RustAssure: Differential Symbolic Testing for LLM-Transpiled C-to-Rust Code

1st Yubo Bai
Department of Computer Science
University of California, Davis
Davis, USA
gabbai@ucdavis.edu

2nd Tapti Palit
Department of Computer Science
University of California, Davis
Davis, USA
tpalit@ucdavis.edu

***Abstract*—Rust is a memory-safe programming language that significantly improves software security. Existing codebases written in unsafe memory languages, such as C, must first be transpiled to Rust to take advantage of Rust's improved safety guarantees. RustAssure presents a system that uses Large Language Models (LLMs) to automatically transpile existing C codebases to Rust. RustAssure uses prompt engineering techniques to maximize the chances of the LLM generating *idiomatic* and *safe* Rust code. Moreover, because LLMs often generate code with subtle bugs that can be missed under traditional unit or fuzz testing, RustAssure performs differential symbolic testing to establish the semantic similarity between the original C and LLM-transpiled Rust code. We evaluated RustAssure with five real-world applications and libraries, and showed that our system is able to generate compilable Rust functions for 89.8% of all C functions, of which 72% produced equivalent symbolic return values for both the C and Rust functions.**

I. Introduction

Rust is a memory-safe programming language that is increasingly gaining popularity. Automatically transpiling codebases written in memory-unsafe languages such as C, to Rust, can significantly improve software security. Previous syntax-directed and program analysis-driven methods [1], [2], [3] have achieved limited success, correctly transpiling only a fraction of the C code to safe Rust. Moreover, the Rust code generated by these approaches is difficult to read and often does not follow idiomatic Rust conventions [4]. More recently, Large language models (LLMs) have shown significant promise in code completion [5], code generation [6], [7], [8], and code transpilation [4]. However, LLMs are not guaranteed to always generate correct code [9]. Therefore, it is essential to verify the functional correctness of LLM-generated code.

Existing approaches for testing LLM-transpiled Rust code suffer from various limitations. Approaches based on unit testing inherently suffer from code coverage issues, especially for large complex applications, leading to significant portions of the code remaining untested. Approaches based on differential fuzzing [4] compare the test case outputs for the original and transpiled code. However, the exact return values of the C and Rust functions can differ due to runtime differences. For example, the values of pointers returned by the functions would differ across different language runtime environments. Therefore, a mismatch in the concrete runtime values do not necessarily indicate an error.

To mitigate these challenges, we present RustAssure—a system that uses Large Language Models (LLMs) to transpile C code into Rust and establishes the *semantic similarity* between the original and transpiled code using *differential symbolic testing*. Unlike dynamic testing approaches such as unit or fuzz testing, where the transpiled Rust function is executed with concrete inputs, RustAssure *symbolically* executes both the original C and LLM-transpiled Rust code to detect functional divergence between the original C and Rust code. Unlike dynamic testing which is *input-specific*, symbolic execution can explore all paths, thus proving that all inputs produce equivalent outputs and guaranteeing correctness, while differential testing can only show that the behavior is the same for the executed input. Furthermore, by detecting the divergence at the symbolic level, RustAssure avoids false positives stemming from differences in the runtime environment, such as memory layout and heap addresses.

RustAssure first transpiles the C code to Rust using Large Language Models (LLMs). Multiple factors must be considered when using LLMs to transpile C code to Rust. First, the LLM-based transpiler must ensure that the C code is transpiled to *safe* Rust to the greatest extent possible. To provide programmers with more flexibility, Rust provides an "escape-hatch" that relaxes certain memory safety rules if the code is enclosed in an `unsafe` block. Unsafe Rust code introduces potential memory corruption vulnerabilities in the program, therefore, the amount of unsafe Rust code generated by the LLM must be minimized. RustAssure uses various prompt-engineering techniques to ensure that the code generated by the LLM compiles successfully, and also minimizes the use of unsafe Rust.

After the LLM generates compilable Rust code, RustAssure performs differential symbolic testing on the original C and LLM-transpiled Rust code. RustAssure contains a *Semantic Similarity Checker* that symbolically executes individual C functions and their LLM-transpiled Rust counterparts, deriving symbolic values for all implicit and explicit return variables of the function. RustAssure then compares the symbolic value for each return value of the function and reports a *Semantic Similarity Score (S^3 score)* for the Rust function. Moreover, differential symbolic testing can not only identify which return values diverge, but also *how* they diverge, making it easier for

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00051

the programmer to pinpoint the bug.

RustAssure's symbolic execution is *composable.* RustAssure can analyze each Rust function individually and determine if it is semantically equivalent to the original C function. If the Rust function being analyzed invokes another function, RustAssure abstracts away the implementation of that function and only ensures that the called function returns symbolic values. This allows the caller function to be symbolically tested even if the called function does not compile or link successfully with the rest of the LLM-transpiled Rust code. As a result, the programmer can test each LLM-transpiled Rust function for semantic equivalence and improve their confidence in the LLM-produced code, even if a few LLM-generated Rust functions do not successfully compile. In fact, from our experience even with state-of-the-art LLM models such as GPT-4o [10], only 89.8% of all transpiled C functions successfully compile without user intervention. RustAssure is particularly useful in such scenarios.

Establishing semantic equivalence using symbolic return values across different languages is not trivial. Often, Rust code uses types that are missing in C completely. For example, safe Rust code does not allow explicit `NULL` values but requires *wrapping* values that can potentially be `NULL` in an `Option` *enum* type. Accurate extraction and comparison of symbolic values across different languages requires *aligning* these differing C and Rust types correctly. RustAssure identifies three such commonly occurring patterns and maps the Rust type correctly to the C type, thus facilitating the symbolic value comparison of the return values.

We used RustAssure to transpile five C real-world codebases to Rust, and used the *Semantic Similarity Checker* to compute the *Semantic Similarity Score (*S^3 *score)* for the transpilations. These libraries had a total of 176 functions, of which 158, 146, 124, and 85 functions were compiled successfully by the GPT-4o, GPT-4o-mini, GPT-3.5-turbo [10], [11] and Claude-3.5-Sonnet [12] models, respectively. Across all of these LLM models, RustAssure's S^3 metric allowed us to find 12 complex semantic bugs, along with 13 simple bugs. Because these 12 and 13 bugs are present in more than one LLM model, the total number of buggy functions detected are 20 and 17, for semantic and simple bugs, respectively. Moreover, we computed the average precision of the S^3 score, across all LLM models to be 85.7% and 88.2%, for the *libcsv* and *u8c* codebases, respectively, while the average recall was 100% for both codebases. An end-to-end comparison shows that RustAssure can generate code with 72% fewer raw pointers than the state-of-the-art syntax-directed C-to-Rust transpilation toolchain, CROWN [13], and find 11 semantic bugs that the testing framework, FLUORINE [4], is unable to find.

In summary, we make the following contributions—

- *RustAssure*, an LLM-based pipeline for transpiling C code to safe Rust. The pipeline uses various prompt engineering techniques to assist the LLM in generating safe Rust code.
- *Semantic Similarity Checker*, a symbolic execution-based system to identify the semantic similarity between the original C and LLM-transpiled Rust code.
- Three patterns for reconciling the type differences between C and Rust that make it possible to compare the symbolic return values in C and Rust code.
- Evaluation of RustAssure using five real-world libraries that demonstrate RustAssure's bug-finding capabilities, and comparisons against the CROWN [13] and FLUORINE [4] C-to-Rust toolchains.

Our toolchain can be found at https://github.com/davsec-lab/rustassure.

II. Motivating Example

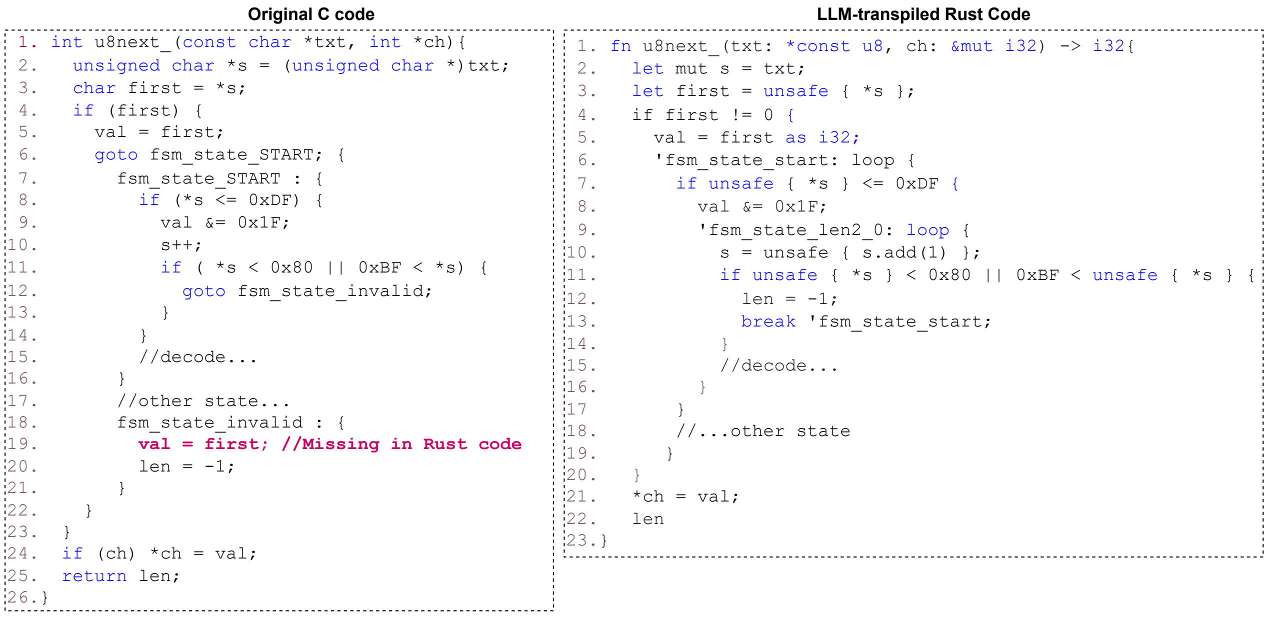
Code generated by Large Language Models (LLMs) often contain subtle bugs. Consider the simplified code snippet for the function `u8next_` from the *u8c* library, shown in Figure 1. The original C function takes as arguments an input UTF-8-encoded string, `txt`, and an output argument, `ch`. It returns the number of bytes encoding the first Unicode codepoint of the string `txt`, and stores the converted Unicode codepoint in `ch`. If the encoding is not valid, it returns `-1` and stores the *first byte of the input string* in `ch`.

To perform the conversion, the function maintains a state machine, as shown in lines 6-23 in the original C code. The LLM-transpiled Rust code contains a subtle bug in this state machine code. When the input is not a valid UTF string and contains an invalid byte in any of the *non-first* byte offsets, the functionalities of the original C and the LLM-transpiled Rust functions diverge. For example, consider the two character input sequence `{0xC2, 0x41}`. The first byte, `0xC2` is valid, but the second byte, `0x41`, is invalid. When executed with this byte sequence, the C program will return the original first byte, which is `0xC2`, but the transpiled Rust program will return an intermediate decoding state, specifically, the bit-masked value `0x02`. This occurs because the generated Rust code is missing the operation that resets the variable `val` to the first character of the input string `txt` (line 19, in the C code). The variable `val` is finally copied to the `ch` argument, resulting in the functional divergence. Thus, the C and Rust API semantically diverge, and if the users of the Rust API are not aware of this change they may misinterpret error conditions or incorrectly assume behavior consistent with the C API.

Detecting this bug requires knowledge of Unicode/UTF-8 semantics, which traditional fuzzers lack, limiting their effectiveness. Moreover, the unit test cases provided for the *u8c* library do not verify the returned byte in the `ch` output argument when an invalid UTF string is provided, and only focuses on the explicit return value for checking error conditions. Because the RustAssure toolchain relies on symbolic execution, it can correctly identify this semantic divergence.

III. Design

Figure 2 shows RustAssure's two major components—the LLM Invoker and the Symbolic Analyzer. The LLM Invoker is responsible for invoking the LLM model for the transpilation



```c
int u8next_(const char *txt, int *ch){
  unsigned char *s = (unsigned char *)txt;
  char first = *s;
  if (first) {
    val = first;
    goto fsm_state_START; {
      fsm_state_START : {
        if (*s <= 0xDF) {
          val &= 0x1F;
          s++;
          if ( *s < 0x80 || 0xBF < *s) {
            goto fsm_state_invalid;
          }
        }
        //decode...
      }
      //other state...
      fsm_state_invalid : {
        val = first; //Missing in Rust code
        len = -1;
      }
    }
  }
  if (ch) *ch = val;
  return len;
}
```

```rust
fn u8next_(txt: *const u8, ch: &mut i32) -> i32{
  let mut s = txt;
  let first = unsafe { *s };
  if first != 0 {
    val = first as i32;
    'fsm_state_start: loop {
      if unsafe { *s } <= 0xDF {
        val &= 0x1F;
        'fsm_state_len2_0: loop {
          s = unsafe { s.add(1) };
          if unsafe { *s } < 0x80 || 0xBF < unsafe { *s } {
            len = -1;
            break 'fsm_state_start;
          }
          //decode...
        }
      }
      //...other state
    }
  }
  *ch = val;
  len
}
```

Fig. 1. Bug detected by RustAssure. When executed with an invalid UTF string, the LLM-generated Rust code returns an incorrect intermediate state instead of a pointer to the start of the input string, which is the expected functionality.

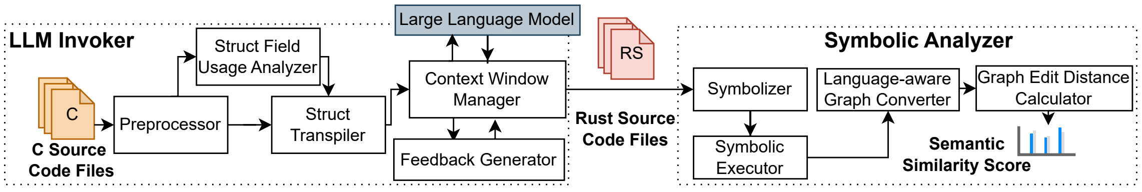


Fig. 2. RustAssure transpiler toolchain.

task, followed by the Symbolic Analyzer performing differential symbolic testing to identify functional divergences.

A. LLM Invoker

RustAssure's LLM Invoker is responsible for taking a C codebase, preprocessing it, and invoking the configured LLM model for transpilation.

Header File Preprocessor. We observed that naively sending the code in the C source code files (`.c` files) to the Large Language Model for transpilation is not adequate. This code often lacks much of the contextual information that is essential for the correct transpilation of each function in the source code file, such as, `struct` definitions, function declarations, preprocessor macros, and `typedef` aliases, which are defined in `.h` header files.

However, when the C preprocessor processes the `.c` file, it naively adds all contents of all header files included by the `.c` file, even if the content is not actually used by the C code. Therefore, including all output produced by the C preprocessor in the LLM request is unnecessary. Moreover, LLM models have a fixed context window [14] that limits the amount of input it can process at a time. Therefore, RustAssure's LLM Invoker includes only the *essential* macro expansions, `typedef` instructions, and `struct` and `function` definitions in the LLM request. This contextual information helps the LLM produce accurate transpilations.

Struct Transpiler. Multiple functions, whose transpilation spreads out over multiple LLM invocations can use the same `struct` type. We observed that due to the inherent non-determinism and randomness [15] of LLM models these different invocations can sometimes generate slightly different Rust `struct` definitions for the same C types. For example, the names of certain transpiled `struct` fields can differ between different LLM invocations. These differing definitions cause linkage issues when the transpiled Rust functions are linked together to build the final executable. To avoid such linkage issues, the LLM Invoker first extracts all C `struct` definitions and invokes the LLM to *pre-translate* them. These Rust transpilations of `struct` types are stored in a *Struct Transpilation Cache*. Then, for each function that uses these `struct` types, the pre-generated `struct` definition is provided along with the C function as the prompt, and the LLM is instructed to use the pre-generated definition. This ensures consistent transpilation of `struct` types across different functions.

Struct Field Usage Analyzer. In many cases, the correct Rust type for a variable depends on the *usage* of the corresponding C variable. For example, in C/C++, `struct` fields of `char*` type can be used to store both null-terminated strings, or arrays of characters. The usage of the `char*` field must be analyzed to determine whether it is used as a string or an array. In fact, without such analysis, from our experience, LLM models transpile such `struct` fields into Rust *raw pointers*, such as `*mut T`, and enclose any code using these pointers fields with `unsafe` blocks. To limit the amount of unsafe Rust code, the LLM Invoker extracts all uses of such `char*` fields and adds them to the LLM request, when pre-translating the `struct` types, and instructs the LLM to consider these field usages for the transpilation task.

Context Window Manager. While LLM context window sizes are ever increasing, many LLM models, such as GPT-3.5-turbo [10] and Mistral 7B [16] have smaller context windows of 16K tokens. When using these models, a single large function can exceed the context window length. To support these models, the LLM Invoker automatically *chunks* a function into multiple LLM requests if it does not fit within the LLM's context window. The LLM's response for each chunk is then collected and assembled into the final Rust function.

Feedback Generator. Like previous approaches [17], when the LLM produces a transpilation that does not compile, RustAssure re-attempts the transpilation, by sending the compiler error message along with the original C function to the LLM, requesting a corrected translation. This approach leverages the precise and diagnostically rich error messages emitted by the Rust compiler to provide feedback to the LLM, thus increasing the chances of it fixing the offending code. In addition to compilation errors, if the LLM-transpiled Rust code contains any `unsafe` block, or invokes unsafe functions such as the functions from the `libc` crate, the LLM Invoker requests a re-translation up to a configurable number of times. Thus, each C function is transpiled using the LLM, maximizing the possibility of safe Rust code.

B. Symbolic Analyzer

After the LLM Invoker produces compilable Rust code, RustAssure symbolically executes them to derive the symbolic return values for each function. We note that C and Rust functions will not be symbolically equivalent if the C code has memory corruption bugs that are absent in Rust—a limitation that is present even when using concrete test cases. Such cases can result in false positives.

Symbolizer. The Symbolizer instruments the C and Rust counterparts of each function with a *prologue* and an *epilogue.* The function prologue initializes the function arguments with symbolic values, enabling the Symbolic Executor to symbolically execute the function under test. If the function takes pointer arguments or arguments of `struct` type, the Symbolizer ensures that symbolic memory regions are allocated for each pointer and `struct` field. Pointer type arguments and fields can be nested to arbitrary levels. For example, an argument of `struct` type can contain a pointer field pointing to another `struct` object. The Symbolizer symbolizes nested pointers and nested `struct` fields up to a configurable nesting limit. For our evaluation, we use a maximum limit of 10.

The Symbolizer also instruments the function execution completion with a *function epilogue* to log the symbolic values of the explicit return value and output parameters. For complex types such as `struct` and array types, the Symbolizer logs the symbolic values for each field or element. If these fields or elements are pointers, the Symbolizer recursively traverses the symbolic object pointed to by its pointer, and logs its symbolic values. The Symbolizer tracks which fields of `struct` type output parameters were modified and logs the symbolic values of only those fields. This selective logging ensures that only meaningful symbolic state changes are reported.

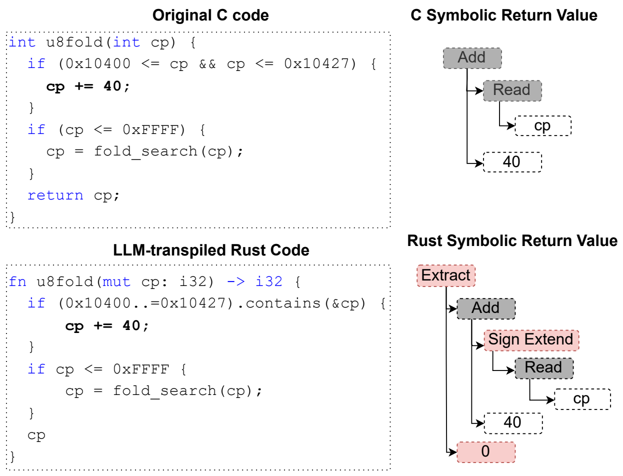
The Symbolizer also instruments any external function calls to return symbolic values, ensuring that symbolic execution can proceed without requiring concrete results from unknown or unanalyzed functions. By replacing the outputs of external function with symbolic expressions, our analysis remains fully composable, and each function can be symbolically executed independently.

Symbolic Executor. After the Symbolizer modifies the C and Rust functions, the Symbolic Executor performs the core symbolic execution process using the KLEE [18] symbolic execution engine. When the symbolic execution completes, the *function epilogue* inserted by the Symbolizer is executed, and the symbolic values of the function return values are printed. To facilitate the symbolic execution of both C and Rust code using KLEE, we convert both the original C and the transpiled Rust code to an Intermediate Representation (IR) and symbolically execute this IR code. However, symbolically executing the Rust-produced IR code adds Rust language-specific artifacts to the symbolic return values. We discuss how we handle these language-specific artifacts in the symbolic values in §III-C.

C. Semantic Similarity Checker

The *Semantic Similarity Checker* compares the semantic equivalence between the original C code and the LLM-transpiled Rust code by comparing the symbolic return values of the corresponding C and Rust functions. The symbolic values are represented by a Backus-Naur Form grammar, called *KQuery* [19]. RustAssure processes these KQuery symbolic values as discussed below.

Language-aware Graph Converter. The *Language-aware Graph Converter* component converts the symbolic values to a graphical representation. In some cases, language differences between C and Rust can cause differences in symbolic representations between C and Rust, even when the high-level


Original C code

```c
int u8fold(int cp) {
  if (0x10400 <= cp && cp <= 0x10427) {
    cp += 40;
  }
  if (cp <= 0xFFFF) {
    cp = fold_search(cp);
  }
  return cp;
}
```

LLM-transpiled Rust Code

```rust
fn u8fold(mut cp: i32) -> i32 {
  if (0x10400..=0x10427).contains(&cp) {
      cp += 40;
  }
  if cp <= 0xFFFF {
      cp = fold_search(cp);
  }
  cp
}
```

Fig. 3. C and Rust symbolic return values for the `u8fold` function. The red boxes are language-specific artifacts added by the Rust compiler.

semantics of the code remain the same. The graph representations allows us to easily apply normalization techniques on the symbolic values due to such language-level differences.

Consider the C and transpiled Rust code, from the *u8c* library, shown in Figure 3. The function `u8fold` maps a Unicode codepoint to its lowercase form. Although both the C and Rust code perform the same `cp += 40` operation in the first *if* block, the symbolic output graphs differ due to Rust's explicit type extension operations needed to support integer overflow checks necessary for memory safety. These checks are completely missing in C. These additional nodes in Rust's symbolic execution graph cause a mismatch even when the underlying operation is identical. To address these differences, the Language-aware Graph Converter applies graph normalization techniques to remove redundant type extension operations that do not affect program semantics, by removing their corresponding nodes from the symbol graph.

Graph Edit-distance Calculator. RustAssure uses the graph-edit distance between the corresponding graphical representations of the symbolic return values of the C and Rust functions as the Semantic Similarity Score *(S^3 score)* between return values in the original C code and the LLM-transpiled Rust code. An edit distance of zero means that the Rust function returns the same symbolic return value as the original C function. A higher edit distance implies a higher degree of semantic divergence. RustAssure uses a standard graph edit-distance algorithm [20] to compute the edit distance between the return values. We used a graph edit-distance-based comparison instead of SMT solving to simplify our implementation. In the future, we plan to explore using SMT solvers to establish strict equivalence.

IV. Handling Language-level Differences

Fundamental differences between C and Rust often results in a mismatch between types and object memory layout when variables are transpiled from C to Rust. To limit false positives, RustAssure aligns these mismatches as described below.

A. *Aligning Memory Layout.*

We observe that the memory layouts of complex types such as `structs` generated by C compilers, such as Clang [21], differ from the memory layouts generated by the Rust compiler, `rustc`. By default, the Rust compiler aligns fields differently and adds additional padding in between different fields. Moreover, the Rust compiler often also reorders the struct fields for memory optimization. This differing memory layout makes it challenging to compare the symbolic value of the original C objects and the transpiled Rust objects.

Fortunately, Rust allows the programmer to specify alternative data layout strategies. In particular, it provides the attribute `repr(C)` to specify a C-like memory layout. Similarly, the `repr(packed)` attribute instructs the compiler not to insert any additional padding between the `struct` fields. Therefore, RustAssure applies the `#[repr(C, packed)]` attribute on every Rust struct type. This ensures that the compiler lays the struct fields out in memory exactly the way a C compiler would, keeping struct fields in order and using C-style alignment and padding. This simplifies the recovery of the symbolic return values from these fields.

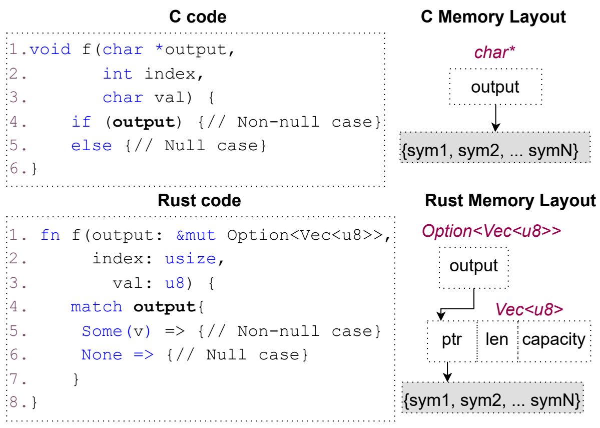
B. *The* Option *type.*

Unlike C, Rust does not support `NULL` values and instead uses the `Option<T>` enum to represent an *optional* value. The `Option<T>` enum type can either contain a value of type `T (Some(T))` or be empty `(None)`. Thus, C pointer variables that can potentially be `NULL`, are wrapped in the `Option<T>` enum type in Rust. Therefore, the apparent type of the C and transpiled Rust variable will not be the same. This difference means that RustAssure cannot simply use the return value types to retrieve and compare the symbolic values of the C and Rust variables. Additional processing is required to align the C and Rust types in such cases.

However, in spite of the difference in the types of the variables, when the Rust compiler lowers the `Option` enum type variable to the machine code, it treats the enum variable simply as *syntactic sugar* around the inner object. The memory layout of the original C nullable pointer of type `T` and Rust's `Option` wrapper around the type `T` are the same, assuming the requirements outlined in §IV-A are followed. The only special operations the Rust compiler inserts are `NULL` checks that panic on failure. Therefore, when RustAssure's Semantic Similarity Checker encounters a situation where a return value or output parameter in Rust is of `Option` type, it simply casts it to the underlying type to access its symbolic representation.

C. *String and Slice types.*

The C language represents strings as null-terminated arrays of `char` type. The corresponding type in Rust is *String*. A



```c
void f(char *output,
       int index,
       char val) {
    if (output) {// Non-null case}
    else {// Null case}
}
```

```rust
fn f(output: &mut Option<Vec<u8>>,
     index: usize,
       val: u8) {
    match output{
     Some(v) => {// Non-null case}
     None => {// Null case}
    }
}
```

Fig. 4. Differing memory layout between C and Rust. The argument `input` is a `char*` array for which the transpiled Rust code uses `Vec` type.

related Rust type is the slice type *&str* that represents an immutable reference to a sequence of characters. Similar to string slices, Rust also allows slices of arrays. Both `String` and slice type variables are internally represented by `struct` types by the Rust compiler. The first field of this `struct` is a *data* pointer pointing to the array of `char` type, and the second field represents the length. Therefore, for return values of these types, RustAssure's Semantic Similarity Checker accesses the first field of the `struct` type and casts it to `char*` before comparing its symbolic values.

Internally, the Rust compiler initializes the slice types, `&str` and `&[T]` as arrays of initial length zero, indicating that their actual length is determined at runtime. When symbolizing such slice types, RustAssure's Symbolizer initializes these slices to symbolic buffers of size `100` and explicitly initializes the `len` field of the struct to `100` to safely access its contents.

D. The vector type.

C arrays are transpiled to Rust's `Vec` type objects. Unlike C arrays, Rust guarantees that any out-of-bounds accesses in a `Vec` type will cause a panic at runtime. Similar to the `String` type, the `Vec<T>` type is represented by a `struct` type that contains three fields, a *data* pointer to the array of type `T`, a *len* field to represent the length of the array indicating the actual number of elements contained, and a *capacity* field to represent the capacity of the array. The semantic checker directly accesses this field and compares the symbolic values of each array element in the Rust code with its C counterpart.

When the `String`, `str&`, `Vec` are wrapped in an `Option`, we extract and cast the inner value as explained in §IV-B, before comparing the symbolic values. Figure 4 shows an example of a nullable array of characters, `output`, being transpiled into a Rust `Vec` object, wrapped in an `Option` enum object. The figure also shows the C and Rust code used to access the respective objects, along with the objects' memory layouts. As shown, the Rust variable `output` of type `Option<Vec<u8>>` is a pointer to the `Vec` object that contains three fields, of which the first `ptr` field points to the actual data buffer. The actual memory layout of the data buffer referenced by the `ptr` field of Rust's `Vec` object is the same as the layout of the C `output` variable. Therefore, for the Rust code, RustAssure's Semantic Similarity Checker just dereferences the `output` variable to access the `Vec` object and then accesses the symbolic contents pointed to by the `ptr` field pointer. These symbolic contents are compared against those of the `output` variable in the original C program.

V. Implementation

RustAssure uses Clang 14 and Rustc v1.64 for compiling the C and transpiled Rust code. It interfaces with the LLM models using Python modules. Both Clang and Rustc uses the LLVM compiler to compile C and Rust code to machine code, respectively. RustAssure leverages this common compiler toolchain to compile both the C and the Rust code to LLVM Intermediate Representation (IR), facilitating the subsequent symbolic execution. RustAssure uses a sequence of *Clang* tools, based on ASTMatchers [22], to statically analyze and perform various preprocessing tasks on the C source code files, such as expanding all header files, removing unused `typedef` definitions, variable and `struct` definitions.

RustAssure uses the KLEE v3.1 [18] symbolic execution engine to symbolically execute the LLVM IR code for both the C and Rust code. While KLEE supports C standard library functions, it lacks support for the Rust standard library. Therefore, we first compiled Rust's standard library to obtain its LLVM IR bitcode representation, and then linked it with the bitcode of each function being symbolically executed. Finally, we developed an Antlr [23] parser for KLEE's *Kquery* grammar and parsed the symbolic return values to produce symbolic graphs represented in the *GraphViz* DOT format [24]. We used the *networkx* [25] library's graph edit distance algorithm [26] function to compute the edit distance between the symbolic return values and obtain their S^3 score.

VI. Evaluation

We evaluated RustAssure with five C codebases—*libcsv*, *urlparser*, *optipng*, *libbmp*, and *u8c*, with 405, 435, 3039, 287, and 334 lines of code, respectively. Our dataset consists of applications commonly used in previous works [1], [27], [3], [28], [2], [29]. We selected the following LLM models—OpenAI's GPT-4o, GPT-4o-mini, GPT-3.5-turbo, and Anthropic's Claude-3.5-Sonnet, for the evaluation.

A. Compilation Results

Table I presents the compilation results for the five codebases for the four LLM models. GPT-4o outperformed all other LLM models across all codebases, successfully transpiling 89.8% of all C functions into compilable Rust functions. GPT-4o-mini produced compilable transpilations for 83.0% of C functions, while the GPT-3.5-turbo and Claude-3.5-Sonnet models produced compilable Rust translations for 70.5% and 48.3% of all C functions.

TABLE I
COMPILATION AND SEMANTIC EQUIVALENCE STATISTICS. EQ. OUTPUT INDICATES RETURN VALUES WITH S^3 SCORE = 0.

Codebase	LLM Model	Functions			Code	Return values and output parameters			
		C functions	Rust compiled	Perc. compiled	Perc. unsafe	Output values	Eq. Output	% of Eq. Output	Bug
libcsv	GPT-4o	29	27	93.1%	3%	73	62	84.9%	5
	GPT-4o-mini	29	23	79.3%	16%	66	55	83.3%	1
	GPT-3.5-turbo	29	25	86.2%	15%	59	35	59.3%	2
	Claude-3.5-Sonnet	29	15	51.7%	79%	48	35	72.9%	2
urlparser	GPT-4o	26	23	88.5%	10%	32	18	56.3%	1
	GPT-4o-mini	26	24	92.3%	27%	35	20	57.1%	0
	GPT-3.5-turbo	26	15	57.7%	77%	24	20	83.3%	0
	Claude-3.5-Sonnet	26	9	34.6%	66%	15	13	86.7%	0
optipng	GPT-4o	86	75	87.2%	15%	162	108	66.7%	0
	GPT-4o-mini	86	69	80.2%	18%	145	105	72.4%	1
	GPT-3.5-turbo	86	59	68.6%	28%	121	79	65.3%	0
	Claude-3.5-Sonnet	86	35	40.7%	61%	69	54	78.3%	1
libbmp	GPT-4o	16	16	100.0%	4%	38	35	92.1%	0
	GPT-4o-mini	16	14	87.5%	10%	42	36	85.7%	0
	GPT-3.5-turbo	16	13	81.3%	12%	36	34	94.4%	0
	Claude-3.5-Sonnet	16	13	81.3%	5%	38	35	92.1%	0
u8c	GPT-4o	19	17	89.5%	20%	23	14	60.9%	7
	GPT-4o-mini	19	16	84.2%	31%	22	14	63.6%	4
	GPT-3.5-turbo	19	12	63.2%	23%	13	12	92.3%	0
	Claude-3.5-Sonnet	19	13	68.4%	57%	17	13	76.5%	3

Unsafe Rust. Table I also presents the percentage of unsafe code across all five codebases. The average percentage of unsafe code across all codebases is 13.05%, 20.88%, 31.03%, and 59.10%, for the GPT-4o, GPT-4o-mini, GPT-3.5-turbo, and Claude-3.5-Sonnet models, respectively. Similar to the compilation success results, GPT-4o generates transpiled Rust code with the least number of unsafe program statements. The reduced percentage of Unsafe Rust code for the GPT-4o model derives from its superior *idiomatic* type mapping capabilities.

Impact of LLM model. Our results show that some LLM models are more effective in deriving idiomatic Rust types from C type usages. For example, the `char*` type variables can either be used as a null-terminated string, or as an array of `char` type. Whether or not the variable should be translated to a Rust `string` type or a `Vec` type depends on how it is used in the application. Consider the simplified code snippet from the *libcsv* codebase shown in Figure 5. The `struct csv_parser` contains a field `unsigned char* entry_buf`. The *libcsv* codebase uses this field as a char array. This is evidenced by the presence of functions such as `csv_increase_buffer` which increases the buffer size dynamically by using the `realloc` function, without considering any null-termination logic. The GPT-4o model can correctly analyze the uses of this struct field, provided by the *Struct Field Usage Analyzer* described in §III-A, and transpile it to use the `Vec<u8>` type. However, the other models simply translate this type as a raw pointer of type `*mut u8`.

Similarly, the *urlparser* codebase stores the URL string in a `char*` variable. The GPT-4o model can correctly determine that the type of the `url` argument should be `mut String`. On the other hand, the GPT-3.5-turbo and Claude-3.5-Sonnet LLM models transpile this argument as a raw pointer of type `*const c_char` In fact, across all five codebases, GPT-4o model generates translated Rust variables of the idiomatic `String` type for a total of nine times, whereas the transpilations generated by the GPT-3.5-turbo and Claude-3.5-Sonnet models generate only one and zero variables of type `String` respectively, using raw pointers for the remaining variables.

A) char* used as a buffer, transpiled as Vec<u8>

```
struct csv_parser {
  unsigned char * entry_buf;
  size_t entry_pos;
  size_t entry_size;
  size_t blk_size;
  void *(*realloc_func)(void *, size_t);
};

int csv_increase_buffer(struct csv_parser *p) {
  size_t to_add = p->blk_size;
  void *vp;
  while ((vp = p->realloc_func(...)) == ((void*)0)) {
    to_add /= 2;
    if (!to_add) {
      p->status = 2;
      return -1;
    }
  }
  p->entry_buf = vp;
  p->entry_size += to_add;
  return 0;
}
```

B) char* used as a string, transpiled as String

```
url_data_t* url_parse (const char* url) {
  url_data_t *data = (url_data_t *) calloc(...);
  if (!data) return NULL;
  char* p = strdup(url);
  // ... more code
}
```

Fig. 5. Type usage based transpilation. The GPT-4o model can correctly identify that the `Vec<u8>` type is more appropriate in the first case, and the `String` type is correct for the second case.

B. Semantic Similarity Score

We symbolized all arguments for each C and Rust function and symbolically executed them, with a timeout of 180 minutes. Table I shows the S^3 Score for the C and Rust return values. Across all return values for all applications, the LLM

TABLE II
RUST INSTRUCTION COVERAGE (FOR COMPILABLE CODE)

Codebase	Original C code	GPT-4o	GPT-4o-mini	GPT-3.5-turbo	Claude-3.5-Sonnet
libcsv	94.59%	90.63%	88.89%	93.96%	89.92%
urlparser	82.01%	72.81%	76.94%	72.21%	94.02%
optipng	79.24%	78.08%	79.19%	76.51%	80.59%
libbmp	83.59%	79.21%	88.08%	91.77%	75.07%
u8c	95.04%	89.46%	90.19%	94.62%	95.91%

TABLE III
SYMBOLIC EXECUTION TIME USING 8 CORES (FOR COMPILABLE CODE).

Codebase	GPT-4o	GPT-4o-mini	GPT-3.5-turbo	Claude-3.5-Sonnet
libcsv	5h2m	5h2m	5h2m	5h3m
urlparser	5h4m	5h4m	5h1m	5h2m
optipng	15h5m	11h15m	7h20m	7h42m
libbmp	6h14m	3h46m	1h33m	2h0m
u8c	11h56m	5h4m	5h39m	6h55m

models—GPT-4o, GPT-4o-mini, GPT-3.5-turbo, and Claude-3.5-Sonnet, produced semantically equivalent Rust transpilations for 72%, 74%, 71%, and 80% of all cases. As we discussed in VI-A, although Claude-3.5-Sonnet has the highest semantic similarity score, it can only compile 48.3% functions successfully, while GPT-4o can compile 89.8% functions. Considering the number of the functions that have been successfully compiled, GPT-4o provides the highest number of semantically equivalent transpilations.

Table II shows the symbolic execution instruction coverage. The instruction coverage for the C codebases ranges from 79.24% to 95.04%, while that of the Rust code ranges from 72.21% to 95.91%. This indicates a significant percentage of code was tested for both C program and Rust program. Table III shows the total time taken to execute all functions in each of the codebases symbolically using 8 cores. The Rust code symbolic execution for the GPT-4o model took the longest time because GPT-4o generated the highest number of compilable functions that could be symbolically executed.

C. Selected Bugs

Table IV shows the semantic bugs found by RustAssure. We exclude trivial bugs, such as functions generated with empty bodies, and focus only on the semantic bugs. In this section, we will discuss two such semantic bugs.

Incorrect pointer increment. Figure 6 shows the simplified snippet of the `scan_option` function from the *libcsv* codebase. This function parses configuration options of the form `-option=value`. The option name can be preceded by an arbitrary number of "-" characters. In the C code, the *for* loop in lines 11-23 first scans the option name (lines 12-15). After the execution of line 15, `ptr` points to the *next* character to be consumed. Then, in line 17, if it encounters a = character indicating the separation between the option name and the value, the C code increments the `ptr` pointer to consume and discard the `'='` symbol, and then stores the value to the `opt_arg_ptr` pointer. However, in the Rust code generated by GPT-4o-mini and Claude-3.5-Sonnet, the next character is consumed at the *end* of the iteration. Specifically, line 15 is missing in the Rust code, which results in the = symbol being appended to the option name, instead of being discarded. Therefore, for an input in the form `option=value`, Rust will return `option=` as the option name, indicating a bug. Symbolically analyzing the original C function and the Rust function shows that the minimum symbolic return value is `ptr = ptr + 3` for the C function and `ptr = ptr + 2`. This shows that there is a bug in the Rust function and identifies that a pointer increment operation is missing.

Incorrect conditional check. A bug in the Rust code generated by the GPT-4o, for the *libcsv* codebase, is shown in Figure 7. The original C code sets the `blk_size` field in the `csv_parser`, as long as the pointer `p` is not `NULL`. The LLM-generated Rust code sets the `blk_size` field only if the previous value of that field was not `0`. Symbolic execution captures this semantic divergence because of the additional symbolic constraint `p.blk_size != 0`.

D. Precision and Recall

Table V shows the precision and recall analysis for the S^3 score. Since the recall computation requires manual verification of the LLM-generated Rust code, we report these statistics only for *libcsv* and *u8c*.

Precision. The overall precision across all LLM models for *libcsv* and *u8c* is 85.7% and 88.2%. Besides the semantic bugs we report in Table I, the true positive category also includes LLM-generated functions that are trivially incorrect, such as empty functions. The false positives arise due to optimizations KLEE performs when symbolically modeling arrays and aligning these differences is primarily an engineering effort.

Recall. To compute the recall of the S^3 score, we first determine if our system has any false negatives with respect to the test cases contained in the original C codebase. We manually rewrote the existing test cases for *libcsv* and *u8c*, and used them to test the LLM-transpiled Rust code. This includes 39 tests for *libcsv*, and 5 tests consisting of 13 test inputs for *u8c*. Our results showed that the existing test cases could not find any new bugs that RustAssure missed. Only the bug in the `u8next_FAST` code generated by GPT-4o-mini, that affects *all inputs* can be detected by the original end-to-end test cases.

We also manually reviewed the generated Rust code for these two codebases and did not find any bugs missed by our system. Finally, as we discuss in §VI-F, we also use FLUORINE[4] to test the transplilation results. Our experiments showed that, due to various crashes, FLUORINE does not find any bugs that cannot be revealed by RustAssure.

E. End-to-end Comparison against CROWN

CROWN [3] is a syntax-driven transpilation toolchain that performs ownership analysis to promote raw pointers to safe owned pointers. Table VI compares the translation results of RustAssure, using the best performing LLM model—GPT-4o,

Original C code

```c
int scan_option(
    const char *str,
    char opt_buf[],
    size_t opt_buf_size,
    const char **opt_arg_ptr)
{
   // ... skip any leading "-" in str
   const char *ptr;
   opt_len = 0;
   ptr = str + 1;
   for ( ; ; ) {
     if (opt_len < opt_buf_size)
        opt_buf[opt_len] = (char)tolower(*ptr);
       ++opt_len;
       ++ptr; // Missing in Rust code
       // Code to handle end of string
       if (*ptr == '=') {
        ++ptr;
        *opt_arg_ptr = ptr;
        break;
       }
    }
    // ... more code
}
```

LLM-transpiled Rust Code

```rust
fn scan_option(
   str: *const c_char,
   opt_buf: *mut c_char,
   opt_buf_size: usize,
   opt_arg_ptr: *mut *const c_char,
) -> i32 {
   // ... some code ...
   let str = unsafe { CStr::from_ptr(str) }.to_str().unwrap();
   let mut opt_len = 0;
   let mut ptr = str[1..].trim_start_matches('-');
   for (i, c) in ptr.char_indices() {
       if opt_len < opt_buf_size {
           opt_buf[opt_len] = c.to_ascii_lowercase() as u8;
       }
       opt_len += 1;
       // Code to handle end of string
       if c == '=' {
          unsafe {
            *opt_arg_ptr = ptr[i + 1..].as_ptr() as *const c_char;
          }
          break;
       }
    // ... more code
}
```

Fig. 6. Bug detected by RustAssure. Rust code generated by Claude-3.5-Sonnet misses a pointer increment that results in an incorrect output.

Original C code

```c
struct csv_parser {
  size_t blk_size;
  //... more fields
};
void
csv_set_blk_size(struct csv_parser *p, size_t size){
  if (p) p->blk_size = size;
}
```

LLM-transpiled Rust Code

```rust
struct CsvParser<'a> {
   blk_size: usize,
   //... more fields
}
fn csv_set_blk_size(p: &mut CsvParser, size:usize) {
    if p.blk_size != 0 { // incorrect check
       p.blk_size = size;
    }
}
```

Fig. 7. Bug case detected by RustAssure. Rust code generated by GPT-4o has a wrong conditional check that results in a failed assignment.

to those of CROWN. The overall compilation success rate for RustAssure and CROWN is 89.8% and 97.3%, respectively. Since CROWN is based on syntactic translation, it has a higher compilation success rate than RustAssure, and does not have any runtime functional divergences, but it does not generate idiomatic Rust code. Across all codebases, RustAssure and CROWN have 35 and 271 raw pointer declarations, and 72 and 258 raw pointer uses, respectively, showing that RustAssure provides more idiomatic Rust transpilations. Note that `opti_png` declares `18` raw pointers of which `0` are used within the codebase—the raw pointers are strictly passed to dependent libraries. Even for the remaining models, the average raw pointer declaration and use, normalized to the total number of Rust pointers, is less than that of CROWN.

F. FLUORINE Comparison

FLUORINE [4] is an LLM-based transpiler toolchain that uses differential fuzzing to test the transpilations. To facilitate fuzzing across C and Rust, the toolchain also requires a JSON data-exchange file, specifying critical information, such as the number of function arguments and their types.

End-to-end Comparison. FLUORINE first extracts the individual functions and their dependencies from the C codebase. However, the available artifact does not provide the source code for this step. Instead, the artifact provides the extracted functions, along with their dependencies, and the JSON-mappings for 31 (out of 42) functions for the *libopenaptx* library, along with their Rust transpilations generated using the GPT-4o model. We transpiled *libopenaptx* using RustAssure, and performed the comparison against the results provided in FLUORINE's artifact. Table VII shows the results of this comparison. Out of the 31 compilable Rust functions generated by FLUORINE, only 10 pass FLUORINE's differential fuzzing stage, due to crashes in the fuzzer, while RustAssure can establish the semantic equivalence of 25 out of the 31 functions handled by FLUORINE, and 33 out of all 38 compilable functions generated by RustAssure. Because we manually verified the RustAssure transpilations, we did not fuzz them. This is indicated by the `N/A` value in Table VII.

Effectiveness on Bugs Found by RustAssure. We manually crafted the JSON data-exchange file for the 20 buggy functions generated by different LLM models, listed in Table IV, and found that only 1 of them can be detected by FLUORINE, with the remaining functions failing to successfully execute to completion.

Across the 20 tested buggy functions, 14 failures are caused by the fuzzer not supporting the argument data type. For

TABLE IV
DETAILS OF BUGS DETECTED BY RUSTASSURE. NOTE THAT CLAUDE-3.5 INDICATES THE CLAUDE-3.5-SONNET LLM MODEL

Codebase	**LLM Model**	**Function Name**	**Bug Description**
libcsv	GPT-4o, GPT-3.5-turbo, GPT-4o-mini	csv_write2	The C version writes the source string into a `char*` pointer, and the Rust version writes to a string slice `&str`. If the slice is empty, indicating zero capacity, the Rust code returns `0`, whereas the C version returns the length of the input string if the output buffer is `NULL`.
libcsv	GPT-4o	csv_set_blk_size	The Rust code has an incorrect conditional check. We discuss this in §VI-C.
libcsv	GPT-4o	csv_set_delim	This function sets the `delim_char` field into the `csv_parser` struct. The Rust code has an incorrect check where it skips assigning the input value to the field if the current `delim_char` field is "`\0`". The original C code simply overwrites the existing character even if it is "`\0`".
libcsv	GPT-3.5-turbo	csv_free	This function frees the `char *` field `entry_buf` inside `csv_parser` and sets `entry_size` to 0. Rust code uses a raw pointer for `entry_buf` and contains an incorrect check where it sets `entry_size` to `0` only if `entry_buf` is not `NULL`, whereas the original C code unconditionally sets the `entry_size` variable to `0`.
libcsv	GPT-4o, Claude-3.5	csv_fini	This function finalizes the CSV parsing by flushing the last field. The bug arises because in C `switch` statements *fall through*, but in Rust the `match` statement does not. The LLM converts a `switch` statement to a `match` statement, without accounting for the fall-through behavior.
urlparser	GPT-4o	scan_decimal_number	This function scans a string and returns a pointer to the first non-digit character. The Rust code returns the digit part instead of the non-digit part.
optipng	Claude-3.5, GPT-4o-mini	scan_option	The Rust code has a pointer increment bug. We discuss it in detail in §VI-C.
u8c	GPT-4o	u8encode_	This function converts a single Unicode code-point (ch) into its UTF-8 byte sequence. The generated Rust code does not handle invalid Unicode sequences at all and ultimately tries to prematurely add the terminator character at offset `-1` of the output string slice, leading to a panic.
u8c	GPT-4o, GPT-4o-mini	u8next_	The generated Rust code returns an incorrect value when the input contains invalid Unicode sequences. We discuss this bug in detail in §II.
u8c	GPT-4o-mini	u8next_FAST	The original C function traverses the raw UTF-8 byte stream one byte at a time and applies bitmasks to individual *bytes* to reconstruct the correct Unicode code-point. The generated Rust code applies the bitmasks on the individual codepoints in the *already-decoded* Unicode sequence, instead of individual bytes, generating garbage values.
u8c	Claude-3.5	u8next_FAST	When the input string is empty, the C code returns `0` in the output parameter `ch*`, but Rust does not update it at all, leaving any garbage value in it intact.
u8c	GPT-4o, Claude-3.5	u8strncpy	This function is a `UTF-8` version of strncpy that copies at most n bytes from input to output and then trims any incomplete multibyte sequence at the end so the output will always end with a valid UTF- 8 character. The Rust code sets the last byte of the buffer to `0` incorrectly, which will overwrite the meaningful byte of the input.
u8c	GPT-4o, GPT-4o-mini	u8strncat	This function appends characters from the source to the end of the target input string and trims any incomplete multibyte `UTF-8` sequence. The Rust code performs the copy operation without trimming the incomplete `UTF-8` sequence.

TABLE V
PRECISION AND RECALL ANALYSIS.

Codebase	**Model**	**TP**	**FP**	**FN**	**Precision**	**Recall**
libcsv	GPT-4o	5	2	0	71%	100%
	GPT-4o-mini	7	1	0	88%	100%
	GPT-3.5-turbo	19	0	0	100%	100%
	Claude-3.5-Sonnet	5	3	0	63%	100%
u8c	GPT-4o	7	1	0	88%	100%
	GPT-4o-mini	4	1	0	80%	100%
	GPT-3.5-turbo	0	0	0	-	100%
	Claude-3.5-Sonnet	4	0	0	100%	100%

TABLE VI
COMPARISON OF RUSTASSURE (USING GPT-4O) VS CROWN.

Codebase	**Perc. compiled**		**Raw Pointer Decls.**		**Raw Pointer Uses**	
	Rust Assure	**CROWN**	**Rust Assure**	**CROWN**	**Rust Assure**	**CROWN**
libcsv	93%	100%	8	23	2	13
urlparser	88%	88%	5	69	1	90
optipng	87%	98%	18	135	0	66
libbmp	100%	100%	0	18	0	37
u8c	89%	100%	4	26	69	52

TABLE VII
COMPARISON OF RUSTASSURE VS FLUORINE (LIBOPENAPTX LIBRARY).

Method	**Total Function**	**Compilation Success**	**Raw Pointer**	**Passes Fuzzing**	**Passes Sym. Testing**
FLUORINE	31	31	0	10	25
RustAssure	42	38	1	N/A	33

example, for `Option` type function arguments, FLUORINE's fuzzer supports only owned values wrapped in `Option<T>` types, and not borrowed references (`Option<&T>`). Moreover, the JSON-based argument mapper assumes that for `struct` arguments, the transpiled Rust `struct` definitions will have the same field names as their C counterparts, while RustAssure does not have this restriction. This demonstrates how FLUORINE requires case-by-case handling of all types. Ultimately, RustAssure can automatically handle a significantly a larger set of types because the symbolic execution occurs at the LLVM IR level, which has a much smaller set of supported types. For example, owned values and references are both mapped to the LLVM IR `Pointer` type. Moreover, the automatic handling of the language level differences, described in §IV, facilitates the comparison of across different types.

TABLE VIII
COMPARISON OF C-TO-RUST TRANSLATION SYSTEMS, APPROACHES, STRATEGIES, AND TESTING METHODS.

System Name	Approach	Detailed Strategy	Testing Method
LAERTES[1]	Syntactic Translation	Uses compiler feedback to progressively rewrite unsafe pointers to owned pointers and references.	Comparison of dynamic execution traces
CROWN[3]	Syntactic Translation	Uses novel *ownership analysis* to derive Rust ownerships and rewrite unsafe pointers to safe Rust.	Existing concrete test cases
Tymcrat[30]	LLM-based	Generates multiple Rust function signature candidates for each C function and uses the translated signatures of callee functions to guide idiomatic translation of callers.	Manually checked
C2SAFERRUST[31]	LLM-based	Analyzes file-level dependencies, providing them to the LLM, in an optimal order.	Existing concrete test cases
RustMap[32]	LLM-based	Decomposes large C projects into smaller units to simplify transpilation.	Existing concrete test cases
Code Segmentation[33]	LLM-based	Performs code segmentation by organizing C functions in topological order.	Existing concrete test cases
FLUORINE[4]	LLM-based	Provides translation feedback to LLM, using cross-language differential fuzzer.	Differential fuzzing
SACTOR[34]	LLM-based	Uses C2Rust[35] and LLMs to first translate C code to Rust and then refines pointers using ownership information from CROWN[3].	Existing concrete test cases
SYZYGY[36]	LLM-based	Uses dynamic analysis to augment LLM-based transpilation.	Existing concrete test cases
LAC2R[37]	LLM-based	Uses LLMs to transpile C code to Rust and also generate new test cases.	LLM-generated test cases
VERT[28]	LLM-based	Generates "oracle" Rust transpilation using rWASM.	Model checking against rWASM-based oracle.
PR2[29]	LLM-based	Lifts raw pointers in C2rust [35] output to idiomatic Rust types using LLMs.	Existing concrete test cases.
RustAssure	**LLM-based**	**Uses preprocessing and replays already transpiled structs to improve compilation success.**	**Differential symbolic testing.**

VII. DISCUSSION

Use Cases. RustAssure's testing approach is not restricted to LLM-generated code and can be generalized to codebases transpiled using any approach. As more teams adopt Rust for its safety guarantees, having an automated checker to validate functional parity between legacy and rewritten components reduces both migration effort and operational risk.

Enhancements. RustAssure can be improved with a feedback loop from the Semantic Checker that returns divergences to the LLM for targeted re-transpilation. Integrating lightweight static analysis to prioritize symbolic paths, improving support for common third-party libraries, and caching partial executions can improve the symbolic execution coverage. Additionally, reducing false positives by expanding the set of language-level normalization patterns can improve RustAssure's equivalence checking. Compositional symbolic execution techniques [38][39] can improve its scalability.

VIII. RELATED WORK

C-to-Rust Transpilation Techniques. Table VIII provides a conceptual comparison with other key approaches. Various analysis-based techniques [1], [2], [13], [40] transpile C to Rust using the intermediate unsafe Rust output of the C2Rust tool[35]. These approaches guarantee the correctness of the translation but can produce unidiomatic Rust code[4]. Concrat [41] focuses on replacing C Lock API with Rust Lock API for concurrent programs. Hong and Ryu [42] focus on automatically replacing C pointer-based return arguments with Rust tuples. GenC2Rust [43] translates C code into generic Rust code through static analysis. Fromherz and Protzenko [44] focus on transpiling formally-verified C code to Rust. Various approaches combine program analysis techniques, such as, ownership analysis [34], type analysis [30], code-slicing [32], code-segmentation [27], control and data flow analysis [31], with LLM-based transpilation. PR^2 [29] uses LLMs to perform peephole optimization to rewrite raw pointers with idiomatic Rust types. Oxidizer [45] translates Go to Rust by combining translation rules with LLMs. The benchmark suites [46], [47] evaluate the effectiveness of C-to-Rust transpilers.

Testing and LLMs. Current approaches for testing LLM-generated Rust code use existing test cases [32] and differential fuzzing [4]. Vert[28] uses property-based testing and bounded model-checking by compiling both the original C code and the LLM-generated Rust code to WASM [48]. Unlike RustAssure, however, Vert does not handle any language level differences. HoHyun et al. [37] presents an LLM agent that performs a fuzzing-based equivalence test to iteratively refine Rust code, using LLMs to generate fuzzing inputs. Syzygy [36] combines LLM-driven code and test generation with dynamic analysis. Various works [49], [50], [51] have used LLMs to *generate* new unit tests. These approaches can complement RustAssure.

Symbolic execution. Differential Symbolic Execution [38] is a technique for finding diverging behavior between different versions of the same program. RustAssure extends this concept to a cross-language setting. Zhang et al. [52] created an approach to automatically verify Rust programs with KLEE.

IX. CONCLUSION

We presented RustAssure—an LLM-based C-to-Rust transpiler, that performs differential symbolic testing on the transpiled code. Across five C codebases, RustAssure generated compilable Rust functions for 89.8% of all C functions, of which 72% produced equivalent symbolic return values.

References

[1] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, “Translating c to safer rust,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.

[2] M. Emre, P. Boyland, A. Parekh, R. Schroeder, K. Dewey, and B. Hardekopf, “Aliasing limits on translating c to safe rust,” *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 551–579, 2023.

[3] H. Zhang, C. David, Y. Yu, and M. Wang, “Ownership guided c to rust translation,” in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 459–482.

[4] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, “Towards translating real-world code with llms: A study of translating to rust,” *arXiv preprint arXiv:2405.11514*, 2024.

[5] “Inside GitHub: Working with the LLMs behind GitHub Copilot,” 2024. [Online]. Available: https://github.blog/ai-and-ml/github-copilot/inside-github-working-with-the-llms-behind-github-copilot/

[6] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models,” in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[7] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration code generation via chatgpt,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[8] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, “Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2332–2354, 2024.

[9] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, “Lost in translation: A study of bugs introduced by large language models while translating code,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[10] “Openai models,” 2024. [Online]. Available: https://platform.openai.com/docs/models

[11] “Openai platform: Gpt 4.o mini,” 2024. [Online]. Available: https://platform.openai.com/docs/models#gpt-4o-mini

[12] “Meet claude anthropic,” 2024. [Online]. Available: https://www.anthropic.com/claude

[13] C. D. Hanliang Zhang, Y. Yu, and M. Wang, “Ownership guided c to rust translation.”

[14] “OpenAI Models,” https://platform.openai.com/docs/models, 2024.

[15] “What is LLM Temperature?” https://www.ibm.com/think/topics/llm-temperature, 2025.

[16] Mistral AI, “Mistral 7b,” https://mistral.ai/news/announcing-mistral-7b/.

[17] P. Deligiannis, A. Lal, N. Mehrotra, and A. Rastogi, “Fixing rust compilation errors using llms,” *arXiv preprint arXiv:2308.05177*, 2023.

[18] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.” in *OSDI*, vol. 8, 2008, pp. 209–224.

[19] “The reference manual for the KQuery language,” https://klee-se.org/docs/kquery/, 2025.

[20] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, “An exact graph edit distance algorithm for solving pattern recognition problems,” in *4th International Conference on Pattern Recognition Applications and Methods 2015*, 2015.

[21] “Clang: a C language family frontend for LLVM,” https://clang.llvm.org/, 2024.

[22] “AST Matcher Reference,” https://clang.llvm.org/docs/LibASTMatchersReference.html, 2025.

[23] “ANTLR (ANother Tool for Language Recognition) ,” https://github.com/antlr/antlr4, 2025.

[24] “GraphViz) ,” https://graphviz.org/, 2025.

[25] “NetworkX - Network Analysis in Python ,” https://networkx.org/, 2025.

[26] A. Hagberg, P. J. Swart, and D. A. Schult, “Exploring network structure, dynamics, and function using networkx,” Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[27] M. Shiraishi and T. Shinagawa, “Context-aware code segmentation for c-to-rust translation using large language models,” *arXiv preprint arXiv:2409.10506*, 2024.

[28] A. Z. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, “Vert: Verified equivalent rust transpilation with large language models as few-shot learners,” *arXiv preprint arXiv:2404.18852*, 2024.

[29] Y. Gao, C. Wang, P. Huang, X. Liu, M. Zheng, and X. Zhang, “Pr2: Peephole raw pointer rewriting with llms for translating c to safer rust,” *arXiv preprint arXiv:2505.04852*, 2025.

[30] J. Hong and S. Ryu, “Type-migrating c-to-rust translation using a large language model,” *Empirical Software Engineering*, vol. 30, no. 1, p. 3, 2025.

[31] V. Nitin, R. Krishna, L. L. d. Valle, and B. Ray, “C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques,” *arXiv preprint arXiv:2501.14257*, 2025.

[32] X. Cai, J. Liu, X. Huang, Y. Yu, H. Wu, C. Li, B. Wang, I. N. B. Yusuf, and L. Jiang, “Rustmap: Towards project-scale c-to-rust migration via program analysis and llm,” *arXiv preprint arXiv:2503.17741*, 2025.

[33] M. Shiraishi and T. Shinagawa, “Context-aware code segmentation for c-to-rust translation using large language models,” 2024. [Online]. Available: https://arxiv.org/abs/2409.10506

[34] T. Zhou, H. Lin, S. Jha, M. Christodorescu, K. Levchenko, and V. Chandrasekaran, “Llm-driven multi-step translation from c to rust using static analysis,” *arXiv preprint arXiv:2503.12511*, 2025.

[35] “C2rust,” 2024. [Online]. Available: https://github.com/immunant/c2rust

[36] M. Shetty, N. Jain, A. Godbole, S. A. Seshia, and K. Sen, “Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis,” 2024. [Online]. Available: https://arxiv.org/abs/2412.14234

[37] H. Sim, H. Cho, Y. Go, Z. Fu, A. Shokri, and B. Ravindran, “Large language model-powered agent for c to rust code translation,” 2025. [Online]. Available: https://arxiv.org/abs/2505.15858

[38] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, “Differential symbolic execution,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 226–237.

[39] P. Godefroid, “Compositional dynamic test generation,” in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007, pp. 47–54.

[40] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, “In rust we trust: a transpiler from unsafe c to safer rust,” in *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, 2022, pp. 354–355.

[41] J. Hong and S. Ryu, “Concrat: An automatic c-to-rust lock api translator for concurrent programs,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 716–728.

[42] ——, “Don’t write, but return: Replacing output parameters with algebraic data types in c-to-rust translation,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 716–740, 2024.

[43] X. Wu and B. Demsky, “ GenC2Rust: Towards Generating Generic Rust Code from C ,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 664–664. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00127

[44] A. Fromherz and J. Protzenko, “Compiling c to safe rust, formalized,” 2024. [Online]. Available: https://arxiv.org/abs/2412.15042

[45] H. Zhang, C. David, M. Wang, B. Paulsen, and D. Kroening, “Scalable, validated code translation of entire projects using large language models,” 2024. [Online]. Available: https://arxiv.org/abs/2412.08035

[46] G. Ou, M. Liu, Y. Chen, X. Peng, and Z. Zheng, “Repository-level code translation benchmark targeting rust,” 2025. [Online]. Available: https://arxiv.org/abs/2411.13990

[47] A. Khatry, R. Zhang, J. Pan, Z. Wang, Q. Chen, G. Durrett, and I. Dillig, “Crust-bench: A comprehensive benchmark for c-to-safe-rust transpilation,” 2025. [Online]. Available: https://arxiv.org/abs/2504.15254

[48] WebAssembly Community Group, “WebAssembly,” https://webassembly.org/, 2025, official website, accessed 26 May 2025.

[49] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[50] A. Nunez, N. T. Islam, S. K. Jha, and P. Najafirad, “Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing,” *arXiv preprint arXiv:2409.10737*, 2024.

[51] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, “Chatunitest: A framework for llm-based test generation,” in *Companion Proceedings of*

the 32nd ACM International Conference on the Foundations of Software Engineering, 2024, pp. 572–576.

[52] Y. Zhang, P. Li, Y. Ding, L. Wang, D. Williams, and N. Meng, "Broadly enabling klee to effortlessly find unrecoverable errors in rust," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 441–451. [Online]. Available: https://doi.org/10.1145/3639477.3639714