Translating Large-Scale C Repositories to Idiomatic Rust

SAMAN DEHGHAN**, University of Illinois at Urbana-Champaign, USA
TIANRAN SUN*, Shanghai Jiao Tong University, China
TIANXIANG WU, University of Illinois at Urbana-Champaign, USA
ZIHAN LI, University of Illinois at Urbana-Champaign, USA
REYHANEH JABBARVAND, University of Illinois at Urbana-Champaign, USA

Existing C to Rust translation techniques fail to balance quality and scalability: transpilation-based approaches scale to large projects but produce code with poor safety, idiomaticity, and readability. In contrast, LLM-based techniques are prohibitively expensive due to their reliance on frontier models (without which they cannot reliably generate compilable translations), thus limiting scalability. This paper proposes Rustine, a fully automated pipeline for effective and efficient repository-level C to idiomatic safe Rust translation. Evaluating on a diverse set of 23 C programs, ranging from 27 to 13,200 lines of code, Rustine can generate fully compilable Rust code for all and achieve 87% functional equivalence (passing 1,063,099 assertions out of 1,221,192 in test suites with average function and line coverage of 74.7% and 72.2%). Compared to six prior repository-level C to Rust translation techniques, the translations by Rustine are overall safer (fewer raw pointers, pointer arithmetic, and unsafe constructs), more idiomatic (fewer Rust linter violations), and more readable. When the translations cannot pass all tests to fulfill functional equivalence, human developers were able to complete the task in 4.5 hours, on average, using Rustine as debugging support.

1 Introduction

C is one of the most widely used programming languages[1]. However, its permissive memory model and lack of built-in safety mechanisms make it prone to memory safety vulnerabilities [10, 28, 38, 40, 41, 53, 63]. Rust has emerged as a compelling alternative, offering strong memory safety guarantees. Manually migrating large-scale C projects to idiomatic Rust, however, is costly, time-consuming, and error-prone [20, 21, 33, 68, 79]. Automating this translation process at the repository level, while preserving correctness, idiomatic usage, and maintainability, has thus become a highly relevant research challenge. Existing approaches mark important progress in automating C to Rust translation [13, 22, 45, 54, 73, 80]. However, they suffer from the following limitations:

- **Non-Idiomatic Rust Output.** The majority of prior work [22, 45, 80] relies on C2Rust [12] transpiler for initial translation. C2Rust translations frequently mirror the original C code structure and *unsafe* semantics too closely, relying heavily on unsafe blocks [49]. While these techniques improve some safety issues in the initial translation, their translation are not pure, failing to interact with the broader Rust ecosystem (e.g., crates or build tools).
- **Scalability–Quality Tradeoff.** To overcome the previous limitation, recent techniques leverage large language models (LLMs) to translate C code to Rust from scratch [13, 54, 73]. These techniques rely on raw model power to overcome large-scale translation challenges, thereby, only work with frontier models. This can be prohibitively expensive, failing to scale to large projects[2]. For example, Syzygy [54] translated *zopfli* project (2,937 LoC), costing around $800. These techniques also require initial *manual* effort, e.g., manual translation of structs in Syzygy, to help LLMs produce a *compilable* translation.

** equally contributing lead authors
[1]Second rank in TIOBE Index: https://www.tiobe.com/tiobe-index
[2]Such techniques either translated one or two relatively small projects, or a subset of the project.

Authors' Contact Information: Saman Dehghan*, University of Illinois at Urbana-Champaign, USA, samand2@illinois.edu; Tianran Sun*, Shanghai Jiao Tong University, China, seriousss@sjtu.edu.cn; Tianxiang Wu, University of Illinois at Urbana-Champaign, USA, tw50@illinois.edu; Zihan Li, University of Illinois at Urbana-Champaign, USA, zihanl19@illinois.edu; Reyhaneh Jabbarvand, University of Illinois at Urbana-Champaign, USA, reyhaneh@illinois.edu.

- **Insufficient Translation Validation.** Translation bugs are inevitable in LLM-based transpilation [49], and LLM-based techniques struggle to generate even a fully compilable code [8, 32, 55, 61, 65, 74, 81]. Majority of compilation bugs are not just syntactic inaccuracy, but violation of Rust's safety guarantee—strict ownership, lifetime, and borrowing rules. This makes achieving 100% compilability the *bare minimum* validation requirement. The next level of validation is assessing functional equivalence, mostly done through testing in repository-level code translation[3]. Prior work barely reports details of test suites (e.g., number of tests, assertions, and coverage), and test suites of many evaluation subjects have low coverage or tests with no assertions, making them unsuitable to assess functional equivalence of translations.
- **Lack of Comprehensive Evaluation Metrics.** Existing evaluation metrics focus only on a subset of safety aspects, e.g., the number of raw pointer declarations and dereferences or unsafe constructs, failing to assess the presence or absence of essential idiomaticity features in Rust translations, e.g., readability and usage pointer arithmetic/comparison.
- **Limited Error Reporting and Debugging Support.** Existing tools still cannot fully automate the translation of C projects to Rust and therefore require substantial human effort. However, they offer limited actionable diagnostics or tool support, reducing developer productivity when repairing partial translations. Notably, no prior C-to-Rust translation approach has conducted a human study to evaluate the maintainability or usability of the generated translations.

We propose Rustine, an end-to-end pipeline to translate C repositories (application/test code and configuration/build files) to a *maintainable*, *idiomatic*, and *functionally equivalent* Rust version:

- **Idiomatic Rust Translation.** Rustine translates C projects from scratch. With all the translations produced by LLMs, the generated code is more readable and safer, hence better aligned with Rust idiomaticity. Rustine success is not just from using LLM, but also employing sound program analysis techniques to *automatically* (1) refactor the C code that would otherwise confuse the LLM for translation (resolving preprocessor directives, minimizing pointer operations, expanding macros, and maximizing constness); and (2) realize dependencies to guide the translation (caller-callee relationships, references in structures to other types, global state flows, and lifetime dependencies between variables).
- **Cost-effective Repository-level Translation.** Rustine follows a two-tier prompting strategy in leveraging LLMs for code translation: an affordable LLM for the majority of the translation and a reasoning model in case of first model's failure. Rustine considers several automated prompt crafting strategies to optimize the performance of LLMs, including RAG-based API translation and adaptive in-context learning example generation and usage.
- **Validation Beyond Syntactic Check.** Rustine employs test translation and execution to evaluate functional correctness. This entails the existence of test suites with high coverage *and* high-quality assertions. We manually augment the test suite of existing benchmarks with more tests and assertions, and report the functional equivalence based on the percentage of assertions passed, not just test passes. This transparency avoids bias against passing tests with no assertions, which, at best, indicate runtime correctness and not functional equivalence.
- **Comprehensive Evaluation Metrics.** We evaluate different quality aspects of translations using metrics from prior work, as well as new ones such as # pointer arithmetic/comparisons, idomaticity metric, readability metrics (cognitive complexity, Halstead, and SEI Maintenance Index), assertion pass rate, execution time, and cost.
- **Effective Debugging Procedure.** Rustine implements a post-translation debugging. Inspired by the idea of delta debugging [16, 39, 76, 77], it disables all failed assertions except one, to determine the culprits of non-equivalence in the translation corresponding to a specific failure.

[3]Formal methods can assess functional equivalence of a subset of program, but struggle to scale to entire repository [73].

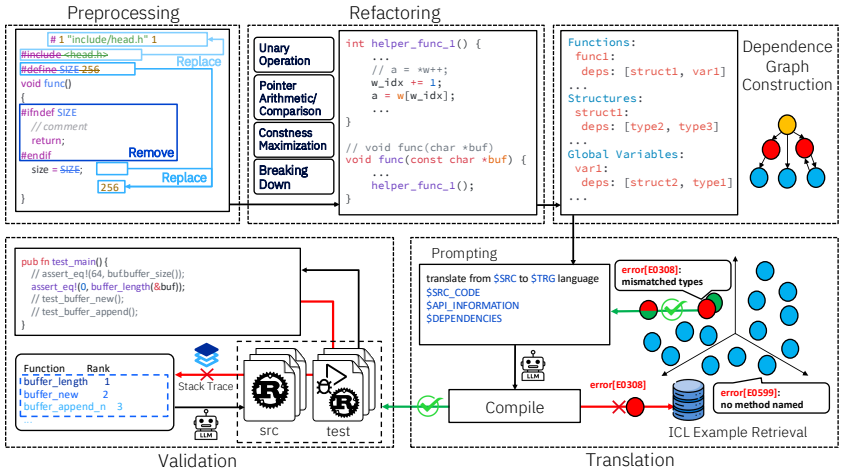

Fig. 1. Rustine framework consisting of five main components

Narrowing down to a few translation units for each assertion enables Rustine to effectively resolve many semantic mismatches, i.e., compilable code resulting in assertion failures.

We evaluated the performance of Rustine in translating 23 C repositories into Rust, in comparison with *six* leading prior approaches. The C repositories range from 27 to 13,200 lines of code, and are diverse in terms of using structs, unions, macros, raw pointer declarations/dereferences, and pointer arithmetic/comparisons. Rustine generates *fully compilable* Rust translations (both application and test code) for *all the projects*, without any human intervention. In overall comparison with alternative approaches, Rustine translations are *safer* (less use of raw pointers and pointer arithmetic/comparison), *more idiomatic* (fewer `Clippy` [2] warnings), and *more readable* concerning three readability and maintainability metrics. For the subjects where translations, after Rustine automated debugging, could not pass all assertions, a human developer resolved semantic mismatches in 4.5 hours, on average, assisted by Rustine debugging support. The final translations are 87% *functionally equivalent* (passing 1,063,099 assertions out of 1,221,192 in test suites of subjects, with average function and line coverage of 74.7% and 72.2%). These notable advancements in automated translation of C repositories to Rust come with a *low time and cost overhead*: on average, it takes 3.92 hours for Rustine to finish the translation, with the cost of $0.48.

Our contributions are: (1) an automated, cost-effective pipeline to translate large-scale C repositories into compilable, functionally equivalent, and idiomatic Rust; (2) the first approach to target and evaluate resolution of pointer arithmetic during translation; (3) comprehensive evaluation metrics to promote more transparent assessment of C to Rust translation; (4) enhancing benchmarking of C to Rust translation by augmenting the test suite of existing benchmarks with more tests and assertions; and (5) a comprehensive analysis of proposed technique compared to notable approaches, that can be used as baseline for future research (we discovered several issues in evaluation of other studied pipelines and recalculated metrics using our pipeline). Our artifacts are publicly available [5].

2 Approach

Rustine follows a five-stage architecture (Figure 1) to progressively refine a C project into idiomatic Rust: *Preprocessing* standardizes the code structure (§2.1), *Refactoring* resolves ambiguous, unsafe, anc challenging constructs in C code (§2.2), *Analysis* realizes intra- and inter-procedural dependencies of the C project for context engineering and determining translation order (§2.3), an

```c
static const unsigned char *
GetMatch(const unsigned char *scan,const unsigned char *match,
 const unsigned char *end,const unsigned char *safe_end) {
 //... omitted code
 } else {
  while (scan < safe_end && *scan == *match &&
   *++scan == *++match && *++scan == *++match &&
   /** omitted code **/) {
      scan++;match++;
    }
  }
  while (scan != end && *scan == *match) {
    scan++; match++;
  }
  return scan;
}
```

(a)

```c
static const unsigned char *
GetMatch(const unsigned char *scan,const unsigned char *match,
 const unsigned char *end,const unsigned char *safe_end) {
 int scan_idx = 0, match_idx = 0;
  //... omitted code
 } else {
  while (&scan[scan_idx] < safe_end &&
  scan[scan_idx] == match[match_idx] &&
  scan[++scan_idx] == match[++match_idx] &&
  scan[++scan_idx] == match[++match_idx] &&
   /** omitted code **/) {
     scan_idx += 1; match_idx += 1;
  }
 }
 while (&scan[scan_idx] != end &&
 scan[scan_idx] == match[match_idx]) {
    scan_idx += 1; match_idx += 1;
 }
 scan = &scan[scan_idx];
 match = &match[match_idx];
 return scan;
```

(b)

```rust
pub fn GetMatch<'a>(scan:_&'a [u8],
match_: &'a_[u8],_end:_&'a [u8], safe_end: &'a_[u8])
->_Option<&'a [u8]> {
 //... omitted code
 } else {
  while (scan_ptr as usize) < (safe_end_ptr as usize) {
   unsafe {
    if *scan_ptr != *match_ptr ||
    *scan_ptr.add(1) != *match_ptr.add(1) ||
    *scan_ptr.add(2) != *match_ptr.add(2) ||
     // ... omitted code {
      break;
    }}}}
while (scan_ptr as usize) < (end_ptr as usize) {
 unsafe {
  if *scan_ptr != *match_ptr { break; }
  scan_ptr = scan_ptr.add(1);
  match_ptr = match_ptr.add(1);
```

(c)

```rust
pub fn GetMatch<'a>(scan:_&'a [u8], match_: &'a_[u8],
end:_&'a [u8], safe_end: &'a_[u8])_->_Option<&'a [u8]> {
  let mut match_idx = 0;let mut scan_idx = 0;
  //... omitted code
  } else {
    while scan_idx < scan.len() && scan_idx < safe_end.len()
    && scan[scan_idx] == match_[match_idx]
    {
      scan_idx += 1;match_idx += 1;
    }
  }
  while scan_idx < scan.len() && scan_idx < end.len()
   && scan[scan_idx] == match_[match_idx] {
     scan_idx += 1; match_idx += 1;
  }
  Some(&scan[..scan_idx])
}
```

(d)

Fig. 2. An illustrative example from *zopfli* project (a), refactored version with no pointer arithmetic (b), and corresponding translations of them (c) and (d)

LLM *translates* application and test code through an iterative feedback loop (§2.4), and *Validation* determines functional equivalence using translated tests, and debugs non-equivalent cases (§2.5).

2.1 Automated Preprocessing of the Source Code

Rustine uses the C preprocessor [26] to insert the contents of header files, substitute macro definitions, and resolve conditional compilation directives, e.g., `#ifdef`, `#ifndef`, and `#if defined()`. This step standardizes the code and allows subsequent stages to focus solely on translating C syntax to Rust. Otherwise, the LLM must expand the macros itself, a process that can involve complex token concatenation [56], stringification [56], and recursive expansions [24], obscuring the underlying code logic. Some macros are context-dependent, i.e., the same macro produces different expansions based on prior definitions. Macros lack access to type definitions and function signatures from header files, making it difficult to generate correct Rust types and function calls.

2.2 Automated Refactoring of C Project

Following preprocessing, Rustine automatically refactors ambiguous or unsafe constructs that, if translated literally, would result in unidiomatic or unsafe Rust code. Specifically, Rustine refactors usage of *unary operators* (§2.2.1) and *pointer arithmetic* (§2.2.2), maximizes constness (§2.2.3), and breaks down large functions that do not fit into the context window of LLMs into smaller segments (§2.2.4). This pre-translation step can significantly increase the likelihood of LLM generating correct, idiomatic Rust code. Figure 2-a shows an example from *zopfoli* project, containing pointer arithmetic/comparisons (highlighted lines). Without refactoring, Rustine results in the unsafe

```
1  double const*
2  genann_run(genann* ann, double* inputs){
3    double const* w = ann->weight;
4    double* o = ann->output + ann->inputs;
5    double const* i = ann->output;
6    // ... omitted code
7    for (j = 0; j < ann->outputs; ++j){
8      double sum = *w++ * -1.0;
9
10     for (k = 0; k < ann->inputs; ++k
11     {
12       sum += *w++ * i[k];
13
14     }
15     *o++ = genann_act_output(ann, sum);
16   }
```

```
1  double const*
2  genann_run(genann* ann, double* inputs){
3    double const* w = ann->weight;
4    double* o = ann->output + ann->inputs;
5    double const* i = ann->output;
6    // ... omitted code
7    for (j=0;j < ann->outputs;j += 1){
8      double sum = *w * -1.0;
9      w += 1;
10     for (k = 0;k < ann->inputs;k += 1)
11     {
12       sum += *w * i[k];
13       w += 1;
14     }
15     *o = genann_act_output(ann, sum);
16     o += 1;
17   }
```

```
1  double const*
2  genann_run(genann* ann, double* inputs){
3    int w_idx = 0;
4    int o_idx = 0;
5    double const* w = ann->weight;
6    double* o = ann->output + ann->inputs;
7    double const* i = ann->output;
8    // ... omitted code
9    for (j = 0; j < ann->outputs; j += 1){
10     double sum = w[w_idx] * -1.0;
11     w_idx += 1;
12     for (k = 0; k < ann->inputs;k+=1){
13       sum += w[w_idx] * i[k];
14       w_idx += 1;
15     }
16     o[o_idx]=genann_act_output(ann,sum);
17     o_idx += 1;
```

Fig. 3. Refactoring of the *genann* project for resolving unary (middle) and pointer arithmetic (right) operations

translation of Figure 2-c, which contains unsafe lines and several raw pointer arithmetic. Figure 2-b shows a refactored version, which LLM can translate into a more idiomatic Rust (Figure 2-d). As a proxy for checking whether refactoring breaks functionality, Rustine requires a successful post-refactoring test execution. Otherwise, it reverts the refactoring.

2.2.1 Refactoring Unary Operators. Automated translation of unary operators is challenging since Rust requires explicit modification statements for clarity and error prevention. If done by an LLM, the translation may not consider the evaluation order and result in a semantic mismatch. Rustine detects all pre- and post-increment/decrement operators. If the unary operator is a standalone expression, e.g., `*var++;`, Rustine desugars it to a compound assignment, e.g., `var+=1`. Otherwise, it separates the expression into one compound assignment statement and one statement for the rest of the expression, preserving the order. When there are multiple unary operations in one statement, Rustine skips refactoring to avoid semantic alternation. Line 7 of Figure 2-a shows such a case, where increments should only occur if the previous conditions succeed.

2.2.2 Refactoring Pointer Operations. Pointer operations (arithmetic/comparisons) enable efficient memory manipulation in C, but sacrifice safety (buffer overflows, dangling pointers, and other memory safety violations [7]) and readability. In contrast, Rust's ownership model prohibits pointer arithmetic or comparison [9], requiring the LLM to find an equivalent safe translation. Such operations can be complex in C programs, making their translation a recipe for disaster. To alleviate this, *following the refactoring of the unary operators*, Rustine identifies the usages of pointer arithmetic/comparisons and transforms them into array-style indexing operations, using Algorithm 1.

Algorithm 1: Pointer Operation Refactoring

Input: Function F
Output: Modified F with pointer arithmetic converted to array indexing

```
1  all_pointers ← LocatePtrs(F);
2  possible_pointers ← ∅;
3  foreach pointer p ∈ all_pointers do
4      if HasArithmCompOperations(p) then
5          possible_pointers ← possible_pointers ∪ {p};
6          name ← p.name + "_idx";
7          decl ← CreateDeclaration("int", name, 0);
8          DeclareBeforScope(F, p, decl);
9          if AccessContents(p) then
10             ReplaceWithIndex(F, p);
11         else if PtrUsed(p) then
12             ReplaceWithPtrAccessWithIndex(F, p);
13         else if PtrReInitialized(p) then
14             ResetIndex(F, p);
15 foreach p ∈ possible_pointers do
16     if PtrTransformed(p)&PtrIsArgument(p) then
17         UpdatePtrBeforeReturn(F, p);
18 return F;
```

Rustine starts by traversing the function's Abstract Syntax Tree (AST) to locate all pointers within the scope of the function (declared or passed) that are involved in arithmetic/comparison operations, e.g., addition or subtraction. For each such pointer, a corresponding integer index variable is declared at the beginning of the pointer declaration scope. This index variable tracks the pointer's position within the underlying array or memory block. For instance, for `double const* w` in *genann* project (Figure 3), which undergoes

arithmetic operations inside the loop, an integer variable `int w_idx = 0` is declared at the beginning of the function. RUSTINE transforms the identified pointers depending on the type of pointer arithmetic: when pointers are used to access memory contents, RUSTINE converts them into array indexing operations using the base array and the associated index variable (`double sum = *w * -1.0` in Figure 3 is transformed to `double sum = w[w_idx] * -1.0`); when arithmetic operations are performed directly on pointers, RUSTINE applies index transformation on pointer and adds `&` to access literal value of pointer (`scan != end` in Figure 2 transforms to `&scan[scan_idx] != end`); when pointers are reintialized within the code, RUSTINE resets their indices after re-initialization statements in the same scope.

For *alias-safe* transformations, RUSTINE updates all transformed pointers that were passed to the function as an argument (regardless of them escaping or not). This ensures that changes in pointers are reflected outside of the function scope. In the examples of Figure 2-a, two pointers `scan` and `match` are *aliased* outside of the `GetMatch` scope. Both these pointers are transformed, and although they are passed by value, RUSTINE updates them to ensure usage of them outside of `GetMatch` is not broken. This also ensures that, if only one of the aliased pointers is transformed, the semantics are not broken. In the example of Figure 3, `i` and `o` point to `output` (although to non-overlapping sections). RUSTINE only refactors `o += 1`, and since `i` is only accessed by index `sum += *w * i[k]`, the semantics are preserved. In addition to post-refactoring test execution, we also prove RUSTINE pointer operation transformations are semantic preserving (details in Appendix).

Algorithm 1 is *bound-agnostic*, eliminating the need to specify object bounds explicitly, assuming safe pointer arithmetic. If this assumption does not hold, the translated Rust is either non-compilable or be corrected by the LLM into a safe implementation.

2.2.3 Constness Maximization. RUSTINE analyzes data flow to identify opportunities for maximizing constness in function parameters that are never modified within their scope, and transforms them by adding `const` qualifiers to their declarations. For example, `char *buffer` and `int` are transformed to `const char *buffer` and `const int`, respectively. The modified C code with maximized constness directly guides the LLM to generate appropriate Rust constructs: `const` parameters translate naturally to immutable references (`&T`) or owned immutable values, while non-const parameters indicate the need for mutable references (`&mut T`) or owned mutable values.

2.2.4 Large Function Breakdown. Large functions may exceed the LLM context window, or even if not, the LLM may struggle to maintain attention across the entire function scope, leading to suboptimal translation quality [36]. Furthermore, in case of inevitable translation bugs, the LLM is more likely to succeed in repair by focusing on smaller code rather than a large one. To account for this, RUSTINE employs a flow-sensitive analysis to break down large functions—those larger than 50% of LLM context window—prior to translation, using Algorithm 2.

Algorithm 2: Large Function Breakdown

Inputs: Function F, Breakdown threshold θ_f, Block threshold θ_b
Output: Updated code with breakdown

if $|F| < \theta_f$ **then**
  return F
candidates, *helpers* $\leftarrow \emptyset$;
foreach B *in* F **do**
  if $|B| > \theta_b$ ***and*** $\neg$HASEARLYEXIT(B) **then**
    candidates $\leftarrow$ *candidates* $\cup \{B\}$;
foreach $B \in$ *candidates* **do**
  $V_{read}, V_{write} \leftarrow$ EXTRACTREADWRITEVARIABLES(B);
  $B.dependencies \leftarrow V_{read} \cup V_{write}$;
  name $\leftarrow$ "helper_" + $F.name$ + "_" + $B.identifier$;
  params $\leftarrow \emptyset$;
  foreach $v \in B.dependencies$ **do**
    if $v \in V_{write}$ **then**
      params $\leftarrow$ *params* $\cup \{v^*\}$ `// Pass by reference`
    else
      params $\leftarrow$ *params* $\cup \{v\}$ `// Pass by value`
  $H \leftarrow$ CREATEFUNCTION(*name*, *params*, void, $B.body$);
  helpers $\leftarrow$ *helpers* $\cup \{H\}$;
UPDATECODE(F, *helpers*);

For a given function F in C program that is larger than a breakdown threshold θ_f, RUSTINE determines basic blocks within control flow structures, i.e., `if-else` statements, `while`/`for` loops, and `switch`/`case`. The blocks

larger than threshold θ_b (to avoid excessive breakdown) with no early exit points, e.g., `return`, `goto`, or `break` statements are selected for breakdown. RUSTINE moves candidate blocks from the previous step to new helper functions, and replaces each with a functional call. The return type of new helper functions is `void` and the input parameters are all variables accessed or modified within the block, passed by reference where necessary.

2.3 Dependence Graph Construction

RUSTINE performs a context-sensitive static analysis to construct a Dependence Graph (DG) for C projects. Such a data structure helps RUSTINE track caller-callee relationships, references in structures to other types, global state flows, and lifetime dependencies between variables. RUSTINE will use DG for three purposes: (1) to construct a project skeleton, (2) to guide the translation order, and (3) to provide a rich context to guide the LLM with translation (§2.4).

Definition 1. Dependence Graph (DG) is a directed graph $DG = (V := F \uplus S \uplus U \uplus G \uplus T, E)$, reflecting dependencies among program components ($v \in V$), including functions (F), structs (S), unions (U), global variables (G), and type definitions/enums (T). A directed edge $e_{i,j} = (v_i, v_j) \in E$ demonstrates an *explicit* use-def or caller-callee relationship between v_i and v_j.

Figure 4 shows a DG snapshot for *genann* project, corresponding to Figure 3. Cyclic DGs are inevitable in large C codebases due to the common use of mutual recursion, forward declarations, and complex header inclusion patterns. After construction of the initial DG, RUSTINE detects strongly connected nodes (using Tarjan's algorithm [57]) and condenses each into a *super translation unit*. This enables LLM to generate appropriate Rust constructs such as mutually recursive data structures or carefully designed module hierarchies.

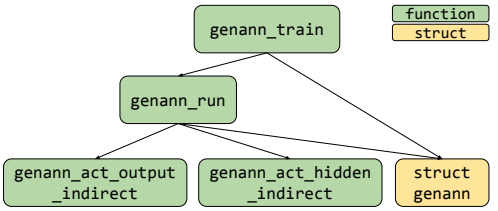

Fig. 4. DG snapshot of Figure 3

RUSTINE creates a compilable, memory-safe Rust project skeleton to support incremental translation. The structure includes a `src/` directory with `*.rs` files, aligning C units to Rust modules. The files contain placeholder functions with *incomplete* Rust signatures (but with a complete commented signature in C for the context) and the `todo!()` body. RUSTINE also consolidates global variables in `src/globals.rs` for maintainability, and identifies main functions, generating [bin] sections in `Cargo.toml`. Each struct/union will be placed in its own `[struct_name].rs` module for clarity.

2.3.1 Enhancing Cross-language API Translation. An important consideration in real-world code translation is ensuring accurate translation of API calls (core libraries or external dependencies), which requires understanding the semantics of the original invocation and finding the equivalence in the target programming language, if any. This makes API calls in the source language essential dependency points that must be explicitly identified and analyzed to support translation.

APIs invocations either belong to core libraries, e.g., `std::atof` and `isatty`, or external libraries, e.g., `X11` [70] and `ncurses` [25]. RUSTINE does not consider API invocation as a separate node in DG, but augments function node F with invoked APIs listed as dependencies, accompanied by *suggestions* about how to handle them during translation. To that end, RUSTINE first locates all the API invocations. For core libraries, RUSTINE retrieves API description and synopsis from Linux documentation [58], helping the model better understand semantics of the API to find its equivalent in Rust, or write the semantics itself if no equivalent Rust API is found. For external libraries, RUSTINE automatically generates Foreign Function Interface (FFI) using bindgen [59], and provides the corresponding FFI API as a suggestion during translation.

In practice, such analysis proved to be useful for accurate API translation, as we will show in §3: most C core libraries now have an equivalent in Rust, and the provided context helped RUSTINE using the equivalent Rust API in the translation. For example, RUSTINE used `f64::abs` Rust API as an equivalent to `std::fabs` in C. RUSTINE used FFI only in one case, where the library had no exact equivalence in Rust (`X11` library in *xzoom* and *grabc* projects). For other cases, RUSTINE used equivalent API or corresponding wrappers in Rust, e.g., `ncurses-rs` crate [67] for `ncurses` in C.

2.4 Translation

RUSTINE translates a project in the reverse topological order of the DG, using algorithm 3. For each translation unit, it starts by prompting a general LLM. The initial prompt contains a translation unit in C, a fixed in-context learning (ICL) [11] example (to illustrate an idiomatic Rust translation of a sample C code to model), language-level hint rules, dependencies (e.g., global variables, used structs, function signatures of translated callees), and the context related to invoked C libraries as discussed in §2.3.1. RUSTINE re-prompts the general LLM with a maximum budget of A_{base}, providing compilation errors as feedback to the model, to generate a compilable Rust translation.

Algorithm 3: RUSTINE Translation Loop

Inputs : Translation Unit TU, Base Model M, Reasoning Model R, Max Attempts A_{max}, Base Attempts A_{base}, Reasoning Attempts A_{reason}, ICL Pool $\mathcal{I}$
Output: Rust Translation T

```
1  T, compOut ← ∅;
2  for i ← 1 to A_max do
3      for k ← 1 to A_base do
4          T, compOut ← Translate(M, TU, compOut, ICL_f);
5          if IsCompilable(compOut) then
6              return T;
7      for k ← 1 to A_reason do
8          ICL_TU ← RetrieveGenerateICL(I, TU, compOut);
9          T, compOut ← Translate(R, TU, compOut, ICL_TU);
10         if IsCompilable(compOut) then
11             return T;
12 return T;
13 Function Translate(LLM, TU, compOut, ICL)
14     prompt ← GeneratePrompt(TU, compOut, ICL);
15     T ← TranslateUnit(LLM, prompt);
16     compOut ← CompilationCheck(T);
17     return T, compOut;
```

State-of-the-art LLMs can reliably produce syntactically correct Rust code. Most compilation errors, hence, stem from violations of Rust's strict safety rules, particularly ownership and borrowing. Additional iterations, even when accompanied by error feedback, may not be helpful to resolve such complexities. Thereby, when reaching the A_{base} budget, RUSTINE switches to a reasoning model to analyze the nature of errors. RUSTINE helps the model with *adaptive* ICL examples [35, 52, 69] (§2.4.1).

If the reasoning model also cannot generate a compilable code, RUSTINE continues the translation loop with the latest feedback from the reasoning model. This design decision is motivated by a cost-quality trade-off: reasoning models are slower and more expensive, and using them multiple times increases the total runtime and cost. Reasoning models are also likely to resolve many major issues in the translation, leaving minor issues for the general model to resolve. Translation terminates after A_{max} iterations, or when RUSTINE generates a compilable translation.

2.4.1 Adaptive ICL Generation and Retrieval. In-context learning has been shown to effectively instruct LLMs with new tasks or help them solve challenging problems within known tasks. RUSTINE uses a fixed ICL example for initial prompting to instruct the LLM with idiomatic Rust translation. Rust errors are diverse, i.e., 500 official errors [4], with some very complex, requiring more help for LLM than compilation feedback to resolve them. In the presence of persistent compilation errors, RUSTINE switches to *adaptive* ICL examples. Adaptivity means that, for each specific combination of translation unit and error message, RUSTINE retrieves (from an ICL pool) or generates a similar ICL example to include in the prompt as a context.

The ICL pool of RUSTINE is indexed by Rust error codes [4], e.g., `E0308` (type mismatch). When the ICL pool has examples corresponding to an error code, RUSTINE retrieves one of the existing

```rust
pub fn run_ti_ref(info: &TiIndicatorInfo,
                  options: &[f64], goal: i32) -> i32 {
  // ...
  let mut outref = OUTREF.lock().unwrap();
  let row0 = &mut outref[0][..];
  let row1 = &mut outref[1][..];
  let row2 = &mut outref[2][..];

  let outputs_ref: &mut [&mut [f64]]
      = &mut [row0, row1, row2];
  let ret = (info.indicator_ref)(
    4000, Some(inputs_ref), Some(options), Some(outputs_ref)
  );
}
```

(a)

```
error[E0499]: cannot borrow `outref` as mutable more than once
at a time
   --> src/benchmark.rs:6:19
    |
5   | let row0 = &mut outref[0][..];
    |                ------ first mutable borrow occurs here
6   | let row1 = &mut outref[1][..];
    |                ^^^^^^ second mutable borrow occurs here
8   | let outputs_ref: &mut [&mut [f64]] = &mut [row0, row1, row2];
    |                                      ---- first borrow
    | later used here
```

(b)

```rust
fn process_data(
  mut data: [[f64; 100]; 3]
) {
  let outputs = &mut [
    &mut data[0][..],
    &mut data[1][..],
    &mut data[2][..],
  ];
  some_function(outputs);
}
```

(c)

```
error[E0499]: cannot borrow `data` as mutable more than once at a time
   --> src/main.rs:5:14
    |
4   |  let outputs = &mut [
    |  _____________-
5   | |  &mut data[0][..],
    | |       ---- first mutable borrow occurs here
6   | |  &mut data[1][..],
    | |       ^^^^ second mutable borrow occurs here
7   | |  &mut data[2][..],
8   | |  ];
    | |_____- first borrow later used here
```

(d)

```rust
fn process_data(mut data: [[f64; 100]; 3]) {
  let (first, rest) = data.split_at_mut(1);
  let (second, third) = rest.split_at_mut(1);
  let outputs = &mut [&mut first[0][..], &mut second[0][..], &mut third[0][..]];
  some_function(outputs);
}
```

(e)

```rust
pub fn run_ti_ref(info: &TiIndicatorInfo,
                  options: &[f64],
                  goal: i32
) -> i32 {
  // ...
  let mut outref = OUTREF.lock().unwrap();
  let rows = &mut *outref;
  let (first, rest) = rows.split_at_mut(1);
  let (second, third) = rest.split_at_mut(1);
  let row0 = &mut first[0][..];
  let row1 = &mut second[0][..];
  let row2 = &mut third[0][..];
  let outputs_ref: &mut [&mut [f64]]
    = &mut [row0, row1, row2];
  let ret = (info.indicator_ref)(
    4000, Some(inputs_ref),
    Some(options), Some(outputs_ref)
  );
}
```

(f)

Fig. 5. An example of in-context learning from *tulpindicator* project. (a) buggy translation; (b) error message; (c, d) ICL example: buggy code; (e) ICL example: fixed code; (f) compilable Rust translation

examples based on embedding similarity. If no example exists or existing examples are not similar, RUSTINE prompts OpenAI o4-mini with the (1) buggy Rust code and (2) the corresponding error message to create a *similar, yet simple* Rust code with the same issue, as well as a fix to it. Figure 5 shows an example from *tulpindicator* project, where RUSTINE produced an incorrect translation after A_{base} attempts. The error stems from creating multiple mutable subslices of the same buffer, `outref`, within one scope, which leads to overlapping mutable borrows and triggers error `E0499`. The ICL example abstracts this buggy behavior on a two-dimensional array and offers an effective fix: it first partitions the slice using `split_at_mut` to obtain disjoint chunks, then derives each mutable reference from those chunks. Guided by the ICL example, RUSTINE implements the same transformation to `outref` and produces three non-overlapping mutable slices.

2.5 Validation and Debugging

RUSTINE leverages test translation and execution for validation, and to determine an *upper bound* for functional equivalence: after completing the translation of the entire project, RUSTINE executes the translated tests. Even when the translation is compilable and safe, it may not be semantically equivalent to the source code. To account for post-translation validation issues, specifically test execution failures, RUSTINE implements an automated debugging process.

When a failure occurs, RUSTINE temporarily disables the failed assertions (corresponding to the reported panics) or the test (in case of other runtime errors or the absence of assertions) to complete test execution. It then enables failed assertions/tests, one by one, *from the beginning*, and identifies the top n functions on the stack trace as potential culprits. If a function in this shortlist is a *focal method* examined by a *passing assertion* in the test suite, RUSTINE removes it from the shortlist and adds the next function in the stack trace to the shortlist. This exclusion avoids re-translating functions that have already demonstrated correct behavior under other test scenarios, following established principles of test-based behavioral consistency [50].

Next, Rustine prompts a (reasoning) LLM to fix the semantic mismatch of the culprits, providing C source, Rust translation, and `stderr` context. If the result of re-prompting is not compilable, Rustine reverts changes and disables the failing assertion again. Otherwise, it re-executes the test suite. If the re-execution results in a corresponding test/assertion pass without failing prior passing tests/assertions, Rustine considers the semantic mismatch resolved.

The debugging approach is not perfect due to heuristics and looking at a subset of potential culprits, but it is lightweight in terms of time and cost. When automated debugging cannot fix a semantic mismatch (due to the limitations of LLM or incompleteness of heuristics), the debugging process can help a human expert in the loop to rescue: the human will see the full stack trace and re-translation, but use their expertise and reasoning to locate and fix the semantic mismatch.

3 Empirical Evaluation

We evaluate contributions of Rustine concerning the following research questions:

- **RQ1.** *Effectiveness of Rustine and Comparison with Alternative Approaches.* To what extent can Rustine facilitate translation of the real-world C projects to idiomatic Rust? How much effort is required by a human developer to fix semantic mismatches in the translations? How does the performance of Rustine in code translation compare with other techniques?
- **RQ2.** *Effectiveness of the Refactoring.* To what extent refactoring is successful in resolving pointer arithmetic in Rustine translations?
- **RQ3.** *Translation Bugs.* What are the most common translation issues produced by Rustine?
- **RQ4.** *Performance and Cost.* How long does it take for Rustine to translate subject projects from C to Rust? How much does it cost to use Rustine for translation?

3.1 Experimental Setup

3.1.1 Alternative Approaches and Subject Programs. We compare Rustine with leading prior work that attempted repository-level translation of C projects to Rust. Given new evaluation metrics that were not considered by prior work, but are necessary for proper evaluation of translation quality (§3.1.3), we focused on the tools that either made their translations publicly available or we were able to run them with no runtime exception to generate translations[4]. Considering these criteria, we compare Rustine with the following techniques (alphabetically ordered):

C2Rust [12] is a rule-based transpiler that converts C code into Rust while preserving semantics. **C2SaferRust** [45] improves the *idiomaticity* of C2Rust translations by breaking down the translations by C2Rust into smaller translation units, and prompting GPT-4o mini with static analysis-provided contexts to improve the translation. **CROWN** [80] builds on top of C2Rust and leverages ownership analysis to resolve unsafe usages in the translation. **Laertes** [22] also relies on C2Rust for initial Rust translation and lifts raw pointers into safe Rust references through an iterative, compiler-in-the-loop approach.

RustMamp [13] and **Syzygy** [54] perform a neurosymbolic compositional approach [30] to translate C projects into idiomatic Rust *from scratch*. Both techniques leverage GPT-4o for translation and, due to the high cost, only translate one (RustMamp) and two (Syzygy) repositories.

Table 1 shows the list of 23 programs from the evaluation of the prior technique that met our subject selection criteria. The programs range from 27 to 13,200 lines of code, and are diverse in terms of using structs, unions, macros, and raw pointer declarations/dereferences, with 17 and 22 of them containing 452 and 2,636 instances of pointer operations and macros, respectively[5]. Another

[4]We did not include techniques such as Fluorine [23] or VERT [73], since they only translate a few program slices of real-world projects.

[5]We did not compare with CRUST-Bench [32] as it evaluates the power of specific LLMs rather than the pipeline.

Table 1. Subject programs and corresponding properties. Subjects are sorted by size (LoC).

ID	Subject	LoC	# Files	# Functions	# Structs	# Unions	# Global Variables	# Macros		# Pointer Arithmetic		# Raw Pointers		Tests		
														% Coverage		# Tests (Assertions)
								Config	Function	Before	After	Declarations	Dereferences	Func	Line	
1	qsort	27	1	3	0	0	0	0	0	0	0	2	4	100	100	5(21)
2	bst	65	1	5	3	0	0	2	0	0	0	9	0	100	100	6(6)
3	rgba	411	3	19	1	0	1	12	1	20	0	14	61	92	85	5(20)
4	quadtree	437	7	36	8	0	2	17	1	0	0	55	7	100	95	4(34)
5	buffer	452	3	42	1	0	0	17	2	0	0	61	1	86	83	17(54)
6	grabc	490	1	10	0	0	11	24	0	1	0	19	15	33	18	3(2)
7	urlparser	563	5	27	3	0	5	23	3	17	0	82	49	83	80	1(46)
8	xzoom	659	1	8	0	0	32	48	2	40	0	13	24	–	–	–
9	genann	690	8	30	2	0	12	51	10	28	4	62	93	96	94	9(521556)
10	ht	699	10	23	4	0	3	50	0	7	0	68	28	60	56	8(1)
11	robotfindskitten	838	3	17	1	0	8	49	5	0	0	1	0	53	63	7(47)
12	libcsv	1035	7	31	2	0	5	64	5	5	0	65	8	75	92	45(7406)
13	avl-tree	1170	7	50	6	0	3	41	7	0	0	122	4	40	45	11(12)
14	libopenaptx	1333	4	44	15	0	41	29	3	2	0	65	20	95	84	4(9)
15	libtree	1412	4	30	17	0	1	69	0	17	3	91	31	100	64	57(121)
16	opl	1642	2	117	20	0	7	23	4	7	0	148	28	63	56	10(14)
17	libzahl	2575	52	59	0	0	35	98	28	26	0	15	43	98	93	13(1570)
18	zopfli	2937	26	105	10	0	1	254	1	26	0	289	859	95	88	11(40)
19	snudown	5271	19	153	89	0	23	138	18	5	0	441	44	35	30	–
20	lodepng	5606	6	235	19	0	16	168	9	31	0	543	222	78	80	28(683994)
21	bzip2	5861	15	118	8	0	36	251	83	10	0	222	129	10	6	1(36)
22	binn	6361	10	315	3	1	18	188	16	74	0	625	220	65	87	14(1949)
23	tulpindicator	13200	152	326	9	0	32	673	149	136	7	1073	155	85	90	9(4252)

3.1.2 Choice of LLM. To balance cost and effectiveness, we use two LLMs in RUSTINE pipeline: DeepSeek-V3 for most of the translation. For cases that DeepSeek-V3 fails in translation due to compilation errors after *eight* re-prompting, RUSTINE uses DeepSeek-R1, a reasoning model, to better analyze the failure reasons with a re-prompting budget of *two*. Users can configure RUSTINE to use any LLM depending on their budget by replacing the API key[6].

3.1.3 Metrics. We measure the following metrics for both C and Rust code. We identified several bugs or incorrect assumptions in calculation of some metrics by existing techniques (marked with †) and report recalculated numbers using our pipeline.

- **# Raw Pointer Declarations/Dereferences**†[7]**.** For raw pointer declaration in C, RUSTINE uses `pycparser` [51] to find `Decl` nodes in AST with `PtrDecl` type. It also looks for `UnaryOp` nodes with the `*` operator for determining pointer dereferences. For Rust, RUSTINE uses (`rustc_hir` crate [17]) to identify nodes corresponding to raw pointer types, e.g., `*const T` and `*mut T`, declared within variable bindings, function signatures, and type annotations. For dereference, RUSTINE locates `ExprKind::Unary` expressions with `Deref` as operator and raw pointer operand.
- **# Pointer Operations.** None of the prior work addresses or measures pointer arithmetic/comparisons in translations. RUSTINE uses `rustc_hir` crate to locates function calls on pointer types (`ExprKind::MethodCall`) that correspond to arithmetic operations, e.g., `add`, and `byte_add` [3].
- **# Unsafe Constructs**†[8]**.** RUSTINE uses Rust compiler (`rustc_hir` crate [17]) to calculate the number of unsafe *lines*, *type casts*, and *calls*.
- **Idiomaticity**[9]**.** RUSTINE uses *Clippy* [2] for measuring idimaticity. RUSTINE excludes `nursery` and `restriction` as they either concern new lints that are still under development or may have false positives. `Clippy` document also suggests against using `restriction` by default due to the subjectiveness of the lints. Except for `clippy::style` that will be used to measure readability, RUSTINE computes the total number of raised flags as $Clippy_{flags} = \sum \texttt{clippy::flag}_i$, where

[6]API- or open-access models hosted using Ollama [46]/vLLM [62] and exposed via an API key-protected endpoint.
[7]Some pipelines used a regex-based approach to calculate these metrics, making them unreliable and error-prone.
[8]Similar to previous metrics, some techniques followed an incomplete regex-based approach for counting unsafe constructs. While not using regex, C2SaferRust does not support newer versions of Rust (1.81 and above) and cannot calculate these metrics inside struct methods or within complex statements such as `sum+=unsafe {*w} * unsafe {*i.add(k as usize)}`. It marks the entire functions as unsafe rather than using unsafe blocks for specific statements, reporting *fewer* unsafe issues.
[9]C2Rust default pipeline suppresses `Clippy` warnings. We updated the config of transpiler-based techniques to ensure proper calculation of warnings.

$i \in \{$`cargo`, `complexity`, `correctness`, `perf`, `suspicious`$\}$. Transpiler-based techniques can increase code size 1.8× when converting C to Rust [22]. Without normalizing the number of Clippy warnings concerning inflation size, aggregating the number of raised flags favors unnecessary verbosity, redundancy, or boilerplate [49]. Rustine measures Translation Size Inflation[10] as $TSI = {}^{LoC_{Rust}}/_{LoC_C}$ and reports idiomaticity as $TSI \times Clippy_{flags}$ per KLoC.

- **Readability.** Rustine uses rust-code-analysis [1, 6] and reports *Cognitive Complexity*, *Halstead* [19], and *SEI Maintainability Index* [47], as well as the number of `clippy::style` violations. These metrics highlight different aspects of readability, making the evaluation of translation more transparent. Except for the *SEI*, the lower number for each metric indicates more readability.
- **Assertion Pass Rate.** We augmented test suites of subjects with more tests and assertions, resulting in average function and line coverage of 74.7% and 72.2%. They include 1,221,192 assertions and cover 93.3% of the refactored code, on average. Rustine reports the number of executed, passed, and failed assertions in the Rust translation.
- **Execution Time and Cost.** Rustine reports the time and cost of translating projects. It also presents the number of tokens involved in translating each project[11], which practitioners can use to approximate the cost of using commercial models with Rustine (§3.5).

3.2 RQ1. Effectiveness of Rustine and Comparison with Alternative Approaches

Rustine successfully translates all the programs and tests, achieving 100% **application and test code compilation success** (gray rows under column *Compilation Success* of Table 2). C2Rust, CROWN, and Laertes generated a non-compilable application code for *libzahl* (ID=17). CROWN also resulted in a non-compilable application and test code in Rust for *zopfli* (ID=18). C2SaferRust test code translation for *urlparser* (ID=7) and *gennan* (ID=9) was non-compilable. C2Rust, C2SaferRust, CROWN, and Laertes all generated non-compilable test code translation for *tulipindicator* (ID=23). We found no test translation by Syzygy for *zopfli* (ID=18). The execution of translated tests by RustMamp also stuck in a loop and never terminates for *bzip2* (ID=21). These cases are marked with "-" in Table 2. The 100% compilation success of Rustine is achieved through **full automation, without any human intervention** as in Syzygy and RustMamp.

Table 2 shows the validation results of Rustine, compared to alternative techniques. Overall, Rustine resulted in 100% functional equivalence in translation of 10 projects **without any human intervention**. That is, executing the translated tests passed all the assertions. We mark the ID of those projects with ∗ in Table 2. For the remaining subjects, with at least one assertion failure, we asked a developer with more than ten years of relevant industry experience—proficiency in C and Rust—and no familiarity[12] with the subjects to repair the translations within a budget of 10 hours per project. The developer used Rustine debugging to facilitate the process.

The human developer achieved 100% assertion pass rate on 11 remaining projects, within 4.5 hours on average (min=0.33 and max=10 hours). For *tulipindicator* (ID=23) and *lodepng* (ID=20), the assertion pass rate after manual effort was 94% and 76%, respectively. Figure 6 shows the effort of the human developer. The Spearman Rank Order Correlation [75] shows that there is a strong correlation between the size of the project and the required effort to fix semantic mismatches in translations ($\rho = 0.86$ with strong statistical significance, p-value= $1.59e-4 < 0.05$). The human developer reported the provided debugging information by Rustine **useful to pinpoint the semantic mismatch**. They also noted **the nature of the semantic mismatch** as an important factor in the amount of debugging and effort, in addition to the size.

[10] Rustine relies on Tokei [60] to measure LoC, which does not include blank lines and comments when measuring LoC.
[11] The numbers reported for `usage.prompt_tokens` and `usage.completion_tokens` properties by the LLM API.
[12] No familiarity with projects determines an upper bound for the amount of time required to fix semantic mismatches in Rustine translations, compared to the developers of these projects.

Table 2. Validation of translations by Rustine and alternative approaches. Abbreviations: C2Rust (C2R), C2SaferRust (C2SR), CROWN (C), Laertes (L), RustMamp (R), and Syzygy (S).

ID	Tool	% Compilation Success	% Coverage		# Assertions		
			Function	Line	Executed	Passed	Failed
1*	Rustine	100	100	92	21	21	0
	⟨ C2R,C2SR,C,L ⟩	⟨ 100,100,100,100 ⟩	⟨ 100,100,100,100 ⟩	⟨ 100,−,100,100 ⟩	⟨ 10,−,10,10 ⟩	⟨ 10,−,10,10 ⟩	⟨ 0,−,0,0 ⟩
2*	Rustine	100	92	95	6	6	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 87,85,74 ⟩	⟨ 73,70,63 ⟩	⟨ 6,6,6 ⟩	⟨ 6,6,6 ⟩	⟨ 0,0,0 ⟩
3*	Rustine	100	99	93	20	20	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 83,83,23 ⟩	⟨ 62,69,22 ⟩	⟨ 20,20,20 ⟩	⟨ 20,20,18 ⟩	⟨ 0,0,2 ⟩
4*	Rustine	100	91	83	34	34	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 81,76,27 ⟩	⟨ 78,63,48 ⟩	⟨ 34,34,34 ⟩	⟨ 34,34,34 ⟩	⟨ 0,0,0 ⟩
5	Rustine	100	91	88	54	54	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 87,92,30 ⟩	⟨ 57,66,42 ⟩	⟨ 54,54,54 ⟩	⟨ 54,50,50 ⟩	⟨ 0,4,4 ⟩
6*	Rustine	100	11	10	4	4	0
	⟨ C2R,C2SR,C,L ⟩	⟨ 100,100,100,100 ⟩	⟨ 10,−,10,10 ⟩	⟨ 11,−,10,9 ⟩	⟨ 4,−,4,4 ⟩	⟨ 4,−,4,4 ⟩	⟨ 0,−,0,0 ⟩
7	Rustine	100	75	74	46	46	0
	⟨ C2R,C2SR,C,L ⟩	⟨ 100,100,100,100 ⟩	⟨ 61,−,60,60 ⟩	⟨ 41,−,39,35 ⟩	⟨ 46,−,46,46 ⟩	⟨ 46,−,46,46 ⟩	⟨ 0,−,0,0 ⟩
8*	Rustine	100	–	–	–	–	–
	⟨ C2R,C2SR,C,L ⟩	⟨ 100,100,100,100 ⟩	–	–	–	–	–
9*	Rustine	100	84	79	521556	521556	0
	⟨ C2R,C2SR,C,L ⟩	⟨ 100,100,100,100 ⟩	⟨ 83,−,81,81 ⟩	⟨ 68,−,67,68 ⟩	⟨ 521556,−,521556,521556 ⟩	⟨ 521556,−,521556,521556 ⟩	⟨ 0,−,0,0 ⟩
10*	Rustine	100	61	67	1	1	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 55,56,54 ⟩	⟨ 40,41,39 ⟩	⟨ 1,1,1 ⟩	⟨ 1,1,1 ⟩	⟨ 0,0,0 ⟩
11	Rustine	100	63	61	47	47	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 51,49,43 ⟩	⟨ 59,55,51 ⟩	⟨ 47,47,47 ⟩	⟨ 47,47,47 ⟩	⟨ 0,0,0 ⟩
12*	Rustine	100	45	52	7406	7406	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 48,45,47 ⟩	⟨ 84,79,77 ⟩	⟨ 7406,7406,7406 ⟩	⟨ 7406,7406,7406 ⟩	⟨ 0,0,0 ⟩
13	Rustine	100	29	31	12	12	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 35,32,33 ⟩	⟨ 33,31,32 ⟩	⟨ 12,12,12 ⟩	⟨ 10,10,10 ⟩	⟨ 2,2,2 ⟩
14	Rustine	100	95	81	9	9	0
	⟨ C2R,C ⟩	⟨ 100,100 ⟩	⟨ 81,75 ⟩	⟨ 87,74 ⟩	⟨ 9,9 ⟩	⟨ 9,9 ⟩	⟨ 0,0 ⟩
15	Rustine	100	82	75	121	121	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 90,89,90 ⟩	⟨ 61,60,61 ⟩	⟨ 121,121,121 ⟩	⟨ 121,121,121 ⟩	⟨ 0,0,0 ⟩
16	Rustine	100	45	44	14	14	0
	⟨ C2R,C ⟩	⟨ 100,100 ⟩	⟨ 45,41 ⟩	⟨ 40,41 ⟩	⟨ 14,14 ⟩	⟨ 14,14 ⟩	⟨ 0,0 ⟩
17	Rustine	100	87	60	1570	1570	0
	⟨ C2R,C,L ⟩	⟨ 0,0,0 ⟩	⟨ −,−,− ⟩	⟨ −,−,− ⟩	⟨ −,−,− ⟩	⟨ −,−,− ⟩	⟨ −,−,− ⟩
18	Rustine	100	92	84	40	40	0
	⟨ C2R,C,S ⟩	⟨ 100,0,100 ⟩	⟨ 86,−,− ⟩	⟨ 80,−,− ⟩	⟨ 40,−,− ⟩	⟨ 40,−,− ⟩	⟨ 0,−,− ⟩
19*	Rustine	100	–	–	–	–	–
	⟨ C2R,C2SR,C,L ⟩	⟨ 100,100,100,100 ⟩	–	–	–	–	–
20	Rustine	100	61	60	683994	526159	157835
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 72,−,− ⟩	⟨ 73,−,− ⟩	⟨ 683994,−,− ⟩	⟨ 683994,−,− ⟩	⟨ 0,−,− ⟩
21	Rustine	100	13	12	36	36	0
	⟨ C2R,C2SR,R,C,L ⟩	⟨ 100,100,100,100,100 ⟩	⟨ 5,−,−,5.5 ⟩	⟨ 5,−,−,4,4 ⟩	⟨ 36,−,−,36,36 ⟩	⟨ 36,−,−,36,36 ⟩	⟨ 0,−,−,0,0 ⟩
22	Rustine	100	55	60	1949	1949	0
	⟨ C2R,C,L ⟩	⟨ 100,100,100 ⟩	⟨ 60,60,61 ⟩	⟨ 79,72,75 ⟩	⟨ 1949,1949,1949 ⟩	⟨ 1949,1949,1949 ⟩	⟨ 0,0,0 ⟩
23	Rustine	100	65	60	4252	3994	258
	⟨ C2R,C2SR,C,L ⟩	⟨ 100,100,100,100 ⟩	–	–	–	–	–

Table 2 shows that Rustine, fully automated or with human in the loop, **generates semantically equivalent translations similar to transpiler-based techniques**, which preserve semantics by construct. Some test translations by these techniques were not compilable to evaluate functional equivalence. Some techniques remove assertions during translation. For example, the number of executed assertions in translations of *qsort* (ID=1) with C2Rust, C2SaferRust, CROWN, and Laertes is 10, less than the original 21 assertions. C2SaferRust either did not have test translationsor they were not compilable in *bin* mode.

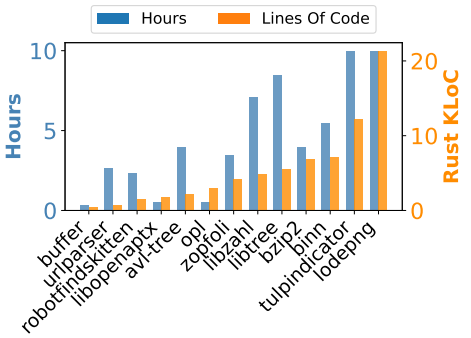

Fig. 6. Manual debugging efforts for projects with at least one assertion failure

Table 3. Safety characteristics of translations by Rustine and alternative approaches. Abbreviations: C2Rust (C2R), C2SaferRust (C2SR), CROWN (C), Laertes (L), RustMamp (R), and Syzygy (S).

ID	Tool	# Pointer Arithmetic	# Raw Pointers		# Unsafe Constructs		
			Declarations	Dereferences	Lines	Type Casts	Calls
1	Rustine	**0**	**0**	**0**	**0**	**0**	**0**
	⟨ C2R,C2SR,C,L ⟩	⟨ 5,5,5,5 ⟩	⟨ 4,1,4,2 ⟩	⟨ 10,6,10,6 ⟩	⟨ 39,19,39,32 ⟩	⟨ 12,13,12,12 ⟩	⟨ 11,10,11,23 ⟩
2	Rustine	0	**0**	**0**	**0**	**0**	**0**
	⟨ C2R,C,L ⟩	⟨ 0,0,0 ⟩	⟨ 9,4,9 ⟩	⟨ 29,7,29 ⟩	⟨ 60,61,68 ⟩	⟨ 8,6,8 ⟩	⟨ 21,92,21 ⟩
3	Rustine	**0**	**0**	**0**	**0**	**0**	**0**
	⟨ C2R,C,L ⟩	⟨ 33,33,33 ⟩	⟨ 21,16,16 ⟩	⟨ 78,66,66 ⟩	⟨ 706,604,1621 ⟩	⟨ 525,562,974 ⟩	⟨ 155,148,187 ⟩
4	Rustine	**0**	**4**	**2**	**26**	**0**	**8**
	⟨ C2R,C,L ⟩	⟨ 1,1,1 ⟩	⟨ 50,33,47 ⟩	⟨ 210,108,206 ⟩	⟨ 980,759,819 ⟩	⟨ 419,443,376 ⟩	⟨ 294,509,266 ⟩
5	Rustine	**1**	**0**	**0**	**0**	**0**	**0**
	⟨ C2R,C,L ⟩	⟨ 23,23,23 ⟩	⟨ 61,19,57 ⟩	⟨ 82,19,73 ⟩	⟨ 1008,780,871 ⟩	⟨ 575,620,557 ⟩	⟨ 334,518,327 ⟩
6	Rustine	**0**	15	7	112	15	51
	⟨ C2R,C2SR,C,L ⟩	⟨ 8,1,8,2 ⟩	⟨ 17,**7**,17,8 ⟩	⟨ 28,**11**,29,12 ⟩	⟨ 590,**84**,590,213 ⟩	⟨ 247,**10**,247,122 ⟩	⟨ 100,**33**,106,71 ⟩
7	Rustine	**0**	**0**	**0**	**4**	**2**	**3**
	⟨ C2R,C2SR,C,L ⟩	⟨ 52,1,5,10 ⟩	⟨ 85,81,78,74 ⟩	⟨ 133,51,42,2 ⟩	⟨ 1490,702,932,1002 ⟩	⟨ 804,450,706,1167 ⟩	⟨ 441,439,397,572 ⟩
8	Rustine	**21**	**21**	**17**	**453**	**57**	**274**
	⟨ C2R,C2SR,C,L ⟩	⟨ 125,115,125,117 ⟩	⟨ 29,29,29,29 ⟩	⟨ 172,168,172,172 ⟩	⟨ 1473,1467,1473,1530 ⟩	⟨ 877,631,877,897 ⟩	⟨ 329,325,329,321 ⟩
9	Rustine	**0**	**1**	**4**	**43**	**8**	**21**
	⟨ C2R,C2SR,C,L ⟩	⟨ 127,82,113,246 ⟩	⟨ 88,63,71,73 ⟩	⟨ 355,273,317,339 ⟩	⟨ 2140,994,1775,1650 ⟩	⟨ 1385,362,1317,1329 ⟩	⟨ 597,578,643,581 ⟩
10	Rustine	**1**	**1**	**0**	7	**0**	**9**
	⟨ C2R,C,L ⟩	⟨ 66,22,22 ⟩	⟨ 74,13,13 ⟩	⟨ 126,24,31 ⟩	⟨ 1083,178,176 ⟩	⟨ 521,72,72 ⟩	⟨ 339,117,102 ⟩
11	Rustine	**0**	**1**	**0**	20	1	8
	⟨ C2R,C,L ⟩	⟨ 20,21,16 ⟩	⟨ 3,3,2 ⟩	⟨ 24,24,24 ⟩	⟨ 556,513,475 ⟩	⟨ 346,340,331 ⟩	⟨ 164,160,162 ⟩
12	Rustine	4	6	8	113	2	41
	⟨ C2R,C,L ⟩	⟨ 115,63,60 ⟩	⟨ 57,23,19 ⟩	⟨ 185,31,26 ⟩	⟨ 2857,799,873 ⟩	⟨ 1794,423,26 ⟩	⟨ 513,496,19 ⟩
13	Rustine	0	15	12	93	5	45
	⟨ C2R,C,L ⟩	⟨ 21,0,0 ⟩	⟨ 117,5,13 ⟩	⟨ 366,8,53 ⟩	⟨ 2019,84,91 ⟩	⟨ 1296,30,18 ⟩	⟨ 503,202,37 ⟩
14	Rustine	7	1	0	5	2	2
	⟨ C2R,C ⟩	⟨ 236,236 ⟩	⟨ 84,84 ⟩	⟨ 361,361 ⟩	⟨ 2077,2077 ⟩	⟨ 1389,6 ⟩	⟨ 580,8 ⟩
15	Rustine	22	8	4	162	25	131
	⟨ C2R,C,L ⟩	⟨ 198,199,199 ⟩	⟨ 95,78,76 ⟩	⟨ 358,201,246 ⟩	⟨ 2202,2134,2199 ⟩	⟨ 1178,1231,1241 ⟩	⟨ 625,1010,899 ⟩
16	Rustine	12	0	0	5	0	5
	⟨ C2R,C ⟩	⟨ 170,177 ⟩	⟨ 150,150 ⟩	⟨ 566,569 ⟩	⟨ 3839,3839 ⟩	⟨ 2375,1 ⟩	⟨ 622,2 ⟩
17	Rustine	12	6	1	1731	8	658
	⟨ C2R,C,L ⟩	⟨ 281,–,– ⟩	⟨ 193,–,– ⟩	⟨ 650,–,– ⟩	⟨ 2903,–,– ⟩	⟨ 1107,–,– ⟩	⟨ 1240,–,– ⟩
18	Rustine	3	1	0	8	1	5
	⟨ C2R,C,S ⟩	⟨ 819,–,8 ⟩	⟨ 291,–,0 ⟩	⟨ 1359,–,0 ⟩	⟨ 8085,–,0 ⟩	⟨ 3703,–,0 ⟩	⟨ 1924,–,0 ⟩
19	Rustine	2	8	0	52	17	38
	⟨ C2R,C2SR,C,L ⟩	⟨ 916,1873,1460,922 ⟩	⟨ 244,231,432,224 ⟩	⟨ 842,747,1468,765 ⟩	⟨ 5842,5583,10835,7213 ⟩	⟨ 3252,2867,69,5940 ⟩	⟨ 1587,1533,57,1772 ⟩
20	Rustine	90	27	17	117	11	39
	⟨ C2R,C,L ⟩	⟨ 2055,2119,2122 ⟩	⟨ 552,450,400 ⟩	⟨ 2781,2352,1886 ⟩	⟨ 13030,13000,13127 ⟩	⟨ 8632,8929,8985 ⟩	⟨ 3452,5069,5601 ⟩
21	Rustine	7	19	20	114	47	130
	⟨ C2R,C2SR,R,C,L ⟩	⟨ 1149,849,24,1105,1111 ⟩	⟨ 230,199,24,194,206 ⟩	⟨ 4338,2691,69,3674,3699 ⟩	⟨ 13008,10328,974,12592,13115 ⟩	⟨ 7421,4097,70,7672,8699 ⟩	⟨ 2611,2455,372,2929,2788 ⟩
22	Rustine	56	57	20	331	140	335
	⟨ C2R,C,L ⟩	⟨ 110,110,116 ⟩	⟨ 369,317,302 ⟩	⟨ 498,329,339 ⟩	⟨ 4163,4139,4105 ⟩	⟨ 1635,1572,1574 ⟩	⟨ 631,1060,1036 ⟩
23	Rustine	**15**	**7**	**14**	**34**	**16**	**22**
	⟨ C2R,C2SR,C,L ⟩	⟨ 4678,1215,1283,2572 ⟩	⟨ 2279,860,865,866 ⟩	⟨ 6744,1625,1847,1847 ⟩	⟨ 35228,17314,17769,25594 ⟩	⟨ 22861,12274,13195,13050 ⟩	⟨ 8518,2638,2419,2595 ⟩

We further evaluate the quality of the translations across the following metrics (results in Tables 3, gray rows for Rustine and white rows for alternative approaches):

3.2.1 Raw Pointer Declarations/Dereferences. Rustine effectively reduces the usage of raw pointers in translations: **the number of raw pointer declarations/derefenreces is reduced from** 4,080/2,045 **in C to** 198/126 **in Rust, respectively** (*Raw Pointers* columns in Table 1; cf. Table 3). Translations of transpiler-based techniques contain drastically more raw pointer declarations and dereferences. C2Rust, C2SaferRust, CROWN, and Laertes have 4,904/20,169, 1,400/5,510, 3,027/12,298, and 2,255/9,685 **more raw pointer declarations and dereferences**, concerning overlapping projects. Rustine translations contain 5/49 **fewer raw pointer declarations and dereferences** compared to RustMamp in *bzip2*, and only one more raw pointer declarations compared to Syzygy.

3.2.2 Pointer Arithmetic/Comparison. Rustine **effectively resolved many pointer operation cases** through refactoring (*Before* and *After* of *Pointer Arithmetic* columns under Table 1), reducing them from 452 to 14. During translation, Rustine resolves *four* additional instances of pointer arithmetic in project *gennan* (ID=9), but surprisingly, **adds** 243 **new cases of pointer arithmetic** in translations of 14 projects (column *Pointer Arithmetic* in Table 3). We carefully investigated these cases and identified the following two reasons for this: (1) interacting with C libraries with no equivalence in Rust, e.g., operating system libraries, and (2) manipulating strings.

-- bzip2 Source Code in C --

```c
void BZ_API(BZ2_bzclose) (BZFILE* b) {
    // ... omitted code
    fp = ((bzFile *)b)->handle;
    // ... omitted code
}
```

-- Rustine Translation of bzip2 --

```rust
pub fn BZ2_bzclose(b: Option<&mut BZFILE>) {
    // ... omitted code
    let fp = unsafe {
        *bz_fp.offset(fp_idx as isize)
    };
    // ... omitted code
}
```

-- libtree Source Code in C --

```c
int in_exclude_list = soname != MAX_OFFSET_T &&
    is_in_exclude_list(s->string_table.arr[soname_b
    uf_offset]);
```

-- Rustine Translation of libtree --

```rust
let in_exclude_list = soname != u64::MAX && {
    let soname_str = unsafe {
        CStr::from_ptr(s.string_table.arr.as_ref()
        .unwrap().as_ptr()
        .add(soname_buf_offset) as *const i8)
    };
    is_in_exclude_list(Some(soname_str.to_str().unwr
    ap())) != 0
}
```

Fig. 7. Examples of introducing pointer arithmetic in Rust translation by Rustine, which did not exist in the original C code, for *bzip2* (left) and *libtree* (right)

Figure 7 shows such a case. For *bzip2* (left side), the C program casts the opaque pointer `BZFILE` into structure type `bzFile`, and directly accesses its inner handle field. Rust forbids such casting and field access on opaque types. The translation maintains a separate array `bz_fp` of file pointers and computes the correct `FILE` via an offset derived from the opaque pointer's address. For *libtree* (right side), the C program checks whether the variable `soname` is a valid offset not equal to `MAX_OFFSET_T`, and whether the string at position `soname_buf_offset` in `s->string_table.arr` appears in the program's exclude list, storing the result in `in_exclude_list`. Rust does not allow indexing raw bytes with arbitrary offsets, and `&[u8]` or `&str` slicing requires valid UTF-8 and known bounds. To avoid complex lifetime tracking and unsafe transmutes that could violate aliasing rules, the LLM computes the pointer to the start of the target string with `.add(soname_buf_offset)` and constructs a `CStr` directly from that address.

Compared to Rustine, translations of transpiler-based techniques contain **considerably higher** pointer operations. Such techniques not only move existing pointer arithmetic in the C program to translation, but also drastically inflate their usage. The primary reason for pointer arithmetic inflation in transpiler-based techniques, however, is different: these tools convert array subscript accesses (`[]`) to explicit pointer arithmetic, even in cases where the original C code used clean, readable subscript notation. Figure 8 shows an example of such cases in C2Rust translation. The C code computes the address of a pixel in an `XImage` structure using two nested subscript expressions that account for pixel format, row padding (`bytes_per_line`), and per-component offset (`xoffset`). For implementing the same logic in Rust, C2Rust mechanically replaces every `[]` with a chain of `.offset()` and `.wrapping_mul()` calls, repeatedly recomputing the base address of the `ximage` array and the size of `T`. The resulting expression spans 17 lines and contains seven separate pointer arithmetic operations, with none in C code. The **margin with LLM-based approaches is smaller**: RustMamp translations of *bzip2* (ID=22) include 69 pointer arithmetic, compared to 20 in Rustine; Syzygy and Rustine both generate translations with *zero* pointer arithmetic for *zopfli* (ID=18).

-- xzoom Source Code in C --

```c
(T *)(&ximage[t]
    ->data[(ximage[t]->xoffset + (x)) * sizeof(T) +
        (y)*ximage[t]->bytes_per_line])
```

-- C2Rust Translation of xzoom --

```rust
&mut *((**ximage.as_mut_ptr().offset(
            1 as libc::c_int as isize))
        .data)
    .offset((((**ximage.as_mut_ptr().offset(
            1 as libc::c_int as isize))
        .xoffset +
    0 as libc::c_int) as libc::c_ulong)
        .wrapping_mul(
            ::core::mem::size_of::<libc::c_uchar>()
                as libc::c_ulong, )
        .wrapping_add(
            (j * magy *
            (**ximage.as_mut_ptr().offset(
                1 as libc::c_int as isize))
                .bytes_per_line)
                as libc::c_ulong, ) as isize, )
    as *mut libc::c_char as *mut libc::c_uchar;
```

Fig. 8. An example of pointer arithmetic inflation in the translation of *xzoom* project by C2Rust

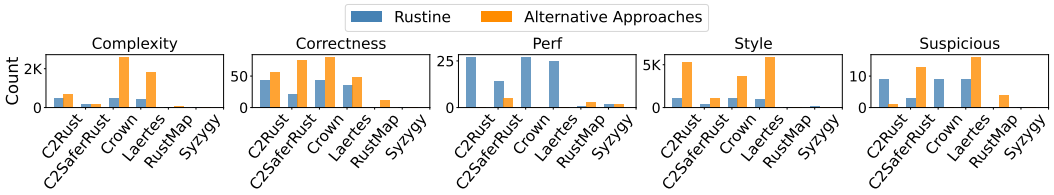

Fig. 10. Pairwise comparison of Rustine with others concerning raised `Clippy` flags on mutual projects

3.2.3 Unsafe Constrcuts. Rustine translations contain **considerably fewer unsafe constructs compared to transpiler-based approaches**, as shown in Table 3. The number of unsafe ⟨lines,type casts,calls⟩ in Rustine translations (⟨114, 47, 130⟩) are **notably less** compared to RustMamp (⟨974, 70, 372⟩). Syzygy translations contain no unsafe constructs, compared to 14 in Rustine translations. We believe the higher cost of translation of *zopfli* by Syzygy ($800) compared to < $0.71 cost of translation by Rustine, can explain the difference: re-prompting the LLM more can be helpful in reducing the safety issues, but can also increase the cost of translation.

3.2.4 Idiomaticity. Figure 9 shows the TSI-normalized number of all raised `Clippy` flags for Rustine, compared to alternative approaches concerning mutual projects. Figure 10 also shows the mutual comparison per each `Clippy` flag. These figures show that **Rustine consistently raises fewer flags per normalized KLoC compared to other approaches**, i.e., it better adheres to Rust's idiomatic patterns compared to existing approaches. Concerning individual flag categories, Rustine consistently raises fewer `complexity` and `correctness` flags compared to other approaches.

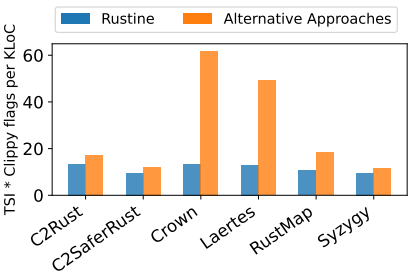

Fig. 9. $TSI \times Clippy_{flags}$ per KLoC comparison of Rustine with alternative approaches

The `correctness` warnings indicate opportunities to improve idiomaticity rather than functional incorrectness. Our investigation confirms the **majority of `correctness` warnings for Rustine are *false positives* or acceptable trade-offs**: (1) C code usually uses `int` for lengths, requiring bound checking. Rustine translates `int` types to `usize`, which are always positive. However, it preserves the C structure and keeps the bound check, resulting in `Clippy::correctness` warning. (2) Some C libraries have no equivalence in Rust, e.g., `X11` [70] lib in *xzoom*, making the use of FFI in Rust to interact with them inevitable (§2.3.1). `Clippy` may raise `correctness` warnings on bindgen-generated FFI bindings rather than translated application logic. (3) Rust provides built-in named constants, e.g., `f64::consts::LOG2_E`, while C does not. When translating a C code containing similar but hard-coded variables, the LLM typically preserves the C structure and redefines them, raising `correctness` warning.

```
-- buffer Source Code in C --
buffer_t *buffer_new_with_copy(char *str) {
  // ... omitted code
  memcpy(self->alloc, str, len);
```

```
-- C2Rust Translation of buffer --
pub unsafe extern "C" fn buffer_new_with_copy(
    mut str: *mut libc::c_char, ) -> *mut buffer_t {
  // ... omitted code
  memcpy((*self_0).alloc as * mut libc::c_void, str as
  ↪  *const libc::c_void, len);
```

```
-- Rustine Translation of buffer --
pub fn buffer_new_with_copy(str: Option<&str>)
    -> Option<BufferT> {
  // ... omitted code
  let to_copy = std::cmp::min(alloc.len(), src.len());
  for i in 0..to_copy {
    alloc[i] = src[i];
  }
CLIPPY: ^ help: try replacing the loop by:
"alloc[..to_copy].copy_from_slice(&src[..to_copy]);"
```

Fig. 11. An example of `clippy::perf` warning for Rustine, while not complaining about C2Rust unsafe block

Transpiler-based approaches, surprisingly, raised almost *zero* `perf` warnings. A deeper investigation reveals such warnings are mostly improvements to *closely* idiomatic Rust code. The transpiler-based techniques suffer from serious safety issues discussed before, far from being idiomatic, and escaping linter checks (false negatives). The example of Figure 11 shows such a

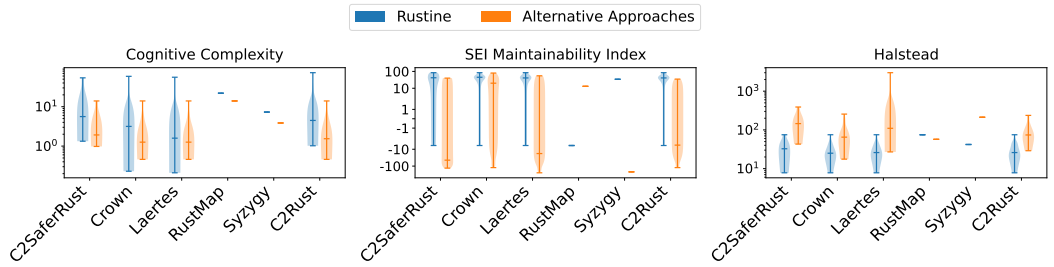

Fig. 12. Comparing the readability of translations generated by Rustine and alternative approaches. For Cognitive Complexity and Halstead, lower values indicate more readability. For SEI, a higher value is better

case: C2Rust translations is unsafe, using C API and raw pointers to copy the buffer. Rustine translation is safe with no pointers, but `Clippy` suggests using `copy_from_slide` rather than a `loop`. This observation suggests that, **`Clippy` warnings should be not be used alone to determine the idiomaticity of translations, without comparison concerning different safety aspects and readability**.

3.2.5 Readability. Rustine generates readable and well-structured Rust code that **significantly outperforms existing translation tools across multiple metrics**. Figure 12 compares the readability of Rustine translations with others on the overlapping projects. Rustine translations consistently score lower cognitive complexity, lower Halstead, and higher SEI. The only exception is comparison with RustMamp, where Rustine translations achieve a lower SEI score and slightly higher Halstead. As shown in Figure 10, Rustine raises considerably fewer `clippy::style` flags, demonstrating its superior adherence to Rust coding conventions. These results confirm that **Rustine preserves functionality and actively enhances code structure and maintainability during translation, producing output that approaches hand-written Rust quality**.

3.3 RQ2. Effectiveness of the Refactoring

An important contribution of Rustine is considering raw pointers, and specifically pointer operations, as first-class citizens. As we mentioned, even when pointer operations do not exist in the C program, both transpiler- and LLM-based techniques result in Rust translations with pointer arithmetic/comparisons. But when they exist, the inflation explodes in translations. To investigate this observation in a more controlled setting, we excluded the pointer arithmetic refactoring component of Rustine and retranslated the translation units involving pointer operations.

Figure 13 corroborates **a significant rise in not only pointer operation cases, but also raw pointer declarations/dereferences, and unsafe constructs**. On average, the number of pointer arithmetic increases by 83.01%, raw pointer declarations/dereferences by 52.6%/154%, and unsafe ⟨lines,type casts,calls⟩ by ⟨64.42%, 31.67%, 61.46%⟩. These results, along with examples shown before (Figure 2), confirm that the **refactoring component is critical in Rustine to minimize safety issues** in Rust translation: in the absence of pointer arithmetic refactoring, direct pointer manipulations are preserved, or even worse, will be inflated during translation.

3.4 RQ3. Translation Bugs

We discuss examples of translation bugs due to improper treatment of *dynamic memory management*, *global variables*, and *Rust APIs*, which were resolved through automated or manual debugging.

3.4.1 Dynamic Memory Management. In C, manual dynamic memory management is unavoidable due to the absence of high-level abstractions. Developers must explicitly track allocation sizes, alignments, and pointer validity. Rustine often mimics this low-level pattern in translations.

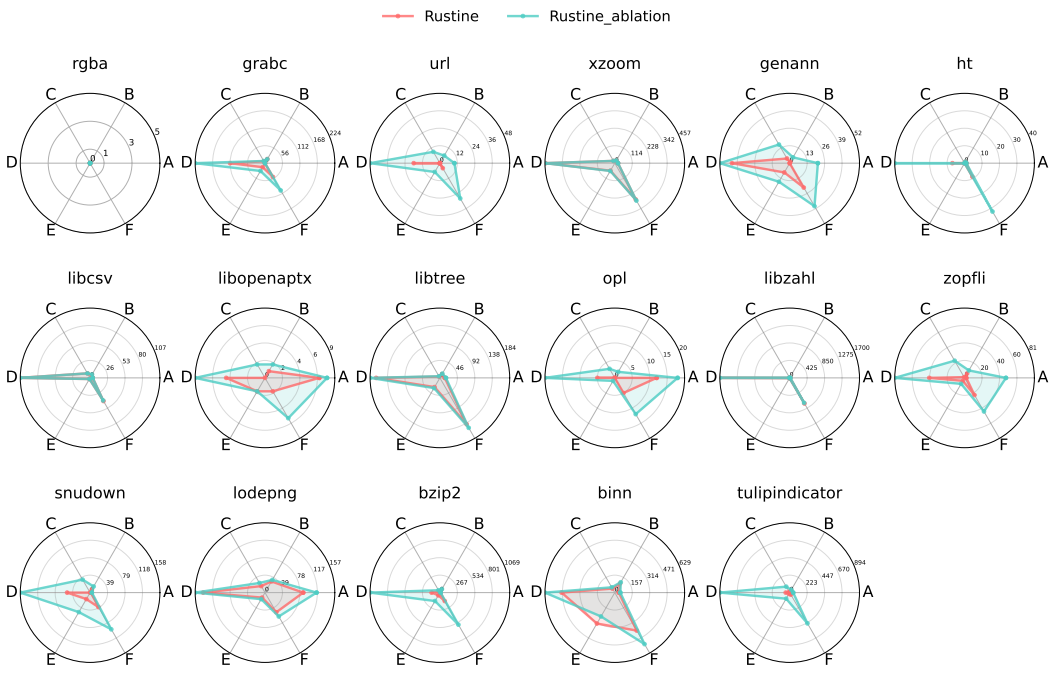

Fig. 13. Impact of removing pointer arithmetic resolution before translation. Comparison dimenstions are: A=Ptr arithmetic, B=Ptr decleration, C=Ptr derefrence, D=Unsafe lines, E=Unsafe type cast, F=Unsafe calls

However, due to the inherent complexity of manual memory management, translations are often incorrect or unsafe (pointer lifetime after reallocation or failing to update references to the data).

The code below is an example from *buffer* project, where the C code uses `realloc` to double the size of a dynamically growing buffer. RUSTINE follows the same pattern, but fails to correctly handle memory alignment and pointer validity across the program. Due to the complexity of Rust's memory management mechanisms, automated debugging cannot fix the issue. The developer resolved the issue by replacing raw pointers and `realloc` with `Vec<T>` directly in the struct layout, eliminating the need for manual memory management.

```
-- Rustine Translation --

v.capacity *= 2;
let layout =
    Layout::array::<u64>(v.capacity).unwrap();
let old_ptr = match & v.p {
    Some(boxed) = > Box::into_raw(boxed.clone()) as
    *mut u64,
    None = > unreachable !(),
};
let ptr = unsafe{std::alloc::realloc(old_ptr as * mut
    u8, layout, v.capacity
    *std::mem::size_of::<u64>()) as * mut u64};
```

```
-- buffer Source Code in C --

v->capacity *= 2;
uint64_t *p = realloc(v->p, v->capacity *
    sizeof(uint64_t));
... //omitted code
v->p = p;
```

```
-- Manually Debugged Translation --

v.capacity *= 2;
if let Some(ref mut inner_vec) = v.p {
    inner_vec.resize(v.capacity, 0);
}
```

In many cases, when provided with sufficient contextual prompting, RUSTINE defines `Vec<T>` in struct layouts and APIs or avoids manual allocation, resulting in compilable, idiomatic, and logically correct Rust code. However, even when using `Vec<T>`, RUSTINE may struggle to translate the C memory allocation into proper Rust equivalent, especially when Rust-specific APIs are involved. As shown in the example below, RUSTINE initially translated the C `realloc` usage to `resize`, which directly increases the vector length, resulta ing in semantic mismatch and assertion failure. During the automated debugging process, RUSTINE successfully patched the semantic mismatch by using `reserve`, which expands the capacity without changing the length.

```
-- buffer Source Code in C --
char *arr = realloc(t->arr, t->capacity *
↪  (sizeof(char)));
```

```
-- Rustine Translation --
match &mut t.arr {
    Some(arr) => {
        arr.resize(t.capacity, '\0');
    },
    None => {
        t.arr = Some(vec!['\0'; t.capacity]);
    }
}
```

```
-- Rustine Debugging --

match &mut t.arr {
    Some(arr) => {
        let current_capacity = arr.capacity();
        if t.capacity > current_capacity {
        arr.reserve(t.capacity - current_capacity);
        }
    },
    None => {
        t.arr = Some(Vec::with_capacity(t.capacity));
    }
}
```

3.4.2 Global Variables. The LLM may struggle with initialization and the use of global variables due to strict ownership rules. Despite explicit guidance (context about global variables and in-context examples) to use `lazy_static` for handling global state in Rust, the LLM sometimes failed to perform proper initialization. Consider the example below from *tulipindicator*, where the C code has a global array of `struct ti_indicator_info`. Rustine initial translation uses unsafe zero-initialization instead of properly populating the structure fields. The manually debugged version properly initializes each field with appropriate values.

```
-- Manually Debugged Translation --

lazy_static ! {
 pub static ref TI_INDICATORS
    : Mutex<Vec<TiIndicatorInfo>> = {
    let mut indicators = Vec::new ();
    indicators.push(
      TiIndicatorInfo {
       name: Some("abs".to_string()),
       full_name: Some("Vector Absolute
       ↪  Value".to_string()),
       start: Some(Box::new(|slice: &[f64]|
           { ti_abs_start(Some(slice))}))
           as Box<dyn Fn(&[f64]) -> i32>),
       indicator: ti_abs,
       ... //omitted code
     }
```

```
-- tulipindicator Source Code in C --
struct ti_indicator_info ti_indicators[] = {
    { "abs", "Vector Absolute Value", ti_abs_start,
        ti_abs, 0, TI_TYPE_SIMPLE, 1, 0, 1, {
    ↪  "real", 0 }, { "", 0 }, { "abs", 0 }, 0, 0,
    ↪  0 },
 ... //omitted code
```

```
-- Rustine Translation --
lazy_static ! {
 pub static ref TI_INDICATORS
  : Mutex<Vec<TiIndicatorInfo>> = {
   let mut indicators = Vec::new ();
   let empty_indicator = unsafe{
   ↪  zeroed::<TiIndicatorInfo>() };
   indicators.push(empty_indicator);
   Mutex::new (indicators)
};}
```

Rustine may fail to properly handle locking when accessing global variables. In Rust, locks acquired through `Mutex::lock()` are held until the guard goes out of scope, which can lead to deadlocks if not managed carefully. Rustine sometimes failed to recognize when locks should be explicitly released, particularly in scenarios when variable accessed by multiple function calls within the same scope. The example below shows such a case in *robotfindskitten* prohect, where the global variable `robot` is accessed but never released for subsequent function calls reading it. In the manually fixed version, the developer explicitly drops the lock and solves the deadlock issue.

```
-- Rustine Translation --
pub fn process_input(input: i32) {
    let mut robot = ROBOT.lock().unwrap();
    let mut check_x = robot.as_ref().unwrap().x;
    let mut check_y = robot.as_ref().unwrap().y;
    // Lock is still held here!
    // Following function calls will read the global
    ↪  variable
    // But they will face a deadlock because lock is
    ↪  not released

    ... //omitted code
}
```

```
-- robotfindskitten Source Code in C --
void process_input(int input) {
    int check_x = robot.x;
    int check_y = robot.y;
    ... //Following function calls will read robot
```

```
-- Manually Debugged Translation --
pub fn process_input(input: i32) {
    let mut robot = ROBOT.lock().unwrap();
    let mut check_x = robot.as_ref().unwrap().x;
    let mut check_y = robot.as_ref().unwrap().y;
    // explicitly drop the lock
    drop(robot);
    ... //omitted code
```

3.4.3 API Misuse. API invocations in translations are either due to mapping the C APIs into their equivalents in Rust, or introducing a Rust API to implement a C code with no API, likely to generate a more idiomatic Rust. Rustine is pretty successful in the former, due to pre-translation analysis (§2.3.1) and use of adaptive ICL examples (§2.4.1). However, we observed several cases where the LLM's creativity in using APIs was incorrect in the initial translation. The example below shows such a case, where Rustine implemented the reverse byte copying in C by combining `to_be_bytes()`, to convert number to *big-endian* bytes, followed by `from_be_bytes()`, to convert *big-endian* bytes to number, which, in fact, cancels the conversion. Automated debugging of Rustine resolves the issue with a proper API usage, `to_ne_bytes()` and `from_ne_bytes()`.

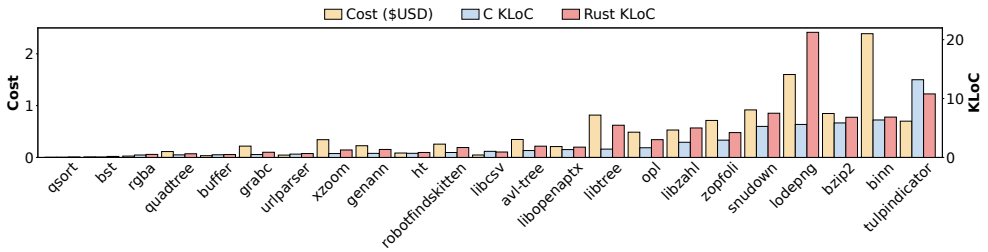

Fig. 14. Subject projects' size (Rust and C), and corresponding translation costs

```c
-- binn Source Code in C --
static void copy_be32(u32 *pdest, u32 *psource) {
  unsigned char *source = (unsigned char *) psource;
  unsigned char *dest = (unsigned char *) pdest;
  dest[0] = source[3]; dest[1] = source[2];
  dest[2] = source[1]; dest[3] = source[0];
```

```c
-- Rustine Debugging --
  let source = psource.to_ne_bytes();
  let mut dest_bytes = [0u8; 4];
  for i in 0..4 {dest_bytes[i] = source[3 - i];}
  *pdest = u32::from_ne_bytes(dest_bytes);
```

```c
-- Rustine Translation --
pub fn copy_be32(pdest: &mut u32, psource: &u32) {
  let source = psource.to_be_bytes();
  let dest = &mut *pdest;
  *dest = u32::from_be_bytes([
      source[3],
      source[2],
      source[1],
      source[0]
  ]);
}
```

3.5 RQ4. Performance and Cost

The cost of translation using Rustine ranged from $0.0026 to $2.39 (average=$0.48), **demonstrating cost-efficiency**. The cost of automated debugging is negligible compared to actual translation, as the LLM attempts fixing a few suspicious functions. For example, in the *libtree* project, translation and debugging cost $0.82 and $0.049 (6% of the total cost), respectively. **There is a significant correlation between the translation cost and project size** in C ($\rho = 0.90$ with strong statistical significance, p-value= $4.96e-9 < 0.05$) and Rust ($\rho = 0.95$ with strong statistical significance, p-value= $1.64e-12 < 0.05$). Besides the size, the complex logic and specific program properties, e.g., heavy use of global variables or dependencies, can also contribute to the cost of translation.

Given that the cost depends on the models, and most of the alternative approaches used GPT-4o in their experiments for translations, we also estimated the price based on the total number of input and output tokens in our experiments. In contrast to Syzygy [54], which reported around $800 for translating the single project *zopfli*, the estimated cost for Rustine would be $10.92 (actual cost of $0.71). The highest translation cost using GPT-4o would be $37 (actual cost of $2.39), showing Rustine potential to scale to large real-world repositories without incurring high costs.

The translation process dominates the overall runtime, ranging from 1 minute to 20 hours (average = 3.92 hours), with other components (*Pre-processing and Refactoring* and *Analysis and Decomposition*) taking less than a few seconds, **demonstrating scalability of Rustine**. The Spearman Rank Order Correlation test shows a *strong correlation* between the translation time and project size in C ($\rho = 0.88$ with strong statistical significance, p-value= $2.41e-8 < 0.05$), corroborating that translation of bigger projects takes more time.

4 Related Work

Automated code translation techniques can be categorized as rule-based techniques or learning-based approaches. For <C,Rust> translation pairs, transpilers such as C2Rust [12], Corrode [18], CROWN [80], and Laertes [22] rely on Clang-based parsing and systematic rewrites. These approaches prioritize preserving semantics but often yield non-idiomatic Rust due to the overuse of unsafe constructs.

The emergence of neural networks and LLMs has greatly advanced automated code translation. Neural machine translation methods [15, 42–44] have been shown to effectively automate translation between different programming language pairs, such as <Java,C#>, <Python,JavaScript>,

<Java,Python>, and <C++,Java>, although poorly scaled to real-world repositories. Pan et al. [49] demonstrated the possible use of LLMs in automating code translation, while highlighting the challenges in real-world code translation. Vert [73] utilized LLMs for verified translation of small program slices from real-world C projects to Rust, and AlphaTrans [30] proposed a scalable technique for repository-level code translation from Java to Python. Subsequent work showed that ideas of compositional or skeleton-guided translation are effective for repository-level C to Rust translation [8, 13, 23, 27, 29, 31, 34, 37, 45, 54, 55, 64–66, 74, 78, 81].

Prior work on translating real-world C projects to Rust suffers from (1) the generation of non-idiomatic Rust code, (2) a lack of scalability due to high cost, (3) limited debugging support, and insufficient translation validation. Assessment of safety in most related work remains at measuring the number of raw pointers. LLM-based techniques, specifically agentic translations, rarely report translation costs, making it impossible to assess the applicability of the developed techniques. Rustine, in contrast, automatically translates C projects to fully compilable, safe, and idiomatic Rust at a reasonable cost and with a high level of semantic equivalence. Due to the high quality of translations and debugging support, the human developer quickly fixed semantic mismatches.

There have also been several attempts to construct C to Rust translation benchmarks [14, 32, 48, 71, 72]. We were not able to use these benchmarks because of (1) quality issues in the artifacts, e.g., not being able to compile or build, or lack of representativeness of challenges in C to Rust translation, e.g., excessive use of raw pointers and pointer arithmetic; or (2) not being able to run LLM-based alternative approaches on them due to technical issues or high cost of evaluation.

5 Threats to the Validity

External Validity. To ensure generalizability of the tool design and applicability in practice, we evaluated Rustine on 23 projects that were translated by recent automated C to Rust translation techniques. The size of these projects ranged from 27 to 13,200 LoCs, and their implementations contained a diverse set of C-specific features that could challenge translation to Rust, such as excessive raw pointer declarations and dereferences, global mutable variables, micros, and pointer arithmetic. Concerning the generalizability of claims, we compared Rustine with several transpilation or LLM-based techniques, with the results consistently showing the superiority of Rustine.

Internal Validity. One threat to the internal validity of code translation results could be inflated functional equivalence success. To account for such bias caused by limited, low coverage tests, we manually augmented the existing test suite of subjects with diverse unit tests, including high-quality assertions. We evaluated the quality of translations using existing and new metrics, providing all the details for transparency. Our artifacts are publicly available [5].

Construct Validity. We obtained the subjects from the artifacts of prior techniques to ensure Rustine translates the same version of a code translated by other approaches. Through careful manual investigation of alternative approaches artifacts, we identified several bugs in calculating the metrics or targeted suppression of Clippy rules. We fixed those issues and used widely-used tools for calculating the metrics on their translation, as explained in §3.1

6 Conclusion

This paper proposes Rustine for scalable and efficient translation of C repositories into safe, idiomatic Rust. Rustine is an important and essential first step to advance the research in repository-level code translation: it generates a fully compilable idiomatic Rust project as the baseline, helping future research to focus on resolving complex issues concerning C to Rust translation, such as concurrency. As the next step, we aim to ensure semantic equivalence in translation in the presence of concurrent implementation in C projects. This requires technical advancements in the translation as well as validation of concurrent behavior under two different concurrency systems.

References

[1] 2020. rust-code-analysis tool. https://lib.rs/crates/rust-code-analysis.

[2] 2025. Clippy Linting Tool. https://github.com/rust-lang/rust-clippy.

[3] 2025. List of Pointer Arithmetic in Rust. https://doc.rust-lang.org/std/primitive.pointer.html.

[4] 2025. Rust Error Codes Index. https://doc.rust-lang.org/stable/error_codes/.

[5] 2025. Rustine Official Website. https://github.com/Intelligent-CAT-Lab/Rustine.

[6] Luca Ardito, Luca Barbato, Marco Castelluccio, Riccardo Coppola, Calixte Denizet, Sylvestre Ledru, and Michele Valsesia. 2020. rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes. *SoftwareX* 12 (2020), 100635.

[7] All Religions are One. 1996. Smashing The Stack For Fun And Profit. https://api.semanticscholar.org/CorpusID:208092205

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

[9] Alexis Kenneth Beingessner. 2016. *You can't spell trust without Rust.* Ph. D. Dissertation. Carleton University.

[10] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davi. 2022. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *USENIX Security Symposium 2022*. –. arXiv:2203.04117.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[12] C2Rust. 2025. C2Rust Project. https://github.com/immunant/c2rust.

[13] Xuemeng Cai, Jiakun Liu, Xiping Huang, Yijun Yu, Haitao Wu, Chunmiao Li, Bo Wang, Imam Nur Bani Yusuf, and Lingxiao Jiang. 2025. RustMap: Towards Project-Scale C-to-Rust Migration via Program Analysis and LLM. *arXiv preprint arXiv:2503.17741* (2025).

[14] Hailong Chang, Guozhu Meng, Shuhui Xiao, Kai Chen, Kun Sun, and Yilin Li. 2025. When Code Crosses Borders: A Security-Centric Evaluation of LLM-based Code Translation. *arXiv preprint arXiv:2509.06504* (2025).

[15] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NeurIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 2552–2562.

[16] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, 342–351. doi:10.1145/1062455.1062522

[17] Rust Compiler. 2025. Crate rustc_hir. https://doc.rust-lang.org/beta/nightly-rustc/rustc_hir/index.html.

[18] Corrode. 2017. Corrode: Automatic semantics-preserving translation from C to Rust. https://github.com/jameysharp/corrode.

[19] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, MA Borst, and Tom Love. 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering* 2 (1979), 96–104.

[20] 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 (OOPSLA1)* 7, OOPSLA1 (2023), 551–579. doi:10.1145/3586046

[21] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to Safer Rust. *Proceedings of the ACM on Programming Languages (OOPSLA)* 5, OOPSLA (2021), 1–29. doi:10.1145/3485498

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

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

[24] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering* 28, 12 (2002), 1146–1170.

[25] Free Software Foundation, Thomas E. Dickey, et al. 1993–present. ncurses: A portable terminal screen library. https://www.gnu.org/software/ncurses/. Version information often depends on your system's installed version, e.g., 6.4.

[26] Free Software Foundation, Inc. 2024. The C Preprocessor (A GNU Manual). Available at: https://gcc.gnu.org/onlinedocs/cpp/.

[27] Ziqi Guan, Xin Yin, Zhiyuan Peng, and Chao Ni. 2025. Repotransagent: Multi-agent llm framework for repository-aware code translation. *arXiv preprint arXiv:2508.17720* (2025).

[28] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, USA, 336–345.

doi:10.1145/2737924.2737979

[29] Jaemin Hong and Sukyoung Ryu. 2025. Forcrat: Automatic I/O API Translation from C to Rust via Origin and Capability Analysis. *arXiv preprint arXiv:2506.01427* (2025).

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

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

[32] Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. 2025. CRUST-Bench: A Comprehensive Benchmark for C-to-safe-Rust Transpilation. *arXiv preprint arXiv:2504.15254* (2025).

[33] Ruishi Li, Bo Wang, Tianyu Li, Prateek Saxena, and Ashish Kundu. 2025. Translating C to Rust: Lessons from a User Study. In *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA. doi:10.14722/ndss.2025.241407

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

[35] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What Makes Good In-Context Examples for GPT-3?. In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, Eneko Agirre, Marianna Apidianaki, and Ivan Vulić (Eds.). Association for Computational Linguistics, Dublin, Ireland and Online, 100–114. doi:10.18653/v1/2022.deelio-1.10

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

[37] Feng Luo, Kexing Ji, Cuiyun Gao, Shuzheng Gao, Jia Feng, Kui Liu, Xin Xia, and Michael R Lyu. 2025. Integrating Rules and Semantics for LLM-Based C-to-Rust Translation. *arXiv preprint arXiv:2508.06926* (2025).

[38] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Craig Disselkoen, Aidan Denlinger, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP) 2023*. –. arXiv:2208.13583.

[39] Ghassan Misherghi and Zhendong Su. 2006. Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 142–151. doi:10.1145/1134285.1134307

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland, 245–258. doi:10.1145/1542476.1542504

[41] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526. doi:10.1145/1065887.1065892

[42] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical Statistical Machine Translation for Language Migration. In *FSE*. ACM, New York, NY, USA, 651–654.

[43] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 651–654. doi:10.1145/2491411.2494584

[44] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 585–596. doi:10.1109/ASE.2015.74

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

[46] Ollama. 2025. Ollama Project. https://github.com/ollama/ollama.

[47] Paul Oman and Jack Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*. IEEE Computer Society, 337–338.

[48] Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xing Peng, and Zibin Zheng. 2024. Repository-level code translation benchmark targeting rust. *arXiv e-prints* (2024), arXiv–2411.

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

[50] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[51] pyparser. 2025. pycparser v2.22. https://github.com/eliben/pycparser.

[52] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning To Retrieve Prompts for In-Context Learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz (Eds.). Association for Computational Linguistics, Seattle, United States, 2655–2671. doi:10.18653/v1/2022.naacl-main.191

[53] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. 2023. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. In *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering (ICSE)*. –. arXiv:2311.09394.

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

[55] HoHyun Sim, Hyeonjoong Cho, Yeonghyeon Go, Zhoulai Fu, Ali Shokri, and Binoy Ravindran. 2025. Large Language Model-Powered Agent for C to Rust Code Translation. *arXiv preprint arXiv:2505.15858* (2025).

[56] Richard M Stallman and Zachary Weinberg. 1987. The C preprocessor. *Free Software Foundation* 16 (1987).

[57] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[58] The Linux man-pages project. 2025. Linux manual pages. https://man7.org/linux/man-pages/ Comprehensive collection of Linux manual pages, maintained by Michael Kerrisk and contributors.

[59] The Rust Project Developers. 2014. rust-bindgen: Automatically generates Rust FFI bindings to C (and some C++) libraries. https://github.com/rust-lang/rust-bindgen. Accessed: 2025-11-13.

[60] Tokei. 2025. Tokei. https://github.com/XAMPPRocky/tokei.

[61] Agent Transpiler. [n. d.]. Rustify: Towards Repository-Level C to Safer Rust via Workflow-Guided Multi-Agent Transpiler. ([n. d.]).

[62] vLLM. 2025. vLLM Project. https://github.com/vllm-project/vllm.

[63] David S. Wallach. 2024. A Viewpoint: A Memory Safety Manifesto. *IEEE Software* 41, 2 (2024), 108–112. doi:10.1109/MS.2024.10621859

[64] Bo Wang, Tianyu Li, Ruishi Li, Umang Mathur, and Prateek Saxena. 2025. Program Skeletons for Automated Program Translation. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 920–944.

[65] Chaofan Wang, Tingrui Yu, Jie Wang, Dong Chen, Wenrui Zhang, Yuling Shi, Xiaodong Gu, and Beijun Shen. 2025. EVOC2RUST: A Skeleton-guided Framework for Project-Level C-to-Rust Translation. *arXiv preprint arXiv:2508.04295* (2025).

[66] Yanlin Wang, Rongyi Ou, Yanli Wang, Mingwei Liu, Jiachi Chen, Ensheng Shi, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. EffiReasonTrans: RL-Optimized Reasoning for Code Translation. *arXiv preprint arXiv:2510.18863* (2025).

[67] Jeaye Wilkerson and Evan Cameron. 2024. ncurses-rs: A very thin wrapper around the ncurses TUI library. https://crates.io/crates/ncurses. Rust Crate.

[68] Xiafa Wu and Brian Demsky. 2025. GenC2Rust: Towards Generating Generic Rust Code from C. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. doi:10.1109/ICSE55347.2025.00127

[69] Zhiyong Wu, Yaoxiang Wang, Jiacheng Ye, and Lingpeng Kong. 2023. q: An Information Compression Perspective for In-Context Example Selection and Ordering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 1423–1436. doi:10.18653/v1/2023.acl-long.79

[70] Xlib 2024. Xlib - C Language X Interface. https://www.x.org/releases/current/doc/libX11/libX11/libX11.html.

[71] Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, et al. 2025. ClassEval-T: Evaluating Large Language Models in Class-Level Code Translation. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1421–1444.

[72] Pengyu Xue, Kunwu Zheng, Zhen Yang, Yifei Pei, Linhao Wu, Jiahui Dong, Xiapu Luo, Yan Xiao, Fei Liu, Yuxuan Zhang, et al. 2025. A New Benchmark for Evaluating Code Translation with Third-Party Libraries. *arXiv preprint arXiv:2509.12087* (2025).

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

[74] Zhiqiang Yuan, Wenjun Mao, Zhuo Chen, Xiyue Shang, Chong Wang, Yiling Lou, and Xin Peng. 2025. Project-Level C-to-Rust Translation via Synergistic Integration of Knowledge Graphs and Large Language Models. *arXiv preprint arXiv:2510.10956* (2025).

[75] Jerrold Zar. 2005. *Spearman Rank Correlation*. Vol. 5. doi:10.1002/0470011815.b2a15150

[76] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th International Symposium on Foundations of Software Engineering (FSE-10)*. ACM, 1–10. doi:10.1145/587051.587053

[77] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. doi:10.1109/32.988498

[78] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2025. Scalable, validated code translation of entire projects using large language models. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1616–1641.

[79] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership Guided C to Rust Translation. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 13966)*. Springer, 459–482. doi:10.1007/978-3-031-37709-9_22

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

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