep-3.11	117,416	84,706	867	725
21	diffutils-3.10	112,943	96,418	853	760
22	findutils-4.9.0	136,421	143,361	1,232	835
23	sed-4.9	91,818	66,580	675	835
24	make-4.4.1	31,866	35,395	445	997
25	wget-1.21.4	**136,566**	**189,761**	1,355	1,320
26	tar-1.34	102,864	137,655	**1,669**	1,399
27	gawk-5.2.2	65,617	134,098	1,450	1,792
28	glpk-5.0	77,420	138,372	1,523	**2,731**

A Appendix

A.1 Details of Experimental Subjects

The details of the experimental subjects are shown in Table 4. Specifically, the columns **C Size** and **Rust Size** report the sizes of the original and C2RUST-translated code, respectively. The columns **#F** and **#RP** show the number of functions and raw pointers in the Rust output, respectively. The benchmark spans programs from a few hundred to 136K LOC and 17 to 2,731 raw pointers, providing a diverse set for evaluating PR^2's effectiveness and scalability.

A.2 An Example of Rust Data Structures

We present a case study demonstrating how PR^2 effectively transforms C code into safer and more idiomatic Rust by leveraging Rust's data structures and APIs. As shown in Figure 9, the code snippet is derived from a variant of the C standard library function *strtoull*, which parses C strings into integer values based on a specified numeric base. Specifically, the original Rust program generated by the tool C2RUST, which is shown in Figure 9(a), contains unreachable branches caused by the unsatisfiable conditions at lines 5 and 7. Meanwhile, it relies on raw pointer arithmetic via

```
let mut s: *const c_char = nptr;
if (base == 0 as c_int || base == 16 as c_int)
   && ({
      let mut __res: c_int = 0;
      if ::std::mem::size_of::<c_uchar>() as c_ulong > 1 {
         if 0 != 0 {...}
      } else {
         __res = toupper(*s.offset(1 as isize) as c_int);
      }
      __res
   }) == 'X' as i32
{
   s = s.offset(2 as c_int as isize);
   base = 16 as c_int;
}
```

(a) Before rewrite

```
let s: &[std::os::raw::c_char] = if nptr.is_null() {
    &[]
} else {
    let mut len = strlen(nptr);
    std::slice::from_raw_parts(nptr, len as usize)
};
if s[index] as c_int == '0' as i32 {
    if (base == 0 as c_int || base == 16 as c_int) &&
        s.get(index + 1).map_or(false, |&x| x as c_int == 'X' as i32)
    {
        index += 2;
        base = 16;
    }
}
```

(b) After rewrite

Figure 9: An example of rewriting a raw pointer with a slice and utilizing Rust inherent APIs

`offset`, further introducing potential invalid memory operations. After the raw pointer rewriting and data structure usage repair, PR^2 replaces the raw pointer `s` with a Rust slice (declared and initialized at line 5) and utilizes the API `get` to safely access elements by index, returning an `Option` instead of performing unchecked access. PR^2 also employs the API `map_or`, which allows the program to apply a closure to the retrieved character if it exists, or return a default value `false` when the index is out of bounds. This effectively eliminates the need for fragile conditional logic and pointer arithmetic. This case illustrates the empirical benefit of PR^2 in semantically understanding the original C logic and intelligently selecting Rust data structures as the abstractions of raw pointers through pointer lifting.

A.3 An Example of Ablation Study

Figure 10 presents an example that is successfully handled by the full-featured PR^2 but not by the ablation NT. For illustration, we show only a partial snippet of the function `DoOptimalDesign`, which in its entirety spans 208 lines—posing substantial challenges for the LLM to reason about raw pointer usage. Guided by the prompt structure shown in Figure 4, PR^2 is able to identify the ownership property, the buffer shape, and the memory read/write operations of the raw pointer `piDesign_mask`. These analyses enable the model

```
pub unsafe extern "C" fn DoOptimalDesign(mut panal: PANALYSIS) {
    let mut nDesignPts: c_long = 0;
    let mut piDesign_mask: *mut c_int = 0 as *mut c_int;
    while i < dim {
        ...
        j = 0 as c_int as c_long;
        while j < nDesignPts {
            if *piDesign_mask.offset(j as isize) == … {...}
            j += 1;
        }
        *piDesign_mask.offset(iBest as isize) =
            (*piDesign_mask.offset(iBest as isize) == 0)   as c_int;
        i += 1;
    }
    free(piDesign_mask as *mut c_void);
    CloseOptFiles(panal);
}
```

(a) Before rewrite

```
pub unsafe extern "C" fn DoOptimalDesign(mut panal: PANALYSIS) {
    let mut nDesignPts: c_long = 0;
    let mut piDesign_mask: Vec<c_int> =
        match nDesignPts > 0   {
            True => vec![0; nDesignPts as usize],
            False => Vec::new()
        };
    while i < dim {
        ...
        j = 0 as c_int as c_long;
        while j < nDesignPts {
            if piDesign_mask[j as usize] == … {...}
            j += 1;
        }
        piDesign_mask[iBest as usize] =
            (piDesign_mask[iBest as usize] == 0) as c_int;
        i += 1;
    }
    CloseOptFiles(panal);
}
```

(b) After rewrite

Figure 10: An example of a raw pointer eliminated by the full-featured PR^2 but not by the ablation NT

to lift the pointer to a `Vec`, following the decision tree defined in the tree in Figure 3. In contrast, without decision tree-based prompting, the LLM fails to identify this rewriting opportunity within the allowed rewriting budget.

A.4 Full Prompts of PR^2

In this section we present prompts of PR^2.

Full prompts for pointer lifting

Here is a Rust function translated from C code and a raw pointer in it:
Function: #FUNCTION_CONTEXT#
Raw pointer: #POINTER_DECLARATION#
Please give a retype option for the pointer based on the following decision tree:
(0) Is this pointer belonging to one of the following cases: a. A pointer with a type of void, which means this pointer may have variable type; b. A pointer that points to pointers(double pointer or higher level pointers); If this pointer belongs to above cases, return with "CANNOT_REWRITE", otherwise, jump to (1)
(1) Is the memory value represented by right hand side(RHS) will be owned by this pointer? Typically, if RHS statements are function calls(could be GNU C library functions or other functions seen previously in this program) allocating memory region or initializing objects or arrays, it is regarded as owning pointers, otherwise the pointers should be considered as non-owning, If owning, jump to (2), non-owning, jump to (3)
(2) Is the memory an individual region or an array? individual then jump to (4), array jump to (5).
(3) Is the memory is an individual region or an array? individual then jump to (6), array jump to (7).
(4) This is a final state of the decision tree, The pointer seems to be proper to convert to Option<Box<ORIG_TY>> by Box::from_raw, and if this pointer is returned, use Box::into_raw to convert back.
(5) This is a final state of the decision tree, The pointer seems to be proper to convert to Option<Vec<ORIG_TY>>.
(6) Will the memory region be written in this function? Yes, jump to (8), No jump to (9)
(7) Will the memory region be written in this function? Yes, jump to (10), No jump to (11)
(8) This is a final state of the decision tree, The pointer seems to be proper to convert to
Option<&mut ORIG_TY>
(9) This is a final state of the decision tree, The pointer seems to be proper to convert to
Option<&ORIG_TY>
(10) This is a final state of the decision tree, The pointer seems to be proper to convert to
&mut [ORIG_TY]
(11) This is a final state of the decision tree, The pointer seems to be proper to convert to
&[ORIG_TY]
You are expected to ONLY return one of following 6 candidates: 'Option<Box<ORIG_TY>>', 'Option<Vec<ORIG_TY>>', 'Option<&mut ORIG_TY>', 'Option<&ORIG_TY>', '&mut [ORIG_TY]', '&[ORIG_TY]', DO NOT rewrite the function now before more examples given.

Full prompts for statement refactoring

Good, now based on previous analysis, please rewrite "Raw pointer" in "Function" below to safe structure you found just now. If the pointer is null, use None or zero length slice to rewrite.
Function: #FUNCTION_CONTEXT#
Raw pointer: #POINTER_DECLARATION#
Here are some examples that may be helpful: #EXAMPLES#
Following are some struct definitions and types of static variables that may be useful(DO NOT MODIFY ANYTHING IN THEM):
Structs: #STRUCTS_USED#
Statics: #STATICS_USED#
Format instructions:
1) Your response should include a rust code snippet starting with ```rust and ending with ```, which contains the rewritten function inside.
2) When rewriting, please DO NOT change types of function parameters or return values or struct field definitions.
3) Only generate rewritten function and DO NOT generate any Rust 'use xxx;' statements outside rewritten function.
4) NEVER omit any lines in "Function" even if they remain unchanged.
5) DO NOT omit any variable type annotations.

Prompts for syntax error fixing

Please fix the syntax error called "#ERROR_DESCRIPTION#", the error is reported at site #ERROR_SITE#. Also, do not generate any explanations besides fixed code. If you cannot fix it due to constraints, response with CANNOT_FIX. the key context snippet around the error is: Snippet: #FOCUS_SNIPPET# You need to fix the error in "Snippet" and return the revised snippet. DO NOT return full function.

Prompts for testcase error fixing

Good. Previous rewrite has passed the compiler checks successfully, but it failed to pass test cases provided in the program with following stack traces:
Backtrace: #EXEC_LOG#

Current state of the key function in this error is shown below:
line #START_LINE#: #REWRITTEN_CODE#
Originally this function is: #ORIGINAL_CODE#
The diff of function: #DIFF_LOG#
Can you try to fix this given error without introducing new raw pointers? Your response should only be a revised version of "Current Function". If you think the semantics cannot be rewritten without introducing raw pointers, reply with CANNOT_FIX.