# Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis

MANISH SHETTY [*], University of California, Berkeley, USA
NAMAN JAIN [*], University of California, Berkeley, USA
ADWAIT GODBOLE [*], University of California, Berkeley, USA
SANJIT A. SESHIA [†], University of California, Berkeley, USA
KOUSHIK SEN [†], University of California, Berkeley, USA

Despite extensive usage in high-performance, low-level systems programming applications, C is susceptible to vulnerabilities due to manual memory management and unsafe pointer operations. Rust, a modern systems programming language, offers a compelling alternative. Its unique ownership model and type system ensure memory safety without sacrificing performance.

In this paper, we present Syzygy[1], an automated approach to translate C to *safe* Rust. Our technique uses a synergistic combination of LLM-driven code and test generation guided by dynamic-analysis-generated execution information. This paired translation runs incrementally in a loop over the program in dependency order of the code elements while maintaining per-step correctness. Our approach exposes novel insights on combining the strengths of LLMs and dynamic analysis in the context of scaling and combining code generation with testing. We apply our approach to successfully translate Zopfli, a high-performance compression library with ~3000 lines-of-code and 98 functions generating ~4500 lines-of-code in Rust. We validate the translation by testing equivalence with the source C program on a broad set of inputs. To our knowledge, this is the largest automated and test-validated C to *safe* Rust code translation achieved so far.

Project Website: https://syzygy-project.github.io/

## 1 INTRODUCTION

C to Rust translation has seen tremendous interest in recent years owing to memory safety vulnerabilities in C [47] on the one hand and Rust's strong static type and ownership system guaranteeing compile-time eradication of these vulnerabilities on the other. It is further motivated by the fact that both C and Rust can target similar applications (low-level, performance-critical libraries) and are supported by Clang-based compiler toolchains. Though desirable, C-Rust translation is challenging: C and (safe) Rust employ different typing systems (strongly typed variables and no raw pointers in Rust) and different variable access rules (arbitrary accesses in C, while strict borrowing rules in Rust), amongst other differences. Manual migration of even moderately sized codebases requires multiple person-weeks of effort, motivating the need for automatic translation techniques.

There are two main approaches for code translation: rule-based/symbolic and LLM (Large Language Model)-based. Rule-based translation approaches often operate on a terse intermediate representation (for achieving full coverage with a limited rule set) and thus often produce uninterpretable target code. Symbolic program synthesis approaches (e.g., [2, 34]), on the other hand, often do not scale to multi-function codebases. LLMs shine in both these respects: they produce natural/interpretable code and have superior scaling capabilities. LLMs cannot, however, perform

---

[*]Equal Contribution, order determined using a dice roll.
[†]Equal Advising, order determined using a coin toss.
[1]Syzygy (pronounced [ˈsɪz.ɪ.dʒi] / si.zuh.jee), derived from ancient Greek, is used to represents the dynamic alignment and conjunction of complementary forces that come together in perfect balance.

---

Authors' addresses: Manish Shetty, manishs@berkeley.edu, University of California, Berkeley, Berkeley, CA, USA; Naman Jain, naman_jain@berkeley.edu, University of California, Berkeley, Berkeley, CA, USA; Adwait Godbole, adwait@berkeley.edu, University of California, Berkeley, Berkeley, CA, USA; Sanjit A. Seshia, sseshia@berkeley.edu, University of California, Berkeley, Berkeley, CA, USA; Koushik Sen, ksen@berkeley.edu, University of California, Berkeley, Berkeley, CA, USA.
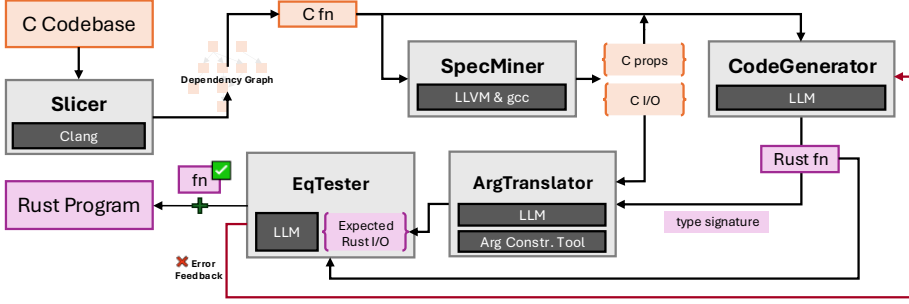
Fig. 1. **Overview of our Syzygy translation approach**: Slicer decomposes the codebase into translation units, CodeGenerator performs translation for that unit, and EqTester checks for equivalence of the generated Rust code with the original C code. ArgTranslator maps the C and Rust function arguments allowing appropriate equivalence checks in EqTester. SpecMiner, our dynamic analysis module mines (property and I/O) specifications for intermediate C functions using top-level tests. These specifications later assist our LLM-driven dual code-test translation. In particular, our identified properties (nullability, aliasing, types) guide LLM to generate appropriate target Rust signature. Similarly, our collected I/O specifications allow EqTester module to provide correctness assurances on incremental translations generating our *Invariant*.

precise inference of semantic features of program executions such as aliasing and allocation sizes. This is especially important in languages with pointer casts and dynamic allocations such as C, where this information is often not syntactically available/interpretable.

We develop Syzygy, an approach to convert medium-large (multi-file, multi-function) C codebases to equivalent *safe* Rust code automatically. As the name suggests, Syzygy[1] exhibits two kinds of dualities. The first is a synergistic combination of superior generative capabilities of LLMs with semantic execution information collected by dynamic analyses [4] on the source C codebase. Secondly, as opposed to using the LLMs+Dynamic Analysis recipe to perform Rust code-generation *alone*, we also build a test translation approach for reliable equivalence testing.

The four tenets of our approach - LLMs + Dynamic Analysis and Code + Test Generation work together as follows. First, C constructs such as dynamic (heap) allocations of unknown sizes, aliasing between pointers, and pointer type casts (and the lack thereof in Rust) lead to challenging translation scenarios in which execution information (e.g., array sizes, aliasing, and types) must be inferred from the code. Our dynamic analysis extracts this semantic execution information and makes it available to the LLM, thus aiding translation. Second, to efficiently leverage the stochastic generative capabilities of LLMs for large codebases, we need *tests* to weed out and/or *repair* incorrect candidates. While tests for top-level functions are often available/easy to construct (due to documented APIs), reliably testing intermediate functions is challenging. Here we once again use our LLM+Dynamic Analysis recipe: we (a) use dynamic allocation analysis to mine intermediate function I/O examples from top-level C tests, and then (b) use LLMs to programmatically map these I/O examples to Rust to construct intermediate function equivalence tests.

This approach of combining incremental code generation with equivalence test generation (for repair and validation) draws inspiration from test-driven development (TDD) practices improving code design and reliability [6, 35]. Notably, just as TDD uses tests to drive implementation, our approach uses intermediate equivalence tests to greedily translate a C codebase while maintaining compatibility with an *invariant*: previously generated Rust code. This allows us to greatly improve the accuracy, and hence scalability, of translation. We compare our technique with other C-to-Rust approaches in Table 1.

| Approach | `safe` **Rust** | **Validity** | **Generation Technique** |
|---|---|---|---|
| C2Rust [17] | ✗ | Cons, ✓ | Rule-based (using C types in Rust) |
| VERT [54] | ✗ | Test, ✗ | LLMs with sampling |
| CROWN [59] | ✗ | Cons, ✓ | Boostrap best-effort analysis on C2Rust |
| Shiraishi, et. al. [44] | ✗ | Test, ✗ | LLMs with sampling |
| Syzygy (ours) | ✓ | Test, ✓ | LLMs with sampling + test feedback |

Table 1. **Comparison of Syzygy with other C to Rust translation approaches**. Legend: ✗= Partial satisfaction, Cons = follows by construction (generation) approach, Test = test-based equivalence validation.

Our approach decomposes (Slicer in Fig. 1) the large codebase into individual *translation units* (e.g., functions, macros, type definitions) and then performs Iterative Aligned Translation (IAT) for each unit. That is, for each unit, we generate its Rust translation that is semantically equivalent and is aligned with respect to the I/O interface (CodeGenerator in Fig. 1). We then test (ArgTranslator, EqTester in Fig. 1) the generated Rust unit against its C counterpart, perform a multi-turn repair if necessary, and move on to the next translation unit when done.

We implement our approach using LLM query APIs for code/test generation, the LLVM toolchain for implementing dynamic analyses, bindgen, and FFI (Foreign Function Interface) for calling C from Rust during equivalence tests. First, we show that using our proposed Dual Code-Test Translation approach, we can successfully translate UrlParser [28], not solved by prior approaches [31, 54]. Next, for our main case study evaluation, we apply the approach/implementation to translate the Zopfli [15] high-performance compression library with 98 functions, 10 structs, and over 3000 LoC. We validate the correctness of our translation by performing equivalence testing on the top-level entry point with one million randomly generated strings used for compression. These tests achieve over 95% line coverage and 83% branch coverage[2] on the source C code. To our knowledge, this is the largest test-validated C to safe Rust translation performed so far.

**Contributions.** In summary, our contributions are as follows:

(1) We develop an automated approach for **test-validated** translation from C to **safe** Rust. Our approach leverages the synergy between superior generative and search capabilities of LLMs and semantic execution information collected using Dynamic Analysis.

(2) Our LLMs+Dynamic Analysis recipe performs **dual code and test generation**. Particularly, we enable reliable equivalence testing by (a) using dynamic analysis to mine intermediate I/O examples from top-level tests and (b) using LLMs to translate these examples to Rust.

(3) We demonstrate the approach by translating the Zopfli high compression library, which has over 3000 LoC, and validate our translation by performing top-level equivalence tests

## 2 BACKGROUND

**C constructs: dynamic allocations, pointer casts, and aliasing.** C allows dynamic memory management using `malloc()` and `free()`. While essential when allocation sizes are not known statically, dynamic (heap) allocations require careful reasoning to prevent memory safety issues. C provides low-level stack/heap accesses using pointers. Pointers can be manually constructed, modified, and cast to other types (including the `void*`). Further, multiple pointers can *alias*, i.e., point to/reference the same/overlapping memory. Dynamic allocations, casts, and aliases challenge translation since they require non-local program reasoning. Our dynamic analyses (for allocation sizes, types, and aliasing) help combat this.

---

[2]We identify a large fraction of uncovered code consisting of error states such as early exits and assertions.

**Memory (Un)safety.** Memory safety requires programs not to encounter (runtime) errors like out-of-bounds accesses (e.g., reading from beyond buffer bounds), use-after-free, and null pointer dereferences. One can exploit such behaviors to mount attacks such as reading memory to exfiltrate data (e.g., private keys), injecting data/code to perform arbitrary execution, etc. C is memory-unsafe because it allows direct memory manipulation using pointers, burdening the programmer with the task of avoiding such errors. Rust addresses these issues through its strong typing rules, enforcing strict compile-time checks, and an ownership model. This is the motivation for converting C to Rust: We want to transfer functionality from legacy C programs to (safe) Rust programs.

**Rust constructs: ownership model, smart pointers.** Unlike C pointers, references in Rust have usage restrictions: they can be mutable, e.g., `&mut T`, or immutable, e.g., `&T`. Rust enforces *borrowing rules*, i.e., you can only have *either* one mutable reference or any number of immutable references to an object at a time, and there cannot be any dangling references. This rule-based ownership model eliminates memory leaks, dangling pointers, and memory safety vulnerabilities.

Borrowing rules make representing certain data structures (e.g., doubly linked lists) challenging. To allow more complex ownership and mutability scenarios, Rust provides smart pointers like `Rc<T>` and `RefCell<T>`. `Rc<T>` is a reference-counted (immutable) smart pointer that allows multiple owners (references) to T-typed data. `RefCell<T>` uses runtime checks to enable *interior mutability*, allowing data to be mutated even when there are immutable references. `Rc<RefCell<T>>` combined allows multiple owners to mutate shared data, which is useful for data structures like doubly-linked lists. Using smart pointers provides flexibility while maintaining safety, albeit with some runtime overhead and the potential for panics if misused.

## 3 PROBLEM FORMULATION

We aim to perform source-level translation from a C codebase to an equivalent `safe` Rust program. We formulate this as a program-synthesis task: *given as specification a C program, synthesize a safe Rust implementation that satisfies it by preserving its functional behavior.*

$$\begin{aligned} \text{Given:} \quad & \text{a C program } P_C \in \mathcal{C}, \quad \text{a set of test inputs } T \subset \mathcal{I}, \\ \text{Find:} \quad & \text{a Rust program } P_R \in \mathcal{R}_{\text{safe}}, \\ \text{Such that:} \quad & \forall t \in T. \ P_C(t) \simeq P_R(t). \end{aligned}$$

We check conditions for a satisfactory synthesis using the notion of "observational equivalence" [1], i.e., the generated Rust program produces equivalent outputs to the C program on a set of test inputs $T$ in the space of inputs $I$. In addition to generating equivalent Rust code, our main motivation for C to Rust translation is eliminating memory safety vulnerabilities. To do so, we disallow any `unsafe` blocks in the generated Rust program. Notably, the two main requirements described above (equivalence and memory safety) may conflict: e.g., some input tests may exercise unsafe behaviour during execution (not compilation). As a consequence of enforcing the `safe` fragment of Rust, we allow undefined behaviors (possibly runtime panics) in such cases. Next, we discuss details about the scope of the input C and output Rust programs we consider:

**Input C Program.** We restrict the input C program to the following conditions:

(1) Acyclic data structures: data structures with pointer cycles require special care in Rust to avoid memory leaks (e.g., `Weak` pointers to free disconnected memory cycles).
(2) No multithreading: requires `Arc`-wrapped references to thread-safely perform borrowing.
(3) No type punning: performing raw memory accesses on untyped or multi-typed memory regions would hinder the best effort type analysis that our approach aims to perform.

These are soft restrictions on our Rust code generation pipeline since we use an LLM's capability to perform the task. However, our current testing and analysis infrastructure is limited to these cases.
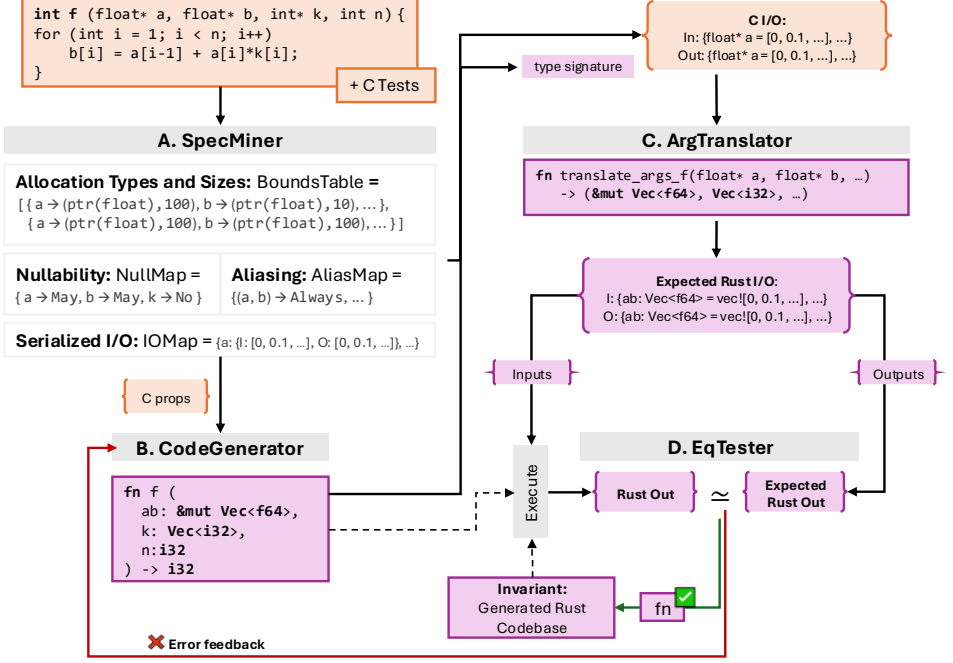
Fig. 2. **Translation Pipeline for a C function f**: Given a C function and tests for the C codebase, first, the SpecMiner uses dynamic analysis to mine input-output (I/O) and property specifications for the translation. These properties (e.g., nullability and aliasing) guide a CodeGenerator to generate a candidate Rust translation of f. Given the translated signature and the collected C I/O, the ArgTranslator generates an translateArgs function that constructs aligned Rust inputs and expected outputs. Finally, the EqTester generates an equivalence test that executes the candidate translation, along with the previously generated Rust code, and checks whether the outputs match the expected values. If the equivalence checks pass, the translated function is committed to the Rust codebase; otherwise, error feedback is provided for multi-round repair.

**Output Rust Program.** We forbid all `unsafe` behaviors in the generated Rust program, irrespective of whether the unsafety is *exploitable* or *unexploitable*. In general, C programs may purposefully leverage unexploitable unsafety for performance or low-level control (e.g., [30]). However, distinguishing between exploitable and unexploitable is extremely challenging; hence, we take the conservative approach here. We discuss the challenges and implications of this in §8.1.

## 4 APPROACH

Our high-level approach, Syzygy is illustrated in Figure 1 with an overview provided in §4.1. At a high level, our approach follows a dual-translation approach–we incrementally translate both code and tests and use execution filtering to ensure validity. This dual translation is guided by synergistically combining LLMs with dynamic analysis.

### 4.1 Overview

*4.1.1 Incremental Aligned Translation.* Our approach performs translation at the granularity of *translation units* such as functions, structs, `typedef`s, and macros. That is, we decompose the C codebase into granular units (Slicer in Figure 1) and loop over them, translating each incrementally.

We call this incremental aligned translation because the translated Rust code aligns with the source C code primarily in terms of the function signatures and behavior, as well as the function call graph, structs, and type definitions. Incremental translation, in this way, decomposes code generation and enables the LLM to focus on individual functions, thus improving accuracy.

*4.1.2   Semantic Type Alignment & Dynamic Analysis.* Translation between imperative typed languages can be conceptually thought of as a combination of two operations: (1) converting/lifting types of variables and (2) translating expressions and statements on these variables. The key challenge here arises due to pointers in C. This is because *pointers hide information regarding the underlying object structure and how it's used.* (Safe) Rust does not allow C-like pointers, thus translation needs to recover this hidden information. In addition to all of these void pointers in C further obfuscate information by also hiding the type of the underlying object. We now discuss this challenge further through the following table.

| Scenario | C type | Possible Rust type |
|---|---|---|
| Array pointer | `(T *)` | `Vec<T>`, `[T; N]` |
| Singleton pointer immutable | `(T *)` | `& T` |
| Singleton pointer mutable | `(T *)` | `&mut T`, `RefCell<T>`, `Rc<RefCell<T>>` |
| Standard objects (e.g., strings) | `(char *)` | `crate::string`, `Vec<u8>`, `[u8; N]` |

*Pointers hide object structure.* A C pointer can be used for a singleton object and arrays. The target corresponding Rust type may vary between these cases. Further, C does not have standard library objects (e.g., strings); rather, these are just represented as `char`/uchar arrays.

*Pointers hide usage information.* For example, if argument `a` in function `f` (Fig. 3) aliased with a pointer from an outer scope, the Rust version of `f` would have to make sure that the shared object (and not a copy) was modified. Although this is easier in C due to full control over pointers reads/writes, Rust imposes strict borrowing and mutability rules that must be followed: (`&`, `&mut`, `RefCell<T>`, `Rc<RefCell<T>>`). In addition to type universality, C pointers enable arbitrary reads and writes, which is not the case in Rust. In Rust, we must specify mutability and nullability and carefully manage aliases between pointers.

*Dynamic Analysis.* While LLM-driven code generation is powerful, we observe that such aspects of programs are difficult to reason about purely syntactically. Thus, we develop dynamic analyses (SpecMiner, §4.2) to collect properties of function arguments such as aliasing, nullability, and allocation sizes. We use these to guide the LLM during code and test generation. We use instrumented execution to collect this since static approaches require complex/infeasible program analysis.

*4.1.3   Sampling for Reliable Code Translation.* For each C translation unit, an LLM generates corresponding Rust code (CodeGenerator, §4.3). However, LLMs can generate incorrect code, making their use unreliable. Prior work has shown that scaling the number of attempts LLMs take at problems can help increase performance [5, 32]. Following this, we use an intuitive approach by *sampling* multiple Rust code generations and *testing* them to filter valid generations. By relying on tests, we transfer the soundness of our approach to the reliability of our tests.

*4.1.4   Reliable Testing by I/O Translation.* However, tests take a black-box approach to verifying code and can have issues such as incompleteness, low code coverage, or even incorrectness. Simply generating tests using LLMs for arbitrary Rust code does not work due to complexities with invalid types, inter-argument constraints like aliasing, and incorrect assertions on expected outputs. We address this by observing that C codebases contain tests, particularly for top-level entry points

(e.g., main). Invoking the entry point results in calls to internal functions whose inputs and outputs can be captured. Consequently, we make testing reliable using LLMs to programmatically translate these C test inputs to Rust rather than generating from scratch (ArgTranslator, §4.4.1). Then, we validate the translated Rust unit using LLM generated value-based equivalence tests (EqTester, §4.4). These intermediate equivalence tests provide two levels of guarantee about the translated Rust unit: (a) it behaves similarly to the source C unit, and (b) it maintains compatibility with all previously generated Rust code (*Invariant* in Fig. 2). If the test fails, we perform multi-round repair (§4.5); otherwise, we commit the translated unit to the Rust codebase.

## 4.2 Dynamic Specification Mining

We use dynamic analysis to assist the LLM when performing codegen-testgen-repair in our approach. This analysis produces different kinds of *specifications* (information) about the execution of the C program. For each C function $f_C$, we gather two kinds of information. Firstly, collect *specifications/properties* of the arguments to $f_C$, such as types, bounds, nullability, and aliasing (§4.2.1). Secondly, we obtain *input-output* examples for $f_C$ (§4.2.2).

   We use the mined specifications in two ways. First, this information guides LLM-driven code generation by providing hints distilled into the LLM's prompt (§4.3). Secondly, collected I/O samples are used in equivalence test generation with LLM generated tests (§4.4). Below, we describe the kinds of specifications we collect using the function f in Fig. 3 as an example.

### 4.2.1 Mining Specifications from C Executions.
Our approach relies on robust analyses to collect information about function arguments. This information is consumed by code and test generation (see §4.2.2). We use dynamic analysis to collect this information (*per execution*) since static approaches require complex program analysis (and may even be infeasible).

   Our approach operates by (a) maintaining a runtime that tracks variables and their properties and (b) instrumenting the C codebase (using the LLVM compiler toolchain [29]) with handlers that interact with the runtime. Then, simply compiling and executing the instrumented codebase allows us to collect information about the types, bounds, nullability, and aliasing for function input and outputs. Depending on the property, we either aggregate or maintain context-sensitive information *per execution* (e.g., nullability is aggregated and bounds are not) Our approach is general and extensible also to collect other kinds of dynamic information which may be useful in code/test generation. We now discuss the particular specifications we collect along with how they are used in the pipeline, using the code in Fig. 3 as a running example.

**Types.**  We first map inputs and outputs to their types. These can be recovered from the LLVM IR types with an instrumentation pass. In particular, we track the following types:

   Types (T) = Primitives (int, uint, char, $\cdots$) | Structs | Data pointers to T | Function pointers

We store a mapping from function arguments to their types in our instrumented runtime. For example, in function f (Fig. 3), we would extract the following type mapping:

$$\text{TypeInfo} = \{\text{arr} \mapsto \text{pointer(float)}, \text{brr} \mapsto \text{pointer(float)},$$
$$\text{krr} \mapsto \text{pointer(int)}, \text{num} \mapsto \text{int}, \text{kernel} \mapsto \text{function}\}$$

   Our type inference process manages several challenging cases. When encountering a malloc call, the initial allocation returns a void pointer (void*) representing untyped memory. Our LLVM pass tracks subsequent bitcast instructions that convert this raw memory to typed pointers. Consider the example in Figure 3, where caller2 allocates arrays of floats and integers. For each allocation, we observe a bitcast from i8* (LLVM's representation of void*) to the target pointer type. By tracking these bitcast operations, we recover the intended type of the allocated memory. We employ

```
1  float kernel1(float a, float b, int k) { return b - a*k; }
2
3  void f(float* arr, float* brr, int* krr, int num, float (*kernel)(float, float, int)) {
4      for (int i = 1; i < num; i++)
5          brr[i] = (*kernel)(arr[i-1], arr[i], krr[i]);
6  }
7
8  float* caller1() {
9      int sizea = 100, sizeb = 10;
10     float* arr = malloc(sizea*sizeof(float));
11     float* brr = malloc(sizeb*sizeof(float));
12     int* krr = malloc(sizea*sizeof(int));
13     // populate arr, brr, krr ...
14     f(arr, brr, krr, min(sizea, sizeb), kernel1);
15     return brr;
16 }
17
18 float* caller2() {
19     int size = 100;
20     float* arr = malloc(size*sizeof(float));
21     int* krr = malloc(size*sizeof(int));
22     // populate arr, krr ...
23     f(arr, arr, krr, size, kernel1);
24     return arr;
25 }
26
27 int main() {
28     float* res;
29     if (...) {
30         res = caller1();
31     } else {
32         res = caller2();
33     }
34 }
```

Fig. 3. **Dynamic Specification Mining**: An example C code snippet which features dynamic allocations (in `caller1` and `caller2`), function pointers (in `f`), aliasing between arguments (in the `main` – `caller2` – `f` call chain). While this information can greatly assist CodeGenerator and EqTester, inferring it statically is challenging. We use a combination of dynamic analyses (see §4.2.1).

a conservative approach for void pointers, mapping them to a base character type while maintaining pointer semantics. This strategy extends to function pointers, where we preserve the complete function signature during analysis. Finally, we treat arrays and pointers equivalently: singleton arrays are pointers (with size 1), and arrays are pointers with a larger size. The allocation size information captures this difference (see below).

**Allocation sizes.** Dynamic allocations lead to a major challenge with C-based translation. Our dynamic analysis tracks allocation size by maintaining a bounds table at runtime.

The bounds table maintains comprehensive allocation information in an interval tree data structure. Each allocation record contains metadata, memory bounds, and inner element sizes for nested types. This information is collected through two mechanisms: compiler-inserted instrumentation for stack variables and allocation function interception for heap memory. We maintain type-aware size tracking for both allocation types, accounting for structure fields and array dimensions. Allocation size information is later used when collecting serialized I/O examples for functions. For our example function `f`, in the `caller1` path, we would record the following bounds information:

$$\text{SizeInfo} = \{\texttt{arr} \mapsto 100, \texttt{brr} \mapsto 10, \cdots\}$$

**Nullability.** We also determine nullability by dynamically analysing argument values in various test (execution) contexts. Given a function with pointer arguments, e.g., `f(int* a)`, we can

determine whether a is ever NULL. If so, we make this information available to the LLM. The same argument can be NULL in some executions and non-NULL in others. We build a map from arguments to booleans, identifying whether the argument is ever NULL (in some execution). This map directs whether `Option` type must be applied on an argument. For example, for function f, if the arr argument is never NULL, then:

$$\text{NullInfo} = \{\text{arr} \mapsto \text{false}\}$$

**Aliasing.** Additionally, we use dynamic analysis to maintain a map that tracks aliasing between pointer arguments to a function. This is a map from pairs of function arguments to one of three possible aliasing occurrences: {Always, Some, Never}. Aliasing between arguments leads to varied signature choices on the Rust side:

(1) `&mut T` (collapsed mutable reference): if the arguments *always alias* then we can collapse the two into a single Rust mutable reference argument.

(2) `Rc<RefCell<T>>`: if arguments Sometimes alias, then, we might need to maintain full flexibility and use smart pointers (`Rc<RefCell<T>>`) to allow aliasing in Rust.

For example, for the function f, we end up with the following map, since arguments arr and brr alias via the `caller2` path, but they don't via the `caller1` path.

$$\text{AliasInfo} = \{(\text{arr}, \text{brr}) \mapsto \text{Some}, (\text{arr}, \text{krr}) \mapsto \text{Never}, (\text{brr}, \text{krr}) \mapsto \text{Never}\}$$

*4.2.2 C I/O Specification.* Recall that we validate the correctness of each translated function through test-based equivalence (against the corresponding C function). This requires input-output test examples for each function $f_C$ in the codebase.

**Valid inputs.** Determining valid inputs to every function $f_C$ in the codebase is challenging. For example, functions can have complex inter-argument pre-conditions (e.g., two array arguments must have the same length), pointer arguments may have to be non-NULL or alias each other in specific ways. Inferring these pre-conditions (towards generating valid inputs) is hard and requires expensive program analysis. LLMs similarly struggle to generate valid inputs satisfying such pre-conditions. We address this by observing that calling the top-level (e.g., main) entry point results in calls to internal functions with valid inputs. Further, the top-level entry point often has much simpler input constraints and/or is better documented. Thus, we invoke (fuzz) the top-level function on multiple inputs and collect function call arguments for each resulting execution.

**Internal function input capture.** To capture I/O examples for function $f_C$, we instrument the entry and exit points of $f_C$ with handlers that dump serialized input and output values. We serialize values of non-pointer objects (e.g., primitive types) directly. For pointers, we dereference the argument (possibly multiple times) to store the internal object. At the function's exit point, we dump both the output and the input arguments to ensure we capture side effects. This gives us an input-output test example: $(I_C, O_C)$. We use these C I/O examples to generate equivalent Rust test inputs ($I_{Rust}$) and to check equivalence (see §4.4).

## 4.3 Rust Code Generation

*4.3.1 High-level strategy: slicing and greedy translation.* As discussed in §4.1, our approach performs incremental aligned translation: we use the program dependency graph to slice the C codebase into individual *translation units* and then translate them starting from the leaves (units with no dependencies) upwards in isolation. Translation units are top-level statements in the C program: functions, struct definitions, macros, `typedef`s. Operating on individual units at a time, rather than entire files or the full codebase, ensures focused translation and increases accuracy.
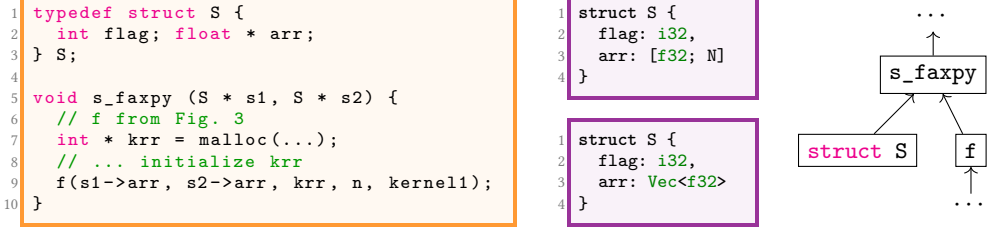
```c
1  typedef struct S {
2    int flag; float * arr;
3  } S;
4
5  void s_faxpy (S * s1, S * s2) {
6    // f from Fig. 3
7    int * krr = malloc(...);
8    // ... initialize krr
9    f(s1->arr, s2->arr, krr, n, kernel1);
10 }
```

```rust
1  struct S {
2    flag: i32,
3    arr: [f32; N]
4  }
```

```rust
1  struct S {
2    flag: i32,
3    arr: Vec<f32>
4  }
```



Fig. 4. Struct field choices induce function signatures higher in the call chain. Left: C struct `S` and a function `s_faxpy` dependent on the struct. Middle: two choices for the `arr` field in the struct. Right: a dependency graph showing relation of `S` with other functions.

**Manual struct definitions.** C structs often form leaf nodes in the dependency graph (the exception being if they use a macro definition). The choice of struct fields, however, induces type/code generation decisions higher up in the dependency graph. Figure 4 illustrates an example where the struct S is used by the function `s_faxpy` (float a-x-plus-y), which in turn calls function `f` (from Fig. 3). The choice of the Rust struct `arr` fields (middle Fig. 4) induces decisions elsewhere in the dependency graph. Depending on whether it is an array or a `Vec`, either `f` must take array arguments, or `s_faxpy` must perform the appropriate type conversions.

These constraints may generally propagate through large parts of the function dependency graph. In our incremental aligned translation approach, incompatibility may be detected later on in the translation process, which may require backtracking (see also Discussion). To avoid this, our current approach requires manual specification of struct definitions. This is not a challenge for functions since we can infer function-level usage constraints from I/O examples and tests.

*4.3.2    Per-unit translation.* For each translation unit, we use LLMs to generate candidate Rust translations. We follow the following two principles to identify valid compiling translations.

**Using information from dynamic analysis.** We collect property specifications using SpecMiner module providing type, nullability, and aliasing information about the function arguments. These properties constrain the signature for Rust translation and we provide them to LLM as additional context. For example, in Figure 5, our nullability map allows LLM to infer *non-intuitive* properties and correctly generate the correct code. Aliasing information similarly directs the function signature in terms of using smart pointer (`Rc<RefCell<RT>>`) or leveraging argument collapsing strategies.
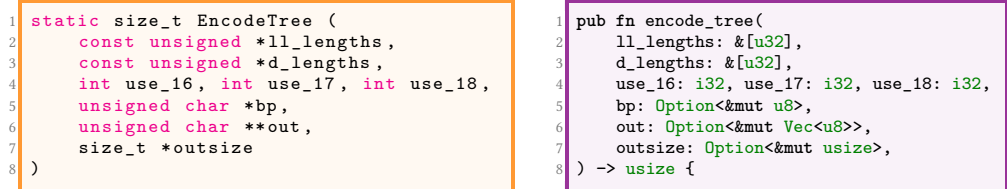
```c
1  static size_t EncodeTree (
2      const unsigned *ll_lengths,
3      const unsigned *d_lengths,
4      int use_16, int use_17, int use_18,
5      unsigned char *bp,
6      unsigned char **out,
7      size_t *outsize
8  )
```

```rust
1  pub fn encode_tree(
2      ll_lengths: &[u32],
3      d_lengths: &[u32],
4      use_16: i32, use_17: i32, use_18: i32,
5      bp: Option<&mut u8>,
6      out: Option<&mut Vec<u8>>,
7      outsize: Option<&mut usize>,
8  ) -> usize {
```

Fig. 5. Nullability information guides Rust function signature. The arguments `bp`, `out`, `outsize` of `EncodeTree` function can be NULL. However, the LLM cannot infer these properties from function context and generates an incorrect signature without `Option`, failing equivalence. Property specifications inferred using our dynamic analysis allow LLM to generate the correct signature on the right (using `Option`).

**Sampling-driven search.** We use LLMs to generate multiple translation candidates for a given translation unit. In the CodeGenerator module, we only focus on achieving compiling solutions and defer execution correctness to the EqTester module.

## 4.4  Rust Test Generation

Given a newly generated Rust function $f_R$, the goal is to check for equivalence between $f_R$ and the source C function $f_C$. We check equivalence using tests created in two steps. First, we translate input C arguments to *equivalent* Rust arguments (§4.4.1). Then, functions $f_C$ and $f_R$ are invoked on these inputs, and their outputs are compared for equivalence (§4.4.3).

```c
1  void square (int* a) {
2    // In-place square
3    *a = (*a) * (*a);
4  }
```

```c
1  int sum (int* a) {
2    int s = 0; int i = 0;
3    for (; i < SIZE; i++) s += i;
4    return s;
5  }
```

```c
1  void caps (char* s) {
2    int i = 0;
3    for (; i < SIZE; i++)
4      s[i] = toupper(s[i]);
5  }
```

```rust
1  fn square (a: &mut u32) {
2    // In-place square using
       mutable reference
3    *a = (*a) * (*a);
4  }
```

```rust
1  fn sum (a: Vec<u32>) -> u32 {
2    let mut s = 0;
3    for i in a { s += i; }
4    s
5  }
```

```rust
1  fn caps(s: &mut String) {
2    *s = s.to_uppercase();
3  }
```

Fig. 6. Challenges with mapping function arguments from C to Rust: both `square` and `sum` C functions have `int*` arguments while their Rust arguments are different since the `a` argument in `sum` points to a single `int` while that in `sum` points to an array of `int`s. The translation is further complicated by cases where C arrays may map to non-arrays in Rust, e.g., in `caps`.

***4.4.1  C to Rust I/O Translation.*** Consider an input-output example ($I_C$, $O_C$) for a C function $f_C$ gathered during dynamic specification mining (§4.2.2). We now want to compare the behavior of $f_R$ against $f_C$ for this example. This requires translating the C argument objects $I_C$ and $O_C$ into equivalent Rust objects $I_{Rust}$ and $O_{Rust}$. The challenge is that *argument translation must adapt to the context-dependent signature of $f_R$ that is chosen by code generation* (§4.3).

For example, in Fig. 6, while all three C functions `square`, `sum`, and `caps` take a single pointer argument, they reference different objects: a singleton integer, an array of integers, and a string, respectively. Consequently, the Rust translations of these functions have different signatures. Particularly, as the `String` example shows, even hardcoding C array pointer to Rust array/`Vec` mappings may be inadequate. As such, it is very hard to cover all possible cases using rule-based mappings. In light of this, we rely on using the LLM itself to generate a translation mapping function, translateArgs, that maps C objects to Rust objects (both for inputs and expected outputs):

$$\text{translateArgs}(I_C) \mapsto I_{Rust} \qquad \text{translateArgs}(O_C) \mapsto \text{(expected) } O_{Rust}$$

*Programmatically translating I/O by generating such translation mapping alleviates the stochastic nature of LLMs, providing more reliability when translating many tests.*

***4.4.2  Argument Construction LLM Tool.*** Certain aspects of the argument translation, however, require broader program reasoning. Examples include aliasing between pointer arguments and allocation sizes (e.g. when size is not explicitly available as another argument). In the `sum` example (Fig. 6), constructing a `Vec` from the array pointer would require size information. Our dynamic analysis (Allocation Sizes) comes to our aid!

| Function | (input) C Type | (output) Auxilliary Rust Type |
|---|---|---|
| `translate_prim_single` | `int`, `char`, ··· | `u32`, `i8`, ··· |
| `translate_prim_vec` | `int*`, `char*`, ··· | `Vec<u32>`, `Vec<i8>`, ··· |
| `translate_string` | `char*` | `String` |
| `translate_struct` | `struct T*` | `Rc<RefCell<Struct>>` |
| `translate_1d` | `T*` | `Vec<Rc<RefCell<RT>>>` |
| `translate_2d` | `T**` | `Vec<Vec<Rc<RefCell<RT>>>>` |

Table 2. Argument Construction API Functions: functions to translate primitive singletons and arrays, strings, and structs. For n-D arrays, we provide functions to translate them into (possibly multi-dimensional) `Vec` objects. In these cases, we preserve aliasing matching C using `Rc<RefCell<RT>>` (Rust smart pointers) - here the `RT` is the Rust type that the C type `T` maps to.

We bake the dynamic analysis information (e.g., AliasInfo, SizeInfo) into an Argument Construction API. These API functions convert C objects into *auxilliary* Rust objects that adhere to size/aliasing relations between arguments. API functions are exposed as tools to the LLM. The LLM can then write a wrapper translateArgs function that coerces the auxiliary Rust objects into the final objects that match the function signature.

We provide the API functions in Table 2. These include functions translating singleton primitive objects (e.g., integers), strings, and structs. For pointers referencing n-D arrays, we provide functions to translate them into (possibly multi-dimensional) `Vec` objects. For the `sum` and `caps` functions in Fig. 6, the LLM can use the `translate_1d` and `translate_string` API functions in translateArgs. The API functions use the dynamic analysis under the hood to infer the size of the `Vec` and the length of the string. The API functions also use the dynamic analysis information (e.g., AliasInfo, SizeInfo) to translate the aliasing properties between C objects to the Rust objects. For example, if two C struct pointers `struct s * a, b` reference the same struct object, the translation API will create a single Rust struct (`rust_s`) object and return two `Rc<RefCell<rust_s>>` references to it. Below, is an example LLM-generated translateArgs for `sum`. Notably, the APIs are helpful high-level primitives, while LLM is allowed to write arbitrary Rust when constructing Rust objects (e.g., converting `Vec<i32>` to `Vec<u32>` in the code below).

```
pub unsafe fn translateArgs_sum(a: *mut c_int) -> Vec<u32> {
    let rust_a = translate_1d::<c_int>(
        a as *mut c_void
    ).expect("Failed to translate a")
    .into_iter()
    .map(|x| x as u32).collect::<Vec<u32>>(); // Convert from Vec<i32> to Vec<u32>

    rust_a
}
```

**4.4.3 Equivalence Testing.** With a valid translateArgs, we can execute $f_C$ and $f_R$ on equivalent inputs. Here, we observe that functional equivalence between $f_C$ and $f_R$ can be elegantly demonstrated through a commutative relationship centered on the translateArgs function.

$$
\begin{array}{ccc}
\text{I}_\text{C} & \xrightarrow{\ f_C\ } & (\text{I}_\text{C}, \text{O}_\text{C}) \\
{\scriptstyle\text{translateArgs}}\Big\downarrow & & \Big\downarrow{\scriptstyle\text{translateArgs}} \\
\text{I}_\text{Rust} & \xrightarrow{\ f_R\ } & (\text{I}_\text{Rust}, \text{O}_\text{Rust})
\end{array}
$$

That is, for $f_C$ and $f_R$ to be equivalent, directly executing $f_C$ and then translating its output must produce the same result as first translating the inputs and then executing $f_R$. Consequently, we ask

an LLM to generate an equivalence test that performs this validation logic and `assert` that each of the C and corresponding Rust outputs agree. We limit the assertions to perform *value-based equivalence* only. Below is a simple example of an equivalence test for `caps` in Fig. 6:

```rust
pub unsafe fn eqtest_caps(filename: &str) -> bool {
    let c_pre_arg0 = load_pre_args_from_json(filename);          // Load C inputs
    let mut rust_string = translateArgs_caps(c_pre_arg0);        // Translate to Rust input

    let c_post_arg0 = load_post_args_from_json(filename);        // Load C outputs
    let expected_rust_string = translate_args_caps(c_post_arg0); // Translate to expected Rust output

    caps(&mut rust_string); // Call Rust function with translated input

    assert_eq!(rust_string, expected_rust_string, "Rust string does not match expected Rust string");

    true
}
```

### 4.5 Multi-round & Rejection Sampling

Overall, with this pipeline, specification mining (§4.2) → code generation (§4.3) → testing (§4.4), we can identify whether a generated $f_R$ is a valid translation of $f_C$. We use multiple rounds of this pipeline with error feedback to correctly translate all functions.

*Rejection Sampling.* As mentioned in §4.1, we make use of sampling to scale LLM inference and verify generated solutions. We use this rejection sampling approach at each pipeline module that invokes an LLM, as illustrated in Figure 7. For each module, we first sample a set of solutions, i.e., rust translations for CodeGenerator, translateArgs for ArgTranslator, and equivalence tests for EqTester. We then use either compilation or execution as the signal to filter out bad generations and pass the filtered set to the next module.

*Error Feedback.* Finally, failing equivalence tests are used as error feedback. However, since these functions can produce large outputs, we perform a `diff` operation between the true and desired output to isolate the discrepancy. We provide these diffs and assertion messages to the LLM and ask it to regenerate the (incorrect) function. Consider an anecdotal example for equivalence-based feedback from our Zopfli case study (§6.2). Below, the `BoundaryPMFinal` function grabs the `next` field from a `pool` of nodes (left). However, in the first round, the LLM interprets `node->next` as incrementing the index field (right), corresponding to the `next` iterator in the C code.

```c
if (lcount < nsymb && sum > leaves[lcount
    ].weight) {
  Node *newch = pool->next;
  // ^^^^^^^^^^^^^^^^^^ this line
  Node *oldch = lists[index][1]->tail;
  ...
}
```

```rust
let newch = pool
  .next.get(pool.index)
  .expect("Pool has no more nodes")
  .clone();
// Move to next free node in the pool.
pool.index += 1;
// ^^^^^^^^^^^ this line
```

Here, our testing approach allows deserializing iterators with index information and catching bugs in the LLM's code, which unnecessarily updates the iterator's position. As a result, providing error feedback allowed the LLM to correct the code using our diff-based localization.

### 5 IMPLEMENTATION

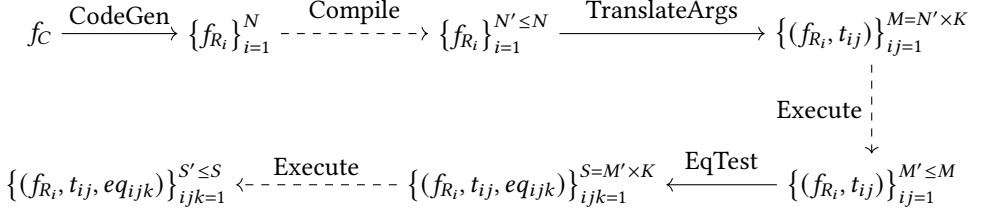Following, we describe the implementation details of the various components of our approach.

$$f_C \xrightarrow{\text{CodeGen}} \{f_{R_i}\}_{i=1}^{N} \dashrightarrow^{\text{Compile}} \{f_{R_i}\}_{i=1}^{N' \leq N} \xrightarrow{\text{TranslateArgs}} \{(f_{R_i}, t_{ij})\}_{ij=1}^{M=N' \times K}$$

$$\downarrow \text{Execute}$$

$$\{(f_{R_i}, t_{ij}, eq_{ijk})\}_{ijk=1}^{S' \leq S} \xleftarrow{\text{Execute}} \{(f_{R_i}, t_{ij}, eq_{ijk})\}_{ijk=1}^{S=M' \times K} \xleftarrow{\text{EqTest}} \{(f_{R_i}, t_{ij})\}_{ij=1}^{M' \leq M}$$

Fig. 7. Illustration of the rejection sampling approach used to translate a C function $f_C$. At each stage, the pipeline oscillates between a *generation* ($\rightarrow$) and *verification* ($\dashrightarrow$) step that filters valid generations. The process starts by the LLM generating $N$ candidate Rust functions, from which compilation filters valid programs ($N' \leq N$). Then the ARGTRANSLATOR samples $K$ `translate_args` functions for each compilable Rust function, resulting in $M$ function-translator pairs. Execution filters these to $M'$ valid pairs, which the EQTESTER expands into $S$ triples by sampling $K$ equivalence tests per function-translator pair. A final execution phase filters these to $S'$ triples, yielding successfully translated and equivalent Rust functions.

## 5.1 SLICER

We implement our program slicer using CLANG-17 that consumes an entire C codebase, analyzes the definitions and usages, and builds a topologically sorted dependency graph of all the code elements in the C program. Additionally, we implement an iterator over the graph that progressively traverses and yields code elements in dependency order. By performing a topological sort on this graph, we can identify an incremental translation unit and determine its dependency slice (the set of units it depends on). Additionally, we incorporate extra dependency edges between function types and the functions matching those types, as inferred via dynamic analysis.

## 5.2 SPECMINER

SPECMINER module of our pipeline is responsible for collecting I/O specifications for C functions. These I/O are later used for inferring various property specifications such as nullability related to the function. Since C does not support reflection and even primitive information such as pointer sizes and types are not directly available, we use dynamic analysis to collect this information. The SPECMINER module of our pipeline is responsible for collecting input/output (I/O) specifications for C functions. These I/O specifications are later used to infer various function properties, such as nullability and aliasing. Since C does not support reflection, and even primitive information like pointer sizes and types are not directly available, we employ dynamic analysis to collect this information. In particular, we implement an allocation tracking LLVM pass (using LLVM-14[3]) to collect the bounds and type information for all allocations in the program. We then implement a C++ runtime to access this information for serializing the C function arguments and outputs. Because C functions can be stateful and may modify their arguments, we serialize the arguments both before and after function execution along with the return value.

## 5.3 CODEGENERATOR

We implement our RUST code translation module using an LLM based *parallel* sampling and filtering approach. In particular, for a given C translation unit, we use the SLICER to collect its immediate parents. We then prompt the LLM with the C source code elements, the corresponding RUST elements already translated, and the dynamic analysis annotations (such as nullability and aliasing information about arguments) to generate the translated RUST code.

---

[3]Note that opaque pointers in more recent LLVM versions do not allow easily accessing type information.

We use parallel sampling and a multi-turn repair approach to generate the Rust translations. In particular, we sample multiple candidate translations and use Rust compiler with `#![forbid(unsafe_code)]` directive to identify safe compiling translations. We keep the compiling translations and prompt the LLM with the compile-error feedback to refine the failing translations (with backoff to explore more candidate translations). We repeat this procedure across multiple rounds to generate a capped number of minimum compiling translations. This approach allows us to generate a diverse set of translations efficiently which can allow finding a *correct* (that is passing the EqTester) translation faster. Particularly, we can defer needing execution feedback based multi-turn sampling after EqTester if any one of the candidate translations passes the EqTester.

## 5.4 ArgTranslator

We use `bindgen` to generate Rust function signatures from the C function signatures and employ FFI to access the C functions and their arguments. We then use LLMs to programmatically map the types and arguments between the C and Rust functions. Specifically, for a given C and its candidate Rust translation, translation, we provide the LLM with the *aligned* C and Rust function contexts and ask it to generate a program that maps the C arguments to appropriate Rust arguments. We use successful execution of the generated translation program (that is, we can load and map the C arguments to Rust arguments) as a filter to select the candidate Rust functions.

## 5.5 EqTester

For a given C function, its candidate translated Rust function, and the corresponding type alignment function, we prompt the model to generate a test function which checks whether the C and Rust functions are equivalent. We perform value-based equivalence checks by implementing the `PartialEq` traits to Rust types, enabling the use of the built-in equality operator to assert equivalence (performing nested equality comparisons internally). We use these generated tests to filter the candidate Rust functions that successfully compile and pass the equivalence tests. If none of the candidate functions pass the tests, we provide the model with execution error feedback and iteratively repair the translation over multiple rounds.

We run our experiments on a 96 core Intel Xeon machine with 720 GB of RAM. We use o1-preview and o1-mini to perform all the steps unless specified otherwise.

## 6 EXPERIMENTS

In this section, we discuss two experimental results. First, in §6.1, we present a case study on translating UrlParser, a modest C program comprising over 400 lines, which was studied in prior work [31]. Next, in §6.2, we present our results from translating Zopfli, an optimized C compression program with over 3000 lines.

## 6.1 UrlParser Case Study: Motivating Dual Translations

We translate the UrlParser program [28], a C application that parses URLs and extracts their components (protocol, domain, path, query, and fragment) from a URL string. To gather deeper insights, we run our approach with human intervention to identify failure models and solutions. This program has been used in prior work [31], where the authors applied existing LLM-based approaches, namely Flourine [13] and VERT [54], and reported challenges in generating correct translations. In particular, they faced difficulties in program decomposition, where inconsistent translations of decomposed functions resulted in compilation errors.

*Incremental sampling and filtering alleviate challenges.* We use our Slicer and CodeGenerator modules to incrementally generate compilable translations. By providing dependency context to the

LLMs during translation and leveraging the multi-turn compiler feedback-based CodeGenerator module, we can generate a compilable translation for the UrlParser program.

*Compilable code translations do not suffice.* However, at this stage, we do not utilize execution filtering when generating individual function translations. We run tests for the top-level Rust `url_parse` method and find that none of the candidate translations passes test test case.

*Testing alleviates challenges.* To address this issue, we semi-manually construct tests for individual functions in the UrlParser program and use these tests as additional filtering criteria during translation. These tests are translated from the UrlParser repository by prompting LLMs with C tests and incorporating manual interventions to ensure test correctness. By using test execution as additional feedback, we can generate a correct translation for the top-level `url_parse` function, thereby demonstrating the effectiveness of our dual translation approach.

## 6.2   Zopfli Translation

hWe translate the C program for the Zopfli compression algorithm [15], which is known to achieve superior compression ratios by extensively optimizing the compression process. The codebase consists of over three thousand lines of C code (excluding comments), comprising ninety-eight functions and ten structs, and spans over twenty-one files. It provides a challenging testbed for evaluating our approach due to the diversity of the source code constructs, including heap-based data structures (such as linked lists and array iterators), function pointers, and `void*` arguments while still agreeing with our assumptions about acyclic data structures, concurrency, and lack of `void*` abuse (type punning). The C code implements nuanced algorithms, including LZ77 compression, Huffman encoding, and block splitting. Following, we detail our implementation and results.

**6.2.1   Implementation.** We use the `unifdef` tool [39] to preprocess the codebase and standardize the `#ifndef` preprocessor configuration options. This process addresses a combination of system configurations (operating system, compiler) and optimization settings implemented in Zopfli. Next, we run Syzygy on Zopfli, using `ZopfliDeflate` as the top-level entry point interfacing with the core deflate algorithm. This function receives the input string to be compressed, along with `ZopfliOptions` configurations that determine how block splitting is performed. We semi-automatically collect 26 test inputs for the `ZopfliDeflate` function. These inputs achieve 88% line coverage and 70% branch coverage on the original C program.

We use SpecMiner to collect I/O and property specifications for every function in the C code using the above-mentioned tests dynamic allocation analysis. Next, we iterate on the translation units in dependency order and run our pipeline to generate Rust translations for each unit and test for equivalence. Since we do not have equivalence tests for non-function translation units, such as structs, global assignments, and macros, we manually verify translation correctness. In particular, we manually construct the appropriate Rust structs and check other global assignments or macros as necessary. This process discovered a bug in the `ZOPFLI_APPEND_DATA` macro where LLM translation did not handle a corner case in the original code which we repaired manually.
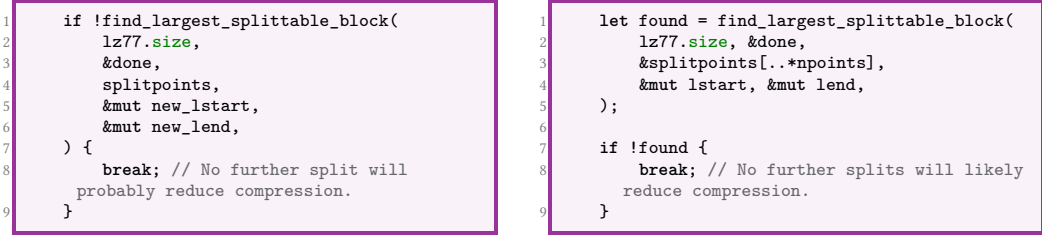
**6.2.2   Results.** We run our approach and translate Zopfli in about 15 hours costing about 2500\$[4].

**Equivalence Test Suite.** To ensure that our translation is correct, we collect a comprehensive test suite for the `ZopfliDeflate` function comprising 1000000 compression inputs ranging between $1e1$ and $1e7$ characters (having a combined size over 20GB). We measure coverage of our larger tests and find that they achieve 95% line coverage and 83% branch coverage. Note that we identified a considerable portion of uncovered code regions (example 5.9% of the branches) as error regions

---

[4]We believe the translation can be done within 1500\$ with better hyper-parameters (choice of models, number of samples) and more robust infrastructure of our pipeline. Given the high cost, we do not study pipeline optimizations at this time.

(like `assert` statements, `exit` branches) and thus supposed to be unreachable. We use these to check equivalence in compression ratios between the C and Rust programs.

**Test Results and Repair.** We observed run-time exceptions for a fraction of our tests. We use the failing equivalence test and our SpecMiner module to identify the *innermost* function that fails the per-function I/O specification and *restart* our pipeline from the failure point. Particularly, we identify the failure in `zopfli_block_split_lz77` function and prompt the model to repair the failing test. This bug was caused by less comprehensive tests used in the previous run of the pipeline, and we find that LLM could easily fix this with a simple and local change: slicing the `splitpoints` `Vec` (shown in Figure 8). Note that this repair does not follow the greedy translation principle from our approach (since functions dependent on `zopfli_block_split_lz77` already exist) and can have cascading effects. However, we found that the bug mentioned above and the corresponding fix were local. After performing this repair, we successfully passed our entire test suite, thus providing high confidence in the correctness of our translated code.

```
1   if !find_largest_splittable_block(
2       lz77.size,
3       &done,
4       splitpoints,
5       &mut new_lstart,
6       &mut new_lend,
7   ) {
8       break; // No further split will
     probably reduce compression.
9   }
```

```
1   let found = find_largest_splittable_block(
2       lz77.size, &done,
3       &splitpoints[..*npoints],
4       &mut lstart, &mut lend,
5   );
6
7   if !found {
8       break; // No further splits will likely
     reduce compression.
9   }
```

Fig. 8. We discovered some runtime exceptions when running our equivalence test suite. We used our SpecMiner approach to find the *inner-most* function which failed the *extracted* function-level I/O specification and re-run the pipeline from that point. We found that LLM was able to successfully repair the rust code by appropriately performing the slicing on `splitpoints` `Vec` (line 4 on the left and line 3 on the right). Moreover, after applying this local change, our Rust code passes the entire test suite of 1000000 tests.

**Pass-rates.** We study the quality of the translation candidates across different functions. Particularly, we measure two metrics – compilation pass rate and execution pass rate. Compilation pass rate is computed as the number of translations that compile against the total number of translations generated. The execution pass rate is computed as the number of translations that pass the equivalence test compared to the total number of compiling translations.

Figure 9 depicts a box plot highlighting the mean, median, and distribution of pass rates over different function samples. First, we observe that the mean pass rates are high, ranging over **75%** for both getting a compilable solution and an executable solution. This indicates that our approach can generate high-quality translations for a large fraction of the functions somewhat easily (with high pass rates). However, this distribution is quite skewed with a long tail of *challenging* function with low pass rates (below 20%).

*6.2.3 Analysis.* Next, we analyze the quality and efficiency of translations.

**Qualitative Analysis.** Our translated Zopfli program ranges over 7k lines (including overly verbose comments and docstrings). Interestingly, we observe that LLMs eagerly modify the C program's logic (usually in the spirit of optimizing or simplifying), often leading to bugs. We address these issues by prompting the models with strict warnings to limit such behaviors. For example, since C does not provide native support for data structures like vectors, it maintains additional attributes like the length of an array to mimic the functionality. LLMs are eager to write Rust code that mitigates using external length (and using `Vec` `length`) but do so inconsistently causing bugs.
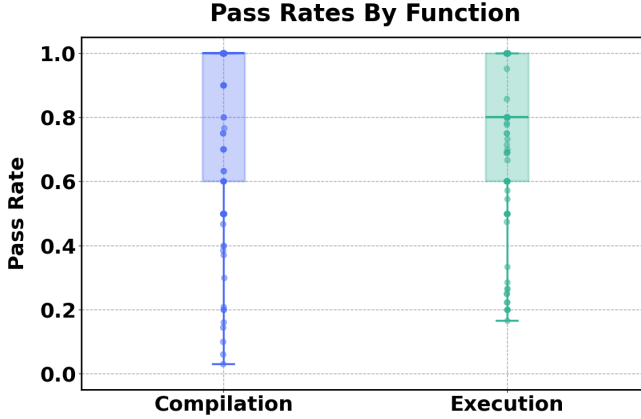
**Pass Rates By Function**



Fig. 9. **Box plot demonstrating pass rates for Code Compilation and Test Execution.** We measure the per-function pass rates for compilation and test execution. Note that for test execution, we use the compiling solutions as the set of submitted solutions (denominator when computing pass rates).

We also note LLMs struggle to generate correct (or even compiling) code for long functions, often missing or skipping out the translation of some intermediate component. One particularly nefarious case for this originates from the difference in C and Rust `for` loops. Specifically, in C, you can modify the loop iterator variable inside the loop body. Rust does not provide this option and the translation needs to map the `for` loop to `while` loop, handling incrementing the iterator at all loop `continue` statements, often not managed correctly in long programs by LLMs.

**Efficiency.** We next compare the efficiency of the original and translated programs on two test suites – first with six strings formed by repeating `'a'` varied number of times (between $1e1$ and $1e6$ times) and second with twelve random strings of varying lengths (between $1e3$ and $1e6$ characters). Additionally, we compare the execution times under default compilation configuration ('`gcc -O0 -g`' and '`cargo debug`') and optimized compilation configuration ('`gcc -O3`' and '`cargo build --release`') for both the C and Rust implementations.

We find that the translated Rust code is slower, with larger slowdowns, in the default compilation configuration compared to the optimized compilation configuration. Next, even with the optimized compilation configuration, the Rust code is up to 3.67x slower than the C code. We study this further by comparing the flamegraphs for the default compiled versions. We estimate a considerable fraction of slowdown from time spent on `Vec` allocations and bound checks in Rust. Additionally, we identified `ZopfliUpdateHash`, a function that modifies data structure containing large arrays, as a large contributor to this slow-down. Note that Rust might already be optimizing some of these issues, and understanding performance differences between the optimized versions would require more careful time measurements.

Table 3. Performance Comparison of C and Rust Implementations

| Input Type | Default | | | Optimized | | |
|---|---|---|---|---|---|---|
| | C | Rust | Slowdown | C | Rust | Slowdown |
| Repeated Input | 0.17 | 1.56 | 8.9x | 0.040 | 0.059 | 1.47x |
| Random Input | 0.242 | 3.336 | 13.8x | 0.094 | 0.330 | 3.67x |

*6.2.4  Ablations.* Following, we describe ablations for our approach.

**Choice of Models.** Recall (from §5) that we use O1 models for our experiments. We also experiment with GPT-4O models to perform translation. Since GPT-4O models are less powerful, we increase the sampling budget to collect more candidate translations. Additionally, since we discovered that GPT-4O is insufficient for performing execution-feedback-based repair operations, we fell back to O1 models for repair. We run our pipeline, passing all intermediate tests, and generate an alternative translation for Zopfli. Notably, using significantly cheaper GPT-4O models, we can generate a correct translation for Zopfli in $ < 800$. However, the generated translation is noticeably slower than the original C implementation and crashes on some long test inputs. We conclude that O1 models produce more reliable and higher-quality translations and hope to improve the efficiency of our pipeline by using robust combinations of O1 and GPT-4O models in future work.

**Choice of using dynamic analysis and Testing.** Recall that we use dynamic analysis to construct intermediate tests and incrementally generate correct code. As an ablation, we remove the testing module directly to generate code without any intermediate filtering except compiler checks, resembling the approach from [44]. Using this approach, we can successfully generate a compiling version of Zopfli using LLM queries within 100$. However, we find that the generated code does not run even for trivial inputs such as `"Hello, World"` and crashes with `index out of bounds` error. This aligns with our findings from Figure 9 where compiling code solutions achieves only 80% median pass-rate with long-tail achieving < 20% pass-rate.

# 7  RELATED WORK

## 7.1  C-to-Rust Interest and Applications

C to Rust full code translation has recently garnered much interest owing to memory safety vulnerabilities [47] in C, strong typing and ownership in Rust, and the relative ease of combining C and Rust through Clang compilation toolchains, and the C-Rust FFI (Foreign Function Interface). In particular, the NSA (US National Security Agency) has strongly advocated migration to Rust, and DARPA has launched the TRACTOR program [12] aimed at automating translation.

   We based our main case study on Zopfli [15], a high-performance compression library by Google. We chose Zopfli because (a) compression libraries form a project milestone of the DARPA TRACTOR program, and (b) Zopfli has been manually translated to Rust [7] with the experience well-documented. We note that our auto-translation radically differs from the handwritten implementation (e.g., in terms of code architecture, organization, and Rust features used). There is also emerging user-study research comparing human expert-translated and auto-generated translations (e.g., using LLM-based techniques) [31].

   Translation from C to Rust is considered especially relevant for sensitive codebases such as cryptographic and network libraries and OS kernel modules/drivers. However, the opinion is still divided on *how much* and *what part* of the source C codebase to migrate. For example, Li et al. [30] makes the case that certain low-level utilities (e.g., drivers) are best left in C since migration can lead to more subtle bugs (e.g., issues with memory mapping/address spaces).

## 7.2  Automated C-to-Rust Translation

We provide a summary of C-to-Rust translation approaches in Table 1. The work on C-to-Rust translation closest to ours is [44]. They, too, perform LLM-driven C-to-Rust translation. However, unlike us, they focus on sampling alone, without intermediate testing (for internal functions). As a result, they achieve lower accuracy on the validation tests. One of our key insights is that incremental translation combined with internal function tests can result in major accuracy gains

for moderate-large (multi-file/function) codebases. Accordingly, we enable internal equivalence tests by collecting C I/O examples (§4.2.2) and translating them to Rust (§4.4.3).

VERT [54] also uses LLMs to generate Rust code from multiple source languages (including C) using an MSWasm-based (memory-safe WebAssembly) testing infrastructure. However, they also do not perform incremental translation/internal function tests. Further, they evaluate their approach on much smaller (<200 LoC) code sizes. Related to this is [13] which performs LLM-driven translation with differential testing and repair (when top-level tests fail). They evaluate the performance of different LLMs with respect to accuracy and repair feedback strategies. However, they do not perform intermediate tests and only achieve limited success on small benchmarks.

Other C-to-Rust translation approaches include c2rust [17], which translates C to *unsafe* Rust making heavy use of C-like constructs and C-types through C-Rust FFI. Beyond `unsafe`-ness, this results in artificial and less interpretable code. Immunant and Galois aim to extend c2rust to produce *safer* Rust [18]. They, too, identify a use-based (semantic) type correspondence between C pointers and Rust reference types, similar to our Semantic Type Alignment (ref. §4.1.2). Further, very recently, they too have explored nullability analysis (as their blog [14] indicates), albeit using LLMs. We, on the other hand, use classical dynamic analysis (§4.2), which we believe is more predictable/reliable. CROWN [59] takes the output of c2rust and performs best-ownership effort analysis and rule-based rewriting to convert it to `safe` Rust. While their approach scales well, usage of `unsafe` remains.

## 7.3  LLM-driven Code Translation

LLM driven code translation has received considerable interest in recent years. These works have explored training LLMs for code translation [40, 41, 49], leveraging compiler representations [46], and directing prompting based approaches [24, 26, 48, 56]. However, these works have primarily focused on translating simple competition or interview-style programming problems. Cassano et al. [8] used a rule-based test translation to translate small (single-function) program snippets from high-resource to low-resource languages.

Two recent works use LLMs to perform repository-level translation with test-based validation. Particularly, Ibrahimzada et al. [16] study Java to Python and Zhang et al. [58] study Go to Rust translation. While our work shares themes of incremental translation and test-based validation with these works, key aspects of our approach are unique to C-to-Rust translation as we now discuss.

**Semantic execution information aids code/test generation.** We observe that C constructs such as dynamic allocations, pointer-based direct memory accesses, aliasing, and nullability make interpreting (and hence translating) C code challenging. Analyses that extract these properties, and exposing them to the LLM can enable more accurate, and hence, scalable translation.

**Dynamic analysis enables collecting semantic information.** While semantic execution information and function I/O aids code generation and testing respectively, extracting these is challenging due to weak typing and pointer casts in C. We tackle this challenge this by developing a dynamic specification mining infrastructure (§4.2) that collects this information.

While Java-to-Python and Go-to-Rust have their own challenges, memory management patterns translate fairly directly. Furthermore, these languages have rich standard libraries with parallel high-level constructs, unlike C's minimal set. Finally, our SYZYGY approach to translation attempts to scale LLM inference using techniques such as repeated sampling and execution-feedback-based repair to achieve completely valid and equivalent translations.

## 7.4  Agentic Code Generation

There is a tremendous amount of research on LLM-driven code generation. Here, we highlight relevant works on agentic code generation and direct readers to appropriate surveys on the topic [25, 33]. LLM based programming agents have been gaining popularity with benchmarks [20, 21, 27, 42]

and agentic approaches for solving them [50, 51, 53, 55]. Our approach indeed forms a non-agentic pipeline for solving challenging long-horizon code (translation) problems. Bairi et al. [3] proposed CodePlan which performs repository-level code editing using program analysis (dependency graphs). Recent works have also focused on inference-optimal compute usage [11, 45, 52, 60] to optimize inference costs, which is orthogonal to our work but nonetheless an important aspect. Finally, we believe many ideas from traditional code generation work will transfer to the code translation domain [9, 10, 19, 22, 23, 37, 38, 43, 57, 61].

## 8 DISCUSSION

### 8.1 Challenges and Future Work

Now we enumerate some unalleviated challenges we plan to address in future work.

**Golden Translation Specification.** Note that we require that the generated Rust program satisfy (a) equivalence on test inputs and (b) not use `unsafe` Rust. In general, requiring no-`unsafe` may be an overly strong constraint. Some unsafe executions may be acceptable if they are unexploitable, i.e., user-facing functions cannot be invoked in a way that leads to an attack. Low-level C code often uses (safely abuses) such behaviors. Prohibiting unsafe constructs in such cases can lead to artificial Rust code that may suffer from other bugs (e.g., due to subtle differences in C/Rust kernel allocation methods [30]). Thus, a golden (ideal) specification for C to Rust translation would strike a balance by only forbidding exploitable unsafe behaviors.

Determining which usages of unsafe behavior are unexploitable is extremely challenging, and requires a combination of security (attacker) modelling and program analysis. A more lightweight approach would be to allow the user to annotate certain C functions, e.g., as `NO_TRANSLATE`, meaning that the said function should be kept in C and integrated with the rest of the translated Rust codebase through the FFI (Foreign Function Interface). We believe that our infrastructure can support this with relatively low effort. In general, future work may take a more nuanced approach to thread the needle between ensuring C-equivalence and enforcing `safe` Rust.

**Translation performance, Cost and Speed.** Our approach relies on LLM capabilities to perform the translation and uses program analysis and testing to scaffold the process, improving reliability. Here, we identify that while LLMs can be unreliable and make mistakes, we can make them reliable by sampling many candidate translations and filtering them using tests. Additionally, we decompose the translation process into smaller parts, which further reduces the complexity of the translation, improving the reliability of LLM based translation. Thus, our performance is bounded by Pass@Any of LLMs–often considerably higher than Pass@1 [5].

This increases translation time and cost, which becomes crucial as we scale to larger programs. Future work should explore how to orchestrate the translation *search* process to reduce the cost of translation following *agentic* approaches in existing LLM for code literature [50].

**Dependency Ordered Translation.** We translate the code in a greedy topo-sorted order of the program dependency graph (§4). The greedy algorithm reduces to combinatorial search complexity but may lead to sub-optimal translations. Particularly, as described in §4, accurately translating structs requires global context of how the struct is used in the program (providing information about how to refactor the struct fields). We currently resolve this by manually translating the structs and aim to explore how LLMs and usage analysis can guide the translation of structs in the future. Next, while translating functions, we identify various dynamic analysis-based property annotations (nullability, aliasing, etc.) that can guide the translation process without requiring the entire program context. However, this may again lead to sub-optimal translations due to *bad* translation choices made early in the translation process. For example, translating a pointer argument to an array early on might lead to conflicts when later translations require the argument

to be a vector. This would add unnecessary type casts, reducing program efficiency. Future work should explore how to solve this global consistency problem in the translation process.

**Test-based Equivalence: Incomplete specifications.** It is well known that test-based equivalence is prone to issues of incomplete coverage. Accordingly, our test-based validation (with randomly sampled tests) also has some chance of missing bugs in translation. We perform coverage analysis for our test suite and report the numbers in §6. For large (multi-file, multi-function) codebases, test-based/fuzzing-based validation remains the primary scalable testing strategy. Future work can improve testing by using better sampling/fuzzing heuristics to improve coverage.

**Test-based Equivalence: Overly strict inner equivalence.** While test-based validation of the final translation can lead to incomplete coverage, enforcing equivalence at intermediate functions results in the opposite issue: it may be unnecessarily strict. Relaxing this may result in simpler, more interpretable/performant code. However, determining which equivalence checks to relax (and how) is challenging and requires knowledge of future (downstream) context. Our current dependency-ordered greedy translation approach provides limited flexibility for such relaxations.

*Example.* A particularly interesting instance is the `zopfli_append_data` function. The C code calls `realloc` to double the allocated memory of arrays in case the size runs out while appending elements (mimicking vectors) and stores the current size of the array separately. In Rust, `Vec` is a natural choice for mapping the C array, which internally performs the doubling logic. However, to maintain equivalence between the C and Rust code, the Rust code also needs to artificially double the array size and use a separate variable to store the *size* of the vector. Notably, this leads to non-idiomatic Rust code, and the LLM confuses `Vec.length` as the size of the vector instead of using the secondary size stored in a variable.

**Cascading post-generation repairs.** Recall that a set of equivalence tests guides our greedy dependency-ordered translation. Once we have generated the complete Rust codebase (post-generation), we validate the entire translation by performing equivalence tests from the top-level codebase entry point. However, some of these validation tests may still fail. We discuss such a case in our experimentation section (§6). While we can localize the error using our EqTester infrastructure, repairs to the erroneous function may cascade to its dependent functions. In such cases, we want to perform *minimal* edits to the translated codebase so that the validation tests pass. While we manually strategize this in our experiments, future work can develop more systematic techniques for LLM-guided repair, viewing this as an agentic LLM system with edit primitives/actions.

**Type Punning and `void*`.** C provides full memory control using manually constructed addresses. Interpreting programs written in this way is difficult and forms an active research area (e.g. software reverse engineering survey [36]). Even when a program does not use raw memory accesses, `void*` pointers can obfuscate information about the underlying object. Our approach, particularly our dynamic analyses (e.g., §4.2, type analysis) relies on information being available during compilation. Thus, type punning and rampant usage of `void*` can cause imprecise/incomplete analysis and lead to incorrect type inference. While this forms a limitation, we do not see a straightforward solution. We believe that future work that aims to resolve this would need to leverage reverse engineering research/powerful (and likely costly) program analyses.

**Cyclic Structs and Multi-threading.** Our current approach does not support cyclic C structs and multi-threading. Cyclic structs require careful handling in Rust with `Weak` references to break cycles and avoid memory leaks. Similarly, multi-threading requires careful handling of shared mutable state and synchronization primitives using `Arc, Mutex` primitives. While LLMs can theoretically adapt to these cases, implementing the necessary program analysis to enable specification mining and testing for these cases will require further work.

**Argument Translation API.** We design an argument translation API to translate C arguments to Rust arguments (§4.4.1). The API provides higher order primitives, which allow translating primitive types and multi-dimension vectors (of arbitrary types) while taking care of aliasing (using `Rc<RefCell<RT>>`). We identify a simplicity-expressiveness trade-off in the API design. We can provide powerful and general high-level primitives that can handle a wide range of operations but may be difficult to use (requiring lambda function arguments). Alternatively, we can provide simple and easy-to-use primitives that can handle only a limited set of operations. Here, we identify a simple API design that handles the most common cases and allows LLMs to generate arbitrary Rust code for complex cases. However, future work can explore this tradeoff further, improving the API design by making it easier for LLMs to use and more powerful.

**Efficiency of Translated Program.** Our current approach does not focus on the efficiency of the translated program. Indeed, our translated Zopfli program is up to 3.67× slower than the original C program. We attempt to identify the source of the slow-down by comparing the programs statically and using corresponding flame-graphs but do not observe meaningful insights to translate faster. Going forward, we envision two approaches to improve the efficiency of the translated program. One approach is to translate the program and then use profiling to identify and optimize the slow parts of the program. Another approach is to ensure that the translated program is efficient from the start. We aim to explore these directions going forward.

## 8.2 Threats to validity

**External Validity.** We demonstrate the efficacy of our translation approach by successfully translating Zopfli and UrlParser. This can cause over-fitting in the prompts and impact design decisions in our implementation. First note that Zopfli is over 3000 LoC and contains diverse code elements encompassing linked lists, iterators, function pointers, `void*` elements. Secondly, the Syzygy approach is quite generic and only makes moderate assumptions about the C programs. Nevertheless, as we extend Syzygy, we will evaluate it on a broader suite of programs going forward.

**Test-based equivalence.** We assert the validity of our translation for Zopfli using test cases. However, test-based equivalence is known to suffer from incomplete coverage. Here, we generated over 1000000 tests to achieve high coverage to strengthen our claims.

## 8.3 Conclusion

In this work, we presented Syzygy, our approach that synergistically combines program analysis (specifically dynamic analysis and testing) with LLM sampling-driven-search approach to translate C programs to Rust. Particularly, we use the program dependency graph to decompose translation into individual translation units and perform dual code and test translation to generate valid Rust code. We demonstrate the efficacy of our approach by correctly translating Zopfli, a data compression C program with over 3000 LoC. Our approach outlines a way to *naturally* perform test-driven development and is scalable. In particular, improving the empowering our approach with better dynamic analysis and stronger LLM inference time schedules will allow us to translate more and more complex programs.

# REFERENCES

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer, 934–950.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[3] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698.

[4] Thomas Ball. 1999. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 216–234. https://doi.org/10.1145/318774.318944

[5] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787* (2024).

[6] Pedro Calais and Lissa Franzini. 2023. Test-Driven Development Benefits Beyond Design Quality: Flow State and Developer Experience. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 106–111.

[7] carol-10-cents. 2024. *rust-out-your-c*. https://github.com/carols10cents/rust-out-your-c-talk

[8] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 677–708.

[9] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[10] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[11] Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. 2024. Learning How Hard to Think: Input-Adaptive Allocation of LM Computation. *arXiv preprint arXiv:2410.04707* (2024).

[12] DARPA. 2024. *TRACTOR: Translating All C to Rust*. https://www.darpa.mil/research/programs/translating-all-c-to-rust

[13] 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).

[14] Galois Inc. 2024. *Function Argument Nullability Using an LLM*. https://galois.com/blog/2024/11/function-argument-nullability-using-an-llm/

[15] google. 2024. *google/zopfli*. https://github.com/google/zopfli

[16] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-Level Compositional Code Translation and Validation. *arXiv preprint arXiv:2410.24117* (2024).

[17] Immunant Inc. 2020. *immunant/c2rust*. https://github.com/immunant/c2rust

[18] Immunant Inc. 2024. *Emitting Safer Rust with C2Rust*. https://immunant.com/blog/2023/03/lifting/

[19] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. *Advances in Neural Information Processing Systems* 35 (2022), 13419–13432.

[20] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* (2024).

[21] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. 2024. R2E: Turning any Github Repository into a Programming Agent Environment. In *Forty-first International Conference on Machine Learning*.

[22] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. [n.d.]. Jigsaw: Large Language Models meet Program Synthesis. In *ICSE 2022* (Pittsburgh, Pennsylvania).

[23] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E Gonzalez, Koushik Sen, and Ion Stoica. 2023. Llm-assisted code cleaning for training accurate code generators. *arXiv preprint arXiv:2311.14904* (2023).

[24] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2023. Attention, Compilation, and Solver-based Symbolic Analysis are All You Need. *arXiv preprint arXiv:2306.06755* (2023).

[25] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515* (2024).

[26] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the Evaluation of Neural Code Translation: Taxonomy and Benchmark. In *Automated Software Engineering (ASE)*. IEEE, 1529–1541. https://doi.org/10.1109/ASE56229.2023.00114

[27] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[28] jwerle. 2024. *jwerle/url*. https://github.com/jwerle/

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 75–86.

[30] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. 2024. An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 425–443. https://www.usenix.org/conference/atc24/presentation/li-hongyu

[31] Ruishi Li, Bo Wang, Tianyu Li, Prateek Saxena, and Ashish Kundu. 2024. Translating C To Rust: Lessons from a User Study. *arXiv preprint arXiv:2411.14174* (2024).

[32] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[33] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977* (2024).

[34] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (Jan. 1980), 90–121. https://doi.org/10.1145/357084.357090

[35] Nachiappan Nagappan, E Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. 2008. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering* 13 (2008), 289–302.

[36] Michael L. Nelson. 2005. A Survey of Reverse Engineering and Program Comprehension. *arXiv preprint* (2005).

[37] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*. PMLR, 26106–26128.

[38] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is Self-Repair a Silver Bullet for Code Generation?. In *The Twelfth International Conference on Learning Representations*.

[39] OpenBSD Project. n.d.. *unifdef: remove preprocessor conditionals from code*. https://man.openbsd.org/unifdef https://man.openbsd.org/unifdef.

[40] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *NeurIPS*.

[41] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *ICLR*. OpenReview.net.

[42] Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. 2024. Can Language Models Solve Olympiad Programming? arXiv:2404.10952 [cs.CL]

[43] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2024).

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

[45] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314* (2024).

[46] Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2023. Code translation with Compiler Representations. *ICLR* (2023).

[47] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. 48–62. https://doi.org/10.1109/SP.2013.13

[48] Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, 1741–1788. https://doi.org/10.18653/v1/2023.findings-emnlp.119

[49] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2024. StructCoder: Structure-Aware Transformer for Code Generation. *ACM Trans. Knowl. Discov. Data* 18, 3, Article 70 (Jan. 2024), 20 pages. https://doi.org/10.1145/3636430

[50] Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2024. Planning In Natural Language Improves LLM Search For Code Generation. *arXiv preprint arXiv: 2409.03733* (2024).

[51]  Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs.SE] https://arxiv.org/abs/2407.16741
[52]  Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724* (2024).
[53]  Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. *arXiv preprint* (2024).
[54]  Aidan Z. H. 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).
[55]  John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024).
[56]  Xin Yin, Chao Ni, Tien N. Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code Translation with Corrector via LLMs. *CoRR* abs/2407.07472 (2024). https://doi.org/10.48550/ARXIV.2407.07472 arXiv:2407.07472
[57]  Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. 2022. Parsel : Algorithmic Reasoning with Language Models by Composing Decompositions. https://arxiv.org/abs/2212.10561
[58]  Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2024. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *arXiv preprint arXiv:2412.08035* (2024).
[59]  HanLiang Zhang, C. David, Y. Yu, and M. Wang. 2023. Ownership guided C to Rust translation. *International Conference on Computer Aided Verification* (2023). https://doi.org/10.48550/arXiv.2303.10515
[60]  Kexun Zhang, Shang Zhou, Danqing Wang, William Yang Wang, and Lei Li. 2024. Scaling LLM Inference with Optimized Sample Compute Allocation. *arXiv preprint arXiv:2410.22480* (2024).
[61]  Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*. PMLR, 41832–41846.

## A  PROMPTS

In the following, we present prompts for different modules of our approach (please see next page).

## B  EXAMPLE

In the following, we present an example of translating the `GetCostModelMinCost` function in Zopfli. We show the following artefacts from our pipeline:

(1) **Source C function** `GetCostModelMinCost`: (Figure 13)
(2) **Candidate Rust translation** `get_cost_model_min_cost`: (Figure 14)
(3) **Argument translator**: (Figure 15)
(4) **Equivalence test**: (Figure 16)

```
1  You are given the following C code:
2  ```c
3  {c_decls}
4  {c_stmts}
5  ```
6
7  You have already translated the following C code to Rust code:
8  ```rust
9  {rust_stmts}
10 ```
11
12 Now you will translate the following C code to Rust code:
13 ```c
14 {c_symbol}
15 {dynamic analysis annotations}
16 ```
17
18 You will translate the C code to SAFE Rust and return it as the first code block. Follow
       these guidelines:
19   1. You should only translate the given C code element to Rust code.
20   2. Do NOT modify or rewrite any existing Rust code provided above. Your translation
       should be designed to be appended to the existing Rust code. Write the Rust code
       only for the given element so that it can be appended to the existing Rust code. Do
       NOT rewrite existing rust code since it will cause compile errors.
21   3. Precisely follow the program logic, function signatures, types and returns in the C
       program. Do NOT make unnecessary modifications or simplifications to the SAFE Rust
       code. MINIMIZE the differences between the C and Rust code and maintain
       correspondence between the function signatures as much as possible. Since, we are
       aiming for a direct translation of a large codebase it would be ideal to not make
       large refactoring changes.
22   4. Do NOT leave placeholder, empty, or incomplete code. Do NOT skip long code blocks.
       Ensure that the Rust code is complete and correct.
23   5. Create struct methods only when absolutely necessary. Prefer implementing functions
       over structs. If struct methods are needed, write a new `impl` block enclosing only
       that method. Do NOT copy reimplement or copy struct definition or existing struct
       methods. Do not implement additional traits on structs.
24   6. Translate C arrays to Rust vectors whenever possible. Use Rust arrays only when the
       array size is fixed and known at compile time. For example,
25     - Translate `int arr[10]` to `[i32; 10]` since size is fixed and known at compile
       time.
26     - Translate `int* arr` to `Vec<i32>` since size is not fixed and known at runtime.
27     - Translate `int *(arr)[10]` to `[Vec<i32>; 10]` and `int (*arr)[10]` to `Vec<[i32;
       10]>`.
28     - Do NOT use `Option` inside the vector. `Vec<Option<T>>` can be simulated using a
       0 length `Vec<T>`.
29     - In case the length of the vector along a dimension is always supposed to be 1,
       you can 'squeeze' the dimension and construct a smaller dimensional vector. However,
       in case it is not clear, always use the full dimension.
30   7. Translate function pointers to `Fn` or `FnMut` closures.
31   8. Translate void pointers to either concrete types (if clear from context) or use `
       dyn std::any::Any`. Do NOT use `*mut c_void`. The `Any` can later be resolved using
       `downcast_ref` to downcast to the concrete type when C code casts the void pointer
       to a specific type.
32   9. Use the existing translated Rust structs and functions provided above. Additionally
       , `NodePair` struct has been added to Rust code to simplify the translation of `
       lists` which can be translated to Vec<NodePair>.
33   10. Think step by step and reason about the translated rust code in natural language.
       Add detailed comments sketching out logic of the program in the Rust code. Maintain
       equivalence between the C and the Rust programs, logic, and signature as much as
       possible.
34   11. Add comments both at the start of the rust code block (documenting a function) as
       well as inlined in the code block. Always generate public rust code (annotated with
       `pub`).
35   12. Follow the logic of the C code and do NOT make any optimizations or unnecessary
       changes. Maintain 1-1 correspondence between the C and Rust code blocks as much as
       possible. If the C code maintains a variable for length of the array, maintain the
       same in Rust. If the C code indexes into an array, write the Rust code performing
       indexing similarly. MAINTAIN the nullability and aliasing across C and the Rust code
       as provided above in comments. More generally, perform the SIMPLEST AND THE MOST
       DIRECT translation possible ENSURING CORRECT BEHAVIOUR with SAFE Rust.
36
37 Rust Reminder: You cannot have a mutable and immutable reference to the same data at the
       same time. If you need to mutate data, you should use `Rc` `RefCell` to provide
       interior mutability and shared ownership. Alternatively, you can refactor the code
       to avoid the need for shared mutability.
```

Fig. 10.  Rust Code Generation Prompt

```
1  Following is the C code you are provided:
2  ```c
3  {c_decls}
4  {c_stmts}
5  ```
6
7  Following is the translated Rust code
8  ```rust
9  {final_rust_code}
10 ```
11
12 We will use functional equivalence to check the correctness of the translation of `{
       function_name}` function using FFI. Thus, given a C function `{function_name}` and C
       arguments, we will first convert the C arguments to Rust arguments and then call
       the Rust function with these arguments. This is useful to compare the behavior of
       the Rust function with the expected behavior of the C function. For now, you will
       only translate the arguments.
13
14 To aid with translation, you are provided with the following utilities:
15 [TRANSLATION_API_DOCS_AND_EXAMPLES]
16
17 Now, you will implement the `translate_args` function for `{function_name}`. Reminder:
       `{function_name}` signature:
18 ```c
19 {function_declaration}
20 ```
21 You will enclose your answer in a markdown code block. Follow the signature below. Feel
       free to modify the mutability of the returned values as needed.
22 ```rust
23 {translate_signature}
24 ```
25
26 Follow these instructions
27 1. **Scalar arguments**. You can use the `translate_singleton` and `
       translate_singleton_and_peel` functions to translate scalar arguments. These
       functions return `Option<T>` and `T` respectively. Additionally, you can use `expect
       ` to remove the wrapping `Option` as required.
28 2. **N-dimensional arrays**. You can use the `translate_nd_vec` and `
       translate_nd_vec_and_peel` functions to translate n-dimensional arrays. These
       functions return `Vec<Vec<...Rc<RefCell<T>>...>>` and `Vec<Vec<...T...>>`
       respectively. In cases where a dimension is 1, you can "squeeze" the dimension by
       indexing along that dimension. For example, `translate_2d_vec_and_peel` returns `Vec
       <Vec<T>>` and you can get the first row using `[0]` indexing.
29 3. **Types**. These functions work across both primitive types and structs. Therefore,
       you do NOT need to implement custom translations for arbitrary structions. However,
       you might need to massage the output of the translation functions to match the
       expected Rust types.
30 4. **Allowed Types**. The type T can be primitive types (`i8`, `i16`, `i32`, `i64`, `u8
       `, `u16`, `u32`, `u64`, `f32`, `f64`) or structs (... enumerate ...). You can
       construct singleton instances of these types using the `translate_singleton` and
       vectors of these types using the `translate_nd_vec` functions. Note that `NodePair`
       can be used to directly implement argument translation for `lists` (using `
       translate_1d_vec`). Thus Node *(*lists)[2] can be translated to `Vec<NodePair>` via
       `translate_1d_vec_and_peel(c_lists)`.
31 5. **C Structs**. Note that the structs in the C code are already imported in Rust using
        bindgen + FFI. They have `C` prefix attached to them. For example, `ZopfliOptions`
        in C is imported as `CZopfliOptions` in Rust, `Node` in C is imported as `CNode` in
       Rust. You can use these structs directly in the translation functions. Note that the
        Rust structs are named without the `C` prefix (e.g., `ZopfliOptions`, `Node`).
32 6. **Aliasing**. The translation functions use a symbol table internally to handle
       aliasing. The aliasing is captured by the `Rc<RefCell<T>>` type. Therefore, you do
       NOT need to handle aliasing explicitly. In cases aliasing is not present, you can
       use the `*_peel` functions to get the inner value directly.
33 7. **Return By Value**. Do NOT return `&mut T` from the translation functions since the
       variable will not be live after the function is returned. Instead, return the value
       directly or use `Rc<RefCell<T>>` for shared ownership and interior mutability.
34 8. **Function Pointers**. Function pointers can be resolved to function names. You can
       use a match statement to translate to the appropriate function name.
35 9. **Void Pointers**. Void pointers either can be resolved to a specific type (evident
       from the context) or can be translated to `dyn std::any::Any`. In the latter case,
       you can use the function name to determine the type of the void pointer to construct
        rust object and then upcast it to `dyn Any`.
```

Fig. 11. TranslateArgs Prompt

```
1  Following is the C code you are provided:
2
3  ```c
4  {c_decls}
5  {c_stmts}
6  ```
7
8  Following is the translated Rust code
9
10 ```rust
11 {final_rust_code}
12 ```
13
14 We are going to use functional equivalence to check the correctness of the translation
       of `{c_fn_name}` function.
15
16 Reminder of the C function signature:
17 ```c
18 {c_fn_decl}
19 ```
20 Instructions:
21 1. In the equivalence test function, to translate C arguments to Rust arguments call `
       translate_args_{c_fn_name}` which you can assume already exists in the same file.
22 2. Compare equivalence of BOTH the return values and the values of the arguments after
       the function call as appropriate.
23 3. If needed use `translate_args_{c_fn_name}` to convert C function results to Rust **
       after** the C function call to be able to compare with the Rust function's results.
24 4. If any results (argument and returned values) are of type Box<dyn Any>, you should
       downcast them to the expected type using the `downcast_ref::<T>` method since \'==\'
        operator does not work on trait objects. However, you should ensure that the
       downcast type is properly handled and downcast runs successfully.
25
26 Here is the translate_args function:
27 ```rust
28 {translate_args_fn}
29 ```
30
31 Your equivalence test function `eqtest_{c_fn_name}` will be called with the following
       harness which already handles loading C arguments for the function.
32
33 ```rust
34 {harness}
35 ```
36
37 Assuming the above harness, now you will only implement the equivalence testing function
        which should:
38    - load the C arguments from the given file using the following statement:
39          ```let ({c_pre_args}) = load_pre_args_from_json(file_name);```
40    - use the `translate_args_{c_fn_name}` function to convert C arguments to Rust types.
41    - load the expected values of the C arguments from the given file using the following
        statement:
42          ```let ({c_post_args}) = load_post_args_from_json(file_name);```
43     - use the `translate_args_{{c_fn_name}}` function to convert expected C arguments to
        Rust types. Use the appropriate types for the expected values as per the
       translate_args function signature.
44    - call the `{c_fn_name}` C function with C arguments assuming the FFI bindings are
       declared in the bindings crate. Use this only to get the return values.
45    - call the `{rust_fn_name}` rust function with translated Rust arguments.
46    - compare the results for functional equivalence using assertions with the `assert_eq
       !` macro. You must use assertion messages to show what is being compared.
47    - return true if all assertions pass. Otherwise, let the code panic naturally.
48
49 Only return the equivalence testing function and nothing else. You will enclose your
       answer in a markdown code block.
50 ```rust
51 {eqtest_signature}
52 ```
53
54 Ensure that you correctly test the function and check the outputs match the C outputs (
       if the function returns a value). Additionally, if the function modifies the inputs,
        perform equivalence checks on the modified (mutable) inputs appropriately. You DO
       NOT need to do equivalence on the immutable inputs.
```

Fig. 12. Rust Equivalence Test Generation Prompt

```c
/*
Finds the minimum possible cost this cost model can return for valid length and
distance symbols.
*/
static double GetCostModelMinCost(CostModelFun *costmodel, void *costcontext)
{
  double mincost;
  int bestlength = 0; /* length that has lowest cost in the cost model */
  int bestdist = 0;   /* distance that has lowest cost in the cost model */
  int i;
  /*
  Table of distances that have a different distance symbol in the deflate
  specification. Each value is the first distance that has a new symbol. Only
  different symbols affect the cost model so only these need to be checked.
  See RFC 1951 section 3.2.5. Compressed blocks (length and distance codes).
  */
  static const int dsymbols[30] = {
      1, 2, 3, 4, 5, 7, 9, 13, 17, 25, 33, 49, 65, 97, 129, 193, 257, 385, 513,
      769, 1025, 1537, 2049, 3073, 4097, 6145, 8193, 12289, 16385, 24577};

  mincost = ZOPFLI_LARGE_FLOAT;
  for (i = 3; i < 259; i++)
  {
    double c = costmodel(i, 1, costcontext);
    if (c < mincost)
    {
      bestlength = i;
      mincost = c;
    }
  }

  mincost = ZOPFLI_LARGE_FLOAT;
  for (i = 0; i < 30; i++)
  {
    double c = costmodel(3, dsymbols[i], costcontext);
    if (c < mincost)
    {
      bestdist = dsymbols[i];
      mincost = c;
    }
  }

  return costmodel(bestlength, bestdist, costcontext);
}
```

Fig. 13. The `GetCostModelMinCost` function from the Zopfli codebase. The function's complexity stems from using indirect function calls through the `costmodel` function pointer coupled with a `void*` context parameter. Particularly hard is to interpret (purely statically) that the function pointer can point to several cost functions.

```rust
/// Finds the minimum possible cost this cost model can return for valid length and
/// distance symbols.
///
/// # Arguments
/// * `costmodel` – A reference to the cost model function or closure.
/// * `costcontext` – A reference to any context-specific data required by the cost model.
///
/// # Returns
/// Returns the minimum possible cost as an `f64`.
///
/// # Notes
/// – The function uses `ZOPFLI_LARGE_FLOAT` as an initial high cost value.
/// – The best length and distance are found separately and then combined to compute the final cost.
pub fn get_cost_model_min_cost(
    costmodel: &dyn Fn(u32, u32, &dyn std::any::Any) -> f64,
    costcontext: &dyn std::any::Any,
) -> f64 {
    // Initialize variables for minimum cost and best length/distance.
    let mut min_cost: f64;
    let mut best_length: u32 = 0; // Length that has the lowest cost in the cost model.
    let mut best_dist: u32 = 0;   // Distance that has the lowest cost in the cost model.

    /*
    Table of distances that have a different distance symbol in the DEFLATE
    specification. Each value is the first distance that has a new symbol. Only
    different symbols affect the cost model, so only these need to be checked.
    See RFC 1951 section 3.2.5: Compressed blocks (length and distance codes).
    */
    let dsymbols: [u32; 30] = [
        1, 2, 3, 4, 5, 7, 9, 13, 17, 25,
        33, 49, 65, 97, 129, 193, 257, 385,
        513, 769, 1025, 1537, 2049, 3073,
        4097, 6145, 8193, 12289, 16385, 24577,
    ];

    // Find the length with the lowest cost.
    min_cost = ZOPFLI_LARGE_FLOAT;
    for i in 3..259 {
        let c = costmodel(i as u32, 1, costcontext);
        if c < min_cost {
            best_length = i as u32;
            min_cost = c;
        }
    }

    // Find the distance with the lowest cost.
    min_cost = ZOPFLI_LARGE_FLOAT;
    for &dist in dsymbols.iter() {
        let c = costmodel(3, dist, costcontext);
        if c < min_cost {
            best_dist = dist;
            min_cost = c;
        }
    }

    // Return the minimum possible cost using the best length and distance found.
    costmodel(best_length, best_dist, costcontext)
}
```

Fig. 14. A candidate Rust translation generated for the C function `GetCostModelMinCost`

```rust
pub unsafe fn translate_args_GetCostModelMinCost(
    costmodel_ptr: Option<
        unsafe extern "C" fn(
            litlen: ::std::os::raw::c_uint,
            dist: ::std::os::raw::c_uint,
            context: *mut ::std::os::raw::c_void,
        ) -> f64,
    >,
    costcontext: *mut ::std::os::raw::c_void,
) -> (
    Box<dyn Fn(u32, u32, &dyn std::any::Any) -> f64>,
    Box<dyn std::any::Any>,
) {
    rs_symtab_clear();

    // Get the function name from the function pointer
    let fn_ptr = costmodel_ptr.expect("Function pointer is null");
    let fn_name = get_fn_name_from_fn_ptr(fn_ptr as *mut c_void)
        .expect("Failed to get function name from function pointer");

    if fn_name == "GetCostFixed" {
        // For GetCostFixed, no context is needed
        // Create a closure that matches the expected signature
        let costmodel = Box::new(|litlen: u32, dist: u32, _context: &dyn Any| -> f64 {
            GetCostFixed(litlen, dist, std::ptr::null_mut())
        }) as Box<dyn Fn(u32, u32, &dyn std::any::Any) -> f64>;

        // The context is not used, so we can use an empty tuple
        let costcontext = Box::new(()) as Box<dyn Any>;

        (costmodel, costcontext)
    } else if fn_name == "GetCostStat" {
        // For GetCostStat, the context is a SymbolStats structure
        // Translate the context from C to Rust
        let rust_context = translate_singleton_and_peel::<SymbolStats>(costcontext)
            .expect("Failed to translate SymbolStats context");

        // Create a closure that captures the context and matches the expected signature
        let costmodel = Box::new(move |litlen: u32, dist: u32, context: &dyn Any| -> f64 {
            // Downcast the context to SymbolStats
            let stats = context
                .downcast_ref::<SymbolStats>()
                .expect("Failed to downcast context to SymbolStats");
            zopfli_get_cost_stat(litlen as i32, dist as i32, stats)
        }) as Box<dyn Fn(u32, u32, &dyn std::any::Any) -> f64>;

        // Box the context so it can be used as a dynamic Any type
        let costcontext = Box::new(rust_context) as Box<dyn Any>;

        (costmodel, costcontext)
    } else {
        panic!("Unknown function name: {}", fn_name);
    }
}
```

Fig. 15. The LLM generated translateArgs function for the `GetCostModelMinCost`, handling the translation of C function pointers to Rust closures, supporting multiple variants the pointer may point to: `GetCostFixed` and `GetCostStat`. The translateArgs is aligned with the candidate translation in Fig. 14
.

```rust
pub unsafe fn eqtest_GetCostModelMinCost(filename: &str) -> bool {
    let (c_pre_arg0, c_pre_arg1) = load_pre_args_from_json(filename);

    // Translate the C arguments to Rust arguments using the provided translate function
    let (rust_costmodel, rust_costcontext) =
        translate_args_GetCostModelMinCost(c_pre_arg0, c_pre_arg1);

    // Call the C function with the original C arguments
    let c_result = bindings::GetCostModelMinCost(c_pre_arg0, c_pre_arg1);

    // Call the Rust function with the translated Rust arguments
    let rust_result = get_cost_model_min_cost(&*rust_costmodel, &*rust_costcontext);

    // Compare the return values of C and Rust functions, allowing for possible floating-point differences
    let epsilon = 1e-6;
    assert!(
            (c_result - rust_result).abs() < epsilon,
        "C and Rust functions returned different values: C={} vs Rust={}",
        c_result,
        rust_result
    );

    // Since the arguments are not modified by the function, no need to compare them after the call

    // All assertions passed
    true
}
```

Fig. 16. The LLM generated equivalence test for the C function `GetCostModelMinCost` (Fig. 13) and candidate translation `get_cost_model_min_cost` (Fig. 14). Note the use of the previously generated translateArgs (Fig. 15). Also, the model-generated test uses assertions with tolerance for floating-point precision.